Education and Research Department

COBOL

May 2003

Document No.	Authorized By	Ver. Revision	Signature / Date
ER/CORP/CRS/LA01/002	*Ravindra M.P.*	Ver. 3.10	

COPYRIGHT NOTICE

Education and Research Department
Infosys Technologies Limited
Electronic City
Hosur Road
Bangalore - 561 229, India.

Tel: 91 80 852 0261-270
Fax: 91 80 852 0362
www.infy.com
mailto:EnR@infy.com

Document Revision History

Ver. Revision	Date	Author(s)	Reviewer(s)	Description
1.00	Mar-1999	Chandrasekhar P	Anonymous	Baseline Version
2.00	24-Aug-2001	Narendra Raje Urs	Chandrasekhar P and Baljeet Kaur	Re-structured the complete material to comply with the course delivery pattern. More illustrations have been included to explain the concepts discussed in all the chapters. The Chapter on Control Break Processing has been removed. New programs have been added in all the chapters to explain the programming techniques clearly. An appendix has been added to include the JCLs required for executing the sample programs given in the material in the MVS environment.
3.00	10-Feb-2003	Partha Sarathi Brahmachari	Subrahmanya S. V.	Following points have been introduced. - Difference among OS/VS COBOL, VS COBOL II, COBOL/370. - Control Break and Report Writer. - JCL & COBOL interactions. - Processing of Alternate index for VSAM file. - Handling I/O error using DECLARATIVES. - PIC clauses in detail. - Debugging tool. - Static Call and Dynamic call - Global and External data elements. - Performances issues and coding practice. - Data Validation - Compiler Option & Compiler listing. - Index has been included.

3.10	13-Aug-2003	Rakesh Agarwal	Nandakumar N.	The changes have been made in chapter: - Data Division (Fixed insertion); - File Handling (Procedure division for Relative files - WRITE/REWRITE & DELETE Statements); - File Handling (Sample Program)

High Level Design

1 COBOL

1.1 Course Description

Course number	LA01			**Course Name**		Cobol			
Author(s)	Narendra Raje Urs and Chandrasekhar P								
Pre-requisites for attending course Knowledge of Programming Fundamentals will be an advantage									
Stream (Eg. Project Mgmnt/Customer Interface/ Consulting etc.)	Customer Interface and Consultin-g and Project man-agement	**Target Role**[1] (Eg. SE/PA/PM etc.)	SE and above	**Competencies**[2] **(TK, P, T, D, A, PS)**	TK, P	**Type** (E-Essential, D-Desirable)	**E**	**Category**[3] (Eg.PL/OS/DB/NM/SE/TO)	PL
Estimated course duration	09 days (22 hrs of Lecture + 29 hrs of lab + 30 hrs of project)								

[1] PM Stream (Choose one)-SE: Software Engineer, PA: Programmer Analyst, PM: Project Manager

[2] Competencies (Choose appropriate ones)-TK:Technical Knowledge, P:Programming, T:Testing, D:Design, A:Analysis, PS:Pre-sales.

[3] Category (Choose one)-PL: Programming or Programming Language, OS: Operating Systems, DB: Database, NM: Networks and Middleware, SE: Software Engg, TO: Technical Overview

1.2 Course Objectives

Sl#	Objective	Demonstrable knowledge/skills
1.	To introduce the participants to COBOL programming	Knowledge to design and develop simple, structured COBOL programs must be gained.
2.	To enable the participants to learn the use of files with COBOL programs.	Ability to work with Sequential and Index sequential files.
3.	To introduce the participants to Tables.	Ability to use the Tables as required in the programs.
4.	To illustrate the special features of COBOL such as SEARCH, SORT, MERGE, etc.,	Knowledge of language to use internal features of COBOL.
5.	To create the ability to understand the given COBOL programs and make necessary changes.	Ability to understand the given program and make changes if necessary and test them.

1.3 Course Design

Sl#	Unit name	Unit objectives and keywords	Lecture Duration (hrs.)
1.	Introduction	To introduce the basics of COBOL programming. To demonstrate development of Hello World! Program	2hours
2.	DATA DIVISION Re-visited	To discuss the basic data types of literals, description of data-names and arithmetic verbs.	3 hours
3.	PROCEDURE DIVISION Re-visited	To explain the need for control structures. To introduce different structures such as GO TO, IF . . . THEN . . . ELSE, PERFORM and EVALUATE.	3 hours
4	File Processing	To introduce the participants to files. To illustrate different file ORGANIZATIONs and ACCESS MODEs. To demonstrate the use of SEQUENTIAL files and INDEX SEQUENTIAL files. Handling I/O error using DECLARATIVES.	4 hours
5.	Tables	To introduce Tables. To demonstrate the use of Tables. To illustrate the use of SEARCH and SEARCH ALL	3 hours
6.	Miscellaneous utilities	To demonstrate the internal language features such as SORT and MERGE. To make participants clearly understand the underlying differences between the COPY and CALL verbs and demonstrate their use. To introduce the participants to the string handling features of COBOL.	3 hours
7.	Control Break and Report Writer	Control Break Processing. Report Writer	1.5 hours
8.	Advanced Topics	JCL and COBOL Interactions. Processing of Alternate Index for VSAM file. Compiler Option. Compiler Listing. Static and Dynamic Call. Global and External Data Elements Debugging Tools and Techniques	1.5 hours
9.	Good Coding practice	Performance Issue and Good coding practice Data Validation Code walkthrough	1 hours

1.3.1 Sources

1. Stern, Nancy, Stern, Robert A., Structured COBOL Programming, Eight Edition, John Wiley, 2000.
2. Philippakis, A. S., Kazmier, L.J., Structured COBOL, McGraw Hill.
3. Philippakis, A. S., Kazmier, L.J., Advanced COBOL, McGraw Hill.
4. Roy, M.K., Dastidar, D.G., COBOL Programming : Including MS-COBOL and COBOL-85, Second Edition, Tata McGraw Hill.
5. Newcomer, L.R., Programming with Structured COBOL, Schaum's Outline Series, McGraw Hill.

CONTENTS

1 Introduction

OVERVIEW

1. History and Evolution of COBOL.
2. Different Versions of COBOL
3. Nature of COBOL.
4. Structure of a COBOL Program (Bottom-up Approach).
5. Overview of the four DIVISIONs.
6. COBOL Coding Format.
7. Rules for interpreting instruction formats.
8. Terminal input/output statements.
9. Development of "Hello World" program on the Mainframe platform.

1.1. Evolution of COBOL

1.1.1. History of COBOL

During the 1950s, the western parts of the world experienced a tremendous need for a high level programming language suitable for business data processing. To meet this demand the U.S. Dept of Defence convened a conference on the 28th and 29th of May, 1958 to highlight and stress the importance of a high-level business data processing language. The conference led to the formation of a team called CODASYL (Conference On DAta Systems Language). This team had representatives from civil and governmental organisations, academics, computer manufacturers and other software enthusiasts to work towards the goal. As a result, a new language called COBOL (COmmon Business Oriented Language) was developed in September 1959. This was released with some minor modifications in April 1960.

On May 5, 1961, COBOL-61 was published with some more revisions. It is this version which provided the foundation for later versions. In 1965, a re-revised version of COBOL was released with a number of additional features. However, it was only in 1968 that COBOL was approved by ANSI (American National Standards Institute) as a standard language for commercial use. This version of COBOL is known as COBOL-68. The next revised official standard was released in 1974 and is known as COBOL-74. Again, in 1985, a new ANSI version of the language called the COBOL-85 was released incorporating a number of additional features. Today most COBOL compilers adhere to the 1985 standard. The latest release of COBOL is Object Oriented COBOL and is known as OO COBOL or COBOL 9x.

1.1.2. Current Status of COBOL

A typical enterprise application consists of hundreds of COBOL programs running on a mainframe computer such as System 390. In the late 90s, the Fortune 1000 companies spent billions of dollars fixing the Y2K problems in their COBOL programs. As they fixed the date problems, they also upgraded the code so these programs could continue to run indefinitely. As we see it, that speaks volumes. These programs are not going to be replaced by other types of systems at least in the immediate future, for, there's not enough money, time, or programming talent to do that. Besides that, these programs work the way they're supposed to and most users are more than happy with them.

This has two important implications. The first one is that these programs are to be maintained and enhanced regularly. This means that there will be a continuous demand for COBOL programmers with about 70% of the work in the form of maintenance and the other 30% in the form of enhancements.

The second implication is that all new applications should co-exist with this base of applications. This means any new application developed to replace an existing application should be integrated with the existing system. To do that effectively, one has to understand how the programs, files and databases of the existing system work. It also helps if you can read the code in the old system to make sure you get the calculations, lookups and other business logic and rules right. That, of course, means you need to understand COBOL.

1.1.3. Future of COBOL

To help one to deal with the problems of system integration, IBM has developed a master strategy for making COBOL, CICS and DB2 the foundation for the Web and Intranet applications. In this scenario, CICS is the transaction-processing monitor for web applications, DB2 is the database and COBOL provides the business logic that drives these applications. That takes care of all the programming except for the user interfaces on the clients or the terminals and these can be developed with Java, Visual Basic, or other programming languages. This means that all that Java and Visual Basic will do is provide the graphical user interfaces that send data to and receive data from CICS. In other words, the role of these languages will be extremely limited.

It is easy to understand that this state will prevail for quite some time at least for three reasons. First, it's a lot simpler to develop a large application with a single database server than it is to replicate that data over several of smaller servers. Second, a heavy-duty transaction-processing monitor like CICS currently provides processing efficiencies that are superior to those that are possible with client/server technologies. Third, IBM is in the best position to provide workable solutions for system integration because they control the mainframes. Hence even if this master plan changes somewhat during the next few years, DB2 and CICS on a mainframe with COBOL as the self-documenting business language is going to be a tough combination to challenge and beat. Thus COBOL is by far the most dominant language on mainframes and it will continue to be so in the foreseeable future.

In fact, around 15 years back many computer experts thought that COBOL is on the verge on becoming extinct, even today some people believe the same and in my opinion so will some people in the future.

1.2. Different versions of COBOL

The following table captures the History of four important versions of COBOL. They are OS/VS COBOL, VS COBOL II and COBOL/370 and Object Oriented COBOL.

Year	Category	Enhancement
1959	COBOL was originated	
1968	COBOL was revised.	ANSI 68 OS/VS COBOL language level (01).
1968 – 1973	ISAM and VSAM	To incorporate ISAM and VSAM.
1974	COBOL was revised a second time.	OS/VS COBOL language level (02) is COBOL 1974.
1984	VS COBOL II (Release 1.0) was issued by IBM.	The following features of OS/VS COBOL were eliminated: READY TRACE, EXHIBIT, EXAMINE, CURRENT-DATE, TRANSFORM, ON REMARKS, Report Writer, exponentiation, floating-point data, complex OCCURS DEPENDING ON data items, TALLY, and APPLY WRITE-ONLY.
1986	VS COBOL II, Release 2.0	Exponentiation, floating-point data, complex OCCURS DEPENDING ON data items, TALLY, APPLY WRITE-ONLY features of OS/VS COBOL were reinstated by VS COBOL II (Release 2). The 1974 ANSI COBOL standard (except Report Writer, Communications Feature, and level 2 of the debug module) had been supported by VS COBOL II (Release 2). Some of the 1985 ANSI COBOL features (EVALUATE, INITIALIZE, and the in-line PERFORM) had been included on VS COBOL II (Release 2).
1988	VS COBOL II, Release 3.0	Multiple PROCEDURE DIVISIONs and seven levels of subscripts and indexes could be coded. Files and data could be externally shared among many programs. Many other changes to support 1985 ANSI COBOL standard had been incorporated in VS COBOL II (Release 3.0).
1994	COBOL/370 Version 1, Release 1.0	Intrinsic function provides the capability to reference a data item whose value is derived automatically at the time of reference during the execution of the object program.

		The examples are as follows: ANNUITY, CURRENT-DATE, DATE-OF-INTEGER, MAX, MEAN, MIN, PRESENT-VALUE, SQRT, SUM, WHEN-COMPILED, and so on. COBOL 370 supports all the features of VS/COBOL II.
1997-1998	COBOL 97	Object Oriented COBOL

Note: APPENDIX-E describes the important features of the above four different versions of COBOL.

1.3. Nature of COBOL

1.3.1. Business-Oriented Language

As the name itself suggests, COBOL (COmmon Business Oriented Language) is specifically designed for developing business-oriented applications such as Payroll and Inventory Control that typically operate on large volumes of data. It provides excellent file handling features, utilities such as SORT and MERGE, the Report Writer module, which aids report generation, the communication module, which is used for communication across programs, the string handling capabilities for character manipulation, etc., Today COBOL is the most widespread commercial applications language in use. Stern and Stern have made the following estimations.

1. There are around 180 billion lines of COBOL code in use.
2. Around 42.7% of all application programmers are working on COBOL in medium to large-scale industries in the U.S.
3. Revenue from application software developed using COBOL is $176.4 million in 1998.

1.3.2. Standard Language

COBOL is a standard language. This means that COBOL compilers are made available by different third party vendors on almost all types of computers. All these compiler manufacturers adhere to the norms set by ANSI. The implication of this is that the same COBOL program may be compiled and executed on a variety of different machines, such as S/390, VAX-6410, AS-400, or a PC with minimal changes.

1.3.3. Robust Language

COBOL is a highly robust language. This means that there are a number of testing and debugging tools available for COBOL programs on almost all kinds of computer platforms. These tools are developed by different third party vendors based on the ANSI directives.

1.3.4. Structured Programming Language

COBOL is a structured programming language. There is a wealth of logical control structures that makes programs easier to read, debug and modify if changes are required. A typical COBOL program is divided into what are known as modules or paragraphs, where a main module invokes sub-modules as and when required. After the execution of a sub-module the control returns back to the statement following the calling statement in the main module.

Though COBOL provides the GO TO, GO TO . . . DEPENDING ON . . . and EXIT verbs, programmers are advised to limit their use as they are against the idea of a structured programming.

1.3.5. English-like Language

COBOL is an English-like and hence a very user-friendly language. All instructions can be coded using English words rather than complex codes. For instance, to read a file, we use the verb READ. As a result of this, COBOL programs are very easy to read and understand and learning COBOL is one of the easiest things on earth. However, this also makes a COBOL program highly verbose and hence very tedious to code.

1.4. Constituents of a COBOL Program

1.4.1. Characters

The most basic and indivisible unit of the COBOL language is the character. The COBOL character set includes 78 characters that can be classified as letters of the alphabet, digits and special characters. The complete set of characters that form the COBOL character set is shown in the following table.

Character(s)	Meaning
A – Z	Alphabets (Upper case)
a – z	Alphabets (Lower case)
0 – 9	Digits
+	Plus (For addition)
-	Minus (For subtraction) or Hyphen
*	Asterisk (For multiplication)
/	Slant, Stroke or Slash (For division)
=	Equal to sign
$	Dollar (For currency sign)
,	Comma
;	Semicolon
.	Decimal point or Period
"	Quotation mark
(	Left parenthesis
)	Right Parenthesis
>	Greater than
<	Less than
:	Colon
'	Apostrophe

Table 1.1 COBOL Character Set

The individual characters are combined to form character-strings. A character-string is a sequence of contiguous characters that forms a COBOL word which may be of the following three types (1) Reserved word, (2) User-defined word and (3) Literal, also known as a constant.

1.4.2. Reserved words

A reserved word is a character-string with a predefined meaning in a COBOL source program. There are several types of reserved words in COBOL. The following are the most important ones.

Keywords

Keywords are reserved words that are required within a given clause, an entry, or a statement. Within each format, such words appear in uppercase.

Examples: ADD, READ, SEARCH, etc.,

Optional words

Optional words are reserved words that can be included in the format of a clause, an entry, or a statement in order to improve readability. They possess no significance and have no effect on the program execution.

Examples: GIVING, ROUNDOFF, AFTER, etc.,

Figurative constants

Refer to section 2.2.3

1.4.3. User-defined words

Unlike a reserved word, a user-defined word is constructed and used by the application programmer. They are normally used to define paragraph names, SECTION names, file names, temporary variables, etc., The following are the rules for forming user-defined words.

- Length may be up to 30 characters.
- Only letters, digits and hyphen (-) are allowed.
- Embedded blanks are not allowed.
- At least one character must be alphabetic.
- Cannot be COBOL reserved words.
- May not begin or end with hyphen.

Examples: A0000-MAIN SECTION, A1000-PROCESS-PARA, FS-EMP-NAME, WS-INDEX, etc.,

1.4.4. Phrase

A phrase is an ordered set of COBOL words that form a portion of a COBOL statement or a clause.

Examples: AT END, UNTIL COUNT, END-PERFORM, etc.,

1.4.5. Clause

A clause is an ordered set of COBOL words that specify an attribute of an entry.

Examples: PICTURE X (5), VALUE, SELECT. . . ASSIGN. . . , etc.,

1.4.6. Statement

A statement is a valid combination of a COBOL verb and its operands. It specifies an action to be taken by the object program. The COBOL statements can be broadly classified into two types viz. Imperative statements and Conditional statements. An imperative statement begins with a verb and specifies an unconditional action to be taken. On the other hand, a conditional statement is one in which the action to be taken is determined by some condition that is evaluated when the program is executed.

Examples:

1. Imperative statement: MOVE FS-EMP-NAME TO WS-EMP-NAME.
2. Conditional statement: IF MARKS > 80 THEN MOVE "DISTINCTION" TO REMARKS.

1.4.7. Sentence

A sentence is a sequence of one or more statements, ending with a period.

Example:

```
MOVE 5 TO WS-INDEX
MULTIPLY WS-INDEX BY 2
DISPLAY "The value of the index is" WS-INDEX.
```

1.4.8. Paragraph

A paragraph is a collection of sentences that form a logical unit in a COBOL program. It is the basic unit of organisation of a COBOL program and is referred by a user-defined name in the program.

Example:

```
A0000-MAIN-PARA.
        . . . . . .
        . . . . . .
        . . . . . .
```

1.4.9. SECTION

A SECTION is a collection of related paragraphs. The IDENTIFICATION DIVISION doesn't contain any SECTIONs, the ENVIRONMENT AND DATA DIVISIONs have pre-defined SECTIONs that you may or may not include, depending on the requirement and the PROCEDURE DIVISION may contain only user defined SECTIONs. A user defined SECTION is indicated by a user-defined word followed by the reserved word SECTION. It is used mainly to divide the executable program into units that can be loaded and executed independently. Thus, a program larger than the memory size can be accommodated in memory.

Example: (User defined Section)

```
A0000-MAIN-SECTION.

A1000-INPUT-PARA.
        . . . . . .
        . . . . . .

A2000-VALIDATE-PARA.
        . . . . . .
        . . . . . .

A3000-PROCESS-PARA.
```

```
          . . . . . .
          . . . . . .
A4000-OUTPUT-PARA.
          . . . . . .
          . . . . . .
```

1.4.10. DIVISION

A DIVISION is the largest unit in the COBOL program and is a collection of SECTIONs and/or paragraphs. Every COBOL program is divided into four DIVISIONs viz. IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, DATA DIVISION and PROCEDURE DIVISION. Although the ENVIRONMENT DIVISION and DATA DIVISION are optional, most programs will contain all the four DIVISIONs. We shall now briefly discuss some of the details of the four COBOL DIVISIONs.

1.5. Overview of the four DIVISIONs

1.5.1. IDENTIFICATION DIVISION

The IDENTIFICATION DIVISION is the first and the least significant DIVISION of every COBOL source program. It is used to identify program to the computer. It has no effect on the execution of the program, but is nevertheless, mandatory. It is further divided into paragraphs, not SECTIONs. The only mandatory paragraph of this DIVISION is the PROGRAM-ID paragraph. The other paragraphs are optional and are used mainly for documentation purposes.

Format:

IDENTIFICATION DIVISION.	
PROGRAM-ID.	Program-name.
AUTHOR.	Comment-entry.
INSTALLATION.	Comment-entry.
DATE-WRITTEN.	Comment-entry.
DATE-COMPILED.	Comment-entry.
SECURITY.	Comment-entry.

Figure1.1 Entries of IDENTIFICATION DIVISION

Note: Most compilers permit IDENTIFICATION DIVISION to be abbreviated as ID DIVISION.

PROGRAM-ID

The PROGRAM-ID paragraph specifies the name by which the program is known and assigns selected program attributes to that program. It is mandatory and must be the first paragraph in the Identification Division.

AUTHOR

Refers to the name of the programmer. It has no effect on the program execution.

INSTALLATION

Refers to the name of the company or location. It has no effect on the program execution.

DATE-WRITTEN
Refers to the date on which the program was coded. It has no effect on the program execution.

DATE-COMPILED
Refers to the date on which the program was compiled. Some compilers generate the entry corresponding to the DATE-COMPILED paragraph automatically whenever a program is compiled. This is not the case with our compiler on System 390.

SECURITY
Indicates the level of confidentiality of the program. It has no effect on the program execution.

1.5.2. ENVIRONMENT DIVISION

The ENVIRONMENT DIVISION is an optional DIVISION and comprises of the two SECTIONs namely CONFIGURATION SECTION and INPUT-OUTPUT SECTION. Most of the entries in this DIVISION are machine dependent and hence readers are advised to consult system manuals for these implementation specific information.

CONFIGURATION SECTION

The CONFIGURATION SECTION consists of two paragraphs namely SOURCE-COMPUTER paragraph and OBJECT-COMPUTER paragraph. The SOURCE-COMPUTER paragraph specifies the computer on which the source program is to be compiled and the OBJECT-COMPUTER paragraph specifies the computer for which the object program is designated.

INPUT-OUTPUT SECTION

The INPUT-OUTPUT SECTION consists of two paragraphs namely FILE CONTROL paragraph and I-O CONTROL paragraph. In the FILE CONTROL paragraph a file name is selected for each file to be used in the program and assigned to a device. The FILE CONTROL paragraph will be discussed in detail in section 4.4.1. The I-O CONTROL paragraph specifies information needed for efficient transmission of data between the external file and the COBOL program.

Format:

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.    Comment-entry.
OBJECT-COMPUTER.    Comment-entry.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        File Control entries.
I-O CONTROL.
        I-O Control entries.
```

Figure 1.2 Entries of ENVIRONMENT DIVISION

1.5.3. DATA DIVISION

The DATA DIVISION is used to declare the data items that will be processed in the PROCEDURE DIVISION of the source program. Though there are seven different SECTIONs in DATA DIVISION, which appear in the order shown in Figure 1.3, at this point we discuss only the FILE SECTION and WORKING-STORAGE SECTION. The discussions on LOCAL-STORAGE SECTION, COMMUNICATION SECTION and REPORT SECTION are beyond the scope of the present course. Similar to the ENVIRONEMTN DIVISION, this is also an optional DIVISION and will not be required if the program doesn't use any data items (including files).

Format:

DATA DIVISION.
FILE SECTION.
FD entries.
SD entries.
WORKING-STORAGE SECTION.
Declaration of temporary variables and record structures.
LOCAL-STORAGE SECTION.
Declaration of temporary variables and record structures.
LINKAGE SECTION.
Declaration of temporary variables and record structures.
COMMUNICATION SECTION.
Communication section entries.
REPORT SECTION.
RD entries.
SCREEN SECTION.
Screen section entries.

Figure1.3 Entries of DATA DIVISION

FILE SECTION

The FILE SECTION is used to declare the FD (File descriptor) and SD (Sort descriptor) entries for files and sort files used in the program.

WORKING-STORAGE SECTION

The WORKING-STORAGE SECTION is used to declare all the temporary variables and record structures used in the program.

LOCAL-STORAGE SECTION

The LOCAL-STORAGE SECTION defines storage that is allocated and freed on a per-invocation basis. On each invocation, data items defined in the Local-Storage Section are reallocated and initialized to the value assigned in their VALUE clauses.

LINKAGE SECTION

The LINKAGE SECTION is used to describe data items made available from another program. It is important to note that storage is not allocated to the data structures defined in the LINKAGE SECTION as the data exists elsewhere.

COMMUNICATION SECTION

The COMMUNICATION SECTION is used to create a control block required for communication, which is release dependent.

REPORT SECTION

The REPORT SECTION is used to describe RD (Report Descriptor) entries, which will be required for Report Writer Module.

SCREEN SECTION

The SCREEN SECTION is used to describe labels and data items, which will be required for formatted input/output in case of application programs developed for Personal computers.

1.5.4. PROCEDURE DIVISION

The PROCEDURE DIVISION is the most significant DIVISION of a COBOL program. It includes statements and sentences necessary for reading input, processing it and writing the output.

1.6. COBOL Coding format

COBOL is a high level language and hence a COBOL source program must be written in a format acceptable to the compilers. In the ANSI format, each line is considered to be of 80 columns and these columns are divided into five fields. The following table describes these five fields.

Columns	Use	Description
1 – 6	Sequence numbers or Page and Line numbers (Optional)	Used in the olden days for sequence checking when programs were punched into cards (We advise you to omit this)
7	Continuation, comment, or form feed	- To continue non numeric literals from the previous line * To comment the line / To indicate the printer to skip to a new page while printing the source program
8 - 11	Area A	Used for special entries such as DIVISION, SECTION, Paragraph names and some items of the DATA DIVISION
12 – 72	Area B	Used for most COBOL entries, including PROCEDURE DIVISION sentences
73 - 80	Identification Area	Used in olden days for identification purposes (We advise you to omit this)

Table 1.2 COBOL Coding Format

1.7. Rules for interpreting instruction formats

1. Uppercase words are COBOL reserved words and have special significance to the compiler.
2. Underlined words are mandatory.
3. Lowercase words are user-defined words.
4. Braces denote that one of the enclosed items is required.
5. Square brackets mean the clause of paragraph is optional.
6. The use of dots or ellipses (. . .) means that additional entries of the same type may be included if desired.

1.8. Terminal input/output verbs

1.8.1. ACCEPT verb

The ACCEPT verb is used to read low volume data form the operator's console, some other hardware device or from the operating system.

Format:

```
ACCEPT identifier [ FROM { mnemonic – name } ]
                         { DATE            }
                         { DAY             }
                         { TIME            }
```

If the FROM option is omitted, then the data is read into the identifier with left JUSTIFICATION from the operator's console. However, if the FROM option is included, then the data is read from either the specified hardware device or the operating system.

Examples:

1. ACCEPT WS-NUMBER-1.

   Accepts a value for WS-NUMBER-1 from the operator's console.

2. ACCEPT WS-TODAY-DATE FROM DATE YYYYMMDD.

   Transfers system date in the YYYYMMDD format to the identifier WS-TODAY-DATE.

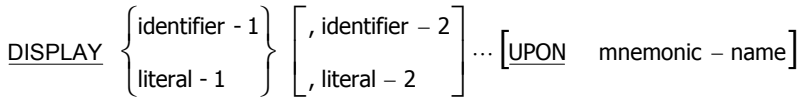
3. ACCEPT FS-EMP-NAME FROM INP-DEV.

   Accepts a value for FS-EMP-NAME from an input device whose mnemonic name is INP-DEV.

1.8.2. DISPLAY verb

The function of DISPLAY verb is exactly opposite to that of the ACCEPT statement. It is used to display low-volume results on the operator's console or some other output device.

Format:



Examples:

1. DISPLAY "Hello World!".
2. DISPLAY "The time now is " WS-PRESENT-TIME.
3. DISPLAY WS-AMOUNT UPON OUT-DEV. Here, OUT-DEV is the mnemonic name of some output device.

1.9. Program to display "Hello World!"

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
PROCEDURE DIVISION.
    DISPLAY "HELLO WORLD!"
    STOP RUN.
```

Remarks: This is the simplest and shortest COBOL program that one can develop. Notice that the following things in the program.

Hello is chosen as the PROGRAM-ID. As a convention it should be same as the member name.

There is no SECTION or paragraph in the PROCEDURE DIVISION. This is perfectly OK for a trivial program, which contains only one paragraph in the PROCEDURE DIVISION.

The DISPLAY and STOP RUN statements are coded in Area B.

The STOP RUN statement is used to specify the end of the program to the compiler. Refer to section 3.3.9 for more details.

2 DATA DIVISION Re-visited

OVERVIEW

1. Basic data types.
2. Literal and its classification.
3. Data-names and their description.
4. RENAMES, USAGE, SYNCHRONIZED, REDEFINES and BLANK WHEN ZERO clauses.
5. Arithmetic verbs {ADD, SUBTRACT, MULTIPLY, DIVIDE and COMPUTE}.
6. ROUNDED and ON SIZE Phrases.

2.1. Basic Data Types

There are five basic data types in COBOL. They are

1. Alphabetic (Made up of only upper and lower case letters),
2. Numeric (Made up of the digits 0, 1, 2, . . . ,9)
3. Alphanumeric data (Made of both letters and digits),
4. Edited Numeric (Made up of digits and special characters) and
5. Edited Alphanumeric (Made up of letters, digits and special characters).

2.2. Literal

A Literal is a symbol whose value does not change in a program. It is also known as constant. There are three types of literals in COBOL namely

1. Numeric literal,
2. Non-numeric literal and
3. Figurative constant.

2.2.1. Numeric literal

A numeric literal is formed using only digits. The following are the rules to be followed while forming a numeric literal.

- May include a sign, which must be the extreme left character,
- There should be no blank between the sign and the first digit,
- May include a decimal point, which must not be the rightmost character and
- Can have at most 18 digits.

Examples:

Valid	Invalid
42	1,000 comma can't be used
+15.8	15. rightmost character can't be a decimal point
-0.97	17.45- sign always precedes numeral

2.2.2. Alphanumeric Literal

An alphanumeric literal is used to display headings and labels. It is a string of characters from the COBOL character set (except quotes) enclosed in single or double quotation marks and can be up to 160 characters long.

Examples:

Valid	Invalid
'ABC 123'	'ABC'S ACCOUNT'
'$1000.00'	'ABC QUOTE'S ACCOUNT'
"MESSAGES"	"MESSAGES'

2.2.3. Figurative Constants

Figurative constants are reserved words that refer to specific constant values. The following are figurative constants and their meanings.

ZERO/ZEROS/ZEROES

Represents the numeric value zero (0), or one or more occurrences of the nonnumeric character zero (0), depending on context. When the context cannot be determined, a nonnumeric zero is used.

SPACE/SPACES

Represents one or more blanks or spaces. SPACE is treated as a nonnumeric literal.

HIGH-VALUE/HIGH-VALUES

Represents one or more occurrences of the character that has the highest ordinal position in the collating sequence used. For the EBCDIC collating sequence, the character is X'FF'; for other collating sequences, the actual character used depends on the collating sequence indicated by the locale. Note that HIGH-VALUE is treated as a nonnumeric literal.

LOW-VALUE/LOW-VALUES

Represents one or more occurrences of the character that has the lowest ordinal position in the collating sequence used. For the EBCDIC collating sequence, the character is X'00'; for other collating sequences, the actual character used depends on the collating sequence indicated by the locale. Note that LOW-VALUE is treated as a nonnumeric literal.

QUOTE/QUOTES

Represents one or more occurrences of the quotation mark character ("). QUOTE or QUOTES cannot be used in place of a quotation mark to enclose a nonnumeric literal.

Note: The singular and plural forms of ZERO, SPACE, HIGH-VALUE, LOW-VALUE and QUOTE can be used interchangeably. For example, if data-name-1 is a 5-character data item, each of the following statements will fill

data-name-1 with five spaces. (1) MOVE SPACE TO DATA-NAME-1, (2) MOVE SPACES TO DATA-NAME-1 and (3) MOVE ALL SPACES TO DATA-NAME-1.

2.3. Data-names

Data-names are named memory locations. They must be described in the DATA DIVISION before they can be used in the PROCEDURE DIVISION. They can be of elementary or group type and further can be qualified or subscripted. They are user-defined words.

Note: A data-name of elementary type is known as an identifier.

2.3.1. Description of Data-names

Data-names are described with the aid of the following

1. Level Number,
2. PICTURE Clause and
3. VALUE Clause.

Level numbers

Level numbers specify the hierarchy of data within a record and identify special-purpose data entries. A level number begins a data description entry and has a value taken from the set of integers between 1 and 49, or from one of the special level-numbers, 66, 77, or 88. The following are the significance of the various level numbers that can be used with data items.

- 01 - For record descriptions.
- 02 to 49 - For fields within records.
- 01 / 77 - For independent items (use 01, as 77 will be removed in future releases)
- 66 - For RENAMES clause.
- 88 - For condition names.

Remarks:

(1) Level numbers 01 and 77 must begin in Area A and must be followed either by a period or by a space, followed by its associated data-name, FILLER, or appropriate data description clause. Level numbers 02 through 49 can begin in Areas A or B and must be followed by a space or a separator period. Level numbers 66 and 88 can begin in Areas A or B and must be followed by a space. Single-digit level-numbers 1 through 9 can be substituted for level-numbers 01 through 09. Successive data description entries can start in the same column as the first or they can be indented according to the level number. Indentation does not affect the magnitude of a level number. When level-numbers are indented, each new level-number can begin any number of spaces to the right of Area A. The extent of indentation to the right is limited only by the width of Area B. Refer to COBOL coding standards in the QSD for indentation details.

(2) After a record has been defined, it can be subdivided to provide more detailed data references. For example, in an employee file for a department, one complete record could contain all data pertaining to one employee. Subdivisions within that record could be employee number, employee name, employee's mail-id, employee's

grade, etc., The basic subdivisions of a record (that is, those fields that are not further subdivided) are called elementary items. Thus, a record can be made up of a series of elementary items, or it can itself be an elementary item. It might be necessary to refer to a set of elementary items; thus, elementary items can be combined into group items. Further groups themselves can be combined into a more inclusive group that contains one or more subgroups.

PICTURE clause

The PICTURE clause specifies the data type and the amount of storage required for a data item. It is denoted by PICTURE (often abbreviated as PIC). A PICTURE clause is specified only for elementary data items and consists of picture characters. Each picture character denotes storage to be reserved for a character of that type. In case of recurring picture characters, abbreviation is allowed. The following are the general picture characters and their meaning.

- A for alphabetic
- X for alphanumeric
- 9 for numeric
- S for sign
- V for implied decimal point.

VALUE clause

The VALUE clause is an optional clause that is used to specify the initial value for the data item. The initial value can be a numeric literal, non-numeric literal or a figurative constant. This initial value can be changed in the PROCEDURE DIVISION based on the requirement.

