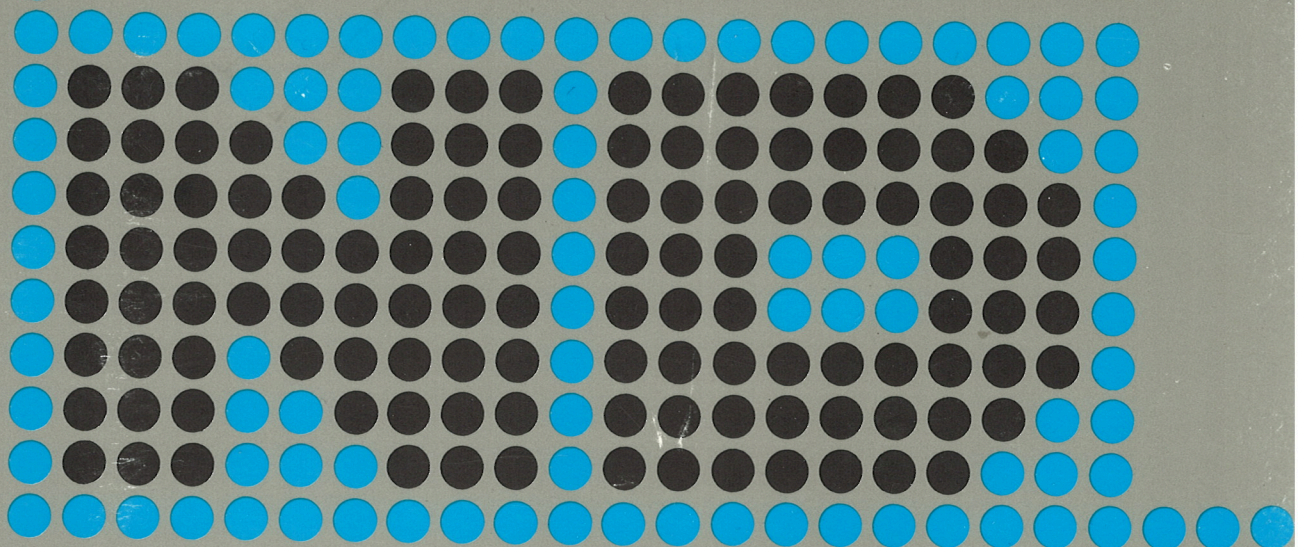
Norsk Data

ND-COBOL Reference Manual

ND-60.144.3 EN



ND-COBOL
Reference Manual

ND-60.144.3 EN

NOTICE

The information in this document is subject to change without notice. Norsk Data A.S assumes no responsibility for any errors that may appear in this document. Norsk Data A.S assumes no responsibility for the use or reliability of its software on equipment that is not furnished or supported by Norsk Data A.S.

PRINTING RECORD

Printing	Notes
01/81	Version 01
03/82	Revision A:
	The following pages have been revised:
	x, xi, xiii,
	1–1, 2–10, 4–1, 4–2, 4–11, 5–30, 6–10, 6–12, 6–22, 6–25, 6–25a, 6–27, 6–80,
	6–90, A–11, A–12
	Index pages 1 through 12.
	The following pages have been added:
	H–1, H–2.
08/82	Version 02
07/83	Revision A:
	The following pages have been revised:
	v, vi, xv,
	1–4, 2–8, 5–5, 5–9, 5–22, 5–27, 5–31, 5–32, 5–34, 6–23, 6–24, 6–25, 6–26,
	6–27, 6–28, 6–29, 6–30, 6–31, 6–32, 6–33, 6–34, 6–35, 6–36, 6–44, 6–44a,
	6–44b, 6–44c, 6–44d, 6–44e, 6–53, 6–55, /-56, 6–70, 6–101, 6–102, 6–103,
	6–104, 6–105, 6–107, 6–110, 9–4, 9–5,
	A–1, A–2, A–3, A–4, A–5, A–6, A–7, A–8, A–9, A-10, A–11, A-12, A–13, A–14
	A–15, A–16, A–17, A–18, A–19, A–20, A–21, A–22, A–23, A–24, A–25, A–26,
	A–27, A–28, A–29, A–30, A–31, D–1, D–2, E–1, F–1, F–2, F–3, H–2,
	–1–, –2–, –3–, –4–, –5–, –6–, –7–, –8–, –9–, –10–, –11–, –12–
06/84	Revision B:
	The following pages have been revised:
	iv, v, vii, xv
	1–2, 1–3, 1–5, 6–28, 6–33, 6–34, 6–35, 6–36, 6–37, to 6–44g, 6–54,
	A–1, A–1a, A–1b, A–9, A–12, A–15, A–16, D–1,
	The following pages have been added:
	J–1, J–2, J–3, J–4, J–5, J–6.
06/85	Version 03

ND COBOL Reference Manual
Publ.No. ND-60.144.3 EN

Norsk Data

Norsk Data A.S
Graphic Center
P.O.Box 25, Bogerud
0621 Oslo 6, Norway

Manuals can be updated in two ways, new versions and revisions. New versions consist of a complete new manual which replaces the old manual. New versions incorporate all revisions since the previous version. Revisions consist of one or more single pages to be merged into the manual by the user, each revised page being listed on the new printing record sent out with the revision. The old printing record should be replaced by the new one.

New versions and revisions are announced in the Customer Support Information (CSI) and can be ordered as described below.

The reader's comments form at the back of this manual can be used both to report errors in the manual and to give an evaluation of the manual. Both detailed and general comments are welcome.

These forms and comments should be sent to:

Documentation Department
Norsk Data A.S
P.O. Box 25, Bogerud
0621 Oslo 6, Norway

Requests for documentation should be sent to the local ND office or (in Norway) to:

Graphic Center
Norsk Data A.S
P.O. Box 25, Bogerud
0621 Oslo 6, Norway

Preface:

THE PRODUCT

COBOL (COmmon Business Oriented Language) is a programming language, based on English, which was developed for use in commercial data processing. The original COBOL specification resulted from the work of the CODASYL (Conference on Data Systems Languages) committee in the U. S. A. in 1959. ND COBOL is based on American National Standard X3.23 - 1974. ND COBOL is COBOL for both the ND-100 and the ND-500. Differences, where they occur, are described in the text.

This manual describes ND COBOL, ND-10176, version H for the ND-100 computer series and ND COBOL, ND-10177, version H for the ND-500 series.

THE READER

The manual is written for the programmer using ND COBOL who requires a detailed and formal explanation of the product as well as an account of the features and facilities available to the user.

PREREQUISITE KNOWLEDGE

A basic knowledge of data processing techniques is necessary for the reader and some familiarity with COBOL would be helpful. The reader should also have some knowledge of the SINTRAN III operating system.

HOW TO USE THE MANUAL

The description is given in the order in which the Divisions and Sections appear in the written programs.

The manual is intended for reference purposes and is organized as follows:

Part I of the manual describes ND COBOL in general terms and gives specific rules for writing COBOL source programs. There is a chapter for each COBOL division. Part II contains an account of each "other feature" or special topic requiring a section of its own. Supplemental information is given in appendixes at the end.

ACKNOWLEDGEMENT

Any organization interested in reproducing the COBOL standard and specifications in whole or in part, using ideas from this document as the basis for an instruction manual or for any other purpose, is free to do so. However, all such organizations are requested to reproduce the following acknowledgement paragraphs in their entirety as part of the preface to any such publication. (Any organization using a short passage from this document, such as in a book review, is requested to mention "COBOL" in acknowledgement of the source, but need not quote the acknowledgement.)

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

No warranty, expressed or implied, is made by any contributer or by the CODASYL Programming Language Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

The authors and copyright holders of the copyrighted material used herein:

> FLOW-MATIC (trademark of Sperry Rand Corporation), Programming for the UNIVAC® I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator Form No. F 28-8013, copyrighted 1959 by IBM; FACT, DSI 27 A5260-2760, copyrighted 1960 by Minneapolis-Honeywell

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

FORMAT NOTATION

Basic formats are prescribed in the manual for the elements of the COBOL language. The notation described here is used to define unambiguously for the programmer how the clauses and statements of COBOL should be written.

RESERVED WORDS

COBOL has a specified list of words for use in source programs which have preassigned meanings and cannot appear in programs as user-defined words or system names. A complete list of the reserved words can be found in appendix 4.

Reserved words may be divided into two categories:

Key Words

These are required by the syntax of the format. They are always in upper case and underlined.

Optional Words

As their name implies, they may be included or omitted without changing the syntax of the program. They appear in upper case but are not underlined.

Words printed in lowercase letters represent information to be supplied by the programmer. All such words are defined within this manual.

The general format is also defined by the use of the following symbols:

- *Braces ({ }).* These enclose vertically stacked items and indicate that one of the enclosed items must appear.

- *Brackets ([ ]).* Square brackets are used to show that the enclosed item is optional, depending on the requirements of the program.

- *Ellipsis (...).* These dots specify that the immediately preceding unit may occur any number of times in succession at the user's option.

The arithmetic and logical operators (+, -, >, <, =). When they appear in formats they are required items even though they are not underlined.

Any other punctuation or special characters which appear in general formats indicate the actual occurrence of these characters and are required by the syntax.

NEW FEATURES IN THIS MANUAL

In this release of the manual, the following features have been changed or included:

1) The scope of descriptions of how to compile and load has been broadened. The treatment of the ND-500 compiler is more extensive, and introductory overviews of the linker-loaders (NRL and BRF-Linker for the ND-100, and the Linkage-Loader for the ND-500) are included

2) New screen handling facilities associated with the DISPLAY and ACCEPT verbs have been described

3) A new chapter containing examples of how how to build overlay program systems on the ND-100 computers, how to use files and how to call subprograms has been included

4) An appendix giving details about the new COBOL System variables has been included

5) The index has been entirely reworked, and much extended. The intention is to ease access to related information. For example, the relative file related information is described in chapters of the manual which are dedicated to different COBOL divisions. In the index, "pointers" to places where relative file usage is described can be found grouped together. Another example is using the index to find extentions in ND-COBOL as compared to ANSI COBOL.

   The index now refers to page numbers instead of section numbers.

TABLE OF CONTENTS

Section Page

Section Page

Section Page

Section Page

APPENDIX

1 INTRODUCTION

The purpose of this chapter is to give an overview of the ND COBOL compiler - how it conforms to the ANSI standard, what is needed to create and run COBOL programs, and how the compiler is used together with the ND linkage-loaders to form executable programs.

1.1 ND COBOL

ND COBOL is a standard high-level language implemented as a conventional compiler and runtime library system operating under SINTRAN III/VS operating system.

ND COBOL is based upon American National Standard X3-23-1974. Elements of the COBOL language are allocated to 12 different functional processing "modules".

Each module of the COBOL Standard has two "levels" - level 1 represents a subset of the full set of capabilities and features contained in level 2.

In order for a given system to be called COBOL, it must provide at least level 1 of the Nucleus, Table Handling and Sequential I-O modules.

The following summary specifies the contents of ND COBOL with respect to the ANSI Standard:

Module	Features Available in ND COBOL
Nucleus	All of level 1 and level 2 except: level 66 the RENAMES clause the switch-status condition the ENTER statement. Additional features are: USAGE is COMPUTATIONAL-1 USAGE is COMPUTATIONAL-2 USAGE is COMPUTATIONAL-3 ACCEPT FROM CPU-TIME. The DO statement. The IF statement with the THEN, ELSE-IF and END-IF clauses. The IMPORT and EXPORT clauses for inter-program communication.
Terminal I-O	Additional features are: The BLANK statement. The ACCEPT-ERROR statement. The ACCEPT and DISPLAY statements with Screenhandling options.
Sequential I-O	All of level 1 and level 2 except: the RERUN the LINAGE and CODE SET clauses with the addition of: the RECORDING MODE clause.
Indexed I-O and Relative I-O	All of level 1 and level 2 except: the RERUN and the SAME RECORD AREA clauses with the addition of: The RECORDING MODE clause. The OPEN statement with the MULTI-USER MODE, IMMEDIATE-WRITE and MANUAL UNLOCK options. The READ statement with LOCK. The UNLOCK statement.
Table Handling	All of level 1 and level 2.
Sort/Merge	All of level 1 and level 2 except: the SAME AREA clause.
Inter-Program Communication	All of level 1 and level 2 except: the CANCEL statement.
Debugging	Conditional compilation: Lines with 'D in column 7' are bypassed unless WITH DEBUGGING MODE.

1.2 OPERATIONAL REQUIREMENTS

The compiler may execute as a reentrant subsystem under the SINTRAN III/VS operating system, when only the necessary 1 kiloword pages are brought into the memory as needed. In this way, several active users may share a common code.

A system scratch file for the active terminal will be used to store compiler information.

The source program is accepted in any media supported by the ND File System, and may be entered and modified using an interactive editor. Once entered, source files are stored on disk, floppy diskette or magnetic tape and can be compiled by using simple compiler commands.

On the ND-10 a special microprogram is required.

The result of a compilation is:

A) A source listing including compiler assigned line numbers, source file name, object file name, date and time.

B) In the event of any source program errors (or warnings), diagnostic messages will appear following the source listing. These messages have the format:

- Line number (5 digits)
- English message text
- (Optional) Further relevant data

C) An object program in library relocatable form (BRF on the ND-100 or NRF on the ND-500) can be used by the ND Relocating Loader for the ND-100 or the ND-500 Linkage Loader for the ND-500, to prepare the object program in a form which is ready for execution.

1.2.1 Known Restrictions

For the time being the following restrictions are applicable on the ND-100 CPUs:

- A 77/01 item must not be greater than 32767 bytes.

1.3 HOW TO USE THE SYSTEM

In addition to the information given in this chapter, a complete example of the compilation, loading and execution of a simple program with some of the features of ND COBOL is shown in chapter 11.

1.3.1 How to Compile a COBOL Program

The COBOL compilers are started by typing:

@COBOL for the ND-100 series

or

@ND COBOL-500 for the ND-500 series

When the compiler has printed * (asterisk) on the terminal, it is ready to accept commands from the user.

All commands may be abbreviated as in SINTRAN III/VS. You can also use the SINTRAN-III command editing characters when in the compilers.

The command to compile is:

COMPILE <source file>,<list file>,<object file>

The source file is your symbolic program containing COBOL statements. A listing of the program is written on the list file while the object program in binary relocatable format is written on the object file.

The files must be specified by their names and these names must be delimited by at least one space or comma. The default source file type is :SYMB. The list file type is :SYMB and the object file type is :BRF on the ND-100 or :NRF on the ND-500. (Scratch file 100 cannot be used as the object file.)

If the source input file is not a disk file, a line containing *END (from column 1) must close the source file.

Example:

```
@COBOL

*COMPILE SOURCE,LINE-PRIN,"OBJ"
```

Note that in this example the object file (OBJ) is a new file and therefore is specified within quotes.

On the ND-100, the compiler produces code in the two-bank mode unless the compiler command

```
*1-BANK-MODE
```

is given before the COMPILE command.

If no diagnostics appear, the compiler has accepted all the statements as syntactically and semantically correct.

When compilation has been done, the compiler is left with the command

```
*EXIT
```

The object (or executable) version of the program is then formed from the relocatable output from the compilers and the system's library files by the:

a) ND Relocating Loader (NRL) for the ND-100;

b) BRF-Linker for the ND-100 series;

c) ND-500 Linkage-Loader for the ND-500 series.

1.3.2 Sample Compilation

The listing below results from compiling a source program which has been prepared using a suitable editor. It is stored under the name EX-001 with type :SYMB.

Note: To show what an error message looks like, an error has been introduced in the last statement in this example. The error is an unnecessary hyphen between the words STOP and RUN.

```
ND-500 COBOL COMPILER - ND-10177H  TIME: 17.57.54  DATE: 85.01.14

  SOURCE FILE: EX-001
  OBJECT FILE: EX-001

     1       ************************************************************
     2       * THIS EXAMPLE CAN SERVE TO FAMILIARIZE US WITH THE        *
     3       * RESULT OF A COBOL COMPILATION.                           *
     4       *                                                          *
     5       * THE PROGRAM COUNTS THE NUMBER OF RECORDS ON THE FILE     *
     6       * "ABC:DATA".                                              *
     7       ************************************************************
     8        IDENTIFICATION DIVISION.
     9        PROGRAM-ID.          X-001.
    10        AUTHOR.              NORSK DATA A/S
    11                             NORWAY.
    12        DATE-WRITTEN.        NOVEMBER 1984.
    13
    14        ENVIRONMENT DIVISION.
    15        CONFIGURATION SECTION.
    16        SOURCE-COMPUTER.     NORD-100.
    17        OBJECT-COMPUTER.     NORD-100.
    18        SPECIAL-NAMES.       DECIMAL-POINT IS COMMA.
    19        INPUT-OUTPUT SECTION.
    20        FILE-CONTROL.
    21            SELECT L-FILE ASSIGN TO "ABC:DATA".
    22
    23        DATA DIVISION.
    24        FILE SECTION.
    25        FD  L-FILE
    26            BLOCK CONTAINS 1 RECORDS
    27            RECORD CONTAINS 10000 CHARACTERS.
    28        01  L-RECORD         PIC X(10000).
    29        WORKING-STORAGE SECTION.
    30        01  NUMBER-OF-RECORDS PIC 9(10) VALUE 0.
```

```
      31
      32        PROCEDURE DIVISION.
      33        1000.
      34            OPEN INPUT L-FILE.
      35        2000.
      36            READ L-FILE     AT END GO TO 9000.
      37            ADD 1           TO NUMBER-OF-RECORDS.
      38            GO TO 2000.
      39        9000.
      40            DISPLAY "NUMBER OF RECORDS IN THE FILE IS "
      41                    NUMBER-OF-RECORDS.
      42            CLOSE L-FILE.
      43            STOP-RUN.

     43 E - SYNTAX ERROR (RESUMPTION AT NEXT PARAGRAPH/VERB): STOP-RUN
--- END OF COMPILATION ----------
NUMBER OF ERRORS FOUND:         1
NUMBER OF WARNINGS GIVEN:       0
NUMBER OF SOURCE LINES:        43
LINES/MINUTE (CPU TIME):     6789
---------------------------------
```

Note the following in the compilation listing:

- The page heading contains date, time, source file name (EX-001) and object file name.
- The source line (first 80 positions only) is listed along with compiler assigned line numbers.
- Diagnostic or warning messages, if any, appear after the source program listing.

The error in the example:

43 E - SYNTAX ERROR (RESUMPTION AT NEXT PARAGRAPH/VERB): STOP-RUN

produces the relevant line number (43) together with explanatory text and the element itself which caused the error.

After successful compilation, the next step will be to link-edit by means of one of the linkage-loaders. Finally, the resultant program module will be executed.

1.3.3 How to Load and Execute a COBOL Program

1.3.3.1 Loading on the ND-100 Computers with the NRL

On the ND-100, the ND Relocating Loader may be recovered from the operating system by entering:

@NRL

When the loader has displayed an asterisk (*) on the terminal, it is ready to accept commands from the user.

Your program(s) may be loaded into a programfile or into main memory.

If you use the command:

**PROG-FILE <file name>*

then the executable program is loaded to the file named with the default extension :PROG. The PROG-FILE must be the first command given after the loader has been started.

Otherwise, the program is loaded into memory, and can be started with the command

**RUN*

after the loading is completed.

The loader gets its input from one or more files/library files. The loading is initiated by the command:

**LOAD <file name>,<file name> ...*

Each of the files specified will be loaded until end-of-file is detected. The prompt * (asterisk) indicates that the loader is ready to accept another command.

To obtain the entry point addressesof the loaded program, use the command:

**ENTRIES-DEFINED <file name>*

The octal addresses which appear on this map denote the last reference address.

There should be no remaining undefined entry points. If your program is loaded into main memory, it may be started by the command:

**RUN*

When the program has been executed, control is transferred to the operating system and @ (commercial at sign) is displayed.

If you wish to leave the loader and enter the operating system, you may simply enter:

**EXIT*

You may restart the loader by using the system command:

@CONTINUE

If the message:

LOADER TABLE OVERFLOW

is given, there is no more room for entries. The table length may be expanded through the command:

**SIZE <number of entries (octal)>*

The NRL is fully documented in the manual *ND Relocating Loader, ND-60.066.*

1.3.3.2 Loading on the ND-100 Computers with the BRF-Linker

The BRF-Linker is a new linker which is taking over from the NRL on ND-100 computers. It performs the same tasks as the NRL, but it is designed to facilitate use of future features of the ND-100 SINTRAN operating system.

The BRF-linker does not have a *RUN* command for immediate execution of programs loaded into memory. Thus, the *PROGRAM-FILE* command must always be your first command when you build executable code. On the other hand, the BRF-Linker makes "multi-segment linking" of programs bigger than the ordinary 256 kilobyte addressing space limit possible, and overlay linking is possible in a more flexible way than with the NRL.

For details on these and other aspects of the BRF-Linker, see the manual *BRF-Linker User Manual, ND-60.196.*

The BRF-Linker is started by typing:

@BRF-LINKER

when you are in the SINTRAN command mode. When the linker is ready to accept your commands, it types the prompt:

Brl:

on your terminal.

For linking smaller programs, the BRF-Linker uses the same or similar commands as the NRL. For instance, every loading of a program must start with the command:

Brl: PROGRAM-FILE <file name>

as in the NRL.Relocatable code files are loaded to the program file by the command:

Brl: LOAD <file name> [, <file name>] ...

until the program is complete. In COBOL on the ND-100, the COBOL-2BANK or COBOL-1BANK files containing COBOL library routines must be loaded after your own programs in order to form a complete program.

The completeness of the program is checked by the command:

Brl: LIST-ENTRIES-UNDEFINED.

If this command gives names of entities (routines, variables etc.) that are needed to form a correct program, the loading is still incomplete; otherwise, an executable program has been built.

This command has a complementing command:

Brl: LIST-ENTRIES-DEFINED

which gives an overview of the procedures, data etc. that you have defined so far in the loading process, together with the addresses of these entities in octal.

To end the loading session, give the command:

Br1: EXIT

To start your new program, type the name of its program file after you have returned to SINTRAN.

1.3.3.3 Loading on the ND-500 Computers with the LINKAGE-LOADER

The ND-500 computers use an ND-100 for I-O processing etc.The SINTRAN III operating system is much used by these computers. For instance, executable ND-500 programs are stored in files which have extensions :PSEG, :DSEG and :LINK as seen from the ND-100 part of the system. A file named DESCRIPTION-FILE:DESC contains descriptions of all ND-500 programs a user has on his area.

You should not delete or otherwise change any of these files while under the SINTRAN III monitor. If you do, you will get error messages from the ND-500 if you try to use any of your ND-500 programs afterwards. Instead, all handling of these files should be done while in the ND-500 Linkage-Loader (NLL) or in the ND-500 monitor.

Further differences from the ND-100 COBOL include the possibility to load programs with up to 2 Gigabytes of program and 2 Gigabytes of data; thus, no overlay linking facilities should be necessary at present.

The user will fing complete documentation on the NLL in the manual: *ND-500 Loader/Monitor, ND-60.136.*

On the ND-500, executable programs are built on *segments* belonging to *domains* by the ND Linkage-Loader (known as the NLL). The NLL is activated by the command:

@ND-500 LINKAGE-LOADER

When the NLL types NLL:, it is ready to accept commands from the user.

The NLL will create a program ready for execution. On the ND-500 a program is termed a domain. Before the domain is loaded, it is named by the command:

NLL:SET-DOMAIN <domain-name>

The *domain-name* is the name you type in when you want to execute your program. The domain name will not be visible when you do a @LIST-FILES in ND-100, for instance - all information pertaining to it is kept on the DESCRIPTION FILE. What *will* turn up when you do @LIST-FILES is the segments you create during loading in a domain, with the :PSEG, :DSEG and :LINK file types, as mentioned above. To create or open a segment for your executable code, use the command:

NLL:OPEN-SEGMENT <segment-name>,<attributes>

If the segment is new to your user area, you will have to enclose it in quotation marks according to the ordinary SINTRAN conventions. The attributes part of this command format can be omitted when loading simple programs.

Now programs and library files can be loaded into the segment by the command:

NLL:LOAD-SEGMENT <file> [,<file>] ...

To build a functioning ND-500 COBOL program, you must finish the loading by loading the file COBOL-LIB:NRF, which is usually found under user SYSTEM.

To obtain the entry point addresses of the loaded program and thus check whether you have an executable program, use the command:

NLL:LIST-ENTRIES-DEFINED

and undefined entries by:

NLL:LIST-ENTRIES-UNDEFINED

If the user wishes to leave the NLL and return to SINTRAN, she can type the command:

NLL:EXIT

1.3.4 Compilation

Using the same source program and commands as in the previous example, the following commands can be used: (all lines start in column 1)

ON THE ND-100

```
@COBOL
COMPILE PROGRAM, TERM, PROGRAM
EXIT
@NRL (or @BRF-LINKER)
PROG-FILE PROGRAM
LOAD PROGRAM, COBOL-1BANK
EXIT
@PROGRAM
```

ON THE ND-500

```
@ND COBOL
COMPILE PROGRAM, TERM, PROGRAM
EXIT
@ND LINKAGE LOADER
SET-DOMAIN PROGRAM
OPEN-SEGMENT PROGRAM
LOAD-SEGMENT PROGRAM,, COBOL-LIB
EXIT
@ND PROGRAM
```

2 LANGUAGE CONCEPTS

2.1 THE STRUCTURE OF COBOL

Every COBOL program is divided into four divisions. Each must be placed in its proper sequence and begin with a division header.

2.1.1 The COBOL Divisions

The four divisions of a COBOL source program and their functions are:

- *Identification Division*

  This names the program and, optionally, documents the compilation date, etc.

- *Environment Division*

  This describes the computer(s) and equipment to be used by the program. It also includes a description of the relationship between the files containing data and the input-output devices.

- *Data Division*

  This defines the names and characteristics of all the data to be processed by the program.

- *Procedure Division*

  This consists of executable statements that direct the processing of data at execution time.

2.1.2 Structure within the Divisions - Clauses and Statements

A *clause* specifies the attributes of an entry which, containing a series of clauses ending with a period, can appear in each division except the procedure division.

A *statement*, written in the procedure division, specifies an action to be taken by the object program. A series of statements, ending with a period, is defined as a sentence.

Every clause or statement in the program may be further subdivided into units called *phrases* or *options*. A phrase is an ordered set of one or more COBOL character strings forming a part of a clause or statement. An option is a phrase in which the programmer can choose between alternative wordings, according to the meanings he wishes the phrase to possess.

Clauses, entries, statements and sentences may be combined into paragraphs and sections which each define a larger part of the program. A section may itself contain paragraphs.

2.2 STRUCTURE OF THE LANGUAGE

2.2.1 COBOL Character Set

The most basic and indivisible unit of the language is the character. The set of characters used to form COBOL character strings and separators is given below.

The complete COBOL character set consists of the 52 following characters:

Character:	*Meaning:*
0, 1, ..., 9	*digit*
A, B, ..., Z	*letter*
	space (blank)
+	*plus sign*
-	*minus sign (hyphen)*
*	*asterisk*
/	*stroke (virgule, slash)*
=	*equal sign*
$	*currency sign*
,	*comma (decimal point)*
.	*period (decimal point)*
;	*semicolon*
"	*quotation mark (double)*
(	*left parenthesis*
)	*right parenthesis*
>	*greater than*
<	*less than*
'	*apostrophe (single quotation mark)*

Note that a reference to 'characters' throughout this manual will be to a subset of the above list, i.e., the list not including 'separators' (defined in section 2.2.3.4).

2.2.2 Character-Strings

A character-string is a character or sequence of contiguous characters which form a COBOL word, a literal, a PICTURE character-string or a comment entry. A character-string is delimited by separators.

2.2.3 COBOL Words

A COBOL word can be a userdefined word, a system word or a reserved word. Its maximum length is 30 characters. System words and reserved words are defined as follows.

2.2.3.1 Userdefined Words

These are COBOL words supplied by the programmer. Characters valid in a userdefined word are:

A through Z
0 through 9
- (hyphen)

The hyphen may not be the first or last character. A list of the sets of userdefined words together with their formation rules is given below.

Userdefined Word Set:	*Characteristics:*
condition name *data name* *record name* *file name* *index name* *mnemonic name*	*Must contain at least one alphabetic character. Within each set the name must be unique. (It can be made unique by qualification if the format rules for the set permit.)*
library name *program name* *routine name*	*The above rules apply.*
paragraph name *section name*	*May be all numeric, otherwise rules in paragraph 1 apply.*

The function of each userdefined word in any clause or statement will be found under the description for that clause or statement.

The function of each system name (Norsk Data defined names for communication with the operating system) will be found in the Glossary.

2.2.3.2 Reserved Words

Reserved words may be divided into the following categories:

1. *KEY WORDS*
2. *OPTIONAL WORDS*
3. *CONNECTIVES*
4. *SPECIAL REGISTERS*
5. *FIGURATIVE CONSTANTS*
6. *SPECIAL CHARACTER WORDS*

A reserved word is a COBOL word having a fixed meaning and it must not be used as a userdefined word or system name. A list is given in Appendix 4.

KEY WORDS

A *key word* is required when the format in which it appears is used in a source program. Within each format, such words are uppercase and underlined.

Key words are of three types:

1) Verbs such as ADD, READ and MOVE.

2) Required words, which appear in statement and entry formats.

3) Words which have a specific functional meaning such as NEGATIVE, SECTION, etc.

OPTIONAL WORDS

Within each format, uppercase words that are not underlined are called *optional words* and may appear at the user's option. The presence or absence of an optional word does not alter the semantics of the COBOL program in which it appears.

CONNECTIVES

These are:

1) Qualifier connectives that are used to associate a data name, a condition name, a text name or a paragraph name with its qualifier: OF, IN.

2) Series connectives that link two or more consecutive operations: (separator comma) or ; (separator semicolon).

3) Logical connectives that are used in the formation of conditions: AND, OR.

SPECIAL REGISTERS

Each compiler generated storage area whose primary function is to store information produced by one of the specific COBOL features, is a *special register.*

Examples:

DATE, DAY, TIME

(see ACCEPT statement in the Procedure Division).

FIGURATIVE CONSTANTS

Certain reserved words are used to name and reference certain constant values which will be generated by the compiler when these words are used. Known as *figurative constants* they must not be bounded by quotation marks. Singular and plural forms may be used interchangeably.

The reserved words and the figurative constant values they generate are listed on the following page.

ZERO, ZEROS, ZEROES	*Represents the value '0' or one or more of the characters '0', depending on context.*
SPACE, SPACES	*Represents one or more of the character ' ' (space) from the computer's character set.*
HIGH-VALUE, HIGH-VALUES	*Represents one or more of the characters that has the highest ordinal position in the program collating sequence.*
LOW-VALUE, LOW-VALUES	*Represents one or more of the characters that has the lowest ordinal position in the program collating sequence.*
QUOTE, QUOTES	*Represents one or more of the characters '"'. The word QUOTE or QUOTES cannot be used in place of a quotation mark in a source program to bound a nonnumeric literal. Thus, QUOTE ABD QUOTE is incorrect as a way of stating the nonnumeric literal "ABD".*
ALL literal	*Represents one or more of the string of characters comprising the literal. The literal must be either a nonnumeric literal of one character length or a figurative constant other than ALL literal. When a figurative constant is used, the word ALL is redundant and is used for readability only.*

When a figurative constant represents a string of one or more characters, the length of the string is determined by the compiler from the context, according to the following rules:

1) When a figurative constant is associated with another data item (e.g., is moved to or compared with another item) the string of characters composing the figurative constant is repeated character by character on the right until the size of the resultant string in characters is equal to that of the associated data item. This is done prior to and independent of any application of a JUSTIFIED clause associated with the data item.

2) When a figurative constant is not associated with another data item, as when the figurative constant appears in a DISPLAY, STRING, STOP or UNSTRING statement, the length of the string is one character.

A figurative constant may be used wherever a literal appears in the format, except that whenever the literal is restricted to having only numeric characters in it, the only figurative constant permitted is ZERO (ZEROS, ZEROES).

Each reserved word which is used to reference a figurative constant value is a distinct character string with the exception of the construction 'ALL literal' which is composed of two distinct character strings.

SPECIAL CHARACTER WORDS

These are the arithmetic operators (+ - / * or **) or the relational characters (< > =). They are described under arithmetic expressions and conditional expressions in the Procedure Division.

2.2.3.3 Literals

A *literal* is a character string with a value specified either by the ordered set of characters by which it is composed, or by a figurative constant. There are two types of literals: nonnumeric and numeric.

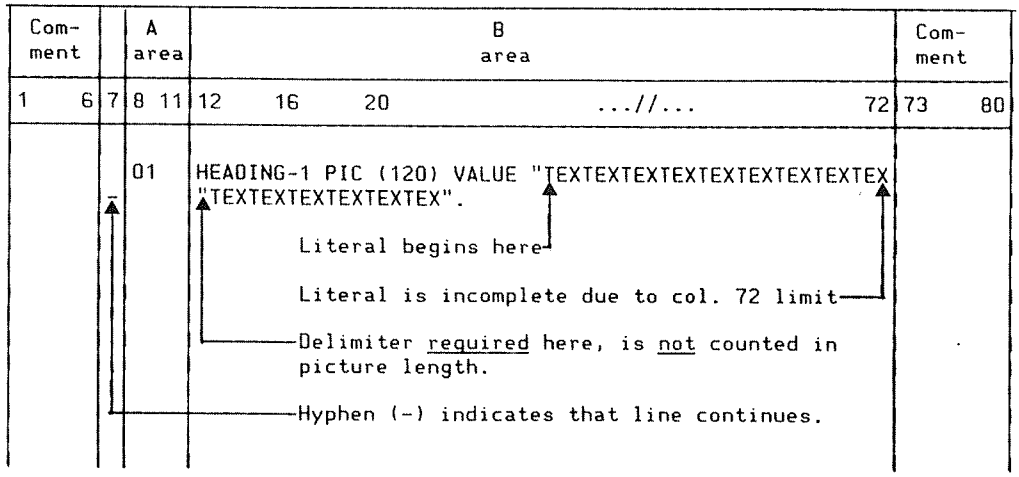
A *nonnumeric literal* is a character string bounded by quotation marks containing any allowable character from the ASCII character set. Its maximum length is 150.

Any punctuation characters included within a character string are part of its value.

A matching pair of either single or double quotes is allowed to bound the character string forming a nonnumeric literal. If the character string is bounded by single quotes, then each embedded quotation mark must be represented by a pair of single quotes. If, however, the bounds are double quotes then each embedded quotation mark must be represented by a pair of double quotes.

Single quotes are not standard COBOL. They are allowed in ND COBOL to increase compatibility with other COBOL systems.

Nonnumeric literals - a coding example:



A *numeric literal* is a character string whose characters are selected from the digits '0' through '9', the plus sign, the minus sign and/or the decimal point. The rules for formation of numeric literals are as follows:

1) A literal must contain at least one digit.

2) 1 through 18 digits are allowed.

3) A literal must not contain more than one sign. The sign must always be in the leftmost position.

4) It must not contain more than one decimal point. This must not be in the rightmost position.

2.2.3.4 Separators

A separator is a character or two contiguous characters formed according to the following rules:

1) The punctuation character space is a separator. Anywhere a space is used as a separator or as part of a separator, more than one space may be used. All spaces immediately following the separators comma, semicolon or period are considered part of that separator and are not considered to be the separator space.

2) Except when the comma is used in a PICTURE character-string, the punctuation characters comma and semicolon, immediately followed by a space, are separators that may be used anywhere the separator space is used. They may be used to improve program readability.

3) The punctuation character period, when followed by a space is a separator. It may only be used to indicate the end of a sentence, or as shown in formats.

4) The punctuation characters right and left parenthesis are separators. Parentheses may appear only in balanced pairs of left and right parentheses delimiting subscripts, reference modifiers, arithmetic expressions, Boolean expressions, or conditions.

5) The punctuation character quotation mark is a separator. An opening quotation mark must be immediately preceded by a space or left parenthesis; a closing quotation mark, when paired with an opening quotation mark, must be immediately followed by one of the separators space, comma, semicolon, period, or right parenthesis.

6) The separator space may optionally immediately precede all separators except:

   The separator closing quotation mark. In this case, a preceding space is considered as part of the nonnumeric literal and not as a separator.

7) The separator space may optionally immediately follow any separator except the opening quotation mark. In this case, a following space is considered as part of the nonnumeric literal and not as a separator.

Any punctuation character which appears as part of the specification of a PICTURE character-string or numeric literal is not considered as a punctuation character, but rather as a symbol used in the specification of that PICTURE character-string or numeric literal. PICTURE character-strings are delimited only by the separators space, comma, semicolon, or period.

The rules established for the formation of separators do not apply to the characters which comprise the contents of the nonnumeric literals or comment lines.

2.2.4 COBOL Format

COBOL programs must be written in a standard format based on an 80 character line. The output listing of the source program is printed in the same format.

The illustration in this section shows the layout of a coding sheet. The following code rules include a description of the fields within it.

Continuation area (column 7)

This column is used to indicate continuation of words and numeric literals from the previous line to the current one. The symbol used is a hyphen.

If there is no hyphen the preceding line is assumed to be followed by a space.

If there is a hyphen in the continuation area, then the first nonblank character of this line immediately follows the last nonblank character of the preceding line without an intervening space.

If there is a nonnumeric literal in the line to be continued which does not have a closing quotation mark, then all spaces up to and including column 72 are considered to be part of this literal. The continuation line must contain a hyphen in its continuation area and the first nonblank character must now be a quotation mark. (See the coding example of a nonnumeric literal in section 2.2.3.3.)

Area A and area B

These occupy columns 8 through 11 and 12 through 72 respectively. The elements that may begin in area A and the placement of elements that can follow them are given in the following chart.

Sequence Rules for Elements in Areas A and B

Elements in Area A	*Followed by:*	*Elements placed in:*
Division header	*(Procedure Division only)* *USING*	*Area B (same or next line)*
	section header *paragraph header* *DECLARATIVES*	*Area A (next line)*
Section header	*USE statement*	*Area B*
	paragraph header *paragraph name* *(either to follow USE, if specified)*	*Area A (next line)*
paragraph header or paragraph name	*Environment division entry* *Procedure division sentence*	*Area B (same or next line)*
level indicator *level number*	*data name*	*Area B (same or next line)*
DECLARATIVES	*Declaratives section name*	*Area A (next line)*
END DECLARATIVES	*section header*	*Area A (next line)*

Comment Lines

A comment line is any line with an * (asterisk) or / (stroke) in column 7. It may appear on any line following the one containing the identification division header. A comment may be written in areas A or B and contain any characters from the ASCII character set.

The * denotes that the comment is to be printed in the output listing immediately following the last preceding line. The / denotes that the current page of the output listing is to be ejected and that the comment will appear on the first line of the next page.

Coding Sheet Layout

Standard COBOL coding sheets are rarely used when programming on the ND system, as most programmers will "code" via the ND Editors. However, the following layout should be useful as the coding sheet fields are referred to in the text.

Com-ment		A area	B area	Com-ment
1 6	7	8 11	12 16 20 .../.../... 72	73 80

3 THE IDENTIFICATION DIVISION

The Identification Division must be included in every source program. This division names the source program and the object program.

A *source program* is the initial COBOL program. An *object program* is the output from the compilation.

In addition, the user may include in this division information such as the date the program was written, etc.

Format:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. program name.
[AUTHOR. [comment entry] ... ]
[INSTALLATION. [comment entry] ... ]
[DATE-WRITTEN. [comment entry] ... ]
[DATE-COMPILED. [comment entry] ... ]
[SECURITY. [comment entry] ... ]
[REMARKS. [comment entry] ... ]
```

The Identification Division must begin with the reserved words IDENTIFICATION DIVISION followed by a period.

The PROGRAM-ID paragraph gives the name a program is identified by, and it must be the first paragraph in the Identification Division. The other paragraphs are optional.

Use of the DATE-COMPILED paragraph does not produce the compilation date on that line. The date of compilation always appears on the first page of the listing, whether or not this paragraph is present.

All commententries serve only as documentation, the syntax of the program is unaffected by them.

4 THE ENVIRONMENT DIVISION

The Environment Division contains a description of the computer on which the source program is compiled together with the functions that are dependent on its physical characteristics. The presence of the Environment Division is optional.

General Format:

```
ENVIRONMENT DIVISION.
[CONFIGURATION SECTION.
[SOURCE-COMPUTER. computer name [WITH DEBUGGING MODE].]
[OBJECT-COMPUTER. computer name]
    [,SEGMENT-LIMIT IS segment number]]
[SPECIAL-NAMES. [,CURRENCY SIGN IS literal]
    [,DECIMAL-POINT IS COMMA].]
[INPUT-OUTPUT SECTION.
FILE-CONTROL. file control entry [file control entry] ...
[I-O-CONTROL. input-output control entry]]
```

4.1 CONFIGURATION SECTION

The Configuration Section is optional in ND cobol.

4.1.1 SOURCE COMPUTER Paragraph

Format:

```
SOURCE-COMPUTER.
        [ ND-10   ]
        [ ND-100  ]   [WITH DEBUGGING MODE].
        [ ND-500  ]
```

The WITH DEBUGGING MODE clause indicates that all debugging lines are to be compiled. If it is not specified, debugging lines will be compiled as if they were comment lines. Use of this clause does not imply any automatic activation of ND's Symbolic Debugger.

A *debugging line* is any line in a source program with a "D" coded in column 7 (the continuation area).

Each line must be written so that a syntactically correct program results when the debugging lines are compiled into the program. Debugging lines may be continued but each continuation line must contain a "D" in column 7.

Debugging lines may be specified only after the SOURCE-COMPUTER paragraph.

4.1.2 OBJECT COMPUTER Paragraph

Format:

```
OBJECT-COMPUTER.  [ ND-10  ]
                  [ ND-100 ]
                  [ ND-500 ]

    [,SEGMENT-LIMIT IS segment number].
```

The SEGMENT-LIMIT clause is treated by the compiler as comments only.

4.1.3 SPECIAL-NAMES Paragraph

The SPECIAL NAMES paragraph provides a substitute character for the currency symbol and specifies whether the functions of the decimal point and comma are to be exchanged in PICTURE clauses and numeric literals. For the format see the beginning of this chapter.

4.1.3.1 CURRENCY IS Clause

The literal which appears in the CURRENCY SIGN IS literal clause is used in the PICTURE clause to represent the currency symbol. The literal is limited to a single character and must not be one of the following characters:

1) digits 0 through 9

2) alphabetical characters A, B, C, D, L, P, R, S, V, X, Z or the space

3) special characters '*', '+', '_', ',', '.', ';', '(',')', '"', '/', '='.

If this clause is not present, only the currency sign is used in the PICTURE clause.

4.1.3.2 DECIMAL-POINT IS COMMA Clause

When specified, this means that the function of the comma and period are exchanged in the character string of the PICTURE clause and in numeric literals.

4.2 INPUT-OUTPUT SECTION

The input-output section names files and provides specifications for other file related information. Its general format is shown at the beginning of this chapter.

4.2.1 File Processing - Language Concepts

The way in which COBOL files in a program are processed depends on how the data is organized on a file and how this data is to be accessed.

4.2.1.1 Data Organization

This refers to the permanent logical structure of the file and is defined as one of three types:

1) *Sequential organization;*

2) *Indexed organization;*

3) *Relative organization.*

Sequential Organization

With this organization, each record in the file except the first has a unique predecessor record, and each record except the last has a unique successor record. These predecessor/successor relationships are established by the order of the WRITE statements when the file is created. Once established, these relationships do not change, however it is possible to add records to the end of the file. The records may be fixed or variable length.

Example:

```
 1       IDENTIFICATION DIVISION.
 2       PROGRAM-ID.
 3           GENSEQ.
 4      ***************************************************************
 5      *    CREATES SQ-FILES AND LISTS.
 6      ***************************************************************
 7       ENVIRONMENT DIVISION.
 8       INPUT-OUTPUT SECTION.
 9       FILE-CONTROL.
10           SELECT SQ-FILE ASSIGN "COB1:DATA",
11               ORGANIZATION IS SEQUENTIAL,
12               ACCESS IS SEQUENTIAL.
13       DATA DIVISION.
14       FILE SECTION.
15       FD  SQ-FILE.
16       01  M-REC.
17           02          PIC X(10).
18           02 SEQNUM   PIC 9(5), BLANK WHEN ZERO.
19           02          PIC X(5).
20           02          PIC X(40).
21       WORKING-STORAGE SECTION.
22       01  RANDNO      PIC  9(4) PACKED-DECIMAL, VALUE ZERO.
23       01  MAXRAND     PIC S9(4) PACKED-DECIMAL, VALUE 1000.
24       01  NORECS      PIC  9(4) PACKED-DECIMAL.
25       01  RECCNT      PIC  99, COMP, VALUE 0.
26
27       PROCEDURE DIVISION.
28       INIT-01.
29           OPEN OUTPUT SQ-FILE.
30           DISPLAY "CREATE RECORDS?".
31           PERFORM GET-NORECS.
32           PERFORM CRE-SQ-FILE NORECS TIMES.
33      * BUILDS THE INPUT FILE.
34           CLOSE SQ-FILE.
35           DISPLAY "FILE SQ-FILE CREATED.", RECCNT, " RECORDS.".
36           OPEN INPUT SQ-FILE.
37       LIST-FILE-0.
38           MOVE 0 TO RECCNT.
39       LIST-FILE-1.
40           READ SQ-FILE AT END GO TO LIST-END.
41           ADD 1 TO RECCNT.
42           DISPLAY "REC ", RECCNT, ", SEQNUM = ", SEQNUM.
43           GO TO LIST-FILE-1.
44       LIST-END.
45           CLOSE SQ-FILE.
46           DISPLAY "JOB FINISH".
47           STOP RUN.
48       CRE-SQ-FILE.
49           CALL "RND" USING RANDNO, MAXRAND.
50           MOVE ALL "*" TO M-REC.
51           MOVE RANDNO TO SEQNUM.
52           ADD 1 TO RECCNT.
53           DISPLAY "UT REC = ", RECCNT, " KEY = ", SEQNUM.
54           WRITE M-REC.
55       GET-NORECS.
56           ACCEPT NORECS.
57           IF NORECS NOT NUMERIC,
58               DISPLAY "*** NOT NUMERIC DATA ***",
59               GO TO GET-NORECS
60           END-IF.
```

Indexed Organization

A file with this organization is a mass storage file whose records, which may be of fixed or variable length, are accessed by means of a key. Each record can have one or more keys and each key is associated with a particular index held on that file. Each index provides a logical path to the data records, according to the contents of a data item within each record which acts as the *record key* for that index.

The RECORD KEY clause in the file control entry for each file names the *prime record key* for that file. When inserting, updating or deleting records in a file, each record must be identifiable solely by its prime record key. This value must, therefore, be unique and it must not be changed when updating the record.

The ALTERNATE RECORD KEY clause names an alternate record key for a file. (This value may be nonunique if the DUPLICATES phrase is specified for it.) These keys provide alternate access paths for record retrieval from the file.

Example:

```
 1        IDENTIFICATION DIVISION.
 2        PROGRAM-ID.
 3            GEN-ISAM-1.
 4       ****************************************************************
 5       * ISAM MEANS INDEX-SEQUENTIAL ACCESS METHOD.
 6       *
 7       * THE RECORDS ARE OUTPUT TO AN ISAM-FILE USING THE *UNIQUE*
 8       * (I. E., NOT DUPLICATED) DATA FOUND IN FIELD ISAM-KEY AS
 9       * *KEY* VALUE.
10       *
11       * BEFORE THIS JOB CAN BE RUN, THE FOLLOWING *MUST* BE SO:
12       *    A) FILE "ISAM-EX:DATA" AND FILE "ISAM-EX:ISAM" MUST
13       *        EXIST AND BE ERROR-FREE ; OR
14       *    B) FILE "ISAM-EX:DATA" MUST NOT EXIST OR IF EXISTING
15       *        CONTAIN *NO DATA !!*
16       ****************************************************************
17        ENVIRONMENT DIVISION.
18        INPUT-OUTPUT SECTION.
19        FILE-CONTROL.
20            SELECT ISAM-FILE ASSIGN TO "ISAM-EX:DATA",
21                ORGANIZATION IS INDEXED,
22                ACCESS MODE IS DYNAMIC,
23                RECORD KEY IS ISAM-KEY,
24                FILE STATUS IS ISAMSTATUS.
25        DATA DIVISION.
26        FILE SECTION.
27        FD  ISAM-FILE
28            RECORD CONTAINS 46 CHARACTERS.
29        01  ISAM-REC.
30            02  ISAM-KEY PIC X(6).
31       *          :.....................MUST BE IN RECORD AREA!
32            02  ISAM-TEXT PIC X(40).
33
34        WORKING-STORAGE SECTION.
35        01  ISAMSTATUS PIC XX.
36       *           :.....................RETURN STATUS FROM ISAM.
37
38        PROCEDURE DIVISION.
39        A001.
40            OPEN I-O ISAM-FILE.
41        A002.
42            DISPLAY "ENTER KEY (MAX. 6 CHAR) : ",
43                ACCEPT ISAM-KEY.
44            IF ISAM-KEY = SPACES GO TO LIST.
45       *                :.............SPACES INPUT, END DIALOG.
46            DISPLAY "ENTER TEXT (MAX 40 CHAR) : ",
47                ACCEPT ISAM-TEXT.
48       *              :.................READ RECORDS FROM TERMINAL.
49            WRITE ISAM-REC, INVALID KEY,
50                DISPLAY "ISAM FILE ERROR :", ISAMSTATUS, ":".
51            GO TO A002.
52       *        :.....................OUTPUT RECORD AND ASK AGAIN.
53        LIST.
54            DISPLAY "ENTER ACCESS KEY: ",
55                ACCEPT ISAM-KEY.
56            IF ISAM-KEY = SPACES THEN GO TO FINI.
57            READ ISAM-FILE RECORD KEY IS ISAM-KEY INVALID KEY,
58                DISPLAY "RECORD NOT FOUND!",
59                GO TO LIST.
60            DISPLAY "REC: ", ISAM-KEY, ": ",ISAM-REC.
61            GO TO LIST.
62        FINI.
63            CLOSE ISAM-FILE.
64            DISPLAY "JOB END.".
65            STOP RUN.
```

Relative Organization

Relative file organization is permitted only on mass storage devices. The file may be thought of as a string of areas, each capable of holding a logical record. Each of these is identified by a relative record number which is used for storage and retrieval.

For example, the tenth record is the one addressed by relative record number 10 and is in the tenth record area, whether or not records have been written in the first through ninth record areas.

Records may be of fixed or variable length.

Example:

```
 1        IDENTIFICATION DIVISION.
 2        PROGRAM-ID.     GENRELATIVE.
 3       ****************************************************************
 4       * THIS PROGRAM SHOWS THE USAGE OF A RELATIVE FILE.
 5       *
 6       * THE FILE *MUST* EXIST BEFORE THE RUN, BUT MAY BE EMPTY, EACH
 7       * RECORD IS LOCATED DIRECTLY BY ITS RELATIVE (TO 1) POSITION IN
 8       * THE FILE BY ITS *INTEGER* KEY VALUE.
 9       *
10       * NOTE: EVEN IF SOMETHING IS WRITTEN ON RECORDS 300 AND 700
11       * IN THE RELATIVE FILE, *NO* STORAGE SPACE IS USED FOR THE
12       * "EMPTY" RECORDS BETWEEN RECORD 0 AND 299, OR BETWEEN 301 AND
13       * 699. THUS IT MAKES PERFECT SENSE TO USE, SAY, BIRTH DATES
14       * AS KEYS IN RELATIVE FILES.
15       ****************************************************************
16        ENVIRONMENT DIVISION.
17        INPUT-OUTPUT SECTION.
18        FILE-CONTROL.
19            SELECT RELFILE ASSIGN "RELATIVE-EX:DATA",
20                        ORGANIZATION IS RELATIVE,
21                        ACCESS IS DYNAMIC,
22                        RELATIVE KEY IS REL-KEY,
23                        FILE STATUS IS REL-STATUS.
24        DATA DIVISION.
25        FILE SECTION.
26
27        FD RELFILE
28                        BLOCK CONTAINS 10 RECORDS
29                        RECORD CONTAINS 60 CHARACTERS.
30        01  REL-RECORD  PIC X(60).
31
32        WORKING-STORAGE SECTION.
33        01  REL-STATUS  PIC XX.
34        01  REL-KEY     PIC 999.
35       ********:*******************************************************
36       *       :........THE RELATIVE KEY CAN NOT OCCUR IN THE RECORD
37       *                 AREA. ITS POSSIBLE SIZES ARE 1-99999999999,
38       *                 BUT IT IS RESTRICTED TO 999 IN THIS PROGRAM.
39       ****************************************************************
40        PROCEDURE DIVISION.
41        A000.
42            OPEN I-O RELFILE.
43        A002.
44            DISPLAY "ENTER KEY (MAX 999) : ".
45            PERFORM GET-KEY.
46            IF REL-KEY = ZEROES GO TO A003.
47            DISPLAY "ENTER TEXT (MAX 60 CHARACTERS) : ".
48            ACCEPT REL-RECORD.
49            WRITE REL-RECORD INVALID KEY,
50                        DISPLAY "*** RELFILE ERROR *** :", REL-STATUS.
51            GO TO A002.
52        A003.
53            DISPLAY "ENTER ACCESS KEY: ".
54            PERFORM GET-KEY.
55            IF REL-KEY = ZEROS GO TO A999.
56            READ RELFILE RECORD INVALID KEY,
57                        DISPLAY "*** RECORD NOT FOUND ***",
58                        REL-STATUS, GO TO A003.
59            DISPLAY "REC :", REL-KEY, ": ", REL-RECORD.
60            GO TO A003.
61        A999.
62            CLOSE RELFILE.
63            DISPLAY "JOB END".
64            STOP RUN.
65        GET-KEY.
66            ACCEPT REL-KEY.
67            IF REL-KEY NOT NUMERIC,
68                        DISPLAY "*** KEY MUST BE NUMERIC ***",
69                        GO TO GET-KEY.
70        GET-KEY-EXIT.
71            EXIT.
```

4.2.1.2 Access Modes

Three access modes are available in COBOL: *sequential*, *random*, and *dynamic*.

For *sequential organization*, records can only be accessed in sequential access mode, i.e., in the order in which they were originally written on the file. A sequential mass storage file may be used for input and output at the same time. One file maintenance method made possible by this facility is to read a record, process it and - if it is updated - write the modified record back into the previous position.

For *indexed organization*, using the sequential access mode means that records are accessed in the ascending order of the record key values. (The order of retrieval of records within a set of records having duplicate key record values, is the order in which the records were written into the set.)

Using the random access mode, records are accessed in a sequence determined by the programmer. A desired record is accessed by having its record key defined as a record key data item.

Using the dynamic access mode, the programmer may change from sequential access to random access at will by means of appropriate coding.

For *relative organization*, the file access mode can be either sequential, dynamical or random. Sequential access provides the same results as if the file were organized sequentially. Records are accessed in ascending order of relative record number of records currently existing on the file.

Using random mode, the access sequence is controlled by the programmer. The desired record must have its relative record number placed in a relative key data item.

Such a file may be thought of as a serial string of areas, each capable of holding a logical record. Each of these areas is specified by a relative record number. Records are stored and retrieved based on this number. For example, the tenth record is the one addressed by relative record number 10 and is the tenth record area, whether or not records have been written in the first through the ninth record areas.

In the dynamic access mode, the programmer may change at will from sequential access to random access using appropriate forms of input-output statements.

4.2.2 The File-Control Paragraph

The FILE-CONTROL paragraph associates each file with an external, medium and allows specification of file organization, access mode, etc.

General Format of the FILE-CONTROL paragraph:

```
FILE-CONTROL.

    [select-entry] ...
```

The formats of the various selectentries are given below.

Format 1: Select entry for sequential files

```
SELECT [OPTIONAL] file-name

    ASSIGN TO assignment-name-1
    [                                  ]
    [;RESERVE integer [AREA ]          ]
    [                 [AREAS]          ]
    [                                  ]
    [;ORGANIZATION IS SEQUENTIAL]

    [;ACCESS MODE IS SEQUENTIAL]

    [;FILE STATUS IS data-name-1].
```

Format 2: Select entry for indexed files

```
SELECT [OPTIONAL] file-name

ASSIGN TO assignment-name-1
     [                                    ]
     [                        [ AREA  ]   ]
     [ ;RESERVE integer       [ AREAS ]   ]
     [                                    ]

     ;ORGANIZATION IS INDEXED

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

     ;RECORD KEY IS data-name-2

     [;ALTERNATE RECORD KEY IS data-name-3 [WITH DUPLICATES] ]...

     [;FILE STATUS IS data-name-4].
```

Format 3: SELECT entry for relative files

```
SELECT [OPTIONAL] file-name

    ASSIGN TO assignment-name-1
    [                                          ]
    [  ;RESERVE integer [ AREA  ]              ]
    [                   [ AREAS ]              ]

    ;ORGANIZATION IS RELATIVE

    [                                                                    ]
    [                  { SEQUENTIAL [,RELATIVE KEY IS data-name-5]       }
    [;ACCESS MODE IS   {                                                 }
    [                  {  RANDOM   }                                     }
    [                  {           } ,RELATIVE KEY IS data-name-5        }
    [                  {  DYNAMIC  }                                     }
    [                                                                    ]

    [;FILE STATUS IS data-name-6].
```

Format 4: SELECT entry for SORT/MERGE

```
SELECT file-name ASSIGN TO assignment-name-1.
```

The SELECT clause must appear first in the file control entries but subsequent clauses may appear in any order.

Each file described in the Data Division must appear in one and only one entry in the file control paragraph.

The default access mode is sequential.

The file status data-name, (data-names-1, 4 and 6) must be defined in the Data Division as a two character, alphanumeric item which is not, however, defined in the file section.

All data-names may be qualified.

4.2.2.1 For Sequential Organization

The absence of the ORGANIZATION IS SEQUENTIAL clause implies the existence of this clause.

The OPTIONAL phrase may be specified for input or output files. Its specification is required for input or output files that are not necessarily present each time the object program is executed.

4.2.2.2 For Indexed Organization

Data-names 2 and 3 must be defined as alphanumeric in a record description entry for that file name; neither can describe an item whose size is variable.

Data-name-3 cannot reference an item whose leftmost character position corresponds to the leftmost character position of an item referenced by data-name-2 or by any other data-name-3 associated with this file.

The OPTIONAL phrase may be specified for input or output files. Its specification is required for input or output files that are not necessarily present each time the object program is executed.

4.2.2.3 For Relative Organization

Data-name-5, which must be an unsigned integer, must not be described in a record description entry associated with that file.

If a relative file is referenced by a START statement, then the RELATIVE KEY phrase must appear for that file.

The OPTIONAL phrase may be specified for input or output files. Its specification is required for input or output files that are not necessarily present each time the object program is executed.

4.2.2.4 General Rules

1) The ASSIGN clause specifies the association of a file name with a storage medium.

2) The ORGANIZATION clause defines the logical structure of a file. This is established when the file is created and cannot be subsequently changed.

3) The RESERVE clause is treated as comments and appears for syntax reasons only.

4) When the FILE STATUS clause appears, the COBOL library system, after execution of every statement referencing the file, moves a value indicating the status of the execution into the data item referenced by this clause (see I-O Status under INPUT-OUTPUT statements in the Procedure Division description).

Records in the file are accessed in the sequence determined by the predecessor successor relationships established by the execution of WRITE statements in the file formation.

General Rules for Indexed Organization:

1) When the access mode is sequential, records in the file are accessed in the order of ascending record key values within a given key of reference. If the access mode is random then the value of the record key indicates which record is accessed by it. When the access mode is dynamic, the file may be accessed sequentially and/or randomly.

2) The RECORD KEY clause denotes the prime record key for the file, and its values must be unique. The ALTERNATE RECORD KEY clause specifies an alternate record key for the file. Both record keys provide access paths to the records in the file.

General Rules For Relative Organization

1) When the access mode is sequential, records are accessed in the order of ascending relative record numbers of the records existing on the file. If the access mode is random then the value of the RELATIVE KEY data item is used to locate a record. When the access mode is dynamic, records in the file can be accessed sequentially and/or randomly.

2) All records stored in a file are uniquely identified by relative record numbers. These specify the record's logical ordinary position as follows: the first logical record has a relative record number of one (1) and subsequent records have relative record numbers of 2, 3, 4, ...

4.2.3 The I-O CONTROL Paragraph

(Sequential Files Only)

The I-O-CONTROL paragraph specifies the memory area to be shared by different files.

Format:

```
I-O-CONTROL.

    [SAME AREA for file-name-1 {file-name-2} ... ] ...
```

The I-O CONTROL paragraph is optional. More than one SAME clause may be included in a program, however:

1) A file name must not appear in more than one SAME AREA clause.

2) The files referenced in the SAME AREA clause need not all have the same access.

The SAME AREA clause specifies that two or more files not representing SORT files are to use the same memory area during processing. The area being shared includes all storage areas assigned to the specified files so that it is not valid to have more than one of the files open at the same time.

5 THE DATA DIVISION

5.1 DATA CONCEPTS

The Data Division describes the data that the object program is to accept as input, to manipulate, to create or to produce as output. Data to be processed falls into three categories:

1) That which is contained in files and enters or leaves the computer memory from specified areas. This data is external data.

2) That which is developed internally and placed into intermediate storage. This is known as internal data.

3) Constants defined by the user.

External data is contained in files. A file is a collection of records existing on an input or output device. When discussing records, it is important to distinguish between the terms physical record and logical record. A physical record is a collection of data which is treated as an entity by the particular input or output device on which it is stored. A logical record is a collection of data having a logical relationship between its subdivisions. One logical relationship may extend across physical records, several may be contained within one physical record or the two may be identical in size (i.e., one logical relationship is contained completely in one physical unit of data). Unless otherwise described, the term record refers to a logical record, when used in this manual.

The term block is associated with the use of records, usually to describe a unit of data consisting of one or more logical records. The term is synonomous with physical record.

5.2 STRUCTURE OF THE DATA DIVISION

The Data Division is divided into sections, each one having a specific logical function. The occurrence of indicidual sections is optional but they must appear in the order shown when written in the source program.

format:

```
DATA DIVISION.
[ FILE SECTION.
    [ file-description-entry
                {record-description-entry} ...
      sort-file description-entry
                {record-description-entry} ... ] ... ]

[ WORKING-STORAGE SECTION.
    [ 77-level description entry...
      record-description-entry ... ] ... ]

[ LINKAGE SECTION.
    [ 77-level-description-entry...
      record-description-entry ... ] ... ]
```

The File Section contains a description of all externally stored data (FD) but not that which the program may develop internally. It also contains a description of each SORT/MERGE file (SD) in the program.

The Working Storage Section describes records which are developed and processed internally.

The Linkage Section describes data made available from another program (see the section on Interprogram Communication in the "Other Features" part of this manual).

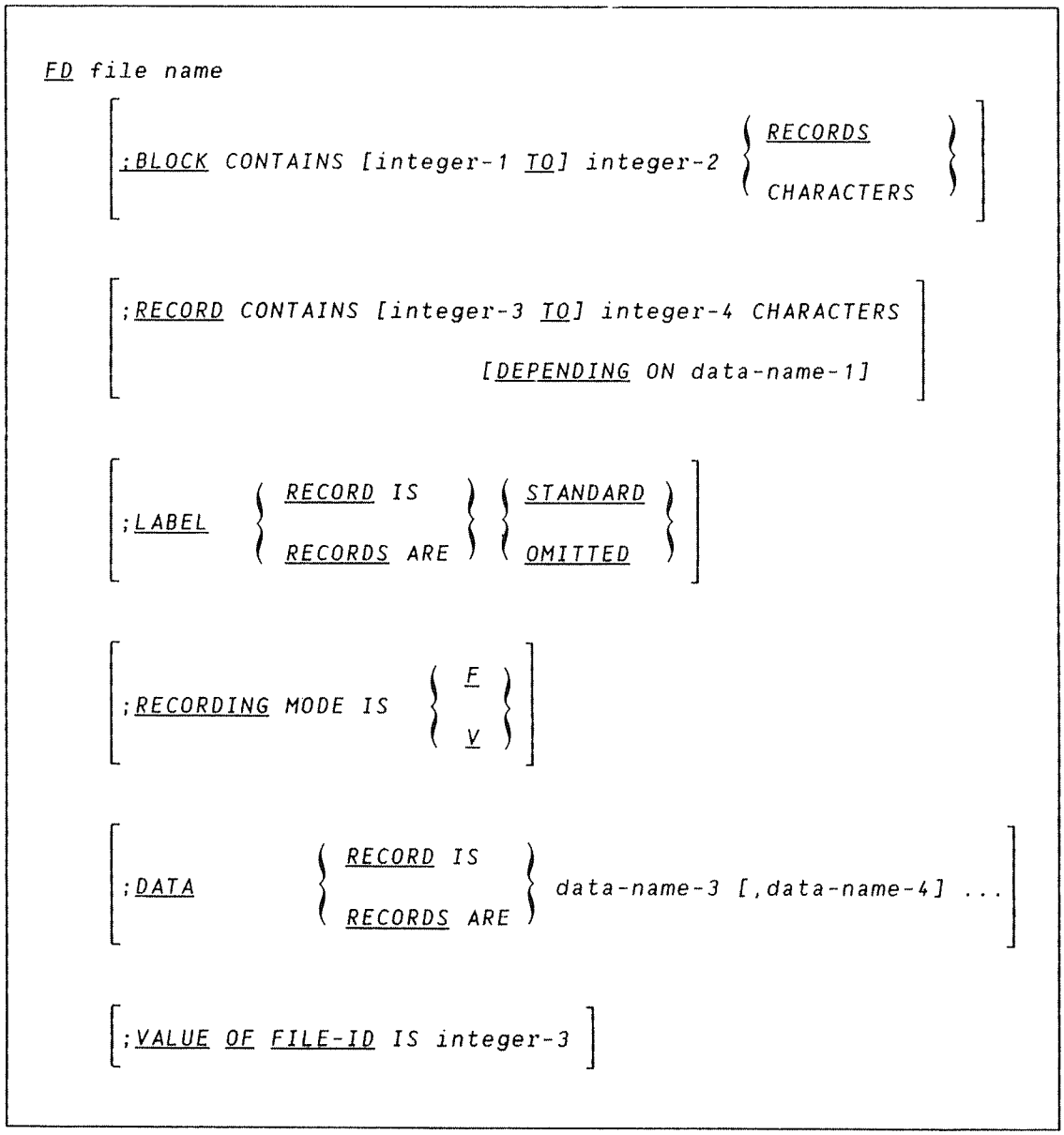
5.3 FILE SECTION

This section must begin with the header FILE SECTION followed by a period. It contains file description entries and sort file description entries, each one followed by its associated record description. All clauses used in the record description entry of the File Section can be used in the Working-Storage Section. The elements allowed in a record description are described later under "Data Description Entry" in the Working Storage Section of the Data Division description (see also "The Concept of Levels" in the same chapter).

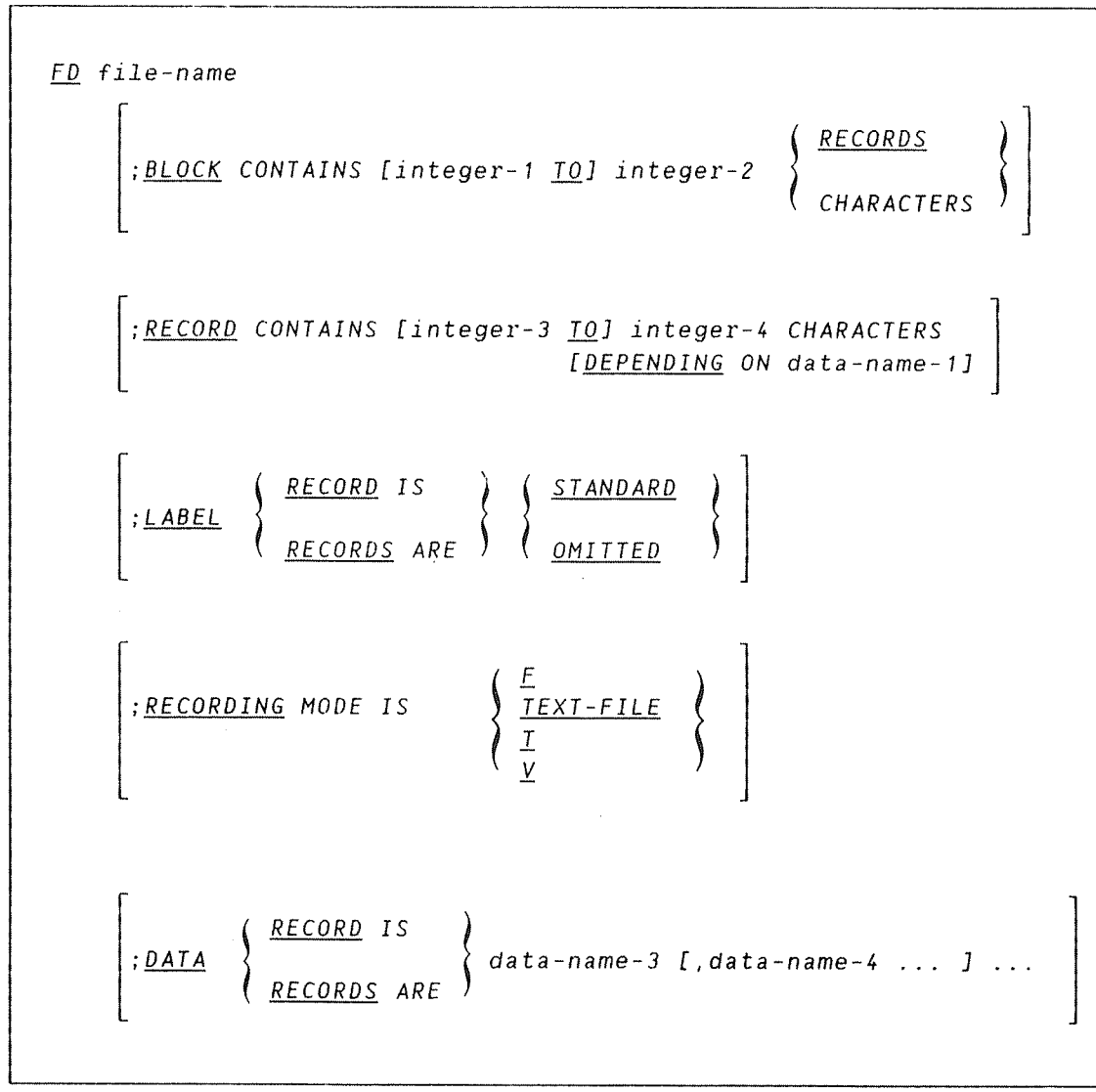
5.3.1 The File Description - Complete Entry Skeleton

The file description entry represents the highest level of organization in the File Section. It follows the File Section header and consists of a level indicator (FD), a file name, and a series of independent clauses specifying the size of the physical and logical records, their structure and their record names on that file. The formats are:

Format 1: Indexed and Relative I-O.



Format 2: Sequential I-O



The level indicator FD identifies the beginning of a file description and must precede the file name. The clauses which follow are optional in many cases and they may appear in any order.

One or more record description entries must follow the file description entry.

5.3.1.1 The BLOCK CONTAINS Clause

The Block Contains clause specifies the size of a physical record.

Format:

```
{ BLOCK CONTAINS [integer-1 TO] integer-2 { RECORDS    } }
                                          { CHARACTERS }
```

General Rules:

1) If this clause is omitted, block size is set to 2048 characters.

2) The size of the physical record may be stated in terms of RECORDS, unless one of the following situations exists, in which case the RECORDS phrase must not be used:

   a) Where logical records may extend across physical records.

   b) The physical record contains padding (area not contained in a logical record).

   c) Logical records are grouped in such a manner that an inaccurate physical record size would be implied.

3) When the word CHARACTERS is specified, the physical record size is specified in terms of the number of character positions required to store the physical record, regardless of the types of characters used to represent the items within the physical record.

4) If only integer-2 is shown, it represents the exact size of the physical record. If integer-1 and integer-2 are both shown, they refer to the minimum and maximum size of the physical record, respectively.

5.3.1.2 The DATA RECORDS Clause

The DATA RECORDS clause serves only as documentation for the names of data records and their associated file.

Format:

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

Data-name-1 and data-name-2 are the names of data records and must have 01 level number record descriptions, with the same names, associated with them.

General Rules:

1) The presence of more than one data name indicates that the file contains more than one type of data record. These records may vary in size, format, etc. The order in which they are listed is not significant.

2) Conceptually, all data records within a file share the same area. This is in no way altered by the presence of more than one type of data record within the file.

5.3.1.3 The LABEL RECORDS Clause

The LABEL RECORDS clause is treated as comments.

Format:

```
[;LABEL { RECORD IS    } { STANDARD }]
        { RECORDS ARE  } { OMITTED  }
```

5.3.1.4 The RECORD CONTAINS Clause

The RECORD CONTAINS clause specifies the size of data records.

Format:

```
[;RECORD CONTAINS [integer-3 TO] integer-4 CHARACTERS
                                 [DEPENDING ON data-name-1]]
```

General Rules:

1) The following notes apply:

   a) Integer-4 may not be used by itself unless all of the data records in the file have the same size. In this case, integer-4 represents the exact number of characters in the data record. If integer-3 and integer-4 are both shown, they refer to the minimum number of characters in the smallest size data record and the maximum number of characters in the largest size data record, respectively.

b) The size is specified in terms of the number of character positions required to store the logical record, regardless of the types of characters used to represent the items within the logical record. The size of a record is determined by the sum of the number of characters in all fixed length elementary items plus the sum of the maximum number of characters in any variable length item subordinate to the record. This sum may be different from the actual size of the record.

2) Data-name-1 must describe an elementary integer in the Working-Storage Section. (Defined as COMPUTATIONAL, with no PICTURE clause specified.)

3) If data-name-1 is specified, the number of character positions in the record must be placed in the data item referenced by data-name-1 before any RELEASE, REWRITE or WRITE staement is executed for the file.

4) If data-name-1 is specified, the execution of a DELETE, RELEASE, REWRITE, START or WRITE statement or the unsuccessful execution of a READ or RETURN statement does not alter the contents of the data item referenced by data-name-1.

5) During the execution of a RELEASE, REWRITE or WRITE statement, the number of character positions in the record is determined as follows:

a) By the contents of the data item referenced by data-name-1 if data-name-1 is specified.

b) By the number of character positions in the record if data-name-1 is not specified.

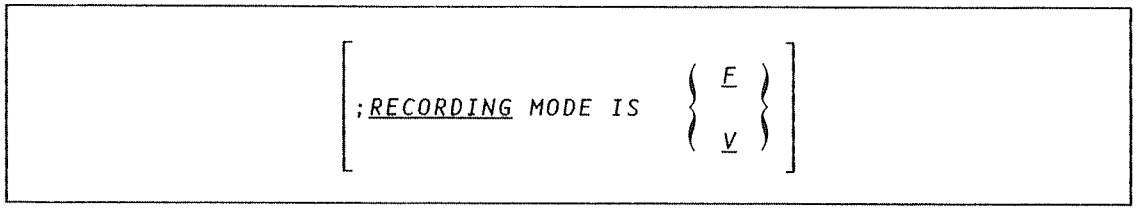
6) If data-name-1 is specified, after the successful execution of a READ or RETURN statement for the file, the contents of the data item referenced by data-name-1 will indicate the number of character positions in the record just read.

7) If the INTO phrase is specified in the READ or RETURN statement, the number of character positions in the current record that participate as the sending data item in the implicit MOVE statement is determined by the maximum size of the sending field.

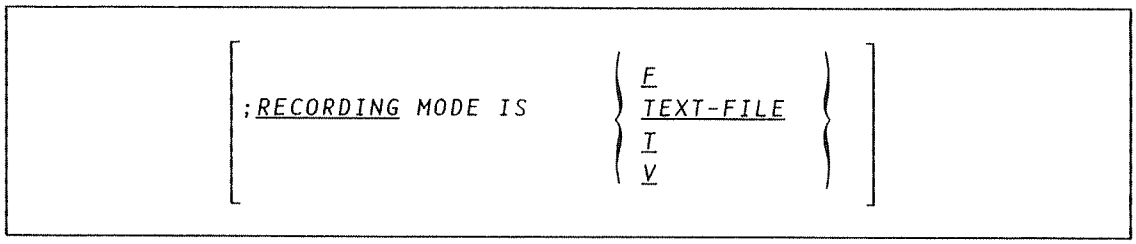
5.3.1.5 The RECORDING MODE Clause

The RECORDING mode clause specifies the record format used in the file.

Format 1: Indexed and Relative I-O.



Format 2: Sequential I-O.



F indicates that all records have exactly the same number of characters, that is, the number which is the length of the file's record area.

V means that the records in the file may have a varying number of characters, never less than 1 (one) and never more than the maximum size of the file's record area. With V format, two extra bytes of information are stored at the beginning of each record in the file. These bytes contain the length of the data portion of the record; they are never available to the COBOL program, except if the DEPENDING ON phrase of the RECORD CONTAINS clause is included.

T (TEXT-FILE) means that the records of the file are in printable format and contain only ASCII characters. The records are separated by the characters carriage return (15 octal) and line feed (12 octal). This format is only valid for sequential files. T and TEXT-FILE are synonymous.

5.3.1.6 The VALUE OF FILE-ID IS Clause

The VALUE OF FILE-ID IS clause is now treated as a comment.

Format:

```
VALUE OF FILE-ID IS integer-3
```

5.4 WORKING-STORAGE SECTION

The Working-Storage Section may describe data records which are not part of external files but are developed and processed internally. It must begin with the words WORKING-STORAGE SECTION followed by a period. It contains record description entries and data description entries for noncontiguous data items.

Data Description Entries

Noncontiguous items in Working-Storage that bear no hierarchical relationship to one another, need not be grouped into records, provided they do not need to be further subdivided. Instead, they are classified and defined as noncontiguous elementary items. Each is defined in a separate data description entry with the special level number 77.

Record Description Entries

Data elements that bear a definite hierarchical relationship to one another must be grouped into records structured by level number.

5.4.1 Data Description

5.4.1.1 The Concept of Level

Because records must often be divided into logical subdivisions, the concept of level is inherent in the structure of a record. Fields which cannot be further subdivided are called elementary items. A record can be made up of elementary items or it can itself be an elementary item. If it is necessary to refer to a set of elementary items, they can be combined as a group item. Note that an elementary item can belong to more than one group.

For example, an employer's payroll file might contain a record for all employees at one location. Each employee's record could be represented as a group item while the subdivisions, or elementary items, might be age, salary, grade, tax code, etc.

Level Numbers

A system of level numbers from 1 to 49 is used to organize elementary and group items into records. Special level numbers 77 and 88 identify items used for special purposes. They do not structure a record, and are used as follows:

77 For independent working storage or linkage section items which are not subdivisions of items or themselves subdivided.

88 For identification of a condition name associated with a particular value of a conditional variable (see the VALUE clause later in the Data Division section).

(Level 77 and 01 entries must have unique data names as they cannot be qualified. Subordinate data names, if qualifiable, need not be unique.)

Record Description Level Numbers

A level number must be assigned to each group or elementary item in a record. The level numbers used to structure records are:

01 This specifies the record itself and is the most inclusive of the numbers. A level 01 entry may be either a group or an elementary item.

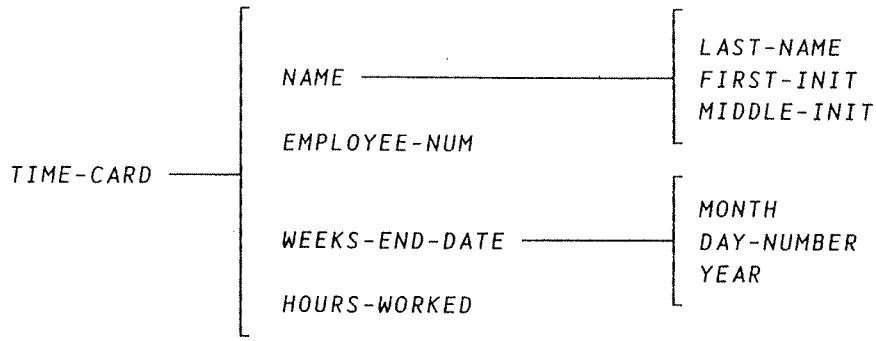
02-49

These are given to group and elementary items within a record. Subordinate items are given higher (not necessarily consecutive) level numbers.

A group item includes all group and elementary items following it until a level number less than or equal to its own is encountered.

All elementary or group items immediately subordinate to one group item must be assigned level numbers higher than the level number of this group item.

For example, data may need to be structured as follows:



A corresponding record might appear in the form:

```
01          TIME-CARD.
  02        NAME.
    03      LAST-NAME PICTURE X(18).
    03      FIRST-INIT PICTURE X.
    03      MIDDLE-INIT PICTURE X.
  02        EMPLOYEE-NUM PICTURE 99999.
  02        WEEKS-END-DATE.
    05      MONTH PIC 99.
    05      DAY-NUMBER PIC 99.
    05      YEAR PIC 99.
  02        HOURS-WORKED PICTURE 99V9.
```

5.4.1.2 Classes and Categories of Data

There are five categories of data items which are grouped into three classes. The relationship between them is shown in the following.

Level of Item:	*Class:*	*Category:*
Elementary	*Alphabetic*	*Alphabetic*
	Numeric	*Numeric*
	Alphanumeric	*Alphabetic* *Numeric Edited* *Alphanumeric Edited* *Alphanumeric*
Group	*Alphanumeric*	*Alphabetic* *Numeric* *Numeric Edited* *Alphanumeric Edited* *Alphanumeric*

Classes and Categories of Data

Note that for alphabetic and numeric the classes and categories are synonomous. The alphanumeric class includes the categories of alphanumeric edited, numeric edited and alphanumeric (without editing). Every elementary item, except for an index data item, belongs to one of the classes and to one of the categories.

Every group item belongs to the alphanumeric class (even if its subordinate items belong to other classes or categories).

Standard alignment rules for positioning data in an elementary item depend on the data category of the receiving item (i.e., the item into which the data is placed).

The following rules apply, according to the category of the receiving item:

1) *Numeric*

The data is aligned by decimal point and moved to the receiving character positions with zero fill or truncation on either end as required.

If there is no assumed decimal point (an assumed decimal point is one that has logical meaning but does not exist as a character in the data), then the item is treated as if an assumed decimal point existed immediately after its rightmost character and is aligned as in the preceding rule.

2) *Numeric Edited*

The data is aligned on the decimal point and (if necessary) truncated or padded with zeros at either end, except when editing causes replacement of leading zeros.

3) *Alphanumeric, Alphanumeric Edited, Alphabetic*

The data is aligned at the leftmost character position and (if necessary) truncated or padded with spaces. If the JUSTIFIED clause is specified then this rule is modified as described in the description of this clause.

Signed Data.

There are two classes of algebraic signs used in COBOL: *operational signs* and *editing signs*.

Operational signs are associated with signed numeric items to indicate their algebraic properties.

Editing signs, which are PICTURE symbols, are used with numeric edited items to indicate the sign of the item in edited output.

Data Reference

Every user specified name of an element in a COBOL program must be unique - either because no other name has a character-string of the same value or because it can be made unique through qualification, indexing or subscripting.

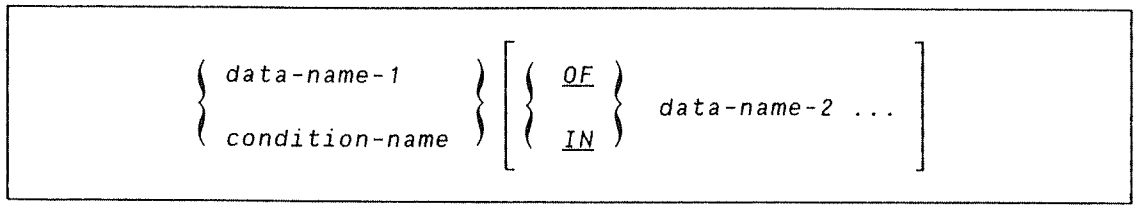
Qualification

A name can be made unique if it exists within a hierarchy of names such that it can be identified by specifying one or more higher level names in this hierarchy. This process is called qualification, and the higher level names are called qualifiers.

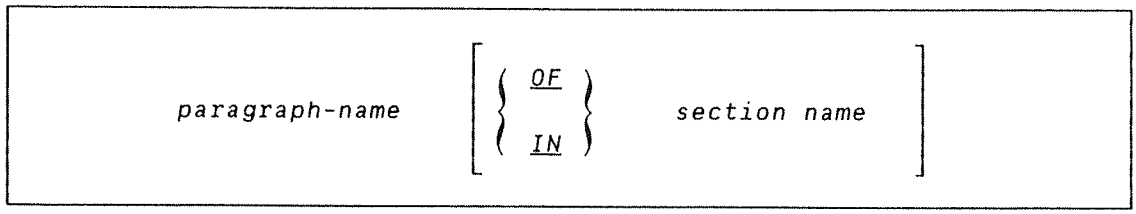
Qualification is performed by following a user specified name by one or more phrases composed of a qualifier preceded by IN or OF. (IN and OF are logically equivalent.)

The Formats are:

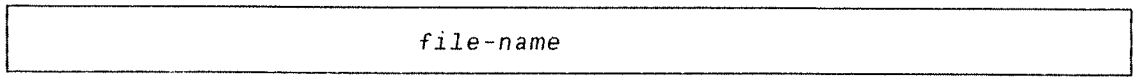
Format 1:



Format 2:



Format 3:



Each qualifier must be of a successively higher level and be within the same hierarchy as the name it qualifies.

The same name must not appear at 2 levels in a hierarchy.

If a data name or condition name is assigned to more than one data item, it must be qualified each time it is referred to.

A paragraph name must not be duplicated within a section. When a paragraph name is qualified by a section name, the word SECTION must not appear. A file name (used in the COPY statement) must name a SINTRAN file. A paragraph name need not be qualified when referred to within the section in which it appears. When it is being used as a qualifier, a data name cannot be subscripted.

If there is more than one combination of qualifiers that ensures uniqueness then any of these combinations can be used.

Note: Although enough qualification must be given to make the name unique, it may not be necessary to specify all the levels of the hierarchy.

No duplicate section names are allowed.

No data name can be the same as a section name or paragraph name.

Duplication of data names must not occur in those places where the data names cannot be made unique by qualifications.

Subscripting and Indexing

Subscripts and indexes are used for referencing an individual element within a table of elements that do not have individual data names. Subscripting and Indexing are explained in the chapter on Table Handling.

5.4.2 The Data Description - Complete Entry Skeleton

The format of the complete entry skeleton has been simplified for easier reading. The format of each clause is given with the individual descriptions.

5.4.2.1 Data Description Entry

A data description entry specifies the characteristics of a particular item of data.

Format 1:

```
level number  { data name }  clause
              { FILLER    }

[;BLANK WHEN ZERO  clause]

[;JUSTIFIED clause]

[;PICTURE clause]

[;REDEFINES clause]

[;SIGN clause]

[  ; { SYNCHRONIZED clause   } ]
     { SYNCHRONIZED-2 clause }

[;USAGE clause]

[;VALUE clause]

[;EXPORT clause]
```

Format 2:

```
88 condition name VALUE clause
```

Format 1 is used for record description entries and for level 77 entries.

General Rules:

1) The *level number* can be any number from 01 to 49 or 77. 01 to 09 can be written as 1 to 9.

2) The data name/FILLER (optional) entry must immediately follow the level number. Otherwise, the clauses may be written in any order.

3) The PICTURE clause must be specified for all elementary items except index data items and for computational, computational-1 and computational-2 items.

4) The BLANK WHEN ZERO, JUSTIFIED, PICTURE and SYNCHRONIZED clauses are valid only for elementary items.

5) Each entry must end with a period followed by a space and all clauses must be separated by a space, comma, or a semicolon followed by a space.

Format 2 describes *condition names* which are user specified names that associate value(s) and/or range of values with a conditional variable. A *conditional variable* is a data item which can take one or more values and is associated with a condition name.

General Rules:

1) Each condition name requires a separate entry with level number 88. Any entry beginning with this level number is a condition name.

2) A condition name can be associated with any data description entry containing a level number except:

   1) another condition name

   2) an index data item.

3) Each entry must end with a period followed by a space. Successive operands must be separated by either a space or a semicolon or comma followed by a space.

5.4.2.2 The BLANK WHEN ZERO Clause

The BLANK WHEN ZERO clause permits the blanking of an item when its value is zero.

Format:

BLANK WHEN ZERO

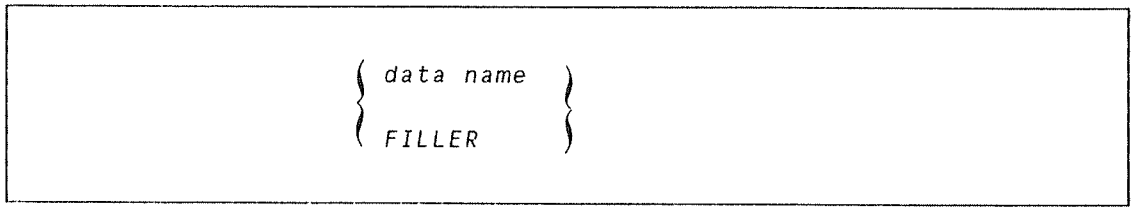
The BLANK WHEN ZERO clause can only be used for an elementary item whose PICTURE is numeric or numeric edited (see the PICTURE clause in this chapter). When it is used for an item whose PICTURE is numeric then the category of the item is considered to be numeric edited.

When the BLANK WHEN ZERO clause is used, the item will contain nothing but spaces if the value of the item is zero.

5.4.2.3 The Data Name/FILLER Clause

A data name explicitly identifies the data being described. The key word FILLER, which may be omitted, specifies an item not explicitly referred to in a program.

Format:



In the File, Working-Storage and Linkage Sections, data name or FILLER must appear as the first word following the level number in each data description entry.

General Rules:

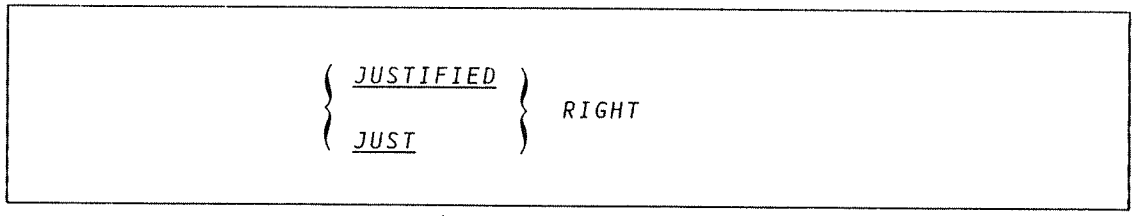
1) A data name identifies a data item used in the program, it may assume a number of different values during program execution.

2) The key word FILLER can name an elementary or group item in a record. Under no circumstances can a FILLER item be referred to explicitly; however, it may be used as a conditional variable since such use does not require explicit reference to the item itself but only to its value.

5.4.2.4 The JUSTIFIED Clause

The JUSTIFIED clause overrides standard positioning rules for a receiving item of the alphabetic or alphanumeric categories.

Format:



The JUSTIFIED clause can be specified only at the elementary item level. JUST is an abbreviation for JUSTIFIED and has the same meaning. It cannot be used with any data item which is numeric or for which editing is specified.

General Rules:

1) When a receiving data item is described with the JUSTIFIED clause and it is smaller than the sending item, the leftmost characters are truncated; if larger, the unused character positions at the left are filled with spaces.

2) When the JUSTIFIED clause is omitted, the standard rules for aligning data within an elementary item apply. (See Standard Alignment Rules in this chapter.)

5.4.2.5 The PICTURE Clause

The PICTURE clause describes the general characteristics and editing requirements of an elementary item.

Format:

```
{ PICTURE }
{ PIC     }  IS character-string
```

The PICTURE clause must be specified for every elementary item except an index data item, or computational, computational-1 and computational-2 items. It may be specified only at the elementary level. PIC is an abbreviated form of PICTURE and has the same meaning.

The character-string is made up of certain COBOL characters used as *symbols*. The allowable combinations determine the category of the elementary item. The maximum number of characters, i.e., symbols, allowed in the string is 30.

List of Symbols

The following list of symbols is used to represent the five categories of data that can be described in a PICTURE clause. (These are: alphabetic, number, alphanumeric, alphanumeric edited and numeric edited.) A brief description is given with each symbol. More detailed descriptions appear later.

A

Each A in the character-string represents a character position that can only contain a letter of the alphabet or a space.

B

Each B in the character-string represents a character position into which the space will be inserted.

S

The letter S is used in a character-string to indicate the presence (but not the representation or, necessarily, the position) of an operational sign; it must be the leftmost character in the PICTURE. It is not counted in determining the size of the elementary item unless an associated SIGN clause specifies the SEPARATE CHARACTER phrase. (An operational sign indicates whether the value of the item is positive or negative.)

V

The V is used in a character position to indicate the location of an assumed decimal point and may appear only once in a character-string. It does not represent a character position and is therefore not counted in the size of the elementary item. When the assumed decimal point is to the right of the rightmost symbol in the string, the V is redundant.

X

Each X in the character-string represents a character position which contains any allowable character from the computer's character set.

Z

Each Z in a character-string may only be used to represent the leftmost leading numeric character positions which will be replaced by a space character when the contents of that character position is zero. Each Z is counted in the size of the item.

9

Each 9 in the character-string represents a character position which contains a numeral and is counted in the size of the item.

0

Each 0 (zero) in the character-string represents a character position into which the numeral zero will be inserted. It is counted in the size of the item.

/

Each / (stroke) in the character-string represents a character position into which the stroke character will be inserted. It is counted in the size of the item.

,

Each , (comma) in the character-string represents a character position into which the character , (comma) will be inserted. This character position is counted in the size of the item and the character must not be the last character in the PICTURE character-string.

When the character . (period) appears in the character-string it is an editing symbol which represents the decimal point for alignment purposes. In addition, it represents a character position into which the . (period) will be inserted. The character is counted in the size of the item. In a program the functions of the period and comma are exchanged if the clause DECIMAL-POINT IS COMMA is stated in the SPECIAL-NAMES paragraph. (In the exchange, the rules for the period apply to the comma and vice versa when they appear in a PICTURE clause.) The insertion character . (period) must not be the last character in the PICTURE character-string.

+, -, CR, DB

These symbols are used as editing sign control symbols and represent the character position into which the editing sign control symbol will be placed. These symbols are mutually exclusive in any one character-string, and each character used in the symbol is counted in determining the size of the data item.

***

Each * (asterisk) in the character-string represents a leading numeric character position into which an asterisk will be placed when the contents of that position is zero. Each * is counted in the size of the item.

CS

The currency symbol in the character-string represents a character position into which a currency symbol is to be placed. This currency symbol is represented either by the currency sign or by the single character specified in the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph. The currency symbol is counted in the size of the item. (The default symbol is $ (dollar).)

Allowable symbols for each data category

The following rules apply:

Alphabetic Items

a) The PICTURE character-string can only contain the symbols A and B.

b) Its contents when represented in standard data format should be any combination of the 26 (twenty-six) letters of the Roman alphabet and the space from the COBOL character set.

Numeric Items

a) The PICTURE character-string may only contain the symbols 9, S and V. The number of digit positions must range from 1 to 18 inclusive.

b) The contents of the item in standard format must be a combination of the 10 Arabic numerals and, if signed, a representation of th operational sign.

Alphanumeric Items

a) The PICTURE character-string is restricted to certain combinations of the symbols A, X and 9. The item is treated as if the character-string contained all X's. A character-string containing all A's or all 9's does not define an alphanumeric item.

b) The contents of the character-string when represented in standard data format are allowable characters in the computer's character set.

Alphanumeric Edited Items

a) The PICTURE character-string can contain: A, X, 9, B, 0 (zero) and /. It must contain at least one of these combinations:

- at least one B and at least one X
- at least one 0 and at least one X
- at least one X and at least one /
- at least one A and at least one 0

- at least one A and at least one /

b) The contents of the items in standard data format may be any allowable character from the computer's character set.

Numeric Edited Items

a) The PICTURE character-string can contain the symbols: B, V, Z, 9, 0 (zero), *, /, , (comma), . (period), +, -, CR (credit), DB (debit) or the $ (currency) symbol. The allowable combinations are determined from the order of precedence of symbols (see chart) and the editing rules (see later in this section).

b) The character-string must contain at least one 0 (zero), B, /, Z, *, +, -, , (comma), . (period), CR (credit), DB (debit) or currency symbol and the number of digit positions that can be represented must range from 1 to 18 inclusive.

c) The contents of the character positions that are allowed to represent a digit in standard format, must be one of the numerals.

The Size of an Elementary Item

The size of an elementary data item (i.e., the number of character positions it occupies in standard data format) is determined by the number of allowable symbols that represent character positions. An integer enclosed in parentheses following the symbols A, , (comma), X,), Z, *, B, /, 0 (zero), +, - or the currency symbol indicates the number of consecutive occurrences of the symbol.

5.4.2.6 Editing Rules for the PICTURE Clause

Editing is performed in two ways, either by *insertion* or *suppression* and *replacement*. Insertion editing breaks down into four types. These are listed below together with the characters and categories each is valid for.

Simple Insertion:

Category:	*Insertion Symbols:*
Alphabetic	*B*
Alphanumeric edited	*B 0 /*
Numeric Edited	*B 0 / ,*

Examples:

Picture:	*Data:*	*Edited Result:*
99,999,000	*12345*	*12,345,000*
999,999	*12345*	*012,345*
A(5)BA(4)	*NORSKDATA*	*NORSK DATA*
X(4)B/BX(2)	*TYPE25*	*TYPE / 25*

Each insertion symbol is included in the size of the item and represents the position where the equivalent character will be inserted.

Special Insertion:

Category:	Insertion Symbol:
Numeric Edited	. (period)

Examples:

Picture:	*Data:*	*Edited Result:*
99.99	*123.4*	*23.40*
99.99	*12.34*	*12.34*
99.99	*1.234*	*01.23*

The insertion symbol . (period) will be counted in the size of the item, and shows the position where the actual decimal point will be inserted. It is not allowed to appear in the same PICTURE character-string as the symbol V (denoting an assumed decimal point); these two symbols are mutually exclusive.

Fixed Insertion:

Category:	Insertion Symbols:
Numeric edited	+ - CR DB (editing sign control symbols) $ (currency symbol)

Only one currency symbol and only one of the editing sign control symbols can be used, in a given PICTURE character-string. When the symbols 'CR' or 'DB' are used they represent two characters positions in determining the size of the item and they must represent the rightmost character positions that are counted in the size of the item. The symbol '+' or '-', when used, must be either the leftmost or the rightmost character position to be counted in the size of the item. The currency symbol must be the leftmost character position to be counted in the size of the item, except that it can be preceded by either a '+' or a '-' symbol. Fixed insertion editing results in the insertion character occupying the same character position in the edited item as it occupied in the PICTURE character-string. Editing sign control symbols produces the following results depending upon the value of the data item.

Editing Symbol in PICTURE character-string	Result:	
	Data Item Positive or Zero	Data Item Negative
+	+	-
-	space	-
CR	2 spaces	CR
DB	2 spaces	DB

Examples:

Picture:	*Data:*	*Edited Result:*
+99.99	*-12.345*	*-12.34*
-99.99	*+12.345*	*12.34*
99.99+	*+12.345*	*12.34+*
$99.99	*-12.34*	*$12.34*
-$99.99	*-12.34*	*-$12.34*
$999.99 CR	*+12.34*	*$012.34*
$999.99 DB	*-12.34*	*$012.34 DB*

Floating Insertion:

Category:	Insertion Symbols:
Numeric edited	$ + -

Floating insertion editing occurs when two or more of the above insertion symbols appear as a string within the given PICTURE character-string.

Examples:

Picture:	*Data:*	*Edited Result:*
$$99	*12*	*$12*
$$$$$99	*1234*	*$1234*
$$$$$9.99	*.12*	*$0.12*
+++/+++,+99	*12*	*+12*
-----9,999	*123456*	*-123,456*
$$$$$99.99CR	*-123*	*$123.00CR*

Within one PICTURE character-string the floating insertion symbols are mutually exclusive. Simple insertion symbols or the period may appear within a string of floating insertion symbols without causing discontinuity (except in the special case where there is only one floating insertion symbol in the string to the left of a simple one or a period).

The leftmost character of the floating insertion string represents the leftmost limit of the floating symbol in the data item. The rightmost character of the floating string represents the rightmost limit of the floating symbols in the data item.

The second floating character from the left represents the leftmost limit of the numeric data that can be stored in the data item. Nonzero numeric data may replace all the characters at this limit or to the right of it.

In a PICTURE character-string, there are only two ways of representing floating insertion editing. One way is to represent any or all of the leading numeric character positions on the left of the decimal point by the insertion character. The other way is to represent all of the numeric character positions in the PICTURE character-string by the insertion character.

If the insertion characters are only to the left of the decimal point in the PICTURE character-string, the result is that a single floating insertion character will be placed into the character position immediately preceding either the decimal point or the first nonzero digit in the data represented by the insertion symbol string, whichever is further to the left in the PICTURE character-string. The character positions preceding the insertion character are replaced

with spaces.

If all numeric character positions in the PICTURE character-string are represented by the insertion character, the result depends upon the value of the data. If the value is zero the entire data item will contain spaces. If the value is not zero, the result is the same as when the insertion character is only to the left of the decimal point.

To avoid truncation, the minimum size of the PICTURE character-string for the receiving data item must be the number of characters in the sending data item, plus the number of nonfloating insertion characters being edited into the receiving data item, plus one for the floating insertion character.

Zero Suppression and Replacement Editing

The symbols Z and * are used to replace leading zeros in the edited result by blanks or asterisks respectively. They can form floating strings in the same way as the floating insertion symbols $, + and - described earlier. (However, a floating string of zero suppression or replacement symbols cannot appear in the same PICTURE character-string as a floating string of insertion symbols.)

Examples:

Picture:	*Data:*	*Result:*
ZZ.ZZ	*00.09*	*.09*
ZZ.ZZ	*00.00*	
.	*00.00*	***.***
ZZ99.99	*0000.00*	*00.00*
******.99CR*	*123*	***123.00*
,*.**+*	*-123.00*	***123.00-*
ZZZ.ZZ+	*0*	

Any simple insertion symbols or the period may appear within a floating string of zero suppression or replacement characters and are regarded as part of this string.

When editing is performed, any leading zero in the data that appears in the same character position as a suppression symbol, is replaced by the replacement character. Suppression stops at the leftmost character that:

1) Does not correspond to a suppression symbol.

2) Is the decimal point.

3) Contains nonzero data.

If, however, the value of the data is zero and all the numeric character positions in the PICTURE character-string are represented by a Z, the resulting item will contain spaces only. If these positions are represented by asterisks, the resulting item, except for the

decimal point, will contain asterisks.

Precedence Rules

Figure 5.1 shows the order of precedence when using characters as symbols in a character-string. An 'X' at an intersection indicates that the symbol(s) at the top of the column may precede the symbol(s) at the left of the row. Arguments appearing in {} (braces) indicate that the symbols are mutually exclusive. The currency symbol is shown as CS.

At least one of the symbols A, X, Z, 9 or *, or at least two of the symbols +, - or CS must appear in a PICTURE string.

First Symbol / Second Symbol		Non-Floating Insertion Symbols									Floating Insertion & Suppressing/ Replacement Symbols						Other Symbols			
		B	0	/	,	.	{+ −}	{+ −}	{CR DB}	CS	{Z *}	{Z *}	{+ −}	{+ −}	CS	CS	9	A X	S	V
Non-Floating Insertion Symbols	B	X	X	X	X	X	X			X	X	X	X	X	X	X	X	X		X
	0	X	X	X	X	X	X			X	X	X	X	X	X	X	X	X		X
	/	X	X	X	X	X	X			X	X	X	X	X	X	X	X	X		X
	,	X	X	X	X	X	X			X	X	X	X	X	X	X	X			X
	.	X	X	X	X		X			X	X		X		X		X			
	{+ −}																			
	{+ −}	X	X	X	X	X				X	X	X			X	X	X			X
	{CR DB}	X	X	X	X	X				X	X	X			X	X	X			X
	CS						X													
Floating Insertion and Suppressing/Replacement Symbols	{Z *}	X	X	X	X		X			X	X									
	{Z *}	X	X	X	X	X	X			X	X	X								X
	{+ −}	X	X	X	X					X			X							
	{+ −}	X	X	X	X	X				X			X	X						X
	CS	X	X	X	X		X								X					
	CS	X	X	X	X	X	X								X	X				X
Other Symbols	9	X	X	X	X	X	X			X	X		X		X		X	X	X	X
	A X	X	X	X													X	X		
	S																			
	V	X	X	X	X		X			X	X		X		X		X		X	

Figure 5.1.

5.4.2.7 The REDEFINES Clause in DATA DIVISION

The REDEFINES clause allows the same computer storage area to be described by different data description entries.

Format:

level number data-name-1; REDEFINES data-name-2

(*Note:* The level number, semicolon and data-name-1 are shown in the above format for reasons of clarity. Level number and data-name-1 are not part of the REDEFINES clause.)

Data-name-2 is the redefined item while data-name-1 supplies an alternative description for the same area, i.e., is the redefining item.

The level numbers of data-name-1 and data-name-2 must be identical but not level 88.

General Rules:

1) Redefinition begins at data-name-1 and ends when a level number less than or equal to that of data-name-2 is encountered. No entry having a level number lower than those of data-names 1 and 2 may occur between these entries.

2) When the level number of data-name-1 is other than 01, it must specify the same number of character positions that the data item referenced by data-name-2 contains. It is important to observe that the REDEFINES clause specifies the redefinition of a storage area, not of the data items occupying the area.

3) Multiple redefinitions of the same character positions are permitted. The entries giving the new descriptions of the character positions must follow the entries defining the area being redefined; no entries that define new character positions may intervene.

4) Multiple level 01 entries subordinate to any given level indicator represent redefinitions of the same area.

5) The entries giving the new description of the character positions must not contain any VALUE clauses, except in condition name entries.

Example:

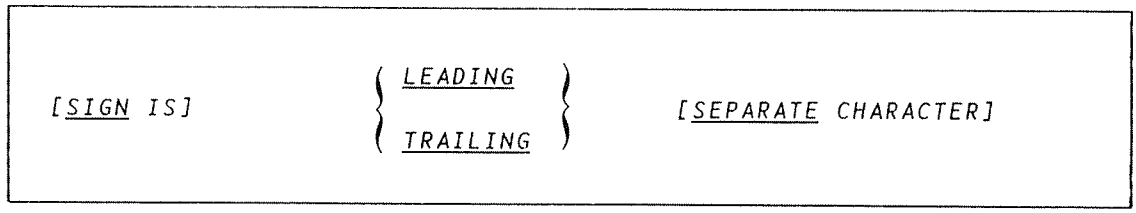
```
02  A PICTURE A(6).
02  B REDEFINES A.
    05  B-1 PICTURE X(2).
    05  B-2 PICTURE 9(4).
02  C PICTURE 9(6).
02  D REDEFINES C.
    05  D-1 PICTURE 99.
    05  D-2 PICTURE 9999.
    05  D-3 REDEFINES D-2 PICTURE 99V99.
```

In this example A, C and D-2 are redefined items while B, D and D-3 are redefining items. Note that the REDEFINES clause has been specified for the item D-3 which is subordinate to the redefining item, D.

5.4.2.8 The SIGN Clause

The SIGN clause specifies the position and mode of representation of the operational sign when it is necessary to describe these explicitly.

Format:



The SIGN clause may be specified only for a numeric data description entry whose PICTURE contains the character 'S', or a group item containing at least one such numeric data description entry.

The numeric data description entries to which the SIGN IS clause applies must be described as USAGE IS DISPLAY.

At most one SIGN IS clause may apply to any given numeric data description entry.

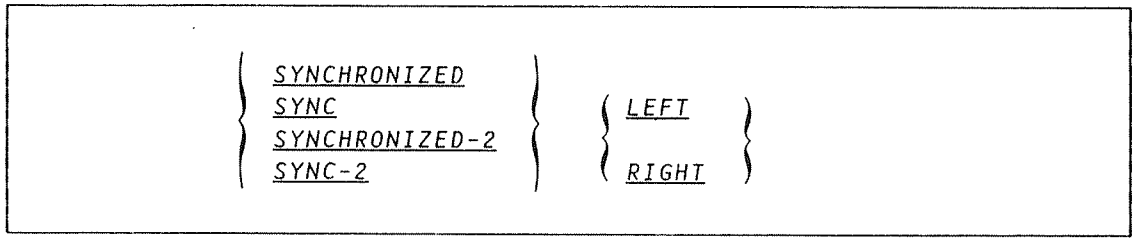
If the SEPARATE CHARACTER option is not present, then the operational sign is assumed to be associated with the LEADING OR TRAILING digit position (whichever is specified). The PICTURE character S is not counted in the size of the item.

If the SEPARATE CHARACTER option is present, then the operational sign is assumed to occupy the LEADING or TRAILING character position. In this case the PICTURE character S is included in the size of the item. The operational signs for positive and negative are the characters + and - (minus) one of which must be present in the data at object time.

5.4.2.9 The SYNCHRONIZED Clause

The SYNCHRONIZED clause specifies the alignment of an elementary item on the natural boundaries of the computer memory.

Format:



This clause may only appear with an elementary item.

SYNC is an abbreviation of SYNCHRONIZED.

General Rules:

1) This clause specifies that the subject data item is to be aligned in the computer such that no other data item occupies any of the character positions between the leftmost (SYNC LEFT) or rightmost (SYNC RIGHT) 16-bits word boundaries delimiting this data item. This applies on both the ND-100 and ND-500 computers. If the number of character positions required to store this data item is less than the number of character positions between the word boundaries, the unused character positions (or some of them) must not be used for any other data item. Such unused character

positions, however, are included in:

a) the size of any group item(s) to which the elementary item belongs and

b) the character positions redefined when this data item is the object of a REDEFINES clause.

2) SYNCHRONIZED LEFT specifies that the elementary item is to be positioned such that it will begin at the left character position of the word boundary in which the elementary item is placed.

3) SYNCHRONIZED RIGHT specifies that the elementary item is to be positioned such that it will terminate at the right character position of the word boundary in which the elementary item is placed

4) Whenever a SYNCHRONIZED item is referenced in the source program, the original size of the item, as shown in the PICTURE clause, is used in determining any action that depends on size, such as justification, truncation or overflow.

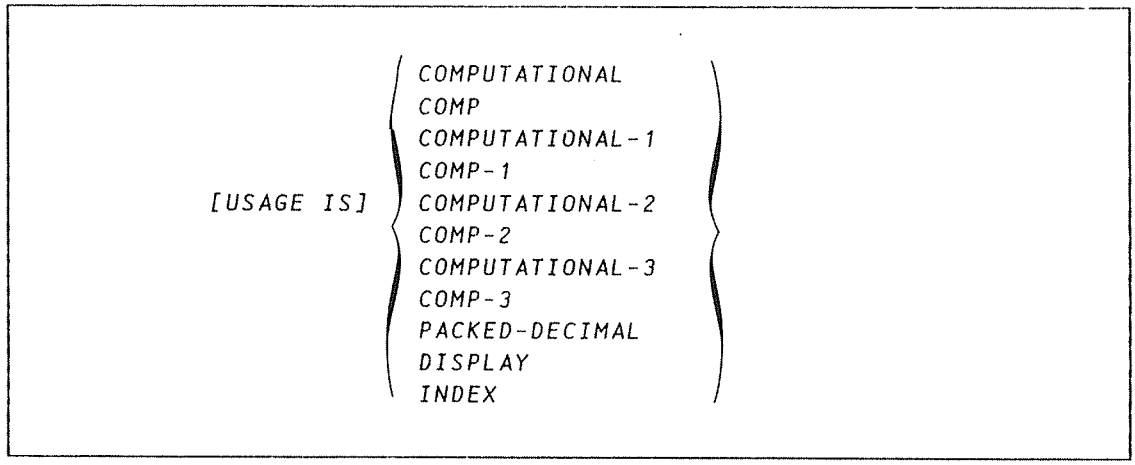
5) If the data description of an item contains the SYNCHRONIZED clause and an operational sign, the sign of the item appears in the normal operational sign position, regardless of whether the item is SYNCHRONIZED LEFT or SYNCHRONIZED RIGHT.

6) When the SYNCHRONIZED clause is specified in a data description entry of a data item that also contains an OCCURS clause, or in one which is subordinate to a data description entry that contains an OCCURS clause, each occurrence of the item is synchronized.

5.4.2.10 The USAGE Clause

The USAGE clause specifies the format of a data item in the computer storage.

Format:



If a COMPUTATIONAL item has a PICTURE character-string, then it can contain only '9's, the operational sign character 'S', or the implied decimal point character 'V'. (Refer to the PICTURE clause section earlier in this chapter.)

COMP is an abbreviation for COMPUTATIONAL.

General Rules:

1) The USAGE clause can be written at any level. If the USAGE clause is written at a group level, it applies to each elementary item in the group. The USAGE clause of an elementary item cannot contradict the USAGE clause of a group to which the item belongs.

2) This clause specifies the manner in which a data item is represented in the storage of a computer. It does not affect the use of the data item, although the specifications for some statements in the procedure division may restrict the USAGE clause of the operands referred to. The USAGE clause may affect the radix or type of character representation of the item.

3) The USAGE IS DISPLAY clause indicates that the format of the data is a standard data format.

4) If the USAGE clause is not specified for an elementary item, or for any group to which the item belongs, the usage is implicitly DISPLAY.

5) All COMPUTATIONAL items are capable of representing a value to be used in computations and must be numeric. If a group item is described as COMPUTATIONAL, the elementary items in the group are COMPUTATIONAL. The group item itself is not COMPUTATIONAL (cannot be used in computations).

6) On the ND-100, COMPUTATIONAL, COMPUTATIONAL-1 and COMPUTATIONAL-2 items are aligned on a word boundary even if the SYNCHRONIZED clause has not been specified.

7) COMPUTATIONAL-3 and PACKED-DECIMAL items are stored in packed decimal format.

The terms COMPUTATIONAL, COMPUTATIONAL-1, COMPUTATIONAL-2, COMPUTATIONAL-3 and PACKED-DECIMAL are explained under "Computational Options", which follows.

5.4.2.11 Computational Options

The terms COMPUTATIONAL and COMPUTATIONAL-1 define integer variables. They can be specified as 16 bit (2 byte) words or 32 bit (4 byte) words. The size depends on the maximum number of digits in the item.

The sizes of COMPUTATIONAL (COMPUTATIONAL-1) items are shown below:

	ND-100	*ND-500*
PICTURE definition is omitted (default integer)	*16 Bits (2 Bytes)*	*32 Bits (4 Bytes)*
PICTURE S9 (n) where n<=4	*16 Bits (2 Bytes)*	*16 Bits (2 Bytes)*
PICTURE S9 (n) where n>=5	*32 Bits (4 Bytes)*	*32 Bits (4 Bytes)*

Integer variables are always treated as if signed, even when there is no sign character(s) in the PICTURE definition.

The range of permissible values is shown below:

Length:	*Range:*
16 bits (2 bytes)	*-32768 through 32767*
32 bits (4 bytes)	*-2147483648 through 2147483647*

COMPUTATIONAL AND COMPUTATIONAL-1 VALUES

Note: For fast performance, integer fields should be used as indexes, as operands in MOVE operations, and for the arithmetic statements of COBOL.

The term COMPUTATIONAL-2 is used for the description of real numbers. The internal representation will be in floating point format.

On the ND-100, the COBOL system is self-adjusting for 48 and 32 bits REAL.

On the ND-500, the size of the real item depends on the numeric length of the PICTURE definition as shown below:

PICTURE definition	*32 Bits*
PICTURE S9(n)V9(m) *where n+m <= 6*	*32 Bits*
PICTURE S9(n)V9(m) *where n+m >= 7*	*64 Bits*

COMPUTATIONAL-2 variables may only be used as parameters in a subroutine call, or for converting (MOVE) to or from COMPUTATIONAL-3 variables.

No VALUE clause can be specified for COMPUTATIONAL-2 items.

COMPUTATIONAL-3 items are identical to PACKED-DECIMAL items. They appear in storage in packed decimal format. This is sometimes known as BCD (Binary Coded Decimal). The digits are each represented by 4 bits so that there are two adjacent digits per byte. The sign is contained in the rightmost 4 bits of the rightmost byte. The numbers always fill an integral number of bytes and are right justified. If necessary, the leftmost half byte is filled with zero.

Each decimal digit is encoded as follows:

Digit/Sign:	*Binary Representation:*	*Hexadecimal Representation:*
0 +	*0 0 0 0*	*0*
1	*0 0 0 1*	*1*
2	*0 0 1 0*	*2*
3	*0 0 1 1*	*3*
4	*0 1 0 0*	*4*
5	*0 1 0 1*	*5*
6	*0 1 1 0*	*6*
7	*0 1 1 1*	*7*
8	*1 0 0 0*	*8*
9	*1 0 0 1*	*9*
+	*1 0 1 0*	*A*
-	*1 0 1 1*	*B*
+	*1 1 0 0*	*C*
-	*1 1 0 1*	*D*
+	*1 1 1 0*	*E*
-	*1 1 1 1*	*F (ND-10 only)*
unsigned	*1 1 1 1*	*F (ND-100/ND-500 only)*

5.4.2.12 The VALUE Clause

The VALUE clause specifies the initial contents of a data item or the value associated with a condition name.

Format 1:

```
VALUE IS literal
```

Format 2:

```
{ VALUE IS  }
{ VALUES ARE } literal-1 [ { THROUGH } literal-2 ]
                         { THRU    }
[ , literal-3 { THROUGH } literal-4 ] ...
              { THRU    }
```

The words THRU and THROUGH are equivalent.

The VALUE clause is used in condition name entries in the File, Linkage and Working-Storage Sections. However, in the Working-Storage Section only, it also serves to specify the initial value of any data item. The item takes this value at the beginning of the program; without the specification, the value is unpredictable.

General Rules:

1) All numeric literals in the VALUE clause of an item must have a value within the range of values indicated by the PICTURE clause, and must not have a value which would require truncation of nonzero digits. Nonnumeric literals must not exceed the size indicated by the PICTURE clause. (A signed literal must have a signed numeric PICTURE character-string assigned with it.)

2) The VALUE clause must not conflict with other clauses in the data description of the item or in the data description within the hierarchy of the item. The following rules apply:

   a) If the category of the item is numeric, all literals in the VALUE clause must be numeric. If the literal defines the value of a Working-Storage item, the literal is aligned in the data item according to the standard alignment rules. (See Standard Alignment Rules.)

   b) If the category of the item is alphabetic, alphanumeric, alphanumeric edited or numeric edited, all literals in the VALUE clause must be nonnumeric literals. The literal is aligned in the data item as if the data item had been described as alphanumeric. (See Standard Alignment Rules.) Editing characters in the PICTURE clause are included in determining the size of the data item (see the PICTURE clause), but have no effect on the initialization of the data item.Therefore, the VALUE for an edited item is presented in an edited form.

   c) Initialization takes place independent of any BLANK WHEN ZERO or JUSTIFIED clause that may be specified.

3) A figurative constant may be substituted in both Format 1 and Format 2 whenever a literal is specified.

4) The VALUE clause must not be written for a group containing items with descriptions including JUSTIFIED, SYNCHRONIZED or USAGE (other than USAGE IS DISPLAY).

5) In a condition name entry, the VALUE clause is required. The VALUE clause and the condition name itself are the only two clauses permitted in the entry. The characteristics of a condition name are implicitly those of its conditional variable.

6) Format 2 can be used only in connection with condition names, and each condition name must have a separate level-88 entry. The special considerations for the use of Format 1 are:

   a) The VALUE clause must not be specified for an entry that contains, or is subordinate to an entry that contains, a REDEFINES or OCCURS clause.

   b) If the VALUE clause is used in an entry at the group level, the literal must be a figurative constant or a nonnumeric literal, and the group area is initialized without consideration for the individual elementary or group items contained within this group. The VALUE clause cannot be stated at the subordinate levels within this group.

See under the "Condition-Name Condition" for an example of the use of the VALUE clause.

5.4.2.13 The EXPORT Clause

The EXPORT clause enables separately-compiled programs to access data items.

Format:

EXPORT identifier

A communication can be established between COBOL and data areas in PLANC or common areas in FORTRAN, without having to use parameters to effect the transfer.

Exported data must be defined in the Working-Storage Section.

For the corresponding IMPORT clause in the Linkage Section see Section 9.1.3.

Note:

- EXPORT/IMPORT are only allowed on 01/77 levels.
- No redefines are allowed on an identifier containing EXPORT/IMPORT, but EXPORT/IMPORT identifiers can be redefined in the usual way.
- The EXPORT and IMPORT clauses are an ND extension.

6 THE PROCEDURE DIVISION

A Procedure Division is needed in every COBOL program. It is composed of two parts:

1) *optional* Declaratives sections, to handle errors and exceptions during execution;

2) procedures which contain the sections, paragraphs, sentences and statements used to solve a data processing problem.

Execution begins with the first statement in the Procedure Division after the Declaratives. Statements are executed in the order in which they are presented for execution (unless the rules imply a different order).

6.1 STRUCTURE OF THE PROCEDURE DIVISION

Format 1:

```
PROCEDURE DIVISION [USING data-name-1 [, data-name-2] ... ].

[DECLARATIVES.
{section-name SECTION [segment-number]. [USE sentence.]

[paragraph-name. [sentence] ... ] ... } ...

END DECLARATIVES. ]

{section-name SECTION [segment-number].

[paragraph-name. [sentence] ... ] ... } ...
```

Format 2:

```
PROCEDURE DIVISION [USING data-name-1 [, data-name-2] ... ].

{paragraph name. [sentence] ... } ...
```

Note: The segment-number will be treated by this compiler as comments only.

6.1.1 Declaratives

Declarative sections are preceded by the key word DECLARATIVES and followed by the key words END DECLARATIVES. They are provided for the processing of exceptional input-output conditions which cannot normally be tested by the programmer. These additional procedures are executed only at the time an I-O error occurs and cannot appear in the regular sequence of procedural statements. Therefore, they are written at the beginning of the Procedure Division in a series of Declarative sections. Each of these sections is preceded by a USE sentence which specifies the actions to be taken when the exceptional condition occurs. (See the USE statement in the I-O Statement section of the Procedure Division description.)

The key word DECLARATIVES is written on the line following the Procedure Division header.

DECLARATIVES and END DECLARATIVES, when they appear, must be followed by a period but without any text on the same lines. They must both be written in area A.

If declarative sections are specified, the Procedure Division must be divided into sections.

6.1.2 Procedures

Procedures, whose names are user-defined, occur in the Procedure Division and may consist of one or more paragraphs and/or one or more sections.

A *section* consists of a section header followed by any number of paragraphs (including none).

A *section header* is a section name followed by the key word SECTION then a period and a space. A section name, which is used to identify a section, is user defined and must be unique.

A *paragraph* consists of a paragraph name, followed by a period followed by a space and then any number of sentences (including none). A paragraph name, which identifies a paragraph, is user defined. It need not be unique since it can be qualified. If one paragraph in the program is contained within a section, then all paragraphs must be contained in sections.

A *sentence* is made up of one or more statements, followed by a period, followed by a space. There are three categories of sentence:

1) A conditional sentence is a conditional statement, optionally preceded by an imperative statement, followed by a period and a space.

2) An imperative sentence is an imperative statement or series of imperative statements finally followed by a period and a space.

3) A compiler directing sentence is a single compiler directing statement followed by a period and a space.

A *statement* is a syntactically valid combination of words and symbols beginning with a COBOL verb. Statements, like sentences, are divided into three types:

1) A conditional statement specifies the action to be taken by the object program, depending on the truth value of a condition.

2) An imperative statement directs that an unconditional action be taken by the object program. It may consist of a series of imperative statements.

3) A compiler directing statement causes a specific action to be taken by the compiler during compilation.

An *identifier* makes unique references to a data item. It may be qualified, indexed or subscripted.

6.2 ARITHMETIC EXPRESSIONS

6.2.1 Definition of an Arithmetic Expression

An *arithmetic expression* can be an identifier of a numeric elementary item, a numeric literal, such identifiers and literals separated by arithmetic operators, two arithmetic expressions separated by an arithmetic operator, or an arithmetic expression enclosed in parentheses. Any arithmetic expression may be preceded by a unary operator. The permissible combinations of variables, numeric literals, arithmetic operator and parentheses are given in the table below.

The identifiers and literals appearing in an arithmetic expression must represent either numeric elementary items or numeric literals on which arithmetic may be performed.

6.2.1.1 Arithmetic Operators

There are five binary arithmetic operators and two unary arithmetic operators that may be used in arithmetic expressions. They are represented by specific characters that must be preceded by a space and followed by a space.

Binary Arithmetic Operators:	*Meaning:*
+	*Addition*
-	*Subtraction*
*	*Multiplication*
/	*Division*
**	*Exponentiation*

Unary Arithmetic Operators:	*Meaning:*
+	*The effect of multiplication by numeric literal +1*
-	*The effect of multiplication by numeric literal -1.*

6.2.1.2 Evaluation Rules

1) Parentheses may be used in arithmetic expressions to specify the order in which elements are to be evaluated. Expressions within parentheses are evaluated first; within nested parentheses, evaluation proceeds from the least inclusive set to the most inclusive set. When parentheses are not used, or parenthesized expressions are at the same level of inclusiveness, the following hierarchical order of execution is implied:

1st - unary plus and minus

2nd - exponentiation

3rd - multiplication and division

4th - addition and subtraction

2) Parentheses are used either to eliminate ambiguities in logic where consecutive operations of the same hierarchical level appear, or to modify the normal hierarchical sequence of execution in expressions where it is necessary to have some deviation from the normal precedence. When the sequence of execution is not specified by parentheses, the order of execution of consecutive operations of the same hierarchical level is from left to right.

3) The ways in which operators, variables and parentheses may be combined in an arithmetic expression are summarized in the table where:

a) The letter 'P' indicates a permissible pair of symbols

b) The character '-' indicates an invalid pair

c) 'Variable' indicates an identifier or literal

Table of Combinations of Symbols in Arithmetic Expressions

First Symbol	*Second Symbol*				
	Variable	* / - + **	*Unary + or -*	*(*	*)*
Variable	-	P	-	-	P
* / + - **	P	-	P	P	-
Unary + or -	P	-	-	P	-
(	P	-	P	P	-
)	-	P	-	-	P

4) An arithmetic expression may only begin with the symbol '(', '+', '-', or a variable and may only end with a ')' or a variable. There must be a one-to-one correspondence between left and right parentheses of an arithmetic expression such that each left parenthesis is to the left of its corresponding right parenthesis.

5) Arithmetic expressions allow the user to combine arithmetic operations without the restrictions on composite of operands and/or receiving data items. See, for example, syntax rules given for the ADD statement.

6.3 ARITHMETIC STATEMENTS

The arithmetic statements are the ADD, COMPUTE, DIVIDE, MULTIPLY and SUBTRACT statements. They have several common features.

1) The data descriptions of the operands need not be the same; any necessary conversion and decimal point alignment is supplied throughout the calculation.

2) The maximum size of each operand is eighteen (18) decimal digits. The composite of operands, which is a hypothetical data item resulting from the superimposition of specified operands in a statement aligned on their decimal points, must not contain more than eighteen decimal digits.

Overlapping Operands

When a sending and receiving item in an arithmetic statement or an INSPECT, MOVE, SET, STRING or UNSTRING statement share a part of their storage areas, the result of the execution of such a statement is undefined.

Multiple Results in Arithmetic Statements

The ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT statements may have multiple results. Such statements behave as though they had been written in the following way:

1) A statement which performs all arithmetic necessary to arrive at the result to be stored in the receiving items, and stores that result in a temporary storage location.

2) A sequence of statements transferring or combining the value of this temporary location with a single result. These statements are considered to be written in the same left-to-right sequence the multiple results are listed in.

Incompatible Data

Except for the class condition (see the next section on Conditional Expressions), when the contents of a data item are referenced in the Procedure Division and the contents of that data item are not compatible with the class specified for that data item by its PICTURE clause, the result of such a reference is undefined.

6.3.1 Common Options

The three options common to the arithmetic statements are ROUNDED, SIZE ERROR and CORRESPONDING. They are described in the following subsections.

6.3.1.1 The ROUNDED Option

If, after decimal point alignment, the number of places in the fraction of the result of an arithmetic operation is greater than the number of places provided for the fraction of the resultant-identifier, truncation is relative to the size provided for the resultant-identifier. When rounding is requested, the absolute value of the resultant-identifier is increased by one (1) whenever the most significant digit of the excess is greater than or equal to five.

6.3.1.2 The SIZE ERROR Option

If, after decimal point alignment, the absolute value of a result exceeds the largest value that can be contained in the associated resultant-identifier, a size error condition exists. Division by zero always causes a size error condition. The size error condition applies only to the final results of an arithmetic operation. It does not apply to intermediate results, except in the MULTIPLY and DIVIDE statements, in which case the size error condition applies to the intermediate results as well.

If the ROUNDED phrase is specified, rounding takes place before checking for size error. When such a size error condition occurs, the subsequent action depends on whether or not the SIZE ERROR phrase is specified.

1) If the SIZE ERROR phrase is not specified and a size error condition occurs, the value of the resultant-identifier(s) affected is underlined. Values of resultant-identifier(s) for which no size error condition occurs, are unaffected by size errors that occur for other resultant-identifier(s) during this operation.

2) If the SIZE ERROR phrase is specfied and a size error condition occurs, then the values of resultant-identifier(s) affected by the size errors are not altered. Values of resultant-identifier(s) for which no size error condition occurs are unaffected by size errors occurring for other resultant-identifier(s). After execution is complete, the imperative-statement in the SIZE ERROR phrase is performed.

6.3.1.3 The CORRESPONDING Option

This option allows operations to be performed on elementary items of the same name by specifying the group items to which they belong. The following rules apply:

1) Both identifiers used must be group items.

2) CORRESPONDING is equivalent to the abbreviation CORR and is valid for the MOVE statement.

3) A pair of data items from two different group items correspond if the following conditions are true:

   a) The two data items have the same name and the same qualifiers up to but not including the group level.

   b) In the case of a MOVE statement with the CORRESPONDING option, at least one of the data items is an elementary item.

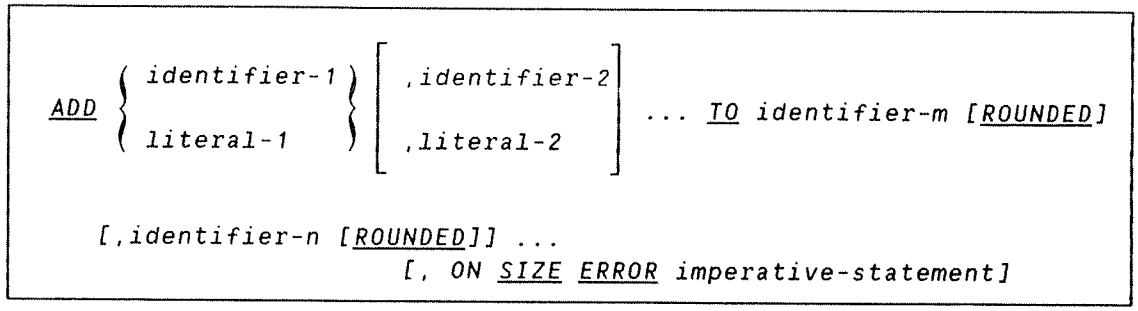
   c) The two data items do not include a REDEFINES, OCCURS, or USAGE IS INDEX clause. Such items will be ignored together with any subordinate items containing REDEFINES, OCCURS or USAGE IS INDEX clauses.

d) The group items themselves, however, may contain or be subordinate to data items containing REDEFINES or OCCURS clauses.

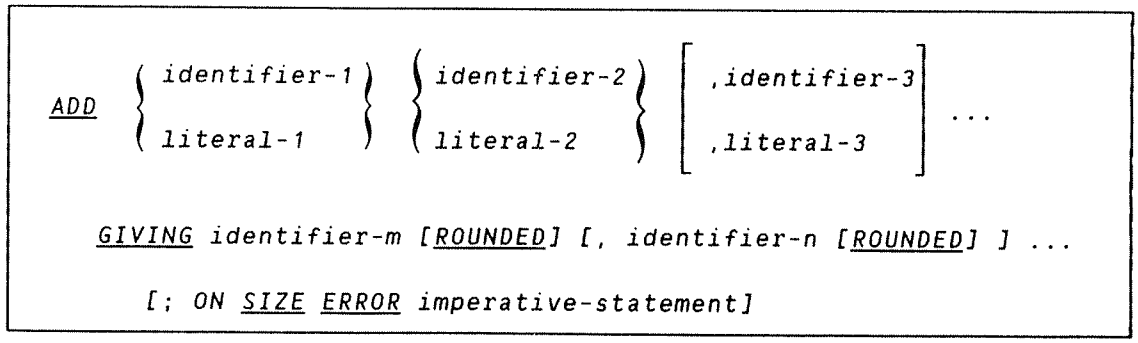
6.3.1.4 The ADD Statement

The ADD statement adds together two or more numeric operands, and stores the resulting sum.

Format 1:



Format 2:



In format 1, each identifier must name an elementary numeric item.

In format 2, each identifier, except those following the word GIVING, must be elementary numeric items. Each identifier following the word GIVING must be either elementary numeric or numeric edited items.

Each literal must be a numeric literal.

When the TO option is used (format 1), all identifiers preceding it are added together and then added to and stored immediately in identifier-m. Then, if specified, they are added to identifier-n and stored there.

When the GIVING option is used (format 2), the values of the preceding operands are added together and the sum is stored as the new value of identifier-m and (if specified) identifier-n, etc.

For the ROUNDED and SIZE ERROR options, see section 6.3.1 Common Options.

6.3.1.5 The COMPUTE Statement

The COMPUTE statement assigns the value of an arithmetic expression to one or more data items.

Format:

```
COMPUTE identifier-1 [ROUNDED] [, identifier-2 [ROUNDED] ] ...
    = arithmetic expression [; ON SIZE ERROR imperative-statement]
```

Identifiers that only appear to the left of = must refer to either elementary numeric or elementary numeric edited items.

The COMPUTE statement allows the user to combine arithmetic operations without the restrictions on receiving items in the ADD, SUBTRACT, MULTIPLY and DIVIDE statements. (When arithmetic operations must be combined, using the COMPUTE statement is more efficient than writing the separate arithmetic statements in series.)

When the COMPUTE statement is executed, the value of the arithmetic statement is calculated and then this value is stored as the new value of identifier-1, identifier-2, etc. (The arithmetic expression can be any arithmetic expression as defined earlier in this chapter.)

For the ROUNDED and SIZE ERROR phrases, see 'Common Options' earlier in this section.

An arithmetic expression consisting of a single identifier or literal provides a method of setting the values of identifier-1, identifier-2, etc. equal to the values of that single identifier or literal.

The number of integer and decimal places provided by the compiler for intermediate results is shown in appendix 11. It is the user's responsibility to define the operands of any arithmetic statement so that they have large enough fields to provide the required accuracy of results.

6.3.1.6 The DIVIDE Statement

The DIVIDE statement divides one numeric data item into others and stores the resultant values as quotient and remainder.

Format 1:

```
DIVIDE { identifier-1 }
       { literal-1    }  INTO identifier-2 [ROUNDED]

   [, identifier-3 [ROUNDED] ] ...

      [;ON SIZE ERROR imperative-statement]
```

Format 2:

```
DIVIDE { identifier-1 } { INTO } { identifier-2 }
       { literal-1    } { BY   } { literal-2    }  GIVING

   identifier-3 [ROUNDED] [, identifier-4 [ROUNDED] ]...

      [;ON SIZE ERROR imperative-statement]
```

Format 3:

```
DIVIDE { identifier-1 } { INTO } { identifier-2 } GIVING
       { literal-1    } { BY   } { literal-2    }

    identifier-3 [ROUNDED] [REMAINDER identifier-4]

        [;ON SIZE ERROR imperative-statement]
```

Each identifier, except those following the words GIVING and REMAINDER, must be elementary numeric items. The identifiers following GIVING and REMAINDER may also be numeric edited items.

Each literal must be a numeric literal.

In format 1, the value of identifier-1 or literal-1 is divided into the value of identifier-2, and the quotient obtained replaces this value. Similarly for identifiers 3, ... n, if specified.

In format 2, only one division takes place; the value of identifier-1 or literal-1 is divided into/by the value of identifier-2 or literal-2, and the quotient is then stored in identifier-3 and (if specified) identifier-4, etc.

In format 3, the division process is as for format 2 except that the quotient is stored in identifier-3 and the value of the remainder in identifier-4.

For the ROUNDED and SIZE ERROR options, see section 6.3.1 on Common Options.

6.3.1.7 The MULTIPLY Statement

The MULTIPLY statement computes the product of two numeric data items and stores it.

Format 1:

```
MULTIPLY { identifier-1 } BY identifier-2 [ROUNDED]
         { literal-1    }

   [, identifier-3 [ROUNDED] ] ...

      [;ON SIZE ERROR imperative-statement]
```

Format 2:

```
MULTIPLY { identifier-1 } BY { identifier-2 }
         { literal-1    }    { literal-2    }

   GIVING identifier-3 [ROUNDED] [, identifier-4 [ROUNDED] ] ...

      [;ON SIZE ERROR imperative-statement]
```

Each literal must be a numeric literal.

Each identifier must be a numeric elementary item, except that identifiers following the word GIVING in format 2 may also be elementary numeric items.

In format 1, identifier-2 is replaced by the product of it and the first operand. This process is continued for all subsequent identifiers.

When format 2 is used, the value of identifier-1 or numeric-literal-1 is multiplied by the value of the second operand. The result is stored in identifier-3, identifier-4, etc.

6.3.1.8 The SUBTRACT Statement

The SUBTRACT statement subtracts one or more numeric data items from one or more items and stores the results.

Format 1:

```
SUBTRACT { identifier-1 } [ ,identifier-2 ] ...
         { literal-1    } [ ,literal-2    ]

    FROM identifier-m [ROUNDED] [, identifier-n [ROUNDED]] ...

       [;ON SIZE ERROR imperative-statement]
```

Format 2:

```
SUBTRACT { identifier-1 } [ ,identifier-2 ] ...
         { literal-1    } [ ,literal-2    ]

    FROM { identifier-m }
         { literal-m    }

       GIVING identifier-n [ROUNDED] [, identifier-o [ROUNDED] ] ...