Example:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  WS-EMPLOYEE-NUMBER        PIC X (5).
01  WS-EMPLOYEE-NAME.
    05  WS-FIRST-NAME  PIC A (15).
    05  WS-MIDDLE-NAME        PIC A (15).
    05  WS-LAST-NAME   PIC A (15).
01  WS-AGE                    PIC 9(2)V99.
01  WS-BONUS                  PIC 9(4)    VALUE 1000.
```

2.3.2. Edited PICTURE Symbols

There are two general methods of editing in a PICTURE clause.

Insertion

There are four types of Insertion editing namely

1. Simple insertion,
2. Special insertion,

3. Fixed insertion and
4. Floating insertion.

Suppression and Replacement

This is of two kinds

1. Zero suppression and replacement with asterisk and
2. Zero suppression and replacement with spaces.

The type of editing allowed for a data item based on its category is given in figure 2.1.

Data Category	Type of Editing	Insertion Symbol
Numeric	Simple Insertion	B, 0 and /
	Special Insertion	.
	Fixed Insertion	$, +, -, CR and DB
	Floating Insertion	$, + and –
	Zero Suppression	Z and *
	Replacement	Z, *, +, - and $
Alphanumeric Edited	Simple Insertion	B, 0 and /

Figure 2.1 Type of Editing

Figure 2.2 gives the list of edit PICTURE characters and their meanings.

Symbol	Meaning
Z	Zero suppression character.
.	Decimal point insertion.
+	Plus sign insertion.
-	Minus sign insertion.
$	Dollar sign insertion.
,	Comma insertion.
CR	Credit symbol.
DB	Debit symbol.
*	Check protection symbol.
B	Field separator - blank insertion character.
0	Zero insertion character.

Figure 2.2 Editing PICTURE characters

Examples:

1. Simple Insertion

Picture Clause	Contents of Data item	Edited Result
X(7)/X(7)	INFOSYSLIMITED	INFOSYS / LIMITED
X(7)BX(7)	INFOSYSLIMITED	INFOSYS LIMITED
99,999	12345	12,345
99,B999,B000	1234	01, 234,000

2. Special Insertion

Picture Clause	Contents of Data item	Edited Result
999.99	1.234	001.23
999.99	12.34	012.34
999.99	123.45	123.45
999.99	1234.5	234.50

3. Fixed Insertion

Picture Clause	Contents of Data item	Edited Result
999.99+	+1234.567	234.56+
+9999.99	-1234.567	-1234.56
9999.99	+1234.56	1234.56
$999.99	123.45	$123.45
-$999.99	-123.456	-$123.45
-$999.99	+123.456	$123.45
$9999.99CR	+123.45	$123.45
$9999.99DB	-123.45	$0123.45DB

2.3.3. RENAMES clause (66-level)

The RENAMES clause specifies alternative, possibly overlapping and groupings of elementary data items.

Format: 66 data-name-1 RENAMES data-name-2 THROUGH or THRU data-name-3.

The special level-number 66 must be specified for data description entries that contain the RENAMES clause. Level number 66 and data-name-1 are not part of the RENAMES clause itself and are included in the format only for clarity. One or more RENAMES entries can be written for a logical record. All RENAMES entries associated with one logical record must immediately follow that record's last data description entry.

Data-name-1 identifies an alternative grouping of data items. A level-66 entry cannot rename a level-01, level-77, level-88, or another level-66 entry. Data-name-1 cannot be used as a qualifier; it can be qualified only by the names of level indicator entries or level-01 entries. Data-name-2 and data-name-3 identify the original grouping of elementary data items; that is, they must name elementary or group items within the associated level-01 entry and must not be the same data-name. Both data-names can be qualified.

When data-name-3 is specified, data-name-1 is treated as a group item that includes all elementary items starting with data-name-2 (if it is an elementary item) or the first elementary item within data-name-2 (if it is a group item) and ending with data-name-3 (if it is an elementary item) or the last elementary item within data-name-3 (if it is a group item). Further when data-name-2 is a group item, data-name-1 is treated as a group item and when data-name-2 is an elementary item, data-name-1 is treated as an elementary item.

Examples:

1. (Valid)

```
01 RECORD-1.
        05 FIELD-1 PIC X (5).
        05 FIELD-2 PIC X (5).
        05 FIELD-3 PIC X (5).
        05 FIELD-4 PIC X (5).
66 RECORD-2 RENAMES FIELD-1 THROUGH FIELD-3.
```

2. (Invalid)

```
01 RECORD-3.
        05 FIELD-1 PIC.
                10 FIELD-11 PIC X (5).
                10 FIELD-12 PIC X (5).
        05 FIELD-2 PIC X (5).
66 RECORD-4 RENAMES FIELD-12 THROUGH FIELD-2.
```

2.3.4. USAGE clause

Normally, a computer can store data in more than one internal form. In COBOL, a programmer is allowed to specify the internal form of the data item so as to facilitate its use in the most efficient manner. Broadly speaking, there are only two general forms of internal representation namely COMPUTATIONAL and DISPLAY. Only numeric data items can be specified as USAGE IS COMPUTATIONAL and the name itself suggests a data item specified as USAGE IS COMPUTATIONAL can take part in arithmetic operations more efficiently. On the other hand, any data item can be specified as USAGE IS DISPLAY. This form is suitable for input-output and character manipulations.

Syntax:

USAGE IS {DISPLAY, COMPUTATIONAL, COMP} [- {1, 2, 3}].

USAGE is DISPLAY

This is the most common form of internal data. Each character of the data is represented in one byte. The default USAGE of a data item is DISPLAY.

USAGE IS COMPUTATIONAL

In this case the data item is represented in pure binary. The item must be an integer. Such data items are often used as subscripts. Depending on the size of the data item, it can be stored either in a half-word (2 bytes with range –32,768 to +32767) or full-word (4 bytes with range –2,147,483,648 to 2,147,483,647). The PICTURE Clause of a COMPUTATIONAL data item should not contain any character other than 9 or S.

USAGE IS COMP-1

In this case the data item will be represented in one word in the floating point form. The number is actually represented in hexadecimal format and is suitable for arithmetic operations. The PICTURE Clause cannot be specified for COMP-1 items.

USAGE IS COMP-2

This is same as COMP-1, except that the data is represented internally in two words. The advantage is that this increases the precision of the data, which means that more significant digits are available. Similar to COMP-1, the PICTURE Clause cannot be specified for COMP-2 items also.

USAGE IS COMP-3

In this case the data is represented in the decimal form, but one digit takes half a byte. The sign is stored separately as the rightmost half a byte character regardless of whether S is specified in the PICTURE Clause or not. The hexadecimal number C or F denotes a positive sign and the hexadecimal number D denotes a negative sign.

Remarks:

1. When an elementary data item is moved to another elementary data item, their USAGES need not be the same. Conversion of USAGE will take place automatically if they differ.
2. The USAGE Clause can be specified data items defined at with any level number.
3. The USAGE of a group item is valid for all its sub items.

NOTE: Refer to the **APPENDIX-D** to see different numeric PICTURE clauses in detail.

2.3.5. SYNCHRONIZED or SYNC clause

Most processors are efficient at picking up data which start on a word boundary (a word is usually 2 bytes) and not so efficient at picking up data from a byte boundary. In COBOL, data items are not normally word aligned. To cause alignment along a word boundary, the SYNC clause is used. It is obvious that this form of alignment is likely to lead to some unused memory area between data items. It can be used with numeric or alphanumeric data items, but they must be elementary items.

Format:

{SYNCHRONIZED/SYNC} [LEFT/RIGHT]

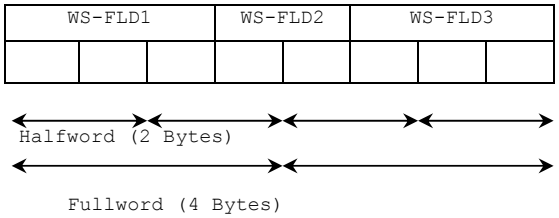
Example:

```
01      WS-INDEX        PIC S9 (04) SYNC.
```

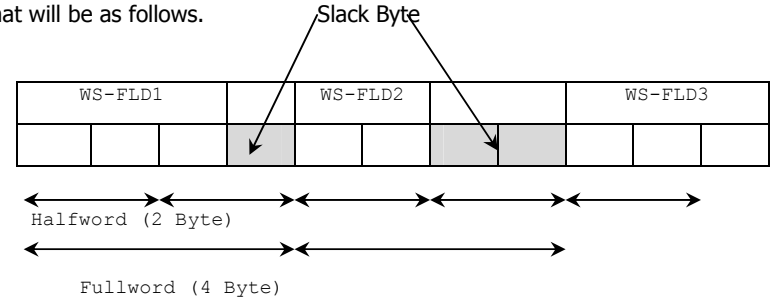
The following example shows use of SYNC clause in detail.

```
01  WS-REC.
  05  WS-FLD1  PIC X(3).
  05  WS-FLD2  PIC S9(4) USAGE COMP SYNC.
  05  WS-FLD3  PIC S9(5) USAGE COMP SYNC.
```

If we don't mention the clause SYNC against the subordinate fields WS-FLD2 and WS-FLD2 then the storing format will be as follows.



If we mention the clause SYNC against the subordinate fields WS-FLD2 and WS-FLD2 then the storing format will be as follows.



Here WS-FLD1 will occupy first 3 bytes. Since WS-FLD2 (of S9(4) COMP) is declared as SYNC, It will align at Half-word boundary. Here halfward boundaries are 3rd bytes, 5th byte, 7th byte, 9th byte and so on. Since, WS-FLD1 occupies first 3 bytes, so WS-FLD2 will start at 5th byte leaving an unused space (Slack bytes) at 4th byte. Likewise, WS-FLD3 (of S9(5) COMP) having SYNC clause can start at fullward boundary. Here, WS-FLD1 and WS-FLD2 takes first 6 bytes (including Slack byte). After 6 bytes the Fullward boundary starts at 9th byte. Hence WS-FLD3 will be stored in 9th to 11th byte, leaving 7th and 8th byte unused. Using SYNC clause we are wasting memory space (for slack byte) space and improving performance by faster address resolution of binary data.

Here are some rules for alignment at halfword and Fullword boundary.

- The item is aligned at halfword (Multiple of 2 bytes) boundary, if its USAGE is BINARY (i.e. COMP) and its PICTURE is in the range of S9 through S9(4).
- The item is aligned at Fullword (Multiple of 4 bytes) boundary, if its USAGE is BINARY (i.e. COMP) and its PICTURE is in the range of S9(5) through S9(18).

If **SYNC LEFT** is specified, the leftmost character will occupy the leftmost position in the contiguous memory area. The right-hand positions of the area will be unoccupied. Likewise, if **SYNC RIGHT** is

specified, the rightmost character will occupy the right-hand position in the contiguous memory area with leftmost positions of the area unoccupied.

Note: The SYNCHRONIZED clause is meaningless for COMP-3 and DISPLAY items, and the compiler ignores it.

2.3.6. REDEFINES clause

The REDEFINES clause allows you to use different data description entries to describe the same computer storage area.

Format:

Level number {data-name-1, FILLER} REDEFINES data-name-2.

Here,

- Level number, data-name-1 and FILLER are not part of the REDEFINES clause itself and are included in the format only for clarity.
- Level-numbers of data-name-1 and data-name-2 must be identical and must not be level 66 or level 88.
- Data-name-1, FILLER identifies an alternate description for the same area and is the redefining item or the REDEFINES subject.
- Data description entry for the redefined item cannot contain an OCCURS clause.
- More than one redefinition of the same storage area is permitted. The entries giving the new descriptions of the storage area must immediately follow the description of the redefined area without intervening entries that define new character positions.
- Multiple redefinition must all use the data-name of the original entry that defined this storage area.
- Redefining entry (identified by data-name-1) and any subordinate entries, must not contain any VALUE clauses.

Examples:

(1)
```
05  REGULAR-EMPLOYEE.
    10 LOCATION                                  PICTURE A (8).
    10 GRADE                                     PICTURE X (4).
    10 SEMI-MONTHLY-PAY                          PICTURE 9(4)V9(2).
    10 WEEKLY-PAY REDEFINES SEMI-MONTHLY-PAY     PICTURE 9(3)V9(3).
  05 TEMPORARY-EMPLOYEE REDEFINES REGULAR-EMPLOYEE.
    10 LOCATION                                  PICTURES A (8).
    10 FILLER                                    PICTURE X (6).
    10 HOURLY-PAY                                PICTURE 99V99.
```

(2)
```
05  REGULAR-EMPLOYEE.
    10 LOCATION                                  PICTURE A (8).
    10 GRADE                                     PICTURE X (4).
```

```
    10 SEMI-MONTHLY-PAY                          PICTURE 9(3)V9(3).
05 TEMPORARY-EMPLOYEE REDEFINES REGULAR-EMPLOYEE.
    10 LOCATION                                  PICTURE A (8).
    10 FILLER                                    PICTURE X (6).
    10 HOURLY-PAY                                PICTURE 99V99.
    10 CODE-H REDEFINES HOURLY-PAY               PICTURE 9(4).
```

2.3.7. BLANK WHEN ZERO clause

The BLANK WHEN ZERO clause specifies that an item contains nothing but spaces when its value is zero. It can be specified only for elementary numeric or numeric-edited items. These items must be described, either implicitly or explicitly, as USAGE IS DISPLAY. When the BLANK WHEN ZERO clause is specified for a numeric item, the item is considered a numeric-edited item.

Note: The BLANK WHEN ZERO clause must not be specified for level-66 or level-88 items.

2.4. Arithmetic verbs

2.4.1. ADD verb

The ADD statement sums two or more numeric operands and stores the result.

Format 1:

$$\underline{\text{ADD}} \left\{ \begin{matrix} \text{identifier - 1} \\ \text{literal - 1} \end{matrix} \right\} \cdots \underline{\text{TO}} \ \text{identifier} - 2$$

Here, all identifiers or literals preceding the key word TO are added together and this sum is added to and stored in identifier-2. This process is repeated for each successive occurrence of identifier-2, in the left-to-right order in which identifier-2 is specified.

Examples:

1. ADD DEPOSIT TO BALANCE.

   In this case the value of DEPOSIT is added to the value of BALANCE. The value of DEPOSIT will remain unaltered after addition.

2. ADD 12 TO WS-NUMBER-1 WS-NUMBER-2.

Format 2:

$$\underline{\text{ADD}} \left\{ \begin{matrix} \text{identifier - 1} \\ \text{literal - 1} \end{matrix} \right\} \cdots \underline{\text{GIVING}} \ \text{identifier} - 2 \cdots$$

Here, the values of the operands preceding the word GIVING are added together and the sum is stored as the new value in each successive occurrence of identifier-2.

Examples:

1. ADD MARKS-1 MARKS-2 MARKS-3 GIVING TOTAL-MARKS. In this case the values of MARKS-1, MARKS-2 and MARKS-3 are added to the value of TOTAL-MARKS and the values of MARKS-1, MARKS-2 and MARKS-3 remain unaltered after addition.

2. ADD 25 AMOUNT GIVING NEW-AMOUNT.

3. ADD NUMBER-1 NUMBER-2 GIVING NUMBER-3 NUMBER-4 NUMBER-5.

Format 3:

$$\underline{\text{ADD}} \left\{ \begin{matrix} \text{identifier - 1} \\ \text{literal - 1} \end{matrix} \right\} \cdots \underline{\text{TO}} \left\{ \begin{matrix} \text{identifier - 2} \\ \text{literal - 2} \end{matrix} \right\} \underline{\text{GIVING}} \text{ identifier} - 3 \cdots$$

Here, all identifiers or literals preceding the key word TO are added together and this sum is added to and stored in identifier-2. This process is repeated for each successive occurrence of identifier-2, in the left-to-right order in which identifier-2 is specified.

Examples:

1. ADD 10 20 30 40 50 TO BALANCE GIVING NEW-BALANCE. Here sum of 10, 20, . . . , 50 is subtracted from BALANCE and the result is stored in NEW-BALANCE. The value of BALANCE will remain unaltered.

2. ADD NUM-1 NUM-2 TO NUM-3 GIVING NUM-4.

Format 4:

$$\underline{\text{ADD}} \ \underline{\text{CORRESPONDING}} \text{ dataname} - 1 \text{ TO dataname - 2} \cdots$$

Here, the elementary data items within data-name-1 are added to and stored in the corresponding elementary items within data-name-2.

2.4.2. SUBTRACT verb

The SUBTRACT statement subtracts one numeric item, or the sum of two or more numeric items, from one or more numeric items and stores the result.

Format 1:

$$\underline{\text{SUBTRACT}} \left\{ \begin{matrix} \text{identifier - 1} \\ \text{literal - 1} \end{matrix} \right\} \cdots \underline{\text{FROM}} \text{ identifier} - 2 \cdots$$

Here, all identifiers or literals preceding the key word FROM are added together and this sum is subtracted from and stored immediately in identifier-2. This process is repeated for each successive occurrence of identifier-2, in the left-to-right order in which identifier-2 is specified.

Examples:

1. SUBTRACT WITHDRAW FROM BALANCE. In this case the value of WITHDRAW will be subtracted from the value of BALANCE and stored in BALANCE. The value of WITHDRAW will remain unaltered after subtraction.

2. SUBTRACT 10 20 NUMBER FROM OLD-NUMBER.

Format 2:

SUBTRACT { identifier - 1 / literal - 1 } ... FROM { identifier – 2 / literal - 2 } GIVING identifier - 3. . .

Here, all identifiers or literals preceding the key word FROM are added together and this sum is subtracted from identifier-2 or literal-2. The result of the subtraction is stored as the new value of each data item referenced by identifier-3.

Examples:

1. SUBTRACT WITHDRAW FROM BALANCE GIVING NEW-BALANCE.
   In this case the value of WITHDRAW will be subtracted from the value of BALANCE and stored in NEW-BALANCE. The value of WITHDRAW and BALANCE will remain unaltered after subtraction.
2. SUBTRACT 10 FROM NUMBER GIVING NEW-NUMBER.
3. SUBTRACT NUMBER-1 FROM NUMBER-2 GIVING NUMBER-3 NUMBER-4 NUMBER-5.

Format 3:

SUBTRACT CORRESPONDING identifier – 1 TO identifier – 2 . . .

Here, the elementary data items within identifier-1 are subtracted and stored in the corresponding elementary items within identifier-2.

2.4.3. MULTIPLY verb

The MULTIPLY statement multiplies numeric items and sets the values of data items equal to the results.

Format 1:

MULTIPLY { identifier - 1 / literal - 1 } BY identifier – 2. . .

Here, the value of identifier-1 or literal-1 is multiplied by the value of identifier-2; the product is then placed in identifier-2. For each successive occurrence of identifier-2, the multiplication takes place in the left-to-right order in which identifier-2 is specified.

Examples:

1. MULTIPLY NUMBER-1 BY NUMBER-2.
2. MULTIPLY 5 BY NUMBER-2.

Format 2: MULTIPLY { identifier - 1 / literal - 1 } BY { identifier – 2 / literal - 2 } GIVING identifier - 3 ⋯

Here, the value of identifier-1 or literal-1 is multiplied by the value of identifier-2 or literal-2. The product is then stored in the data item(s) referenced by identifier-3.

Examples:

1. MULTIPLY NUMBER-1 BY NUMBER-2 GIVING NUMBER-3 NUMBER-4 NUMBER-5.
2. MULTIPLY DAILY-WAGES BY NUMBER-OF-DAYS GIVING TOTAL-WAGES.

2.4.4. DIVIDE verb

The DIVIDE statement divides one numeric data item into or by other(s) and sets the values of data items equal to the quotient and remainder.

Format 1:

DIVIDE { identifier - 1 / literal - 1 } INTO identifier – 2 ⋯

Here, the value of identifier-1 or literal-1 is divided into the value of identifier-2 and the quotient is then stored in identifier-2. For each successive occurrence of identifier-2, the division takes place in the left-to-right order in which identifier-2 is specified.

Examples:

1. DIVIDE 2 INTO NUMBER-1.
   In this case the value of NUMBER-1 is divided by 2 and is stored in NUMBER-1.
2. DIVIDE NUMBER-1 INTO NUMBER-2.

Format 2:

DIVIDE { identifier - 1 / literal - 1 } INTO { identifier – 2 / literal - 2 } GIVING identifier - 3 ⋯

Here, the value of identifier-1 or literal-1 is divided into the value of identifier-2 or literal-2. The value of the quotient is stored in each data item referenced by identifier-3.

Examples:

1. DIVIDE 60 INTO SECONDS GIVING MINUTES. In this case the value of SECONDS is divided by 60 and is stored in MINUTES.
2. DIVIDE 12 INTO ANNUAL-SALARY GIVING MONTHLY-SALARY.

Format 3:

$$\underline{\text{DIVIDE}} \left\{\begin{matrix}\text{identifier - 1}\\ \text{literal - 1}\end{matrix}\right\} \underline{\text{BY}} \left\{\begin{matrix}\text{identifier} - 2\\ \text{literal - 2}\end{matrix}\right\} \text{ GIVING identifier - 3} \cdots$$

Here, the value of identifier-2 or literal-2 is divided into the value of identifier-1 or literal-1. The value of the quotient is stored in each data item referenced by identifier-3.

Examples:

1. DIVIDE HOURS BY 60 GIVING MINUTES. In this case the value of HOURS is divided by 60 and is stored in MINUTES.

2. DIVIDE ANNUAL-SALARY by 52 GIVING WEEKLY-SALARY.

Format 4:

$$\underline{\text{DIVIDE}} \left\{\begin{matrix}\text{identifier - 1}\\ \text{literal - 1}\end{matrix}\right\} \underline{\text{INTO}} \left\{\begin{matrix}\text{identifier} - 2\\ \text{literal - 2}\end{matrix}\right\} \text{ GIVING identifier - 3 REMAINDER identifier - 4}$$

Here, the value of identifier-1 or literal-1 is divided into the value of identifier-2 or literal-2. The value of the quotient is stored in identifier-3 and the value of the remainder is stored in identifier-4.

Example:

DIVIDE NUMBER-1 INTO NUMBER-2 GIVING NUMBER-3 REMAINDER NUMBER-4.

Format 5:

$$\underline{\text{DIVIDE}} \left\{\begin{matrix}\text{identifier - 1}\\ \text{literal - 1}\end{matrix}\right\} \underline{\text{BY}} \left\{\begin{matrix}\text{identifier} - 2\\ \text{literal - 2}\end{matrix}\right\} \text{ GIVING identifier - 3 REMAINDER identifier - 4}$$

Here, the value of identifier-1 or literal-1 is divided by the value of identifier-2 or literal-2. The value of the quotient is stored in identifier-3 and the value of the remainder is stored in identifier-4.

Example:

DIVIDE NUMBER-1 BY NUMBER-2 GIVING NUMBER-3 REMAINDER NUMBER-4. This example and the example given for Format 4 are equivalent.

2.4.5. COMPUTE verb

The COMPUTE statement assigns the value of an arithmetic expression to one or more data items. With the COMPUTE statement, arithmetic operations can be combined without the restrictions on receiving data items imposed by the rules for the ADD, SUBTRACT, MULTIPLY and DIVIDE statements.

Format:

$$\underline{\text{COMPUTE}}\ \text{identifier} - 1 \ldots = \left\{ \begin{array}{l} \text{arithmeticexpression} \\ \text{literal} - 1 \\ \text{identifier} - 2 \end{array} \right\}$$

Examples:

1. COMPUTE ROOT-1 = ((-B + (B**2 – 4 * A * C) ** (0.5)) / 2 * A).
2. COMPUTE ROOT-2 = ((-B - (B**2 – 4 * A * C) ** (0.5)) / 2 * A).

2.4.6. ROUNDED phrase

After decimal point alignment, the number of places in the fraction of the result of an arithmetic operation is compared with the number of places provided for the fraction of the resultant identifier. When the size of the fractional result exceeds the number of places provided for its storage, truncation occurs unless the ROUNDED phrase is specified. When ROUNDED phrase is specified, the least significant digit of the resultant identifier is increased by 1 whenever the most significant digit of the excess is greater than or equal to 5. When the resultant identifier is described by a PICTURE clause containing rightmost Ps and when the number of places in the calculated result exceeds the number of integer positions specified, rounding or truncation occurs, relative to the rightmost integer position for which storage is allocated.

2.4.7. ON SIZE phrase

A size error condition can occur in following situations.

1. When the absolute value of the result of an arithmetic evaluation, after decimal point alignment, exceeds the largest value that can be contained in the result field
2. When division by zero occurs.

The size error condition applies only to final results, not to any intermediate results. If the ROUNDED phrase is specified, rounding takes place before size error checking. When a size error occurs, the subsequent action of the program depends on whether or not the ON SIZE ERROR phrase is specified.
If the ON SIZE ERROR phrase is not specified and a size error condition occur, truncation rules apply and the value of the affected resultant identifier is computed.

If the ON SIZE ERROR phrase is specified and a size error condition occurs, the value of the resultant identifier affected by the size error is not altered--that is, the error results are not placed in the receiving identifier. After completion of the execution of the arithmetic operation, the imperative statement in the ON SIZE ERROR phrase is executed, control is transferred to the end of the arithmetic statement and the NOT ON SIZE ERROR phrase, if specified, is ignored.

For ADD CORRESPONDING and SUBTRACT CORRESPONDING statements, if an individual arithmetic operation causes a size error condition, the ON SIZE ERROR imperative statement is not executed until all the individual additions or subtractions have been completed. Further, if the NOT ON SIZE ERROR phrase has been specified and, after execution of an arithmetic operation, a size error condition does not exist, the NOT ON SIZE ERROR phrase is executed.

2.5. Program to add two numbers

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ADDITION.
DATA DIVISION.
WORKING-STORAGE SECTION.
      01 NUM1 PIC 99.
      01 NUM2 PIC 99.
      01 NUM3 PIC 999.
PROCEDURE DIVISION.
A1000-MAIN-PARA.
      ACCEPT NUM1.
      ACCEPT NUM2.
      ADD NUM1 TO NUM2 GIVING NUM3.
      DISPLAY "THE SUM OF " NUM1 " AND " NUM2 " IS " NUM3.
      STOP RUN.
```

3 PROCEDURE DIVISION Re-visited

OVERVIEW

1. Data Movement statements.
2. JUSTIFIED RIGHT clause.
3. IF . . . THEN . . . ELSE statement.
4. Classification of conditions.
5. PERFORM statements.
6. EVALUATE . . . WHEN . . .
7. EXIT, CONTINUE and STOP RUN.

3.1. Introduction

The PROCEDURE DIVISION is unmistakably the most significant of all the four DIVISIONs in a COBOL program. It consists of SECTIONs and/or Paragraphs, sentences and statements. The statements specify the operations to be performed by the computer. In Chapter 1, we dealt with COBOL verbs for Terminal input/output and in Chapter 2, we dealt with COBOL verbs for performing arithmetic operations. In this chapter, we will discuss two more types of COBOL verbs viz. Data movement verbs and sequence control verbs.

3.2. Data Movement verb

The MOVE verb transfers data from one area of storage are to one or more other storage areas. After execution of a MOVE statement, the sending field(s) contain the same data as before execution. The category of MOVE that is, numeric or alphanumeric is dictated by the type of the target item. If the target is numeric, then the move is a numeric move, else an alphanumeric move. An alphanumeric move is done from left to right, with possible truncation if the size of the target is less than that of the source. A numeric move is aligned around the decimal point, i.e., digits before the decimal point are moved right to left and after the decimal point, left to right. In the case of a numeric move, too, truncation can occur. It is important to note that overlapping operands in a MOVE statement can cause unpredictable results. There are essentially two types of moves viz. Elementary moves and Group moves.

3.2.1. Elementary moves

An elementary move is one in which the receiving item is an elementary item and the sending item is an elementary item or a literal. Any necessary conversion of data from one form of internal representation to another takes place during the move.

Note: If the size of the sending item is greater than the size of the receiving item, excess characters on the right are truncated after the receiving item is filled.

Format 1:

MOVE {identifier-1, literal} TO identifier-2 [, identifier-3] . . .

Table 3.1 shows the valid and invalid elementary MOVEs.

	Receiving Field					
Sending Field	Alphabetic and Space	Alphanumeric	Edited Alphanumeric	Numeric integer	Numeric non-integer	Edited Numeric
Alphabetic and Space	Y	Y	Y	N	N	N
Alphanumeric	Y	Y	Y	Y	N	Y
Edited Alphanumeric	Y	Y	Y	Y	N	Y
Numeric integer	N	Y	Y	Y	Y	Y
Numeric non-integer	N	N	N	Y	Y	Y
Edited Numeric	N	Y	Y	Y	Y	Y

Table 3.1 Valid and Invalid MOVEs

Examples:

Sending item		Receiving item	
Before sending	After sending	Before sending	After sending
PICTURE 9(3) 123	123	PICTURE 9(4) 4567	0123
PICTURE 9(4) 1234	1234	PICTURE 9(3) 567	234
PICTURE 99V9 12 . 3	12 . 3	PICTURE 9(3)V9(2) 456.78	012 . 30

Sending item		Receiving item	
Before sending	After sending	Before sending	After sending
PICTURE 9(3)V9(2) 123 . 45	123 . 45	PICTURE 99V9 67 . 8	23 . 4
PICTURE X(5) PAYAL	PAYAL	PICTURE X(8) SHEETHAL	PAYALbbb
PICTURE X(8) SHEETHAL	SHEETHAL	PICTURE X(5) PAYAL	SHEET

Table 3.2 Data Movement Examples

Format 2:

MOVE elementary-item-1 OF group-item-1 TO elementary-item-2 OF group-item-2

Example:

```
MOVE FS-NAME OF FS-EMPLOYEE-RECORD TO WS-NAME OF WS-EMPLOYEE-RECORD
```

3.2.2. Group moves

A group move is one in which one or both of the sending and receiving fields are group items. A group move is treated exactly as though it were an alphanumeric elementary move, except that there is no conversion of data from one form of internal representation to another. Hence all group moves are valid.

Format:

MOVE {CORRESPONDING, CORR} {identifier-1, literal} TO identifier-2 [identifier-3] . . .

3.3. JUSTIFIED RIGHT clause

The JUSTIFIED RIGHT clause can be used to change the default movement type of alphabetic and alphanumeric data. It can be specified only for elementary data items and not group items. It is important to note that it cannot be used for Edited alphanumeric items and with level 66 and level 88. JUSTIFIED can be abbreviated as JUST.

3.4. Sequence control verbs (Logical control structures)

The instructions coded in the PROCEDURE DIVISION are executed in the order in which they are written, from top to bottom. Most often, we need to alter this flow. COBOL provides programmers with a number of sequence control verbs that can alter the normal flow. We shall discuss them one by one in detail.

3.4.1. GO TO verb

The GO TO verb is used to transfer control from one part of the Procedure Division to another. There are essentially two types of GO TO statements in COBOL namely (1) Unconditional and (2) Conditional.

Unconditional GO TO

The unconditional GO TO statement transfers control to the first statement in the specified paragraph/section.

Format: GO TO procedure name.

Example: GO TO A4000-READ-PARA.

Conditional GO TO

The conditional GO TO statement transfers control to one of a series of procedures, depending on the value of the identifier.

Format: GO TO procedure name-1 [procedure-name-2 . . .] DEPENDING ON identifier

Here, identifier must be an integer data item. The control will be transferred to procedure name-1, procedure name-2, etc., depending on the value of identifier. The maximum number of procedure names that can be specified is 255.

Example: GO TO 500-INSERT-PARA, 600-UPDATE-PARA, 700-DELETE-PARA DEPENDING ON TRANS-CODE.

Note: Programmers are advised to keep the use of GO TO statements to the minimum possible extent as their use is against the idea of structured programming.

3.4.2. IF . . . THEN . . . ELSE . . .

The IF . . . THEN . . . ELSE . . . statement evaluates a condition and provides for alternative actions depending on the evaluation.

Format:

```
IF condition [THEN] imperative statements
        [ELSE imperative statements]
END-IF
```

Example:
```
IF PERCENTAGE-OF-MARKS >= 50 THEN MOVE "PASS" TO REMARKS
        ELSE MOVE "FAIL" TO REMARKS
END-IF
```

3.4.3. NESTED IFs

The THEN and ELSE parts of an IF statement can contain other IF statements. The included IF statements in turn may also contain other IF statements. Such inclusion of one or more IF statements within the scope of an IF statement is called nesting of IF statements.

Example:
```
IF NUMBER-1 <= 0
        THEN IF NUMBER-2 <= 0
                THEN DISPLAY "The product of the numbers is positive"
        ELSE DISPLAY "The product of the numbers is negative"
ELSE
```

```
                IF NUMBER-2 >= 0
                        THEN DISPLAY "The product of the numbers is positive"
                ELSE DISPLAY "The product of the numbers is negative"
        END-IF
```

3.4.4. Classification of conditions

There are five types of conditions that are valid in COBOL. They are

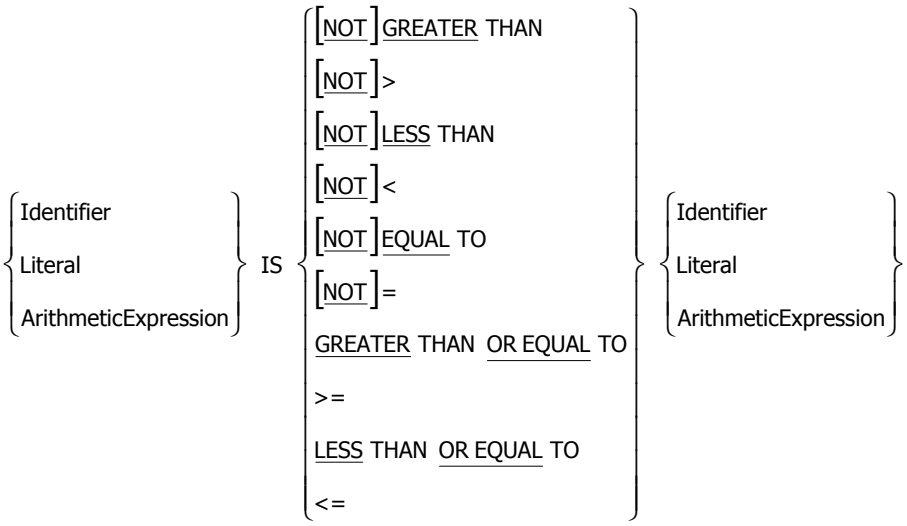
1. Relational condition
2. Sign condition
3. Class condition
4. Compound condition
5. Condition names

We will discuss each one of these in detail.

Relational condition

A relation condition compares two operands, either of which can be an identifier, literal, arithmetic expression, or index-name.

Format:



Sign condition

The sign condition determines whether or not the algebraic value of a numeric operand is greater than, less than, or equal to zero.

Format: Arithmetic expression IS [NOT] {POSITIVE, NEGATIVE, ZERO}

Example: IF DISCRIMINANT IS NEGATIVE THEN DISPLAY "The roots are imaginary".

Class condition

The class condition determines whether the content of a data item is alphabetic, alphabetic-lower, alphabetic-upper, numeric, or some user defined class type.

Format:

Identifier IS [NOT] { NUMERIC | ALPHABETIC | ALPHABETIC - LOWER | ALPHABETIC - UPPER | UserDefinedClassName }

Example: IF REGNO IS NUMERIC THEN DISPLAY "The Primary Key must be alphanumeric".

Compound condition

A compound condition is formed by combining two or more simple conditions and the logical operators.

Format: condition-1 {AND, OR} conditon-2

Examples:

1. IF NUMBER-1 = 0 OR NUMBER-2 = 0 THEN DISPLAY "The product of the numbers is zero".
2. IF NUMBER-1 IS NOT = 0 AND NUMBER-2 IS NOT = 0 THEN DISPLAY "The product of the numbers is non zero".

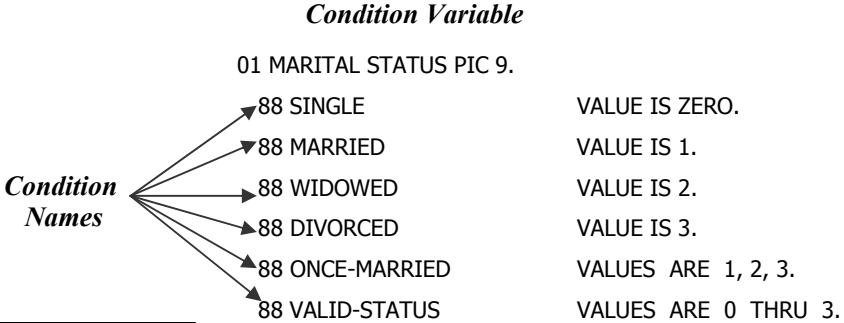
Note: IF NUMBER-1 OR NUMBER-2 = 0 is invalid.

Condition names

A condition name is a user-defined word that gives a name to a specific value that an identifier can assume. It is essentially a Boolean variable and is always associated with date names called condition variables. It is coded with level number 88 and has only a VALUE clause associated with it. Since it is not the name of an identifier, it will not possess PIC clause.

Format: 88 condition name {VALUE IS , VALUES ARE} literal-1 [{THROUGH, THRU} literal-2].

Example:



PROCEDURE DIVISION statements.

```
IF SINGLE SUBTRACT 125 FROM DEDUCTIONS.
IF ONCE-MARRIED ADD 300 TO SPECIAL-PAY.
IF NOT VALID-STATUS PERFORM 900-ERROR-PARA.
```

3.4.5. EVALUATE . . . WHEN . . .

The EVALUATE . . . WHEN . . . statement provides a shorthand notation for a series of nested IF statements. It can evaluate multiple conditions. That is, the IF statements can be made up of compound conditions.

Format:

```
EVALUATE subject-1  [ALSO subject-2]. . .
        {{WHEN object-1  [ALSO object-2]. . . }. . . } imperative-statement-1 } . . .
        [WHEN OTHER imperative-statement-2] [END-EVALUATE]
```

Here subject = {identifier, expression, TRUE, FALSE} and object = {condition, TRUE, FALSE}.

Example:

```
EVALUATE TRUE
        WHEN PERCENT  >= 80  MOVE  'A'  TO  GRADE
        WHEN PERCENT  >= 70  MOVE  'B'  TO  GRADE
        WHEN PERCENT  >= 60  MOVE  'C'  TO  GRADE
        WHEN PERCENT  >= 50  MOVE  'D'  TO  GRADE
        WHEN OTHER  MOVE 'E'  TO  GRADE
END-EVALUATE.
```

3.4.6. PERFORM verb

The PERFORM statement transfers control explicitly to one or more procedures and implicitly returns control to the next executable statement after execution of the specified procedure(s) is completed. The PERFORM statement can be of one of the following two types.

1. An out-of-line PERFORM statement. In this case a Procedure-name is specified.
2. An in-line PERFORM statement. In this case the Procedure-name is omitted. An in-line PERFORM must be delimited by the END-PERFORM phrase.

There are four formats of PERFORM statements. They are:

1. Basic PERFORM
2. TIMES phrase PERFORM
3. UNTIL phrase PERFORM

4. VARYING phrase PERFORM

Basic PERFORM

The procedure(s) referenced in the basic PERFORM statement are executed once and the control passes to the next executable statement following the PERFORM statement. It is important to note that a PERFORM statement must not cause itself to be executed. Such a recursive PERFORM statement can cause unpredictable results.

Format:

PERFORM procedure-name-1 [{THRU, THROUGH} procedure-name-2].

Examples:

1. PERFORM B1000-READ-PARA.
2. PERFORM C1000-BEGIN-CALCULATION-PARA THRU C5000-END-CALCULATION-PARA.
   In this case the execution starts with the first statement in C1000-BEGIN-CALCULATION-PARA and ends with the last statement of C5000-END-CALCULATION-PARA.

TIMES phrase PERFORM

The procedure(s) referred to in the TIMES phrase PERFORM statement are executed the number of times specified by the value by the identifier or integer.

Format:

PERFORM procedure-name-1 [{THRU, THROUGH} procedure-name-2] {identifier, integer} TIMES.

Examples:

1. PERFORM A1000-READ-PARA 3 TIMES.
2. PERFORM A5000-WRITE-PARA N TIMES.
3. PERFORM C1000-BEGIN-CALCULATE-PARA THRU C3000-END-CALCULATE-PARA 5 TIMES.

UNTIL phrase PERFORM

In this case, the procedure(s) referred are performed until the condition specified by the UNTIL phrase is true. If the TEST BEFORE phrase is specified or assumed, the condition is tested before any statements are executed (corresponds to DO WHILE) and if the TEST AFTER phrase is specified, the statements to be performed are executed at least once before the condition is tested (corresponds to DO UNTIL).

Format: PERFORM procedure-name-1 [{THRU, THROUGH} procedure-name-2] UNTIL condition

[WITH TEST {AFTER, BEFORE}].

Examples:

1. PERFORM A1000-READ-PARA UNTIL EOF = 'Y' WITH TEST AFTER.
2. PERFORM A5000-INPUT-PARA THRU A8000-OUTPUT-PARA UNTIL EOF = 'Y' WITH TEST BEFORE.

VARYING phrase PERFORM

The VARYING phrase PERFORM is similar to the UNTIL phrase PERFORM in the sense that the specified range is executed repetitively until the condition becomes true. Further, an identifier is set to some initial value and it is incremented or decrement each time the procedure(s) is executed.

Format: PERFORM procedure-name-1 [{THRU, THROUGH} procedure-name-2]

VARYING {identifier-1, index-name-1} FROM {identifier-2, index-name-2, literal-1}

BY {identifier-3, literal-2} UNTIL condition.

The execution of VARYING phrase PERFORM can be easily understood from the following flow chart.

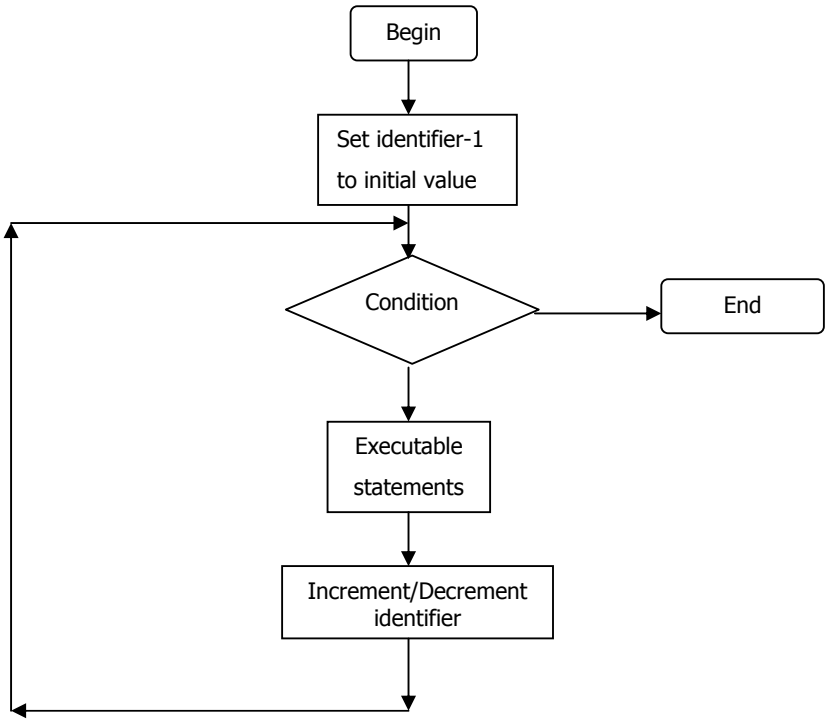


Figure 3.1 Flow chart for VARYING phrase PERFORM

Example:

```
PERFORM A1000-READ-PARA VARYING I FROM 1 BY 1 UNTIL I > 20.
```

Note: When TEST BEFORE is indicated, all specified conditions are tested before the first execution and the statements to be performed are executed, if at all, only when all specified tests fail. When TEST AFTER is indicated, the statements to be performed are executed at least once, before any condition is tested. If neither the TEST BEFORE nor the TEST AFTER phrase is specified, the TEST BEFORE phrase is assumed.

3.4.7. EXIT verb

The EXIT statement is normally used to return from a paragraph to a calling paragraph. It provides a common end point for a series of procedures.

Note: Similar to the GO TO verb, the EXIT verb should be used only under inevitable circumstances.

3.4.8. CONTINUE verb

The CONTINUE statement allows you to specify a no operation statement. CONTINUE indicates that no executable instruction is present.

3.4.9. STOP RUN

The STOP RUN statement terminates the execution of the object program and transfers the control to MVS. If a STOP RUN statement appears in a sequence of imperative statements within a sentence, it must be the last or the only statement in the sequence. The STOP RUN statement does not have to be the last statement in a sequence, but the statements following the STOP RUN will not be executed. In COBOL-85, the STOP RUN statement closes all the files that were opened/processed.

3.5. Sample Programs

3.5.1. Program to find the smallest of 5 numbers

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SMALL.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 NUM1 PIC 9.
01 NUM2 PIC 9.
01 NUM3 PIC 9.
01 NUM4 PIC 9.
01 NUM5 PIC 9.
01 SML  PIC 9 VALUE 9.
PROCEDURE DIVISION.
A0000-MAIN-PARA.
      PERFORM A1000-PROCESS-PARA
      PERFORM A2000-DISPLAY-PARA
      STOP RUN.
A1000-PROCESS-PARA.
      ACCEPT NUM1
      IF NUM1 < SML THEN MOVE NUM1 TO SML
      ACCEPT NUM2
      IF NUM2 < SML THEN MOVE NUM2 TO SML
      ACCEPT NUM3
      IF NUM3 < SML THEN MOVE NUM3 TO SML
      ACCEPT NUM4
      IF NUM4 < SML THEN MOVE NUM4 TO SML
      ACCEPT NUM5
      IF NUM5 < SML THEN MOVE NUM5 TO SML
      .
A2000-DISPLAY-PARA.
      DISPLAY "THE SMALLEST OF " NUM1 ", " NUM2
              ", " NUM3 ", " NUM4 " AND " NUM5 " IS "
              SML.
```

3.5.2. Program to find the Total, Percentage and Grade for a student

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. MARKS.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01 SUBJECT-1 PIC 99.
       01 SUBJECT-2 PIC 99.
       01 SUBJECT-3 PIC 99.
       01 TOTAL     PIC 999.
       01 PERCENT   PIC 99V99.
       01 GRADE     PIC A.
       01 O-PERCENT PIC 99.99.
       PROCEDURE DIVISION.
       A0000-MAIN-PARA.
           PERFORM A1000-ACCEPT-PARA
           PERFORM A2000-COMPUTE-PARA
           PERFORM A3000-DISPLAY-PARA
           STOP RUN
           .
       A1000-ACCEPT-PARA.
           ACCEPT SUBJECT-1
           ACCEPT SUBJECT-2
           ACCEPT SUBJECT-3
           .
       A2000-COMPUTE-PARA.
           COMPUTE TOTAL = SUBJECT-1 + SUBJECT-2 + SUBJECT-3
           COMPUTE PERCENT = TOTAL / 3
           EVALUATE TRUE
               WHEN PERCENT >= 80 MOVE 'A' TO GRADE
               WHEN PERCENT >= 70 MOVE 'B' TO GRADE
               WHEN PERCENT >= 60 MOVE 'C' TO GRADE
               WHEN PERCENT >= 50 MOVE 'D' TO GRADE
               WHEN OTHER         MOVE 'E' TO GRADE
           END-EVALUATE
           .
       A3000-DISPLAY-PARA.
           DISPLAY SUBJECT-1
           DISPLAY SUBJECT-2
           DISPLAY SUBJECT-3
           DISPLAY TOTAL
           MOVE PERCENT TO O-PERCENT
           DISPLAY O-PERCENT
           DISPLAY GRADE
           .
```

4 File Handling

OVERVIEW

1. Introduction to File processing
2. File operations.
3. File ORGANIZATION and ACCESS methods.
4. FILE CONTROL paragraph and FD entries.
5. Sequential file processing.
6. Index sequential file processing.
7. Handling I/O errors
8. Relative file processing and Hashing function.

4.1. Introduction

A file is a collection of data related to a set of entities and typically exists on a magnetic tape or a disk. The data contained in a file is logically organised into an ordered set of data items and is known as a record. The individual data items in a record are called its fields. For example, a file may contain data related to the employees of Infosys. The data pertaining to any individual corresponds to a record. The fields may be employee number, employee name, employee's mail-id, employee's age, etc., The number of characters in any field is known as the field size and the cumulative size of all the fields in a record is known as the record size. It is important to note that there exists some restriction on record size. For example, the lower and upper limit for record size may be 2 bytes and 4096 bytes respectively.

The records present in a file may be of fixed length or variable length. In most applications, fixed length records are used. However in certain cases it is essential that records be of different lengths. This happens when some of the fields are irrelevant to a set of records. In case of variable length records, usually the first four bytes are reserved for record length.

A record as defined above is often referred to as a logical record. While processing large files stored on disk or tapes, it is inefficient to read or write single record at a time. Instead, the usual practice is to group a number of consecutive records to form what is known as a physical record or a block. For instance a block may contain 4 logical records. This means the first 4 records will form the first block, the next 4 records will form the second block and so on. The number of records in a block is termed as the blocking factor. There are two advantages of blocking logical records into a physical record. Firstly, it results in saving the I/O time required for processing a file (How?) and secondly it results in saving the storage space for a file (How?).

MVS uses the programmer's description of the record to set aside sufficient memory to hold one record at a time. The memory allocated for storing a record is termed as record buffer. It is the only link between the program and the physical file. The record buffers are created and destroyed when the corresponding files are opened and closed in the programs using them. MVS facilitates overlapping of I/O operations with CPU operations. To take advantage of this feature, we may specify the number of buffers that can be used by the program. From experience, it is

found that specifying two buffers is very effective. In this case while the program reads record into one buffer, the CPU processes the record already read into the other buffer.

4.2. File operations

There are three basic types of file operations namely

1. Create: Refers to producing a brand new file and writing one or more logical records into it.
2. Retrieve: Refers to reading the logical records from a file.
3. Update: Refers to maintenance of records in a file to ensure that it is up to date. There are three types of updating operations. They are Record deletion, Record insertion and Record modification.

4.3. File ORGANIZATION and ACCESS MODE

The term file ORGANIZATION refers to the way in which the logical records are organized within the storage space allocated to a file. The term ACCESS MODE refers to the way in which the records in the file will be accessed (sequential or random). ANSI COBOL provides three standard file ORGANIZATIONs viz. Sequential, Index sequential and Relative. Corresponding to each of these three ORGANIZATION, there exist one or more ACCESS MODEs.

4.3.1. SEQUENTIAL ORGANIZATION

There are two categories in Sequential ORGANIZATION viz. Entry sequential and Line sequential. In case of Entry sequential ORGANIZATION, the records are stored in the file in the same order in which they are entered. Here, the records can be accessed only sequentially. i.e., to process any record, one has to read all its preceding records. Further, records cannot be inserted or deleted. Records can only be added to the end of the file. However, a record can be overwritten if the lengths of the old record and the new record matches.

In case of Line sequential ORGANIZATION, each record contains a sequence of characters ending with a record terminator. The terminator is not counted in the length of the record. When records are written to a line sequential file, the trailing blanks (if any) are removed and while reading the record from a line sequential file, characters are read one at a time into the record buffer until the first record terminator is encountered. It is important to note that Line sequential ORGANIZATION is available only on workstations.

Though sequential files are the most storage efficient and the simplest to handle, they are highly inflexible as they do not facilitate insertion and deletion of records. For this reason, sequential files are not normally used for permanent storage but rather as scratch files (Files that will be used once or twice and then destroyed).

4.3.2. INDEX SEQUENTIAL ORGANIZATION

The index sequential ORGANIZATION is the most sophisticated and widely used among the three file ORGANIZATIONs. It facilitates both sequential and random access of records. An index sequential file is conceptually made up of two files, a data file and an index file. Each record has a key value associated with it. Though the records are stored in the order in which they are entered, a sorted index is maintained which relates the key value to the position of the record in the file and hence provides a way to access the records both sequentially and randomly.

There are several methods that are used for storing the index. One method is the sparse index. In this method, data within a particular range of key values is stored together. In the latest indexed file implementations, a dense index is maintained. Here, the key value of each record is stored in the index along with the address in the file where the data is to be found. This is normally maintained as a B-tree to aid searching and changes.
In index sequential ORGANIZATION, we can even specify alternate indexes. For example, suppose a file contains data pertaining to the employees of Infosys, it may be necessary to access data by employee number as well as mail-id. Hence mail-id can be used as an alternate index. It is important to note that the number of indexes improves performance on reads but affects performance on writes. This is because all indexes have to be updated on each write operation.

4.3.3. RELATIVE ORGANIZATION

In relative ORGANIZATION, a file is thought of as a string of record areas, each of which contains a single record. Each record area is identified by a relative record number, the access method stores and retrieves a record, based on its relative record number. For example, the 1st record buffer is addressed by relative record number 1, the 2nd is addressed by relative record number 2 and so on. The physical sequence in which the records were placed in the file has no bearing on the record area in which they are stored and thus on each record's relative record number. Similar to Index sequential ORGANIZATION here also, the records can be accessed both sequentially and randomly.

A relative file provides the fastest access to records but has some disadvantages. Even if some of the intermediate records are missing, they occupy space. Hence it is suitable for data which can be converted to some unique record number through some transformation. This transformation must result in high degree of packing for the file to be completely filled. Also, problems arise if the transformation is not unique. Another disadvantage of this kind of organisation is that for some hashing function to be effective, some idea of the average number of records that will be present in the file is required. This information may not be available.

4.4. Making entries for a file in a program

There are two types of entries that are required in a program for any file. They are file description entries and record description entries. The file description entries specify the physical aspects of the data such as the size relationship between physical and logical records, the size and name(s) of the logical record(s), labeling information, etc., These entries are made in the ENVIRONMENT DIVISION. The record description entries describe the logical records in the file, including the category and format of data within each field of the logical record, different values the data might be assigned, etc,. These entries are made in the DATA DIVISION.

4.4.1. File description entries for a sequential file

The file description entries are made in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION using the SELECT . . . ASSIGN . . . clause. Its format is given below.

SELECT logical-file-name ASSIGN TO physical-file-name

[; RESERVE integer {AREA, AREAS}]

[; ORGANIZATION IS SEQUENTIAL]

[; ACCESS MODE IS SEQUENTIAL]

[; FILE STATUS IS data-name]

Corresponding to every file there must be a SELECT . . . ASSIGN . . . clause. The purpose of this is to establish a relationship between the logical file name (internal to COBOL) used in the program and the physical file name (external file name) used to store the file on DASD. After the relationship between physical and logical records has been established, only logical records are made available to the programmer. For this reason, when we say record, we mean the logical record and not the physical record.

RESERVE clause

The RESERVE clause allows the user to specify the number of record buffers to be allocated at run-time for the files. Thus RESERVE 2 AREAS mean that two record buffers are to be allocated. If the RESERVE clause is omitted, the number of buffers at run time is taken from the DD statement when running under MVS. If none is specified, the system default is taken.

ORGANIZATION clause

The ORGANIZATION clause identifies the logical structure of the file. The logical structure is established at the time the file is created and cannot subsequently be changed. Even if you omit the ORGANIZATION clause, the compiler assumes ORGANIZATION IS SEQUENTIAL.

ACCESS MODE clause

The ACCESS MODE clause defines the manner in which the records of the file are made available for processing. If the ACCESS MODE clause is not specified, sequential access is assumed The FILE STATUS clause monitors the execution of each input-output operation for each and every file.

FILE STATUS clause

When the FILE STATUS clause is specified, the system moves a value into the two byte alphanumeric data-name defined in the WORKING-STORAGE SECTION after each input-output operation that explicitly or implicitly refers to this file. The value indicates the status of execution of the statement. It is a very good practice to code the FILE STATUS clause for every file.

4.4.2. Record description entries for a sequential file

The record description entries are made in the FILE SECTION of the DATA DIVISION under the FD paragraph.

Format 1: (Fixed length records)

FD filename
[; RECORD CONTAINS integer-1 CHARACTERS]
[; BLOCK CONTAINS integer-2 {RECORDS, CHARACTERS}]
[; DATA {RECORD IS, RECORDS ARE} data-name-1 [, data-name-2] . . .]

Format 2: (Variable length records)

FD filename

[; RECORD CONTAINS integer-1 to integer-2 CHARACTERS]

[; BLOCK CONTAINS integer-3 to integer-4 {RECORDS, CHARACTERS}]

[; DATA {RECORD IS, RECORDS ARE} data-name-1 [, data-name-2] . . .]

RECORD CONTAINS clause

The RECORD CONTAINS clause specifies the size of the logical records. This clause cannot be used for LINE SEQUENTIAL files.

BLOCK CONTAINS clause

The BLOCK CONTAINS clause specifies the size of the physical records. If the records in the file are not blocked, the BLOCK CONTAINS clause can be omitted. When it is omitted, the compiler assumes that records are not blocked. Even if each physical record contains only one complete logical record, coding BLOCK CONTAINS 1 RECORD would result in fixed blocked records. The BLOCK CONTAINS clause can be omitted when the associated FILE CONTROL entry specifies a VSAM file, since the concept of blocking has no meaning for VSAM files.

DATA RECORD clause

The DATA RECORD clause specifies the record names defined for the file. It is used only to provide better documentation.

4.5. PROCEDURE DIVISION statements for sequential files

4.5.1. OPEN statement

The OPEN statement initiates the processing of files. The successful execution of an OPEN statement determines the availability of the file for processing. A file is available if it is physically present and is recognised by the input-output control system (IOCS is a sub system of MVS that supports file processing). The successful execution of the OPEN statement makes the associated record area available to the program; it does not obtain or release the first data record. If the FILE STATUS clause is specified in the FILE-CONTROL entry, the associated status key is updated when the OPEN statement is executed.

Format:

OPEN {INPUT, OUTPUT, EXTEND, I-O} file-name-1 [, file-name-2] . . .

[{INPUT, OUTPUT, EXTEND, I-O} file-name-3 [, file-name-4] . . .] . . .

A sequential file can be opened in one of the following four modes. INPUT, OUTPUT, EXTEND and I-O.

- A file can be opened in the INPUT mode only if it is already existing. Such a file becomes an input file from which records can be read sequentially.

- When a file is to be created for the first time, it must be opened in the OUTPUT mode. Note that, opening an existing file in the OUTPUT will result in the loss of all the data.
- The EXTEND mode also opens a file for writing, but the file pointer is positioned after the end of the last record. Thus any records written will get appended to the file.
- A file is opened in the I-O mode when it needs to be updated. This mode provides both reading and re-writing of records.

4.5.2. CLOSE statement

The CLOSE statement terminates the processing of the file. As a result of the execution of the CLOSE statement, the IOCS performs the end of file processing. The record buffer created for the corresponding file gets destroyed and thus the link between the program and the file is lost. The CLOSE statement is optional from COBOL-85. The STOP RUN statement automatically closes all the files that were opened by the program. The CLOSE statement can be used with the LOCK option, which prevents the file to be opened again in the same program.

Format: CLOSE file-name-1 [WITH LOCK] [, file-name-2 [WITH LOCK]] . . .

Note: Though it is possible to OPEN and CLOSE more than one file at a time, programmers are advised not to do so. This is because if opening or closing of any particular file is unsuccessful, then it is impossible to identify that file using the FILE STATUS as it applies to all the files.

4.5.3. READ statement

If a file is opened in the INPUT or the I-O mode, then we can use the READ statement to make the next logical record from a file available to the object program. Though the primary function of the READ statement is to fetch records from a file and place the file pointer at an appropriate position after READ, it performs certain checks to ensure proper execution of the program. It checks the length of the input record to ensure that is corresponds to the length specified in the RECORD CONTAINS clause. It also uses the BLOCK CONTAINS clause, if specified, to perform a check on the blocking factor. The READ statement can be used with INTO option for getting a copy of the logical record into a WORKING-STORAGE variable. The READ statement can be also used with an AT END and NOT AT END clauses. The AT END determines whether there is any more input and the programmer can decide what to do based on the answer. The NOT AT END can be used to accomplish specific tasks when an AT END has not been reached.

Format:

READ file-name INTO data-name
[AT END imperative statements]
[NOT AT END imperative statements]
[END-READ]

4.5.4. WRITE statement

If a file is opened in the OUTPUT or the EXTEND mode, then we can use the WRITE statement to transmit data to the physical file. Once a record has been written to a file, it is no longer available in the record buffer. It is important to note that although we read files, we write records. The WRITE statement can be used with FROM option for writing data directly from a WORKING-STORAGE variable to the required file, otherwise the data must be moved to the record buffer and then written to the file. Further the WRITE statement can be used with ADVANCING option to write records on a fresh line/page.

Format:

WRITE record-name [FROM data-name]
{AFTER, BEFORE} ADVANCING integer {LINE, LINES, PAGE}
[END-WRITE]

4.5.5. REWRITE statement

If a file is opened in the I-O mode and a record has been read successfully into the record buffer, then we can use the REWRITE statement to update an existing record. Similar to the WRITE statement, the REWRITE statement can be used with FROM option for writing data directly from a WORKING-STORAGE variable to the required file.

Format:

REWRITE record-name [FROM data-name]
{AFTER, BEFORE} ADVANCING integer {LINE, LINES, PAGE}
[END-REWRITE]

4.6. Sample Programs

4.6.1. Program to create a sequential file

```
       IDENTIFICATION DIVISION.
        PROGRAM-ID. SEQFILE.
        ENVIRONMENT DIVISION.
        INPUT-OUTPUT SECTION.
        FILE-CONTROL.
            SELECT STUDFILE ASSIGN TO STUDENT.
        DATA DIVISION.
        FILE SECTION.
        FD STUDFILE.
        01 FS-STUD-REC.
            05 FS-REGNO PIC X(5).
            05 FS-NAME  PIC A(15).
            05 FS-AGE   PIC 9(2).
            05 FILLER   PIC X(58).
        WORKING-STORAGE SECTION.
        01 N PIC 9(2) VALUE 1.
        01 WS-STUD-REC.
            05 WS-REGNO PIC X(5).
            05 WS-NAME  PIC A(15).
            05 WS-AGE   PIC 9(2).
            05 FILLER   PIC X(58).
        PROCEDURE DIVISION.
        A0000-MAIN-PARA.
            ACCEPT N
            OPEN OUTPUT STUDFILE
            PERFORM A1000-CREATE-PARA N TIMES
            CLOSE STUDFILE
            STOP RUN
            .
        A1000-CREATE-PARA.
            ACCEPT WS-REGNO
            ACCEPT WS-NAME
            ACCEPT WS-AGE
            MOVE WS-STUD-REC TO FS-STUD-REC
            WRITE FS-STUD-REC
            .
```

4.6.2. Program to update a sequential file

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. UPDPGM.
       ENVIRONMENT DIVISION.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT STUDFILE ASSIGN TO STUDENT.
       DATA DIVISION.
       FILE SECTION.
       FD STUDFILE.
       01 FS-STUD-REC.
           05 FS-REGNO PIC X(5).
           05 FS-NAME  PIC A(15).
           05 FS-AGE   PIC 9(2).
           05 FILLER   PIC X(58).
       WORKING-STORAGE SECTION.
       01  WS-STUD-REC.
           05 WS-REGNO PIC X(5).
           05 WS-NAME  PIC A(15).
           05 WS-AGE   PIC 9(2).
           05 FILLER   PIC X(58).
       01  WS-U-REGNO  PIC X(5).
       01  WS-U-NAME   PIC A(15).
       01  WS-U-AGE    PIC 9(2).
       01  RECORD-FOUND PIC X VALUE 'N'.
       01  EOF          PIC X VALUE 'N'.
       PROCEDURE DIVISION.
       A0000-MAIN-PARA.
           OPEN I-O STUDFILE
           PERFORM A1000-INPUT-PARA.
           PERFORM A2000-PROCESS-PARA UNTIL RECORD-FOUND = 'Y'
           CLOSE STUDFILE
           STOP RUN
           .
       A1000-INPUT-PARA.
           ACCEPT WS-U-REGNO
           ACCEPT WS-U-NAME
           ACCEPT WS-U-AGE
           .
       A2000-PROCESS-PARA.
           PERFORM A3000-READ-PARA
           IF EOF = 'N'
           IF WS-U-REGNO = WS-REGNO MOVE 'Y' TO RECORD-FOUND
              PERFORM A4000-UPDATE-PARA
              END-IF
           END-IF
           .
       A3000-READ-PARA.
           READ STUDFILE INTO WS-STUD-REC AT END MOVE 'Y' TO EOF
           .
       A4000-UPDATE-PARA.
           MOVE WS-U-REGNO TO WS-REGNO
           MOVE WS-U-NAME TO WS-NAME
           MOVE WS-U-AGE TO WS-AGE
           REWRITE FS-STUD-REC FROM WS-STUD-REC
           .
```

4.7. File description entries for an Index sequential file

SELECT logical-file-name ASSIGN TO physical-file-name

[; RESERVE integer {AREA, AREAS}]

[; ORGANIZATION IS INDEXED]

[; ACCESS MODE IS {SEQUENTIAL, RANDOM, DYNAMIC}]

[; RECORD KEY IS data-name-1]

[; ALTERNATE RECORD KEY is data-name-2 [WITH DUPLICATES]] . . .

[; FILE STATUS IS data-name-2]

Here the RECORD KEY clause specifies the index based on which the file is sequenced. The data-name-1 must be an alphanumeric field within the record description for the file. In case there are multiple record descriptions, the key field from any of the descriptions can be used.

Although an index sequential file is sorted and maintained on the primary key, the records can also be accessed using the ALTERNATE KEY. Further, the ALTERNATE KEY data item may also find duplicate entries for records. To incorporate this, specify WITH DUPLICATES option. Please note that this option has not yet been incorporated in some compilers.

4.8. PROCEDURE DIVISION statements for index sequential files

4.8.1. READ statement

The format of READ statement for an index sequential file is almost similar to that of a sequential file.

Format:

READ file-name [NEXT RECORD]

INTO data-name

[KEY is data-name]

[INVALID KEY imperative statements]

[NOT INVALID KEY imperative statements]

[END-READ]

Here, the data-name in the KEY phrase must be either the primary key or one of the alternate keys. The option NEXT RECORD is specified when an index sequential file is being read sequentially. INVALID KEY condition arises when the specified key is not found in the file.

4.8.2. WRITE statement

If a file is opened in the OUTPUT mode, then the WRITE statement releases the records to the file in the ascending order of the record key values regardless of the access mode. If the records are to be released in random order, then the file must have been opened in the I-O mode and the access mode must be specified as RANDOM or DYNAMIC only. The INVALID KEY condition arises for an index sequential file in the following situations.

- An attempt is made to write beyond the externally defined boundary of the file.
- ACCESS mode SEQUENTIAL is specified and the file is opened OUTPUT and the value of the primary key is not greater than that of the previous record.
- The file is opened in the OUTPUT or I-O modes and the value of the primary record key is equal to that of an already existing record.

Format:

WRITE record-name [FROM data-name]
{INVALID KEY imperative statements}
{NOT INALID KEY imperative statements}
[END-WRITE]

4.8.3. REWRITE statement

As in case of sequential file, the REWRITE statement requires that the file must be opened in the I-O mode and if the SEQUENTIAL ACCESS mode is specified, the value of the RECORD KEY being replaced must be equal to that of the record last read from the file. The INVALID KEY condition arises in the following situations.

- The access mode is sequential and the value contained in the RECORD KEY of the record to be replaced is not equal to the value of the RECORD KEY data item of the last-retrieved record from the file.
- The value contained in the RECORD KEY is not equal to that of any record in the file.
- The value of an ALTERNATE RECORD KEY data item for which DUPLICATES is not specified is equal to that of a record already in the file.

Format:

REWRITE record-name [FROM data-name]
{INVALID KEY imperative statements}
{NOT INVALID KEY imperative statements}
[END-REWRITE]

4.8.4. DELETE statement

To delete a record from an index sequential file, the file must be opened in the I-O mode. If the access mode is sequential, then the INVALID KEY phrase should not be specified. Instead, the last I/O statement executed on the file must be a successful READ statement for the record specified. IF the access mode is RANDOM or DYNAMIC, then the record to be deleted is determined by the value of the RECORD KEY. In this case the INVALID KEY phrase should be specified. The INVALID KEY condition arises if the specified is not found in the file.

Format:

DELETE file-name RECORD
{INVALID KEY imperative statements}
{NOT INVALID KEY imperative statements}
[END-REWRITE]

4.8.5. START statement

The START statement provides a means of positioning the file pointer at a specific location within an index sequential file for subsequent sequential record retrieval. The access mode must be SEQUENTIAL or DYNAMIC and the file must be opened in the INPUT or I-O modes. Further, if the KEY phrase is specified, the file pointer is positioned at the logical record in the file whose key field satisfies the comparison and if it is omitted, then KEY IS EQUAL (to the RECORD KEY) is implied. If the comparison is not satisfied by any record in the file, an invalid key condition exists; the position of the file position indicator is undefined and (if specified) the INVALID KEY imperative-statement is executed.

Format:

START file-name
[KEY is {=, <, >} data-name]
[INVALID KEY imperative statements]
[NOT INVALID KEY imperative statements]
[END-READ]

4.9. Sample Programs

4.9.1. Program to add a record to an Index sequential file.

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. ADDVSAM.
       ENVIRONMENT DIVISION.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT STUDFILE ASSIGN TO STUDENT
           ORGANIZATION IS INDEXED
           ACCESS IS RANDOM
           RECORD KEY IS REGNO.
       DATA DIVISION.
       FILE SECTION.
       FD  STUDFILE.
       01  FS-STUD-REC.
           05 FS-REGNO PIC X(5).
           05 FS-NAME  PIC A(15).
           05 FS-AGE   PIC 9(2).
           05 FILLER   PIC X(58).
       WORKING-STORAGE SECTION.
       01 EOF            PIC X VALUE 'N'.
       01  WS-STUD-REC.
           05 WS-REGNO PIC X(5).
           05 WS-NAME  PIC A(15).
           05 WS-AGE   PIC 9(2).
           05 FILLER   PIC X(58).
       PROCEDURE DIVISION.
       A1000-MAIN-PARA.
           PERFORM A2000-OPEN-PARA
           PERFORM A3000-INPUT-PARA
           PERFORM A4000-INSERT-PARA
           PERFORM A5000-CLOSE-PARA
           STOP RUN
           .
       A2000-OPEN-PARA.
           OPEN I-O STUDFILE
           .
       A3000-INPUT-PARA.
           ACCEPT WS-REGNO
           ACCEPT WS-NAME
           ACCEPT WS-AGE
           .
       A4000-INSERT-PARA.
           WRITE FS-STUD-REC FROM WS-STUD-REC
               INVALID KEY DISPLAY "INVALID RECORD", STUD-REC
               NOT INVALID KEY DISPLAY "RECORD INSERTED"
           END-WRITE
           .
       A5000-CLOSE-PARA.
           CLOSE STUDFILE
           .
```

4.9.2. Program to delete a record from an index sequential file.

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. DELVSAM.
       ENVIRONMENT DIVISION.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT STUDFILE ASSIGN TO STUDENT
           ORGANIZATION IS INDEXED
           ACCESS MODE IS SEQUENTIAL
           RECORD KEY IS REGNO.
       DATA DIVISION.
       FILE SECTION.
       FD STUDFILE.
       01  STUD-REC.
           05 REGNO     PIC X(5).
           05 NAME      PIC A(15).
           05 AGE       PIC 9(2).
           05 FILLER    PIC X(58).
       WORKING-STORAGE SECTION.
       01  WS-STUD-REC  PIC X(22).
       01  WS-REGNO     PIC X(5).
       01  WS-D-REGNO   PIC X(5).
       01  EOF          PIC X VALUE 'N'.
       01  RECORD-FOUND PIC X VALUE 'N'.
       PROCEDURE DIVISION.
       A1000-MAIN-PARA.
           OPEN I-O STUDFILE
           PERFORM A2000-INPUT-PARA
           PERFORM A3000-PROCESS-PARA UNTIL RECORD-FOUND = 'Y'
           CLOSE STUDFILE
           STOP RUN
           .
       A2000-INPUT-PARA.
           ACCEPT WS-D-REGNO
           .
       A3000-PROCESS-PARA.
           PERFORM A4000-READ-PARA
           IF EOF = 'N'
              IF WS-D-REGNO = WS-REGNO MOVE 'Y' TO RECORD-FOUND
              PERFORM A5000-DELETE-PARA
              END-IF
           END-IF
           .
       A4000-READ-PARA.
           READ STUDFILE AT END MOVE 'N' TO EOF
           END-READ
           MOVE REGNO TO WS-REGNO
           .
       A5000-DELETE-PARA.
           DELETE STUDFILE RECORD
           .
```

4.10. File description entries for a Relative file

SELECT logical-file-name ASSIGN TO physical-file-name

[; RESERVE integer {AREA, AREAS}]

[; ORGANIZATION IS RELATIVE]

[; ACCESS MODE IS {SEQUENTIAL [, RELATIVE KEY IS data-name-1]

{RANDOM, DYNAMIC}, RELATIVE KEY is data-name-1}]

[; FILE STATUS IS data-name-2]

Here, the phrase RELATIVE KEY must be specified when the access mode is RANDOM or DYNAMIC. The data-name-1 is called the relative key data item and it indicates the field that contains the relative record number. The programmer must place an appropriate value in the relative key data item while accessing a record randomly. Further data-name-1 must be an unsigned integer, but it should not be a part of the record description.

4.11. PROCEDURE DIVISION statements for Relative files

4.11.1. READ statement

Format 1:

READ file-name RECORD [INTO identifier]

[; AT END imperative statements]

[END-READ]

This format is applicable to SEQUENTIAL ACCESS MODE. If the RELATIVE KEY phrase is also specified with the ACCESS MODE SEQUENTIAL clause, then upon the successful completion of the READ statement, the relative record number of the accessed record is placed in the relative key data item.

Format 2:

READ file-name RECORD [INTO identifier]

[; INVALID KEY imperative statements]

This format is applicable when ACCESS MODE is either RANDOM or DYNAMIC. In this case the record to be read is identified from the contents of the RELATIVE KEY data item. The INVALID KEY case arises when the READ is unsuccessful, i.e., when an attempt is made to read a record from either (i) an empty record position of the file or (ii) outside the externally defined boundaries of the file.

Format 3:

READ file-name [NEXT] RECORD [INTO identifier]

[; INVALID KEY imperative statements]

This format is applicable when the ACCESS MODE is DYNAMIC and the records are to read sequentially. Here the NEXT RECORD is identified according to the following rules.

(i) If the READ NEXT statement is the first statement to be executed after the OPEN statement, then the NEXT RECORD is the first record itself.

(ii) If the READ NEXT statement follows a successful execution of another READ NEXT on the same file, the NEXT RECORD is the record following the one previously read record.

4.11.2. WRITE statement

If a file is opened in the OUTPUT or, I-O mode, then the WRITE statement releases the records to the file and the record area associated with record name no longer contains the record. When the records are to be released in random order, then the statement releases the record to that relative record position on the file which is indicated by the record number in the relative key data item. But, during sequential access, the records are released to the file in the sequential record positions. The imperative statement of the INVALID KEY phrase is executed in the following situations:

- An attempt is made to write beyond the externally defined boundary of the file.
- An attempt is made to write in the record position which already contains a valid record

Format:

WRITE record-name [FROM identifier]

[; INVALID KEY imperative-statement]

[; NOT INALID KEY imperative-statement]

[END-WRITE]

4.11.3. REWRITE statement

When a file is opened in the OUTPUT or, I-O mode, the REWRITE replaces an existing record by the contents of the record specified in the record name. The FORM phrase has the same meaning as in the case of the WRITE statement. When the access mode is RANDOM or DYNAMIC, the record to be replaced is identified by the contents of the relative key data item. But, during sequential access, prior to the REWRITE statement, a READ statement on the file must be successfully executed, and there should not be any other I-O statement on the file in between. The imperative after the INVALID KEY is executed when an attempt is made to replace a record position which is empty. Upon successful execution of the REWRITE statement, the record area doesn't contain the released data.

Format:

REWRITE record-name [FROM identifier]

[; INVALID KEY imperative-statement]

[; NOT INVALID KEY imperative statements]

[END-REWRITE]

4.11.4. DELETE statement

The record position whose contents are to be deleted is identified in the same manner as in the case of REWRITE statement. The INVALID KEY condition arises when an attempt is made to delete the record of an empty record position. In this case the statement after the INVALID KEY is executed while the file must be open in I-O mode. The execution of a DELETE statement doesn't affect the contents of the record area as it deletes the data contained in the specified record position of a relative file.

Format:

DELETE file-name RECORD

[; INVALID KEY imperative-statement]

[; NOT INVALID KEY imperative-statement]

[END-REWRITE]

4.11.5. START statement

The START statement enables the programmer to position the relative file at some specified point so that subsequent sequential operations on the file can start from this point instead of the beginning. They KEY IS phrase indicates how the file is to be positioned. The data-name in this phrase must be the data-name in the RELATIVE KEY phrase of the SELECT . . . ASSIGN . . . clause. When the EQUAL TO or NOT LESS THAN condition is specified, the file is positioned at the point indicated by the relative key-data item. When the GREATER THAN condition is specified, the file is positioned at the next relative position of the position indicated by the RELATIVE KEY data item.

Format:

START file-name

KEY IS { EQUAL TO | = | GREATER THAN | > | NOT LESS THAN | NOT < } data – name

[; INVALID KEY imperative statements]

[END-START]

4.12. Handling I/O Errors

I/O errors can be handled in two ways:

- Directly by an INVALID KEY clause in case of Index file (or VSAM file), or by checking FILE STATUS.
- Using **DECLARATIVES**.

The direct handling of an I/O error is the simpler method of the two, but a separate I/O error-handling routine offers more comprehensive functionality.

4.12.1. Error handling using INVALID KEY clause:

All I/O statements can include a clause (per se INVALID KEY) that performs a procedure if an error condition is detected as shown below.

```
...
    WRITE KSDS-VSAM-REC
        INVALID KEY
            PERFORM 250-WRITE-MASTER-ERROR
    END-WRITE.
...
250-WRITE-MASTER-ERROR.
    EVALUATE WS-FILE-STATUS
        WHEN "21"
            DISPLAY "KEY SEQUENCE ERROR: ", KSDS-VSAM-REC
        WHEN "22"
            DISPLAY "DUPLICATE PRI KEY: ", KSDS-VSAM-REC
        WHEN OTHER
            DISPLAY "I/O ERROR # ", WS-FILE-STATUS
            DISPLAY "WRITING REC ", KSDS-VSAM-REC
            CLOSE NDX-MASTER-FILE
            STOP RUN
    END-EVALUATE.
```

The above example above assumes the indexed KSDS-VSAM file has been SELECTed to include the clause FILE STATUS IS WS-FILE-STATUS, and that the corresponding data item has been declared in WORKING-STORAGE. The procedure invoked by the INVALID KEY clause displays an appropriate error message for out of sequence or duplicate keys, and processing continues. For other I/O errors (something more serious, like a hardware error), the program is aborted.

4.12.2. Error Handling using DECLARATIVES:

With the above technique, file I/O errors are handled by a procedure coded as a regular PROCEDURE DIVISION paragraph. I/O errors can also be processed by procedures coded as paragraphs in a separate SECTION of the PROCEDURE DIVISION, by use of a **DECLARATIVES** segment. This requires some changes in the organization of the PROCEDURE DIVISION by adding an additional level of hierarchy. Its format is given below.

```
    PROCEDURE DIVISION.
    DECLARATIVES.
    section-name1 SECTION.
        USE AFTER ERROR PROCEDURE
            ON { file-name | INPUT | OUTPUT | I-O | EXTEND }.
  [ error-handler-para-name(s). ]
        stmts ...
    END DECLARATIVES.

    section-name2 SECTION.
    regular-processing-para-name(s).
        stmts ...
```

The DECLARATIVES statement must be the first statement in the PROCEDURE DIVISION. SECTION names begin in area A, and must meet all standard COBOL requirements for paragraph names. In the DECLARATIVES, each SECTION must have a USE statement that determines when that section will be invoked: for any I/O errors on the stated file, or for the type of I/O error on every file. One or more (optional) paragraphs in the SECTION can be used to handle the error. More than one SECTION can be used to process errors on different files, or different type of errors.

Coding a separate SECTION for I/O error handling may not be necessary if all I/O to a file is performed in a single statement. But if I/O occurs in several different locations, using a separate SECTION can increase uniformity and modularity.

The 'END DECLARATIVES' statement is not hyphenated. Once an error-handler has been written, the INVALID KEY clause should not be used for I/O to the file specified. An error condition will automatically invoke the error handler. Control will be returned to the statement following the one that generated the error.

This is an example that performs the same function as the above procedure:

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. ERRMOD.
       ENVIRONMENT DIVISION.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT INPUT-FILE ASSIGN TO INFILE
           FILE STATUS IS WS-STATUS-INFILE.

       DATA DIVISION.
       FILE SECTION.
       FD INPUT-FILE
           RECORDING MODE F.
       01 IN-REC.
           05 CUST-NO           PIC 9(5).
           05 PART-NO           PIC X(8).
           05 QTY               PIC 9(3).

       WORKING-STORAGE SECTION.

       01 ARE-THERE-MORE-RECORDS    PIC X(3) VALUE 'YES'.
          88 NO-MORE-RECORDS        VALUE 'NO '.
       01 WS-STATUS-INFILE          PIC X(02) VALUE '00'.

       PROCEDURE DIVISION.
       DECLARATIVES.
       A000-IO-ERROR SECTION.          <-- SECTION header must follow DECLARATIVES
           USE AFTER ERROR PROCEDURE
           ON INPUT-FILE.              <-- Will invoke the following paragraph if any I/O error occurs with this file
       A100-CHK-FILE-STAT.
           IF WS-STATUS-INFILE NOT = '00'
              DISPLAY 'ERROR FROM DECLARATIVES'
              GOBACK
           END-IF.
       END DECLARATIVES.

      B000-REGULAR-PROCESS SECTION.
      100-MAIN-MODULE.

          OPEN INPUT INPUT-FILE
          OPEN INPUT INPUT-FILE       <-- Try to open already open file to trigger an I/O error.

          READ INPUT-FILE
             AT END MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
          END-READ

          PERFORM 200-READ-LOOP
             UNTIL NO-MORE-RECORDS
```

```
           CLOSE INPUT-FILE

           GOBACK.

       200-READ-LOOP.
           READ INPUT-FILE
              AT END MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
           END-READ.

****************************************
Result of the above programs:
'ERROR FROM DECLARATIVES'
```

4.13. Different File ORGANIZATIONs – a comparative study

Table 4.1 lists the advantages and disadvantages of using different file ORGANIZATIONs for quick reference.