          [;ON SIZE ERROR imperative-statement]
```

Each identifier must represent a numeric elementary item, except when following the word GIVING when it may also be an elementary numeric edited item.

In format 1, the identifiers or literals preceding FROM are added together and subtracted from identifiers m, n, ... in turn. After each subtraction, the results are stored in these identifiers m, n, ...

In format 2, the identifier or literals preceding FROM are added together and subtracted from identifier-m or literal-m. The result of the subtraction is stored as the new value of identifier-n and any other specified identifiers.

6.4 CONDITIONAL EXPRESSIONS

Conditional expressions identify conditions that are tested to enable the object program to select between alternate paths of control depending upon the truth value of the condition. Conditional expressions are specified in the IF, PERFORM and SEARCH statements. There are two categories of conditions associated with conditional expressions: simple conditions and complex conditions.

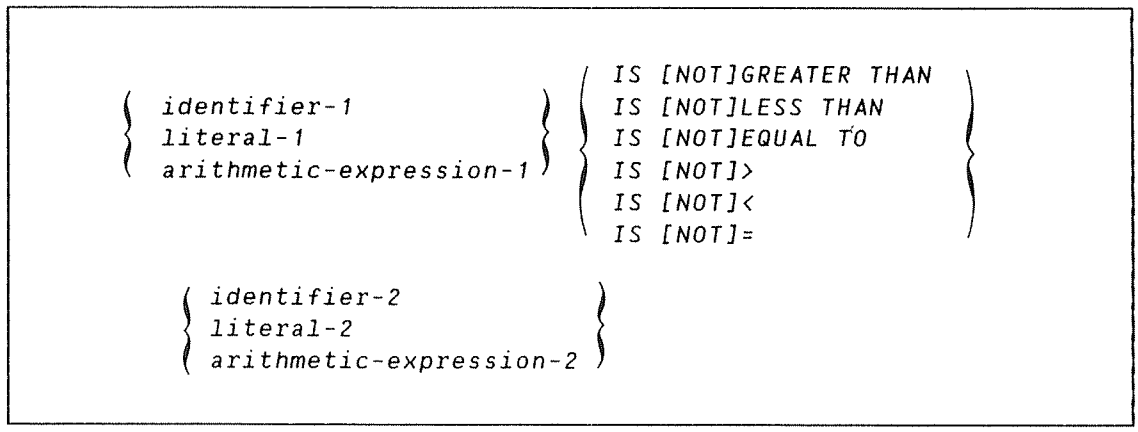
Simple Conditions

The *simple conditions* are the relation, class, condition-name and sign conditions. A simple condition has a truth value of 'true' or 'false'.

Relation Condition

A *relation condition* causes a comparison of two operands, each of which may be the data item referenced by an identifier, a literal, or the value resulting from an arithmetic expression. A relation condition has a truth value of 'true' if the relation exists between the operands. Comparison of two numeric operands is permitted regardless of the formats specified in their respective USAGE clauses. However, for all other comparisons the operands must have the same usage. If either of the operands is a group item, the nonnumeric comparison rules apply.

Format:



NOTE: The required relational characters '>', '<', and '=' are not underlined in this format to avoid confusion with other symbols such as " (greater than or equal to).

Comparison of Numeric Operands

For operands whose class is numeric (refer to the Data Division, Classes and Categories of Data), the algebraic values of the operands are compared. The length of the literal or arithmetic expession operands, in terms of number of digits represented, is not significant. Zero is considered an unique value regardless of the sign.

Comparison of these operands is permitted regardless of the manner in which their usage is described. Unsigned numeric operands are considered positive for purposes of comparison.

Comparison of Nonnumeric Operands

For nonnumeric operands, or one numeric and one nonnumeric operand, a comparison is made with ND's standard character set. If one of the operands is numeric it must be an integer data item or integer literal and the following rules apply:

a) If the nonnumeric operand is an elementary data item or a literal, the numeric operand is treated as though it were moved to an elementary alphanumeric data item of the same size, and the contents of this alphanumeric item were then compared to the nonnumeric operand.

b) If the nonnumeric item is a group item, the numeric item is treated as though it were moved to a group item of the same size, and the contents of this group were compared to the nonnumeric operand.

c) A non-integer numeric operand cannot be compared to a nonnumeric operand.

The size of an operand is the total number of characters contained in it.

If the operands are *equal* in size:

Characters in corresponding positions are compared, beginning with the leftmost character. If a pair of unequal characters is encountered, they are tested to ascertain their relative positions in the collating sequence. The operand having the character higher in the sequence is considered to be the greater operand.

If the operands are *unequal* in size:

The comparison is made as if the shorter operand were extended to the right with enough spaces to make the operands of equal size.

Class Condition

The class condition determines whether the operand is alphabetic or numeric.

Format:

identifier IS [NOT] { *NUMERIC* / *ALPHABETIC* }

The identifier is determined to be numeric if its contents consist only of a combination of the digits 0 through 9.

If its PICTURE does not contain an operational sign, then the identifier is considered as numeric if the contents are numeric and an operational sign is not present. Otherwise, if its PICTURE contains an operational sign, the identifier is considered to be numeric if it is an elementary item with numeric contents and an operational sign.

Valid operational signs are:

For items described with the SIGN clause -

+ (53 octal) and - (55 octal)

The embedded operation signs -

+0 to +9 = 173, 101 to 111 (octal)
-0 to -9 = 175, 112 to 122 (octal)

For PACKED-DECIMAL items, see under Computational Options.

The NUMERIC test is not valid for alphabetic items or for group items which have operational signs present in items subordinate to them.

The ALPHABETIC test cannot be used with an item whose data description describes the item as numeric. The item being tested is determined to be alphabetic only if the contents consist of any combination of the alphabetic characters 'A' through 'Z' and the space.

Condition-Name Condition (Conditional Variable)

This condition determines whether a conditional variable has a value equal to any of the value(s) associated with the condition-name.

Format:

condition-name;

The use of this condition is as an abbreviation for the relation condition, and the rules for comparing a conditional variable with a condition-name are the same as specified for the relation condition.

If the condition-name is associated with a range or ranges of values, then the conditional variable is tested to determine whether or not its value falls in this range, including the end values.

The result of the test is true if one of the values corresponding to the condition-name equals the value of its associated conditional variable.

As an example of its use, if the following is specified:

```
05  TYPE-REC PIC X.
  88  TYPE-1 VALUE A THRU F.
  88  TYPE-2 VALUE H.
  88  TYPE-3 VALUE J THRU Z.
```

(where TYPE-REC is a conditional variable) then, to determine a type classification of a record, the code:

IF TYPE-1 ...

can cause a branch for values of A, B, C, D, E or F. (Refer to VALUE clause in the Data Division and to 'Comparison of Nonnumeric Operands' earlier in this section.)

Sign Condition

The sign condition determines whether or not the algebraic value of an arithmetic expression is less than, greater than, or equal to zero.

General format for a sign condition:

arithmetic-expression IS [NOT] { *POSITIVE* / *NEGATIVE* / *ZERO* }

When used, 'NOT' and the next key word specify one sign condition that defines the algebraic test for truth value; e.g., 'NOT ZERO' is a truth test for a nonzero (positive or negative) value. An operand is positive if its value is greater than zero, and zero if its value is equal to zero. The arithmetic expression must contain at least one reference to a variable.

Complex Conditions

A complex condition is formed by combining simple conditions, combined conditions and/or complex conditions with logical connectors (logical operators 'AND' and 'OR'), or negating these conditions with logical negation (the logical operator 'NOT'). The truth value of a complex condition, whether parenthesized or not, is that which results from the interaction of all the logical operators on the individual values of simple conditions, or the intermediate values of conditions logically connected or logically negated.

The logical operators and their meanings are:

Logical Operator:	*Meaning:*
AND	*Logical conjunction; the truth value is 'true' if both conditions are true; 'false' if one or both conditions is false.*
OR	*Logical inclusive OR; the truth value is 'true' if one or both of the conditions is true; 'false' if both conditions are false.*
NOT	*Logical negation or reversal of truth value; the truth value is 'true' if the condition if false; 'false' if the condition is true.*

The logical operators must be preceded by a space and followed by a space.

Negated Simple Conditions

A simple condition (see earlier in this section) is negated through the use of the logical operator 'NOT'. The negated simple condition effects the opposite truth value for a simple condition.

Format:

```
NOT simple condition;
```

Combined Conditions

A combined condition results from connecting conditions with one of the logical operators 'AND' or 'OR'.

Format:

```
condition { { AND } } condition ... }
          { { OR  } }
```

where 'condition' may be:

1) A simple condition, or

2) A negated simple condition, or

3) A combined condition, or

4) A negated combined condition (i.e., the 'NOT' logical operator followed by a combined condition enclosed within parentheses), or

5) Combinations of the above, specified according to the rules summarized in the following table, Combinations of Conditions, Logical Operators, and Parentheses.

Parentheses are optional when 'AND', 'OR' or 'NOT' are used. The table indicates the ways in which conditions and logical operators may be combined and parenthesized. There must be a one-to-one correspondence between left and right parenthesis such that each left parentheses is to the left of its corresponding right parenthesis.

TABLE OF COMBINATIONS OF CONDITIONS, LOGICAL OPERATORS, AND PARENTHESES

Given the following element	Location in conditional expression		In a left-to-right sequence of elements:	
	First	Last	Element, when not first, may be immediately preceded by only:	Element, when not last, may be immediately followed by only:
simple-condition	Yes	Yes	OR, NOT, AND, (	OR, AND,)
OR or AND	No	No	simple-condition,)	simple-condition, NOT, (
NOT	Yes	No	OR, AND, (	simple-condition, (
(	Yes	No	OR, NO, AND, (	simple-condition, NOT, (
)	No	Yes	simple-condition,)	OR, AND,)

Thus, the element pair 'OR NOT' is permissible whereas the pair 'NOT OR' is not permissible; 'NOT (' is permissible whereas 'NOT NOT' is not permissible.

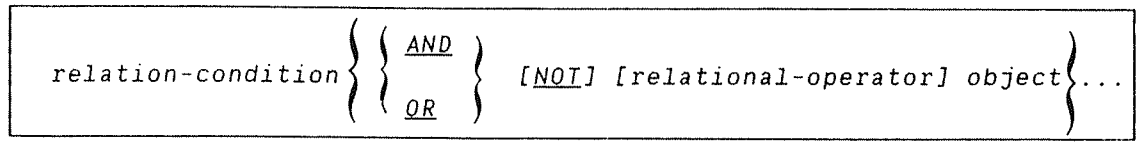
Abbreviated Combined Relation Conditions

When simple or negated simple relation conditions are combined with logical connectives in a consecutive sequence, such that a succeeding relation condition contains a subject or subject and relational operator that is common with the preceding relation condition (and no parentheses are used within such a consecutive sequence), any relation condition except the first may be abbreviated by:

1) The omission of the subject, or

2) The omission of the subject and relational operator.

Format for an abbreviated combined relation condition:



Within a sequence of relation conditons, both of the above forms of abbreviation may be used. The effect of using such abbreviations is as if the last preceding stated subject were inserted in place of the omitted subject, and the last stated relational operator were inserted in place of the omitted relational operator. The result of such implied insertion must comply with the rules given in the table, Combinations of Conditions, Logical Operators and Parentheses, shown above. This insertion of an omitted subject and/or relational operator terminates once a complete simple condition is encountered within a complex condition.

The interpretation of the word 'NOT' in an abbreviated combined relation condition is as follows:

1) If the word immediately following 'NOT' is 'GREATER', '>', 'LESS', '<', EQUAL, '=', then the 'NOT' participates as a part of the relational operator; otherwise

2) The 'NOT' is interpreted as a logical operator and, therefore, the implied insertion of subject or relational operator results in a negated relation condition.

Some examples of abbreviated combined and negated combined relation conditions and expanded equivalents follow.

Abbreviated Combined Relation Condition	*Expanded Equivalent*
a > b AND NOT < c OR d	*((a > b) AND (a NOT < c))OR (a NOT < d)*
a NOT EQUAL b OR c	*(a NOT EQUAL b) OR (a NOT EQUAL c)*
NOT a = b OR c	*(NOT (a = b))OR (a = c)*
NOT (a GREATER b OR < c)	*NOT ((a GREATER b) OR (a < c))*
NOT (a NOT > b AND c AND NOT d)	*NOT ((((a NOT > b) AND (a NOT > c)) AND (NOT (a NOT > d))))*

Condition Evaluation Rules

Parentheses may be used to specify the order in which conditions are evaluated when it is necessary to depart from the implied evaluation sequence. In this case, logical evaluation proceeds in the following order:

1) Conditions within parentheses are evaluated first.

2) Within nested parentheses, evaluation proceeds from the least inclusive condition to the most inclusive condition. If parentheses are not used, then the evaluation order is:

    1) Arithmetic expressions

2) Simple conditions in the following order:

relation

class

condition-name

sign

3) Negated simple-conditions in the same order as in 2.

4) Combined conditions in the following order:

OR

AND

NOT

5) Negated combined conditions in the order as in 4.

Consecutive operands at the same hierarchical level are evaluated from left to right.

6.5 CONDITIONAL STATEMENTS

6.5.1 The IF Statement

The IF statement causes a condition to be evaluated. The subsequent execution sequence depends upon whether the condition is true or false.

Format 1:

```
IF condition { statement-1      } [ ELSE statement-2      ]
             { NEXT SENTENCE   }   [ ELSE NEXT SENTENCE ]
```

Format 2 (An ND Extension):

```
IF condition THEN { statement-3      } [ ELSE statement-4      ]
                  { NEXT SENTENCE   }   [ ELSE NEXT SENTENCE ]

   [END-IF]
```

Format 3 (An ND Extension):

```
IF condition-1 THEN { statement-5      }
                    { NEXT SENTENCE   }

[ ELSE-IF condition-2 THEN { statement-6      } ] ...
                           { NEXT SENTENCE   }

[ ELSE { statement-7      }    [ END-IF ] ]
       { NEXT SENTENCE   }
```

General Rules for Format 1:

1) If the condition tested is true, one of the following actions takes place:

   a) Statement-1, if specified, is executed. If this contains a procedure-branching statement, control is transferred according to the rules of that statement. If it does not, the ELSE phrase, if specified, is ignored and control passes to the next executable sentence.

   b) If the NEXT SENTENCE phrase is specified instead of statement-1, the ELSE phrase, if present, is ignored and control passes to the next executable sentence.

2) If the condition tested is false, one of the following actions occurs:

   a) ELSE statement-2, if specified, is executed. If this statement contains a procedure-branching statement, control is transferred according to the rules for that statement. Otherwise control is passed to the next executable sentence.

   b) ELSE NEXT SENTENCE, if specified, is executed, i.e. statement-1, if present, is ignored and control passes to the next executable sentence.

   c) If ELSE NEXT SENTENCE is omitted, control passes to the next executable sentence.

3) Statement-1 and/or statement-2 may contain an IF statement. In this case, the statement is said to be nested. Statements 1 and 2 represent either an imperative statement or a conditional statement. Either of these may be followed by a conditional statement.

4) The ELSE NEXT SENTENCE option may be omitted if it immediately precedes the terminal period of the sentence.

General Rules for Format 2:

1) Statements 3 and 4 represent imperative statements.

2) If the condition is true and the ELSE clause is omitted, then if statement-3 has been coded, this statement together with any further imperative statements preceding the sentence terminator, will be executed. Control is then passed implicitly to the next sentence unless a GO TO procedure-name appears in statement-3. If the condition is true and NEXT SENTENCE is coded, control passes explicitly to the next sentence.

3) If the condition is true and the ELSE clause is present then statement-4 (together with any further imperative statements preceding the sentence terminator) is executedor the NEXT SENTENCE of this clause. If the ELSE clause is absent, control passes to the next sentence following the END-IF.

4) No period character (.) should occur between the IF and END-IF verbs inclusively.

General Rules for Format 3:

1) Statements 5, 6 and 7 are imperative statements.

2) If the ELSE-IF clause is omitted then the rules are as for format 2. (Except that statement-5 should be substituted for statement-3 and statement-7 for statement-4.)

3) If condition-1 is false, then if condition-2 is true, the rules are as for format-2 if statement-6 is substituted for statement-3 and statement-7 for statement-4.

6.5.1.1 Nested IF Statements

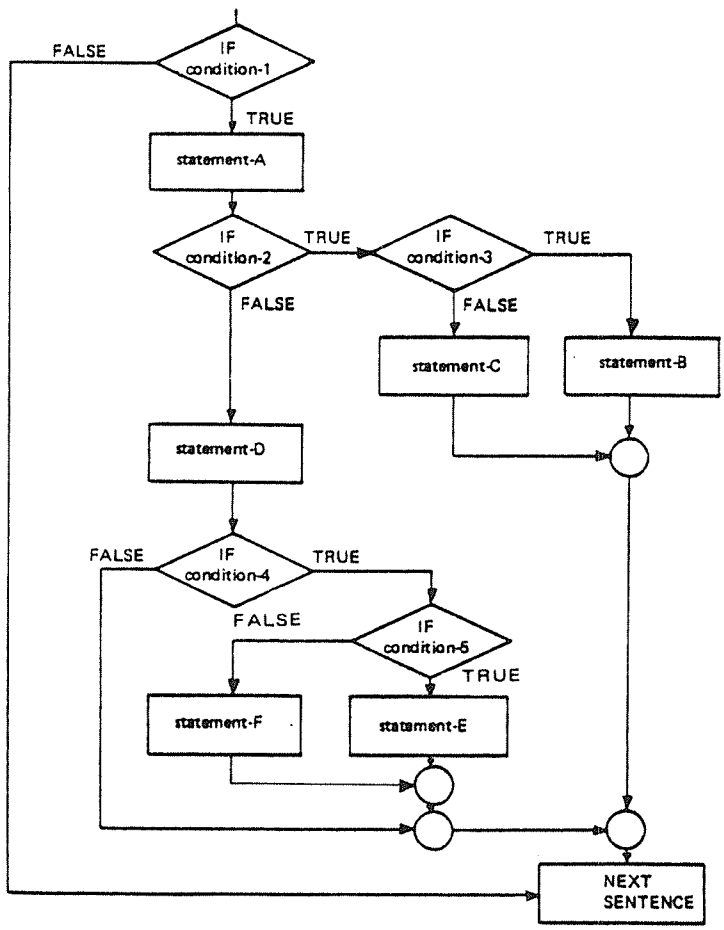
The presence of one or more IF statements within an initial IF statement constitutes a "nested" IF statement. Statements 1 and 2 may consist of one or more imperative statements and/or a conditional statement. If an IF statement appears as the whole or part of statements 1 or 2, it is said to be nested.

IF statements within IF statements may be considered as paired IF and ELSE combinations, proceeding from left to right. Thus, any ELSE encountered is considered to apply to the immediately preceding IF that has not already been paired with an ELSE.

The structure of a possible nested IF statement may be as follows:

```
IF condition-1 statement-A
   IF conditon-2
      IF condition-3 statement-B
      ELSE statement-C
   ELSE statement-D
      IF condition-4
      IF condition-5statement-E
         ELSE statement-F.
```

The Flowchart for this example would appear as:



6.5.2 The DO Statement (An ND-Extension)

A DO statement specifies a loop which can be used for coding iterative procedures. There are two basic formats:

Format 1:

```
DO [statement] [ { WHILE condition } sentence] ... END-DO
```

Format 2:

```
DO FOR identifier-1 FROM { identifier-2 [{+} integer-1] }
                         {          integer-2           }

   [ UP   ]
   [ DOWN ]  [BY integer-3] TO { identifier-5 [{+} integer-4] }
                               {          integer-5           }

        sentence [ {WHILE condition} sentence] ... END-DO
```

In format 2, the identifiers must be numeric items whose PICTURE specification does not contain a decimal point. Integers 2, 3, and 5 are initial, incremental, and terminal parameters respectively and they must be integers. The incremental parameter should be greater than or equal to 1, if it is not present it is assumed to be 1 (one). The UP/DOWN options, if specified, denote positive or negative increments, respectively.

At execution time, the identifier takes the value of the initial parameter, and the loop is performed until either the initial parameter is greater than the terminal parameter, or until the condition in the WHILE phrase (if present) is no longer true. Control then passes to the next executable statement following the corresponding END-DO statement.

In format 1, the identifier must be specified as in format 2. If the WHILE condition phrase does not appear, the DO-loop may be regarded as an infinite loop (see Example 3 for an example of its use).

The WHILE condition phrase which appears in both formats, may also appear any number of times within the DO-loop. DO-loops may be nested up to 50 levels. Any DO-loop may be left via the EXIT verb. (See also EXIT-DO and EXIT-ALL-DO in section 6.8.3.)

EXAMPLE 1:

```
      DO FOR N FROM 1 BY 1 TO 50
      MOVE CORRESPONDING MASTER-REC(N) TO OUT-REC(N).
      WRITE OUTRECT(N).
      END-DO.
```

EXAMPLE 2:

```
      DO WHILE I <100.
      WHILE M = N.
      WHILE P NOT EQUAL R OR S.
      WRITE OUT-FILE.
      END-DO.
```

EXAMPLE 3:

```
      DO.
*****************************************
* read file with unknown number of records
*****************************************
      READ FILE IN-FILE AT END EXIT-DO.
      END-DO.
```

6.6 DATA MANIPULATION STATEMENTS

6.6.1 Screen Handling Facilities

Screen Handling for COBOL is an ND Extension for which the following Data Manipulation Statements can be used. These are ACCEPT (format 3), ACCEPT-ERROR, ACCEPT-RETURN, BLANK, RESET and DISPLAY (format 2), and they are described individually below. Section 6.6.2 provides a few examples of screen handling in which the function and interaction of these statements are demonstrated.

These features can be used on terminals which are suitable for the ND editors NOTIS-WP, PED, etc.

The ESC key is automatically disabled when one of the following screen handling statements is used:

1) BLANK

2) ACCEPT {position specifier}

3) DISPLAY {position specifier}

This is done to avoid nonsensical screen pictures and inconsistent indexed and random access files. The ESC key is automatically enabled when the program terminates.

6.6.1.1 The ACCEPT Statement

The ACCEPT statement allows the user to enter data into specified identifiers from her terminal.

Note that when the user has started entering or editing data in the data field of an ACCEPT-statement, the data field can only be left through

1) pressing the carriage return key;

2) pressing the CANCEL key if any actions for that key are specified in the ACCEPT statement. When a data field is left through CANCEL, the contents of the field will be left as they were when the key was pressed;

3) reaching the end of the data field if the AUTO-SKIP option has been specified.

While entering and editing data, the NOTIS conventions for cursor movements etc. will be followed.

Format 1 - Data Transfer:

ACCEPT identifier

Format 2 - System Information Transfer:

```
ACCEPT identifier FROM { DATE     }
                       { DAY      }
                       { TIME     }
                       { CPU-TIME }
```

Format 3 - Screen Handling:

```
ACCEPT position specifier identifier [WITH

   [BEEP]
   [SPACE-FILL]
   [LENGTH-CHECK]
   [AUTO-SKIP]
   [PROMPT]
   [BLANK-WHEN-ZERO]
   [MUST]
   [UPDATE]
   [JUSTIFIED-RIGHT]
   [INVISIBLE]
   [INVERSE-VIDEO]
   [BLINK]
   [UNDERLINE]
   [UPPER-CASE]
   [LOW-INTENSITY]
   [NORMAL]
   [HELP Label]
   [RE-DISPLAY Label]
   [CANCEL Label]
   [F1-F8 Label]
   [UP Label]
   [DOWN Label]
   [HOME Label]
   [EXIT Label]
   [LEFT Label]
   [RIGHT Label]
   [CONTROL Label]]
```

where:

Label is a paragraph or a section name, and

identifier is the name of the receiving field.

*[F1-F8 Label]*Label denotes the actions to be taken when one of the function keys on the user's terminal is pressed. This adds 16 more keys to the ones available with the ACCEPT-statement - 8 shifted and 8 unshifted. Details will be given later in this section.

Position specifier is the screen position defined as:

```
(Line, column)
```

both *line* and *column* being defined by:

```
{ identifier [ { + } integer ] }
{          [ { - }           ] }
{ integer                      }
```

Format 1 is used to transfer data from an input-output device into the identifier. The input device is assumed to be the system console in the case of RT-users, and a screen terminal in the case of Timesharing and Batch users. When running batch or mode files in background mode, data is accepted from the next line on the respective file. (IF the FROM option is present, then the mnemonic-name is treated as a comment only.)

Format 2 is used to transfer system information (DAY, DATE, TIME, CPU-TIME) into the identifier according to the rules of the MOVE statement.

DATE is composed of a sequence of data elements as follows:

2 digits for year of century, 2 digits for month of year, 2 digits for day of month. Therefore, September 1, 1980 would be expressed as 800901.

DAY has the sequence of data elements as follows:

2 digits for year of century, 3 digits for day of year. Thus, September 1, 1980 is expressed as 80245.

TIME is composed of the data elements hours, minutes, seconds and hundreths of a second. For example, 2:41 p.m. would be expressed as 14410000.

CPU-TIME consists of the data element CPU-time expressed in milliseconds.

In format 3, the receiving field (identifier) is described by a PICTURE or USAGE specification. The data input field is a string of character positions, starting at the location indicated by the position specifier. Valid data which may be entered is governed by the rules for the associated PICTURE specification (see the Picture Clause

The identifier may have its USAGE described as COMPUTATIONAL (see Section 5.4.2.10). In this case, if the PICURE clause is omitted, then the size of the field for single-word items is 5 + a sign position, and for double-word items the size is 10 + the sign position.

The identifier may also have its USAGE described as PACKED-DECIMAL.

Format 3 is used to accept data into a field from a screen. The options in the WITH phrase which describe the appearance of the field on the screen, can appear in any order or combination. However, in some cases the type of options which are operative simultaneously will dependent on the terminal type.

The effects of each option are as follows:

BEEP will sound the terminal's audio alarm when the system is ready to ACCEPT the field.

SPACE-FILL is for use with numeric fields. Where the identifier has a PICTURE specification of 9's only, leading zeros are set to blanks. (On the screen only.)

LENGTH-CHECK causes the entry of a field terminator to be ignored until each input position has been operated upon.

AUTO-SKIP specifies that when an input field has been filled by the operator, the field will be terminated automatically.

PROMPT sets the data input field on the screen to indicate that all positions contain the period character (".") before input is accepted.

UPDATE will initialize the data input field with the initial contents of the receiving field. These data can be edited as if they were typed by the operator. UPDATE and PROMPT can be used in the same ACCEPT statement.

INVISIBLE will prevent the data entered into the input field from being displayed on the screen. This may be required for security reasons, such as when typing passwords etc.

INVERSE-VIDEO produces a bright background in the identifier display area.

BLINK causes the display of the identifier to flash on and off.

UNDERLINE underlines the identifier.

LOW-INTENSITY reduces the intensity of the display.

NORMAL resets the effect of a previous INVERSE-VIDEO, LOW-INTENSITY, BLINK or UNDERLINE.

MUST implies that some data must be entered into the field of the ACCEPT statement before it can be left.

BLANK-WHEN-ZERO causes the item to be displayed as all blanks if the item's value is zero.

UPPER-CASE causes the field to be converted to upper case when it contains input data.

JUSTIFIED-RIGHT causes right justification of the field when it has been entered. The option can be used with alphanumeric fields only.

UP, DOWN, HOME, EXIT, LEFT, and RIGHT
represent terminal-dependent control keys which are used for moving the cursor between specific data input fields. These data input fields are identified by the name of the paragraph or section in which they occur in the Procedure Division.

CONTROL label provides the user with an opportunity to test for errors of her own definition. On entering carriage return, the section or paragraph with the label "Label" receives control. If a user-defined error is found then an ACCEPT-ERROR or ACCEPT-RETURN statement following the test will return control to the ACCEPT statement, at the end of the section or paragraph. The field must now be reentered. If the possible error did not occur, then the statement following the ACCEPT is executed.

F1-F8 label

The label denotes a paragraph or section to be PERFORMed when one of the function keys is pressed during data entry. By using the shift key, up to 16 different actions may be defined. On exit from the section or paragraph, the control will automatically be given to the current field or the next field according to the next section.

If one of the 16 possible keys has been hit, the program must identify which one it was by using the system variable C850. Details of C850 and other system variables with a table of the possible values they can have, are given in appendix 8.

HELP label

The label points to a paragraph or section to be used if the HELP key is pressed. The HELP information is best displayed inside a FRAME with AUTO-ERASE. (See further description in the DISPLAY-statement and the COB-GEN Manual.)

RE-DISPLAY label

This option is used together with the HELP option only, and only when the HELP information overlaps the original image. (See also the COB-GEN Reference Manual, ND-60.172.)

CANCEL label

Denotes a label that is the name of a paragraph/section which will be executed like the EXIT option, except:

- it will work regardless of the other field attributes.
- it will work inside an ACCEPT field as well as at the start.

Carriage return (CR) acts as a terminator character. If LENGTH-CHECK has not been coded, it terminates the ACCEPT. Then the cursor automatically moves to the beginning of the next data input field.

The carriage return may be used at any position in the data input field, unless LENGTH-CHECK has been coded with the associated ACCEPT statement.

Editing within data input fields of alphanumeric types before termination of the ACCEPT statement may be performed using CTRL+A or <DEL> key to delete a single character at a time, CTRL+E or <EXPAND> to insert characters, CTRL+W to delete all characters and left and right arrows to move the cursor inside the field.

CTRL+R twice or CTRL+F twice will position the cursor at the beginning or end of the ACCEPT field.

CTRL+A and CRTL+W may be used also in numeric fields.

Upon termination of the ACCEPT statement, data is transferred to the receiving field and edited according to the rules of the corresponding PICTURE specifications. With numeric fields there is an automatic display after "acceptance" of data input.

6.6.1.2 The ACCEPT-ERROR Statement

Format:

ACCEPT-ERROR

This statement is used in conjunction with an ACCEPT statement having the option CONTROL label or F1-F8 label. ACCEPT-ERROR is coded within the section or paragraph designated by "label". If a user-defined error has been detected, ACCEPT-ERROR causes a return to the ACCEPT statement, at the end of the section or paragraph. The field must now be reentered. The navigation keys (→, ←, ↑, ↓, \) will be disabled. If the user-defined error is not detected, control passes to the statement following the ACCEPT statement.

6.6.1.3 The ACCEPT-RETURN Statement

Format:

ACCEPT-RETURN

This statement has the same effect as ACCEPT-ERROR, except that the navigation keys (→, ←, ↑, ↓, \) will *not* be disabled.

6.6.1.4 The BLANK Statement

The BLANK statement erases the whole or a part of the screen.

Format 1:

```
BLANK SCREEN
```

Format 2:

```
BLANK [ LINE  ]  n1 [TO n2]  [COLUMN n3 TO n4]
      [ LINES ]
```

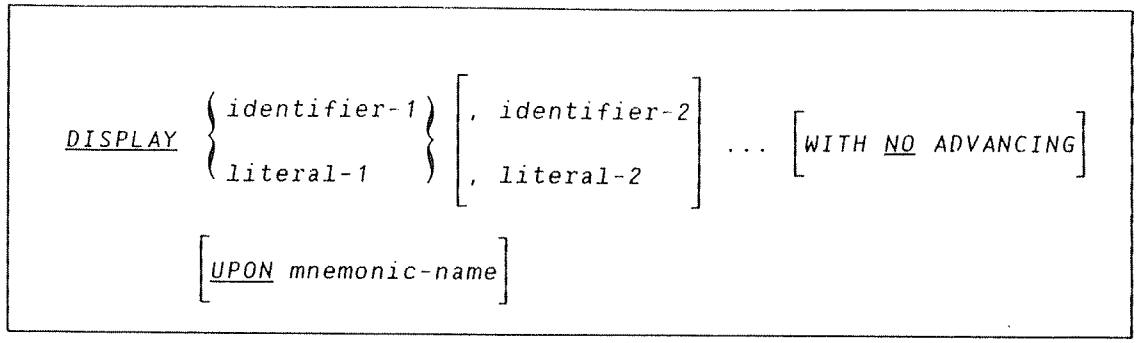
where n1, n2 , n3, and n4 must be integers or identifiers defined with no decimal point.

With format 1, the entire screen is erased, and the cursor is placed in the home position (line 1, column 1). Format 2 will blank out the line n1 to n2 inclusively, between columns n3 and n4 inclusively.

6.6.1.5 The DISPLAY Statement

The DISPLAY statement causes low volume data to be transferred to printing terminals or screens.

Format 1:



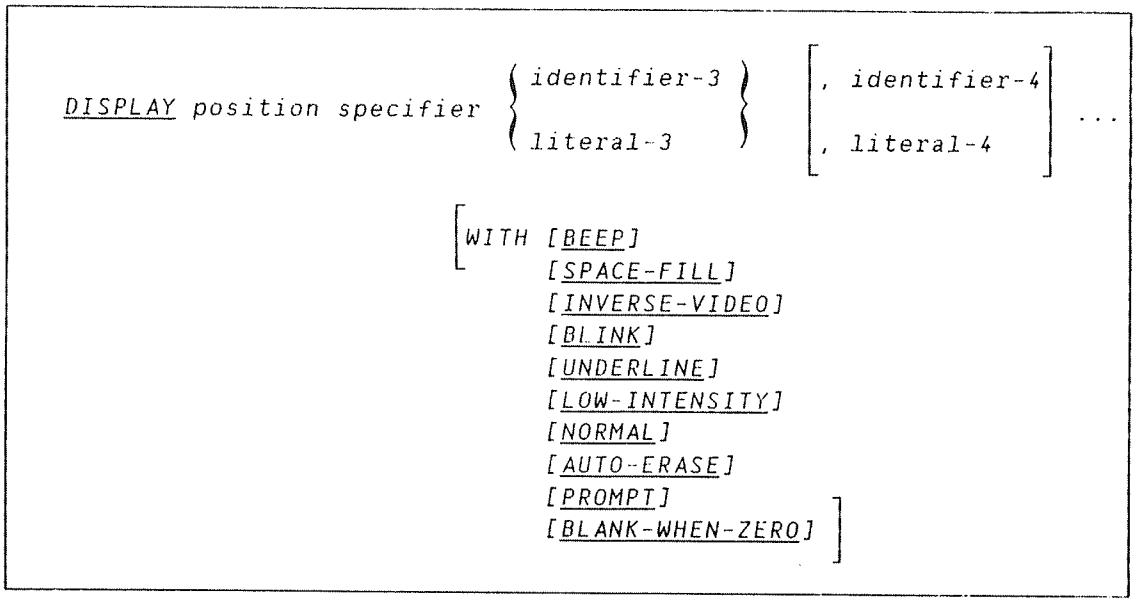
Literal-1 and literal-2 may be any figurative constant, except ALL.

The operand(s) are transferred to the system output device, if necessary with conversion.

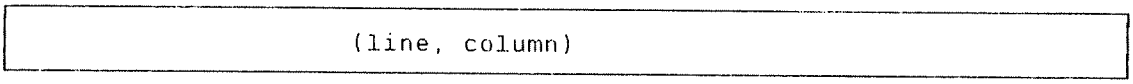
The UPON option has no effect and exists for syntax reasons only.

If the WITH NO ADVANCING phrase is specified, the system output device will not advance one line on the page before displaying the output. Otherwise, automatic advancement of one line will occur.

Format 2:



Position specifier, the screen position, is defined as:



both line and column being defined by:

```
{ identifier [ {+} integer] }
{                           }
{          integer          }
```

Format 2, which forms part of Screen Handling, displays data on a video terminal. Messages or the contents of a data item can appear on the screen with various forms of visual emphasis. The data consists of either literal-3 or identifier-3 and the display is described by the options listed in the WITH phrase. These options may appear in any order. However in some cases the number of options which may appear simultaneously will be terminal-dependent.

They have the following meanings:

BEEP the terminal beeps when the DISPLAY statement is initialized.

SPACE-FILL is for use where identifier-4 describes a numeric field. If the PICTURE specification contains only 9's, leading zeros are set to blanks (on the screen only).

INVERSE-VIDEO produces a bright background in the display area of identifier-4 or literal-4. The characters themselves appear in the normal background intensity.

BLINK the display of identifier-4 or literal-4 flashes off and on.

UNDERLINE underlines literal-4 or the contents of identifier-4 when they are displayed on the screen.

LOW-INTENSITY gives reduced display intensity.

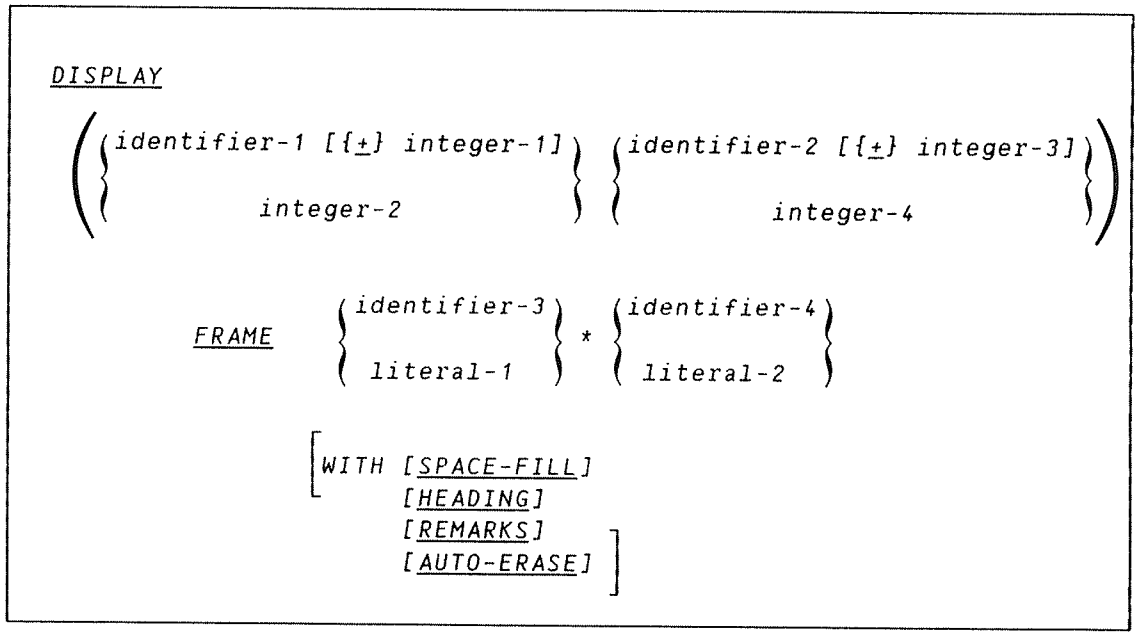
NORMAL resets the effect of a previous INVERSE-VIDEO, LOW-INTENSITY, BLINK, or UNDERLINE.

AUTO-ERASE. When the first character of a following ACCEPT statement is entered, all fields coded with AUTO-ERASE (up to 16) will disappear automatically.

PROMPT. If the field is all zeros or all spaces, the prompt character period (".") will appear in each position instead.

BLANK-WHEN-ZERO.
causes the item to be displayed as all blanks if the item's value is zero.

Format 3:



Format 3 is used to draw frames around selected areas of the screen. The part within the first parentheses is a position specification, as in the previous format. The specified position is taken to be the upper left corner of a frame of the size given after the FRAME phrase. The first number after FRAME gives the number of lines down from the specified point that the frame will reach. The second number gives the number of columns that the frame will reach to the right of the specified point.

The format has four additional options:

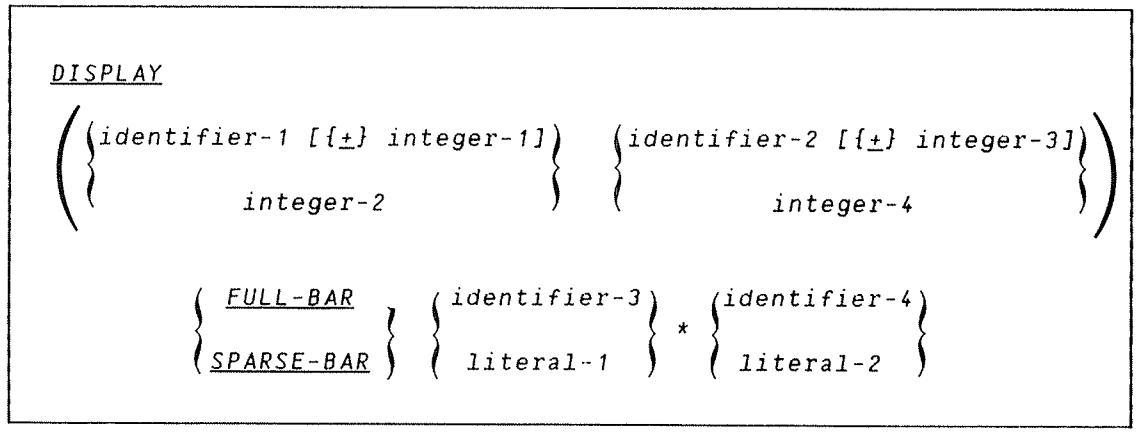
SPACE-FILL erases the interior of the frame, i.e., it writes blanks into each character position inside the frame.

HEADING makes COBOL draw a line segment across the third line inside the frame, thus making room for a headline at the second line of the frame.

REMARKS leaves room for a remark at the second line from the bottom line of the frame, with a line across the frame abovethe remark.

AUTO-ERASE erases the frame (with contents) automatically upon the following ACCEPT. A very useful DISPLAY option when using the HELP option in the ACCEPT statement.

Format 4:



Format 4 allows COBOL to draw vertical histogram bars from available data. It has a position specification part, like the previous formats. In format 4, however, the position specified inside the parentheses is the lower left corner of the bar. Select one of two different shadings:

FULL-BAR dense shading.

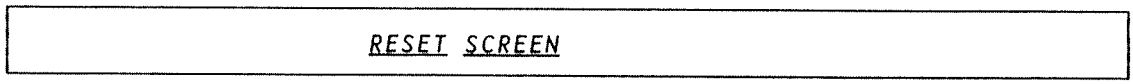
SPARSE-BAR half-tone shading.

The size of the bar must be specified after the shading option. The first of the two numbers defines the height of the bar, the second defines the width. The height of the bar may be up to four times the number of lines available for it. That means that a bar of height 4 is one line high, while a bar of height 88 may reach from the bottom to the top of a 22-line screen.

6.6.1.6 The RESET SCREEN Statement

The RESET SCREEN statement resets the terminal to the initial state.

Format:



6.6.2 Screen Handling Examples

This section shows five simple programs to illustrate some of the features of ND COBOL screen handling. A description of the statements used will be found in section 6.6.1.

EXAMPLE 1:

```
 1        IDENTIFICATION DIVISION.
 2        PROGRAM-ID. DIAGONALS.
 3       ****************************************************************
 4       * This program produces a pattern of two crossing diagonals
 5       * which appear as blanked areas on a filled-in background.
 6       ****************************************************************
 7
 8        DATA DIVISION.
 9
10        WORKING-STORAGE SECTION.
11        01  M            PIC 99   VALUE ZERO.
12        01  J            PIC 99   VALUE 78.
13        01  I            PIC 999  VALUE ZERO.
14        01  N            PIC 99   VALUE ZERO.
15
16        PROCEDURE DIVISION.
17        100.
18            BLANK SCREEN.
19        1200.
20            DO FOR N FROM 1 BY 1 TO 80.
21                DO FOR I FROM 1 BY 1 TO 25.
22                    DISPLAY (I, N) '#'.
23                END-DO.
24            END-DO.
25        1300.
26            DO FOR N FROM 2 BY 2 TO 25.
27                MOVE N TO I.
28                ADD N TO I.
29                ADD N TO I.
30                MOVE I TO M.
31                ADD 3 TO M.
32                BLANK LINE N COLUMN I TO M.
33            END-DO.
34        1400.
35            DO FOR N FROM 2 BY 2 TO 25.
36                SUBTRACT 6 FROM J.
37                MOVE J TO M.
38                ADD 3 TO M.
39                BLANK LINE N COLUMN J TO M.
40            END-DO.
41        1700.
42            STOP RUN.
```

EXAMPLE 2:

```
 1        IDENTIFICATION DIVISION.
 2        PROGRAM-ID. FORMS.
 3       ****************************************************************
 4       * This program shows how a form might be created containing
 5       * information - in this case, names, addresses and codes.
 6       * The contents are displayed and the opportunity is given
 7       * to "accept" an update for each entry. It is possible to move
 8       * between the fields using control keys as individually coded
 9       * in the program with each ACCEPT statement.
10       ****************************************************************
11
12        DATA DIVISION.
13
14        WORKING-STORAGE SECTION.
15        01  SQUARE.
16            02 HORIZ-LINE PIC X(80) VALUE '----------------------------
17       -    '----------------------------------------------------'.
18            02 NAM  PIC X(15) OCCURS 10 TIMES.
19            02 ADDR PIC X(30) OCCURS 10 TIMES.
20            02 CODE PIC 999   OCCURS 10 TIMES.
21        01  N       PIC 99    VALUE ZERO.
22        01  M       PIC 99    VALUE ZERO.
23
24        PROCEDURE DIVISION.
25
26        5.  MOVE 'ROSE COTTAGE'             TO ADDR(1).
27            MOVE '10 STRAWBERRY HILL'       TO ADDR(2).
28            MOVE 'THE OLD MILL'             TO ADDR(3).
29            MOVE '132 OXFORD ROAD'          TO ADDR(4).
30            MOVE '1 DONNINGTON SQUARE'      TO ADDR(5).
31            MOVE '5 WHITE HORSE LANE'       TO ADDR(6).
32            MOVE 'TUDOR LODGE'              TO ADDR(7).
33            MOVE '3 DEER LEAP WOOD'         TO ADDR(8).
34            MOVE 'RIVERSIDE HOUSE, HENLEY'  TO ADDR(9).
35            MOVE 'THE BARN, ABBOTS ANN'     TO ADDR(10).
36
37        15.
38            MOVE 'ANDERSON'         TO NAM(1).
39            MOVE 'ARCHER'           TO NAM(2).
40            MOVE 'BROWN'            TO NAM(3).
41            MOVE 'CARTER'           TO NAM(4).
42            MOVE 'EVANS'            TO NAM(5).
43            MOVE 'HYDE'             TO NAM(6).
44            MOVE 'LEWIS'            TO NAM(7).
45            MOVE 'NORTON'           TO NAM(8).
46            MOVE 'RUSSELL'          TO NAM(9).
47            MOVE 'WOOD'             TO NAM(10).
48
49        20.
50            MOVE '505' TO CODE(1).
51            MOVE '399' TO CODE(2).
52            MOVE '002' TO CODE(3).
53            MOVE '900' TO CODE(4).
54            MOVE '417' TO CODE(5).
55            MOVE '015' TO CODE(6).
56            MOVE '666' TO CODE(7).
57            MOVE '818' TO CODE(8).
58            MOVE '077' TO CODE(9).
59            MOVE '202' TO CODE(10).
60
61        30.
62            BLANK SCREEN.
63
64       * Insert form headers.
65
66        35.
67            DISPLAY (1, 1)  HORIZ-LINE.
68            DISPLAY (2, 1)  '~'.
69            DISPLAY (2, 10) 'NAME'.
70            DISPLAY (2, 31) '~'.
71            DISPLAY (2, 40) 'ADDRESS'.
72            DISPLAY (2, 72) '~'.
73            DISPLAY (2, 74) 'CODE'.
74            DISPLAY (2, 80) '~'.
75            DISPLAY (3, 1)  HORIZ-LINE.
```

```
* Remainder of form.