SEQUENTIAL	INDEXED	RELATIVE
Slow when the hit rate is low.	Slowest Direct Access ORGANIZATION.	Wasteful of storage if the file is only partially populated.
Complicated to change (insert, delete, amend)	Especially slow when adding or deleting records.	Cannot recover space from deleted records.
Most storage efficient.	Not very storage efficient. Must store the Index records, the alternate Index records, the data records and the alternate data records.	Only a single, numeric key allowed.
Simple ORGANIZATION.	Can use multiple, alphanumeric keys.	Keys must map on to the range of the Relative Record numbers.
Recovers space from deleted records.	Can have duplicate alternate keys.	Fastest Direct Access ORGANIZATION.
-	Can be read sequentially on any of its keys.	Can be read sequentially.
-	Can partially recover space from deleted records.	Very little storage overhead.

Table 4.1 File ORGANIZATIONs – An analysis

4.14. Hashing functions

Hashing functions are used to convert a record key value to a relative key value. These functions normally take numeric values as input. However, alphanumeric values can be converted to an equivalent numeric value by some methods, like adding up the ASCII values of each character.

The most commonly used hashing function is the Division Method. In this method, the record key is divided by a number to arrive at a relative key value. The choice of the divisor is very important. It must be chosen such that, it provides unique key values. For this reason, a prime number greater than the number of records is chosen. The following formula illustrates this method.

Example:

R (x) = x mod m + 1 Where, R (x) is the relative key, x is the record key and m is the divisor.

Another common method is called the Mid-Square method. In this, the key is squared and a certain number of digits chosen from the middle of it. For example, if the record key 123456 is squared it gives the value 15241383936. Choosing the fourth through the seventh digits yields a relative key value of 4138.
Another hashing function is the Folding method. In this method, the record key is partitioned into a number of sets, each of which is the size of the relative key. Then, the corresponding digits in each of the sets are added to arrive at the relative key, ignoring the final carry. For example, if the relative key size is 4 and the record key is 12345678, the relative key is 6912.

In the Digit Analysis method, some digits are chosen and their order is reversed to get the relative key. For example, choosing digits 3 to 6 and reversing them in the case of 7546123 yields a relative key value 2164. In the Length Dependent method, the length and some portion of the key is used along with the division method to arrive at the relative key.

4.15. Collision Resolution

Collision is the phenomenon, which occurs when two record keys map onto the same relative key. This has to be handled somehow. There are two broad classes of techniques available. The first is the Direct Chaining method. In this method, an overflow area is maintained. If there is a collision, the record is written to the overflow area. An entry is made in the prime area to indicate that there is an overflow. The mechanism is shown in the table below.

Example of Direct chaining method:

Prime area	Relative key 1	1
	Relative key 3	2
	Relative key 5	
Overflow area	Relative key 1	
	Relative key 3	3
	Relative key 3	

Table 4.2 Direct chaining method

In the above table, there is a collision in relative key values 1 and 3. The records are written to the overflow area. In the last column is the record number in the overflow area of the next record, which has the same key value. Another method of collision resolution is by open addressing. The two most commonly used techniques are linear probing and random probing.

The linear probing technique implies that if there is a collision, the record is written to the next free slot. However, this method has two drawbacks. Because of this, there is likely to be more collisions. If the file is filled, this figure is likely to be significant. Secondly, a record cannot be physically deleted. When the record was written, it was written to the first empty slot. If after this an intermediate record is deleted, then a search would fail. The search assumes that if a slot is empty, the record was not found. Hence, only logical deletions are permitted in this technique. Another form of probing called random probing uses a random sequence to determine the next slot. The formula used is shown below.

R(x) = (R(x) + c) mod m, Where R(x) is the relative key and c and m are numbers that are relatively prime. For example for R(x) = 100, c = 4 and m = 97 the series below is produced.

Whatever technique we use for collision resolution, there will be collisions because the data distribution can't be completely studied before designing the system. Hence the relative files will be seldom used in developing application programs.

4.16. File ORGANIZATION Triplet

Here in this section we draw the summary of different file organisations – access methods – operations that can be performed on files in COBOL. Table 4.3 gives the file organisation triplet for quick reference.

			⇐ OPEN MODE ⇒			
File ORGANIZATION	**Access Mode**	**Statement**	**INPUT**	**OUTPUT**	**I-O**	**EXTEND**
SEQUENTIAL	SEQUENTIAL	READ WRITE REWRITE	X	 X	X X	 X
LINE SEQUENTIAL	SEQUENTIAL	READ WRITE	X	 X		 X
RELATIVE and INDEXED	SEQUENTIAL	READ WRITE REWRITE DELETE START	X X	 X	X X X X	
	RANDOM	READ WRITE REWRITE DELETE START	X	 X	X X X X	
	DYNAMIC	READ WRITE REWRITE DELETE START	X X	 X	X X X X X	

Table 4.3 File ORGANIZATION Triplet

5 Table Processing

OVERVIEW

1. Introduction
2. Defining a Table.
3. Accessing the elements of a Table.
4. Subscript Vs Index.
5. SEARCH and SEARCH ALL statements.

5.1. Introduction

A Table (referred to as an Array in Pascal, C, C++, etc.,) is a linear data structure that is used to store, access and process data easily and efficiently. It is a collection of homogenous data items (Data items that possess the same size and structure) that can be referred by a single name. The data items contained in a Table are called its elements and can be used in arithmetic and logical operations. The elements of a Table are stored contiguously in the memory. The elements of a Table can be either elementary items or group items and further a Table can be either of fixed length or variable length. In COBOL-85, a Table can have dimension up to 7. Though a Table is a very simple data structure, it is a very powerful tool available to the programmer. It can save many lines of code and simplify the programming logic to a considerable extent. COBOL has a wealth of logical control structures and statements that allows the programmer to exploit the complete range of Table handling capabilities.

5.2. Defining a Table

Similar to any other data item, a Table is defined in the DATA DIVISION of a COBOL program. To specify the repeated occurrence of data items with the same format, the OCCURS clause is used. The OCCURS clause specifies the maximum number of elements that can be stored in the Table. It is important to note that OCCURS clause can be used with only level numbers 02 to 49. It cannot be used with level number 01, for, it must be used to define fields and not records.

Example 1: (One-dimensional)

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  WS-DAILY-TEMPERATURE.
        05  WS-HOURLY-TEMPERATURE  PIC  9(2)V9  OCCURS  24  TIMES.
```

Example 2: (One-dimensional Table initialized using VALUE clause)

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  WS-MONTHS  VALUE  "JANFEBMARAPRJUNJULAUGSEPOCT NOVDEC".
        05 WS-MONTH OCCURS 12 TIMES PIC A(3).
```

Example 3: (Two-dimensional)

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  WS-TEMPERATURE.
        05  WS-DAYS  OCCURS  7  TIMES.
                10  WS-HOURS  OCCURS  24  TIMES.
                        15   WS-TEMP  PIC 9(2)V9.
```

Example 4: (Three-dimensional)

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  WS-TEMPERATURE.
        05 WS-MONTHS  OCCURS  12 TIMES.
                10 WS-DAYS  OCCURS  31  TIMES.
                        15 WS-HOURS  OCCURS  24  TIMES.
                                20   WS-TEMP  PIC 9(2)V9.
```

Example 5: (Group data-items)

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-BOOKS.
        05 WS-DETAILS  OCCURS  1000  TIMES.
                10 WS-AUTHOR.
                        15 WS-FIRST-NAME  PIC  A(15).
                        15 WS-MID-NAME     PIC  A(10).
                        15 WS-LAST-NAME   PIC  A(10).
                10   WS-TITLE  PIC  A(60).
```

5.3. Variable length Tables

Tables can also have a variable number of elements. This does not imply that the size of the table is dynamically adjusted depending on the number of elements. The variable size is useful to avoid out of range accesses. For example, if a table has been declared to have between 10 and 200 entries and it currently has 30 entries, an access to the 55th element is illegal.

Example:

```
01   WS-EMP-TABLE.
     05  WS-NUMBER-OF-EMPLOYEES  PIC 9(6).
     05   WS-EMP-REC OCCURS 1 TO 500 TIMES DEPENDING ON WS-NUMBER-OF-EMPLOYEES.
        10   WS-EMP-NAME  PIC A(30).
        10   WS-EMP-DEPT  PIC X(04).
```

5.4. Accessing the elements of a Table

The elements of a Table can be accessed with the data-item that is lowest in the hierarchy with either a subscript or an index. The subscript/index must be enclosed within a pair of parenthesis. If the elements are to be accessed using the INDEXED BY phrase, then it should be declared in the DATA DIVISION using the INDEXED BY clause as shown in the following examples.

Example 1: (Two-dimensional Table defined using INDEXED BY clause)

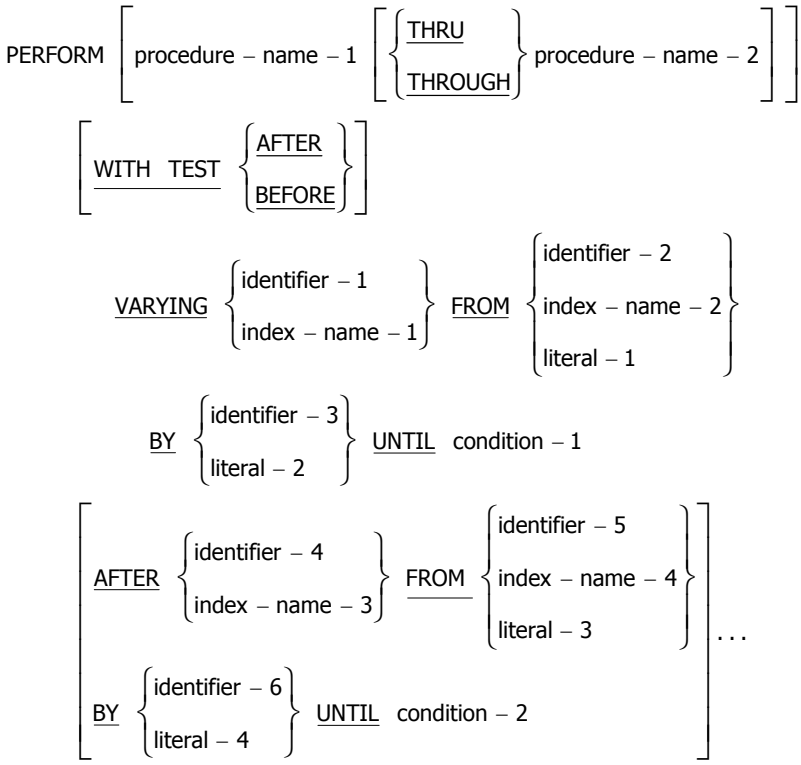
```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-TEMPERATURE.
    05 WS-DAYS  OCCURS  7  TIMES  INDEXED BY  I.
        10 WS-HOURS  OCCURS  24  TIMES  INDEXED  BY  J.
            15   WS-TEMP  PIC 9(2)V9.
```

Example 2: (Three-dimensional Table defined using INDEXED BY clause)

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-TEMPERATURE.
    05 WS-MONTHS  OCCURS  12 TIMES  INDEXED  BY  I.
        10 WS-DAYS  OCCURS  31  TIMES  INDEXED  BY  J.
            15 WS-HOURS  OCCURS  24  TIMES  INDEXED  BY  K.
                20   WS-TEMP  PIC 9(2)V9.
```

The subscript/index can be a positive integer or a numeric data item or a integer type data item + or – an integer or another integer type data item. If a Table with name WS-EMP-TABLE has 1000 elements, then they are WS-EMP-TABLE(1), WS-EMP-TABLE(2), . . . , WS-EMP-TABLE(1000). Then the largest value that the subscript/index can take is the integral value specified in the OCCURS clause. To deal with multi-dimensional arrays, there is flavour of PERFORM statement that can be used.

Format:



5.5. SET statement

Though the value of a subscript can be modified using MOVE and ARITHMETIC statements, the value of an index can be modified using only the SET statement. There are two basic formats of the SET statement.

Format 1: SET index – name – 1 [, index – name – 2] . . . TO { identifier / integer / index – name – 3 }.

Examples:

1. SET I TO 4.

2. SET I, J TO K. Here I, J and K are all indexes.

Format 2: SET index – name – 1 [, index – name – 2] { TO / UP BY / DOWN BY }. { identifier / integer }

Examples:

1. SET I UP BY 1.

2. SET I, J DOWN BY K. Here I, J and K are all indexes.

5.6. Subscript Vs Index

Subscript	Index
Represents an occurrence of a Table element.	Represents the displacement from the address of the first element in the Table.
Is defined explicitly in the WORKING-STORAGE SECTION.	An index is a special subscript created and maintained by the operating system.
To change the value of subscript, use a PERFORM . . . VARYING or a MOVE statement or a Arithmetic statement.	To change the value of subscript, use a PERFORM . . . VARYING or a SET statement.

Table 5.1 Subscript Vs Index

5.7. Sample Programs

5.7.1. Program to find the sum and average of 10 numbers stored in an array

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. ARREX2.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  NUMS.
           05  NUM OCCURS 10 TIMES PIC 9(2).
       01  S    PIC 9(3) VALUE ZERO.
       01  I    PIC 9(2) VALUE 1.
       01  A    PIC 9(2).99.
       PROCEDURE DIVISION.
       A1000-MAIN-PARA.
           PERFORM A2000-PROCESS-PARA 10 TIMES.
           PERFORM A3000-DISPLAY-PARA.
           STOP RUN.
       A2000-PROCESS-PARA.
           ACCEPT NUM(I).
           COMPUTE S = S + NUM(I).
           COMPUTE I = I + 1.
       A3000-DISPLAY-PARA.
           DISPLAY "THE SUM OF THE NUMBERS IS " S.
           COMPUTE A = S / 10.
           DISPLAY "THE AVERAGE OF THE NUMBERS IS " A.
```

5.7.2. Program to add two 2 X 2 matrices

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. ARREX3.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  MATA.
           05  ROWA OCCURS 2 TIMES.
              10  COLA OCCURS 2 TIMES PIC 9.
       01  MATB.
           05  ROWB OCCURS 2 TIMES.
              10  COLB OCCURS 2 TIMES PIC 9.
       01  MATC.
           05  ROWC OCCURS 2 TIMES.
              10  COLC OCCURS 2 TIMES PIC 9(2) VALUE ZEROS.
       01  I    PIC 9  VALUE 1.
       01  J    PIC 9  VALUE 1.
       PROCEDURE DIVISION.
       A1000-MAIN-PARA.
           PERFORM A2000-PROCESS-PARA VARYING I FROM 1 BY 1
             UNTIL I > 2 AFTER J FROM 1 BY 1 UNTIL J > 2
           PERFORM A3000-DISPLAY-PARA
           STOP RUN
           .
       A2000-PROCESS-PARA.
           ACCEPT COLA(I, J)
           ACCEPT COLB(I, J)
           COMPUTE COLC(I, J) = COLA(I, J) + COLB(I, J)
           .
       A3000-DISPLAY-PARA.
           DISPLAY COLA(1, 1) "  "  COLA(1, 2
           DISPLAY COLA(2, 1) "  "  COLA(2, 2)
           DISPLAY " "
           DISPLAY COLB(1, 1) "  "  COLB(1, 2)
           DISPLAY COLB(2, 1) "  "  COLB(2, 2)
           DISPLAY " ".
           DISPLAY COLC(1, 1) "  "  COLC(1, 2)
           DISPLAY COLC(2, 1) "  "  COLC(2, 2)
           .
```

5.8. SEARCH Statement

The SEARCH statement is used to locate the elements in a one-dimensional Table.

Format:

SEARCH Table - name

[AT END imperative statements]

WHEN condition – 1 { imperative statements / NEXT SENTENCE } ...

[END – SEARCH]

The following rules apply to the SEARCH statement.

1. The SEARCH statement can be applied to a Table only if it is indexed using INDEXED BY phrase.
2. Before using the SEARCH statement, the index must be initialized using a SET statement.
3. If the SEARCH statement terminates without finding the particular element in the Table, then the index has an unpredictable value.
4. The SEARCH statement cannot be used to find multiple matches.
5. The SEARCH statement does a sequential or serial search of the table.

5.9. SEARCH ALL Statement

When the Table entries are arranged in sequence by some field, such as EMP-ID, the most efficient type of look-up is a binary search. The SEARCH ALL statement provides a means for doing the same.

Format:

SEARCH ALL Table - name

[AT END imperative statements]

WHEN { data – name – 1 { IS EQUALTO / IS = } { identifier – 2 / literal – 1 / arithmetic – expression – 1 } }

[AND { data - name - 2 } { IS EQUALTO / IS = } { identifier – 3 / literal – 2 / arithmetic – expression – 2 }].

.....

{ imperative statements / NEXT SENTENCE }

[END – SEARCH]

The following are the limitations of the SEARCH ALL statement.

1. The condition following the word WHEN can only test for equality.
2. IF the condition following the WHEN is a compound conditional, then
   (a) Each part of the conditional can only consist of a relational test that involves an equal condition.
   (b) The only compound condition permitted is with ANDs and not ORs.
3. Only one WHEN clause can be used.
4. The VARYING option cannot be used.
5. The OCCURS item and its index, which define the Table argument, must appear to the left of the equal to sign.

Example:

```
SEARCH ALL WS-EMPLOYEE-TABLE
        AT END PERFORM C5000-EXIT-PARA
        WHEN  WS-EMPLOYEE-ID = 1000
                MULTIPLY BASIC-SALARY BY 2 GIVING NEW-BASIC-SALARY
END-SEARCH.
```

5.10. SEARCH Vs SEARCH ALL

SEARCH	SEARCH ALL
The Table entries need not be in sequence.	The Table entries must be in some sequence.
Requires SET statement prior to SEARCH.	Does not require SET statement prior to SEARCH ALL.
Can include any relational condition.	The condition has to be an equality.
May use multiple WHENs.	Cannot use multiple WHENs.

Table 5.2 SEARCH Vs SEARCH ALL

6 Miscellaneous utilities

OVERVIEW

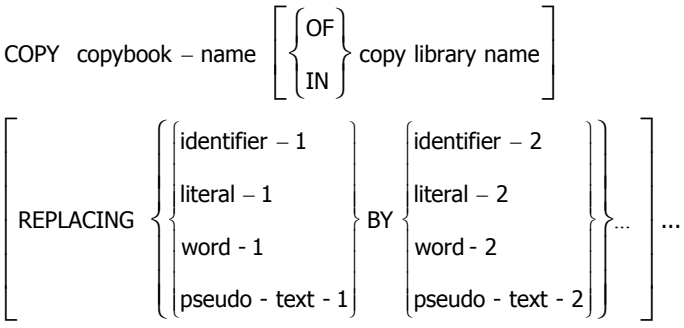
1. COPY utility.
2. CALL utility.
3. SORT utility.
4. MERGE utility.
5. String handling utilities.

6.1. COPY utility

Most business computer systems consist of many programs and usually files are accessed by more than one program in the system. In addition, record structures and routines such as date validation routine are generally used by several programs. In these situations, it is important to ensure that each program has the same file description, record structure or code. Maintaining several copies of the same thing leads to errors and is also time consuming. Each time you modify one copy, you also need to modify all the other copies and each modification is an opportunity to introduce errors. To address all these issues, COBOL provides the COPY statement.

The COPY statement inserts the specified copybook from the copy library "OPERN.CICS3.COPYLIB" into the source program during compilation. Thus, unlike all other COBOL statements, which gets executed during runtime, the COPY statement gets executed during the compile time. In addition to merely inserting a copybook from a copy library, the COPY statement can also modify the text as it is being inserted, by replacing words contained in the copybook. To facilitate, we must use the REPLACING BY clause. A COPY statement can be placed anywhere in the source program where a character string can be used. But a very important point to note is that the COPY statement cannot be nested.

Format:



Examples:

1. COPY STUDREC.
2. COPY STUDREC REPLACING ALL "FS" BY "WS".

6.2. CALL utility

The CALL statement facilitates another way of implementing code sharing and modularity in COBOL programs. In COBOL, it is possible to execute another COBOL program with a CALL. This feature is the equivalent to the function or subroutine call in languages, such as C, FORTRAN, etc.

When a CALL is made, control is transferred to the called program. The called program executes and on completion returns control to the calling program. The calling program continues execution from the point of the call.

6.2.1. LINKAGE SECTION

It is possible to pass parameters to the called program. This is done by the USING clause. The variables named in the USING clause are passed to the called program. In the called program we must code LINKAGE SECTION to receive the passed variables.

Example:

```
CALL DEPOSIT USING WS-AMOUNT.
```

The default method of passing variables is BY REFERENCE. In this method, the called program can modify the parameter. This change is visible in the calling program after it receives control again. As opposed to this, it is possible to pass the variables BY CONTENT. The called program, in this case, may make changes to the variable but these changes are not reflected in the calling program. This method is normally used to pass read only variables.

Example:

```
CALL "JUL-DATE" USING BY CONTENT WS-DATE,
                      BY REFERENCE WS-JULIAN-DATE,
                      BY REFERENCE WS-ERROR-CODE
END-CALL
```

6.2.2. Rules for coding CALLed programs

1. The called program needs to have a LINKAGE SECTION. This must appear after the WORKING-STORAGE SECTION in the DATA DIVISION. In this section, the variables passed to the program must be described.
2. The PROCEDURE DIVISION needs to have a USING clause. This identifies the variables passed to the program and their ordering. Entries in the LINKAGE SECTION can be in any order, but the entries in the USING clause identify the order in the CALL statement.
3. Instead of a STOP RUN statement, the called program must contain an EXIT PROGRAM statement to transfer control back to the calling program.

Remark:

A called program continues to exist in the same state as when it was executing even after termination. This also means that all variables retain values between calls. To free the space occupied by the program, the CANCEL statement can be included in the calling program.

6.3. Program to illustrate the use of CALL verb

6.3.1. Main Program

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. ADDPROD.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01 CHOICE  PIC 9.
       01 WS-NUM1 PIC 99.
       01 WS-NUM2 PIC 99.
       01 WS-NUM3 PIC 9999 VALUE ZEROES.
       PROCEDURE DIVISION.
       A1000-MAIN-PARA.
           ACCEPT CHOICE
           ACCEPT WS-NUM1
           ACCEPT WS-NUM2
           EVALUATE CHOICE
               WHEN 1 CALL "CALLADD" USING WS-NUM1, WS-NUM2,
      -            WS-NUM3
               WHEN 2 CALL "CALLPROD" USING WS-NUM1, WS-NUM2,
      -            WS-NUM3
           END-EVALUATE
           DISPLAY WS-NUM1, WS-NUM2, WS-NUM3.
           STOP RUN
           .
```

6.3.2. Subprogram to add two numbers

```
IDENTIFICATION DIVISION.
       PROGRAM-ID. CALLADD.
       DATA DIVISION.
       LINKAGE SECTION.
       01 NUM1 PIC 99.
       01 NUM2 PIC 99.
       01 NUM3 PIC 9999.
       PROCEDURE DIVISION USING NUM1, NUM2, NUM3.
       A1000-MAIN-PARA.
           ADD 1 TO NUM1
           ADD 1 TO NUM2
           COMPUTE NUM3 = NUM1 + NUM2
           EXIT PROGRAM
           .
```

6.3.3. Subprogram to multiply two numbers

```
IDENTIFICATION DIVISION.
       PROGRAM-ID. CALLADD.
       DATA DIVISION.
       LINKAGE SECTION.
       01 NUM1 PIC 99.
       01 NUM2 PIC 99.
       01 NUM3 PIC 9999.
       PROCEDURE DIVISION USING NUM1, NUM2, NUM3.
       A1000-MAIN-PARA.
           ADD 1 TO NUM1
           ADD 1 TO NUM2
           COMPUTE NUM3 = NUM1 * NUM2
           EXIT PROGRAM
           .
```

6.4. SORT utility

A very common necessity in almost all business oriented applications is the sorting of data in a file or merging of two or more files. Sorting is normally a precursor to report generation. There are two techniques used for sorting files processed by COBOL programs. One is to use a utility sort program, which is typically supplied along with the operating system. This sort utility is completely different from, or external to the COBOL program and would be executed first if records needed to be in a sequence other than the sequence in which they were created. For an external utility, one has to just specify the key fields to sort on. As an alternative, COBOL has the SORT verb, which can make it very useful as a part of a COBOL program.

Format 1:

SORT file - name – 1

{ ON { DESCENDING / ASCENDING } KEY data – name }...

USING file - name - 2

GIVING file - name - 3

As you notice in the above format, the SORT statement includes three files.

1. An Input file containing the unsorted records (Referred by file – name – 2).
2. A scratch file (Work file) for sorting (Referred by file – name – 1).
3. An Output file to place the sorted records (Referred by file – name – 3).

This SORT statement does the following.

1. Opens SORT-FILE in the I-O mode, UNSORTED-STUDENT in the INPUT mode and SORTED-STUDENT in the OUTPUT mode.
2. Transfers the records present in UNSORTED - FILE to SORT-FILE.
3. Sorts SORT-FILE on ascending sequence by WS-REGNO.

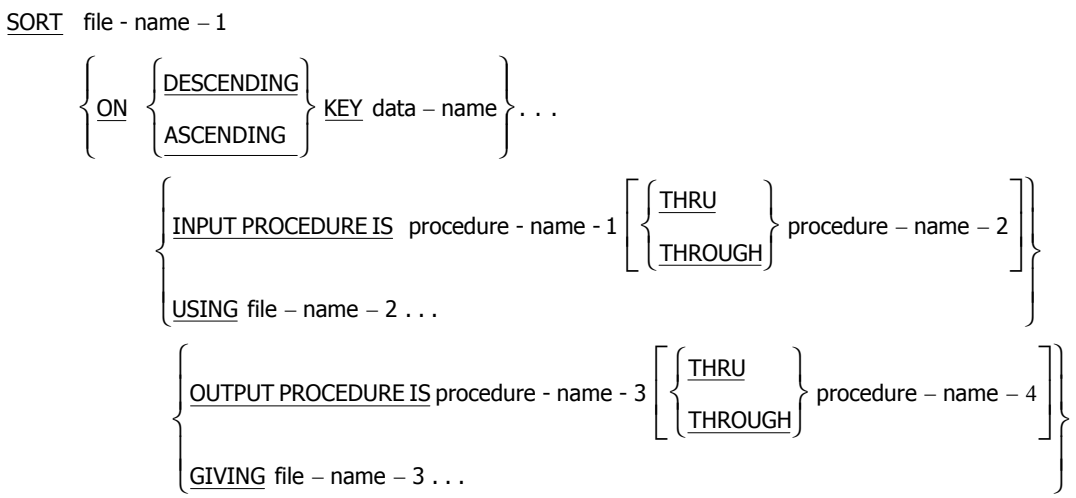
4. Transfers the sorted records from SORT-FILE to the SORTED-STUDENT.
5. Closes the UNSORTED-STUDENT and SORTED-STUDENT files and deletes the SORT-FILE.

6.5. Program to sort a SEQUENTIAL file

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. SORTPGM.
       ENVIRONMENT DIVISION.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT UNSORTED-STUDENT ASSIGN TO STUDENT.
           SELECT SORT-FILE ASSIGN TO SORTWK01.
           SELECT SORTED-STUDENT ASSIGN TO SORTSTUD.
       DATA DIVISION.
       FILE SECTION.
       FD  UNSORTED-STUDENT.
       01  UNSORTED-RECORD.
           05 U-REGNO PIC X(5).
           05 U-NAME  PIC A(15).
           05 U-AGE   PIC 9(2).
           05 FILLER  PIC X(58).
       SD  SORT-FILE.
       01  SORT-REC.
           05 W-REGNO PIC X(5).
           05 FILLER  PIC X(75).
       FD  SORTED-STUDENT.
       01  SORTED-RECORD.
           05 S-REGNO PIC X(5).
           05 S-NAME  PIC A(15).
           05 S-AGE   PIC 9(2).
           05 FILLER  PIC X(58).
       PROCEDURE DIVISION.
       A1000-MAIN-PARA.
           SORT SORT-FILE
                ON ASCENDING KEY W-REGNO
                USING UNSORTED-STUDENT
                GIVING SORTED-STUDENT
           STOP RUN.
```

At times it is required to do some pre-processing of input and post processing of output from the sort. For example, only some input in data file like employees in a particular department needs to be processed. It is possible to do this by opening the input file, filtering the data and writing it to another data file. The filtered data file could then be sorted. However, if this could be done while the copying of data from the input file to sort file, then it would be much faster. To do this, a variant of the SORT statement with INPUT PROCEDURE IS phrase is used. The output from the sort is normally used for report generation. This can be done efficiently with the OUTPUT PROCEDURE IS phrase.

Format 2:



Here, the INPUT PROCEDURE IS phrase specifies the name of a procedure that is to select or modify input records before the sorting operation begins. In this procedure do the following.

1. Open the Input file.
2. Perform FILTER-PARA.
3. After all the records have been processed, close the Input file.

In the FILTER-PARA do the following.

1. Perform any operations required on the Input file.
2. Move the input data to the Sort record.
3. RELEASE each Sort record for making it available for sorting.
4. Continue to read the input, until there is no more data.

Example:

```
A1000-MAIN-MODULE.
      SORT WORK-FILE ON ASCENDING KEY S-EMP-ID
           INPUT PROCEDURE 200-TEST-IT GIVING SORTED-FILE
      STOP RUN
      .
A2000-TEST-IT.
      OPEN INPUT INP-FILE
      PERFORM UNTIL EOF = 'N'
            READ IN-FILE AT END MOVE 'N' TO EOF
                        NOT AT END PERFORM 300-PROCESS-PARA
            END-READ
      END-PERFORM
      CLOSE INP-FILE
      .
A3000-PROCESS-PARA.
      IF QTY = ZEROS CONTINUE
      ELSE MOVE IN-REC TO WORK-REC
           RELEASE SORT-REC
      .
```

The OUTPUT PROCEDURE IS phrase specifies the name of a procedure that is to select or modify output records from the sorting operation. In this procedure do the following.

1. Open the Output file.
2. Perform FILTER-PARA.
3. After all the records have been processed, close the Output file.

In the FILTER-PARA do the following.

1. Perform any operations on the sorted records.
2. Move the sort record to the output area.
3. Write each sort record to the Output file.

6.6. MERGE utility

Merging is commonly used when data files having the same form of data have to be combined to form a larger unit for processing. A common example is the processing at the head office of data from branch offices. The branch offices send files containing their data to the head office and at the end of the month the head office merges this data and generates a consolidated report.

The MERGE statement combines two or more identically sequenced files (that is, files that have already been sorted according to an identical set of ascending/descending keys) on one or more keys and makes records available in merged order to an output procedure or output file. Similar to the SORT statement, the MERGE statement also handles opening, closing of files and reading and writing data from and to files automatically.

Format:

```
MERGE  file - name – 1
       { ON { DESCENDING } KEY data – name }...
                { ASCENDING }
            USING file - name - 2 {file - name - 3}...
       { OUTPUT PROCEDURE IS procedure - name - 3 [ { THRU      } procedure – name – 4 ] }
                                                  { THROUGH   }
       { GIVING file – name – 4 . . .                                                  }
```

6.7. String handling utilities

Any data-name whose USAGE is DISPLAY is regarded to be a string in COBOL. The term "String handling" refers to each of the following.

1. Comparison,
2. Concatenation (joining of two or more strings),
3. Segmentation (Reverse of Concatenation),
4. Scanning (Searching a string for the appearance of specific character or a group of characters or counting the number of occurrences of character(s)) and
5. Replacement (Replacing a specified character(s) by another character(s)).

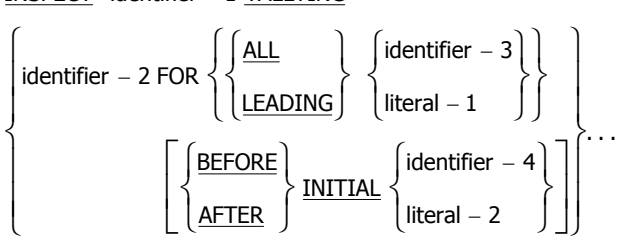
COBOL provides three verbs for string handling, viz. INSPECT, STRING and UNSTRING.

6.7.1. INSPECT verb

The INSPECT verb can be used for two purposes.

1. Count the number of occurrences of a given character in a field.
2. Replace specific occurrences of a given character with another character.

Format 1:



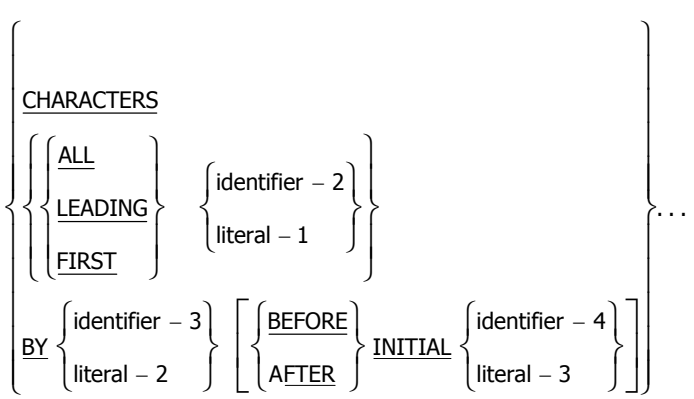
This format is used to count the number of occurrences of a given character in a field.

Examples:

1. INSPECT WS-NAME TALLYING WS-COUNTER FOR ALL SPACES.
2. INSPECT WS-NAME TALLYING WS-COUNTER FOR CHARACTERS BEFORE INITIAL SPACE.
3. INSPECT WS-NAME TALLYING WS-COUNTER FOR LEADING SPACES.

Example #	WS-NAME PIC A(17)	WS-COUNTER
1	NARENDRA RAJE URS	2
2	NARENDRA RAJE URS	8
3	NARENDRA RAJE URS	0

Format 2:



This format is used to replace the specified occurrences of a given character with another character.

Examples:

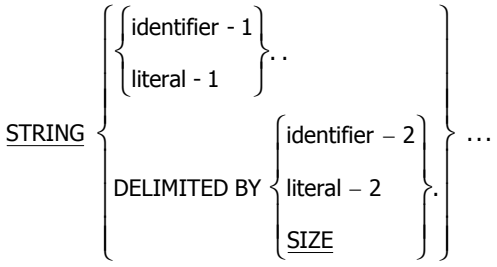
1. INSPECT WS-DATE REPLACING ALL '-' BY '/'.
2. INSPECT WS-NUMBER REPLACING LEADING '1' BY '0'.
3. INSPECT WS-NAME REPLACING FIRST 'A' BY ' B'.
4. INSPECT WS-NAME REPLACING CHARACTERS BY 'X' BY 'D'.

Example #	Input	Output
1	20-06-2001	20/06/2001
2	111034211	000034211
3	NARENDRA	NBRENDRA
4	NARENDRA RAJE URS	XXXXXDRA RAJE URS

6.7.2. STRING verb

The STRING statement strings together the partial or complete contents of two or more data items or literals into one single data item.

Format:



Example: STRING
FIRST-NAME DELIMITED BY ' '
' ' DELIMITED BY SIZE
MIDDLE-NAME DELIMITED BY ' ' LAST-NAME DELIMITED BY ' '
' ' DELIMITED BY SIZE INTO WS-NAME
END-STRING.

OVERFLOW Option

The OVERFLOW option specifies the operation(s) to be performed if the receiving field is not large enough to accommodate the result.

Format:

STRING . . . [ON OVERFLOW imperative statements]

POINTER Option

The POINTER option can be used to count the number of characters actually moved into a receiving field.

Format:

STRING . . . [WITH POINTER identifier]

Rules for using STRING verb

1. The DELIMITED BY clause is required. It can indicate the following.
   (a) SIZE: The entire sending field is transmitted.
   (b) Literal: The transfer of data is terminated when the specified literal is encountered, the literal itself is not moved.
   (c) Identifier: The transfer of data is terminated when the contents of the identifier are encountered.
2. The receiving field must be an elementary data item with no editing symbols or JUSTIFIED RIGHT clause.
3. All literals must be described as non-numeric.
4. The identifier specified must be described as non-numeric.
5. The STRING verb moves data from left to right just like alphanumeric MOVES, but a STRING does not pad with low-order blanks.

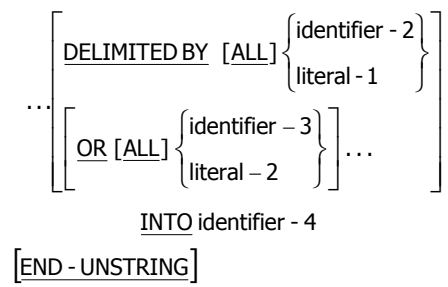
6.7.3. UNSTRING verb

The UNSTRING verb may be used to convert the input data to a more compact form for storing on the DASD. For example, a program may include a statement that causes the following to be displayed on a terminal.

ENTER NAME: LAST, FIRST, MIDDLE INITIAL
: USE COMMAS TO SEPARATE ENTRIES

Since each name has a variable number of characters, there is no way of knowing how large each individual last name and first name is. With the UNSTRING statement, we can instruct the computer to separate the NAME into its components and store them without the commas.

Format:



6.7.4. Rules for using the UNSTRING verb

1. The sending field must be nonnumeric. The receiving fields numeric or nonnumeric.
2. Each literal must be nonnumeric.
3. The WITH POINTER and ON OVERFLOW clauses can be used in the with the STRING statement.

7 Control Break and Report Writer

OVERVIEW

1. Control Break Processing.
2. Report Writer

7.1. Control Break Processing.

We use Control break to generate Control report having following features.

- Print Header at the top of a page, Footer at the bottom of a page. Header/Footer may contain Page number, Date of any other Dynamic values.
- Print Accumulated amount for each value of control field and the grand total for all values of control field put together.
- Skip to a new page based on printing requirement.

Let's take a case study of above kind of report. We have an input file having Customer Number, Part Number (Sold to customer) and the quantity of each Part.

```
CUSTOMER    PART
NUM         NUM         Quentity
=====       ========    ========
12345  PART-001 100
12345  PART-002 112
12345  PART-003 123
12345  PART-004 567
12345  PART-005 980
. . .
56789  PART-013 113
. . .
67890  PART-001 100
67890  PART-002 112
67890  PART-003 123
```

We want a report having Report Title, Header on each page and subtotal of quantity sold to each customer, after reading the above input file. The Report layout is given below.

```
        SUMMARY REPORT FOR PART NUMBER
    CUSTOMER NUMBER     PART NUMBER    PART QTY                      PAGE  1
    12345               PART-001       100
    12345               PART-002       112
    12345               PART-003       123
    . . .
                                       TOTAL QTY = 2117
        SUMMARY REPORT FOR PART NUMBER
    CUSTOMER NUMBER     PART NUMBER    PART QTY                      PAGE  2
    56789               PART-013       123
    56789               PART-017       112
    . . .
    56789               PART-018       123
        SUMMARY REPORT FOR PART NUMBER
    CUSTOMER NUMBER     PART NUMBER    PART QTY                      PAGE  3
       56789               PART-020       980
                                       TOTAL QTY = 3529
    67890               PART-001       100
    67890               PART-002       112
    67890               PART-003       123
                                       TOTAL QTY =  235
```

Below is the COBOL program to generate the above report using Control Break.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CTRLBRK.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FILE-IN ASSIGN TO INFILE
    FILE STATUS IS WS-STATUS-INFILE.

    SELECT FILE-OUT ASSIGN TO OUTFIL
    FILE STATUS IS WS-STATUS-OUTFIL.

DATA DIVISION.
FILE SECTION.
FD FILE-IN
    RECORDING MODE F.
01 IN-REC.
    05 CUST-NO           PIC 9(5).
    05 PART-NO           PIC X(8).
    05 QTY               PIC 9(3).
FD FILE-OUT.
01 OUT-REC                   PIC X(80).

WORKING-STORAGE SECTION.
```

```
01 ARE-THERE-MORE-RECORDS    PIC X(3) VALUE 'YES'.
   88 NO-MORE-RECORDS        VALUE 'NO '.
   88 MORE-RECORDS           VALUE 'YES'.
01 WS-STATUS-INFILE          PIC X(02) VALUE '00'.
01 WS-STATUS-OUTFIL          PIC X(02) VALUE '00'.
01 WS-AMT                    PIC S9(5) COMP-3.
01 WS-PG-CNT                 PIC 9(2)  VALUE ZERO.
01 WS-LN-CNT                 PIC 9(2)  VALUE ZERO.
01 WS-PREV-CUST-NUM          PIC 9(5)  VALUE ZERO.

01 WS-REPORT.
   05 WS-RH.
      10                    PIC X(09) VALUE SPACES.
      10                    PIC X(30) VALUE
         'SUMMARY REPORT FOR PART NUMBER'.
      10                    PIC X(41) VALUE SPACES.
        05 WS-PH.
      10                    PIC X(04) VALUE SPACES.
      10                    PIC X(15) VALUE
             'CUSTOMER NUMBER'.
      10                    PIC X(05) VALUE SPACE.
      10                    PIC X(11) VALUE
            'PART NUMBER'.
      10                    PIC X(04) VALUE SPACES.
      10                    PIC X(08) VALUE 'PART QTY'.
      10                    PIC X(22) VALUE SPACES.
      10                    PIC X(05) VALUE 'PAGE '.
      10 WS-PAGE            PIC Z9.
      10                    PIC X(04) VALUE SPACES.
   05 WS-DETAIL-REC.
      10                    PIC X(04) VALUE SPACES.
      10 WS-CUST-NO         PIC 9(05) VALUE ZEROS.
      10                    PIC X(15) VALUE SPACES.
      10 WS-PART-NO         PIC X(08) VALUE SPACE.
      10                    PIC X(07) VALUE SPACES.
      10 WS-QTY             PIC 9(03) VALUE ZEROS.
      10                    PIC X(38) VALUE SPACES.
   05 WS-TOTAL-REC.
      10                    PIC X(39) VALUE SPACES.
      10                    PIC X(11) VALUE
           'TOTAL QTY ='.
      10 WS-PART-TOTAL      PIC Z(4)9.
      10                    PIC X(25) VALUE SPACES.

PROCEDURE DIVISION.
1000-MAIN-MODULE.
    OPEN INPUT FILE-IN
    OUTPUT FILE-OUT

    READ FILE-IN
```

```
    AT END SET NO-MORE-RECORDS TO TRUE
    END-READ

    IF MORE-RECORDS
       MOVE CUST-NO       TO WS-CUST-NO
                             WS-PREV-CUST-NUM
       MOVE PART-NO       TO WS-PART-NO
       MOVE QTY           TO WS-QTY
                             WS-AMT
       PERFORM 1100-PAGE-HEADER
       PERFORM 1200-DETAIL-LINE
               UNTIL NO-MORE-RECORDS
   END-IF

   CLOSE FILE-IN
         FILE-OUT
   GOBACK.

1100-PAGE-HEADER.
    MOVE WS-RH            TO OUT-REC
    WRITE OUT-REC AFTER ADVANCING PAGE
    ADD 1                 TO WS-PG-CNT
    MOVE WS-PG-CNT        TO WS-PAGE
    MOVE WS-PH            TO OUT-REC
    WRITE OUT-REC AFTER 2 LINES.
    MOVE ZERO             TO WS-LN-CNT.

1200-DETAIL-LINE.
    IF WS-CUST-NO NOT = WS-PREV-CUST-NUM
       PERFORM 1210-CONTROL-BREAK
    END-IF

    IF WS-LN-CNT > 5
       PERFORM 1100-PAGE-HEADER
    END-IF.

    MOVE WS-DETAIL-REC   TO OUT-REC
    MOVE WS-CUST-NO      TO WS-PREV-CUST-NUM
    WRITE OUT-REC AFTER ADVANCING 2 LINES
    ADD 1                TO WS-LN-CNT.

    READ FILE-IN
         AT END SET NO-MORE-RECORDS TO TRUE
    END-READ

    IF MORE-RECORDS
       MOVE CUST-NO    TO WS-CUST-NO
       MOVE PART-NO    TO WS-PART-NO
       MOVE QTY        TO WS-QTY
       ADD QTY         TO WS-AMT
```

```
    ELSE
       PERFORM 1210-CONTROL-BREAK
    END-IF.

1210-CONTROL-BREAK.
    MOVE WS-AMT        TO WS-PART-TOTAL
    MOVE WS-TOTAL-REC  TO OUT-REC
    WRITE OUT-REC AFTER ADVANCING 2 LINES
    INITIALIZE WS-AMT.
```

Let's focus on the shaded line of the above program.

- We are predefining how many records to be printed on each page.

  IF WS-LN-CNT > 5

  PERFORM 1100-PAGE-HEADER

  Here we are printing 5 detail records per page. The printing of Page header is controlled by checking the value of WS-LN-CNT. The moment the WS-LN-CNT crossed predefined page limit, the page header is being printed on the next page using '**AFTER ADVANCING PAGE**' clause.

- Likewise, using '**AFTER ADVANCING 2 LINES**' clause we are leaving one blank line before printing the record for Part number.

- The subtotal for Parts are printed for each customer using Control Break. The input records have to be sorted on Control fields (here it is CUST-NO) before using such kind of technique (We can use the SORT utility of COBOL to do so). Here the current CUST-NUM is being compared with the Previous CUST-NUM after reading each record. The moment the current CUST-NUM doesn't match with previous one, paragraph 1210-CONTROL-BREAK is called to print the subtotal of Parts.

- At **END-OF-FILE** condition (of input file) we should call the paragraph **1210-CONTROL-BREAK** to print the subtotal for the last CUST-NUM.

In the above example, CUST-NUM is the only Control file and hence we used one-level Control break as shown in the above example. We can have multiple-level Control break if we have more than one control fields.

Note: The length of output file (that stores the Report generated using the Control Break processing) in JCL should be 1 byte more than the record length used in the COBOL program. This extra one byte stores the Control break information (Page break and Line Break).

Suggestive Example:

Generate a control report having multiple control fields. Print the subtotal amount for each value of control field and the grand total for all values of the control fields. Use an input file, which is not sorted on the control file and use SORT utility before using Control break processing.

7.2. Report Writer.

We have seen how the generate Control Report using Control Break Processing. We can generate Control report using Report Writer facility of COBOL in an easier way. Report writer automatically handles the points mentioned below.

- Print Header at the top of a page, Footer at the bottom of a page. Header/Footer may contain Page number, Date of any other Dynamic values.
- Print Accumulated amount for each value of control field and the grand total for all values of control field put together.
- Skip to a new page based on printing requirement.
- Printing repeated value of Control field only once at the first row.

The following case study explains the above features. We have an input file having Customer Number, Part Number (Sold to customer) and the quantity of each Part.

CUSTOMER NUM	PART NUM	Quentity
12345	PART-001	100
12345	PART-002	112
12345	PART-003	123
12345	PART-004	567
12345	PART-005	980
12345	PART-012	112
67890	PART-001	100
67890	PART-002	112
67890	PART-003	123

We need a report having Report Title, Header on each page and subtotal of quantity sold to each customer as give below.

```
                    SUMMARY REPORT FOR PART NUMBER
    CUSTOMER NUMBER     PART NUMBER    PART QTY                    PAGE 01

    12345               PART-001       100
                        PART-002       112
                        PART-003       123
                        PART-004       567
                        PART-005       980
                        PART-012       112
                                       TOTAL QTY =     1994

    CUSTOMER NUMBER     PART NUMBER    PART QTY                    PAGE 02

    67890               PART-001       100
                        PART-002       112
                        PART-003       123
                                       TOTAL QTY =      335
```

```
                ***END OF REPORT***
```

Below the COBOL code to generate the above said report.

```
. . .
DATA DIVISION.
FILE SECTION.
FD INPUT-FILE
    RECORDING MODE F.
01 INPUT-RECORD.
    05 CUST-NO                    PIC 9(5).
    05 PART-NO                    PIC X(8).
    05 QTY                        PIC 9(3).

FD OUT-FILE
    RECORDING MODE F
    REPORT IS CUSTOMER-REPORT.

 WORKING-STORAGE SECTION.
01 INFILE-EOF                   PIC X(3) VALUE 'YES'.
    88 INFILE-NOT-EOF                 VALUE 'NO '.

REPORT SECTION.
RD CUSTOMER-REPORT
   CONTROLS ARE CUST-NO
   PAGE LIMIT IS 25 LINES
   HEADING 3
   FIRST DETAIL 9.
01 TYPE RH.
   05 LINE 3.
      10 COLUMN 10                PIC X(30)
         VALUE 'SUMMARY REPORT FOR PART NUMBER'.
      10 COLUMN 79                PIC X(1)
         VALUE SPACES.
01 TYPE PH.
   05 LINE  5
      COLUMN 5                    PIC X(15)
         VALUE 'CUSTOMER NUMBER'.
   05 COLUMN 25                   PIC X(11)
         VALUE 'PART NUMBER'.
   05 COLUMN 40                   PIC X(8)
         VALUE 'PART QTY'.
   05 COLUMN 70                   PIC X(4)
          VALUE 'PAGE'.
    05 COLUMN 75                  PIC 99
          SOURCE PAGE-COUNTER.
 01 DETAIL-LINE TYPE DE LINE PLUS 2.
    05 COLUMN 5                   PIC 9(5)
          SOURCE IS CUST-NO
          GROUP INDICATE.
    05 COLUMN 25                  PIC X(8)
```

```
          SOURCE IS PART-NO.
    05 COLUMN 40                  PIC 9(3)
          SOURCE IS QTY.
 01 TYPE CF CUST-NO, NEXT GROUP NEXT PAGE.
    05 LINE PLUS 2
       COLUMN NUMBER IS 40        PIC X(12)
          VALUE 'TOTAL QTY = '.
    05 AMT COLUMN 55              PIC ZZ999
          SUM QTY.
 01 TYPE RF.
    05  LINE NEXT PAGE
        COLUMN   18               PIC X(19)
       COLUMN   18                PIC X(19)
       VALUE '***END OF REPORT***'.

PROCEDURE DIVISION.
1000-MAIN-MODULE.
    OPEN INPUT INPUT-FILE
         OUTPUT OUT-FILE

    INITIATE CUSTOMER-REPORT

     READ INPUT-FILE
        AT END MOVE 'NO ' TO INFILE-EOF
     END-READ

     PERFORM 2000-PROCESS
        UNTIL INFILE-NOT-EOF

     TERMINATE CUSTOMER-REPORT

     CLOSE INPUT-FILE
           OUT-FILE
     GOBACK.

2000-PROCESS.
    DISPLAY INPUT-RECORD
    GENERATE DETAIL-LINE

    READ INPUT-FILE
       AT END MOVE 'NO ' TO INFILE-EOF
    END-READ.
```

Explanation of different key words used in the above program is given below.

Printing Heading and Footing:

REPORT HEADING (RH): Prints identifying information about the report only once, at the top of the first page of the report.

PAGE HEADING (PH): Prints identifying information at the top of each page. This may include page number, column heading and so on.

CONTROL FOOTING (CF): Prints control totals for the previous group of detail records just printed, after a control break has occurred.

PAGE FOOTING (PF): Prints at the end of each page.

REPORT FOOTING (RF): Prints only once, at the end of the report. This generally included the reporting ending message.

REPORT SECTION in the DATA DIVISION:

The REPORT SECTION is located in DATA DIVISION and must the coded in the order shown.

- FILE SECTION.
- WORKING STORAGE SECTION.
- REPORT SECTION (in case Report Writer is used).

Let's consider the above said program.

- The Report name is associated with an output file (that stores the report) in the FD section in the following fashion.

```
FD file-name
   [RECORD CONTAINS n CHARACTERS]
  { REPORT IS    } Report-name-1 . . .
  { REPORTS ARE  }
```

The REPORT SECTION specifies the following points.

1. The report group type that describes each type of line (e.g. Report Heading, Page Heading, Detail).
2. The line on which each record is to print. This can be specified as an actual line number or a relative line number that relates to a previous line.
3. The control fields on which the report is divided into sections (here CUST-NO). Likewise we can use FINAL as control field to print grand total of an amount filed.
4. The positions within each line where data is to print. Each field can be given a VALUE or can be used as summarization fields.

With these specifications in the REPORT SECTION, the PROCEDURE DIVISION need not include coding for control break or summary operations.

The **INITIATE** clause begins the processing of a report. It is usually coded immediately after the OPEN statement. It initiates the Report Writer. The INITIATE clause sets all sum and COUNTER fields to zero, including LINE-COUNTER and PAGE-COUNTER.

The **GENERATE** clause is used to produce the report. It usually names a detail report group to be printed after an input record has been read.

The **TERMINATE** clause completes the processing of a report after all records have been processed. The TERMINATE clause produces all CONTROL FOOTIING report groups. This is usually coded just before the files are closed.

Suggestive question: Write a COBOL program that will print the total Part amount for each Customer as well as the grand total for Part amount for all customers put together, for the given input file.

8 Advanced Topics

OVERVIEW

1. JCL and COBOL Interactions.
2. Processing of Alternate Index for VSAM file.
3. Compiler Option.
4. Compiler Listing.
5. Static and Dynamic Call.
6. Global and External Data Elements
7. Debugging Tools and Techniques.

8.1. JCL and COBOL Interactions.

To run COBOL program in batch mode we use JCL. The sample JCL to run COBOL program is given below.

```
//TR99999C JOB ,,NOTIFY=TR99999,CLASS=B,MSGLEVEL=(1,1),MSGCLASS=X
//JOBLIB DD DSN=TRN001.COBOL.EXE.LIB,DISP=SHR
//STEP1 EXEC PGM=COBPGM1,PARM='02/07/03'
//STEPLIB DD DSN=TRN001.COBOL.EXE.LIB,        <-- Stores the link-edited version of COBPGM1
//         DISP=SHR
//SYSOUT DD SYSOUT=*
//INFILE DD DSN=TRN001.COBOL.COBPGM1.INFILE1,DISP=SHR
//OUTFILE DD DSN=TRN001.COBOL.COBPGM1.OUTFILE1,
//          DISP=(NEW,CATLG,DELETE),UNIT=SYSDA,
//          DCB=(LRECL=100,RECFM=FB,BLKSIZE=1000),
//          SPACE=(1000,(100,50),RLSE)
//SYSIN DD *
123
456
/*
//*********END OF JOB************************
```

The description of the above JCL is given below.

Program:

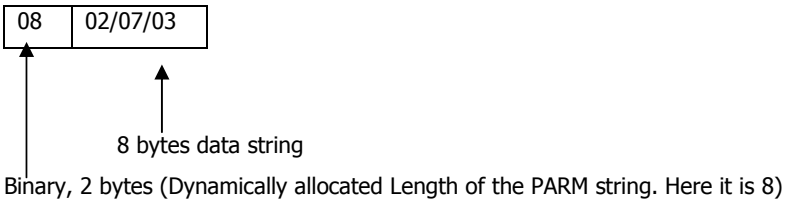
COBPGM1 is the COBOL program.

PARM (Input parameter):

We can pass a string of from JCL to COBOL program using PARM. The COBOL program will receive the parameter string in the field declared in the LINKAGE SECTION. Internally the PARM string (on the EXEC statement) is preceded by a half-word binary field containing the number of bytes of string in the PARM. For example, if we code

```
// STEP1 EXEC PGM=PROGRAM1,PARM='02/07/03'
```

The data passed will be



The main program would need something like this:

```
LINKAGE SECTION.
01   PARM-DATA.
   05 PARM-LENGTH     PIC S9(4) COMP.
   05 PARM-CONTENT     PIC X(8).

PROCEDURE DIVISION USING PARM-DATA.
. . .
```

If the length of PARM in your JCL does not match with the length of the field defined in the linkage section, the job will abend with **0C4**.

STEPLIB and JOBLIB:

The Partitioned Dataset 'TRN001.COBOL.EXE.LIB' mentioned against DD name **STEPLIB** stores program executable (Link edited version). We can also mention this PDS (Partitioned Dataset) name against DD name **JOBLIB** in the JOB card of the JCL. The JOBLIB information is ignored, if both JOBLIB and STEPLIB are used in the JCL. While allocating the PDS 'TRN001.COBOL.EXE.LIB', the RECFM should be 'U' (Undefined length record) and record length should be blank.

SYSOUT:

All messages written using **DISPLAY** command inside COBOL program will be directed to DD name **SYSOUT**. Instead of directing **SYSOUT** to System log (using SYSOUT DD SYSOUT=*), we can route it to a dataset (of maximum record length = 137).

SYSIN:

The value mention in the SYSIN is received by **ACCEPT** command inside the COBOL program. Multiple rows in the SYSIN card are received by issuing multiple **ACCEPT** commends.

Input File:

The DD name of the file is one that is mentioned in the 'ASSIGNED TO' clause inside the COBOL program. The length of this name should be 8 character of less. We use Disposition parameter 'SHR' or 'OLD' for input file.

Output File:

The following points are focusing on Sequential output file.

- If we want to crate a new file, we use DISP=(NEW,CATLG,DELETE). In this case we need to provide the file allocation parameter like DCB, SPACE and UNIT. The number of Primary and secondary blocks (in SPACE parameter) should be chosen properly so that size of the output file becomes greater than or equal to the size (number of Bytes) used by the total number records to be written to the output file. For DISP=(NEW,CATLG,DELETE), the Insufficient file size results to Space abend.
- For a fixed length file, value of RECFM should be equal to the record length of the file used inside the COBOL program.
- For variable length file value of RECFM should be 4 bytes more than the maximum length of the variable length file used inside the COBOL program. This extra 4 bytes store the length information of each variable length record.
- TO append record in an existing output file, we can use either of the following method.
  1) Use DISP= OLD for the output file and open the file in EXTEND mode inside COBOL program.
  2) Use DIS=(MOD,CATLG,DELETE) for the output file and open the file in OUTPUT mode inside COBOL program.
- To overwrite an existing output file, we use DISP= OLD as disposition and open the file in OUTPUT mode.
- To rewrite an existing output file, we use DISP= OLD as disposition and open the file in I/O mode.

8.2. Processing of Alternate Index for VSAM file

Before going to code level detail with Alternate index, let's discuss the very basic concept of VSAM in couple of sentences and examples. In this section KSDS and ESDS are explained out of four types of VSAM files (KSDS, ESDS, RRDS and LDS).

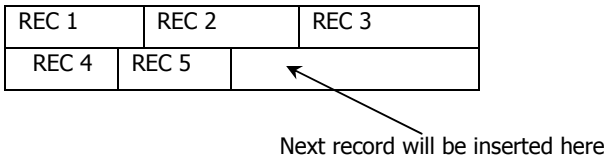
KSDS (Key sequential data set)

In a KSDS, the records are physically sorted in ascending collating sequence of the prime key field. Records can be retrieved and inserted, both randomly and sequentially. FREE SPACE is allocated at regular interval.

ESDS (Entry sequential data set)

An EDDS is analogous to physical sequential file. As in a physical sequential file, new insertions in an ESDS are always added to the end of the data set.

Data component:



The records can be accessed randomly or sequentially. The records cannot be physically deleted. Since records in an ESDS are not sequenced on any key field, there is no primary key index component. However as in a KSDS, ESDS can have Alternate index. Embedded FREE SPACE is not allocated at the time of allocation of an ESDS, since the records are not added to the middle of the file.

8.2.1. KSDS with Alternate Index

Let's consider we have a KSDS VSAM file having record layout as mentioned below.

```
01 EMP-REC.
   05 EMP-SSN              PIC 9(9).
   05 EMP-ORG              PIC X(26).
   05 EMP-ID               PIC 9(02).
   05 EMP-SEX              PIC X(1).
   05 EMP-BDATE            PIC 9(6).
   05 EMP-NAME             PIC X(6).
```

And we stored the following data in the above KSDS file.

```
111111111 NAM                          01M121256DATE
222222222 SAM                          02M020589STERN
333333333 CANE                         03M101083MOOR
444444444 WAM                          04F060990SHELBY
555555555 AERO                         05F080979STERN
```

EMP-NAME has duplicate value

We want making **EMP-SSN** as a primary key, **EMP-ID** as an Alternate index and **EMP-NAME** as duplicate Alternate index. Here we need to have three VSAM files; One is the main file and other two are for two Alternate indexes.

The following JCL will create our main VSAM file.

```
//STEP1 EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
 DEFINE CLUSTER                          -
 (NAME(COBOL.KSDS1.CLUSTER)              -
 CYLINDER(1,1)                           -
 CISZ(4096)                              -
 FREESPACE(10,20)                        -
 KEYS(9,0)                               -
 RECORDSIZE(50,50))                      -
 DATA                                    -
 (NAME(COBOL.KSDS1.DATA))                -
 INDEX                                   -
 (NAME(COBOL.KSDS1.INDEX)                -
 CISZ(1024))
/*
```

- In our case record length is fixed and of 50 bytes. So in the RECORDSIZE(min-len, max-len) parameter the minimum length and maximum length are kept as 50. If we use variable record length in COBOL program then we need to choose the minimum and maximum record length accordingly.
- Our primary key is EMP-SSN of length 9 and starting from 1st position. So the KEY(len, starting-pos) is defined as KEYS(9,0). In case of VSAM first position is taken as 0^{th} position.

Now, we need to define the VSAM for each of the Alternate indexes. Following is the JCL to create the unique Alternate index on EMP-ID for the give record layout.

```
//STEP1 EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
 DEFINE AIX                        -
 (NAME(COBOL.KSDS1.AIX1.CLUSTER)   -
 RELATE(COBOL.KSDS1.CLUSTER)       -
 CYLINDER(1,1)                     -
 CISZ(4096)                        -
 FREESPACE(10,20)                  -
 KEYS(2,35)                        -
 UNIQUEKEY                         -
 UPGRADE                           -
 RECORDSIZE(16,16))                -
 DATA                              -
 (NAME(COBOL.KSDS1.AIX1.DATA))     -
 INDEX                             -
(NAME(COBOL.KSDS1.AIX1.INDEX))
/*
```

Here the Alternate index VSAM file COBOL.KSDS1.AIX1.CLUSTER has been tied up to the main VSAM COBOL.KSDS1.CLUSTER using **RELATE** clause. The clauses **KEY(2,35)** and **UNIQUEKEY** specify that the unique alternate key EMP-ID is of length 2 is positioned at 36 column (refer to record layout given above).

To understand the RECORDSIZE(16,16), we need to see the record structure of alternate key.

5 Bytes (Control bytes)	Value of Alternate key	Value of primary key

In our case, record length = 5 + 2 (length of EMP-ID) + 9 (length of EMP-SSN) = 16. Since the EMP-ID unique there cannot be multiple EMP-SSN against one EMP-ID. This makes the record length fixed. Hence, we have RECORDSIZE(min-len, max-len) is RECORDSIZE(16,16)).

The content of COBOL.KSDS1.AIX1.CLUSTER:

```
..... 01 111111111
..... 02 222222222
..... 03 333333333
..... 04 444444444
..... 05 555555555
```

To create duplicate alternate key, we need to pass NONUNIQUE in the SYSIN parameter of IDCAMS and RECORDSIZE(min-len, max-len) has to be chosen properly. Let's assume that we can have minimum 2 records with duplicate EMP-NAME maximum of 5 records for the same. In this case, alternate key VSAM file (for EMP-NAME) will have RECORDSIZE as RECORDSIZE(29,56).

Min-len = 5 (control byte) + 6 (EMP-NAME) + 2 * 9 (EMP-ID) = 29
Max-len = 5 (control byte) + 6 (EMP-NAME) + 5 * 9 (EMP-ID) = 56

For the given input file the alternate Key VSAM file on EMP-NAME will contain the following data.

COBOL.KSDS1.AIX2.CLUSTER:

```
..... DATE   111111111
..... MOOR   333333333
..... SHELBY 444444444
..... STERN  222222222555555555
```

Each Alternate index is attached to the main VSAM file via a Path. Path doesn't take any physical space. It just establishes a link between those two. We define the path in the following manner.

```
//STEP1 EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
 DEFINE PATH                  -
 (NAME(COBOL.KSDS1.PATH1)     -
 PATHENTRY(COBOL.KSDS1.AIX1.CLUSTER))
/*
```

The above explanation clarifies how KSDS stores data and how an alternate index is linked to a KSDS file. Now let's see a COBOL program which accesses the above KSDS file using ALTERNATE key.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. VSMREAD.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT IN-FILE ASSIGN TO INFILE
       ORGANIZATION IS INDEXED
       ACCESS MODE IS DYNAMIC
       RECORD KEY IS EMP-SSN
       ALTERNATE RECORD KEY IS EMP-ID
       ALTERNATE RECORD KEY IS EMP-NAME
                        WITH DUPLICATES
       FILE STATUS IS WS-INFILE-STAT.
DATA DIVISION.
FILE SECTION.
FD IN-FILE.
01 EMP-REC.
    05 EMP-SSN              PIC 9(9).
    05 EMP-ORG              PIC X(26).
    05 EMP-ID               PIC 9(02).
    05 EMP-SEX              PIC X(1).
    05 EMP-BDATE            PIC 9(6).
    05 EMP-NAME             PIC X(6).
 WORKING-STORAGE SECTION.
 01 WS-INFILE-STAT          PIC X(2) VALUE '00'.
 01 WS-EOF                  PIC X(1) VALUE 'N'.
 PROCEDURE DIVISION.
 1000-FIRST.
     OPEN INPUT IN-FILE
     MOVE 2 TO EMP-ID
     START IN-FILE KEY >= EMP-ID
     PERFORM 2000-READ-FILE UNTIL WS-EOF IS = 'Y'
     CLOSE IN-FILE
     STOP RUN.
2000-READ-FILE.
    READ IN-FILE NEXT RECORD
    AT END MOVE 'Y' TO WS-EOF
    NOT AT END DISPLAY EMP-SSN.
```

In the above COBOL program, we are reading the KSDS records sequentially starting from EMP-ID = 2 till end of file. We have started using START clause and then read sequentially using READ <file> NEXT RECORD command.

We can update all the fields except Primary key in the KSDS file. While update/insert record in a KSDS the Alternate key VSAM files will automatically gets updated with the corresponding value provided we have specified to do so (using UPGRADE parameter in the SYSIN of IDCAMS while defining Alternate key).

To run the above COBOL program we need the following JCL.

```
//STEP1 EXEC PGM=VSMREAD
//STEPLIB DD DSN=TRN001.COBOL.EXE.LIB,DISP=SHR
//SYSOUT  DD SYSOUT=*
//INFILE  DD DSN=COBOL.KSDS1.CLUSTER,DISP=SHR
//INFILE1 DD DSN=COBOL.KSDS1.PATH1,DISP=SHR
//INFILE2 DD DSN=COBOL.KSDS1.PATH2,DISP=SHR
```

Following output will be directed to SYSOUT:

222222222
333333333
444444444
555555555

One thing to be noticed here. In the JCL, we are mentioning the main VSAM file name (COBOL.KSDS1.CLUSTER) and the PATH names (COBOL.KSDS1.PATH1 and PATH2) of the alternate keys, but not the file names of the Alternate keys (COBOL.KSDS1.AIX1.CLUSTER and AIX2.CLUSTER). The DD names for the PATH are mentioned as INFILE1 and INFILE2. If we have n number alternate keys for a VSAM file with DD name INFILE, the DD names for the PATH of the alternate keys will be INFILE1, INFILE2, . . , INFILEn.

Suggestive Questions:

1. Write a COBOL program to read the input KSDS VSAM file using Duplicate Alternate index.

8.2.2. ESDS with Alternate Index

The allocation of Alternate key for an ESDS follows the same rule as that of KSDS. But it is important to remember that the COBOL II complier (including new VS COBOL II) does not support alternate indexes on ESDS. Alternate indexes on ESDS are supported by CICS and not supported in a batch COBOL environment.

8.3. Compiler Option

Compiler options help to control the compilation of your program. Submit a compile job and go to Sysout, you will find all the compiler option used by your compiler job in the following format.

```
PP 5688-197 IBM COBOL for MVS & VM   1.2.2
Date
PO 5798-DYR COBOL Report Writer Release 4.00 05/01/92
PROCESS(CBL) statements:
       CBL PGMNAME(COMPAT)
       CBL NONAME,NOTERM
Compiler options in effect:             Precompiler options in effect:
    NOADATA                                   ADV
    NOCMPR2
EXIT(NOINEXIT,NOLIBEXIT,NOPRTEXIT)
      DATA(31)                              NOFMODE
   NODYNAM                                 NOMGENER
   NOFASTSRT                                 OSVS
     FLAG(I,W)                               PPSNS(132)
   NONUMBER
     NUMPROC(NOPFD)
      RENT
    NOSSRANGE
SIZE(MAX)
    NOTEST
TRUNC(STD)
. . .
```

Specify these options in the PARM field of the compilation JCL or on the PROCESS statement in your program.

Compiler option in PARM:

Below is the compilation JCL passing different compiler options in JCL PARM (shaded line). The following JCL uses the VS COBOL II compiler program IGYCRCTL.

```
//TR99999C JOB ,,NOTIFY=TR99999,CLASS=B,MSGLEVEL=(1,1),MSGCLASS=X
//SET1    SET COPYLIB=OPERN.CICS1.COPYLIB
//SET2    SET LOAD=OPERN.CICS1.LOADLIB
//SET3    SET VSAMLIB=OPERN.CICS1.VSAMLIB
//SET4    SET WORK=UNIT=SYSDA,SPACE=(460,(350,100))
//************************************************
//*   COMPILATION                                *
//************************************************
//COMPILE  EXEC PGM=IGYCRCTL,REGION=4M,
//     PARM='DYNAM,LIST,RENT,LIB,OBJECT,TEST,APOST'
//STEPLIB  DD DSN=IGY.V2R1M0.SIGYCOMP,DISP=SHR
//SYSIN    DD DSN=TR99999.COBOL(COBPGM1),DISP=SHR   <-- Stores the Source code of program COBPGM1
//SYSUT1   DD UNIT=SYSDA,SPACE=(460,(350,100))
//SYSLIB   DD DSN=OPERN.CICS1.COPYLIB,DISP=SHR
//         DD DSN=OPERN.CICS1.VSAMLIB,DISP=SHR
//SYSUT2   DD UNIT=SYSDA,SPACE=(460,(350,100))
//SYSUT3   DD UNIT=SYSDA,SPACE=(460,(350,100))
//SYSUT4   DD UNIT=SYSDA,SPACE=(460,(350,100))
//SYSUT5   DD UNIT=SYSDA,SPACE=(460,(350,100))
//SYSUT6   DD UNIT=SYSDA,SPACE=(460,(350,100))
//SYSUT7   DD UNIT=SYSDA,SPACE=(460,(350,100))
//SYSLIN   DD DSN=&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
//         SPACE=(80,(300,100)),DCB=BLKSIZE=400
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSABEND DD SYSOUT=*
//************************************************
//*   LINKEDIT                                   *
//************************************************
//LINKEDIT EXEC PGM=HEWL,COND=(5,LT),REGION=4M
//SYSLIB   DD  DSN=CEE.SCEELKED,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN   DD DSN=&LOADSET,DISP=(OLD,DELETE)
//         DD DDNAME=SYSIN
//SYSUT1   DD UNIT=SYSDA,SPACE=(460,(350,100))
//SYSUT2   DD UNIT=SYSDA,SPACE=(460,(350,100))
//SYSUT3   DD UNIT=SYSDA,SPACE=(460,(350,100))
//SYSLMOD  DD DSN=OPERN.CICS1.LOADLIB(COBPGM1),   <-- Stores the executable of COBPGM1
//           DISP=SHR
```

The PROCESS (Synonym: CBL) Statement:

You can code compiler options on the PROCESS statement. The PROCESS statement is placed before the Identification Division header and has the following format:

```
PROCESS option 1 [,option 2] . . . [,option n]
      IDENTIFICATION DIVISION.
```

One or more blanks must separate PROCESS and the first option. Separate options with a comma or a blank. The PROCESS statement must be placed before any comment lines or compiler-directing statements. There must not be any imbedded spaces within options.

8.3.1. Important Compiler Options

Functionality-wise we can list the compiler options under the following categories.

Functionality-wise category	Important Compiler Options
Control Object Code generation	NAME/NONAME
	CMPR2/ NOCMPR2
Control Object Code execution	DATA(24)/DATA(31)
	RENT/NORENT
	SRANGE/NOSSRANGE
	TRUNC(STD/OPT/BIN)
Control maps, listing and diagnostics	FLAG(x,[y])

Description of Important compiler options:

NAME/NONAME:

We use this option when we want to compile and linkedit multiple programs with a single invocation of the compiler. If we mention NAME as complier option in a program, it will be linkedited with the other programs previously complied in the same batch compilation. The default is NONAME.

Following example explains NAME option: In this example, there COBOL programs PROG1, PROG2 and PROG3 are coded in the SYSIN card of Catalog procedure **COB2UCL**, used for compiling COBOL code.

```
//STEPNAME EXEC   COB2UCL
//COB.SYSIN DD *
010100 IDENTIFICATION DIVISION.
010200  PROGRAM-ID PROG1.
              . . .
019000  END PROGRAM PROG1.
CBL NAME
020100 IDENTIFICATION DIVISION.
020200  PROGRAM-ID PROG2.
              . . .
029000  END PROGRAM PROG2.
030100 IDENTIFICATION DIVISION.
030200  PROGRAM-ID PROG3.
              . . .
039000  END PROGRAM PROG3.
/*
//LKED.SYSLMOD DD DSN=&&GOSET
/*
//P2 EXEC   PGM=PROG2
//STEPLIB DD DSN=&&GOSET,DISP=(SHR,PASS)
             (other DD statements and input required
             to execute PROG1 and PROG2)
/*
//P3 EXEC PGM=PROG3
//STEPLIB DD DSN=&&GOSET,DISP=(SHR,PASS)
             (other DD statements and input required
             to execute PROG3)
/*
```

In the above example,

- PROG2 will be link-edited with previously compiled PROG1 in the same batch compilation. The linkedited executable name will be PROG2. Although, the entry point of this load module defaults to the first program in the load module, PROG1.
- PROG3 is link-edited by itself into a load module with a name of PROG3. Since it is the only program in the load module, the entry point is also PROG3.
- In the above JCL, first step compiles and linkedits using VS COBOL II compiler catalog procedure COB2UCL. The next two steps execute executables PROG2 and PROG3 respectively.

CMPR2/NOCMPR2:

This option introduced in VS COBOL II, Release 3. It determines whether compilation results are compatible with the VS COBOL II, Release 2. The default is NOCMPR2.

DATA(24)/DATA(31):

- Use DATA(24) to have your dynamic data areas allocated from storage below 16 megabytes. Specify DATA(24) for programs running in an extended addressing environment in 31-bit mode that are passing data parameters to programs in 24-bit mode. Otherwise, the data may not be addressable by the called program.
- Use DATA(31) to have your dynamic data areas allocated from unrestricted storage. Specifying this option can result in the acquisition of storage in virtual addresses either above or below 16 megabytes. The operating system generally satisfies the request by providing space in virtual addresses above 16 megabytes, if it is available. The default is DATA(31) .

RENT/NORENT:

RENT option makes the program re-entrant. This means that one copy of a program can be used to satisfy multiple requests. So, if several users will be requesting a program at the same time, it should be compiled with the RENT option. Also, if a program is to be executed above 16MB, it must be re-entrant. Default is NORENT.

SSRANGE/ NOSSRANGE:

SSRANGE checks for out of range conditions during program execution. If a subscript, index, or OCCURS DEPENDING ON clause refers to an ENTRY beyond the end of a table, an error message is displayed and the program continues. It's always advisable to use SSRANGE while testing the program that uses a subscript, index, or OCCURS DEPENDING ON clause. We can remove the SSRANGE once the testing is over or code the NOSSRANGE run time option.