     DO FOR N FROM 4 BY 1 TO 24.
         DISPLAY (N, 1 ) '~'.
         DISPLAY (N, 31) '~'.
         DISPLAY (N, 72) '~'.
         DISPLAY (N, 80) '~'.
     END-DO.

* Loop to display contents.

     DO FOR N FROM 4 BY 1 TO 13.
         SUBTRACT 3 FROM N.
         MOVE N TO M.
         ADD 3 TO N.
         DISPLAY (N, 10) NAM(M).
         DISPLAY (N, 40) ADDR(M).
         DISPLAY (N, 74) CODE(M).
     END-DO.

* Use of ACCEPT statement to update form.

 100.
     MOVE 4 TO N.
     MOVE 1 TO M.

 101.
     ACCEPT (N, 10) NAM(M)
                    WITH UPDATE PROMPT
                         DOWN 201
                         RIGHT 102
                         LEFT 103
                         HOME 100
                         UP 301
                         EXIT 900.

 102.
     ACCEPT (N, 40) ADDR(M)
                    WITH UPDATE PROMPT
                         DOWN 202
                         RIGHT 103
                         LEFT 101
                         HOME 100
                         UP 302
                         EXIT 900.

 103.
     ACCEPT (N, 10) CODE(M)
                    WITH UPDATE PROMPT
                         DOWN 203
                         RIGHT 102
                         LEFT 101
                         HOME 100
                         UP 303
                         EXIT 900.

 201.
     PERFORM 500.
     GO TO 101.

 202.
     PERFORM 500.
     GO TO 102.

 203.
     PERFORM 500.
     GO TO 103

 301.
     PERFORM 600.
     GO TO 101

 302.
     PERFORM 600.
     GO TO 102.
```

```
153         303.
154             PERFORM 600.
155             GO TO 103.
156
157         500.
158             ADD 1 TO N.
159             IF N IS GREATER THAN 13 THEN
160                 SUBTRACT 1 FROM N
161                 ELSE ADD 1 TO M
162             END-IF.
163
164         600.
165             SUBTRACT 1 FROM N.
166             IF N IS LESS THAN 4 THEN
167                 ADD 1 TO N
168                 ELSE SUBTRACT 1 FROM M
169             END-IF.
170
171         900.
172             STOP RUN.
```

Example 3:

```
 1        IDENTIFICATION DIVISION.
 2        PROGRAM-ID.         SCREEN-PLAY.
 3       ********************************************************
 4       * This program illustrates a few of the various ways of
 5       * visually displaying fields which the user wants to
 6       * update. Specific fields are accessed by use of control
 7       * keys. CR moves the cursor from field to field in the
 8       * order they are displayed. The screen is first filled
 9       * with background characters.
10       ********************************************************
11        DATA DIVISION.
12
13        WORKING-STORAGE SECTION.
14        77  LIN      PIC 99.
15        77  POS      PIC 99.
16        01  N        PIC XX VALUE 'ND'.
17        01  N1       PIC X(9) VALUE 'NORWAY   '.
18        01  N2       PIC X(9) VALUE '.........'.
19        01  N3       PIC 9(9) VALUE ZERO.
20        01  N4       PIC S9(3) COMP VALUE 0.
21
22        PROCEDURE DIVISION.
23        100.
24            BLANK SCREEN.
25        500.
26            PERFORM DISP
27                VARYING LIN FROM 24 BY -1 UNTIL LIN < 1
28                AFTER POS FROM 1 BY 2 UNTIL POS > 80.
29        DISP.
30            DISPLAY (LIN, POS) 'ND'.
31        1500.
32            ACCEPT (1, 1) N WITH UPDATE BEEP.
33        1700.
34            BLANK LINE 3 TO 9.
35        1800.
36            DISPLAY (5, 20) 'COUNTRY: ' WITH UNDERLINE.
37            ACCEPT (5, 30) N1 WITH UPDATE
38                               UP 1500
39                               EXIT 6000
40                               HOME 5000
41                               DOWN 3000.
42        2000.
43            BLANK LINE 12 COLUMN 8 TO 45.
44        3000.
45            DISPLAY (12, 10) 'MONTH: ' WITH INVERSE-VIDEO.
46            ACCEPT (12, 20) N2 WITH UPDATE
47                               DOWN 3050
48                               EXIT 1500
49                               HOME 1800
50                               UP 3000.
51        3050.
52            BLANK LINE 16 COLUMN 35 TO 55.
53        4000.
54            DISPLAY (16, 30) 'SALES: ' WITH BLINK.
55            ACCEPT (16, 40) N3 WITH UPDATE
56                               UP 2000
57                               HOME 3000
58                               DOWN 4050
59                               EXIT 6000.
60        4050.
61            BLANK LINE 21 COLUMN 20 TO 50.
62            BLANK LINE 22 COLUMN 20 TO 50.
63            BLANK LINE 23 COLUMN 20 TO 50.
64        5000.
65            DISPLAY (22, 24) '% CHANGE (+/-): ' WITH LOW-INTENSITY
66                                                    UNDERLINE.
67            ACCEPT (22, 42) N4 WITH UPDATE
68                               UP 4000
69                               DOWN 6000
70                               HOME 3000
71                               EXIT 2000.
72        6000.
73            STOP RUN.
```

Example 5:

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. VIDEO.
*****************************************************************
* This program interrogates an existing file which
* contains information on a video-film library. The choice is
* of viewing either: a list of all films in the same category,
* details of a film in any category, or the whole file in
* alphabetic sequence. (The whole file can also be printed
* out.) Only the more relevant parts of the program are shown.
*****************************************************************
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT VIDEO-FILE
         ASSIGN "VIDEO:DATA"
         ORGANIZATION INDEXED
         ACCESS DYNAMIC
         RECORD KEY CODE-NO
         ALTERNATE RECORD KEY CATEG WITH DUPLICATES
         ALTERNATE RECORD KEY TITLE WITH DUPLICATES
         STATUS V-STATUS.
     SELECT PRINT-FILE
         ASSIGN "L-P"
         STATUS V-STATUS.

 DATA DIVISION.
 FILE SECTION.
 FD VIDEO-FILE
 01  VIDEO-REC.
     03 CODE-NO      PIC X(5).
     03 CATEG        PIC X(7).
     03 TITLE        PIC X(35).
     03 STARS        PIC X(20).
     03 RENT         PIC 9V99.
     03 IN-STOCK     PIC X.
     03 DATE-OUT     PIC X(6).
     03 DATE-BACK    PIC X(6).
     03 INCOME       PIC 999V99.
     03             PIC X(30).

 WORKING-STORAGE SECTION.
 77  V-STATUS    PIC XX.
 77  REC-COUNT   PIC 9(4).
 77  LINE-COUNT  PIC 99.
 77  OPTION      PIC X.
 77  REPLY       PIC X.
 77  CATEGORY    PIC X(7).
 77  NAME        PIC X(35).

 PROCEDURE DIVISION.

 BEGIN.
     OPEN I-O VIDEO-FILE WITH MULTI-USER-MODE.
     OPEN OUTPUT PRINT-FILE.

* Select an option.

 CHOOSE.
     BLANK SCREEN.
*         :.............. This statement disables the ESC key.
     DISPLAY (5, 20) 'VIDEO LIBRARY INFORMATION PROGRAM'
                                                  WITH UNDERLINE.
     DISPLAY (8, 24) 'OPTIONS ARE:'
                                                  WITH AUTO-ERASE
     DISPLAY (10, 22) 'DISPLAY FILE BY CATEGORY    - 1'
                                                  WITH AUTO-ERASE
     DISPLAY (12, 22) 'DISPLAY RECORD BY TITLE     - 2'
                                                  WITH AUTO-ERASE
     DISPLAY (14, 22) 'DISPLAY FILE ALPHABETICALLY - 3'
                                                  WITH AUTO-ERASE
     DISPLAY (16, 22) 'PRINT FILE ALPHABETICALLY   - 4'
                                                  WITH AUTO-ERASE
     DISPLAY (18, 22) 'EXIT FROM THE PROGRAM       - 5'
                                                  WITH AUTO-ERASE
```

```
    DISPLAY (20, 22) 'PLEASE ENTER YOUR OPTION: '
                                                    WITH AUTO-ERASE
    ACCEPT (20, 49)  OPTION WITH MUST.
    IF OPTION IS LESS THAN 1 OR GREATER THAN 5 GO TO OPTION-ERROR
    GO TO ONE, TWO, THREE, FOUR, FIVE, DEPENDING ON OPTION.

OPTION-ERROR.
    DISPLAY (20, 22) 'INVALID OPTION, PRESS CR TO CONTINUE'
                                                    WITH BEEP.
    ACCEPT (20, 61) REPLY.
    GO TO CHOOSE.

* List all films in a cathegory.

ONE.
    BLANK SCREEN.
    DISPLAY (4, 13) 'DISPLAY THE CONTENTS OF ONE CATHEGORY'
                                                    WITH UNDERLINE.
    DISPLAY (6, 13) 'ENTER REQUIRED CATHEGORY: '.
    ACCEPT (6, 43) CATHEGORY WITH PROMPT
                                BLINK
                                CONTROL VALID.
* If an invalid category has been entered, the above ACCEPT
* will not have been "accepted". A known cathegory must be
* re-submitted.

    MOVE CATHEGORY TO CATEG.
    START VIEOFILE KEY IS EQUAL TO CATEG
        INVALID KEY DISPLAY (10, 20) 'ISAM FILE ERROR'
        WITH BEEP
        DISPLAY (10,36) V-STATUS

* CR to try again

        ACCEPT (10, 41)  REPLY
        GO TO CHOOSE.
    PERFORM HEADER.
    GO TO ONE-NEXT.

VALID.
    IF CATHEGORY IS NOT EQUAL TO
                        'HORROR'  OR
                        'WESTERN' OR
                        'DRAMA'   OR
                        'ROMANCE' OR
                        'SCI-FI'  OR
                        'CRIME'   ACCEPT-ERROR.
ONE-NEXT.
    READ VIDEO-FILE NEXT RECORD
        AT END ACCEPT (LIN-COUNT, 80) REPLY
        GO TO CHOOSE.
    IF CATEG IS NOT EQUAL TO CATHEGORY
        ACCEPT (LINE-COUNT, 80) REPLY
        GO TO CHOOSE.
    DISPLAY (LINE-COUNT, 2)  CODE-NO.
    DISPLAY (LINE-COUNT, 9)  TITLE.
    DISPLAY (LINE-COUNT, 47) CATEG.
    DISPLAY (LINE-COUNT, 53) STARS.
    DISPLAY (LINE-COUNT, 76) RENT.
    ADD 1 TO LINE-COUNT.
    IF LINE-COUNT IS GRETER THAN 24
        ACCEPT (24,80) REPLY
        PERFORM HEADER.
    GO TO ONE-NEXT.

HEADER.
    BLANK SCREEN.
    DISPLAY (1, 28) 'VIDEO LIBRARY CATALOGUE'.
    MOVE 4 TO LINE-COUNT.
    DISPLAY (LINE-COUNT, 2)  CODE-NO.
    DISPLAY (LINE-COUNT, 9)  TITLE.
    DISPLAY (LINE-COUNT, 47) CATEG.
    DISPLAY (LINE-COUNT, 53) STARS.
    DISPLAY (LINE-COUNT, 76) RENT.

TWO.
```

```
* Display the details for one title.

     BLANK SCREEN
     DISPLAY (5, 20) 'SEARCH FOR RECORD BY TITLE'
                                     WITH UNDERLINE.
     DISPLAY (8, 10) 'ENTER REQUIRED TITLE: '
                                     WITH INVERSE-VIDEO.
     ACCEPT (8,34) NAME WITH PROMPT CONTROL ALPHA.

* The name of the video-film is checked for non-alphabetic
* characters. If any are found, the above ACCEPT will not be
* taken and must be re-entered.

     MOVE NAME TO TITLE.
     START VIDEO-FILE KEY IS NOT LESS THAN TITLE
         INVALID KEY
         DISPLAY (10, 10) 'TITLE NOT IN LIBRARY' WITH BEEP
         DISPLAY (11, 10) 'PRESS CR'
         ACCEPT (11, 19) REPLY
         GO TO TWO.
     PERFORM HEADER.

 TWO-NEXT.
     READ VIDEO-FILE NEXT RECORD
         AT END DISPLAY (LINE-COUNT, 35) 'END OF FILE'
                                     WITH BEEP
         ACCEPT (LINE-COUNT, 47) REPLY
         GO TO CHOOSE.

     DISPLAY (LINE-COUNT, 2)  CODE-NO.
     DISPLAY (LINE-COUNT, 9)  TITLE.
     DISPLAY (LINE-COUNT, 47) CATEG.
     DISPLAY (LINE-COUNT, 53) STARS.
     DISPLAY (LINE-COUNT, 76) RENT.
     ADD 1 TO LINE-COUNT.

     DISPLAY (LINE-COUNT, 35) 'CORRECT RECORD (Y/N)?'.
     ACCEPT (LINE-COUNT, 55) REPLY.
     IF REPLY EQUAL TO 'Y' GO TO CHOOSE.
     BLANK LINE LINE-COUNT
         IF LINE-COUNT IS GREATER THAN 23
         PERFORM HEADER.
     GO TO TWO-NEXT.

 ALPHA.
     IS NAME IS NOT ALPHABETIC
         ACCEPT-ERROR.

 THREE.

* Display the file in alphabetic order of title.

     BLANK SCREEN.
     MOVE LOW-VALUES TO TITLE.
     START VIDEO-FILE KEY IS GREATER THAN TITLE
         INVALID KEY
         DISPLAY (2, 13) 'ISAM FILE ERROR' WITH BEEP
         DISPLAY (2, 38) V-STATUS
         ACCEPT (2,35) REPLY
         GO TO CHOOSE
     PERFORM HEADER.

 THREE-NEXT.
     READ VIDEO-FILE NEXT RECORD
         AT END ACCEPT (LINE-COUNT, 80) REPLY
         GO TO CHOOSE.
     DISPLAY (LINE-COUNT, 2)  CODE-NO.
     DISPLAY (LINE-COUNT, 9)  TITLE.
     DISPLAY (LINE-COUNT, 47) CATEG.
     DISPLAY (LINE-COUNT, 53) STARS.
     DISPLAY (LINE-COUNT, 76) RENT.
     ADD 1 TO LINE-COUNT.
     IF LINE-COUNT IS GREATER THAN 24
         ACCEPT (24, 80) REPLY
         PERFORM HEADER.
     GO TO THREE-NEXT.
```

```
 FOUR.

* Print the full catalogue alphabetically.

     BLANK SCREEN.
     DISPLAY (5, 25) 'PRINTING FULL ALPHABETIC CATALOGUE.'
                                              WITH UNDERLINE.
     MOVE LOW-VALUES TO TITLE.
     MOVE ZERO TO REC-COUNT.
     START VIDEO-FILE KEY IS GREATER THAN TITLE
         INVALID KEY
         DISPLAY (7, 25) 'ISAM FILE ERROR' WITH BEEP
         DISPLAY (7, 42) V-STATUS
         ACCEPT (7, 50) REPLY
         GO TO CHOOSE.
           .
           .

* Create the print-header
           .
           .

 FOUR-NEXT.
     READ VIDEO-FILE NEXT RECORD
         AT END
         DISPLAY (12, 22) PROCESSING IS NOW COMPLETE'
                                     WITH BEEP
         ACCEPT (12,49) REPLY
         GO TO CHOOSE.
           .
           .

* Create the print record.
           .
           .

     IF LINE-COUNT IS GREATER THAN 60
         PERFORM PRINT-HEADER
     GO TO FOUR-NEXT.

 FIVE.

* Exit from program

     BLANK SCREEN.
     CLOSE VIDEO-FILE.
     CLOSE PRINT-FILE.
     DISPLAY (12, 20) '>>> Returning to main menu >>>'
                                     WITH UNDERLINE.
     STOP RUN.
```

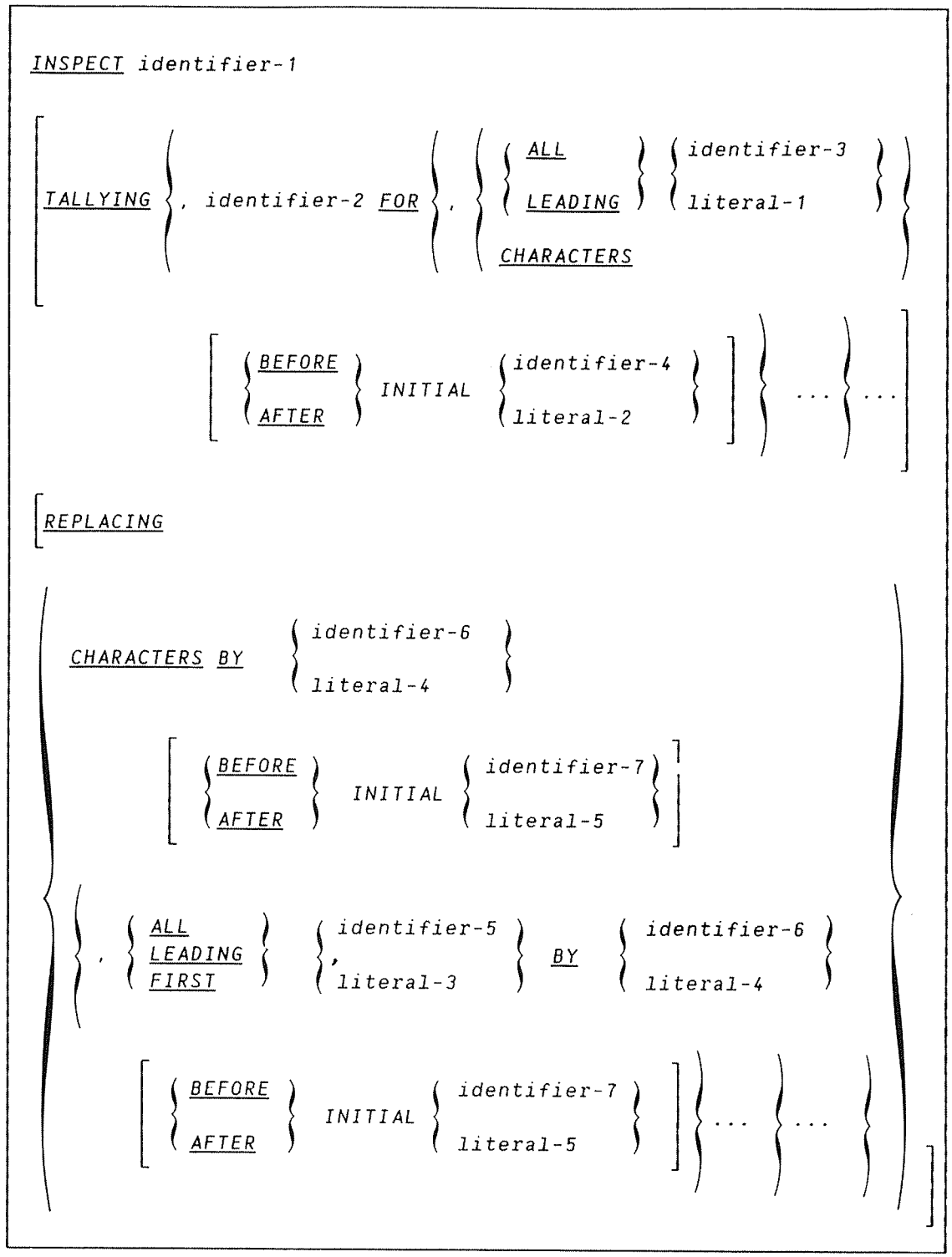
Example 6:

```
 1       IDENTIFICATION DIVISION.
 2       PROGRAM-ID. X-001.
 3      ****************************************************************
 4      * This program demonstrates the facilities for framing selected
 5      * parts of the screen, and for writing histogram bars.
 6      ****************************************************************
 7       DATA DIVISION.
 8       WORKING-STORAGE SECTION.
 9       01  NAME    PIC X(30).
10       01  ANSWER  PIC X.
11       01  I       COMP.
12       01  J       COMP.
13       01  K       COMP.
14       01  X       COMP.
15       01  Y       COMP.
16
17       PROCEDURE DIVISION.
18
19       500.
20           COMPUTE X=19.
21           COMPUTE Y=9.
22
23       1000.
24           BLANK SCREEN.
25           DISPLAY (10, 1) 'Your name:'.
26           ACCEPT (10, 12) NAME WITH PROMPT.
27           BLANK LINE 10.
28           DISPLAY (1, 1) FRAME 18 * 75 WITH HEADING.
29           DISPLAY (2, 28) 'My name is:'.
30           DO FOR I FROM 4 TO 17
31               DISPLAY (I, 3) NAME WITH BLINK
32               DISPLAY (I, 42) NAME WITH UNDERLINE
33           END-DO.
34
35       1500.
36           BLANK LINE 22.
37           DISPLAY (22, 1) 'Continue execution?' WITH UNDERLINE.
38           ACCEPT (22, 20) ANSWER WITH PROMPT.
39           IF ANSWER EQUAL 'N' OR 'n' THEN PERFORM 3000.
40           DISPLAY (Y, X) FRAME 12 * 34 WITH SPACE-FILL.
41           DO FOR I FROM Y + 1 TO Y + 10
42               DISPLAY (I, X + 2) NAME WITH INVERSE-VIDEO.
43           END-DO.
44
45       2000.
46           BLANK LINE 22.
47           DISPLAY (22, 1) 'Continue execution?' WITH UNDERLINE.
48           ACCEPT (22, 20) ANSWER WITH PROMPT.
49           IF ANSWER EQUAL 'N' OR 'n' THEN PERFORM 3000.
50           BLANK SCREEN.
51           DISPLAY (1, 1) FRAME 20 * 73.
52           DO FOR I FROM 2 BY 3 TO 71
53               COMPUTE J = I
54               DISPLAY (19, I) FULL-BAR J * 1
55               COMPUTE J = 72 - I
56               COMPUTE K = I + 1
57               DISPLAY (19, K) SPARSE-BAR J * 1
58           END-DO.
59
60       2500.
61           COMPUTE X = 5.
62           COMPUTE Y = 3.
63           PERFORM 1500.
64
65       3000.
66           BLANK LINE 22.
67           DISPLAY (22, 11)
68               'You have now used the ND COBOL Screen Handling'
69               WITH UNDERLINE.
70           STOP RUN.
```

6.6.3 The INSPECT Statement

The INSPECT statement specifies that characters in a data item are to be counted, *or* replaced, *or* counted and replaced.

Format:



Identifier-1, the inspected item, must either be a group item or any category of elementary item with USAGE DISPLAY.

Identifier-2, the count field, must be an elementary integer data item.

All literals must be nonnumeric and any figurative constant except ALL. (If a figurative constant is used as literal-3, then the size of identifier-6 and -7 must be one character in length.)

When the CHARACTERS phrase is used, literal-4 and -5 or identifier-6 and -7 must be one character in length.

General Rules:

1) Either the TALLYING or REPLACING option must be given. Both may appear, but in this case all tallying occurs before any replacement is made.

2) All identifiers (except identifier-2) are treated by the INSPECT statement according to its category:

   a) If alphabetic or alphanumeric - as a character-string.

   b) If alphanumeric edited, numeric edited or unsigned numeric - as though redefined as alphanumeric, and the INSPECT statement refers to the alphanumeric item.

   c) If signed numeric - as though moved to an unsigned numeric data item of the same length and then treated as in rule b above.

3) Inspection includes the comparison cycle, the establishment of boundaries for the BEFORE and AFTER phrase, and the mechanisms for tallying and/or replacing. It begins at the leftmost character position of the data item identifier-1 and proceeds to the rightmost character position as described in the remaining general rules.

4) The rules for comparison are:

   a) The first TALLYING/REPLACING operand is compared with an equal number of the leftmost contiguous characters in the inspected item. A match occurs only if both are equal character-for-character.

   b) If no match occurs, the comparison is repeated for each successive TALLYING/REPLACING operand until either a match is found or all the operands have been compared;

   c) If a match is found, then tallying/replacing occurs according to the following TALLYING/REPLACING option descriptions. The first character of the inspected item following the rightmost matching character is now the subject of the operations described in rules a and b above.

d) If no match is found, then the first character following the leftmost inspected character in the inspected item becomes the leftmost character position and processes of a and b above are repeated. The steps a to d, the comparison cycle, are repeated until the rightmost character has participated in a match or has been considered as the leftmost character position.

5) If the BEFORE/AFTER option is used, then the previous rules are modified as described in the following TALLYING/REPLACING option descriptions.

TALLYING OPTION

Identifier-2 (an elementary integer item) is the *count field*. Identifier-3 or literal-1 is the *tallying field*.

If the BEFORE/AFTER option is not specified, then the following actions occur on execution of INSPECT with TALLYING:

a) If ALL is used, the count field is increased by 1 for each non-overlapping occurrence of the tallying field.

b) If LEADING is specified, the count field is increased by 1 for each contiguous non-overlapping occurrence of this tallying field in the inspected item, provided that the leftmost such occurrence is at the point where comparison began in the first comparison cycle which this tallying field took part in.

c) If CHARACTERS is specified, the count field is increased by 1 for each character position in the inspected item.

REPLACING OPTION

Identifier-5 or literal-3 is the *subject field*, and identifier-6 or literal-4 is the *substitution field*. These two fields must be the same length and the following rules apply:

1) When the subject and substitution fields are character strings, each non-overlapping occurrence of the subject field in the inspected item is replaced by the character-string specified in the substitution field.

2) After replacement has occurred in any character position of the inspected item, no further replacement for that position is made during this INSPECT statement execution. When the BEFORE/AFTER option is not given, the following actions take place on execution of INSPECT with REPLACING:

a) If CHARACTERS is specified, the substitution field must be one character in length. Each character in the inspected item is replaced by the substitution field, beginning at the leftmost character and continuing to the rightmost.

b) If ALL is specified, each non-overlapping occurrence of the subject field in the inspected item is replaced by the substitution field, beginning at the leftmost character and continuing to the rightmost.

c) If LEADING is specified, each contiguous non-overlapping occurrence of the subject field of the inspected item is replaced by the substitution field, provided that the leftmost occurrence is at the point where comparison began in the first comparison cycle in which this substitution field can participate.

d) If FIRST is specified, the leftmost occurrence of the subject field in the inspected item is replaced by the substitution field.

BEFORE/AFTER OPTIONS

When these are specified, the above rules for counting and replacing are modified thus:

Identifiers 4 and 7 and literals 2 and 5 are *delimiters* and are themselves not counted or replaced.

In the REPLACING option, if CHARACTERS is specified then the delimiter must be one character in length.

When BEFORE is used, counting and/or replacement of the inspected it begins at the leftmost character and continues until the first occurrences of the delimiter are encountered. If no delimiter occurs in the inspected item, counting and/or replacement continues to the rightmost character.

When AFTER is present, counting and/or replacement of the inspected item begins with the first character to the right of the delimiter and continues to the rightmost character in the inspected item. If no delimiter exists in the inspected item no counting/replacement takes place.

Six examples of the INSPECT statement follow:

(*Note:* identifier-2, the count field, must be initialized before execution of the INSPECT statement.)

EXAMPLE 1.

```
INSPECT word TALLYING count FOR LEADING "L" BEFORE INITIAL "A",
    count-1 FOR LEADING "A" BEFORE INITIAL "L".

Where word = LARGE, count = 1, count-1 = 0.
Where word = ANALYST, count = 0, count-1 = 1.
```

EXAMPLE 2.

```
INSPECT word TALLYING count FOR ALL "L", REPLACING LEADING "A"
    BY "E" AFTER INITIAL "L".

Where word = CALLAR, count = 2, word = CALLAR.
Where word = SALAMI, count = 1, word = SALEMI.
Where word = LATTER, count = 1, word = LETTER.
```

EXAMPLE 3.

```
INSPECT word REPLACING ALL "A" BY "G" BEFORE INITIAL "X".

Where word = ARXAX, word = GRXAX.
Where word = HANDAX, word = HGNDGX.
```

EXAMPLE 4.

```
INSPECT word TALLYING count FOR CHARACTERS AFTER INITIAL "J"
    REPLACING ALL "A" BY "B".

Where word = ADJECTIVE, count = 6, word = BDJECTIVE.
Where word = JACK, count = 3, word = JBCK.
Where word = JUJMAB, count = 5, word = JUJMBB.
```

EXAMPLE 5.

```
INSPECT word REPLACING ALL "X" BY "Y", "B" BY "Z", "W"
    BY "Q" AFTER INITIAL "R".

Where word = RXXBQWY, word = RYYZQQY.
Where word = YZACDWBR, word = YZACDWZR.
Where word = RAWRXEB, word = RAQRYEZ.
```

EXAMPLE 6.

```
INSPECT word REPLACING CHARACTERS BY "B" BEFORE INITIAL "A".

word before: 1 2   X Z A B C D
word after:  B B B B B A B C D
```

6.6.4 The MOVE Statement

The MOVE statement transfers data to one or more data areas in accordance with the editing rules.

Format 1:

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

Format 2:

```
MOVE { CORRESPONDING } identifier-1 TO identifier-2
     { CORR          }
```

In this format, identifier-1 and literal represent the sending area; identifier-2, identifier-3, ..., represent the receiving area.

When format 2 is specified, both identifiers must be group items. When CORRESPONDING is used, selected items in identifier-1 are moved to identifier-2 according to the rules given for the CORRESPONDING option, which are given in the next paragraphs. The results are the same as if each pair of corresponding identifiers had been referred to in a separate MOVE statement.

For the purpose of this discussion, identifier-1 and identifier-2 must each be identifiers that refer to group items. A pair of data items, one from identifier-1 and one from identifier-2 *correspond* if the following conditions exist:

1) A data item in identifier-1 and a data item in identifier-2 are not designated by the key word FILLER and have the same data-name and the same qualifiers up to, but not including, identifier-1 and identifier-2.

2) At least one of the data items is an elementary data item in the case of a MOVE statement with the CORRESPONDING phrase.

3) The description of identifier-1 and identifier-2 must not contain level-number 77 or 88 or the USAGE IS INDEX clause.

4) A data item that is subordinate to identifier-1 and identifier-2 and contains a REDEFINES, RENAMES, OCCURS or USAGE IS INDEX clause is ignored, as well as those data items subordinate to the data item that contains the REDEFINES, OCCURS or USAGE IS INDEX clause. However, identifier-1 and identifier-2 may have REDEFINES or OCCURS clauses or be subordinate to data items with REDEFINES or OCCURS clauses.

CORR is an abbreviation for CORRESPONDING.

General Rules:

1) The data in the sending area is moved into the first receiving area (identifier-2), then into the second receiving area (identifier-3) etc. Any subscripting or indexing associated with the sending item is evaluated immediately before the data is moved to the first receiving field. Similarly, any subscripting or indexing associated with receiving items is evaluated immediately before the data is moved in.

2) The result of the statement:

   MOVE a (b) TO b, c (b)

   is equivalent to:

   MOVE a (b) TO temp

   MOVE temp TO b

   MOVE temp TO c (b)

   where temp has been defined as an intermediate result.

3) Any MOVE in which the sending and receiving items are both elementary items is an elementary move. Every elementary item belongs to one of the following categories: numeric, alphabetic, alphanumeric, numeric edited, alphanumeric edited. These categories are described in the PICTURE Clause. Numeric literals belong to the category numeric, and nonnumeric literals belong to

the category alphanumeric. The figurative constant ZERO belongs to the category numeric. The figurative constant SPACE belongs to the category alphabetic. All other figurative constants belong to the category alphanumeric.

The following rules apply to an elementary move between these categories:

a) The figurative constant SPACE, an alphanumeric edited, or alphabetic data item must not be moved to a numeric or numeric edited data item.

b) A numeric literal, the figurative constant ZERO, a numeric data item or a numeric edited data item must not be moved to an alphabetic data item.

c) A non-integer numeric literal or a non-integer numeric data item must not be moved to an alphanumeric or alphanumeric edited data item.

d) All other elementary moves are legal and are performed according to the rules given in general rule 4.

e) A numeric edited item must not be moved to another numeric edited item.

f) (An ND-Extension.) A numeric edited item may be moved to a numeric item which is either integer or non-integer. This is equivalent to "de-editing".

4) Any necessary conversion of data from one form of internal representation to another takes place during legal elementary moves, along with any editing specified for the receiving data item:

a) When an alphanumeric edited or alphanumeric item is a receiving item, alignment and any necessary space filling takes place as defined under Standard Alignment Rules in the 'Working-Storage' Section of the Data Division. If the size of the sending item is greater than the size of the receiving item, the excess characters are truncated on the right after the receiving item is filled. If the sending item is described as being signed numeric, the operational sign will not be moved; if the operational sign occupies a separate character position (see the SIGN Clause), that character will not be moved and the size of the sending item will be considered to be one less than its actual size.

b) When a numeric or numeric edited item is the receiving item, alignment by decimal point and any necessary zero-filling takes place as defined under the Standard Alignment Rules (except where zeros are replaced because of editing requirements).

1) When a signed numeric item is the receiving item, the sign of the sending item is placed in the receiving item. (See the SIGN Clause). Conversion of the respresentation of the sign takes place as necessary. If the sending item is unsigned, a positive sign is generated for the receiving item.

2) When an unsigned numeric item is the receiving item, the absolute value of the sending item is moved and no operational sign is generated for the receiving item.

3) When a data item described as alphanumeric is the sending item, data is moved as if the sending item were described as an unsigned numeric integer.

4) When a receiving field is described as alphabetic, justification and any necessary space-filling takes place as defined under the Standard Alignment Rules. If the size of the sending item is greater than the size of the receiving item, the excess characters are truncated on the right after the receiving item is filled.

Data in the following chart summarizes the legality of the various types of MOVE statements. The references are to the rule that prohibits the move or the behaviour of a legal move.

CATEGORY OF SENDING DATA ITEM		CATEGORY OF RECEIVING DATA ITEM			
		ALPHABETIC	ALPHANUMERIC EDITED ALPHANUMERIC	NUMERIC INTEGER NUMERIC NON-INTEGER	NUMERIC EDITED
ALPHABETIC		Yes/4c	Yes4/a	No/3a	No/3a
ALPHANUMERIC		Yes/4c	Yes/4a	Yes/4b	Yes/4b
ALPHANUMERIC EDITED		Yes/4c	Yes/4a	No/3a	No/3a
NUMERIC	INTEGER	No/3b	Yes/4a	Yes/4b	Yes/4b
	NON-INTEGER	No/3b	No/3c	Yes/4b	Yes/4b
NUMERIC EDITED		No/3b	Yes/4a	Yes/3f	No/3e

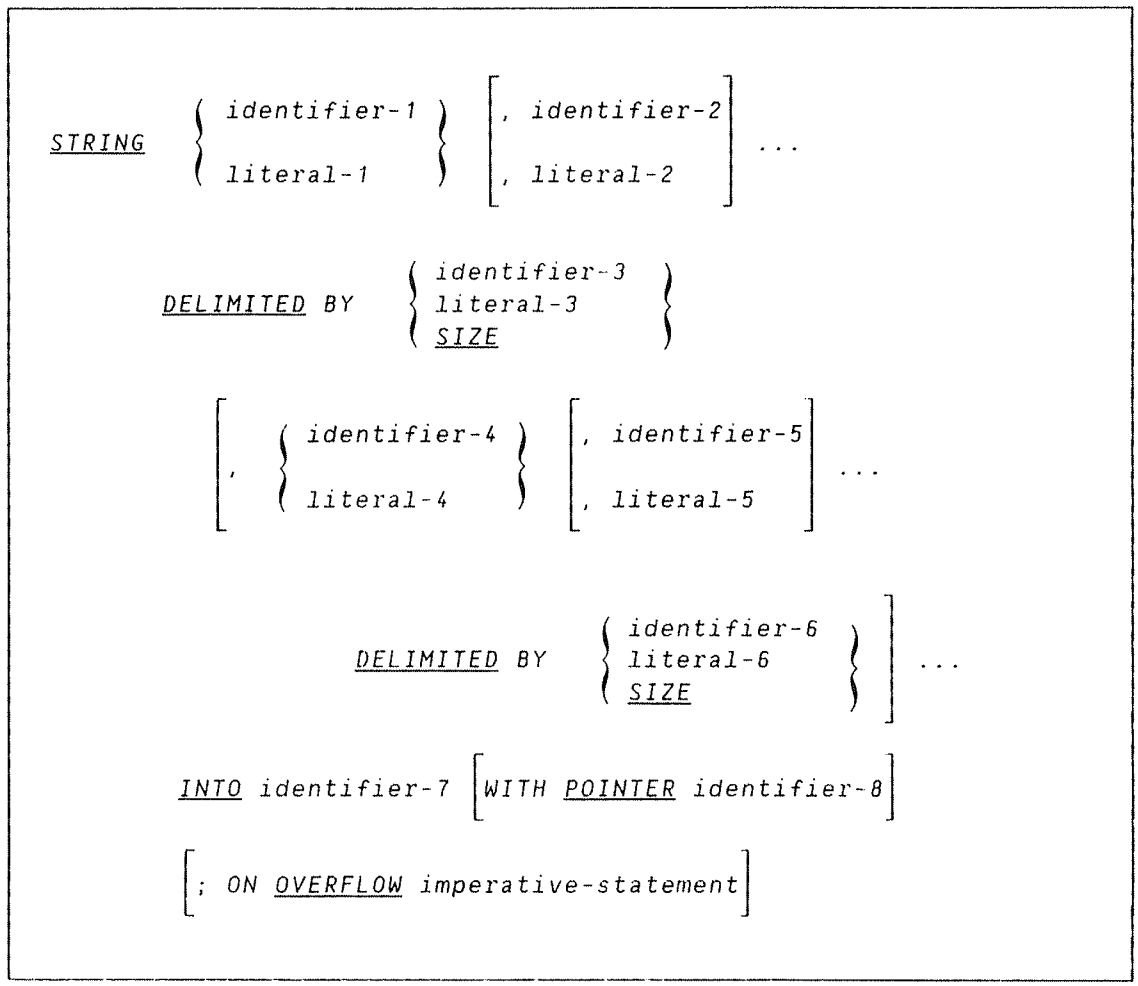
The characters /4c etc. refer to points and subpoints in the text.

c) Any move that is not an elementary move is treated exactly as if it were an alphanumeric to alphanumeric elementary move, except that there is no conversion of data from one form of internal respresentation to another. In such a move, the receiving area will be filled without consideration for the individual elementary or group items.

6.6.5 The STRING Statement

The STRING statement enables the programmer to concatenate the complete or partial contents of two or more data items into a single data item.

Format:



Each literal must be nonnumeric or any figurative constant except ALL.

The *sending fields* are given by identifier-7 which must represent an elementary alphanumeric data item.

The *pointer* field is identifier-8 which must represent an elementary numeric integer data item large enough to contain a value equal to the size plus one of the fields referenced by identifier-7. If no POINTER phrase exists, the default value of the logical pointer is 1.

The *delimiters* are identifiers 3 and 6 or their corresponding literals.

When DELIMITED BY is specified, the contents of each sending field is transferred character-by-character until the rightmost character has been sent, or a delimiter for the sending field is reached.

All identifiers (except identifier-8) must have USAGE DISPLAY.

When the STRING statement is executed, the transfer of data is governed by the following rules:

1) Characters from the sending field are transferred to the sending field according to the rules for an alphanumeric to alphanumeric move, except that no space filling is provided.

2) If DELIMITED BY SIZE is specified, each sending field is moved in its entirety to the receiving field.

3) If DELIMITED is specified without SIZE then the contents of each sending item is transferred character-by-character, starting with the leftmost one and continuing until the end of the data, or its delimiter is reached. (The delimiter itself is not transferred.)

4) If the POINTER option appears, the pointer field is explicitly available to the programmer. If this option does not appear, it is as if the user had specified identifier-8 with an initial value of 1.

5) When characters are transferred to the receiving field, the moves behave as though these characters were moved one at a time with the pointer field being incremented by one after each character is positioned. The value in the pointer cannot be changed in any other way. When processing is complete, this value will be one character position greater than that of the last character transferred.

6) If this pointer value, at or after initiation of the STRING statement execution, becomes less than one or greater than the length of the receiving field, data transfer ceases. ON OVERFLOW, if specified, is now raised.

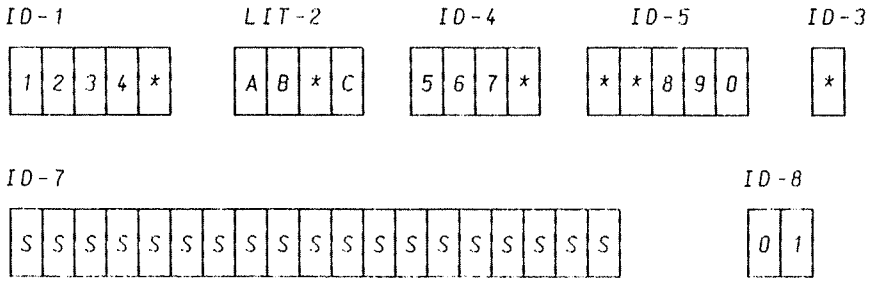
7) If ON OVERFLOW has not been specified, then, when the above conditions are encountered, control passes to the next executable statement.

Example:

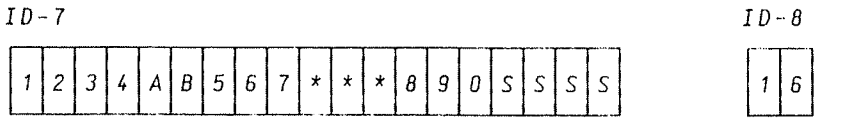
If the following STRING statement is coded:

```
STRING ID-1 LIT-2 DELIMITED BY ID-3
    ID-4 ID-5 DELIMITED BY SIZE INTO
    ID-7 WITH POINTER ID-8.
```

and at execution time the fields contain:



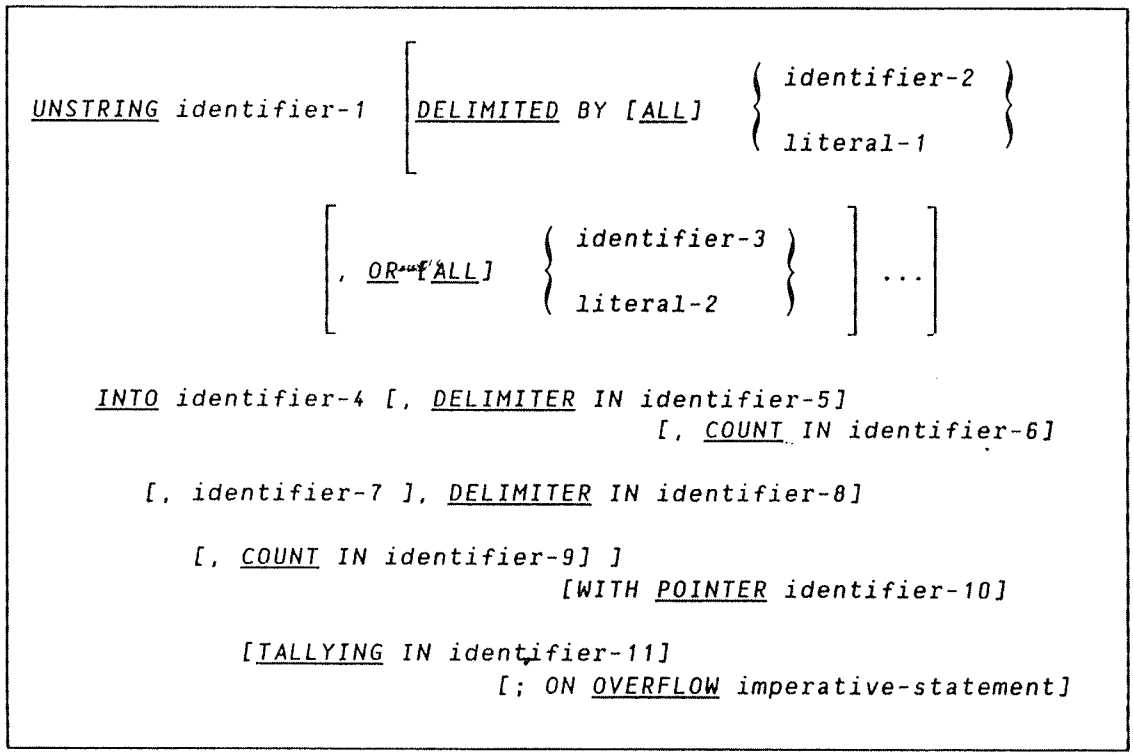
Then after execution the receiving field and the pointer field will appear as:



6.6.6 The UNSTRING Statement

The UNSTRING statement causes contiguous data in a single sending field to be separated and placed into multiple receiving fields.

Format:



The DELIMITER IN option and the COUNT IN option may only appear if the DELIMITED BY option is also present.

Each literal must be a nonnumeric literal. Each may be any figurative constant without the word ALL.

Identifier-1 (the sending field) must be an alphanumeric data item.

Identifier-6 and identifier-9 must be type computational.

The DELIMITED BY option specifies the delimiters which control the amount of data (transferred from the sending field).

The delimiters are identifiers 2 and 3, or their corresponding literals, and each of them (representing one delimiter) must be an alphanumeric data item. The maximum number of delimiters is 15.

If a delimiter contains two or more characters it will act as a delimiter only if the delimiter characters appear contiguously in the sending field, and in the sequence specified.

When two or more delimiters are specified in the DELIMITED BY option, an 'OR' condition exists between them. Each non-overlapping occurrence of any of the delimiters in the sending field in its specified sequence, is considered to be a match.

DELIMITED BY ALL results in one occurrence, or two or more contiguous occurrences, of any delimiter being treated as if they were only one occurrence; this occurrence is moved to the delimiter receiving field (if any), (identifiers 5, 8, ...).

If DELIMITED BY ALL is not specified, and two or more occurrences of any delimiter are found, then the current receiving field is filled with either space or zero according to the description of this field.

When the UNSTRING statement is initiated, identifier-4 is the current receiving field. Receiving fields must have USAGE DISPLAY and must be one of the types:

- alphabetic
- alphanumeric (not edited)
- numeric (not edited)

Data is transferred from the sending field according to the following rules:

If the POINTER option appears, it contains a value indicating a relative position in the sending field (it must be initialized prior to statement execution).

DELIMITED BY causes the examination to proceed from left to right, character-by-character, until a delimiter is encountered. If no delimiter is found, the examination ends with the last character in the sending field.

If the DELIMITED BY option does not appear, the number of characters examined will be equal to the size of the current receiving field. (If the sign of the receiving field has been defined as occupying a separate character position, then the number of characters examined is one less than the size of the field.)

The characters thus examined (excluding delimiters if any) are treated as an elementary alphanumeric data item and are transferred to the receiving field according to the rules of the MOVE statement.

The DELIMITED IN option causes the delimiting characters in the sending field to be treated as an elementary alphanumeric item and to be moved to the current delimiter receiving field (identifier-5) according to the rules of the MOVE statement. If, however, the delimiting condition is the end of the sending field, identifier-5 is filled with spaces.

If the COUNT IN option is specified, a value equal to the number of examined characters (excluding delimiter(s)) is moved to the data count field (identifier-6) according to rules for an elementary move (identifier-6 must be of type computational).

If the DELIMITED BY option appears then the sending field is further examined, beginning with the first character to the right of the delimiter. Otherwise, examination of the sending field begins with the first character to the right of the last character examined.

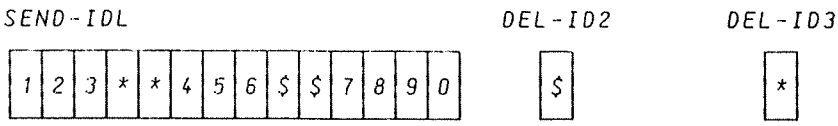
After data is transferred to the first receiving field (identifier-4), identifier-7 becomes the next receiving field. The preceding procedure is now repeated for this (and subsequently, for any succeeding receiving fields), until all characters in the sending field have been transferred or there are no more unfilled receiving fields.

Example:

The following UNSTRING statement:

```
UNSTRING SEND-IDL DELIMITED BY ALL DEL-ID2 OR DEL-ID3
   INTO REC-ID4  DELIMITER IN DREC-ID5 COUNT IN CT-ID6
        REC-ID7  DELIMITER IN DREC-ID8 COUNT IN CT-ID9
        REC-ID12 DELIMITER IN DREC-ID13 COUNT IN CT-ID14
   WITH POINTER P-ID10
   TALLYING INT-ID11
   ON OVERFLOW GO TO UNSTRING-OFL.
```

might have the following field contents at execution time:



and the remaining fields might have the following contents after execution:

REC-ID4 *DREC-ID5* *CT-ID6*

REC-ID7 *DREC-ID8* *CT-ID9*

REC-ID12 *DREC-ID13* *CT-ID14*

P-ID10

T-ID11

where b represents a space (blank character). Since SEND-ID1 still contains untransferred characters, the ON OVERFLOW condition will be raised.

If a further receiving field had been specified for this UNSTRING statement, the first character moved to it from the sending field would have been the leftmost character following the second $,i.e., the number 7. (Note the difference in effect of coding DELIMITED BY with or without ALL.)

When the execution of the UNSTRING statement has been completed, if a TALLYING IN option is present, then the field-count field (identifier-11) will have had its initial value incremented by the number of data receiving areas acted upon (including any null fields).

At this point, if a POINTER option has been specified, the pointer field (identifier-10) will contain a value equal to its initial value plus the number of characters examined in the sending field.

Execution of the UNSTRING statement will cease if an overflow condition exists. If ON OVERFLOW is specified the imperative-statement is executed. If ON OVERFLOW is not specified control passes to the next executable statement. An overflow condition occurs if:

a) The value in the pointer field is less than one or greater than the length of the sending field when UNSTRING is initiated.

b) During execution of the UNSTRING statement, after all receiving fields have been acted upon, the sending field still contains unexamined characters.

Any subscripting or indexing associated with the identifiers is evaluated immediately before data transfer.

6.7 INPUT-OUTPUT STATEMENTS

COBOL input-output statements transfer data to and from files stored on external devices, and they control low-volume data going to or coming from media such as console typewriters and terminals. In this manual and it's index, INPUT-OUTPUT is usually abbreviated to I-O.

The unit of data used by the COBOL program is called a record.

The input-output statements which may be used in the Procedure Division are determined by the file descriptions in the Environment and Data Divisions.

6.7.1 I-O Status

If the FILE STATUS clause is specified in a file-control entry, a value is placed into the specified 2-character data item during the execution of an OPEN, CLOSE, START, READ, WRITE, REWRITE or DELETE statement and before any applicable USE procedure is executed, to indicate the status of the I-O operation.

See appendix 7, Indexed/Relative I-O Status Summary.

6.7.1.1 Status Key 1

The leftmost character position of the FILE STATUS data item, upon completion of an I-O operation, is set according to the following:

'0' indicates Successful Completion

'1' indicates At End

'2' indicates Invalid Key

'3' indicates Permanent Error

'9' indicates Other Error

The meanings of the indications are:

0 - Successful Completion. The I-O statement was successfully executed.

1 - At End. *Indexed and Relative I-O.*

The Format 1 READ statement was unsuccessfully executed following an attempt to read a record when no next logical record exists in the file.

At End. *Sequential I-O.*

The sequential READ statement was unsuccessfully executed, either as a result of attempting to read a record when no next logical record existed in the file or because the first READ statement being executed for a file was described with the OPTIONAL clause, and that file was not available to the program at the time its associated OPEN statement was executed.

2 - Invalid Key. The I-O statement was unsuccessfully executed as one of the following:

Sequence Error (Indexed I-O only)

Duplicate Key

Record Not Found

Boundary Violation

Two programs attempting to access the same record (only with Indexed or Relative I-O, with MULTI-USER Access)

Invalid Key does not apply to Sequential I-O.

3 - Permanent Error. The input-output statement was unsuccessfully executed due to a boundary violation for a sequential file or as the result of an input-output error, such as data check parity error, or transmission error.

9 - Some other error.

6.7.1.2 Status Key 2

The rightmost character position of the FILE STATUS data item is known as status key 2 and is used to further describe the results of the input-output operation.

The value this character contains will have the meanings given in the following table, according to the appropriate file organization.

Status Key 2:	Meaning:
0	No further information
1	If Status Key 1 is '2' - Sequence error Otherwise - Password failure
2	If Status Key 1 is '0' - Duplicate key (Indexed files) If Status Key 1 is '2' - Duplicate key (Relative and Indexed files) Otherwise - Logic error
3	If Status Key 1 is '2' - No record found (Relative and Indexed files) Otherwise - Resource not available
4	If Status Key 1 is '2' - Boundary violation (Relative and Indexed files) If Status Key 1 is '3' - Boundary violation (Sequential files) Otherwise - No current record pointer
5	Invalid or incomplete file information
6	No file information given
7	Open successful

VALID COMBINATIONS OF STATUS KEYS 1 AND 2

The valid combinations of the values of status key 1 and status key 2 are shown in the following figures. An 'X' at an intersection indicates valid combination.

INDEXED I-O

Status Key 1	Status Key 2				
	No Further Information (0)	Sequence Error (1)	Duplicate Key (2)	No Record Found (3)	Boundary Violation (4)
Successful Completion (0)	X		X		
At End (1)	X				
Invalid Key (2)		X	X	X	X
Permanent Error (3)	X				
Other Error (9)					

RELATIVE I-O

Status Key 1	Status Key 2			
	No Further Information (0)	Duplicate Key (2)	No Record Found (3)	Boundary Violation (4)
Successful Completion (0)	X			
At End (1)	X			
Invalid Key (2)		X	X	X
Permanent Error (3)	X			
Other Error (9)				

SEQUENTIAL I-O

Status Key 1	Status Key 2	
	No Further Information (0)	Boundary Violation (4)
Successful Completion (0)	X	
At End (1)	X	
Permanent Error (3)	X	X
Other Error (9)		

6.7.1.3 The INVALID KEY Condition (Indexed and Relative I-O Only)

The INVALID KEY condition can occur as a result of the execution of a START, READ, WRITE, REWRITE or DELETE statement. For details of the causes of the condition see under the relevant statement headings.

When the INVALID KEY condition is recognized, the runtime system takes these actions in the following order:

1) A value is placed into the FILE STATUS data item, if specified for this file, to indicate an INVALID KEY condition. (See under I-O status earlier in this section.)

2) If the INVALID KEY phrase is specified in the statement causing the condition, control is transferred to the INVALID KEY imperative-statement. Any USE procedure specified for this file is not executed.

3) If the INVALID KEY phrase is not specified, but a USE procedure is specified, either explicitly or implicitly, for this file, that procedure is executed.

6.7.1.4 The AT END Condition

The AT END condition can occur as the result of a READ statement. For details see under the statement heading.

6.7.1.5 Current Record Pointer

The current record pointer is a conceptual entity for identifying the next record to be accessed within a given file. (It has no meaning for a file opened in output mode.)

The OPEN statement positions it at the first record in the file.

For a READ statement note the following:

1) If the OPEN statement positioned the current record pointer, the record identified by it is made available.

2) If a previous READ statement positioned the current record pointer then this is updated to point to the next existing record which is then made available.

3) (Indexed and Relative I-O only.) The START statement positions the current record pointer at the first record in the file that satisfies the comparison specified.

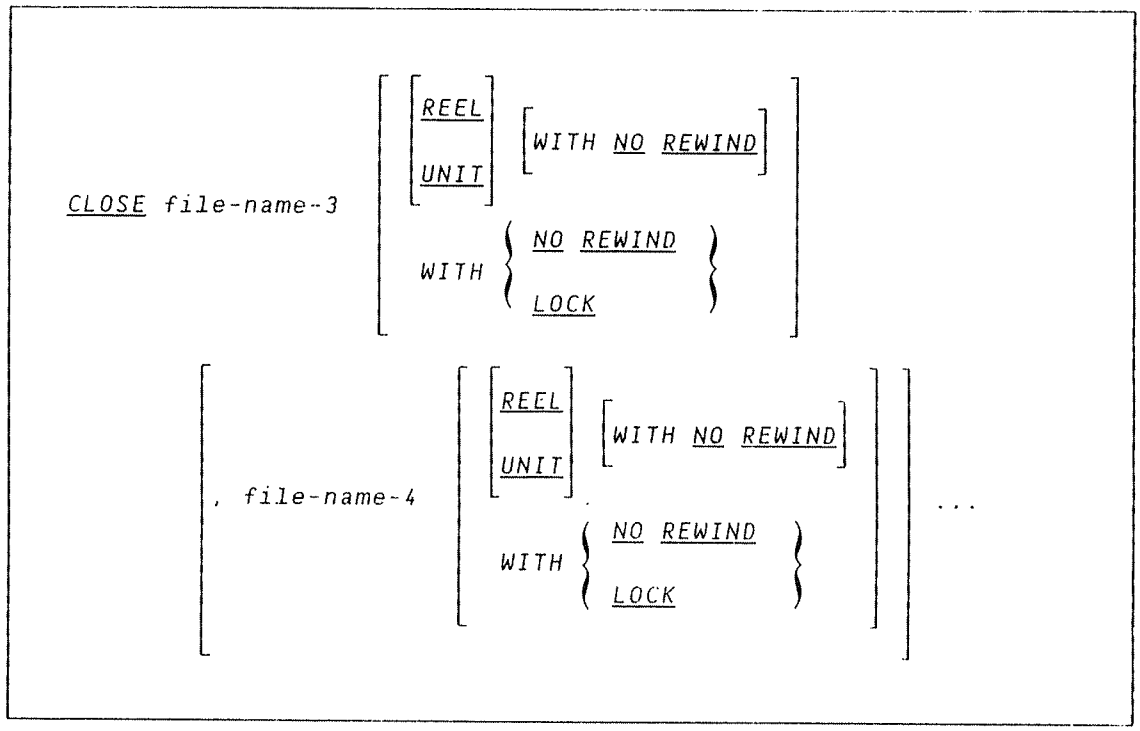
6.7.1.6 The CLOSE Statement

The CLOSE statement terminates the processing of files (with optional rewind for Sequential I-O).

Format 1 - Indexed and Relative I-O.

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

Format 2 - Sequential I-O.



The files referenced in the CLOSE statement need not all have the same organization or access.

General Rules:

1) A CLOSE statement may only be executed for a file in an open mode.

2) The action taken if a file is in the open mode when a STOP RUN statement is executed is that the file will be closed. Note, however, that the last block in memory will not be written out so that the last record on the file may be lost.

General rules for indexed and relative I-O:

1) If a CLOSE statement has been executed for a file, no other statement can be executed that references this file, whether explicitly or implicitly, unless an intervening OPEN statement for this file is executed.

2) Following the successful execution of a CLOSE statement, the record area associated with file-name is no longer available. The unsuccessful execution of such a CLOSE statement leaves the availability of the record area undefined.

General Rule For Sequential I-O:

1) Treatment of mass storage files is logically equivalent to the treatment of a file on a tape.

6.7.1.7 The DELETE Statement

The DELETE statement logically removes a record from a mass storage file. It is not used with Sequential I-O.

Format:

```
DELETE file-name RECORD [; INVALID KEY imperative-statement]
```

The INVALID KEY phrase must not be specified for a DELETE statement which references a file which is in sequential access mode.

The INVALID KEY phrase must be specified for a DELETE statement which references a file not in sequential access mode and for which an applicable USE procedure is not specified.

General Rules:

1) The associated file must be open in the I-O mode at the time of the execution of this statement.

2) For files in the sequential access mode, the last input-output statement executed for file-name prior to the execution of the DELETE statement must have been a successfully executed READ statement. The runtime system logically removes from the file the record that was accessed by that READ statement.

3) For a file in random or dynamic access mode, the runtime system logically removes from the file the record identified by the contents of the RELATIVE KEY data item associated with file-name. If the file does not contain the record specified by the key, an INVALID KEY condition exists.

4) After the successful execution of a DELETE statement, the identified record has been logically removed from the file and can no longer be accessed.

5) The execution of a DELETE statement does not affect the contents of the record area associated with file-name.

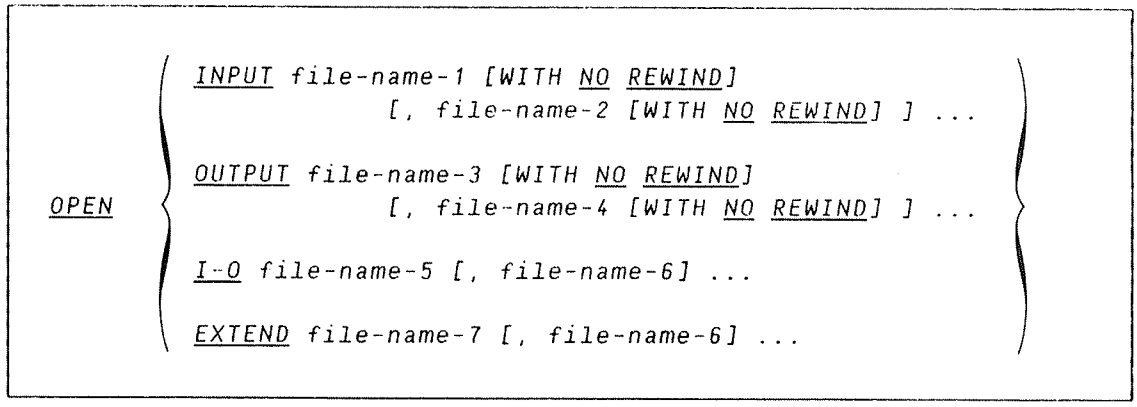
6) The current record pointer is not affected by the execution of a DELETE statement.

7) The execution of the DELETE statement causes the value of the specified FILE STATUS data item (if any) associated with file-name, to be updated.

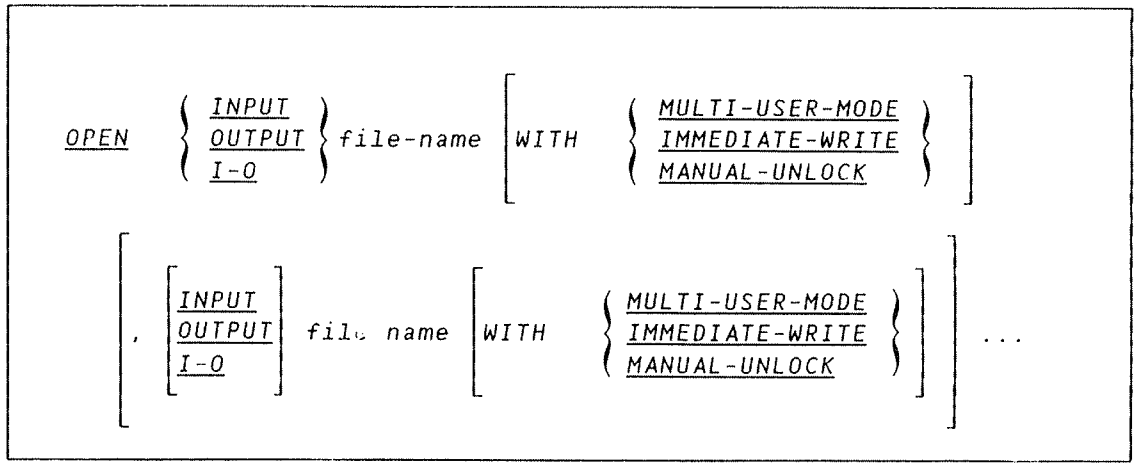
6.7.1.8 The OPEN Statement

The OPEN statement initiates the processing of files. For Indexed and Relative I-O it also performs checking and/or writing of labels and other input-output operations.

Format 1. Sequential Files.



Format 2: Indexed and Relative Files.



The files referenced in the OPEN statement need not all have the same organization or access. The I-O option can only be used for mass storage files. The EXTEND option is only valid for Sequential files.

The successful execution of an OPEN statement determines the availability of the file and results in that file being in an open mode. It also makes the associated record area available to the programs.

Prior to the successful execution of an OPEN statement for a file, no statement can be executed (except in Sequential I-O, for a SORT or MERGE statement with either GIVING or USING phrases) that references that file.

An OPEN statement must be successfully executed prior to the execution of any of the permissible input-output statements. The following tables show permissible statements for each I-O classification.

PERMISSIBLE STATEMENTS FOR INDEXED AND RELATIVE I-O-OPEN

File Access Mode	Statement	Open Mode		
		Input	Output	Input-Output
Sequential	READ	X		X
	WRITE		X	
	REWRITE			X
	START	X		X
	DELETE			X
Random	READ	X		X
	WRITE		X	X
	REWRITE			X
	START			
	DELETE			X
Dynamic	READ	X		X
	WRITE		X	X
	REWRITE			X
	START	X		X
	DELETE			X

For Indexed I-O, an 'X' indicates that the specified statement, used in the access method given for that row, may be used with the indexed file organization and the open mode given at the top of the column.

For Relative I-O, an 'X' indicates that the specified statement, used in the access method given for that row, may be used with the relative file organization and the open mode given at the top of the column.

Various kinds of errors or system malfunctions may partly or entirely destroy indexed and relative files. The programs ISAM-SERVICE and ISAM-INTER (both are self-documenting) may be used to examine such files and to restore them as far as possible.

PERMISSIBLE STATEMENTS FOR SEQUENTIAL I-O-OPEN

Statement	Open Mode			
	Input	Output	Input-Output	Extend
READ	X		X	
WRITE		X		X
REWRITE			X	

An 'X' indicates that the specified statement, used in sequential access mode, may be used with the sequential file organization and the open mode given at the top of the column.

General Rules For Indexed And Relative I-O:

1) A file may be opened with the INPUT, OUTPUT and I-O options in the same program. After the initial execution of an OPEN statement for a file, each subsequent OPEN statement execution for this file must be preceded by the execution of a CLOSE statement (without the LOCK phrase for Indexed I-O) for the same file.

2) Execution of the OPEN statement does not obtain or release the first data record.

3) The file description entry for files open for INPUT or I-O must be equivalent to that used when this file was created.

4) For files being opened with the INPUT or I-O option, the OPEN statement sets the current record pointer to the first record currently existing within the file. For indexed files, the prime record key is established as the key of reference and is used to determine the first record to be accessed. If no records exist in the file, the current record pointer is set such that "the next executed format 1 READ statement for the file will result in an AT END condition.

5) The I-O option permits the opening of a file for both input and output operations. Since this option implies the existence of the file, it cannot be used if the file is being initially created.

6) Upon successful execution of an OPEN statement with the OUTPUT option specified, a file is created. As yet, the associated file contains no data records.

7) The options *MULTI-USER-MODE*, *IMMEDIATE WRITE*, and *MANUAL-UNLOCK* are ND Extensions. They are used with relative or indexed organized files and they have the following meanings:

   MULTI-USER-MODE allows one program to be running concurrently on several terminals, each accessing the same relative or indexed organized file, and it also allows different programs running concurrently on several terminals to access the same relative or indexed organized file. In both of these cases, if the programs access the same record in the file, conflicts are prevented.

   Note that when you use MULTI-USER-MODE, output to the files is done in a different way from what is usually the case. When writing files in other modes, data are buffered, so that the computer can organize the data transfers etc. in a convenient way. When MULTI-USER-MODE is chosen, the buffers are bypassed to achieve immediate updates of the data. The consequence of this is that MULTI-USER-MODE file output is somewhat *slower* than ordinary access.

IMMEDIATE-WRITE (single user only) causes records to be written immediately back to the file - a process which happens automatically in MULTI-USER-MODE; otherwise output is buffered. This option is useful, for instance, where high security is required.

MANUAL-UNLOCK will prevent the automatic unlock of records until an UNLOCK statement is encountered. However, the user is strongly advised to allow automatic unlock of records since the use of MANUAL-UNLOCK can lead to deadlocks.

Note that the multi-user supervisor must be active before running programs in multi-user mode. The system supervisor for the users installation should do this.

General Rules for Sequential I-O:

1) A file may be opened with the INPUT, OUTPUT, EXTEND and I-O options in the same program. Following the initial execution of an OPEN statement, each subsequent OPEN statement for the same file must be preceded by the execution of a CLOSE statement for it.

2) Execution of the OPEN statement does not obtain or release the first data record. The file description entry for file-names 1, 2, 5, 6, 7, or 8 must be the equivalent to that used when the file was created.

3) If an input file is designated with the OPTIONAL phrase in its SELECT clause, the object program causes an interrogation for the presence or absence of this file. If the file is not present, the first READ statement for this file causes the AT END condition to occur. (See the READ statement later in this section.)

4) For files being opened with the INPUT or I-O option, the OPEN statement sets the current record pointer to the first record currently existing within the file. If no records exist in the file, the current record pointer is set such that the next executed READ statement for the file will result in an AT END condition.

5) The EXTEND option allows the file to be opened for output operations. (The OPEN statement positions the file immediately following the last logical record of that file. Subsequent WRITE statements referencing the file will add records to the file as though the file had been opened with the OUTPUT option.)

6) The I-O option allows the opening of a mass storage file for both input and output operations. Since this option implies the existence of the file, it cannot be used if the mass storage file is being initially created.

7) Upon successful execution of an OPEN statement with the OUTPUT option specified, a file is created. As yet the associated file contains no data records.

8) If the OPTIONAL phrase has been given for the file in the FILE-CONTROL paragraph of the Environment Division and the file is not present, then the standard end-of-file processing is performed for that file if it is an input file. If it is an output file it is created.

6.7.1.9 The READ Statement

The READ statement makes the next logical record from a file available. The formats are:

Format 1:

```
READ file-name [ NEXT     ] RECORD [INTO identifier] [WITH LOCK]
               [ PREVIOUS ]
    [; AT END imperative-statement]
```

Format 2. Indexed I-O Only:

```
READ file-name RECORD [INTO identifier] [WITH LOCK]
   [; KEY IS data-name] [; INVALID KEY imperative-statement]
```

Format 3. Relative I-O Only:

```
READ file-name RECORD [INTO identifier] [WITH LOCK]
   [;INVALID KEY imperative-statement]
```

The storage areas associated with file-name and with identifier must not be the same.

Format 1.

The NEXT/PREVIOUS phrase and the WITH LOCK phrase are not valid for sequential files. This format is used for sequential retrieval of records when files organized in another way are used in the dynamic access mode. The WITH LOCK phrase applies only to files opened in MULTI-USER-MODE, and is an ND Extension (see the OPEN statement).

Format 1 must be used for all files in sequential access mode.

If the AT END phrase appears and if no applicable USE procedure is given for file-name, a runtime error will result.

Format 2. Indexed I-O Only:

Data-name, which may be qualified, must identify a record key associated with file-name.

Formats 2 and 3:

These formats are used for files in dynamic or random access modes when records are to be retrieved randomly.

If the INVALID KEY phrase appears and if no applicable USE procedure is given for file-name, a runtime error will result.

The WITH LOCK phrase is an ND Extension and applies only to files opened in MULTI-USER-MODE (see the OPEN statement and General Rule 3 for Indexed and Relative I-O later in this section).

General Rules:

1) The associated file must be open in the INPUT or I-O mode when this statement is executed. (See the OPEN statement in the previous section.)

2) The execution of the READ statement causes the value of the FILE STATUS data item, if any, to be updated. (See I-O Status at the beginning of the I-O section.)

3) If the INTO phrase is specified, the record being read is moved from the record area to the area specified by identifier according to the rules specified for the MOVE statement. The implied MOVE does not occur if the execution of the READ statement was unsuccessful. Any indexing associated with identifier is evaluated after the record has been read and immediately before it is moved to the data item.

4) When the INTO phrase is used, the record being read is available in both the input record area and the data area associated with identifier.

5) If, at the time of execution of a format 1 READ statement, the position of the current record pointer for that file is undefined, the execution of that READ statement is unsuccessful. (See I-O Status at the beginning of the I-O section.)

6) When the AT END condition is recognized, the following actions are taken in the specified order:

   a) A value is placed into the FILE STATUS data item (if specified for this file), to indicate an AT END condition.

   b) If the AT END phrase is specified in the statement causing the condition, control is transferred to the AT END imperative-statement. Any USE procedure specified for this file is not executed.

   c) If the AT END phrase is not specified, then a USE procedure must be specified, either explicitly or implicitly, for this file, and the procedure is executed.

      When the AT END condition occurs, execution of the input-output statement which caused the condition is unsuccessful.

7) Following the unsuccessful execution of any READ statement, the contents of the associated record area and the position of the current record pointer are undefined. For indexed files the key of reference is also undefined.

8) For a file for which dynamic access mode is specified, a format 1 READ statement with the NEXT/PREVIOUS phrase specified causes the next or previous logical record to be retrieved from that file, as described in rule 1.

General Rules For Indexed I-O:

1) The record to be made available by a format 1 READ statement is determined as follows:

   a) The record, pointed to by the current record pointer, is made available provided that the current record pointer was positioned by the START or OPEN statement and the record is still accessible through the path indicated by the current record pointer; if the record is no longer accessible (which may have been caused by deletion of the record or by a change in an alternate record key) the current record pointer is updated to point to the next existing record within the established key of reference, and that record is then made available.

   b) If the current record pointer was positioned by the execution of a previous READ statement, it is updated to point to the next existing record in the file with the established key of reference. That record is then made available.

2) For an indexed file being sequentially accessed, records having the same duplicate value in an alternate record key which is the key of reference, are made available in the same order in which they are released, by execution of WRITE statements, or by execution of REWRITE statements which create such duplicate values.

3) For an indexed file if the KEY phrase is specified in a format 2 READ statement, data-name is established as the key of reference for this retrieval. If the dynamic access mode is specified, this key of reference is also used for retrievals by any subsequent executions of format 1 READ statements for the file until a different key of reference is established for the file.

4) If the KEY phrase is not specified in a Format 2 READ statement, the prime record key is established as the key of reference for this retrieval. If the dynamic access mode is specified, this key of reference is also used for retrievals by any subsequent executions of format 1 READ statements for the file until a different key of reference is established for it.

5) If execution of a format 2 READ statement causes the value of the key of reference to be compared with the value contained in the corresponding data item of the stored records in the file, until the first record having an equal value is found. The current record pointer is positioned to this record which is then made available. If no record can be so identified, the INVALID KEY condition exists and execution of the READ statement is unsuccessful.

General Rules For Indexed And Relative I-O:

1) PREVIOUS is defined to be the opposite of NEXT.

2) If, at the time of the execution of a format 1 READ statement, no next logical record exists in the file, the AT END condition occurs, and the execution of the READ statement is considered unsuccessful.

3) When the AT END condition has been recognized, a format 1 READ statement for that file must not be executed without first executing one of the following:

   a) A successful CLOSE statement followed by the execution of a successful OPEN statement for that file.

   b) A successful START statement for that file.

   c) A successful format 2 (for Indexed I-O) READ statement for that file (or format 3 for Relative I-O).

4) The WITH LOCK phrase exists for the access of files which have been opened in MULTI-USER-MODE (see the OPEN statement). If the file has been opened in single-user mode, the phrase is treated as comments only.

   If coded, READ WITH LOCK ensures that two programs cannot modify the same record at the same time. A locked record can only be read. It will become unlocked automatically when it has been rewritten by the program which locked it, or when another record has been read, or when the file is closed. The record can also be unlocked upon execution of the UNLOCK statement (see Section 6.7.1.12).

   An attempt by two programs to modify the same record will raise the INVALID KEY condition with a file status code of 68 or 78 depending on whether the records are "locked" or not.

   It is the responsibility of the user program to provide the code which enables it to wait for a record to become accessible; a read loop might otherwise occur.

   Note also the requirements for relative or indexed organized files accessed in multi-user mode. All relative or indexed organized files, both the index and data part, must be contiguous SINTRAN files. The size of the index part (in SINTRAN pages) is found by using the ESTIMATE-INDEX-FILE-SIZE function in the INDEXED SEQUENTIAL ACCESS METHOD SERVICE (or ISAM-Service) program. This self-explaining program is started by typing @ISAM-SERVICE when in SINTRAN. The size (in SINTRAN pages) of the data part is:

   *(maximum number of records * record length/2048) + 1*

General Rules For Relative I-O Only:

1) The existence of records numbered, say, 2000 and 4000, does not imply that mass storage file space for records 0 - 1999 and 2001 - 3999 is occupied. A relative file only occupies enough space for the records it is presently holding. Thus, it makes perfect sense to access records by, say, social security numbers or numerical representations of birth dates.

2) The record to be made available by a format 1 READ statement is determined as follows:

   a) The record, pointed to by the current record pointer, is made available provided that the current record pointer was positioned by the START or OPEN statement and the record is still accessible through the path indicated by the current record pointer; if the record is no longer accessible, which may have been caused by the deletion of the records, the current record pointer is updated to point to the next existing record in the file and then that record is made available.

b) If the current record pointer was positioned by the execution of a previous READ statement, the current record pointer is updated to point to the next existing record in the file with the established key of reference and then that record is made available.

3) If the RELATIVE KEY phrase is specified, the execution of a format 1 READ statement updates the contents of the RELATIVE KEY data item such that it contains the relative record number of the record made available.

4) The execution of a format 2 READ statement sets the current record pointer to, the record whose relative record number is contained in the data item named in the RELATIVE KEY phrase for the file and makes this record available. If the file does not contain such a record, the INVALID KEY condition exists and execution of the READ statement is unsuccessful.

General Rules For Sequential I-O:

1) The record to be made available by a format 1 READ statement is determined as follows:

a) If the current record pointer was positioned by the execution of the OPEN statement, the record pointed to by the current record pointer is made available.

b) If the current record pointer was positioned by the execution of a previous READ statement, the current record pointer is updated to point to the next existing record in the file and then that record is made available.

2) When the AT END condition has been recognized, a READ statement for that file must not be executed without first executing a successful CLOSE statement followed by the execution of a successful OPEN statement for that file.

3) If at the time of the execution of a READ statement, no next logical record exists in the file, the AT END condition occurs, and the execution of the READ statement is considered unsuccessful.

4) If a file described with the OPTIONAL phrase is not present at the time the file is opened, then at the time of execution of the first READ statement for the file, the AT END condition occurs and the execution of the READ statement is unsuccessful. The standard end-of-file procedures are not performed. Execution then proceeds as specified in general rule 6.

EXAMPLE:

```
 1        IDENTIFICATION DIVISION.
 2        PROGRAM-ID.
 3            GEN-ISAM-1.
 4       ****************************************************************
 5       * ISAM MEANS INDEX-SEQUENTIAL ACCESS METHOD.
 6       *
 7       * THE RECORDS ARE OUTPUT TO AN ISAM-FILE USING THE *UNIQUE*
 8       * (I. E., NOT DUPLICATED) DATA FOUND IN FIELD ISAM-KEY AS
 9       * *KEY* VALUE.
10       *
11       * BEFORE THIS JOB CAN BE RUN, THE FOLLOWING *MUST* BE MET:
12       *    A) FILE "ISAM-EX:DATA" AND "ISAM-EX:ISAM" MUST EXIST
13       *        AND BE CONSISTENT; OR
14       *    B) FILE "ISAM-EX:DATA" MUST NOT EXIST OR IF EXISTING
15       *        CONTAIN *NO DATA !!*
16       ****************************************************************
17        ENVIRONMENT DIVISION.
18        INPUT-OUTPUT SECTION.
19        FILE-CONTROL.
20            SELECT ISAM-FILE ASSIGN TO "ISAM-EX:DATA",
21                ORGANIZATION IS INDEXED,
22                ACCESS MODE IS DYNAMIC,
23                RECORD KEY IS ISAM-KEY,
24                FILE STATUS IS ISAMSTATUS.
25        DATA DIVISION.
26        FILE SECTION.
27        FD  ISAM-FILE
28                RECORD CONTAINS 46 CHARACTERS.
29        01  ISAM-REC.
30            02  ISAM-KEY PIC X(6).
31       *            :.....................MUST BE IN RECORD AREA!
32            02  ISAM-TEXT PIC X(40).
33
34        WORKING-STORAGE SECTION.
35        01  ISAMSTATUS PIC XX.
36       *            :.....................RETURN STATUS FROM ISAM.
37
38        PROCEDURE DIVISION.
39        A001.
40            OPEN I-O ISAM-FILE.
41        A002.
42            DISPLAY "ENTER KEY (MAX. 6 CHAR) : ",
43                ACCEPT ISAM-KEY.
44            IF ISAM-KEY = SPACES GO TO LIST.
45       *                 :..............SPACES INPUT, END DIALOG.
46            DISPLAY "ENTER TEXT (MAX 40 CHAR) : ",
47                ACCEPT ISAM-TEXT.
48       *               :.................READ RECORDS FROM TERMINAL.
49            WRITE ISAM-REC, INVALID KEY,
50                DISPLAY "ISAM FILE ERROR :", ISAMSTATUS, ":".
51            GO TO A002.
52       *        :.......................OUTPUT RECORD AND ASK AGAIN.
53        LIST.
54            DISPLAY "ENTER ACCESS KEY: ",
55                ACCEPT ISAM-KEY.
56            IF ISAM-KEY = SPACES THEN GO TO FINI.
57            READ ISAM-FILE RECORD KEY IS ISAM-KEY INVALID KEY,
58                DISPLAY "RECORD NOT FOUND!",
59                GO TO LIST.
60            DISPLAY "REC: ", ISAM-KEY, ": ",ISAM-REC.
61            GO TO LIST.
62        FINI.
63            CLOSE ISAM-FILE.
64            DISPLAY "JOB END.".
65            STOP RUN.
```

EXAMPLE:

```
 1        IDENTIFICATION DIVISION.
 2        PROGRAM-ID.     GENRELATIVE.
 3       ****************************************************************
 4       * THIS PROGRAM SHOWS THE USAGE OF A RELATIVE FILE.
 5       *
 6       * THE FILE *MUST* EXIST BEFORE THE RUN, BUT MAY BE EMPTY. EACH
 7       * RECORD IS LOCATED RELATIVE TO RECORD 1 IN THE FILE BY ITS
 8       * *INTEGER* KEY VALUE.
 9       *
10       * NOTE: EVEN IF SOMETHING IS WRITTEN ON RECORDS 300 AND 700
11       * IN THE RELATIVE FILE, *NO* STORAGE SPACE IS USED FOR THE
12       * "EMPTY" RECORDS BETWEEN RECORD 1 AND 299, OR BETWEEN 301 AND
13       * 699. THUS IT MAKES PERFECT SENSE TO USE, SAY, BIRTH DATES
14       * AS KEYS IN RELATIVE FILES.
15       ****************************************************************
16        ENVIRONMENT DIVISION.
17        INPUT-OUTPUT SECTION.
18        FILE-CONTROL.
19            SELECT RELFILE ASSIGN "RELATIVE-EX:DATA",
20                        ORGANIZATION IS RELATIVE,
21                        ACCESS IS DYNAMIC,
22                        RELATIVE KEY IS REL-KEY,
23                        FILE STATUS IS REL-STATUS.
24        DATA DIVISION.
25        FILE SECTION.
26
27        FD RELFILE
28                        BLOCK CONTAINS 10 RECORDS
29                        RECORD CONTAINS 60 CHARACTERS.
30        01  REL-RECORD  PIC X(60).
31
32        WORKING-STORAGE SECTION.
33        01  REL-STATUS  PIC XX.
34        01  REL-KEY     PIC 999.
35       ********:*******************************************************
36       *       :........THE RELATIVE KEY CAN NOT OCCUR IN THE RECORD
37       *                AREA. ITS POSSIBLE SIZES ARE 1-99999999999,
38       *                BUT IT IS RESTRICTED TO 999 IN THIS PROGRAM.
39       ****************************************************************
40        PROCEDURE DIVISION.
41        A000.
42            OPEN I-O RELFILE.
43        A002.
44            DISPLAY "ENTER KEY (MAX 999) : ".
45            PERFORM GET-KEY.
46            IF REL-KEY = ZEROES GO TO A003.
47            DISPLAY "ENTER TEXT (MAX 60 CHARACTERS) : ".
48            ACCEPT REL-RECORD.
49            WRITE REL-RECORD INVALID KEY,
50                        DISPLAY "*** RELFILE ERROR *** :", REL-STATUS.
51            GO TO A002.
52        A003.
53            DISPLAY "ENTER ACCESS KEY: ".
54            PERFORM GET-KEY.
55            IF REL-KEY = ZEROS GO TO A999.
56            READ RELFILE RECORD INVALID KEY,
57                        DISPLAY "*** RECORD NOT FOUND ***",
58                        REL-STATUS, GO TO A003.
59            DISPLAY "REC :", REL-KEY, ": ", REL-RECORD.
60            GO TO A003.
61        A999.
62            CLOSE RELFILE.
63            DISPLAY "JOB END".
64            STOP RUN.
65        GET-KEY.
66            ACCEPT REL-KEY.
67            IF REL-KEY NOT NUMERIC,
68                        DISPLAY "*** KEY MUST BE NUMERIC ***",
69                        GO TO GET-KEY.
70        GET-KEY-EXIT.
71            EXIT.
```

EXAMPLE:

```
 1       IDENTIFICATION DIVISION.
 2       PROGRAM-ID.
 3           GENSEQ.
 4      ******************************************************************
 5      *    CREATES SQ-FILES AND LISTS.
 6      ******************************************************************
 7       ENVIRONMENT DIVISION.
 8       INPUT-OUTPUT SECTION.
 9       FILE-CONTROL.
10           SELECT SQ-FILE ASSIGN "COB1:DATA",
11               ORGANIZATION IS SEQUENTIAL,
12               ACCESS IS SEQUENTIAL.
13       DATA DIVISION.
14       FILE SECTION.
15       FD  SQ-FILE.
16       01  M-REC.
17           02          PIC X(10).
18           02 SEQNUM   PIC 9(5), BLANK WHEN ZERO.
19           02          PIC X(5).
20           02          PIC X(40).
21       WORKING-STORAGE SECTION.
22       01  RANDNO      PIC  9(4) PACKED-DECIMAL, VALUE ZERO.
23       01  MAXRAND     PIC S9(4) PACKED-DECIMAL, VALUE 1000.
24       01  NORECS      PIC  9(4) PACKED-DECIMAL.
25       01  RECCNT      PIC  99, COMP, VALUE 0.
26
27       PROCEDURE DIVISION.
28       INIT-01.
29           OPEN OUTPUT SQ-FILE.
30           DISPLAY "CREATE RECORDS?".
31           PERFORM GET-NORECS.
32           PERFORM CRE-SQ-FILE NORECS TIMES.
33      * BUILDS THE INPUT FILE.
34           CLOSE SQ-FILE.
35           DISPLAY "FILE SQ-FILE CREATED.", RECCNT, " RECORDS.".
36           OPEN INPUT SQ-FILE.
37       LIST-FILE-0.
38           MOVE 0 TO RECCNT.
39       LIST-FILE-1.
40           READ SQ-FILE AT END GO TO LIST-END.
41           ADD 1 TO RECCNT.
42           DISPLAY "REC ", RECCNT, ", SEQNUM = ", SEQNUM.
43           GO TO LIST-FILE-1.
44       LIST-END.
45           CLOSE SQ-FILE.
46           DISPLAY "JOB FINISH".
47           STOP RUN.
48       CRE-SQ-FILE.
49           CALL "RND" USING RANDNO, MAXRAND.
50           MOVE ALL "*" TO M-REC.
51           MOVE RANDNO TO SEQNUM.
52           ADD 1 TO RECCNT.
53           DISPLAY "UT REC = ", RECCNT, " KEY = ", SEQNUM.
54           WRITE M-REC.
55       GET-NORECS.
56           ACCEPT NORECS.
57           IF NORECS NOT NUMERIC,
58               DISPLAY "*** NOT NUMERIC DATA ***",
59               GO TO GET-NORECS
60           END-IF.
```

6.7.1.10 The REWRITE Statement

The REWRITE statement logically replaces a record existing in a mass storage file.

Format 1:

```
REWRITE record-name [FROM identifier]
```

Format 2. Indexed and Relative I-O Only:

```
REWRITE record-name [FROM identifier]
   [; INVALID KEY imperative-statement]
```

Record-name and identifier must not refer to the same storage area.

Record-name is the name of a logical record in the File Section of Data Division.

For Relative I-O, the INVALID KEY phrase must be specified in the REWRITE statement for files in the random or dynamic access mode for which an appropriate USE procedure is not specified. It must not be specified for a REWRITE statement for a file in sequential access mode.

For Indexed I-O, the INVALID KEY phrase must be specified in the REWRITE statement for files which do not have an appropriate USE procedure for them.

General Rules:

1) The file associated with record-name (which must be a mass-storage file for Sequential I-O) must be open in the I-O mode at the time of execution of the statement.

2) The last input-output statement executed for the associated file prior to the execution of the REWRITE statement must have been a successfully executed READ statement.

3) The number of character positions in the record referenced by record-name must be equal to the number of character positions in the record being replaced.

4) The logical record released by a successful execution of the REWRITE statement is no longer available in the record area.

5) The execution of a REWRITE statement with the FROM phrase is equivalent to the execution of:

MOVE identifier TO record-name

followed by the execution of the *same REWRITE* statement without the FROM phrase. The contents of the record area prior to the execution of the implicit MOVE statement have no effect on the execution of the REWRITE statement.

6) The current record pointer is not affected by the execution of a REWRITE statement.

7) The execution of the REWRITE statement causes the value of the FILE STATUS data item (if any) associated with the file, to be updated.

8) For Relative I-O, with a file accessed in either random or dynamic access mode, the runtime system logically replaces the record referenced by the RELATIVE KEY data item for the file. If this file does not contain this record, the INVALID KEY condition exists. The updating operation will not take place.

General Rules for Indexed I-O only:

1) For a file in the sequential access mode, the record to be replaced is indicated by the prime record key. When the REWRITE statement is executed, the value in the prime record key data item of the record to be replaced must be the same as that of the last record read from this file. For a file in random or dynamic access mode, the record to be replaced is specified by the prime record key data item.

2) The contents of alternate record key data items of the record being rewritten may differ from those in the record being replaced. The runtime system utilizes the contents of the record key data items during the execution of the REWRITE statement in such a way that subsequent access to the record may be based upon any of those specified record keys.

3) The INVALID KEY condition exists when:

a) The access mode is sequential and the value contained in the prime record key data item of the record to be replaced is not equal to the value of the prime record key of the last record read from this file, or

b) The value contained in the prime record key data item does not equal that of any record stored in the file, or

c) The value contained in an alternate record key data item for which a DUPLICATES clause has not been specified is equal to that of a record already stored in the file, or

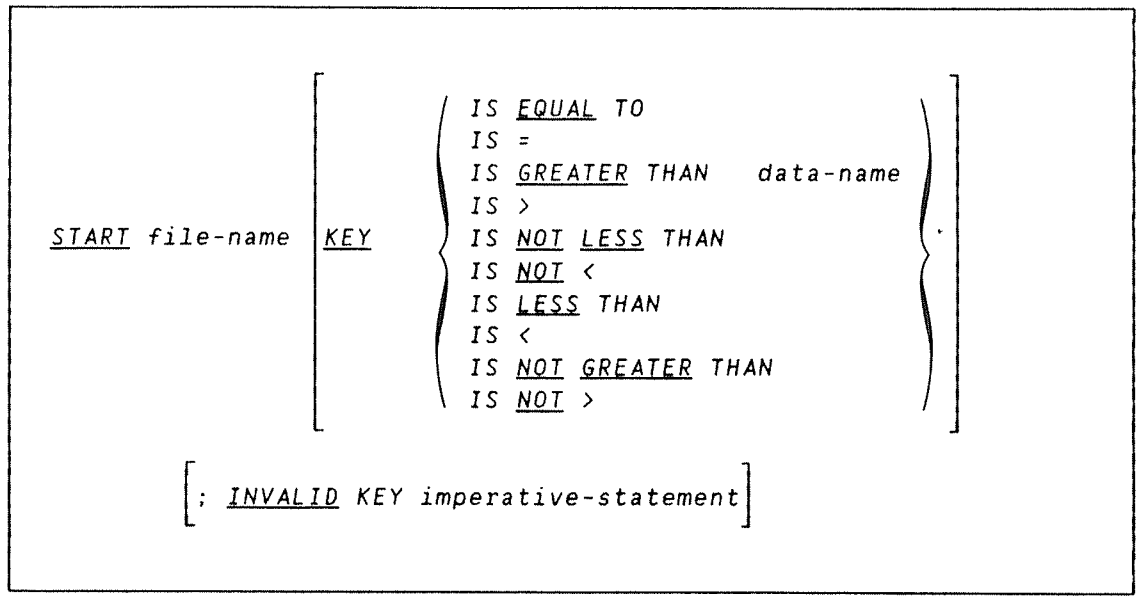
d) The record has been modified by another user. File status is '78' (for files opened in multi-user mode only).