TRUNC(STD/OPT/BIN):

This option introduced in VS COBOL II, Release 3. It determines how results, stored in a binary receiving field, are truncated. The default is TRUNC(STD).

- When TRUNC(STD) is specified, the sending data is truncated to six integer digits to conform to the PICTURE clause of the BINARY receiver.
- When TRUNC(OPT) is specified, the compiler assumes the sending data is not larger than the PICTURE clause precision of the BINARY receiver. The most efficient code sequence in this case performed truncation as if TRUNC(STD) had been specified.
- When TRUNC(BIN) is specified, no truncation occurs because all of the sending data will fit into the binary fullword allocated for BIN-VAR.

Note: TRUNC(BIN) is the recommended option when interfacing with other products that may place values in COBOL binary data items that are larger than that defined by the data item's PICTURE clause (such as DB2, FORTRAN, and PL/I).

FLAG(X[,Y]):

The values X and Y can be 'I' (Informational: RC=0), 'W' (Warning: Return Code= 4), 'E' (Error: RC=8), 'S' (Sever: RC=12) and 'U' (Unrecoverable: RC=16). FLAG indicates for what severity level the compiler will show message in diagnostics listing (x) and the source listing (y).

For compiler option FLAG(I,W), a message will be printed in the diagnostics listing (Sysout message after compilation) for a statement causing for RC = 0 (Informational) and above, and a message will be imbedded in the source listing (Sysout message after compilation) for a statement causing RC = 12 (Sever) and above.

Below is the portion of Sysout message of compilation job to show the message in Source listing and Diagnostics listing for compiler option FLAG(I,W).

```
000009                 01 WS-VAR1-DEC              PIC 9(19) COMP-3.
==000009==> IGYDS1145-S More than 18 digit positions were specified in a
"PICTURE" string for a numeric or numeric edited item.  Check the
repetition factors.  A "PICTURE" string of "S9(18)" was assumed.
  000010                 01 WS-VAR1-DISP             PIC 9(9).

  . . .                                                    Source listing

LineID  Message code  Message text                      Diagnostics Listing

 2  IGYSC3002-I   A severe error was found in the program.  The
"OPTIMIZE" compiler option was cancelled.
 9  IGYDS1145-S   More than 18 digit positions were specified in a
"PICTURE"  string for a numeric or numeric edited item.  Check the
repetition factors.  A "PICTURE" string of "S9(18)" was assumed.
 13     RW-001-I   No Report Writer data entries were found in this
program.

Messages    Total    Informational    Warning    Error    Severe
Terminating
Printed:        3            2                                   1
* Statistics for COBOL program PICTURE1:
*    Main source records =  21
*    Intermediate records = 27
*    Data Division statements = 4
*    Procedure Division statements = 6
End of compilation 1,  program PICTURE1,  highest severity 12.
Return code 12
```

8.4. Compiler Listing

When the compiler finishes processing your source program, it lists the following results (depending on the compiler options, you selected) in the output log (SYSOUT). This is called Compiler Listing.

Result	Compiler Option
Listing of your source program	SOURCE
List of errors the compiler discovered in your program	FLAG
Listing of object code in machine and assembler language	LIST
Map of the data items in your program	MAP
Map of the relative addresses in your object code	OFFSET
Sorted cross-reference listing of procedure-, program-, and data-names	XREF
A system dump, if compilation ended with abnormal termination (requires SYSUDUMP, SYSABEND, or SYSMDUMP DD statement)	DUMP

NOTE: Listing output from compilation will be in the data set defined by SYSPRINT; object output will be in SYSLIN. Progress and diagnostic messages may be directed to the SYSTERM data set, as well as included in the SYSPRINT data set.

Source listing:

Source listing, as its name implies, is a listing of your source program.

Cross-reference listing:

Cross-reference listing is a sorted listing of data names and procedure (i.e. paragraph) names used in your programs. It will show the line number, where a data-field is DEFINED and the line number where the field value gets MODIFIED ('M' is placed before line number). Likewise, for a procedure (i.e. paragraph), it will show in which line the paragraph is DIFINED and where it is PERFORMED ('P' is placed before the line number).

Below is the portion of Sysout message after compilation to clarify the Source list and Cross-reference listing.

```
000004                IDENTIFICATION DIVISION.
000005                PROGRAM-ID. VSMREAD.
. . .
000020                01 EMP-REC.
000021                   05 EMP-SSN              PIC 9(9).
000022                   05 EMP-ORG              PIC X(26).
000023                   05 EMP-ID               PIC 9(02).
. . .
000035                      MOVE 2 TO EMP-ID
000036                      START IN-FILE KEY >= EMP-ID
000037                      DISPLAY 'KEY IS READ'
000038                      PERFORM 2000-READ-FILE UNTIL WS-EOF IS 'Y'
. . .
000042                2000-READ-FILE.
000043                    READ IN-FILE NEXT RECORD
000044      1             AT END MOVE 'Y' TO WS-EOF
000045      1             NOT AT END DISPLAY EMP-SSN.
. . .
PP 5688-197 IBM COBOL for MVS & VM   1.2.2                    V MREAD
Date
An "M" preceding a data-name reference indicates that the data-name is
modified by this reference.

 Defined   Cross-reference of data names   References

      30   ABEND-INFO
      25   EMP-BDATE
      23   EMP-ID . . . . . .  13 M35 36
. . .

PP 5688-197 IBM COBOL for MVS & VM   1.2.2                    V MREAD
Date
Context usage is indicated by the letter preceding a procedure-name
reference.
These letters and their meanings are:
    A = ALTER (procedure-name)
    D = GO TO (procedure-name) DEPENDING ON
    E = End of range of (PERFORM) through (procedure-name)
    G = GO TO (procedure-name)
    P = PERFORM (procedure-name)
    T = (ALTER) TO PROCEED TO (procedure-name)
    U = USE FOR DEBUGGING (procedure-name)

 Defined   Cross-reference of procedures   References

      33   1000-FIRST
      42   2000-READ-FILE . . . . . .  P38
```

Let's look at the shaded lines. The cross reference '23 EMP-ID . . 13 M35 36' indicates that EMP-ID is defined at line 23, and its value is modified at line 35 and referenced at line 36. The cross reference "42 2000-READ-FILE . . P38" indicates the paragraph 2000-READ-FILE is defined at line 42 and PERFORMED at line 38.

8.5. Static and Dynamic Call.

When a subprogram is called, it may already be in main storage having been link-edited in the same load module with the calling program (static call). Or it may be loaded only at the time it is called (dynamic call). With dynamic loading, the called program is loaded only when it is needed. The link-edit process differs, depending on whether your program uses static calls or dynamic calls.

8.5.1. STATIC CALL

A static call occurs when you use the CALL statement in a program that is compiled with the **NODYNAM** compiler option. Regardless of whether it is called or not, a statically called program is loaded into storage

8.5.2. DYMANIC CALL

A dynamic call occurs when you use the CALL statement in a program compiled with the **DYNAM** compiler option. In this case the CALLing and CALLed programs are each processed separately by the link-editor. A dynamically called program is loaded only when it is called at run time. Use a dynamic call statement when you are concerned about ease of maintenance. Applications do not have to be re-link-edited when dynamically called subprograms are changed.

The user should consider the following things while using DYNAMIC call:

- JOBLIB/STEPLIB of the JCL to run the program should contain all the libraries where the needed programs (to be called Dynamically) are located. If, at run-time a need program is not found, you will receive an 806 abend. The problem can be corrected by locating the appropriate load library and making the correction to your JOBLIB/STEPLIB (Remember JOBLIB and STEPLIB are mutually exclusive).

- If the module is too big to fit into memory (at run-time) you will receive an 80A abend. The problem can be corrected by increasing the REGION parameter in JCL.

- The above mentioned ABENDs can be avoided using ON EXCEPTION/NOT ON EXCEPTION clauses of CALL statement

```
CALL PGM-A USING DATA-NAME-A
        ON EXCEPTION
            CANCEL PGM-B
    CALL PGM-A USING DATA-NAME-A
            END-CALL
        END-CALL
```

Here, if PGM-A throws Exception because of memory problem while getting loaded in to memory, the code will delete the PGM-B from memory and then again try to load PGM-A.

8.5.3. Comparison between STATIC and DYNAMIC Call

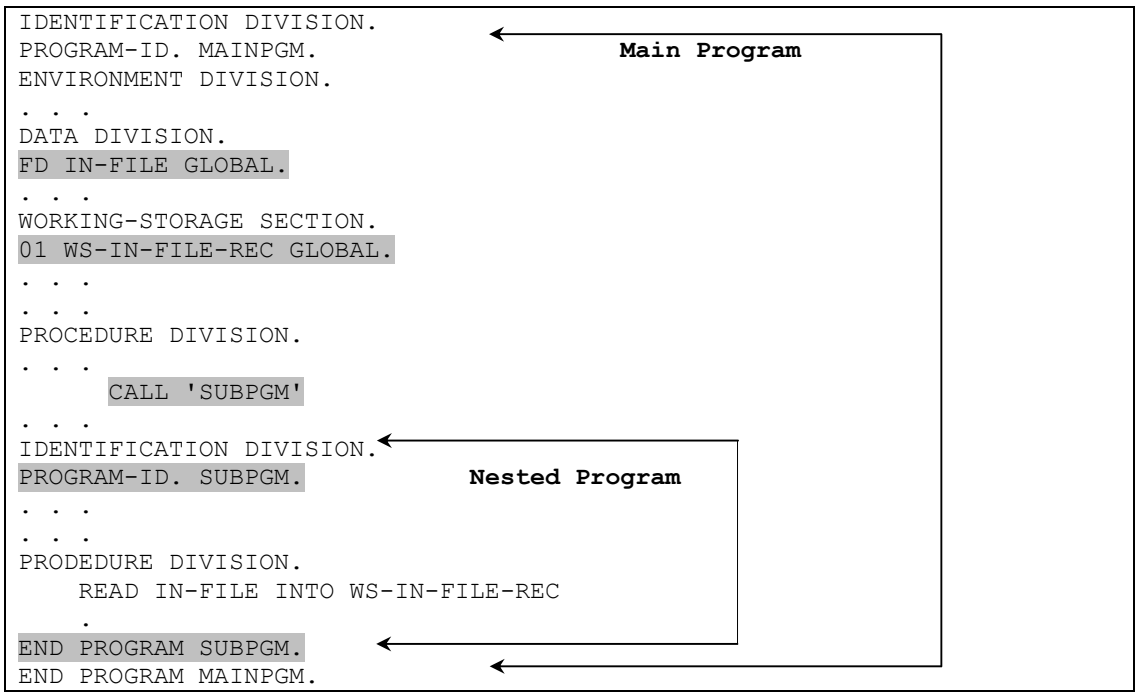
- Dynamic calls take more processing than static calls. However, a dynamic call may use less total storage than a static call.
- A statically called program cannot be deleted, but a dynamically called program can be deleted. Using a dynamic call and then a CANCEL statement to delete the dynamically called program after it is no longer needed in the application (and not after each CALL to it) may require less storage than using a static call.
- Use Dynamic CALL if the subprograms are very large or less frequently used (called only on a few conditions). In such case, the use of static calls might require too much main storage. Less total storage may be required to call and cancel one, then call and cancel another, THAN to statically call both.

8.6. Global and External Data Elements.

These two data elements come to use when programs in the same run unit share, or access to, common files. Before going into detail, we need to know the concept of Nested COBOL program.

8.6.1. Nested Program

A COBOL program may contain other COBOL programs within itself. The contained programs may themselves contain yet other programs. A contained program may be directly or indirectly contained within a program.
Let's take a small example of Nested program structure.



There are several conventions that apply when using nested program structures.

- The Identification Division is required in each program. All other divisions are optional.
- Program names must be unique.
- Contained program names may be any valid COBOL word or a nonnumeric literal.
- Contained programs cannot have a Configuration Section. The outermost program must specify any Configuration Section options that may be required. Each contained program is included in the containing program immediately before its End Program header.
- Contained and containing programs must be terminated by an End Program header.

8.6.2. GLOBAL Data elements

A name that is specified as global (by using the Global clause) is visible and accessible to the program in which it is declared, and to all the programs that are directly and indirectly contained within that program. This allows the contained programs to share common data and files from the containing program, simply by referencing the name of the item. Any item that is subordinate to a global item (including condition names and indexes) is automatically global. The same name may be declared with the Global clause multiple times, provided that each

declaration occurs in a different program. The above example (for Nested program) shows the use of GLOBAL data elements.

8.6.3. EXTERNAL Data elements

Using the EXTERNAL clause for files allows separately compiled programs within the run unit to have access to common files. The following example shows the use of EXTERNAL data item:

```
PROGRAM-ID. MAINPGM.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
      COPY CPYSEL.
DATA DIVISION.
FILE SECTION.
     COPY CPYFD.
WORKING-STORAGE SECTION.
     COPY CPYWS.
PROCEDURE DIVISION.
    OPEN INPUT IN-FILE
    CALL 'SUBPGM'
    . . .
END PROGRAM MAINPGM.
```

```
PROGRAM-ID. SUBPGM.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
      COPY CPYSEL.
DATA DIVISION.
FILE SECTION.
     COPY CPYFD.
WORKING-STORAGE SECTION.
     COPY CPYWS.
PROCEDURE DIVISION.
    READ IN-FILE
    . . .
    . . .
END PROGRAM SUBPGM.
```

Content of COPYBOOKs used in the above two programs:

CPYSEL:

```
SELECT IN-FILE
       ASSIGNED TO INFILE
       FILE STATUS IS WS-INFILE-STAT.
```

CPYFD:

```
FD IN-FILE EXTERNAL
       RECORD CONTAINS 200 CHARACTERS
       RECORDING MODE IS F.
01 FD-INFILE-REC.
   05 FD-EMP-ID   PIC X(9).
```

CPYWS:

```
01 WS-STAT      EXTERNAL.
   05 WS-INFILE-STAT   PIC X(2).
```

The main program can reference the record area of the file, even though the main program does not contain any I/O request. Each subprogram can control a single I/O function, such as OPEN or READ. Each program has access to the file.

The above example also illustrates that the following items are recommended to successfully process an EXTERNAL file:

- The file-name in the SELECT clause of all the programs accessing the file must match.
- The assignment-name in the ASSIGN clause of all the programs accessing the file must match.
- EXTERNAL must be coded in the file's FD entry in all the programs accessing the file. In the above program this is maintained by using same copybook (CPYFD) in both calling and called programs.
- For all programs that want to check the same file status field, the EXTERNAL clause must be coded on the level-01 data definition for the file status field in each program. In the above program this is maintained by using same copybook (CPYWS) in both calling and called programs.

8.7. Debugging tools and Techniques.

Let's discuss about debugging tool available in VS COBOL II. VS COBOL II Debug tool can be used in either batch or interactive mode.

In Interactive mode we need to do proper setting to load the source listing onto interactive debugger panel and to put breakpoints to stop the execution temporarily or to execute the code line by line to see the value of the fields at a give point of execution state.

In batch mode, we run a JCL using VS COBOL II debug program **COBDBG** mentioning the required option to print the value of the fields (onto SYSOUT) at a give point of execution state. In batch mode we cannot control the execution the code line by line but we can do everything (what we do in case of interactive debugging) to debug the code. Below is the description of how we debug a COBOL program in batch mode (Mainframe environment).

STEP 1

Compile the code using compiler option '**TEST**'. If Compile panel (in your mainframe) does not provide facility to set compiler option 'TEST', you may use the Compile (and link-edit) JCL captured in **section [7.2 Compiler Option]**.

STEP 2

Run the following JCL to execute your COBOL program COBPGM1 with the required debugging options.

```
//TR99999C JOB ,,NOTIFY=TR99999,CLASS=B,MSGLEVEL=(1,1),MSGCLASS=X
//DEBUG  EXEC  PGM=COBDBG
//*********************************************************************
//* INCLUDE REQUIRED LIBRARY IN STEPLIB TO PICK UP EXECUTABLE FOR COBBUG *
//*********************************************************************
//STEPLIB  DD DSN=SYS1.COB2LIB,DISP=OLD
//         DD DSN=OPERN.CICS1.LOADLIB,        <-- Stores the executable
//            DISP=OLD                           of COBPGM1
//SYSABOUT DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSDBOUT DD SYSOUT=*
//SYSDBIN  DD *
COBTEST COBPGM1
RECORD
QUALITY COBPGM1
SET WS-EMP-ID = 2
TRACE PARA PRINT
FLOW ON
AT COBPGM1.21 (IF (WS-EMP-ID NOT = 2),
(LIST WS-EMP-ID))
ONABEND (FLOW(5))
/*
```

Let's see the meaning of each lines of the above JCL.

SYSABOUT

It lists any VS COBOL II ABEND intercept information. The data set format for SYSABOUT is LRECL=125,RECFM=VBA.

SYSDBOUT

It lists any output from FDUMP, if specified. The dataset format for this is LRECL=121, RECFM=FBA

SYSPRINT

It captures the compiler listing if some particular compiler options (e.g. LIST, OFFSET etc) are specified.

SYSDBIN

The debugging options are mentioned in the in-stream card of DD name SYSDBIN.

- **<COBTEST *executable-name*>** tells COBTEST the name of the executable (link-edited member) to load for execution.
- **<RECORD>** tells COBTEST to create a log on SYSDBOUT of all commands.
- **<QUALIFY *program-name*>** tells COBTEST the name of the PROGRAM-ID of the program we are debugging. If we are debugging a subprogram, the <COBTEST *executable-name*> will contain the main program name where as <QUALIFY *program-name*> will contain the subprogram name.
- **<SET WS-EMP-ID = 2>** sets the initial value of the working storage filed WS-EMP-ID to 2.
- **<TRACE PARA PRINT**> specifies that a trace of paragraph names is to be printed (default is to SYSDBOUT).
- **<AT COBPGM1.21 (IF (WS-EMP-ID NOT = 2), (LIST WS-EMP-ID))**> specifies that when 21st line executes, print the value of WS-EMP-ID if at that point the value of WS-EMP-ID be other than 2.
- ONABEND (FLOW(5)) specifies the if an ABEND occurs, print the list of most recent 25 paragraphs executed.

Some debugging technique using COBTEST in batch mode:

2. If you are getting an ABEND but unable to pinpoint where (in which line) exactly it is happening, use the following options for COBTEST.

```
//SYSDBIN DD *
COBTEST executable-name
RECORD
QUALIFY Program-name
FLOW ON
ONABEND (FLOW (15))
/*
```

   This will print a list of the 15 most recently executed paragraphs (onto the SYSOUT) prior to the ABEND.

3. Debugging a subprogram:

   Below in the structure of linkage section of a subprogram **SUBPGM1** of a main program **MAINPGM1**.

```
LINKAGE SECTION.
01  LS-REC.
    05  LS-EMP-ID   PIC 9(2).
    05  LS-ERROR-CD PIC 9.
. . .
PROCEDURE DIVISION USING LS-REC.
```

Here we don't want to let the execution start from the entry point of the main program. We want to pass some hardcoded data (per se LS-EMP-ID = 50) to linkage section. In this case we will frame our COBTEST script in the following way.

```
//SYSDBIN DD *
COBTEST MAINPGM1
RECORD
QUALIFY SUBPGM1
LINK (LS-REC)
SET LS-EMP-ID = 50
TRACE PARA PRINT
WHEN TST1 (LS-ERROR-CD NOT = 0) (LIST ALL)
FLOW ON
ONABEND (FLOW (15))
/*
```

Whenever LS-ERROR-CD becomes non-zero during the execution of SUBPROG1, the COBTEST will list the entire DATA DIVISION.

Here is the sample output (portion of SYSOUT message) from the above script, as written to SYSDBOUT.

```
RECORD
QUALIFY SUBPGM1                                    Printed by RECORD
statement
LINK (LS-REC)
SET LS-EMP-ID = 50
TRACE PARA PRINT
WHEN TST1 (LS-ERROR-CD NOT = 0) (LIST ALL)
FLOW ON
ONABEND (FLOW (15))
IGZ100I PP - 5668-958 VS COBOL II DEBUG FACILITY - REL3.2
IGZ101I © COPYRIGTH IBM CORPORATION
IGZ102I SUBPGM1.000012.1
. . .
IGZ106I TRACING SUBPGM1
. . .                                  Program Trace
IGZ106I 000019.1
IGZ106I 000021.1
IGZ106I WHEN TST1 SUBPGM1.000015.1
0000019 01 SUBPGM1.LS-REC AN-GR          Output from WHEN statement
0000020 02 SUBPGM1.LS-EMP-ID 99
. . .
IGZ120I PROGRM UNDER COBTEST ENTED NORMALLY
. . .
IGZ125I **************END OF COBTEST********************
```

Note: SUBPGM1.000015.1 gets printed in the format of **<Program-name>.<line-num>.<number of COBOL verb present in that line>**

9 Good Coding practice

OVERVIEW

1. Performance Issue and Good coding practice
2. Data Validation
3. Code walkthrough

9.1. Performance Issues and Good Coding practice

The thumb rule for good coding:

- Emphasize on neatness and on preventing the program from becoming dense.
- Consider some of the coding techniques that have an influence on the actions of the OPTIMIZER to improve the performance.

First let's see what OPTIMIZER does at a high level.

9.1.1. OPTIMIZER

The VS COBOL II OPTIMIZER is activated by specifying the OPTIMIZE compiler option. To see how the OPTIMIZER works on your program, compile it with and without the **OPTIMIZE** compiler option and then compare the generated code. Below are the functions of OPTMIZER.

- Eliminate unnecessary transfers of control or simplify inefficient branches that are not evident from looking at the source program.
- Simplify the compiled code for both a PERFORM statement and a CALL statement to a contained (nested) program. Where possible, the OPTIMIZER places the statements in-line, eliminating the need for linkage code.
- Eliminate duplicate computations (such as subscript computations) by saving the results for later reuse.
- Aggregate moves of contiguous items (such as those that often occur with the use of MOVE CORRESPONDING) into a single move. Both the source and target must be contiguous for the moves to be aggregated.
- Delete code that can never be executed (unreachable code elimination) from the program

9.1.2. Good Coding Practice

Let's discuss some techniques improve the performance of code.

- Use Structured Programming features and top down approach.

- Use the following preferred COBOL Statements:
  1. EVALUATE: Avoid more that 3 level of nested IF-ELSE statement. Use EVALUATE in such case.
  2. COMPUTE: Usually more efficient than other arithmetic statements, especially when more than one arithmetic operation takes place.
  3. INITIALIZE: If you group common items together, such as accumulators, the INITIALIZE can reset all entries under a group name, numeric field to zero and alphanumeric fields to spaces.

- Avoid the following COBOL statements:
  1. ALTER.
  2. GO TO.
  3. PERFORM THRU. This requires a minimum of two paragraphs, and encourages the use of GO TO.
  4. EXIT.
  5. ENTRY. The ENTRY statement gives a program more than one entry point. Since this is a violation of structured Programming, avoid its use. If a subprogram requires multiple entry points, write multiple subprograms instead.
  6. NEXT SENTENCE. This is an implied GO TO. Use another approach such as CONTINUE statement of rewrite the sentence so NEXT SENTENCE is not required.

- Use 'NOT' carefully. When possible use a positive test, not a negative one. Also, remember that 'NOT' combined with 'OR' is always TRUE. Consider, "IF A NOT = 1 OR 2" is always TRUE condition, because A will always be unequal to one of those values.

- Choose USAGE clause wisely to eliminate unnecessary data conversion. The default for a numeric item is DISPLAY. Using DISPLAY items (external decimal or external floating point) for computation is inefficient, because it requires conversion each time the data is used in a calculation. The hardware operations for manipulating binary data are faster than those for manipulating decimal data. So, when defining numeric items that are used only for internal calculations, define them as BINARY (USAGE COMP) or PACKED-DECIMAL (USAGE COMP-3); when defining floating point items that are used only for internal calculations, define them as COMP-1 or COMP-2.

- For a BINARY data item, the most efficient code is generated if the item has:
  1. A sign (S in its PICTURE clause)
  2. 8 digits or fewer.

- When a data item is used for arithmetic, but is larger than 8 digits or is also used with DISPLAY data items, then PACKED-DECIMAL (same as COMP-3) is a good choice. The code generated for PACKED-DECIMAL data items can be as fast as that for BINARY data items in some cases, especially if the statement is complicated or specifies rounding. For a PACKED-DECIMAL data item, the most efficient code is generated if the item has:
  1. A sign (S in its PICTURE clause).
  2. An odd number of digits (9s in the PICTURE clause), so that it occupies an exact number of bytes without a half-byte left over

3. 15 digits or fewer, because a library routine call must be used to multiply or divide larger items.

- Exponentiations with large exponents can be evaluated more quickly and with more accurate results using floating-point.

  For example, <**COMPUTE fixed-point1 = fixed-point2 ** 100000>** could be computed more quickly and accurately if it is coded as <**COMPUTE fixed-point1 = fixed-point2 ** 100000.E+00**> since the presence of a floating-point exponent would cause floating point to be used to compute the exponentiation.

- Use Factoring Expressions. Factoring can save a lot of computation.

  For example, this code:

```
MOVE ZERO TO TOTAL
PERFORM VARYING I FROM 1 BY 1 UNTIL I = 10
  COMPUTE TOTAL = TOTAL + ITEM(I)
END-PERFORM
COMPUTE TOTAL = TOTAL * DISCOUNT
```

  is more efficient than the following code:

```
MOVE ZERO TO TOTAL
PERFORM VARYING I FROM 1 BY 1 UNTIL I = 10
  COMPUTE TOTAL = TOTAL + ITEM(I) * DISCOUNT
END-PERFORM
```

  The OPTIMIZER does not do factoring for you.

- When using VSAM files, increase the number of data buffers for sequential access or index buffers for random access. Also, select a control interval size (CISZ) appropriate for the application; smaller CISZ results in faster retrieval for the random processing at the expense of inserts, whereas a larger CISZ is more efficient for sequential processing. For better performance, access the records sequentially and avoid using multiple alternate indexes when possible.

9.1.3. Coding Guidelines

Below are some of the guidelines to write readable and maintainable COBOL code.

- DIVISION statement should be preceded and followed by at least two blank lines. SECTION statements should be preceded and followed by at least one blank line. Paragraphs, FDs, SDs and 01-levels should be preceded by at least one blank line.

- Margin A entries (FD, SD, DIVISION, SECTION, Paragraph names, 01- level entries) must begin in column 8 and margin B entries (Procedural statements, subordinate data fields (e.g. 05-level fields)) must begin in column 12.

- For a group variable, subordinate entries are indented 4 columns to the right of their parent entry. Subordinate entries should begin at level 05 and increment by 5.

- User prefixes 'FS-' and 'WS-' for the variable Names in FILE-SECTION and WORKING-STORAGE section respectively. The variable name should be meaningful such as

```
01  WS-DB2-COMMIT-FLAG        PIC  X(01)
                               VALUE 'N'.
   88  SUCCESSFUL-DB2-COMMIT  VALUE 'Y'.
   88  PENDING-DB2-COMMIT     VALUE 'N'.
```

  The variable names should not be short like I, J or K.

- Try to reduce the number of 01-level variables. Put same category of variables (per se Amount fields or Flags or Counters) under one 01-level field.

- Start all the PIC statements (used in a program) at the same column (per se 45th column). Start all the VALUE statements (used in a program) at the same column (per se 45th column) if possible.

- Use Period (.) ONLY at the end of paragraphs, NOT at any other place.

- Use Scope-terminator such as END-IF, END-PERFORM, END-READ, END-CALL etc but never use Period (.) to terminate the IF, PERFORM etc.

- Indent conditionally executed statements at least 4 columns beyond invoking statements. Place scope terminators and ELSE statements in the same column as pervious statement to which it relates. For example,

```
IF SUCCESSFUL-CALL
   SET PROCESS-ITEM TO TRUE
ELSE
   PERFORM 2200-UPDATE-CUSTOMER
END-IF
```

- Start all the TO and FROM statements at the same column (per se 41 column) through the program.

- Avoid hardcoding inside PROCEDUE DIVISION. If hardcoding is necessary, assign them using VALUE clause against some variable in WORKING-STORAGE section and use that variable in the PROCEDUE DIVISION.

- Use adequate comments before variable declaration, Initialization, paragraph and complicated logic. The Comment should be placed in a form of box with '*' lines, as shown below.

```
*****************************************************************
*   -EVALUATE SQLCODE                                           *
*   -IF SUCCESSFUL EXECUTION OF ROW NOT FOUND                   *
*        CONTINUE PROCESSING                                    *
*   -ELSE                                                       *
*       ROLLBACK WORK, WRITE ERROR MESSAGE & ABEND              *
*****************************************************************
```

- Use meaningful and uniform naming convention for PARAGRAPH. See the example below.

```
PROCEDURE DIVISION.
0000-MAINLINE.
    PERFORM 1000-INITIALIZATION.
    PERFORM 2000-RCRT-FOLLOWUP-PROCESS
    GOBACK.
1000-INITIALIZATION.
    PERFORM 1100-GET-PROGRAM-DATES
    PERFORM 1200-GET-HOLDQ-INFO.
    . . .
1100-GET-PROGRAM-DATES.
     PERFORM 1110-GET-PROCESS-DATE.
     . . .
2000-RCRT-FOLLOWUP-PROCESS.
     PERFORM 2100-GET-NEXT-ITEM
          . . .
```

  Here two points need to be followed.

  1. The paragraph name should be physically places in order of their sequence number.
  2. The paragraphs with prefix 1000- should call only the paragraph with prefix 1n00-, where n >=1 to 9. This rule can be violated only if you are calling some multiple used paragraphs (per se 9999-ABEN-PARA). This is how to maintain readability for a big COBOL program.

- All the fields (except linkage section fields, FD section fields and fields used in the variable section for a variable records using OCCURS..DEPENDING ON) should be initialised at the beginning of the program.

- The diagnostics (informational) display statement should not be used inside a loop for the actual production code (for batch program). This may lead to Sysout overflow if the loop is processing huge number of transactions.

- Use minimum number of flags (switches). Too many flags always create confusion.

- For a batch program, the program statistics (per se Number of transaction processed, number of error items, number of roll-backed records) should be displayed at the end of the main program to get the clear picture about the processing.

- All I/O requests should be handled using standard error handling logic. That means return-code should be checked after opening/reading/writing each file and firing each SQL statements. For any unwanted Exception make a call to standard abend routine to abend the job. We can use DECLARATIVES for this purpose.

9.2. Data Validation

The compiler assumes that the values you supply for a data item are valid for the item's PICTURE and USAGE clause and assigns the value you specify without checking for validity. Frequently, values are passed into your program and assigned to items that have incompatible data descriptions for those values. For example, nonnumeric data may be moved or passed into a field in your program that is defined as a numeric item. Or, perhaps a signed number is passed into a field in your program that is defined as an unsigned number. Because of the above reason the 0C4, 0C7, 0C9, and 0CB ABENDs are probably the most familiar ADENDs. Let's look at them separately.

- **0C4** : It usually caused by runaway subscript or index attempting to access unallocated memory. The majority of programs ABENDing with 0C4 have an OCCURS clause. This ABEND can be avoided by testing the range of subscript prior to use. The program testing can be supplemented by the SSRANGE compile option.

- **0C7** : It usually caused using nonnumeric data in an arithmetic statement, which can be avoided by using the IS NUMERIC test.

  While checking the numeric, we should consider the potential defect like:

```
05  DATA-NUM  PIC 9(3).
05  DATA-CHR  REDEFINES DATA-NUM  PIC X(3).
. . .
MOVE '01 ' TO DATA-CHR
IF DATA-NUM > 0 AND < 100
     ADD DATA-NUM TO
. . .
```

  The above code will generate 0C7 ABEND. Here the DATA-NUM stores an alphanumeric value (a Zero, an One and a SPACE in it's 3 bytes) and its value fall under the zero and 100 (as per EBCDIC collating sequence). Hence arithmetic operation (ADD DATA-NUM) will be triggered with an non-numeric value and generates 0C7 ABEND.

- **0C9 and 0CB**: Occurs for DIVIDE BY ZERO. This can be prevented by exception handling using ON SIZE ERROR or by validating the DIVISOR before the DIVIDE statement.

9.3. Code Walkthrough

There are some basic points the code reviewer should look for.

- The formats mentioned in the coding guidelines are followed. The formats includes, the indentation of each and every statements, naming convention of variables, paragraphs and physical placement of paragraphs.

- In case of sub program there is no STOP RUN statement.

- In case of main program there is only one STOP RUN (or GOBACK). Should check there is single entry point and single exit point.

- If PARM is used in JCL, the length should be exactly same as that mentioned in the linkage section for the program receiving the PARM.

- Inside PROCEDURE DIVISION the period (.) is used only at the end of the SECTION and PHARAGRAPH, not at any other place.

- All the fields used in the program are declared, and there is no unused variable present in the program.

- All the fields (except linkage section fields, FD section fields and fields used in the variable section for a variable records using OCCURS..DEPENDING ON) are initialized.

- The length of file record in the FD section (RECORD CONTAINS <N> CHARACTERS) should match with the record length (declared in FILE SECTION/WORKINGSTORAGE SECTION) into/from, which the file is being read/written. Also the length and organization of physical file mentioned in JCL (for batch program). File accessing mode, organization are coded correctly. An empty VSAM file should not be opened in OUTPUT mode.

- File is opened before reading/writing. The file has not been closed before reading/writing.

- If the file is of variable length, the record length of physical file should be 4 bytes more than that present inside the COBOL program.

- The possibility of DIVIDE BY ZERO should be checked. The possibility of INFINITE LOOP should be checked. Should check whether the counter (to control loop) is incremented inside the loop, if any. If the Loop is controlled by a switch, in such case check whether the control flag is set to proper value so that loop does not fall into infinite loop.

- Should check whether DO-UNTIL and DO-WHILE are properly used. Remember, DO-UNTIL checks the condition after executing the statement inside it where as DO-WHILE checks the condition before executing the statement inside it.

- For movement of numeric data, the datatype compatibility should be checked.

- The code reviewer should check where the proper datatype is used for Counter, amount fields.

- Verify whether all Control statements (IF, DOWHILE, PERFORM etc.) satisfy the boundary conditions.

- Should check whether all statements are getting executed, and there is no redundant code (dead code).

- More that 3 level of nested IF-ELSE statement should not be used.

- No diagnostics (informational) DISPLAY statement should be present inside the LOOP.

- Exception handling is incorporated for after each I/O statements (OPEN/READ/WRITE file, SQL statements); Or DECLARATIVES is used for exception handling.

- Should check all the output files (especially Sequential file) have been closed before making call to ABEND routine to abend the job programmatically for an error condition.

- IF a file is used in more that one programs (within the run unit) the file, file status and the FD entry for that file are declared as EXTERNAL.

Appendix – A

1. Instruction Formats

This appendix contains the composite language formats of the ANSI COBOL. These formats are applicable to COBOL-85 and above. The appearance of the italic letters S, R, I, or W to the left of the format for the verbs CLOSE, OPEN, READ, WRITE indicates the Sequential I-O module, Relative I-O module, Indexed I-O module, or Report Writer module in which that general format is used.