The updating operation does not take place and the data in the record area is unaffected.

6.7.1.11 The START Statement

The START statement provides a basis for logical positioning within an indexed or relative file, for subsequent retrieval of records.

Format:



Note: The required relational characters '>', '<', and '=' are not underlined to avoid confusion with other symbols such as ≥ (greater than or equal to).

File-name must be the name of a file with sequential or dynamic access.

The INVALID KEY phrase must be specified if no applicable USE procedure is specified for file-name.

Data-name may be qualified, and for Relative I-O, it must be the data item specified in the RELATIVE KEY phrase of the associated file-control entry.

General Rules:

1) File-name must be open in the INPUT or I-O mode at the time that the START statement is executed. (See the OPEN statement.)

2) If the KEY option is not specified, the relational operator 'IS EQUAL TO' is implied.

3) If the KEY option is present, the comparison specified in the KEY relational operator is made between data-name and the corresponding key field associated with the records of the file.

4) The execution of the START statement causes the current value in the key data-name and the corresponding key field of the file's records to be compared. The current record pointer is positioned at the logical record in the file whose key field satisfies the comparison. (If the comparison is not satisfied by any record in the file, an INVALID KEY condition exists, the execution of the START statement is unsuccessful, and the position of the current record pointer is undefined.)

5) The execution of the START statement also causes the value of the FILE STATUS data item (if any) associated with file-name to be updated. (See I-O Status at the beginning of this section.)

General Rules For Indexed Files:

1) If the KEY option is not specified, then the IS EQUAL TO comparison is made with the prime RECORD KEY data item. After successful execution of the START statement RECORD KEY or ALTERNATE RECORD KEY becomes the key of reference for subsequent READ statements.

2) If a KEY option is present, then the comparison is made with the data item which may be the prime RECORD KEY, an ALTERNATE RECORD KEY, or an alphanumeric data item subordinate to a record key having its leftmost character position corresponding to the leftmost character position of that record key.

3) The current record pointer is positioned as in general rule 4. If the operands in the comparison are of unequal length, the comparison takes place as if the longer field were truncated on the right of the length of the shorter field.

4) If the execution of the START statement is not successful, the key of reference is undefined.

General Rule For Relative Files:

1) The KEY data item used in the comparison is that associated with RELATIVE KEY, whether or not the KEY option appears. Thus, when the KEY option does not appear, the data-name must specify RELATIVE KEY.

EXAMPLE:

```
 1        IDENTIFICATION DIVISION.
 2        PROGRAM-ID.
 3            GEN-ISAM-2.
 4       ****************************************************************
 5       * ISAM MEANS INDEX-SEQUENTIAL ACCESS METHOD.
 6       *
 7       * THE RECORDS ARE OUTPUT TO AN ISAM-FILE USING THE *UNIQUE*
 8       * (I. E., NOT DUPLICATED) DATA FOUND IN FIELD ISAM-KEY AS
 9       * *KEY* VALUE.
10       *
11       * BEFORE THIS JOB CAN BE RUN, THE FOLLOWING *MUST* BE SO:
12       *    A) FILE "ISAM-EX:DATA" MUST EXIST; OR
13       *    B) FILE "ISAM-EX:DATA" MUST NOT EXIST OR IF EXISTING
14       *        CONTAIN *NO DATA !!*
15       ****************************************************************
16        ENVIRONMENT DIVISION.
17        INPUT-OUTPUT SECTION.
18        FILE-CONTROL.
19            SELECT ISAM-FILE ASSIGN TO "ISAM-EX:DATA",
20                ORGANIZATION IS INDEXED,
21                ACCESS MODE IS DYNAMIC,
22                RECORD KEY IS ISAM-KEY,
23                FILE STATUS IS ISAMSTATUS.
24        DATA DIVISION.
25        FILE SECTION.
26        FD  ISAM-FILE
27            RECORD CONTAINS 46 CHARACTERS.
28        01  ISAM-REC.
29            02  ISAM-KEY PIC X(6).
30       *          :.....................MUST BE IN RECORD AREA!
31            02  ISAM-TEXT PIC X(40).
32
33        WORKING-STORAGE SECTION.
34        01  ISAMSTATUS PIC XX.
35       *           :.....................RETURN STATUS FROM ISAM.
36
37        PROCEDURE DIVISION.
38        A001.
39            OPEN I-O ISAM-FILE.
40        A002.
41            DISPLAY 'ENTER KEY (MAX. 6 CHAR) : ',
42                ACCEPT ISAM-KEY.
43            IF ISAM-KEY = SPACES GO TO LIST.
44       *                 :.............SPACES INPUT, END DIALOG.
45            DISPLAY 'ENTER TEXT (MAX 40 CHAR) : ',
46                ACCEPT ISAM-TEXT.
47       *              :.................READ RECORDS FROM TERMINAL.
48            WRITE ISAM-REC, INVALID KEY,
49                DISPLAY 'ISAM FILE ERROR :', ISAMSTATUS, ':'.
50            GO TO A002.
51       *          :.....................OUTPUT RECORD AND ASK AGAIN.
52        LIST.
53            DISPLAY 'ENTER ACCESS KEY: ',
54                ACCEPT ISAM-KEY.
55            IF ISAM-KEY = SPACES THEN GO TO FINI.
56            START ISAM-FILE KEY IS EQUAL TO ISAM-KEY, INVALID KEY,
57                DISPLAY 'KEY NOT FOUND!',
58                GO TO LIST.
59            READ ISAM-FILE RECORD KEY IS ISAM-KEY INVALID KEY,
60                DISPLAY 'RECORD NOT FOUND!',
61                GO TO LIST.
62            DISPLAY 'REC: ', ISAM-KEY, ': ',ISAM-REC.
63            GO TO LIST.
64        FINI.
65            CLOSE ISAM-FILE.
66            DISPLAY 'JOB END.'.
67            STOP RUN.
```

EXAMPLE:

```
 1       IDENTIFICATION DIVISION.
 2       PROGRAM-ID.     GEN-EX2.
 3      ****************************************************************
 4      * THIS PROGRAM SHOWS THE USAGE OF A RELATIVE FILE.
 5      *
 6      * THE FILE *MUST* EXIST BEFORE THE RUN, BUT MAY BE EMPTY, EACH
 7      * RECORD IS LOCATED DIRECTLY BY ITS RELATIVE (TO 1) POSITION IN
 8      * THE FILE BY ITS *INTEGER* KEY VALUE.
 9      *
10      * NOTE: EVEN IF SOMETHING IS WRITTEN ON RECORDS 300 AND 700
11      * IN THE RELATIVE FILE, *NO* STORAGE SPACE IS USED FOR THE
12      * "EMPTY" RECORDS BETWEEN RECORD 0 AND 299, OR BETWEEN 301 AND
13      * 699. THUS IT MAKES PERFECT SENSE TO USE, SAY, BIRTH DATES
14      * AS KEYS IN RELATIVE FILES.
15      ****************************************************************
16       ENVIRONMENT DIVISION.
17       INPUT-OUTPUT SECTION.
18       FILE-CONTROL.
19           SELECT RELFILE ASSIGN "RELATIVE-EX:DATA",
20                       ORGANIZATION IS RELATIVE,
21                       ACCESS IS DYNAMIC,
22                       RELATIVE KEY IS REL-KEY,
23                       FILE STATUS IS REL-STATUS.
24       DATA DIVISION.
25       FILE SECTION.
26
27       FD RELFILE
28                       BLOCK CONTAINS 10 RECORDS
29                       RECORD CONTAINS 60 CHARACTERS.
30       01  REL-RECORD  PIC X(60).
31      ****************************************************************
32      *                RECORD CAN NOT BE OF THE "QED" TYPE.
33      ****************************************************************
34       WORKING-STORAGE SECTION.
35       01  REL-STATUS  PIC XX.
36       01  REL-KEY     PIC 999.
37      ********;*************************************************************
38      *       :........THE RELATIVE KEY CAN NOT OCCUR IN THE RECORD
39      *                AREA. ITS POSSIBLE SIZES ARE 1-99999999999,
40      *                BUT IT IS RESTRICTED TO 999 IN THIS PROGRAM.
41      ****************************************************************
42       PROCEDURE DIVISION.
43       A000.
44           OPEN I-O RELFILE.
45       A002.
46           DISPLAY "ENTER KEY (MAX 999) : ".
47           PERFORM GET-KEY.
48           IF REL-KEY = ZEROES GO TO A003.
49           DISPLAY "ENTER TEXT (MAX 60 CHARACTERS) : ".
50           ACCEPT REL-RECORD.
51           WRITE REL-RECORD INVALID KEY,
52                       DISPLAY "*** RELFILE ERROR *** :", REL-STATUS.
53           GO TO A002.
54       A003.
55           DISPLAY "ENTER ACCESS KEY: ".
56           PERFORM GET-KEY.
57           IF REL-KEY = ZEROS GO TO A999.
58           START RELFILE KEY IS EQUAL REL-KEY, INVALID KEY,
59                       DISPLAY "*** RECORD NOT FOUND ***",
60                       REL-STATUS, GO TO A003.
61           READ RELFILE.
62           DISPLAY "REC :", REL-KEY, ": ", REL-RECORD.
63           GO TO A003.
64       A999.
65           CLOSE RELFILE.
66           DISPLAY "JOB END".
67           STOP RUN.
68       GET-KEY.
69           ACCEPT REL-KEY.
70           IF REL-KEY NOT NUMERIC,
71                       DISPLAY "*** KEY MUST BE NUMERIC ***",
72                       GO TO GET-KEY.
73       GET-KEY-EXIT.
74           EXIT.
```

6.7.1.12 The UNLOCK Statement

The UNLOCK statement unlocks records which have been locked in MANUAL-UNLOCK-MODE.

Format:

UNLOCK file-name

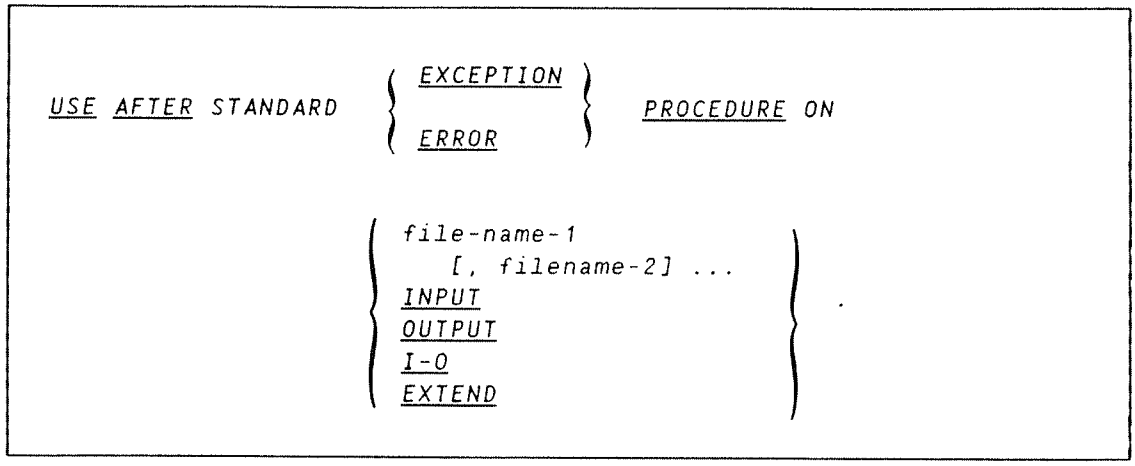
The UNLOCK statement is an ND Extension and it is used for programs accessing relative or indexed organized files in MULTI-USER-MODE (see the OPEN statement).

Records are normally unlocked automatically after a REWRITE or another READ on the same file, or when the file is closed. However the MANUAL-UNLOCK option on the OPEN statement, if present, will prevent this until an UNLOCK statement is encountered for the file.

6.7.1.13 The USE Statement

The USE statement specifies procedures for I-O error handling in addition to the standard procedures provided by the I-O control system.

Format:



The EXTEND option is valid for Sequential I-O only.

A USE statement, when present, must immediately follow a section header in the Declaratives Section of the Procedure Division. (See under Declaratives at the beginning of the Procedure Division description.)

The USE statement itself is never executed, it merely defines the conditions requiring execution of the USE procedure.

The files referenced need not all have the same organization or access.

THE EXCEPTION/ERROR PROCEDURE

This procedure is executed after completion of the standard system I-O routine or when an AT END or INVALID KEY option has not been specified in the INPUT-OUTPUT statement.

EXCEPTION/ERROR procedures are activated when:

a) An OPEN statement is issued for a file already in the open status, or for a nonexistent file.

b) A file is in the OPEN status and the execution of a CLOSE statement is unsuccessful.

c) An I-O error occurs during execution of a READ, WRITE, REWRITE, START or DELETE statement.

After execution of the EXCEPTION/ERROR procedure, control is returned to the statement in the invoking routine following the statement which activated this procedure.

Within a USE procedure there must not be any reference to any non-declarative procedures. There is no interface between the two types. (However, a PERFORM statement may refer to a USE procedure.)

Within an EXCEPTION/ERROR procedure, no statement may be executed that would cause execution of a USE procedure that had been previously invoked and had not yet returned control to the invoking routine.

Note: EXCEPTION/ERROR procedures can be used to check the status key values whenever an input-output error occurs.

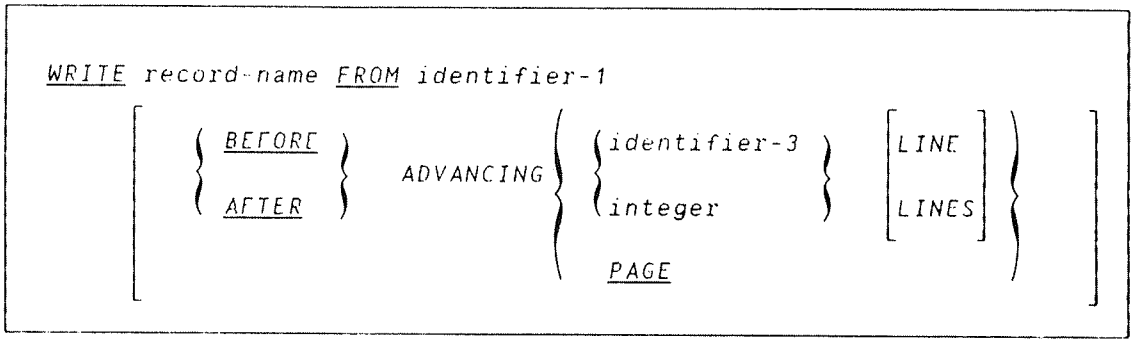
6.7.1.14 The WRITE Statement

The WRITE statement releases a logical record for an output or an input-output file. For Sequential I-O it can be used for vertical positioning of lines within a logical line.

Format 1. Indexed and Relative I-O:

```
WRITE record-name [FROM identifier-1][;
                                INVALID KEY imperative-statement]
```

Format 2. Sequential I-O:



Record-name and identifiers 1 or 2 must not reference the same storage area.

For format 1, record-name is the name of a logical record in the File Section of the Data Division and may be qualified.

With format 1, the INVALID KEY phrase must appear if no USE procedure is specified for the associated file.

For format 2, when identifier-3 is used in the ADVANCING phrase, it must be the name of an elementary data item (whose value may be zero). Integer may also be zero.

General Rules:

1) For Indexed and Relative I-O, the associated file must be open in the OUTPUT or I-O mode at the time of execution of this statement. For Sequential I-O the file must be open in either OUTPUT or EXTEND modes.

2) The results of the execution of the WRITE statement with the FROM phrase are equivalent to the execution of:

   a) The statement:

      MOVE identifier TO record-name

   according to the rules specified for the MOVE statement, followed by:

   b) The same *WRITE* statement without the *FROM* phrase.

   The contents of the record area prior to the execution of the implicit MOVE statement have no effect on the execution of this WRITE statement.

   After execution of the WRITE statement is complete, the information in the area referenced by identifier is available, even though the information in the area referenced by record-name may not be.

3) The current record pointer is unaffected by the execution of a WRITE statement.

4) The execution of the WRITE statement causes the value of the FILE STATUS data item (if any) associated with the file to be updated.

5) The maximum record size for a file is established at the time the file is created and must not subsequently be changed.

6) The number of character positions on a mass storage device required to store a logical record in a file may or may not be equal to the number of character positions defined by the logical description of that record in the program.

7) The execution of the WRITE statement releases a logical record to the operating system.

General Rules For Indexed I-O:

1) Execution of the WRITE statement causes the contents of the record area to be released. The runtime system utilizes the content of the record keys in such a way that subsequent access of the record key may be made based upon any of those specified record keys.

2) The value of the prime record key must be unique within the records in the file.

3) The data item specified as the prime record key must be set by the program to the desired value prior to the execution of the WRITE statement. (See general rule 2.)

4) If sequential access mode is specified for the file, records must be released to the runtime system in ascending order of prime record key values.

5) If random or dynamic access mode is specified, records may be released to the runtime system in any program-specified order.

6) When the ALTERNATE RECORD KEY clause is specified in the file control entry for an indexed file, the value of the alternate record key may be non-unique only if the DUPLICATES phrase is specified for that data item. In this case the runtime system provides storage of records such that when records are accessed sequentially, the order of retrieval of those records is the order in which they are released to the runtime system.

7) The INVALID KEY condition exists under the following circumstances:

   a) When the file is opened in the output or I-O mode, and the value of an alternate record key for which duplicates are not allowed equals the corresponding data item of a record already existing in the file, or

   b) When an attempt is made to write beyond the externally defined boundaries of the file.

8) When the INVALID KEY condition is recognized, the execution of the WRITE statement is unsuccessful, the contents of the record area are unaffected and the FILE STATUS data item, if any, associated with file-name of the associated file is set to a value indicating the cause of the condition. Execution of the program proceeds according to the rules given for the INVALID KEY condition.

General Rules for Relative I-O:

1) When a file is opened in the output mode, records may be placed into the file by one of the following:

   a) If the access mode is sequential, the WRITE statement will cause a record to be released to the runtime system. The first record will have a relative record number of one (1) and subsequent records released will have relative record numbers of 2, 3, 4 .... If the RELATIVE KEY data item has been specified in the file control entry for the associated file, the relative record number of the record just released will be placed into the RELATIVE KEY data item by the runtime system during execution of the WRITE statement.

   b) If the access mode is random or dynamic, prior to the execution of the WRITE statement the value of the RELATIVE KEY data item must be initialized in the program with the relative record number to be associated with the record in the record area. That record is then released to the runtime system by execution of the WRITE statement.

2) When a file is opened in the I-O mode and the access mode is random or dynamic, records are to be inserted in the associated file. The value of the RELATIVE KEY data item must be initialized by the program with the relative record number to be associated with the record in the record area. Execution of a WRITE statement then causes the contents of the record area to be released to the runtime system.

3) The INVALID KEY condition exists under the following circumstances:

   a) When the access mode is random or dynamic, and the RELATIVE KEY data item specifies a record which already exists in the file, or

   b) When an attempt is made to write beyond the externally defined boundaries of the file.

4) When the INVALID KEY condition is recognized, the execution of the WRITE statement is unsuccessful, the contents of the record area are unaffected, and the FILE STATUS data item, if any, of the associated file is set to a value indicating the cause of the condition. Execution of the program proceeds according to the rules given for the INVALID KEY condition.

General Rules For Sequential I-O:

1) The ADVANCING phrase allows control of the vertical positioning of each line on a printed page. If the ADVANCING phrase is not used, automatic advancing will act as if the user had specified AFTER ADVANCING 1 LINE. If the ADVANCING phrase is used, advancing is provided as follows:

   a) If identifier-3 is specified, the page is advanced the number of lines equal to the current value associated with identifier-3

   b) If integer is specified, the page is advanced the number of lines equal to the value of integer.

   c) If the BEFORE phrase is used, the line is presented before the page is advanced according to rules a and b above.

   d) If the AFTER phrase is used, the line is presented after the page is advanced according to rules a and b above.

   e) If PAGE is specified, the record is presented on the logical page before or after (depending on the phrase used) the device is repositioned to the next logical page.

   Note: Since the ND screen editors assume that files *end* with a carriage return/linefeed, this feature implies that the last line on a file written from COBOL without the BEFORE ADVANCING phrase will appear to be lost, while a blank line will appear at the beginning of the file.

2) When an attempt is made to write beyond the externally defined boundaries of a sequential file, an EXCEPTION condition exists and the contents of the record area are unaffected. The following action takes place:

   a) The value of the FILE STATUS data item, if any, of the associated file is set to a value indicating a boundary violation.

   b) If a USE AFTER STANDARD EXCEPTION declarative is explicitly or implicitly specified for the file, that declarative procedure will then be executed.

   c) If a USE AFTER STANDARD EXCEPTION declarative is not explicitly or implicitly specified for the file, the result is undefined.

EXAMPLE:

```
 1       IDENTIFICATION DIVISION.
 2       PROGRAM-ID.
 3           GENSEQ.
 4      ************************************************************************
 5      *    CREATES SQ-FILES AND LISTS.
 6      ************************************************************************
 7       ENVIRONMENT DIVISION.
 8       INPUT-OUTPUT SECTION.
 9       FILE-CONTROL.
10           SELECT SQ-FILE ASSIGN "COB1:DATA",
11               ORGANIZATION IS SEQUENTIAL,
12               ACCESS IS SEQUENTIAL.
13       DATA DIVISION.
14       FILE SECTION.
15       FD  SQ-FILE.
16       01  M-REC.
17           02          PIC X(10).
18           02 SEQNUM   PIC 9(5), BLANK WHEN ZERO.
19           02          PIC X(5).
20           02          PIC X(40).
21       WORKING-STORAGE SECTION.
22       01  RANDNO      PIC  9(4) PACKED-DECIMAL, VALUE ZERO.
23       01  MAXRAND     PIC S9(4) PACKED-DECIMAL, VALUE 1000.
24       01  NORECS      PIC  9(4) PACKED-DECIMAL.
25       01  RECCNT      PIC  99, COMP, VALUE 0.
26
27       PROCEDURE DIVISION.
28       INIT-01.
29           OPEN OUTPUT SQ-FILE.
30           DISPLAY "CREATE RECORDS?".
31           PERFORM GET-NORECS.
32           PERFORM CRE-SQ-FILE NORECS TIMES.
33       * BUILDS THE INPUT FILE.
34           CLOSE SQ-FILE.
35           DISPLAY "FILE SQ-FILE CREATED.", RECCNT, " RECORDS.".
36           OPEN INPUT SQ-FILE.
37       LIST-FILE-0.
38           MOVE 0 TO RECCNT.
39       LIST-FILE-1.
40           READ SQ-FILE AT END GO TO LIST-END.
41           ADD 1 TO RECCNT.
42           DISPLAY "REC ", RECCNT, ", SEQNUM = ", SEQNUM.
43           GO TO LIST-FILE-1.
44       LIST-END.
45           CLOSE SQ-FILE.
46           DISPLAY "JOB FINISH".
47           STOP RUN.
48       CRE-SQ-FILE.
49           CALL "RND" USING RANDNO, MAXRAND.
50           MOVE ALL "*" TO M-REC.
51           MOVE RANDNO TO SEQNUM.
52           ADD 1 TO RECCNT.
53           DISPLAY "WT REC = ", RECCNT, " KEY = ", SEQNUM.
54           WRITE M-REC.
55       GET-NORECS.
56           ACCEPT NORECS.
57           IF NORECS NOT NUMERIC,
58               DISPLAY "*** NOT NUMERIC DATA ***",
59               GO TO GET-NORECS
60           END-IF.
```

EXAMPLE:

```
 1       IDENTIFICATION DIVISION.
 2       PROGRAM-ID.
 3           GEN-ISAM-1.
 4      ****************************************************************
 5      * ISAM MEANS INDEX-SEQUENTIAL ACCESS METHOD.
 6      *
 7      * THE RECORDS ARE OUTPUT TO AN ISAM-FILE USING THE *UNIQUE*
 8      * (I. E., NOT DUPLICATED)DATA FOUND IN FIELD ISAM-KEY AS
 9      * *KEY* VALUE.
10      *
11      * BEFORE THIS JOB CAN BE RUN, THE FOLLOWING *MUST* BE MET:
12      *    A) FILE "ISAM-EX:DATA" AND "ISAM-EX:ISAM" MUST EXIST
13      *        AND BE CONSISTENT; OR
14      *    B) FILE "ISAM-EX:DATA" MUST NOT EXIST OR IF EXISTING
15      *        CONTAIN *NO DATA !!*
16      ****************************************************************
17       ENVIRONMENT DIVISION.
18       INPUT-OUTPUT SECTION.
19       FILE-CONTROL.
20           SELECT ISAM-FILE ASSIGN TO "ISAM-EX:DATA",
21               ORGANIZATION IS INDEXED,
22               ACCESS MODE IS DYNAMIC,
23               RECORD KEY IS ISAM-KEY,
24               FILE STATUS IS ISAMSTATUS.
25       DATA DIVISION.
26       FILE SECTION.
27       FD  ISAM-FILE
28               RECORD CONTAINS 46 CHARACTERS.
29       01  ISAM-REC.
30           02  ISAM-KEY PIC X(6).
31      *            :.....................MUST BE IN RECORD AREA!
32           02  ISAM-TEXT PIC X(40).
33
34       WORKING-STORAGE SECTION.
35       01  ISAMSTATUS PIC XX.
36      *            :.....................RETURN STATUS FROM ISAM.
37
38       PROCEDURE DIVISION.
39       A001.
40           OPEN I-O ISAM-FILE.
41       A002.
42           DISPLAY "ENTER KEY (MAX. 6 CHAR) : ",
43               ACCEPT ISAM-KEY.
44           IF ISAM-KEY = SPACES GO TO LIST.
45      *                 :.................SPACES INPUT, END DIALOG.
46           DISPLAY "ENTER TEXT (MAX 40 CHAR) : ",
47               ACCEPT ISAM-TEXT.
48      *              :..................READ RECORDS FROM TERMINAL.
49           WRITE ISAM-REC, INVALID KEY,
50               DISPLAY "ISAM FILE ERROR :", ISAMSTATUS, ":".
51           GO TO A002.
52      *        :........................OUTPUT RECORD AND ASK AGAIN.
53       LIST.
54           DISPLAY "ENTER ACCESS KEY: ",
55               ACCEPT ISAM-KEY.
56           IF ISAM-KEY = SPACES THEN GO TO FINI.
57           READ ISAM-FILE RECORD KEY IS ISAM-KEY INVALID KEY,
58               DISPLAY "RECORD NOT FOUND!",
59               GO TO LIST.
60           DISPLAY "REC: ", ISAM-KEY, ": ",ISAM-REC.
61           GO TO LIST.
62       FINI.
63           CLOSE ISAM-FILE.
64           DISPLAY "JOB END.".
65           STOP RUN.
```

EXAMPLE:

```
 1        IDENTIFICATION DIVISION.
 2        PROGRAM-ID.     GENRELATIVE.
 3       ****************************************************************
 4       * THIS PROGRAM SHOWS THE USAGE OF A RELATIVE FILE.
 5       *
 6       * THE FILE *MUST* EXIST BEFORE THE RUN, BUT MAY BE EMPTY, EACH
 7       * RECORD IS LOCATED RELATIVELY TO RECORD 1 IN THE FILE BY ITS
 8       * *INTEGER* KEY VALUE.
 9       *
10       * NOTE: EVEN IF SOMETHING IS WRITTEN ON RECORDS 300 AND 700
11       * IN THE RELATIVE FILE, *NO* STORAGE SPACE IS USED FOR THE
12       * "EMPTY" RECORDS BETWEEN RECORD 0 AND 299, OR BETWEEN 301 AND
13       * 699. THUS IT MAKES PERFECT SENSE TO USE, SAY, BIRTH DATES
14       * AS KEYS IN RELATIVE FILES.
15       ****************************************************************
16        ENVIRONMENT DIVISION.
17        INPUT-OUTPUT SECTION.
18        FILE-CONTROL.
19            SELECT RELFILE ASSIGN "RELATIVE-EX:DATA",
20                        ORGANIZATION IS RELATIVE,
21                        ACCESS IS DYNAMIC,
22                        RELATIVE KEY IS REL-KEY,
23                        FILE STATUS IS REL-STATUS.
24        DATA DIVISION.
25        FILE SECTION.
26
27        FD RELFILE
28                        BLOCK CONTAINS 10 RECORDS
29                        RECORD CONTAINS 60 CHARACTERS.
30        01  REL-RECORD  PIC X(60).
31
32        WORKING-STORAGE SECTION.
33        01  REL-STATUS  PIC XX.
34        01  REL-KEY     PIC 999.
35       ********.*****************************************************
36       *       :.........THE RELATIVE KEY CAN NOT OCCUR IN THE RECORD
37       *                 AREA. ITS POSSIBLE SIZES ARE 1-99999999999,
38       *                 BUT IT IS RESTRICTED TO 999 IN THIS PROGRAM.
39       ****************************************************************
40        PROCEDURE DIVISION.
41        A000.
42            OPEN I-O RELFILE.
43        A002.
44            DISPLAY "ENTER KEY (MAX 999) : ".
45            PERFORM GET-KEY.
46            IF REL-KEY = ZEROES GO TO A003.
47            DISPLAY "ENTER TEXT (MAX 60 CHARACTERS) : ".
48            ACCEPT REL-RECORD.
49            WRITE REL-RECORD INVALID KEY,
50                        DISPLAY "*** RELFILE ERROR *** :", REL-STATUS.
51            GO TO A002.
52        A003.
53            DISPLAY "ENTER ACCESS KEY: ".
54            PERFORM GET-KEY.
55            IF REL-KEY = ZEROS GO TO A999.
56            READ RELFILE RECORD INVALID KEY,
57                        DISPLAY "*** RECORD NOT FOUND ***",
58                        REL-STATUS, GO TO A003.
59            DISPLAY "REC :", REL-KEY, ": ", REL-RECORD.
60            GO TO A003.
61        A999.
62            CLOSE RELFILE.
63            DISPLAY "JOB END".
64            STOP RUN.
65        GET-KEY.
66            ACCEPT REL-KEY.
67            IF REL-KEY NOT NUMERIC,
68                        DISPLAY "*** KEY MUST BE NUMERIC ***",
69                        GO TO GET-KEY.
70        GET-KEY-EXIT.
71            EXIT.
```

6.8 PROCEDURE BRANCHING STATEMENTS

6.8.1 The ALTER Statement

Format:

```
ALTER procedure-name-1 TO [PROCEED TO]procedure-name-2
   [, procedure-name-3 TO [PROCEED TO]procedure-name-4] ...
```

This format is used to modify a simple GO TO statement elsewhere in the Procedure Division, thus changing the sequence of execution of program statements.

Each procedure-name-1, procedure-name-3,..., is the name of a COBOL paragraph that consists of a simple GO TO statement only.

Each procedure-name-2, procedure-name-4,..., is the name of a paragraph in the Procedure Divison.

The ALTER statement in effect replaces the former operand of that GO TO by procedure-name. Consider the ALTER statement in the context of the following program segment.

```
GATE.      GO TO MF-OPEN
MF-OPEN.   OPEN INPUT MASTER-FILE
           ALTER GATE TO PROCEED TO NORMAL
NORMAL.    READ MASTER-FILE, AT END GO TO EOF-MASTER
```

Examination of the above code reveals the technique of "shutting a gate", providing for a one-time, initializing-program step.

AVOID THE ALTER STATEMENT

The ALTER statement should not be used as it has a number of undesirable effects.

a) The object code produced will not be completely reentrant. depending on program structure. This could increase dramatically the memory reguirements during execution.

b) The source listing will not show any *obvious* changes and thus be more difficult to debug.

6.8.2 The CONTINUE Statement

Format:

```
CONTINUE
```

This statement has no effect and is treated as comments.

6.8.3 The EXIT Statement

The EXIT statement provides a common end point for a series of procedures.

Format 1:

```
EXIT
```

Format 2:

```
EXIT-DO
```

Format 3:

```
EXIT-ALL-DO
```

General Rules for Format 1:

1) An EXIT statement is used only when assigning a procedure-name to a given point in a program. Such an EXIT statement has no other effect on the compilation of the program.

2) An EXIT statement can be used to leave a DO --- END-DO loop.

General Rule for Formats 2 and 3:

1) The EXIT-DO statement in format 2 is used to leave the single DO-loop whithin which it appears. The EXIT-ALL-DO statement however, is used to leave all nested DO-loops within which it occurs. (See the DO-statement description, section 6.5.2.)

6.8.4 The GO TO Statement

The GO TO statement causes control to be transferred from one part of the Procedure Division to another.

Format 1:

```
GO TO [procedure-name-1]
```

Format 2:

```
GO TO procedure-name-1 [,procedure-name-2] ...,
   procedure-name-n DEPENDING ON identifier.
```

Identifier is the name of a numeric elementary item described without any positions to the right of the assumed decimal point.

When a paragraph is referenced by an ALTER statement, that paragraph can consist only of a paragraph header followed by a Format 1 GO TO statement.

A Format 1 GO TO statement, without procedure-name-1, can only appear in a single statement paragraph.

If a GO TO statement represented by Format 1 appears in a consecutive sequence of imperative statements within a sentence, it must appear as the last statement in that sequence.

General Rules:

1) When a GO TO statement, represented by Format 1 is executed, control is transferred to procedure-name-1 or to another procedure-name if the GO TO statement has been modified by an ALTER statement.

2) If procedure-name-1 is not specified in Format 1, an ALTER statement, referring to this GO TO statement, must be executed prior to the execution of this GO TO statement.

3) When a GO TO statement represented by Format 2 is executed, control is transferred to procedure-name-1 procedure-name-2, etc., depending on the value of the identifier being 1, 2, ..., n. If the value of the identifier is anything other than the positive or unsigned integers 1,2 ..., n, then no transfer occurs and control passes to the next statement in the normal sequence for execution.

4) Integer n must be in the range 1 to 100.

5) The maximum number of procedure-names that can be specified with a GO TO statement is 100.

6.8.5 The PERFORM Statement

The PERFORM statement permits the execution of a separate body of program steps. Three formats of the PERFORM statement are available:

Format 1:

```
PERFORM range [ { identifier-1 } TIMES ]
              [ { integer      }       ]
```

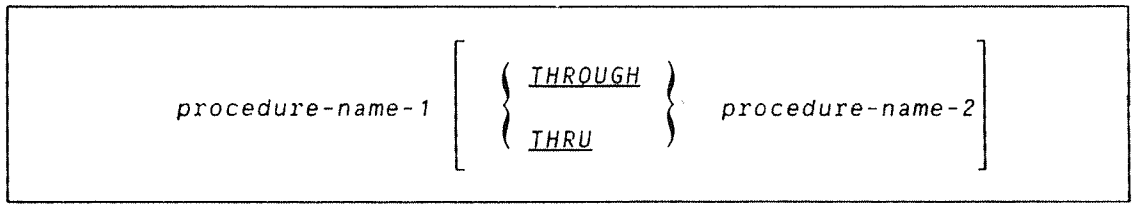
Format 2:

```
PERFORM range UNTIL condition-1
```

Format 3:

```
PERFORM range VARYING { identifier-5 } FROM { identifier-6 }
                     { index-name-3  }      { index-name-4 }
                                            { literal-3    }

                BY { identifier-7 } UNTIL condition-1
                   { literal-4    }

          [AFTER { identifier-8 } FROM { identifier-9 }
                 { index-name-5 }      { index-name-6 }
                                       { literal-5    }

                BY { identifier-10 } UNTIL condition-2
                   { literal-6     }

          [AFTER { identifier-11 } FROM { identifier-12 }
                 { index-name-8  }      { index-name-8  }
                                        { literal-6     }

                BY { identifier-13 } UNTIL condition-3 ] ]
                   { literal-7     }
```

where range is the construct:



and THROUGH is synonymous with THRU.

Procedure-names 1 and 2 must have a section or paragraph in the Procedure Division. Where both are specified, if either is a procedure-name inside Declaratives,then both must be procedure-names in Declaratives.

Each index-name identifies an index to be used in table references.

Each literal represents a numeric literal (in the BY phrase this must not be zero). Condition-names 1,2 and 3 may be any conditional expressions (see under Conditional Expressions'). Each identifier must name an elementary numeric item.

General Rules:

1) Whenever a PERFORM statement is executed, control is transferred to the first statement of the procedure named as procedure-1. Control is always returned to the statement following the PERFORM statement and the point from which it is returned is determined as follows:

   a) If procedure-name-1 is a pragraph name and a procedure-name-2 is not present, the return is made after the execution of the last statement of procedure-name-1.

   b) If procedure-name-1 is a section name and a procedure-name-2 is not present, the return is made after the execution of the last sentence of the last paragraph of procedure-name-1.

   c) If procedure-name-2 is present and it is a paragraph name, the return is made after the execution of the last statement of that paragraph.

   d) If procedure-name-2 is present and it is a section name, the return is made after the execution of the last sentence of the last paragraph in the section.

2) GO TO and PERFORM statements may be specified within the performed procedure. When the performed procedures include another PERFORM statement, the sequence of procedures associated with the embedded PERFORM statement must be included in or excluded from the performed procedures of the first PERFORM statement.

3) The *TIMES* option. Identifier-1, if used, must name an integer item. (If the integer is zero or negative when the PERFORM statement is initiated, control passes to the statement following the PERFORM statement.) The procedure(s) referred to are executed the number of times specified by the integer or the value in identifier-1. Once the PERFORM statement has been initiated, any reference to identifier-1 cannot vary the number of times the procedures are executed.

4) The *UNTIL* option. The procedures referred to are performed until the condition is satisfied. Control is then passed to the next executable statement following the PERFORM statement. If the condition is already true when the PERFORM statement is initiated, then the specified procedure(s) are not executed.

5) The *VARYING* option. This increments or decrements identifers or index-names until the condition(s) in the UNTIL option are satisfied, when control is passed to the next executable statement following the PERFORM statement.

6) With format 3, when varying two identifiers, the AFTER variable (identifier-8) is set to the value of identifier-9. When condition-1 is evaluated, if it is true, control is transferred to the next executable statement. If false, range is executed once before identifier-8 is augmented by identifier-10 or literal-6. And so on.

6.8.6 Using the PERFORM Statement

With format 1, the designated range is performed (i.e., executed remotely) a fixed number of times, as determined by an integer or by the value of an integral data-item.

In format 2, identifier-2 is set to the value of literal-1 or the current value of identifier-3 at the beginning of the execution. If condition-1 is false the designated range is performed and then condition-1 is evaluated again. The cycle is repeated (augmenting data-name-2 with the current BY value) until condition-1 is true.

In format 3 we may now vary not only the object of the VARYING phrase but objects of the AFTER phrases as well.

Varying two identifiers we have:

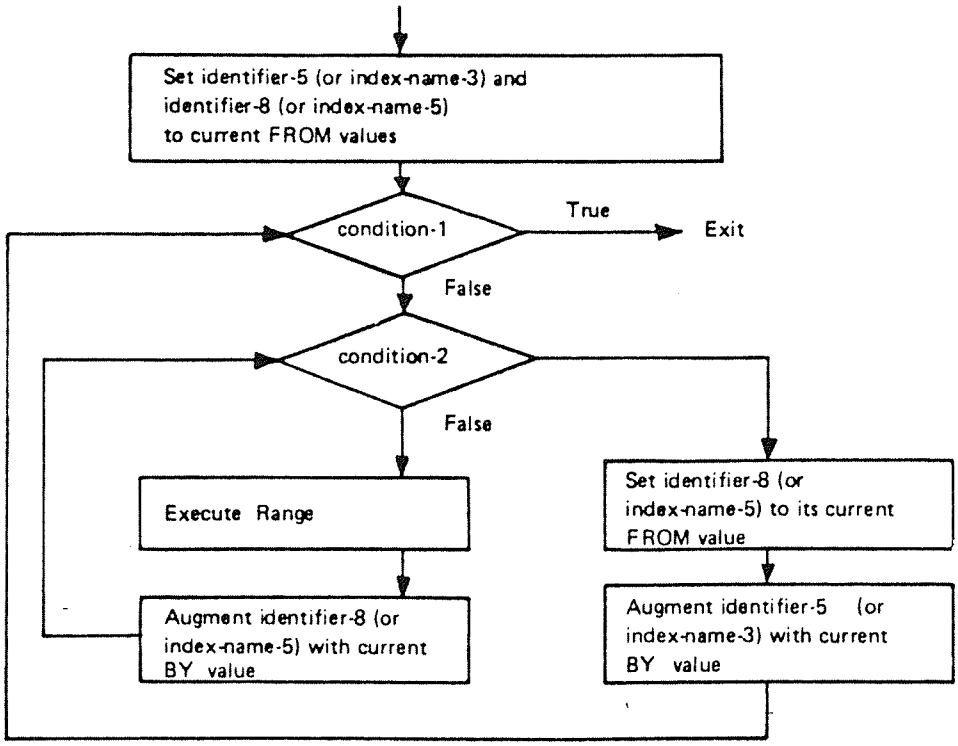


Figure 6.8

Varying three identifiers gives us:

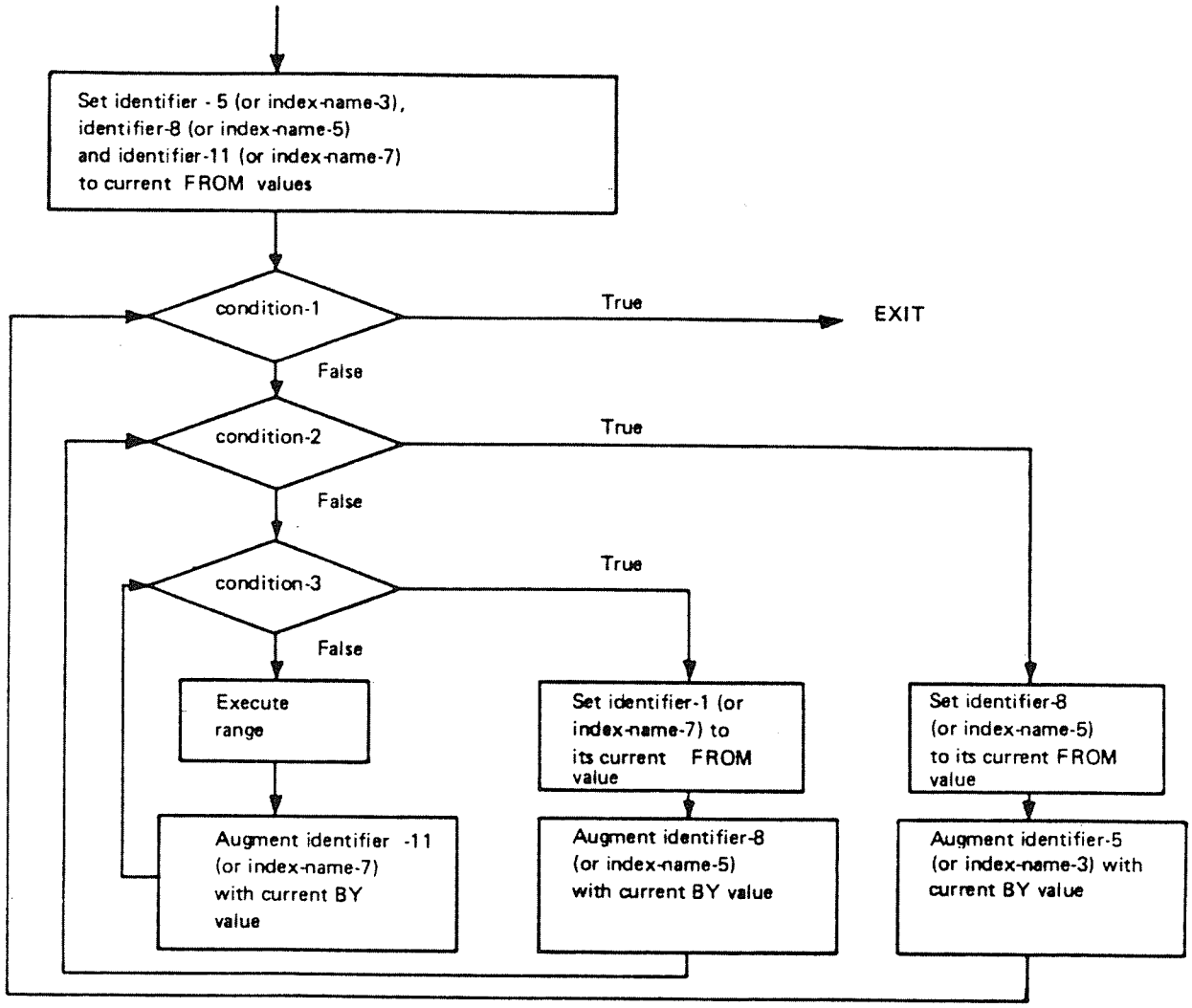


Figure 6.9

The format-3 PERFORM statement is particularly useful in table handling when one statement can search a whole three dimensional table.

At runtime, it is illegal to have concurrently active perform ranges whose terminus points are the same.