2. General Format of IDENTIFICATION DIVISION

```
IDENTIFICATION DIVISION.
                                  ┌      ┌ COMMON ┐              ┐
PROGRAM-ID. program-name          │   IS ┤        ├ PROGRAM     │
                                  └      └ INITIAL┘              ┘

[AUTHOR. [comment-entry]... ]
[INSTALLATION. [comment-entry]... ]
[DATE-WRITTEN. [comment-entry]... ]
[DATE-COMPILED. [comment-entry]... ]
[SECURITY. [comment-entry]... ]
```

3. General Format of ENVIRONMENT DIVISION

```
[ENVIRONMENT DIVISION.
[CONFIGURATION SECTION.
[SOURCE-COMPUTER.  [computer-name  [WITH DEBUGGING MODE].]]
[OBJECT-COMPUTER.  [computer-name
          [PROGRAM COLLATING SEQUENCE IS alphabet-name-1]
          [SEGMENT-LIMIT IS segment-number].]]
[SPECIAL-NAMES. [[implementor-name-1
  ┌ IS mnemonic-name-1 [ON STATUS IS condition-name-1 [OFF STATUS IS condition-name-2]] ┐
  │ IS mnemonic-name-2 [OFF STATUS IS condition-name-2 [ON STATUS IS condition-name-1]] │
 ┤ ON STATUS IS condition-name-1 [OFF STATUS IS condition-name-2]                      ├ ]…
  │ OFF STATUS IS condition-name-2 [ON STATUS IS condition-name-1]                     │
  └                                                                                   ┘
  [ALPHABET alphabet-name-1 IS
   ┌STANDARD-1                          ┐
   │STANDARD-2                          │
   │NATIVE                              │
   ┤implementor-name-2                  ├ ]…
   │┌          ┌┌THROUGH┐         ┐┐    │
   ┤┤literal-1  ┤└       ┘literal-2┘├ …  │
   ││          │└ THRU    ┘         ││    │
   └└          └{ALSO literal-3}…   ┘┘    ┘

┌                          ┌                                                          ┐ ┐
│                          ┌┌                          ┌ IS  ┐          ┐           │ │
│SYMBOLIC CHARACTERS       ┤┤ {symbolic-character-1}…  ┤     ├{integer-1}…┘ …[IN alphabet-name-2] ┘ │…
│                          ┌└                          └ ARE ┘          ┘           │ │
└                          └                                                          ┘ ┘┌
┌          ┌┌ THROUGH┐         ┐┐    ┐
│CLASS class-name IS ┤literal-4├ ┤     ├ literal-5├┘…  ├…
└                    └         └└ THRU  ┘         ┘┘   ┘
[CURRENCY SIGN IS literal-6]
[DECIMAL-POINT IS COMMA].]]]
```

```
[INPUT-OUTPUT SECTION.
FILE-CONTROL.
        {file-control-entry}…
[I-O-CONTROL.
┌┌       ┌RECORD      ┐                                          ┐   ┐
│└ SAME  │ SORT        │ AREA FOR file-name-1{file-name-2}… │ …│
└        └SORT-MERGE  ┘                                          ┘   │] ]
                                                                      │
┌┌MULTIPLE-FILE TAPE CONTAINS            ┐                         │
└└                                        │…   .                    │
        {file-name-3[POSITION IS integer-1]}… ┘                       ┘
```

4. FILE-CONTROL Paragraph

(a) SEQUENTIAL FILE

```
SELECT [OPTIONAL] file-name-1
                  ⌈implementor-name-1⌉
        ASSIGN TO⌈                   ⌉…
                  ⌊literal-1         ⌋
        ⌈                  ⌈AREA ⌉⌉
        │ RESERVE integer-1│      ⌋⌋
        ⌊           ⌊AREAS⌋⌋
        [[ORGANIZATION IS] SEQUENTIAL]
        ⌈                          ⌈data-name-1⌉⌉
        │ PADDING CHARACTER IS⌈              ⌉│
        ⌊                          ⌊literal-2  ⌋⌋
        ⌈                          ⌈STANDARD-1   ⌉⌉
        │ RECORD DELIMITER IS  ⌈                  ⌉│
        ⌊                          ⌊implementor-name-2⌋⌋
        [ACCESS MODE IS SEQUENTIAL]
        [FILE STATUS IS data-name-2].
```

(b) RELATIVE FILE

```
SELECT [OPTIONAL] file-name-1
                  ⌈implementor-name-1⌉
        ASSIGN TO⌈                   ⌉…
                  ⌊literal-1         ⌋
⌈                 ⌈AREA ⌉⌉
│ RESERVE integer-1│     ⌋⌋
⌊                 ⌊AREAS⌋⌋
[ORGANIZATION IS] RELATIVE
⌈                 ⌈SEQUENTIAL [RELATIVE KEY IS data-name-1] ⌉⌉
│                 │                                          ⌋⌋
│ ACCESS MODE IS ⌈⌈RANDOM ⌉                                  ⌉│
│                 │⌈        ⌉ RELATIVE KEY IS data-name-1    ⌋⌋
⌊                 ⌊⌊DYNAMIC⌋                                 ⌋⌋
[FILE-STATUS IS data-name-2]
```

(c) INDEX FILE

```
SELECT [OPTIONAL] file-name  ASSIGN TO implementor-name-1⌉
```

```
[                    [AREA ]]
| RESERVE integer-1|        | |
[           [AREAS]]
[ORGANIZATION IS] INDEXED
[                    [SEQUENTIAL]]
| ACCESS MODE IS{RANDOM       }|
[                    [ DYNAMIC   ]]
RECORD KEY IS data-name-1
[ALTERNATE RECORD KEY IS data-name-2 [WITH DUPLICATES]]…
[FILE STATUS IS data-name-3].
```

5. SORT OR MERGE FILE

```
                                [implementor-name-1]
SELECT file-name-1 ASSIGN TO{                       }…
                                [literal-1          ]
```

6. REPORT FILE

```
SELECT [OPTIONAL] file-name-1
                    [implementor-name-1]
        ASSIGN TO {                     }…
                    [literal-1          ]
        [                    [AREA ]]
        | RESERVE integer-1|        | |
        [           [AREAS]]
        [[ORGANIZATION IS] SEQUENTIAL]
        [                          [data-name-1]]
        | PADDING CHARACTER IS{                 }|
        [                          [ literal-2  ]]
        [                       [STANDARD-1      ]]
        | RECORD DELIMITER IS{                     }|
        [                       [ implementor-name-2]]
        [ACCESS MODE IS SEQUENTIAL]
        [FILE STATUS IS data-name-2].
```

7. General format of Data Division

```
[DATA DIVISION.
[FILE SECTION.
[file-description-entry
{record-description-entry}…]…
[sort-merge-file-description-entry
{record-description-entry}…]…
[report-file-description-entry]…]
[WORKING-STORAGE SECTION.
[77-level-description-entry]…]
[ record-description-entry  ]
[LINKAGE SECTION.
[77-level-description-entry]…]
[ record-description-entry  ]
[COMMUNICATION SECTION.
[communication-description-entry
[ record-description-entry]…]…]
[REPORT SECTION.
[report-description-entry
{record-group-description-entry}…]…]]
```

7.1. File Description Entry

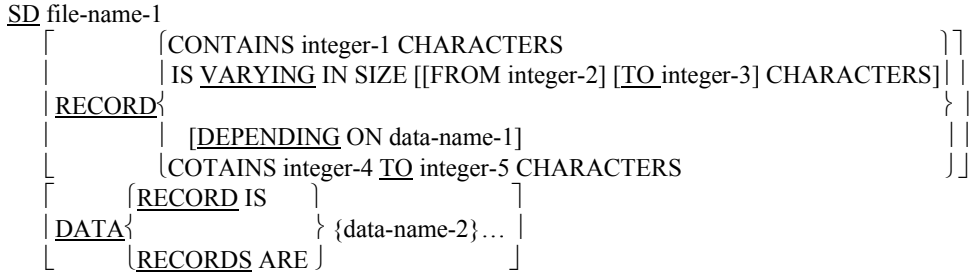
(a) SEQUENTIAL FILE

```
FD file-name-1
    [IS EXTERNAL]
    [IS GLOBAL]
    ┌                                          ⌈RECORDS    ⌉┐
    │ BLOCK CONTAINS [integer-1 TO] integer-2{               }│
    └                                          ⌊CHARACTERS⌋ ┘
    ┌          ⌈CONTAINS integer-3 CHARACTERS                    ⌉ ┐
    │          │IS VARYING IN SIZE[[FROM integer-4] [TO integer-5] CHARACTERS]│ │
    │ RECORD{                                                     } │
    │          │  [DEPENDING ON data-name-1]                      │ │
    └          ⌊CONTAINS integer-6 TO integer-7 CHARACTERS        ⌋ ┘
    ┌       ⌈RECORD IS    ⌉ ⌈STANDARD⌉ ┐
    │ LABEL{               }{          }│
    └       ⌊RECORDS ARE⌋ ⌊OMITTED  ⌋ ┘
    ┌          ⌈                  ⌈data-name-2⌉ ⌉   ┐
    │ VALUE OF{ implementor-name-1 IS{           }}… │
    └          ⌊                  ⌊literal-1   ⌋ ⌋   ┘
    ┌       ⌈RECORD IS   ⌉                  ┐
    │ DATA{               } {data-name-3}… │
    └       ⌊RECORDS ARE⌋                  ┘
    ┌          ⌈data-name-4⌉      ⌈                  ⌈data-name-5⌉ ⌉     ┐
    │ LINAGE IS{           } LINES │ WITH FOOTING AT{           }│     │
    │          ⌊integer-8  ⌋      ⌊                  ⌊integer-9  ⌋ ┘     │
    │                                                                    │
    │      ⌈                ⌈data-name-6⌉ ⌉⌈                  ⌈data-name-7⌉ ⌉ │
    │      │ LINES AT TOP{           }│ │LINES AT BOTTOM{           }│ │
    └      ⌊                ⌊integer-10 ⌋ ⌋⌊                  ⌊integer-11 ⌋ ⌋ ┘
   [CODE-SET IS alphabet-name-1].
```

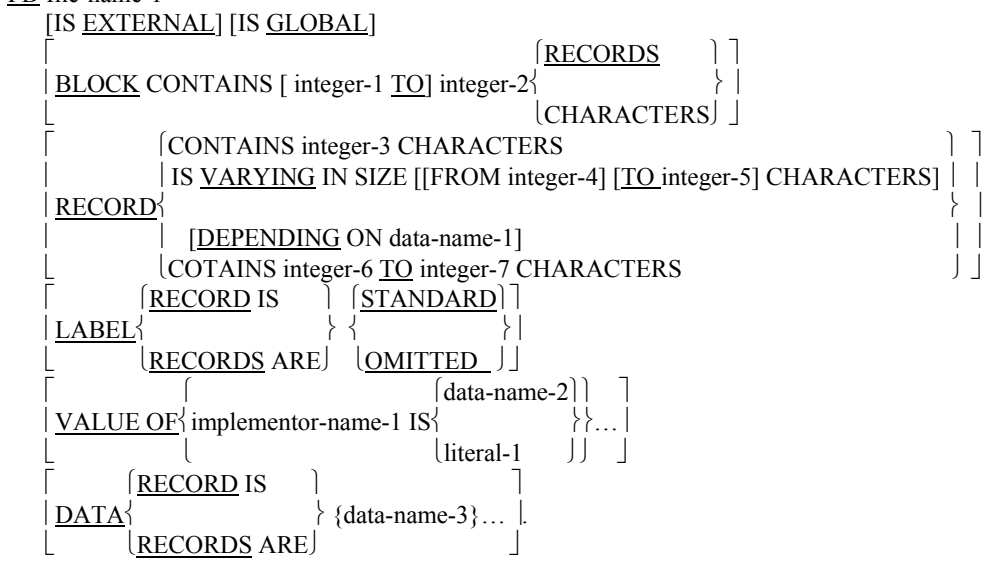
(b) RELATIVE FILE

```
FD file-name-1
    [IS EXTERNAL]
    [IS GLOBAL]
    ┌                                          ⌈RECORDS    ⌉ ┐
    │ BLOCK CONTAINS [ integer-1 TO] integer-2{               } │
    └                                          ⌊CHARACTERS⌋ ┘
    ┌          ⌈CONTAINS integer-3 CHARACTERS                    ⌉ ┐
    │          │IS VARYING IN SIZE [[FROM integer-4] [TO integer-5] CHARACTERS]│ │
    │ RECORD{                                                     } │
    │          │  [DEPENDING ON data-name-1]                      │ │
    └          ⌊COTAINS integer-6 TO integer-7 CHARACTERS         ⌋ ┘
    ┌       ⌈RECORD IS    ⌉ ⌈STANDARD⌉ ┐
    │ LABEL{               }{          }│
    └       ⌊RECORDS ARE⌋ ⌊OMITTED  ⌋ ┘
    ┌          ⌈                  ⌈data-name-2⌉ ⌉   ┐
    │ VALUE OF{ implementor-name-1 IS{           }}… │
    └          ⌊                  ⌊literal-1   ⌋ ⌋   ┘
    ┌       ⌈RECORD IS   ⌉                  ┐
    │ DATA{               } {data-name-3}… │.
    └       ⌊RECORDS ARE⌋                  ┘
```

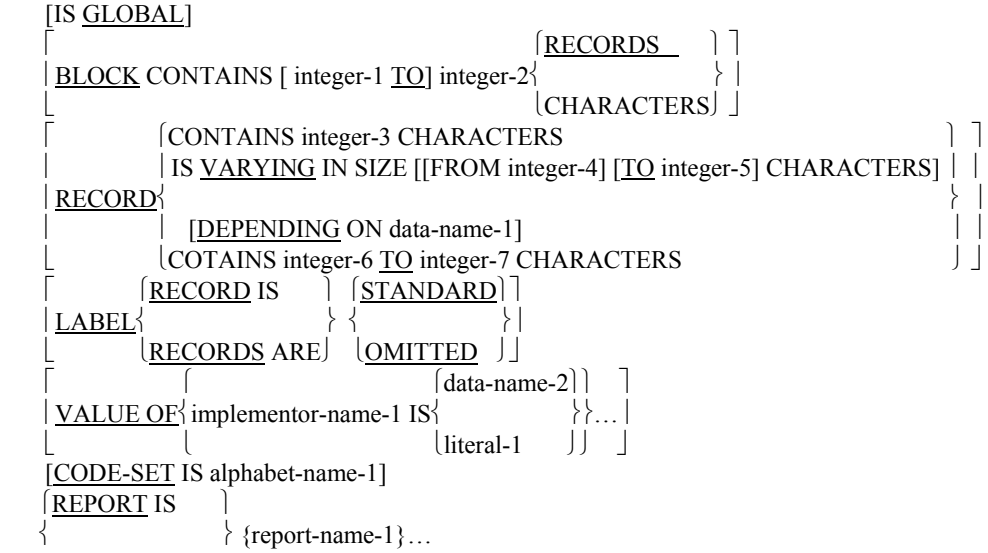
(c) SORT-MERGE FILE



(d) INDEX FILE



(e) REPORT FILE



```
[REPORTS ARE]
```

8. Data Description Entry

FORMAT 1

```
                [data-name-1]
level-number    [            ]
                [FILLER      ]
    [REDEFINES data-name-2]
    [IS EXTERNAL]
    [IS GLOBAL]
    [[PICTURE]                    ]
    [{       } IS character-string]
    [[PIC    ]                    ]
    [                [BINARY              ]]
    [                [COMPUTATIONAL       ]]
    [                [COMP                ]]
    [[USAGE IS]      {                    }]
    [                [DISPLAY             ]]
    [                [INDEX               ]]
    [                [PACKED-DECIMAL      ]]
    [           [LEADING ]                 ]
    [[SIGN IS]  {        } [SEPARATE CHARACTER]]
    [           [TRAILING]                 ]
    [OCCURS integer-2 TIMES                                    ]
    [    [ [ASCENDING ]                    ]                   ]
    [    [ {          } KEY IS {data-name-3}… [...]            ]
    [    [ [DESCENDING]                    ]                   ]
    [    [INDEXED BY {index-name-1}…]                          ]
    [OCCURS integer-1 TO integer-2 TIMES DEPENDING ON data-name-4]
    [    [ [ASCENDING ]                    ]                   ]
    [    [ {          } KEY IS {data-name-3}… [...]            ]
    [    [ [DESCENDING]                    ]                   ]
    [    [INDEXED BY {index-name-1}…]                          ]
    [[SYNCHRONIZED][LEFT ]]
    [{            }[     ]]
    [[SYNC        ] [RIGHT]]
    [[JUSTIFIED]          ]
    [{         } RIGHT    ]
    [[JUST     ]          ]
    [BLANK WHEN ZERO]
    [VALUE IS literal-1].
```

FORMAT 2

```
                                                [[THROUGH]          ]
66 data-name-1 RENAMES data-name-2 [{       } data-name-3 ].
                                                [[THRU   ]          ]
```

FORMAT 3

```
                      [VALUE IS    ] [          [[THROUGH]          ]]
88 condition-name-1 {              }{literal-1 [{       } literal-2 ]}... .
                      [VALUES ARE] [          [[THRU   ]          ]]
```

9. Communication Description Entry

FORMAT 1

```
CD cd-name-1
                          ⌈ [[SYMBOLIC QUEUE IS data-name-1]                ⌉
                          |   [SYMBOLIC SUB-QUEUE-1 IS data-name-2]        |
                          |   [SYMBOLIC SUB-QUEUE-2 IS data-name-3]        |
                          |   [SYMBOLIC SUB-QUEUE-3 IS data-name-4]        |
                          |   [MESSAGE DATE IS data-name-5]                |
                          |   [MESSAGE TIME IS data-name-6]                |
                          |   [SYMBOLIC SOURCE IS data-name-7]             |
FOR [INITIAL] INPUT       |   [TEXT LENGTH IS data-name-8]                 |
                          |   [END KEY IS data-name-9]                     |
                          |   [STATUS KEY IS data-name-10]                 |
                          |   [MESSAGE COUNT IS data-name-11]]             |
                          | [data-name-1, data-name-2, data-name-3,        |
                          |    data-name-4, data-name-5, data-name-6,      |
                          |    data-name-7, data-name-8, data-name-9,      |
                          ⌊  data-name-10, data-name-11]                   ⌋
```

FORMAT 2

```
CD cd-name-1 FOR OUTPUT
[DESTINATION COUNT IS data- name-1]
[TEXT LENGTH IS data-name-2]
[STATUS KEY IS data-name-3]
[DESTINATION TABLE OCCURS integer-1 TIMES
      [INDEXED BY {index-name-1}…]]
[ERROR KEY IS data-name-4]
[SYMBOLIC DESTINATION IS data-name-5].
```

FORMAT 3

```
CD cd-name-1
                          ⌈  [[MESSAGE DATE IS data-name-1]                ⌉
                          |       [MESSAGE TIME IS data-name-2]           |
                          |       [SYMBOLIC TERMINAL IS data-name-3]       |
FOR [INITIAL] I-O         |       [TEXT LENGTH IS data-name-4]             |
                          |       [END KEY IS data-name-5]                 |
                          |       [STATUS KEY IS data-name-6]]             |
                          | [data-name-1, data-name-2, data-name-3,        |
                          ⌊    data-name-4, data-name-5, data-name-6]      ⌋
```

10. Report Description Entry

```
RD report-name-1
    [IS GLOBAL]
    [CODE literal-1]
    ⌈⌈CONTROL IS    ⌉ ⌈{data-name-1}…        ⌉⌉
    |{               }{                      }|
    ⌊⌊CONTROLS ARE⌋ ⌊FINAL [data-name-1]…⌋⌋
          ⌈LIMIT IS   ⌉        ⌈LINE  ⌉
    [PAGE |           | integer-1|      | HEADING integer-2]
          ⌊LIMITS ARE⌋         ⌊LINES⌋
        [FIRST DETAIL integer-3] [LAST DETAIL integer-4]
        [FOOTING integer-5]].
```

(a) Report Group Description Entry

FORMAT 1

```
01  [data-name-1]
    [ [ LINE NUMBER IS { integer-1[ON NEXT PAGE] } ]
    [ [ NEXT GROUP IS { integer-3 } ]
    [                 { PLUS integer-4 }
    [                 { NEXT PAGE }
                  [ [ REPORT HEADING ]                  ]
                  [ { RH             }                  ]
                  [ [ PAGE HEADING   ]                  ]
                  [ { PH             }                  ]
                  [ [ CONTROL HEADING ] [ data-name-2 ] ]
                  [ { CH              } { FINAL       } ]
                  [ [ DETAIL ]                          ]
        TYPE IS { { DE       }                          }
                  [ [ CONTROL FOOTING ] [ data-name-3 ] ]
                  [ { CF              } { FINAL       } ]
                  [ [ PAGE FOOTING ]                    ]
                  [ { PF           }                    ]
                  [ [ REPORT FOOTING ]                  ]
                  [ { RF           }                    ]
[[USAGE IS] DISPLAY]
```

FORMAT 2

```
level-number [data-name-1]
        [ [ LINE NUMBER IS { integer-1 [ON NEXT PAGE] } ]
        [                  { PLUS integer-2            } ]
        [[USAGE IS] DISPLAY].
```

FORMAT 3

```
level-number [data-name-1]
        [ PICTURE ]
        { PIC     } IS character-string
        [[USAGE IS] DISPLAY]
        [ [ [SIGN IS] { LEADING  } SEPARATE CHARACTER ]
        [             { TRAILING }                    ]
        [ [ JUSTIFIED ]      ]
        [ { RIGHT     }      ]
        [ [ JUST      ]      ]
```

```
[BLANK WHEN ZERO]
[                    [integer-1[ON NEXT PAGE]]  ]
[ LINE NUMBER IS     {                         }  ]
[                    [PLUS integer-2            ]  ]
[COLUMN NUMBER IS integer-3]
[SOURCE IS identifier-1                           ]
[ VALUE IS literal-1                              ]
[ {SUM{identifier-2}…[UPON{data-name-2}…]}…       ]
{                                                 }
[       [             [data-name-3]  ]            ]
[       [ RESET ON    {              }  ]          ]
[       [             [FINAL         ]  ]         ]
[GROUP INDICATE].
```

11. General format of Procedure Division

FORMAT 1

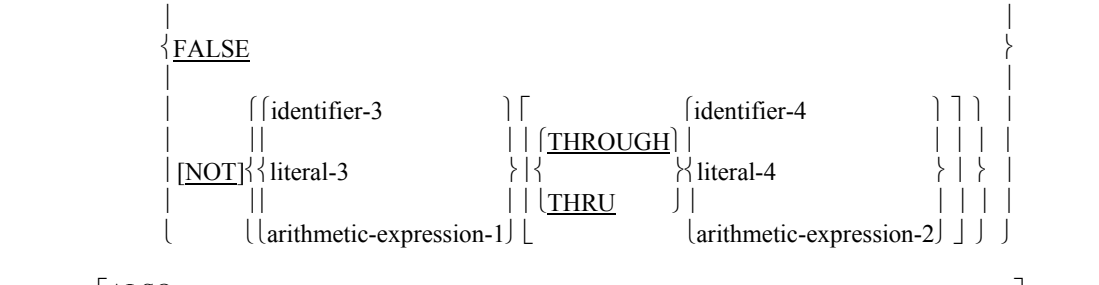
```
[PROCEDURE DIVISION [USING{data-name-1}…].
[DECLARATIVES.
{section-name SECTION [segment-number].
     USE statement.
[paragraph-name.
     [sentence]…]…}…
END DECLARATIVES.]
{section-name SECTION[segment-number].
[paragraph-name.
     [sentence]…]…}…]
```

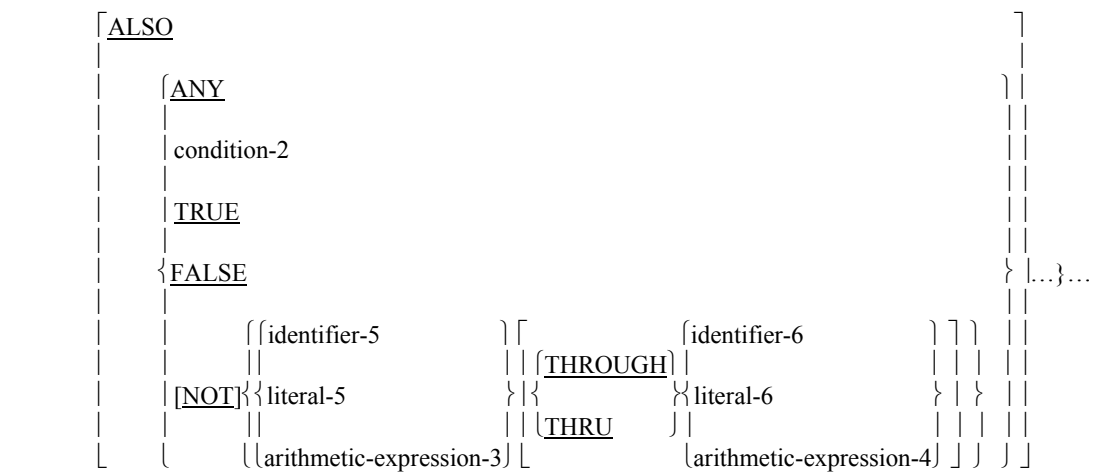
FORMAT 2

```
[PROCEDURE DIVISION [USING {data-name-1}…].
[paragraph-name.
     [sentence]…}…]

ACCEPT identifier-1 [FROM mnemonic-name-1]

                                  [DATE        ]
                                  [DAY         ]
ACCEPT identifier-2 FROM          {            }
                                  [DAY-OF-WEEK ]
                                  [TIME        ]

ACCEPT cd-name-1 MESSAGE COUNT

     [identifier-1]
ADD  {             }…TO{identifier-2 [ROUNDED]}…
     [literal-1    ]
   [ON SIZE ERROR imperative-statement-1]
   [NOT ON SIZE ERROR imperative-statement-2]
   [END ADD]

     [identifier-1]      [identifier-2]
ADD  {             }…TO  {            }
     [literal-1    ]      [literal-2   ]
   GIVING {identifier-3} [ROUNDED]}…
   [ON SIZE ERROR imperative-statement-1]
   [NOT ON SIZE ERROR imperative-statement-2]
   [END ADD]
```

```
        ⎡CORRESPONDING⎤
ADD     ⎨             ⎬identifier-1 TO identifier-2 [ROUNDED]
        ⎣CORR         ⎦
    [ON SIZE ERROR imperative-statement-1]
    [NOT ON SIZE ERROR imperative-statement-2]
    [END-ADD]

ALTER{procedure-name-1 TO [PROCEED TO] procedure-name-2}…

        ⎡identifier-1⎤ ⎡        ⎡[BY REFERENCE]{identifier-2}…⎤   ⎤
CALL    ⎨            ⎬ ⎢USING   ⎨                            ⎬…  ⎢
        ⎣literal-1   ⎦ ⎣        ⎣BY CONTENT{identifier-2}…   ⎦   ⎦

    [ON OVERFLOW imperative-statement-1] [END-CALL]

        ⎡identifier-1⎤ ⎡        ⎡[BY REFERENCE]{identifier-2}…⎤   ⎤
CALL    ⎨            ⎬ ⎢USING   ⎨                            ⎬…  ⎢
        ⎣literal-1   ⎦ ⎣        ⎣BY CONTENT{identifier-2}…   ⎦   ⎦

[ON EXCEPTION imperative-statement-1]
[NOT ON EXCEPTION imperative-statement-2]
[END-CALL]

          ⎡identifier-1⎤
CANCEL    ⎨            ⎬…
          ⎣literal-1   ⎦

          ⎛             ⎡⎡REEL  ⎤                    ⎤⎞
          ⎜             ⎢⎨      ⎬ [FOR REMOVAL]      ⎥⎥
          ⎜             ⎢⎣UNIT  ⎦                    ⎥⎥
S W       CLOSE⎨file-name-1⎥                          ⎥⎬…
          ⎜             ⎥       ⎡NO REWIND⎤          ⎥⎥
          ⎜             ⎢ WITH  ⎨         ⎬          ⎥⎥
          ⎝             ⎣       ⎣LOCK     ⎦          ⎦⎠

R I       CLOSE{file-name-1} [WITH LOCK] }…

CONTINUE

DELETE file-name-1 RECORD
    [INVALID KEY imperative-statement-1]
    [NOT INVALID KEY imperative-statement-2]
    [END-DELETE]

          ⎡INPUT [TERMINAL]⎤
DISABLE   ⎨I-O TERMINAL    ⎬ cd-name-1
          ⎣OUTPUT          ⎦

          ⎡identifier-1⎤
DISPLAY   ⎨            ⎬…[UPON mnemonic-name-1] [WITH NO ADVANCING]
          ⎣literal-1   ⎦
```

```
          ⎡identifier-1⎤
DIVIDE    ⎢            ⎥INTO {identifier-2 [ROUNDED]}…
          ⎣literal-1   ⎦
      [ON SIZE ERROR imperative-statement-1]
      [NOT ON SIZE ERROR imperative-statement-2]
      [END-DIVIDE]

          ⎡identifier-1⎤      ⎡identifier-2⎤
DIVIDE    ⎢            ⎥INTO ⎢            ⎥
          ⎣literal-1   ⎦      ⎣literal-2   ⎦
    GIVING {identifier-3 [ROUNDED]}…
    [ON SIZE ERROR imperative-statement-1]
    [NOT ON SIZE ERROR imperative-statement-2]
    [END-DIVIDE]

          ⎡identifier-1⎤    ⎡identifier-2⎤
DIVIDE    ⎢            ⎥BY  ⎢            ⎥
          ⎣literal-1   ⎦    ⎣literal-2   ⎦
    GIVING {identifier-3 [ROUNDED]}…
    [ON SIZE ERROR imperative-statement-1]
    [NOT ON SIZE ERROR imperative-statement-2]
    [END-DIVIDE]

          ⎡identifier-1⎤      ⎡identifier-2⎤
DIVIDE    ⎢            ⎥INTO ⎢            ⎥ GIVING identifier-3 [ROUNDED]
          ⎣literal-1   ⎦      ⎣literal-2   ⎦
    REMAINDER identifier-4
    [ON SIZE ERROR imperative-statement-1]
    [NOT ON SIZE ERROR imperative-statement-2]
    [END-DIVIDE]

          ⎡identifier-1⎤    ⎡identifier-2⎤
DIVIDE    ⎢            ⎥BY  ⎢            ⎥ GIVING identifier-3 [ROUNDED]
          ⎣literal-1   ⎦    ⎣literal-2   ⎦
     REMAINDER identifier-4
     [ON SIZE ERROR imperative-statement-1]
     [NOT ON SIZE ERROR imperative-statement-2]
     [END-DIVIDE]

          ⎡INPUT [TERMINAL]⎤
ENABLE    ⎢I-O TERMINAL    ⎥ cd-name-1
          ⎣OUTPUT          ⎦

              ⎡identifier-1⎤ ⎡      ⎡identifier-2⎤⎤
              ⎢literal-1   ⎥ ⎢      ⎢literal-2   ⎥⎥
EVALUATE      ⎢expression-1⎥ ⎢ALSO  ⎢expression-2⎥⎥…
              ⎢TRUE        ⎥ ⎢      ⎢TRUE        ⎥⎥
              ⎣FALSE       ⎦ ⎣      ⎣FALSE       ⎦⎦


{{WHEN
      ⎡ANY                                                  ⎤
      ⎢                                                     ⎥
      ⎢condition-1                                          ⎥
      ⎢                                                     ⎥
      ⎣TRUE                                                 ⎦
```





imperative-statement-1}…

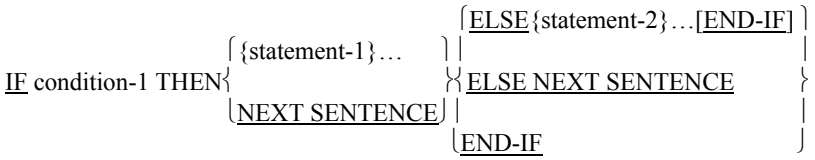
[WHEN OTHER imperative-statement-2]
[END-EVALUATE]

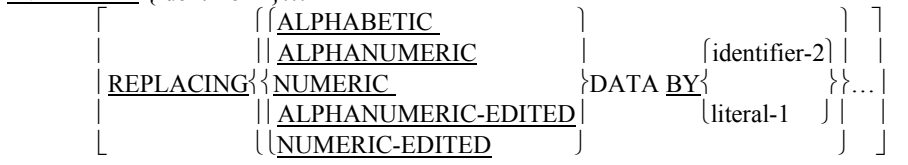
EXIT

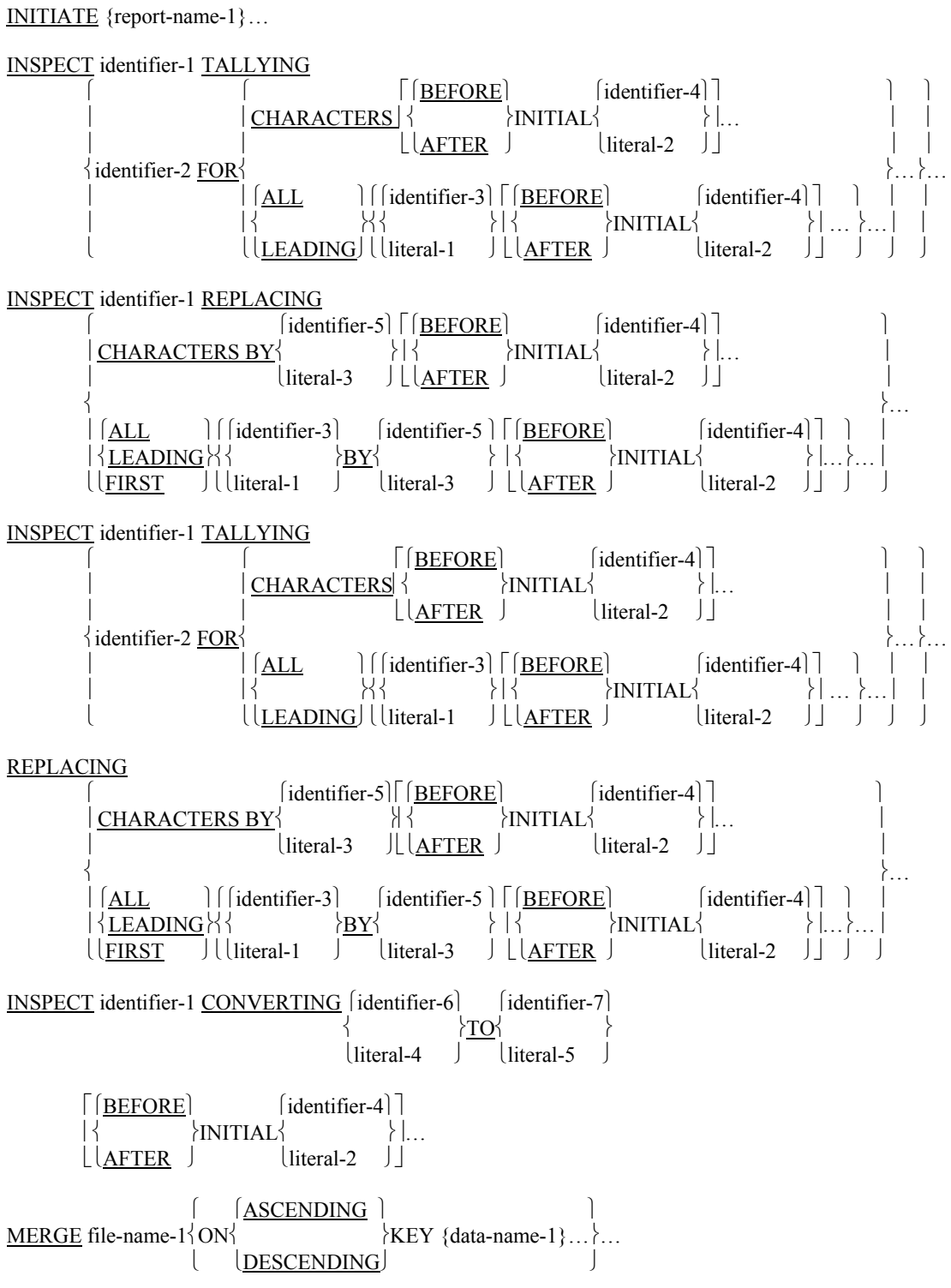
EXIT PROGRAM

GO TO [procedure-name-1]

GO TO {procedure-name-1}…DEPENDING ON identifier-1






INITIATE {report-name-1}…
INSPECT identifier-1 TALLYING
CHARACTERS
BEFORE
INITIAL
AFTER
literal-2
identifier-2 FOR
ALL
LEADING
identifier-3
literal-1
INSPECT identifier-1 REPLACING
CHARACTERS BY
identifier-5
literal-3
INSPECT identifier-1 TALLYING
REPLACING
INSPECT identifier-1 CONVERTING identifier-6 identifier-7
TO
literal-4
literal-5
MERGE file-name-1 ON ASCENDING DESCENDING KEY {data-name-1}…
[COLLATING SEQUENCE IS alphabet-name-1]
USING file-name-2 {file-name-3}…
OUTPUT PROCEDURE IS procedure-name-1
THROUGH
THRU
procedure-name-2

```
          |                                                                  |
          |GIVING {file-name-4}                                              |

         |identifier-1|
MOVE     |            |TO {identifier-2}…
         |literal-1   |


         |CORRESPONDING|
MOVE     |             |identifier-1 TO identifier-2
         |CORR         |

              |identifier-1|
MULTIPLY      |            |BY {identifier-2 [ROUNDED]}…
              |literal-1   |
           [ON SIZE ERROR imperative-statement-1]
           [NOT ON SIZE ERROR imperative-statement-2]
           [END-MULTIPLY]

              |identifier-1 |   |identifier-2|
MULTIPLY      |             |BY |            |
              |literal-1    |   |literal-2   |
           GIVING {identifier-3 [ROUNDED]}…
           [ON SIZE ERROR imperative-statement-1]
           [NOT ON SIZE ERROR imperative-statement-2]
           [END-MULTIPLY]

                 |INPUT {file-name-1 [WITH NO REWIND]}…      |
                 |OUTPUT {file-name-2 [WITH NO REWIND]}…     |
S        OPEN    |                                           |…
                 |I-O {file-name-3}…                         |
                 |EXTEND {file-name-4}…                      |

                 |INPUT {file-name-1}…     |
                 |OUTPUT {file-name-2}…    |
R I      OPEN    |                         |…
                 |I-O {file-name-3}…       |
                 |EXTEND {file-name-4}…    |

                 |OUTPUT {file-name-1 [WITH NO REWIND]}…|
W        OPEN    |                                      |…
                 |EXTEND {file-name-2}…                 |

              |                  |  |          |                    |  |
PERFORM       | procedure-name-1 |  |{         |} procedure-name-2  |  |
              |                  |  |  |THRU   |                    |  |
           [imperative-statement-1 END-PERFORM]


              |                  |  |          |                    |  |
PERFORM       | procedure-name-1 |  |{         |} procedure-name-2  |  |
              |                  |  |  |THRU   |                    |  |
           |identifier-1|
           |            |TIMES [imperative-statement-1 END-PERFORM]
           |integer-1   |
```

```
          ⌈                  ⌈⌈THROUGH⌉                    ⌉⌉
PERFORM⌈ procedure-name-1⌈⌈    ⌋ procedure-name-2⌉⌉
       ⌊                  ⌊⌊THRU   ⌋                    ⌋⌋
      ⌈                ⌈BEFORE⌉                  ⌉
      ⌈ WITH TEST⌈             ⌋UNTIL condition-1⌉
      ⌊          ⌊AFTER ⌋                        ⌋
      [imperative-statement-1 END-PERFORM]
```

```
          ⌈                  ⌈⌈THROUGH⌉                    ⌉⌉
PERFORM⌈ procedure-name-1⌈⌈    ⌋ procedure-name-2⌉⌉
       ⌊                  ⌊⌊THRU   ⌋                    ⌋⌋
      ⌈                ⌈BEFORE⌉ ⌉
      ⌈ WITH TEST⌈             ⌋ ⌉
      ⌊          ⌊AFTER ⌋      ⌋

                ⌈identifier-2    ⌉          ⌈identifier-3    ⌉
      VARYING⌈                   ⌋ FROM ⌈index-name-2⌉
             ⌊index-name-1 ⌋          ⌊literal-1       ⌋

                  ⌈identifier-4⌉
            BY⌈                ⌋ UNTIL condition-1
              ⌊literal-2     ⌋

      ⌈       ⌈identifier-5⌉       ⌈identifier-6    ⌉⌉
      ⌈ AFTER⌈             ⌋FROM⌈ index-name-4⌉⌉
      ⌈       ⌊literal-3   ⌋       ⌊literal-3       ⌋⌉
      ⌈                                             ⌋…
      ⌈       ⌈identifier-7⌉                        ⌉
      ⌈    BY⌈             ⌋ UNTIL condition-2      ⌉
      ⌊       ⌊literal-4   ⌋                        ⌋
      [imperative-statement-1 END-PERFORM]
```

```
PURGE cd-name-1
```

```
S R I   READ file-name-1 [NEXT] RECORD [INTO identifier-1]
            [AT END imperative-statement-1]
            [NOT AT END imperative-statement-2]
            [END-READ]

  R     READ file-name-1 RECORD [INTO identifier-1]
            [INVALID KEY imperative-statement-3]
            [NOT INVALID KEY imperative-statement-4]
            [END-READ]

  I     READ file-name-1 RECORD [INTO identifier-1]
            [KEY IS data-name-1]
            [INVALID KEY imperative-statement-3]
            [NOT INVALID KEY imperative-statement-4]
            [END-READ]

                        ⌈MESSAGE⌉
        RECEIVE cd-name-1⌈       ⌋ INTO identifier-1
                        ⌊SEGMENT⌋
            [NO DATA imperative-statement-1]
            [WITH DATA imperative-statement-2]
            [END-RECEIVE]
```

RELEASE record-name-1[FROM identifier-1]

RETURN file-name-1 RECORD [INTO identifier-1]
AT END imperative-statement-1
[NOT AT END imperative-statement-2]
[END-RETURN]

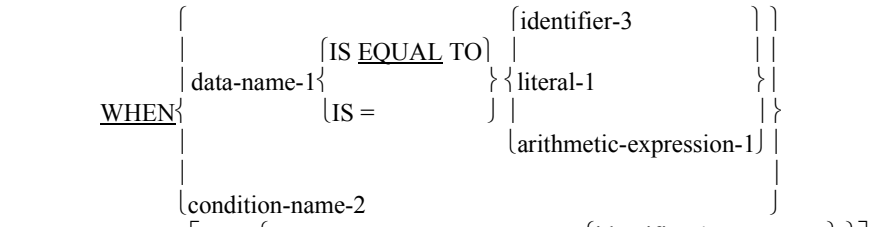
S REWRITE record-name-1 [FROM identifier-1]

R I REWRITE record-name-1 [FROM identifier-1]
[INVALID KEY imperative-statement-1]
[NOT INVALID KEY imperative-statement-2]
[END-REWRITE]

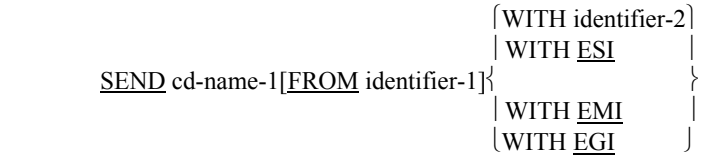
[AT END imperative-statement-1]

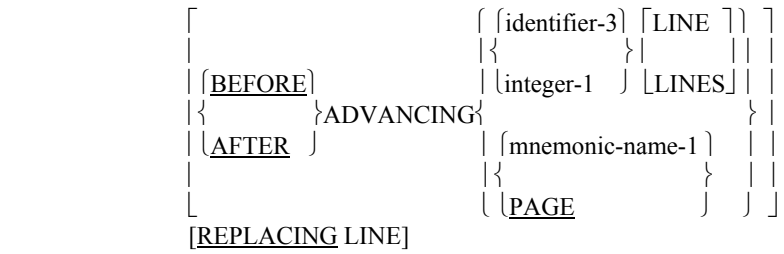
[END-SEARCH]

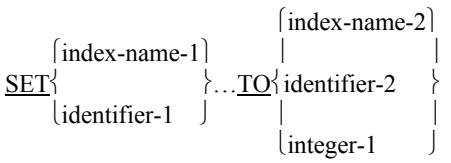
SEARCH ALL identifier-1 [AT END imperative-statement-1]

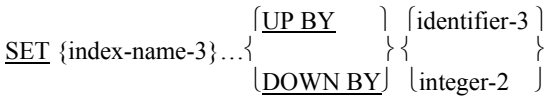


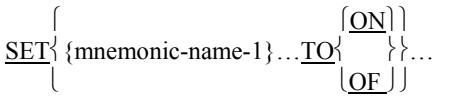
[END-SEARCH]



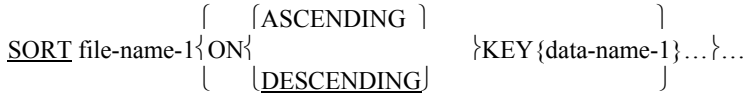




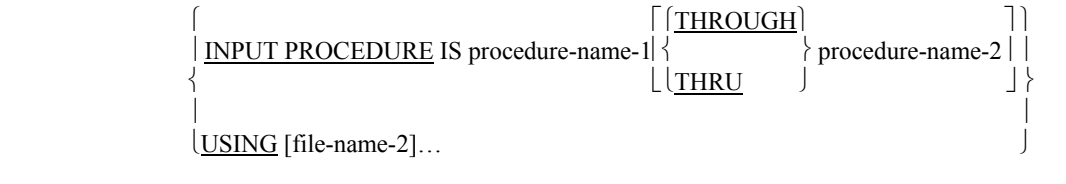


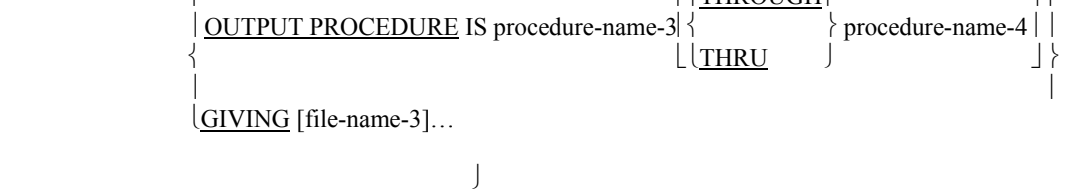


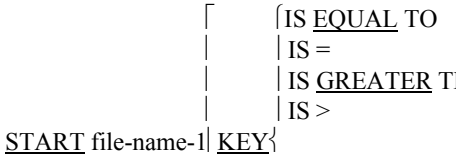
SET {condition-name-1}… TO TRUE



[WITH DUPLICATES IN ORDER]
[COLLATING SEQUENCE IS alphabet-name-1]







```
                    |      | IS NOT LESS THAN                  |           |
                    |      | IS NOT <                          |           |
                    |      | IS GREATER THAN OR EQUAL TO       |           |
                    |      | IS >=                             |           |

        [INVALID KEY imperative-statement-1]
        [NOT INVALID KEY imperative-statement-2]
        [END-START]

      [RUN    ]
STOP{         }
      [literal-1]

          [                                  [identifier-2]  ]
          | [identifier-1 ]                  |               ||
STRING{{  |             }...DELIMITED BY{literal-2     }}...
          | [literal-1    ]                  |               ||
          [                                  [SIZE         ]]
        INTO identifier-3
        [WITH POINTER identifier-4]
        [ON OVERFLOW imperative-statement-1]
        [NOT ON OVERFLOW imperative-statement-2]
        [END-STRING]

            [identifier-1]
SUBTRACT{                 }...FROM {identifier-3[ROUNDED]}...
            [literal-1    ]
        [ON SIZE ERROR imperative-statement-1]
        [NOT ON SIZE ERROR imperative-statement-2]
        [END-SUBTRACT]

            [identifier-1]              [identifier-2]
SUBTRACT{                 }... FROM{                }
            [literal-1    ]             [literal-1    ]
        GIVING  {identifier-3} [ROUNDED]}...
        [ON SIZE ERROR imperative-statement-1]
        [NOT ON SIZE ERROR imperative-statement-2]
        [END-SUBTRACT]

            [CORRESPONDING ]
SUBTRACT{                  } identifier-1 FROM identifier-2 [ROUNDED]
            [CORR          ]
        [ON SIZE ERROR imperative-statement-1]
        [NOT ON SIZE ERROR imperative-statement-2]
        [END-SUBTRACT]

SUPPRESS PRINTING

TERMINATE {report-name-1}...

UNSTRING identifier-1
        [                          [identifier-2] [            [identifier-3] ]   ]
        | DELIMITED BY [ALL]{                   }| OR [ALL]{                   } |...|
        [                          [literal-1   ] [            [literal-2     ] ]   ]
        INTO {identifier-4 [DELIMITER IN identifier-5] [COUNT IN identifier-6]}...
        [WITH POINTER identifier-7]
        [TALLYING IN identifier-8]
        [ON OVERFLOW imperative-statement-1]
```

```
        [NOT ON OVERFLOW imperative-statement-2]
        [END-UNSTRING]

                                                                  ⌈{file-name-1}…⌉
                                                  ⌈EXCEPTION⌉     | INPUT       |
USE [GLOBAL] AFTER STANDARD{                      |           } PROCEDURE ON {OUTPUT      }
                                                  ⌊ERROR    ⌋     | I-O         |
                                                                  ⌊EXTEND      ⌋

        USE [GLOBAL] BEFORE REPORTING identifier-1

                                    ⌈cd-name-1                          ⌉
                                    | [ALL REFERENCES OF] identifier-1  |
USE FOR DEBUGGING ON                {file-name-1                        }…
                                    | procedure-name-1                  |
                                    ⌊ALL PROCEDURES                     ⌋

S       WRITE record-name-1 [FROM identifier-1]

                ⌈                       ⌈ ⌈identifier-2⌉ ⌈LINE ⌉⌉ ⌉
                |                       |{            }|      ||  |
                | ⌈BEFORE⌉              | ⌊integer-1   ⌋ ⌊LINES⌋|  |
                |{       }ADVANCING     {                       } |
                | ⌊AFTER ⌋              | ⌈mnemonic-name-1⌉     |  |
                |                       |{               }      |  |
                ⌊                       ⌊ ⌊PAGE          ⌋      ⌋  ⌋
                ⌈    ⌈END-OF-PAGE⌉                  ⌉
                | AT{               }imperative-statement-1|
                ⌊    ⌊EOP         ⌋                  ⌋
        [END-WRITE]

R I     WRITE record-name-1 [FROM identifier-1]
                [INVALID KEY imperative-statement-1]
                [NOT INVALID KEY imperative-statement-2]
                [END-WRITE]
```

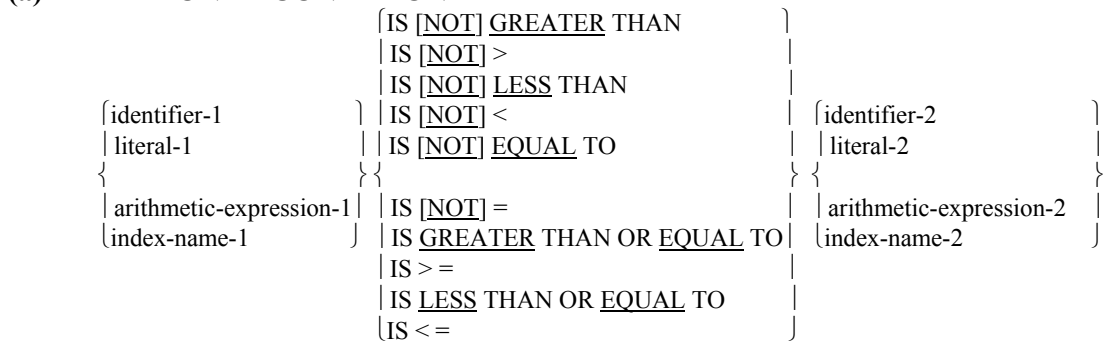
12. COPY and REPLACE Statements

```
                    ⌈⌈OF⌉                 ⌉
COPY text-name-1    |{  }library-name-1    |
                    ⌊⌊IN⌋                 ⌋

⌈                ⌈ ⌈= = pseudo-text-1 = =⌉    ⌈= = pseudo-text-2 = =⌉ ⌉    ⌉
|                || identifier-1           |    | identifier-2           ||    |
| REPLACING{     |{                       }BY{                       }}…  |
|                || literal-1              |    | literal-2              ||    |
⌊                ⌊ ⌊word-1                ⌋    ⌊word-2                ⌋⌋    ⌋
```

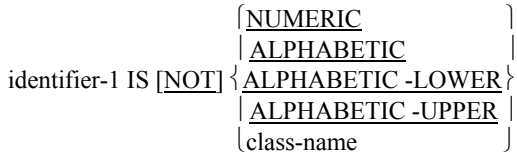
```
REPLACE {= = pseudo-text-1 = = BY pseudo-text-2 = =}…

REPLACE OFF
```

13. Conditions

(a) RELATIONAL CONDITION



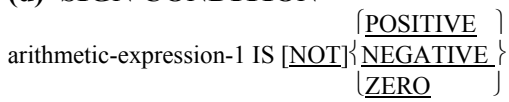
(b) CLASS CONDITION



(c) CONDITION-NAME CONDITION

condition-name-1

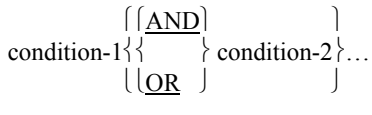
(d) SIGN CONDITION



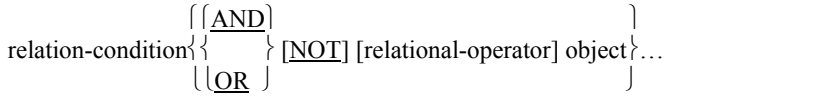
(e) NEGATED CONDITION

NOT condition-1

(f) COMPOUND CONDITION

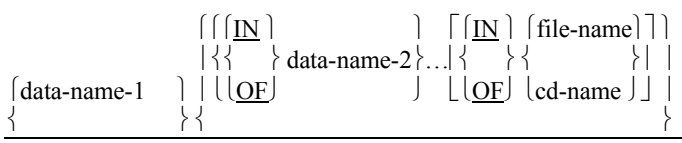


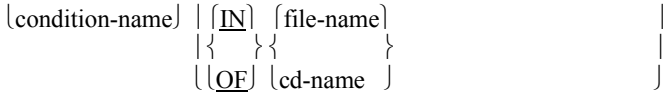
(g) ABBREVIATED COMPOUND RELATIONAL CONDITION



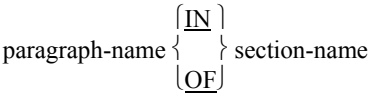
14. Qualification

FORMAT 1

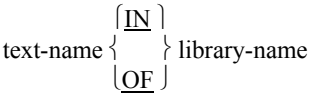




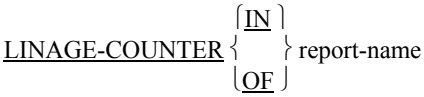
FORMAT 2



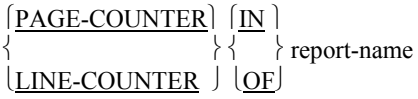
FORMAT 3



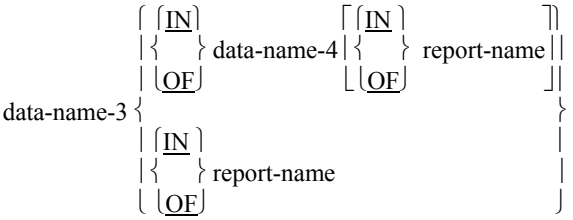
FORMAT 4



FORMAT 5

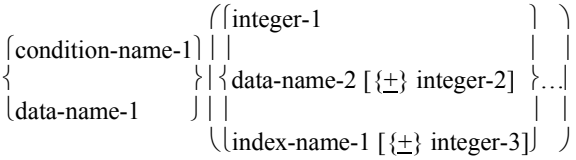


FORMAT 6



15. Miscellaneous Formats

(a) SUBSCRIPTING



(b) REFERENCE MODIFICATION

data-name-1 (leftmost-character-position: [length])

16. Nested Source Programs

IDENTIFICATION DIVISION.
PROGRAM-ID. Program-name-1 [IS INITIAL PROGRAM].

[ENVIRONMENT DIVISION. environment-division-content]
[DATA DIVISION. data-division-content]
[PROCEDURE DIVISION. procedure-division-content]
[[nested-source-program]…
END PROGRAM program-name-1.]

NESTED-SOURCE-PROGRAM

IDENTIFICATION DIVISION.
PROGRAM-ID. program-name-2 [IS {COMMON / INITIAL} PROGRAM].
[ENVIRONMENT DIVISION. environment-division-content]
[DATA DIVISION. data-division-content]
[PROCEDURE DIVISION. procedure-division-content]
[nested-source-program]…
END PROGRAM program-name-2.

Appendix – B

COBOL Reserved Words			
ACCEPT	COLUMN	DETAIL	EXTERNAL
ACCESS	COMMA	DISABLE	FALSE
ADD	COMMON	DISPLAY	FD
ADVANCING	COMMUNICATION	DIVIDE	FILE
AFTER	COMP	DIVISION	FILE-CONTROL
ALL	COMPUTATIONAL	DOWN	FILLER
ALPHABET	COMPUTE	DUPLICATES	FINAL
ALPHABETIC	CONFIGURATION	DYNAMIC	FIRST
ALPHABETIC-LOWER	CONTAINS	EGI	FOOTING
ALPHABETIC-UPPER	CONTENT	ELSE	FOR
ALPHANUMERIC	CONTINUE	EMI	FROM
ALPHANUMERIC-EDITED	CONTROL	ENABLE	GENERATE
ALSO	CONTROLS	END	GIVING
ALTER	CONVERTING	END-ADD	GLOBAL
ALTERNATE	COPY	END-CALL	GO
AND	CORR	END-COMPUTE	GREATER
ANY	CORRESPONDING	END-DELETE	GROUP
ARE	COUNT	END-DIVIDE	HEADING
AREA	CURRENCY	END-EVALUATE	HIGH-VALUE
AREAS	DATA	END-IF	HIGH-VALUES
ASCENDING	DATE	END-MULTIPLY	I-O
ASSIGN	DATE-COMPILED	END-OF-PAGE	I-O-CONTROL
AT	DATE-WRITTEN	END-PERFORM	IDENTIFICATION
AUTHOR	DAY	END-READ	IF
BEFORE	DAY-OF-WEEK	END-RECEIVE	IN
BINARY	DE	END-RETURN	INDEX
BLANK	DEBUG-CONTENTS	END-REWRITE	INDEXED
BLOCK	DEBUG-ITEM	END-SEARCH	INDICATE
BOTTOM	DEBUG-LINE	END-START	INITIAL
BY	DEBUG-NAME	END-STRING	INITIALIZE
CALL	DEBUG-SUB-1	END-SUBSTRACT	INITIATE
CANCEL	DEBUG-SUB-2	END-UNSTRING	INPUT
CD	DEBUG-SUB-3	END-WRITE	INPUT-OUTPUT
CF	DEBUGGING	ENVIRONMENT	INSPECT
CH	DECIMAL-POINT	EOP	INSTALLATION
CHARACTER	DECLARATIVES	EQUAL	INTO
CHARACTERS	DELETE	ERROR	INVALID
CLASS	DELIMITED	ESI	IS
CLOSE	DELIMITER	EVALUATE	JUST
CODE	DEPENDING	EXCEPTION	JUSTIFIED
CODE-SET	DESCENDING	EXIT	KEY
COLLATING	DESTINATION	EXTEND	LABEL

COBOL Reserved Words			
LAST	PADDING	RESET	SUM
LEADING	PAGE	RETURN	SUPPRESS
LEFT	PAGE-COUNTER	REWIND	SYMBOLIC
LENGTH	PERFORM	REWRITE	SYNC
LESS	PF	RF	SYNCHRONIZED
LIMIT	PH	RH	TABLE
LIMITS	PIC	RIGHT	TALLYING
LINAGE	PICTURE	ROUNDED	TAPE
LINAGE-COUNTER	PLUS	RUN	TERMINAL
LINE	POINTER	SAME	TERMINATE
LINE-COUNTER	POSITION	SD	TEST
LINES	POSITIVE	SEARCH	TEXT
LINKAGE	PRINTING	SECTION	THAN
LOCK	PROCEDURE	SECURITY	THEN
LOW-VALUE	PROCEDURES	SEGMENT	THROUGH
LOW-VALUES	PROCEED	SEGMENT-LIMIT	THRU
MERGE	PROGRAM	SELECT	TIME
MESSAGE	PROGRAM-ID	SEND	TIMES
MODE	PURGE	SENTENCE	TO
MOVE	QUEUE	SEPARATE	TOP
MULTIPLE	QUOTE	SEQUENCE	TRAILING
MULTIPLY	QUOTES	SEQUENTIAL	TRUE
NATIVE	RANDOM	SET	TYPE
NEGATIVE	RD	SIGN	UNIT
NEXT	READ	SIZE	UNSTRING
NO	RECEIVE	SORT	UNTIL
NUMBER	RECORD	SORT-MERGE	UP
NUMERIC	RECORDS	SOURCE	UPON
NUMERIC-EDITED	REDEFINES	SOURCE-COMPUTER	USAGE
OBJECT-COMPUTER	REEL	SPACE	USE
OCCURS	REFERENCE	SPACES	USING
OFF	REFERENCES	SPECIAL-NAMES	VALUE
OMITTED	RELATIVE	STANDARD	VALUES
ON	RELEASE	STANDARD-1	VARYING
OPEN	REMAINDER	STANDARD-2	WHEN
OPTIONAL	REMOVAL	START	WITH
OR	RENAMES	STATUS	WORKING-STORAGE
ORDER	REPLACE	STOP	ZERO
ORGANIZATION	REPLACING	STRING	ZEROES
OTHER	REPORT	SUB-QUEUE-1	ZEROS
OUTPUT	REPORTING	SUB-QUEUE-2	
OVERFLOW	REPORTS	SUB-QUEUE-3	
PACKED-DECIMAL	RESERVE	SUBTRACT	

Appendix - C

1. File Processing in COBOL

			⇐ OPEN MODE ⇒			
File ORGANIZATION	**ACCESS MODE**	**Statement**	**INPUT**	**OUTPUT**	**I-O**	**EXTEND**
SEQUENTIAL	SEQUENTIAL	READ WRITE REWRITE	X	 X 	X X	 X
LINE SEQUENTIAL	SEQUENTIAL	READ WRITE	X	 X		 X
RELATIVE and INDEXED	SEQUENTIAL	READ WRITE REWRITE DELETE START	X X	 X 	X X X X	
	RANDOM	READ WRITE REWRITE DELETE START	X	 X	X X X X	
	DYNAMIC	READ WRITE REWRITE DELETE START	X X	 X 	X X X X X	

2. File Status Codes

Contents of the **FILE STATUS** field after an Input or Output operation	**Meaning**
Successful Completion *00 *02 04	➢ Successful completion - no error occurred. ➢ The record being processed has a duplicate alternate record key.(Note: This is not an error when your program includes WITH DUPLICATES for the ALTERNATE RECORD KEY) ➢ A READ statement has been successfully completed, but the length of the record does not conform to the File Description specifications.
Unsuccessful Completion *10 *21 *22 *23 *24 *30 34 37 41 42 *43 9x	➢ A sequential READ statement (READ…AT END) has been attempted, but there are no more input records. ➢ A sequence error has occurred - keys are not in the correct order. ➢ An attempt was made to write a record that would create a duplicate primary record key. ➢ The required record was not found during a READ. ➢ A boundary error has occurred - an attempt has been made to write beyond the pre-established boundaries of an indexed file as established by the operating system. ➢ A permanent date error has occurred (this is a hardware problem.) ➢ A boundary error for a sequential file has occurred. ➢ A permanent error has occurred because an OPEN statement has been attempted on a file that will not support the mode specified in the OPEN statement (e.g., an indexed file is opened as OUTPUT when ACCESS IS RANDOM has been specified, or a print file is opened as I-O). ➢ An OPEN statement has been attempted on a file that is already open. ➢ A CLOSE statement has been attempted on a file that has not been opened. ➢ An attempt has been made to DELETE or REWRITE a record after an unsuccessful READ (e.g., there is no record in storage to delete or rewrite). ➢ Codes of 91-99 are implementation specific - consult your user's manual.

Note: * Indicates that codes applies to index/relatives files.

FILE STATUS field must be a working storage variable with PIC X(002) specification.

Appendix – D

PIC Clauses in detail:

This Appendix describes different numeric PICTURE clauses in detail.

The numeric data you use in your program will be one of the five types available with VS COBOL II:

External decimal (USAGE DISPLAY)

Internal decimal (USAGE PACKED-DECIMAL) (Same as USAGE COMP-3)

Binary (USAGE BINARY) (Same as USAGE COMP)

External floating point (USAGE DISPLAY)

Internal floating point (USAGE COMPUTATIONAL-1, USAGE COMPUTATIONAL-2).

Note: The types COMPUTATIONAL and COMPUTATIONAL-4 are synonymous with BINARY. The type COMPUTATIONAL-3 is synonymous with PACKED-DECIMAL.

Below is the explanation of different types of USAGE clauses:

USAGE IS	Byte Required	Description
External Decimal (DISPLAY) DISPLAY is the default when no USAGE clause is specified.	1 byte for each digit	- Used for receiving and sending numbers between programs and files, terminals, and printers. - The data item is stored in character form, but is converted when used in arithmetic operation.
Internal Decimal (PACKED-DECIMAL) Same as USAGE COMP-3	1 byte for every 2 digits, except for right-most byte, which stores 1 digit plus the sign.	- Used for arithmetic operands and results. - The PICTURE clause should contain odd number of digits for most efficient use. - Better suited for decimal alignment than BINARY. - Converted to and from DISPLAY more efficiently than BINARY is. - Requires less storage than DISPLAY.
Binary (BINARY) Same as USAGE COMP	- 2 bytes if n >=1 and <=4. - 4 bytes if n >= 5 and n <=9 - 8 bytes if n >= 10 and <= 18 Where n is the number of 9's in the	- For subscripting, switches, and arithmetic operands and results. - Binary data items with 9 or more digits require more handling by the compiler. Testing them for the ON SIZE ERROR condition and rounding is more cumbersome than with other types. - Not converted to and from DISPLAY as efficiently as PACKED-DECIMAL.

USAGE IS	Byte Required	Description
	PICTURE clause.	- Requires less storage than DISPLAY.
External Floating Point (DISPLAY) PIC 9(n)V9(m) Where n and m are precision before decimal and after decimal.	1 byte for each character (except for V) 'V' is used for storing decimal point (Not for displaying)	Converted to COMP-2 format when used in arithmetic operation. External floating point is always converted before any arithmetic operation.
Internal Floating Point (COMP-1)	4 bytes	- For arithmetic operands and results, especially when maintaining highest level of accuracy is important. - The PICTURE Clause cannot be specified for COMP-1 items.
Internal Floating Point (COMP-2)	8 bytes	- For arithmetic operands and results, especially when maintaining highest level of accuracy is important. - The PICTURE Clause cannot be specified for COMP-2 items.

The following example shows the use of different PICTURE clause.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PICTURE1.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-VAR1-COM-1            COMP-1.
01 WS-VAR1-DEC              PIC S9(7)V9(2) COMP-3.
01 WS-VAR1-DISP1            PIC -(6)9(2).9(2).
01 WS-VAR1-DISP2            PIC -Z(5)99.99.
01 WS-VAR1-CHAR             PIC X(11).

PROCEDURE DIVISION.
    MOVE -1234.59      TO WS-VAR1-COM-1
    MOVE WS-VAR1-COM-1 TO WS-VAR1-DEC
    MOVE WS-VAR1-DEC   TO WS-VAR1-DISP1
    MOVE WS-VAR1-DEC   TO WS-VAR1-DISP2
    MOVE WS-VAR1-DISP2 TO WS-VAR1-CHAR
    DISPLAY 'WS-VAR1-DEC = ', WS-VAR1-DEC
    DISPLAY 'WS-VAR1-DISP1 = ', WS-VAR1-DISP1
    DISPLAY 'WS-VAR1-DISP2 = ', WS-VAR1-CHAR
    GOBACK.
*********************************
Output of the program:
WS-VAR1-DEC = 00012345R
```

```
WS-VAR1-DISP1 =    -1234.59
WS-VAR1-DISP2 = -   1234.59
```

In the above example,

- WS-VAR1-COM-1 takes 4 bytes.
- WS-VAR1-DEC takes 5 bytes (Stores data in (9+2)/2 = 4.5 bytes, hence required 5 bytes). Each byte stores 2 digits number in Packed Decimal format.
- To display the decimal point and sign character (-/+) the WS-VAR1-DISP1 can declare as -9(7).9(2). Here -(6)9(2).9(2) is used to right shift the "-" sign suppressing up to 5 leading zeros. If we declare WS-VAR1-DISP1 as -Z(5)99.99, it will suppress 5 leading zeros but will the right shift the Sign character. If we want to store WS-VAR1-DISP1 to a string, we need to declare it as PIC X(11) as the Sing bit and decimal point take one byte each.

Suggestive Example: Write a COBOL program to read some positive and negative dollar amount (Prefixed with $) from a character string (stored in a file), do some calculation (add/subtract/multiply, division) and write the result in a character string and direct it to an output file having only character filed. Also display the derived dollar amount on the consol of onto the SYSOUT.

- Some conversion scenarios for Fixed point (external decimal, packed decimal, and binary), and floating-point (External floating point, COMP-1 and COMP-2) data items.

  - If a fixed-point data item is moved to an external floating-point data item where the PICTURE of the fixed-point data item specifies more digit positions than the PICTURE of the external floating-point data item, the extra low-order digits are rounded.
  - If a fixed-point data item with 6 or fewer digits is moved to a USAGE COMP-1 data item and then returned to the fixed-point data item, the original value is recovered.
  - If a USAGE COMP-1 data item is moved to a fixed-point data item of 9 or more digits and then returned to the USAGE COMP-1 data item, the original value is recovered. When a USAGE COMP-1 data item is moved to a fixed-point data item with more than 9 digits, the fixed-point data item will receive only 9 significant digits, and all the remaining digits will be zero.
  - If a fixed-point data item with 15 or fewer digits is moved to a USAGE COMP-2 data item and then returned to the fixed-point data item, the original value is recovered.
  - If a USAGE COMP-2 data item is moved to a fixed-point (not external floating-point) data item of 18 digits and then returned to the USAGE COMP-2 data item, the original value is recovered.
  - If a USAGE COMP-1 data item, a USAGE COMP-2 data item, an external floating-point data item, or a floating-point literal is moved to a fixed-point data item, the extra low-order digits are rounded.
  - If a USAGE COMP-2 data item is moved to a USAGE COMP-1 data item, the extra low-order digits are rounded.

Appendix – E

This Appendix describes the important features of different versions of COBOL.

OS/VS COBOL:

- Adheres to ANSI 1968 and ANSI 1974 and doesn't support structured programming features.
- The period (**.**) is the scope-terminator for control statements like IF-ELSE, PERFORM etc.
- Supports alternate indexes on an ESDS files.
- Generate object code with 24-bit address. Your program and its data are constrained to fit in a 16-megabyte address space. Does not support POINTER, EXTERNAL and GLOBAL data items. Doesn't support Nested Programming (One program inside another program). Does not support Dynamic CALL.

VS COBOL II:

The following points describe how COBOL has been dramatically improved in its VS COBOL II version.

- It adheres to ANSI 1985. Structure programming concept has been introduced in VS COBOL II. The clauses like EVALUATE, CONTINUE, in line PERFORM have been introduced to make the program structured. The scope-terminator for control statements link IF-ELSE, PERFORM have been defined as END-IF, END-PERFORM instead of Period (.).
- Apart from 24-bit addressing, it can generate object code with 31-bit address. This means Programs compiled by VS COBOL II can execute in the 31-bit addressing mode of an operating system that supports extended addressing. With 31-bit addressing, you have more freedom to define or refer to larger data areas, files, and tables, and to create a larger overall program. Your program and its data are no longer constrained to fit in a 16-megabyte address space.
- It can allocate 128MB memory for Working-Storage where as OS/VS COBOL allows only 1MB for the same. Same rule applies for Linkage-section also.
- POINTER, EXTERNAL, GLOBAL data items have been introduced in VS COBOL II.
- Dynamic CALL has been introduced.
- Array elements can be initialized at the Declaration statement using VALUE clause. This is not supported in OS/VS COBOL.
- COBOL II does not support alternate indexes on an ESDS files.
- Nested Programming has been introduced in VS COBOL II.
- Some of the clauses of OS/VS COBOL have been made obsolete; if you code those clauses the VS COBOL II compiler will treat them as Comment line. The examples of such clauses are AUTHER, DATA-WRITTEN, INSTALLATION and PROCESSING MODE etc.
- The use of FILLER is no longer needed. We can declare variables like:

```
01  WS-REC.
05 WS-DATA  PIC X(75).
05           PIC X(05).<-------- Here we are omitting FILLER
```

- VS COBOL II has its own debugging tool called COBTEST that allows you to trace the execution of a program. In addition, it provides improved Compiler Listing and improved dump format to aid in debugging. There is an interactive debugging tool available to debug OS/VS COBOL program, but it's a separately licensed product.
- VS COBOL II allows specifying a program as REENTRANT. This means, only one copy of the program is needed to satisfy multiple request for that program, saving both storage and processing time.

COBOL/370:

Before talk about COBOL/370, let's know what is **LE/370**, because LE/370 has to be installed in the machine to let the COBOL/370 work.

LE/370 establishes a common run-time environment for different programming languages. It combines essential run-time services, such as condition handling and storage management. With LE/370, you can use one run-time environment for your applications, regardless of the application's programming language or system resource needs. DB2 on MVS uses LE/370 to provide a run-time environment for the stored procedure programs. You can have stored procedures written in different languages. All of these stored procedures can execute in the same stored procedures address space. Thus, using LE/370, you do not have to specify the language specific libraries in the JCL procedure of the stored procedures address space; it is enough to have the LE/370 run-time library.

Features of COBOL/370

- Supports all the features of VS COBOL II.
- PROCEDURE POINTER data item has been introduced. It allows passing the address of one program to another program.
- INTRINSIC functions (e.g. MAX, MIN, MEAN, RANGE, ASIGN, ACOS, ATAN) have been introduced in the COBOL/370. This feature is available in any other HLL language like C, FORTRAN.
- LE/370 CALLable Services are supported by COBOL/370. Example of some of those Callable are CEE3ABD (to abend the program), CEE3DMP (to generate dump to assist in dubuging), CEETEST (to invoke the debug tool).

Object-Oriented COBOL:

This version of COBOL supports Object oriented features. It provides ability to define class objects, methods, define data encapsulated inside class objects and objects and other OO features. Most importantly, it provides the ability to use objects as a normal part of COBOL programming in developing new programs and maintaining existing programs.

Appendix – F

Introduction

This Appendix includes the complete set of JCLs required for the running the sample programs given in the course material on the MVS platform. In order to do so the programs must be first compiled using the command TRNGCOB. This is the macro created for the trainees in Infosys to compile the COBOL programs. After compiling the programs one can code the JCLs and submit the JCL using SUB or SUBMIT command. Note that the jobcard has to changed before submitting the JCLs. The output can be viewed using the SDSF.ST option from the primary menu of ISPF.

JCLs for running sample programs.

1. **JCL for Program 1.8.**

```
//IN10878J JOB ,,NOTIFY=&SYSUID,CLASS=D,MSGLEVEL=(1,1),MSGCLASS=X
//STEP001 EXEC PGM=HELLO
//STEPLIB DD DSN=OPERN.CICS3.LOADLIB,DISP=SHR
```

2. **JCL for Program 2.5.**

```
//IN10878J JOB ,,NOTIFY=&SYSUID,CLASS=D,MSGLEVEL=(1,1),MSGCLASS=X
//ST01 EXEC PGM=ADDITION
//STEPLIB DD DSNAME=OPERN.CICS3.LOADLIB,DISP=SHR
//SYSIN DD *
22
75
/*
```

3. **JCL for Program 3.5.1.**

```
//IN10878J JOB ,,NOTIFY=&SYSUID,CLASS=D,MSGLEVEL=(1,1),MSGCLASS=X
//ST01 EXEC PGM=SMALL
//STEPLIB DD DSNAME=OPERN.CICS3.LOADLIB,DISP=SHR
//SYSIN DD *
3
1
5
2
4
/*
```

4. JCL for Program 3.5.1.

```
//IN10878J JOB ,,NOTIFY=&SYSUID,CLASS=D,MSGLEVEL=(1,1),MSGCLASS=X
//ST01 EXEC PGM=MARKS
//STEPLIB DD DSNAME=OPERN.CICS3.LOADLIB,DISP=SHR
//SYSIN DD *
75
68
61
```

5. JCL for Program 4.6.1.

```
//IN10878J JOB ,,NOTIFY=&SYSUID,CLASS=D,MSGLEVEL=(1,1),MSGCLASS=X
//ST01 EXEC PGM=SEQFILE
//STEPLIB DD DSNAME=OPERN.CICS3.LOADLIB,DISP=SHR
//SYSIN DD *
03
BU101
ROOPA
21
BU102
DEEPTHA
21
BU103
BHAVANA
20
//STUDENT DD DSN=IN10878.SESSION3.COBOL(STUDENT),DISP=SHR
```

6. JCL for Program 4.6.2.

```
//IN10878J JOB ,,NOTIFY=&SYSUID,CLASS=D,MSGLEVEL=(1,1),MSGCLASS=X
//ST01 EXEC PGM=UPDATE
//STEPLIB DD DSNAME=OPERN.CICS3.LOADLIB,DISP=SHR
//SYSOUT DD SYSOUT=*
//STUDENT DD DSN=IN10878.SESSION3.COBOL(STUDENT),DISP=SHR
//SYSIN DD *
BU102
DEEPTHA
22
/*
```

7. **JCL for Program 4.8.1.**

```
//IN10878J JOB ,,NOTIFY=&SYSUID,CLASS=D,MSGLEVEL=(1,1),MSGCLASS=X
//ST01 EXEC PGM=ADDVSAM
//STEPLIB DD DSNAME=OPERN.CICS3.LOADLIB,DISP=SHR
//SYSOUT DD SYSOUT=*
//STUDENT DD DSN=IN10878.STUDDATA.CLUSTER,DISP=SHR
//SYSIN DD *
ST109
RAJATH
20
/*
```

8. JCL for Program 4.8.2.

```
//IN10878J JOB ,,NOTIFY=&SYSUID,CLASS=D,MSGLEVEL=(1,1),MSGCLASS=X
//ST01 EXEC PGM=DELVSAM
//STEPLIB DD DSNAME=OPERN.CICS3.LOADLIB,DISP=SHR
//SYSOUT DD SYSOUT=*
//STUDENT DD DSN=IN10878.STUDDATA.CLUSTER,DISP=SHR
//SYSIN DD *
ST103
/*
```

9. JCL for Program 5.7.1.

```
//IN10878J JOB ,,NOTIFY=&SYSUID,CLASS=D,MSGLEVEL=(1,1),MSGCLASS=X
//ST01 EXEC PGM=ARREX1
//STEPLIB DD DSNAME=OPERN.CICS3.LOADLIB,DISP=SHR
//SYSIN DD *
01
02
03
04
05
06
07
08
09
10
/*
```

10. JCL for Program 5.7.2.

```
//IN10878J JOB ,,NOTIFY=&SYSUID,CLASS=D,MSGLEVEL=(1,1),MSGCLASS=X
//ST01 EXEC PGM=ARREX2
//STEPLIB DD DSNAME=OPERN.CICS3.LOADLIB,DISP=SHR
//SYSIN DD *
4
7
6
2
1
9
5
8
/*
```

11. JCL for Program 6.3.1.

```
//IN10878J JOB ,,NOTIFY=&SYSUID,CLASS=D,MSGLEVEL=(1,1),MSGCLASS=X
//ST01 EXEC PGM=ADDPROD
//STEPLIB DD DSNAME=OPERN.CICS3.LOADLIB,DISP=SHR
//SYSIN DD *
2
03
04
/*
```

Index

S

T

U

V

W