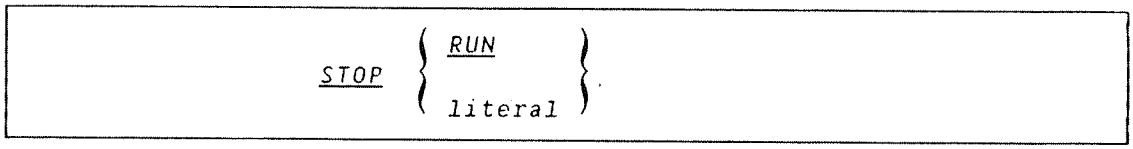
Example:

```
 1       IDENTIFICATION DIVISION.
 2       PROGRAM-ID.
 3           GENSEQ.
 4      *****************************************************************
 5      *    CREATES SQ-FILES AND LISTS.
 6      *****************************************************************
 7       ENVIRONMENT DIVISION.
 8       INPUT-OUTPUT SECTION.
 9       FILE-CONTROL.
10           SELECT SQ-FILE ASSIGN "COB1:DATA",
11               ORGANIZATION IS SEQUENTIAL,
12               ACCESS IS SEQUENTIAL.
13       DATA DIVISION.
14       FILE SECTION.
15       FD  SQ-FILE.
16       01  M-REC.
17           02          PIC X(10).
18           02 SEQNUM   PIC 9(5), BLANK WHEN ZERO.
19           02          PIC X(5).
20           02          PIC X(40).
21       WORKING-STORAGE SECTION.
22       01  RANDNO      PIC  9(4) PACKED-DECIMAL, VALUE ZERO.
23       01  MAXRAND     PIC S9(4) PACKED-DECIMAL, VALUE 1000.
24       01  NORECS      PIC  9(4) PACKED-DECIMAL.
25       01  RECCNT      PIC  99, COMP, VALUE 0.
26
27       PROCEDURE DIVISION.
28       INIT-01.
29           OPEN OUTPUT SQ-FILE.
30           DISPLAY "CREATE RECORDS?".
31           PERFORM GET-NORECS.
32           PERFORM CRE-SQ-FILE NORECS TIMES.
33       * BUILDS THE INPUT FILE.
34           CLOSE SQ-FILE.
35           DISPLAY "FILE SQ-FILE CREATED.", RECCNT, " RECORDS.".
36           OPEN INPUT SQ-FILE.
37       LIST-FILE-0.
38           MOVE 0 TO RECCNT.
39       LIST-FILE-1.
40           READ SQ-FILE AT END GO TO LIST-END.
41           ADD 1 TO RECCNT.
42           DISPLAY "REC ", RECCNT, ", SEQNUM = ", SEQNUM.
43           GO TO LIST-FILE-1.
44       LIST-END.
45           CLOSE SQ-FILE.
46           DISPLAY "JOB FINISH".
47           STOP RUN.
48       CRE-SQ-FILE.
49           CALL "RND" USING RANDNO, MAXRAND.
50           MOVE ALL "*" TO M-REC.
51           MOVE RANDNO TO SEQNUM.
52           ADD 1 TO RECCNT.
53           DISPLAY "WT REC = ", RECCNT, " KEY = ", SEQNUM.
54           WRITE M-REC.
55       GET-NORECS.
56           ACCEPT NORECS.
57           IF NORECS NOT NUMERIC,
58               DISPLAY "*** NOT NUMERIC DATA ***",
59               GO TO GET-NORECS
60           END-IF.
```

6.8.7 The STOP Statement

The STOP statement is used to terminate or delay execution of the object program.

Format:



STOP RUN terminates execution of a program, returning control to the operating system.

The form STOP literal causes the specified text to be displayed on the terminal, and execution to be suspended. Execution of the program is resumed only after operator intervention. The operator will probably perform a function suggested by the content of the literal, prior to resuming program execution.

The operator restarts the program by hitting the carriage return key.

Due caution must be taken when using this facility in MODE and BATCH jobs.

6.9 COMPILER DIRECTING STATEMENTS

6.9.1 The COPY Statement

Prewritten source programs can be included in a source program at compile time. These prewritten programs can be saved in user-created libraries without recoding, and incorporated later in the COBOL program by means of the COPY statement.

Format:

COPY file-name.

Where file-name is the name of a SINTRAN file. (Default file type is :SYMB.)

The COPY statement must be preceded by a space and terminated by a period. It may occur anywhere in the source program where a character string or separator may occur. However, a COPY statement must not occur within a COPY statement.

The effect of processing a COPY statement is that the library text associated with file-name is copied into the source program, logically replacing the entire COPY statement beginning with the word COPY and ending with the period.

7 SORT/MERGE

Sort/Merge enables the programmer to order one or more files of records, or to combine two or more identically ordered files of records, according to a set of user-specified keys contained within each of these records.

COBOL has special language features for sorting and merging so that the programmer does not need to program these operations in detail.

7.1 SORT CONCEPTS

Sort produces an ordered file from one or more files that may be completely unordered with regard to the sort sequence.

A COBOL program containing a sort may have one or more input files handled by an input procedure. Within this procedure a RELEASE statement (analogous to a WRITE statement) places records one at a time onto the sort file. When all the records have been placed on this file the sorting operation is executed. All the sort file records are now arranged in the sequence specified by the keys.

Upon completion of the sorting operation, individual records can be accessed, one at a time, through a RETURN statement, should they need to be modified. If the user does not want to modify the sorted records, the SORT statement's GIVING option names the sorted output file.

7.2 MERGE CONCEPTS

Merge produces an ordered file from two or more input files, each of which is already ordered in the merge sequence.

The COBOL program can contain any number of merge operations each of which can have independent output procedures. After merging, individual records can be accessed, for modification if required, by use of the RETURN statement. Otherwise, the GIVING option is used to name the merged output file. Sort/Merge handles fixed or variable length records.

The files specified in the USING and GIVING phrases of the Sort/Merge statement must be described in the FILE-CONTROL paragraph as having sequential organization. No I-O statement may be executed for the file named in the Sort/Merge file description.

7.3 SORT/MERGE - ENVIRONMENT DIVISION

File-control entries are required for each file to be used as input or output. A file-control entry is also required for the Sort/Merge file itself.

Format:

```
FILE-CONTROL. file-control entry [file-control entry ] ...
```

For the sort file, the format of the allowable clauses in the file-control entry is:

```
SELECT file-name ASSIGN TO assignment-name-1.
```

Each Sort/Merge file described in the Data Division must be named once and once only in a file-control entry.

The ASSIGN clause associates a Sort/Merge file with a storage medium.

7.4 SORT/MERGE - DATA DIVISION

In the File Section there must be FD entries for each I-O file together with a record description entry. For each Sort/Merge, file there must be an SD entry as well as a record description. The SD entry has the following format:

Format:

```
SD file-name

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

          [DEPENDING ON identifier] ]

[                                  { F         }  ]
; RECORDING MODE IS                { TEXT-FILE }
                                   { T         }
                                   { V         }  ]

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

Where the file-name must specify a Sort/Merge file.

The RECORD CONTAINS clause defines the size of the data records. As the size of each record is completely defined within the record description entry, this clause is never required. However, the number of characters in all fixed-length elementary items, plus the sum of the maximum number of those in any variable-length item subordinate to the record, determines its size.

The DATA RECORDS clause serves only as documentation for the names of the data records with their associated file.

Data-name-1 and data-name-2 are the names of data records which must have 01 level-number record descriptions, with the same names, associated with them. The presence of more than one data-name indicates that the file contains more than one type of data record which may be of differing sizes, formats etc.

7.5 SORT/MERGE - PROCEDURE DIVISION

A sort input procedure must contain a RELEASE statement to make each record available to the sorting operation. A Sort/Merge output operation must have a RETURN statement which makes a sorted/merged record available to the output procedure.

Format:

```
RELEASE record-name [ FROM identifier ]
```

Record-name must be the name of a logical record in the associated SD entry and may be qualified.

Record-name and identifier must not refer to the same storage area.

When the FROM option is used, the RELEASE statement is the equivalent of a MOVE statement operation of identifier to record-name, followed by a RELEASE statement operation for the record-name. Moving takes place according to the rules for the MOVE statement without the CORRESPONDING option. After the move, information in the record area is no longer available but that in the data area associated with the identifier may still be accessed.

When control passes from the Input Procedure, the sort file consists of all those records placed in it by execution of RELEASE statements.

The RETURN statement obtains records from the final phase of a sort or merge operation.

Format:

```
RETURN file-name RECORD [ INTO identifier ]
   ; AT END imperative-statement.
```

Within an Output Procedure at least one RETURN statement must be specified.

The file-name must be described by a Data Division SD entry.

The storage areas associated with the identifier and the record area of the file-name must not be the same.

The execution of a RETURN statement causes the next record, in the order specified by the keys listed in the SORT or MERGE statement, to become available by the Output Procedure. If more that one record description is associated with more than one file-name, these records share the same storage.

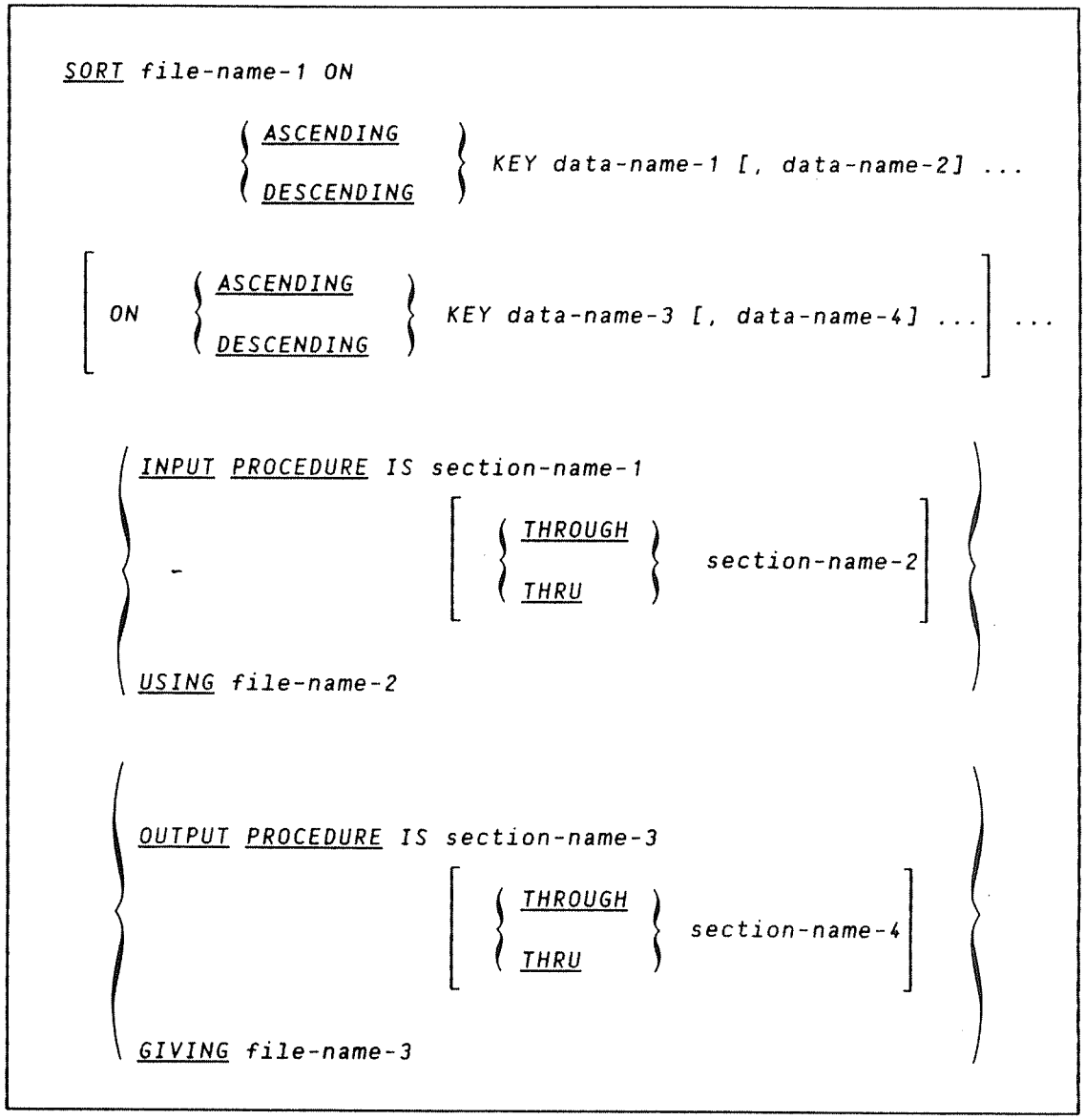
After execution of a RETURN statement, only the contents of the current record are available; if any data items lie beyond the length of the current record their contents are undefined.

After all the records have been returned from file-name, the AT END imperative statement is executed and no further RETURN statements may be executed as part of the current output procedure.

7.5.1 The SORT Statement

The SORT statement creates a sort file by executing input procedures, or by transferring records from another file. It then sorts records in the sort file on a set of specified keys. In its final phase it makes each record from this file available, in sorted order, to some output procedure or to an output file.

Format:



File-name-1 is the name given in the SD entry describing the records being sorted.

When the SORT statement is executed, all records contained on file-name-2 are sorted according to the specified keys. This input file must not be open at the time the SORT statement is executed; it is automatically opened and closed by the SORT operation (and implicit functions are also performed, if any).

The INPUT PROCEDURE option specifies one or more section-names of a procedure that is to modify input records before the sorting operation begins. Control is therefore passed to this procedure before file-name-1 is sequenced by the SORT statement. The compiler inserts a return mechanism at the end of the last section in the input procedure and when control passes the last statement of this procedure, the records that have been released to file-name-1 are sorted.

The input procedure must not contain any SORT statements or any transfer of control to points outside it. The execution of a CALL statement however follows standard linkage conventions.

7.5.2 Options Common to Sort and Merge

The ASCENDING/DESCENDING phrases specify that the records are to be processed in an ascending or descending sequence (whichever option is used) based on the specified sort keys.

The data items identified by KEY data-names must not contain an OCCURS clause or be subordinate to an entry containing an OCCURS clause.

Key data items must be of fixed length, they may be qualified but not subscripted or indexed.

If the USING phrase is specified, all records in file-name-2 for SORT (in file-names 2 and 3 for MERGE) are transferred automatically to file-name-1. At the time the Sort/Merge statements are executed, these files must not be open. The compiler makes code which opens, reads and makes records available, and closes files automatically.

The OUTPUT PROCEDURE option specifies one or more section-names of a procedure that will modify records from the sort or merge operation.

The procedure takes control when all records have been sorted/merged.

The compiler inserts a return mechanism at the end of the last section in the output procedure so that, when the last statement of this procedure has been passed, the return mechanism causes control to pass to the next executable statement following the SORT or MERGE statement.

The output procedure must not itself contain any Sort/Merge statements but it must include at least one RETURN statement to make the sorted/merged records available for processing.

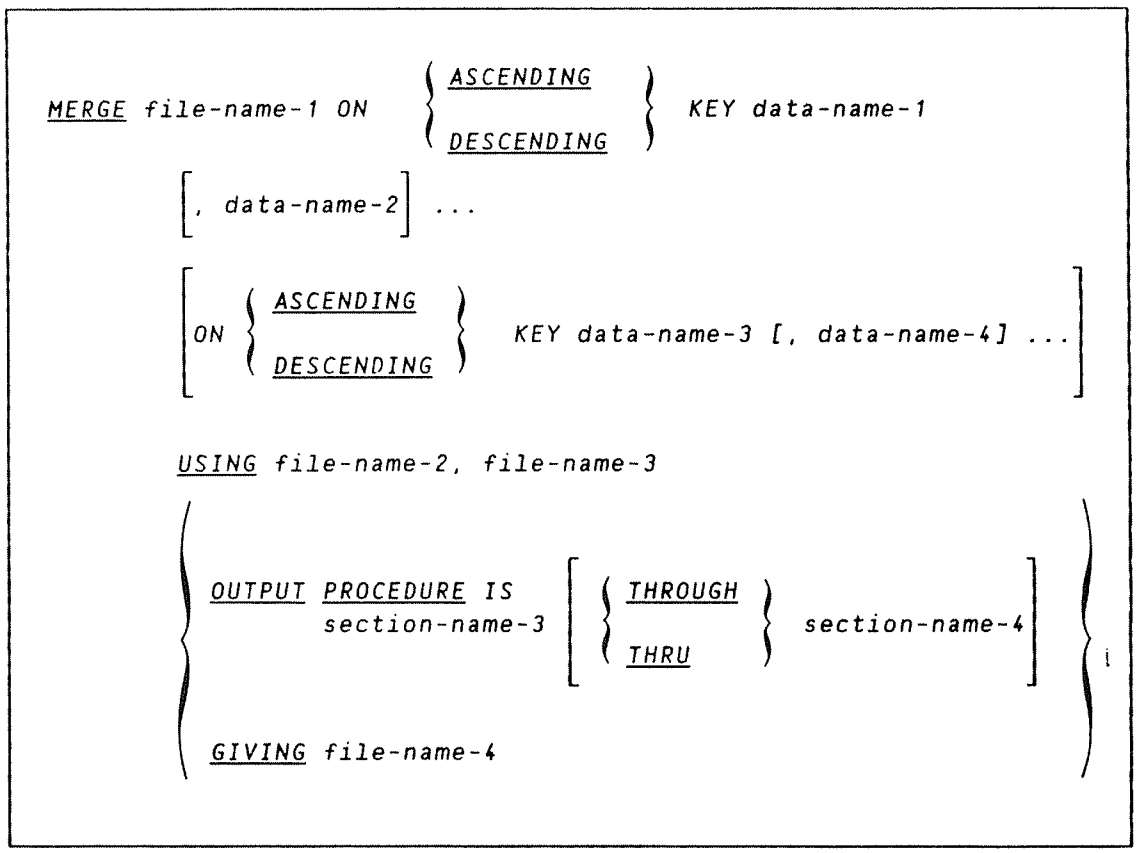
The GIVING phrase causes all the sorted/merged records to be transferred to the output file. (File-name-3 for SORT operations, file-name-4 for MERGE operations).

When the Sort/Merge statements are executed the output file must not be open. The compiler opens, reads and makes records available. The terminating function is performed as for a CLOSE statement.

7.5.3 The MERGE Statement

The MERGE statement combines two identically sequenced files according to a set of specified keys, and during the process makes records available in merge order, to an output file or procedure.

Format:



File-name-1 is the name given in the SD entry which describes the records being merged.

When the MERGE statement is executed, all records on file-names-2 and 3 are merged according to the key(s) specified. These files must not be open when this statement is executed; they are automatically opened and closed by the MERGE operation.

For the statement options, refer to the previous section on the SORT statement.

Example:

```
 1        IDENTIFICATION DIVISION.
 2        PROGRAM-ID.
 3            SORT-EX1.
 4       ************************************************************************
 5       * CREATES IN-FILE, LISTS, SORTS, CREATING OUT-FILE, LISTS.
 6       ************************************************************************
 7        ENVIRONMENT DIVISION.
 8        INPUT-OUTPUT SECTION.
 9        FILE-CONTROL.
10            SELECT IN-FILE ASSIGN "COB1:DATA",
11                    ORGANIZATION IS SEQUENTIAL,
12                    ACCESS IS SEQUENTIAL.
13            SELECT OUT-FILE ASSIGN "COB2:DATA".
14            SELECT S-FILE ASSIGN "SORT:DATA".
15
16        DATA DIVISION.
17        FILE SECTION.
18       ************************************************************************
19       * NOTE: ALL FILES, INPUT/OUTPUT AND SORT CAN BE FIXED (F),
20       * VARIABLE (V) OR TEXT (T), EXCEPT RELATIVE AND INDEXED
21       * THAT CAN'T BE TEXT.
22       ************************************************************************
23        FD  IN-FILE.
24        01  M-REC                PIC X(60).
25        01  MOD-REC.
26            02                   PIC X(10).
27            02 SEQNUM            PIC 9(5).
28            02                   PIC X(5).
29            02                   PIC X(40).
30        FD  OUT-FILE.
31        01  N-REC.
32            02                   PIC X(10).
33            02  SEQNUM2          PIC 9(5).
34            02                   PIC X(5).
35            02                   PIC X(40).
36       ************************************************************************
37       * NOTE THAT THE SORT DESCRIPTOR (SD) DOES NOT NEED ANY
38       * FILE-DESCRIPTION-ENTRY, IF RECORDING MODE T OR V
39       * IS NOT USED.
40       ************************************************************************
41        SD  S-FILE.
42        01  S-REC.
43            02                   PIC X(10).
44            02  S-KEY            PIC 9(5).
45            02                   PIC X(5).
46            02                   PIC X(40).
47
48        WORKING-STORAGE SECTION.
49        01  RANDNO               PIC 9(5), PACKED-DECIMAL, VALUE ZERO.
50        01  MAXRAND              PIC S9(5), PACKED-DECIMAL, VALUE 1000.
51        01  NORECS               PIC 9(5).
52        01  RECCNT               PIC 9(5), PACKED-DECIMAL, VALUE 0.
53
54        PROCEDURE DIVISION.
55
56        MAIN SECTION.
57        INIT-01.
58            OPEN OUTPUT IN-FILE.
59            DISPLAY "CREATE NO RECORDS? (<9999 LEAD 0, S)".
60            PERFORM GET-NORECS.
61            MOVE 0 TO RECCNT.
62            PERFORM CRE-IN-FILE NORECS TIMES.
63       ************************************************************************
64       * BUILDING THE INPUT FILE FOR SORT.
65       ************************************************************************
66            CLOSE IN-FILE.
67            DISPLAY "FILE IN-FILE CREATED.", RECCNT, " RECORDS.".
68            MOVE 0 TO RECCNT.
69       ************************************************************************
70       * ALL FILES REFERRED TO BY THE SORT VERB MUST BE CLOSED
71       * BEFORE THE SORT IS STARTED, OTHERWISE A RUNTIME ERROR OCCURS.
72       ************************************************************************
73            SORT S-FILE ON ASCENDING KEY S-KEY,
```

```
                USING IN-FILE,
                GIVING OUT-FILE.
            OPEN INPUT OUT-FILE.
            PERFORM LIST-OUT-FILE.
            CLOSE OUT-FILE.
            DISPLAY "JOB FINISH".
            STOP RUN.
        CRE-IN-FILE SECTION.
        CRE-FILE-1.
            CALL "RND" USING RANDNO, MAXRAND.
            MOVE ALL "*" TO M-REC.
            MOVE RANDNO TO SEQNUM.
            ADD 1 TO RECCNT.
            DISPLAY "UT REC = ", RECCNT, " KEY = ", SEQNUM.
            WRITE M-REC.
        CRE-FILE-END.
            EXIT.
        LIST-OUT-FILE SECTION.
        LIST-FILE-0.
            MOVE 0 TO RECCNT.
        LIST-FILE-1.
            READ OUT-FILE AT END GO TO LIST-END.
            ADD 1 TO RECCNT.
            DISPLAY "REC NO. ", RECCNT, ", SEQNUM = ", SEQNUM2.
            GO TO LIST-FILE-1.
        LIST-END.
            EXIT.
        GET-NORECS SECTION.
        GET-NO.
            ACCEPT NORECS.
            IF NORECS NOT NUMERIC,
                            DISPLAY "** NOT NUMERIC DATA",
                            GO TO GET-NO.
        GET-EXIT.
            EXIT.
```

Running the program (on an ND-500) gives the following output to screen:

```
@nd (cobol-exam)c-7-8

CREATE NO RECORDS? (<9999 LEAD 0, S)20
UT REC = 00001 KEY = 00403
UT REC = 00002 KEY = 00976
UT REC = 00003 KEY = 00019
UT REC = 00004 KEY = 00832
UT REC = 00005 KEY = 00715
UT REC = 00006 KEY = 00968
UT REC = 00007 KEY = 00891
UT REC = 00008 KEY = 00784
UT REC = 00009 KEY = 00947
UT REC = 00010 KEY = 00680
UT REC = 00011 KEY = 00283
UT REC = 00012 KEY = 00056
UT REC = 00013 KEY = 00299
UT REC = 00014 KEY = 00312
UT REC = 00015 KEY = 00395
UT REC = 00016 KEY = 00848
UT REC = 00017 KEY = 00971
UT REC = 00018 KEY = 00064
UT REC = 00019 KEY = 00427
UT REC = 00020 KEY = 00360
FILE IN-FILE CREATED.00020 RECORDS.
REC NO. 00001, SEQNUM = 00019
REC NO. 00002, SEQNUM = 00056
REC NO. 00003, SEQNUM = 00064
REC NO. 00004, SEQNUM = 00283
REC NO. 00005, SEQNUM = 00299
REC NO. 00006, SEQNUM = 00312
REC NO. 00007, SEQNUM = 00360
REC NO. 00008, SEQNUM = 00395
REC NO. 00009, SEQNUM = 00403
REC NO. 00010, SEQNUM = 00427
REC NO. 00011, SEQNUM = 00680
REC NO. 00012, SEQNUM = 00715
REC NO. 00013, SEQNUM = 00784
REC NO. 00014, SEQNUM = 00832
REC NO. 00015, SEQNUM = 00848
REC NO. 00016, SEQNUM = 00891
REC NO. 00017, SEQNUM = 00947
REC NO. 00018, SEQNUM = 00968
REC NO. 00019, SEQNUM = 00971
REC NO. 00020, SEQNUM = 00976
JOB FINISH
```

Example:

```
 1        IDENTIFICATION DIVISION.
 2        PROGRAM-ID.
 3            SORT-EX2.
 4       ************************************************************************
 5       * CREATES IN-FILE, LISTS, SORTS USING PROCEDURES, CREATING OUT-FILE.
 6       ************************************************************************
 7        ENVIRONMENT DIVISION.
 8        INPUT-OUTPUT SECTION.
 9        FILE-CONTROL.
10            SELECT IN-FILE ASSIGN "COB1:DATA",
11                    FILE STATUS IS IN-FILE-STATUS.
12            SELECT OUT-FILE ASSIGN "COB2:DATA",
13                    FILE STATUS IS OUT-FILE-STATUS.
14            SELECT S-FILE ASSIGN "SORT:DATA",
15                    FILE STATUS IS S-FILE-STATUS.
16        DATA DIVISION.
17        FILE SECTION.
18        FD  IN-FILE.
19        01  IN-REC.
20            02          PIC X(10).
21            02 SEQNUM   PIC 9(5).
22            02          PIC X(5).
23            02          PIC X(80).
24        FD  OUT-FILE.
25        01  UT-REC.
26            02          PIC X(10).
27            02 SEQNUM2  PIC 9(5).
28            02          PIC X(5).
29            02          PIC X(80).
30        SD  S-FILE.
31        01  S-REC.
32            02          PIC X(10).
33            02 S-KEY    PIC 9(5).
34            02          PIC X(5).
35            02          PIC X(80).
36        WORKING-STORAGE SECTION.
37        01  NORECS      PIC 9(4).
38        01  RECCNT      PACKED-DECIMAL, VALUE 0.
39       ************************************************************************
40       * PARAMETERS FOR CALL TO RND (A RANDOM NUMBER GENERATOR).
41       ************************************************************************
42        01  RANDNO      PIC 9(4), PACKED-DECIMAL, VALUE ZERO.
43        01  MAXRAND     PIC S9(4), PACKED-DECIMAL, VALUE 1000.
44       ************************************************************************
45       * STATUS DATA-NAME(S) MUST BE DEFINED AND MUST BE 2 BYTES LONG.
46       ************************************************************************
47        01  IN-FILE-STATUS  PIC XX.
48        01  OUT-FILE-STATUS  PIC XX.
49        01  S-FILE-STATUS   PIC XX.
50       ************************************************************************
51       * START/END-TIME USED FOR ACCEPTING TIME FROM SYSTEM.
52       ************************************************************************
53        01  START-TIME      PIC 9(8).
54        01  END-TIME        PIC 9(8).
55       ************************************************************************
56       * SORT-START/END ARE THE RECEIVING EDIT FIELDS FOR START/END-TIME
57       ************************************************************************
58        01  SORT-START      PIC 99,99,99,99.
59        01  SORT-END        PIC 99,99,99,99.
60
61        PROCEDURE DIVISION.
62        MAIN SECTION.
63        INIT-01.
64            OPEN OUTPUT IN-FILE.
65            DISPLAY "CREATE NO RECORDS?".
66            PERFORM GET-NORECS.
67            MOVE 0 TO RECCNT.
68            PERFORM CRE-IN-FILE NORECS TIMES.
69            CLOSE IN-FILE.
70            DISPLAY "IN-FILE CREATED. ", RECCNT, " RECORDS.".
71            MOVE 0 TO RECCNT.
72            ACCEPT START-TIME FROM TIME.
73            SORT S-FILE ON ASCENDING KEY S-KEY,
```

```
74                INPUT PROCEDURE IS SORT-PROC-IN
75                OUTPUT PROCEDURE IS SORT-PROC-UT.
76           ACCEPT END-TIME FROM TIME.
77           PERFORM SORT-TIMES.
78           DISPLAY "JOB FINISH".
79           STOP RUN.
80       MAIN-END.
81           EXIT.
82      ****************************************************************
83      * CALLING "RND" TO GENERATE RANDOM DATA FOR THE RECORD, WRITES.
84      ****************************************************************
85       CRE-IN-FILE SECTION.
86       CRE-FILE-1.
87           CALL "RND" USING RANDNO, MAXRAND.
88           MOVE RANDNO TO SEQNUM.
89           ADD 1 TO RECCNT.
90           DISPLAY "UT REC = ", RECCNT, ", KEY = ", SEQNUM.
91           WRITE IN-REC.
92       CRE-FILE-END.
93           EXIT.
94      ****************************************************************
95      * MOVE SORT TIMES INTO EDIT FIELDS FOR DISPLAYING
96      ****************************************************************
97       SORT-TIMES SECTION.
98       SORTT.
99           MOVE START-TIME TO SORT-START.
100          MOVE END-TIME TO SORT-END.
101          DISPLAY "START SORT AT : ", SORT-START.
102          DISPLAY "END SORT AT : ", SORT-END.
103      SORT-TIMES-END.
104          EXIT.
105     ****************************************************************
106     * CALLED ONLY FROM THE SORT VERB TO READ AND PASS RECORDS
107     * FROM THE IN-FILE INTO THE SORT.
108     * IN-FILE IS OPENED/READ/CLOSED WITHIN THE ROUTINE.
109     ****************************************************************
110      SORT-PROC-IN SECTION.
111      SORTIN.
112          DISPLAY ":::::::::::::::: SORT-PROC-IN START ::::::::::::::::".
113          OPEN INPUT IN-FILE.
114      SORTIN-1.
115          READ IN-FILE AT END GO TO SORT-IN-END.
116     *================================================================
117     * THE RELEASE PASSES THE RECORD INTO THE SORT.
118     *================================================================
119          RELEASE S-REC FROM IN-REC.
120          GO TO SORTIN-1.
121      SORT-IN-END.
122          CLOSE IN-FILE.
123          DISPLAY ":::::::::::::::: SORT-PROC-IN END ::::::::::::::::".
124      SORT-IN-FINI.
125          EXIT.
126     ****************************************************************
127     * CALL ONLY FROM THE SORT VERB TO ACCEPT RECORDS AND WRITE
128     * THEM ONTO OUT-FILE.
129     * OUT-FILE IS OPENED/WRITTEN/CLOSED WITHIN THE ROUTINE.
130     ****************************************************************
131      SORT-PROC-UT SECTION.
132      SORTUT.
133          DISPLAY "************ SORT-PROC-UT START *****************".
134          MOVE 1 TO RECCNT.
135          OPEN OUTPUT OUT-FILE.
136      SORTUT1.
137     *================================================================
138     * RETURN PASSES A SORTED RECORD FROM THE SORT INTO THE PROGRAM.
139     *================================================================
140          RETURN S-FILE INTO UT-REC, AT END,
141                  DISPLAY "SFILE-ERR, STATUS :=:", S-FILE-STATUS,
142                              GO TO SORTUT-END.
143          DISPLAY "REC ", RECCNT, " SEQNUM = ", SEQNUM2,
144                  "STATUS = ", OUT-FILE-STATUS.
145          WRITE UT-REC.
146          ADD 1 TO RECCNT.
147          GO TO SORTUT1.
148      SORTUT-END.
149          CLOSE OUT-FILE.
150          DISPLAY "************ SORT-PROC-UT ENDED *****************".
```

```
151         SORT-FINI.
152             EXIT.
153
154         GET-NORECS SECTION.
155         GET-NO.
156             ACCEPT NORECS.
157             IF NORECS NOT NUMERIC,
158                     DISPLAY "** NOT NUMERIC DATA",
159                     GO TO GET-NO.
160         GET-EXIT.
161             EXIT.
```

Running this example (on an ND-100) gives the following output:

```
@(cobol-exam)c-7-10

CREATE NO RECORDS?20
UT REC = 00001+, KEY = 00403
UT REC = 00002+, KEY = 00976
UT REC = 00003+, KEY = 00019
UT REC = 00004+, KEY = 00832
UT REC = 00005+, KEY = 00715
UT REC = 00006+, KEY = 00968
UT REC = 00007+, KEY = 00891
UT REC = 00008+, KEY = 00784
UT REC = 00009+, KEY = 00947
UT REC = 00010+, KEY = 00680
UT REC = 00011+, KEY = 00283
UT REC = 00012+, KEY = 00056
UT REC = 00013+, KEY = 00299
UT REC = 00014+, KEY = 00312
UT REC = 00015+, KEY = 00395
UT REC = 00016+, KEY = 00848
UT REC = 00017+, KEY = 00971
UT REC = 00018+, KEY = 00064
UT REC = 00019+, KEY = 00427
UT REC = 00020+, KEY = 00360
IN-FILE CREATED. 00020+ RECORDS.
:::::::::::::: SORT-PROC-IN START ::::::::::::::::
:::::::::::::: SORT-PROC-IN END ::::::::::::::::
************ SORT-PROC-UT START ******************
REC 00001+ SEQNUM = 00019STATUS = 00
REC 00002+ SEQNUM = 00056STATUS = 00
REC 00003+ SEQNUM = 00064STATUS = 00
REC 00004+ SEQNUM = 00283STATUS = 00
REC 00005+ SEQNUM = 00299STATUS = 00
REC 00006+ SEQNUM = 00312STATUS = 00
REC 00007+ SEQNUM = 00360STATUS = 00
REC 00008+ SEQNUM = 00395STATUS = 00
REC 00009+ SEQNUM = 00403STATUS = 00
REC 00010+ SEQNUM = 00427STATUS = 00
REC 00011+ SEQNUM = 00680STATUS = 00
REC 00012+ SEQNUM = 00715STATUS = 00
REC 00013+ SEQNUM = 00784STATUS = 00
REC 00014+ SEQNUM = 00832STATUS = 00
REC 00015+ SEQNUM = 00848STATUS = 00
REC 00016+ SEQNUM = 00891STATUS = 00
REC 00017+ SEQNUM = 00947STATUS = 00
REC 00018+ SEQNUM = 00968STATUS = 00
REC 00019+ SEQNUM = 00971STATUS = 00
REC 00020+ SEQNUM = 00976STATUS = 00
SFILE-ERR, STATUS :=:10
************ SORT-PROC-UT ENDED ******************
START SORT AT : 11,06,00,84
END SORT AT : 11,06,02,12
JOB FINISH
```

8 TABLE HANDLING

A table is a set of contiguous data items having the same data description.

Tables of data are common components of business data processing tasks. Although items of data that make up a table could be described as contiguous data items, there are two reasons why this approach is not satisfactory. First, from a documentation standpoint, the underlying homogeneity of the items would not be readily apparent; and second, it would be unnecessarily difficult to retrieve an element from a table when executing the program.

In COBOL, a table is defined with an OCCURS clause in its data description entry. This clause specifies that the named item is to be repeated as many times as stated. The item so named is considered to be a table element and its name and description apply to each repetition (or occurrence) of the item. Since the occurrences do not have unique data-names, reference to a particular occurrence can only be made by giving the data-names of the table element, together with the occurrence number of the required item within the element.

The occurrence number is known as a subscript and the method of supplying this number for individual table elements is called subscripting. A related technique for table referencing is called indexing and both of these methods of specifying occurrence numbers are described in this section.

8.1 TABLE DEFINITION

COBOL allows tables in one, two, or three dimensions.

To define a one-dimensional table, the programmer uses an OCCURS clause as part of the data description of the table element, but the OCCURS clause must not appear in the description of group items which contain the table element.

Example:

```
01 TABLE-1.
   02 ELEMENT-1 OCCURS 20 TIMES.
      03 ELEMENT-A PIC X (2).
      03 ELEMENT-B PIC 9 (5).
```

TABLE-1 is the group element containing the table. ELEMENT-1 names a table element of a one-dimensional table which occurs 20 times. ELEMENT-A and ELEMENT-B are elementary items.

Defining a one-dimensional table within each occurrence of an element of another one-dimensional table gives rise to a two-dimensional table. To define a two-dimensional table, therefore, an OCCURS clause must appear in the data description of the element of the table, and in the description of only one group item which contains that table element.

Example:

```
01 TABLE-2.
   02 ELEMENT-1 OCCURS 5 TIMES.
      03 ELEMENT-2 OCCURS 4 TIMES.
         04 ELEMENT-A PIC 9(10).
         04 ELEMENT-B PIC X(5).
```

ELEMENT-1 is an element of a one-dimensional table occurring five times. ELEMENT-2 is an element of a two-dimensional table occurring four times within each occurrence of ELEMENT-1.

To define a three-dimensional table, the OCCURS clause should appear in the data description of the element of the table and in the description of 2 group items which contain the element.

Example:

```
01 CENSUS TABLE.
   05 CONTINENT-TABLE OCCURS 6 TIMES.
      10 CONTINENT-NAME PIC X(9).
      10 COUNTRY-TABLE OCCURS 5 TIMES.
         15 COUNTRY-NAME PIC X(12).
         15 CITY-TABLE OCCURS 100 TIMES.
            20 CITY-NAME PIC X(4).
            20 CITY-POPULATION PIC X(5).
```

In the above example we have a table of one dimension for CONTINENT-NAME, two dimensions for COUNTRY-NAME and three dimensions for CITY-NAME and CITY-POPULATION.

8.1.1 Table References

Whenever the user refers to a table element, the reference must indicate which occurrence of the element is intended. For access to a one-dimensional table, the occurrence number of the desired element provides complete information. For access to tables of more than one dimension, an occurrence number must be supplied for each dimension of the table accessed. In the last example then, a reference to the 4th CONTINENT-NAME would be complete, whereas a reference to the 4th COUNTRY-NAME would not. To refer to COUNTRY-NAME, which is an element of a two-dimensional table, the user must refer, for example, to the 4th COUNTRY-NAME within the 6th CONTINENT-TABLE.

One method by which occurrence numbers may be specified is to append one or more subscripts to the data-name. A subscript is an integer whose value specifies the occurrence number of an element. The subscript can be represented either by a literal which is an integer, or by a data-name which is defined elsewhere as a numeric elementary item with no character positions to the right of the assumed decimal point. In either case, the subscript, enclosed in parentheses, is written immediately following the name of the table element. A table reference must include as many subscripts as there are dimensions in the table whose element is being referenced. That is, there must be a subscript for each OCCURS clause in the hierarchy containing the data-name, including the data-name itself. In the example, references to CONTINENT-NAME require only one subscript, reference to COUNTRY-NAME requires two, references to CITY-NAME require two, and references to CITY-NAME and CITY-POPULATION require three.

When more than one subscript is required, they are written in order of successively less inclusive dimensions of the data organization. When a data-name is used as a subscript, it may be used to refer to items in many different tables. These tables need not have elements of the same size. The data-name may also appear as the only subscript with one item and as one of two or three subscripts with another item. Also, it is permissible to mix literal and data-name subscripts, for example: CITY-POPULATION(4, NEWKEY, 42).

8.1.1.1 Subscripting

Subscripting is the method of providing table references using subscripts. A subscript is an integer value specifying the occurrence number of a table element.

Format:

```
{ data-name      }
{ condition-name }

          (subscript-1 [,subscript-2 [, subscript-3 ]])
```

Subscripts can only be used when referring to an individual item within a table element.

Data-name must be the name of a table element and may be qualified.

The subscript can be represented by a literal or a data-name.

If a literal, a subscript must be an integer having a value 1. It must not be negative.

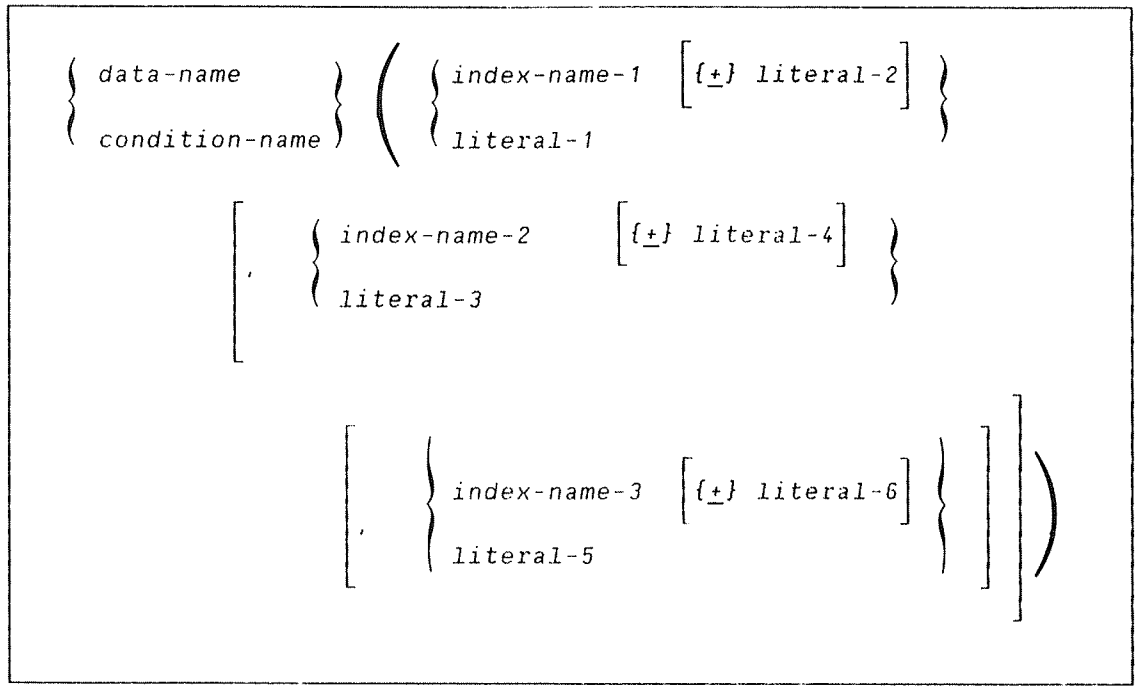
If a data-name, a subscript must be defined as elementary numeric integer.

Where more than one subscript is required, they are written in the order of successively less-inclusive tables. Each subscript must be separated from the next by either a space, or a comma followed by a space. (The comma is not required.)

8.1.1.2 Indexing

Another method of referring to items in a table is indexing. An *index* is used to store table element occurrence numbers; the index contains a displacement value from the beginning of the table element (equivalent to an occurrence number).

Format:



The index-name is specified through the OCCURS clause. It must be initialized (i. e., must have a value assigned to it) before use.

In direct indexing, the index-name is in the form of a subscript. The value contained in the index is calculated as the occurrence number minus one, multiplied by the length of the individual table entry.

For example:

```
03 ELEMENT OCCURS 20 INDEXED BY INDX-1 PIC X(2).
```

The tenth occurrence of ELEMENT generates a value in INDX-1 of (10-1) * 2 = 18.

With *relative indexing*, the index-name is followed by a space, followed by a + or -, followed by another space, followed by an unsigned literal. The literal (i.e., occurrence number) is converted to an index value before being added to or subtracted from the index-name index.

For example, if we have:

```
01 TABLE-3.
   02 ELEMENT-1 OCCURS 2 TIMES INDEXED BY INDX-1.
      03 ELEMENT-2 OCCURS 3 TIMES INDEXED BY INDX-2.
         04 ELEMENT-3 OCCURS 2 TIMES INDEXED BY INDX-3 PIC X(5).
```

then, each occurrence of ELEMENT-1 is 30 characters in length (3*2*5). Each occurrence of ELEMENT-2 is 10 characters in length (2*5) and each occurrence of ELEMENT-3 is 5 characters in length.

A reference using relative indexing such as

```
ELEMENT-3 (INDX-1 + 1, INDX-2 - 1, INDX-3 + 2)
```

would produce the computation for the displacement of:

```
  (address of ELEMENT-3)
+ ((contents of INDX-1)+1-1)*30
+ ((contents of INDX-2)-1-1)*10
+ ((contents of INDX-3)+2-1)*5
```

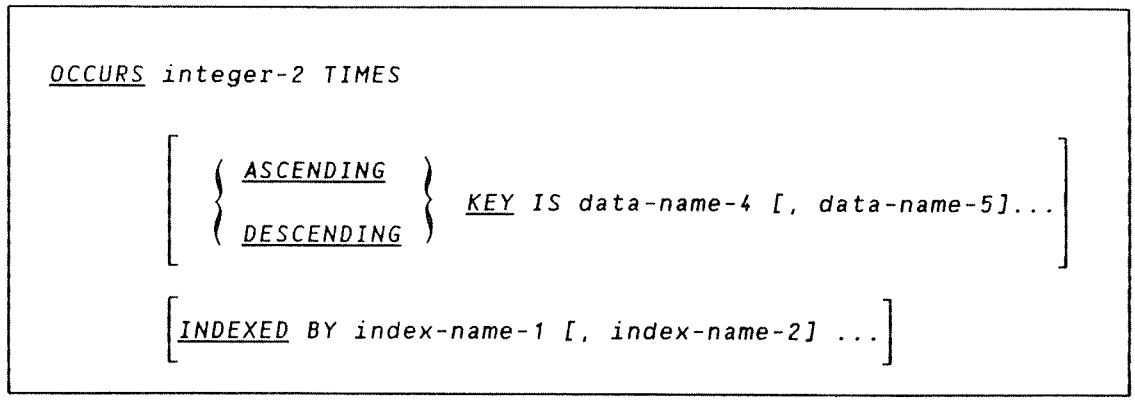
8.2 TABLE HANDLING - DATA DIVISION

The clauses used for Table Handling are OCCURS and USAGE IS INDEX.

8.2.1 The OCCURS Clause

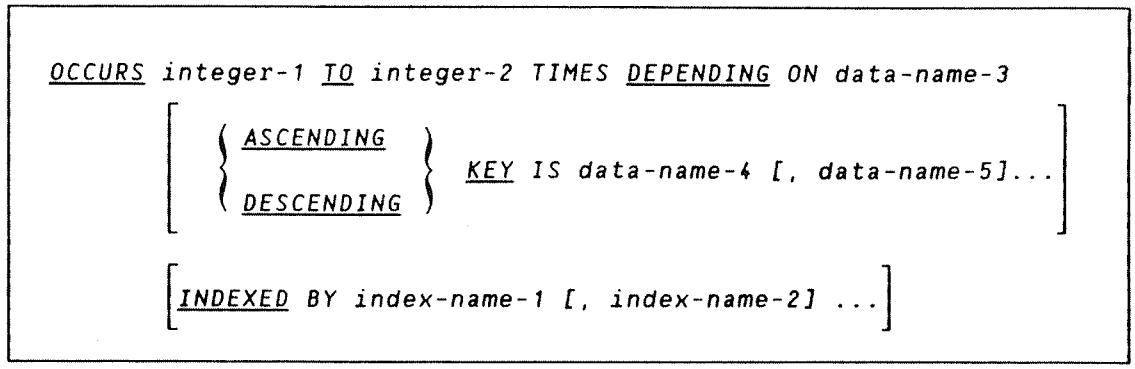
This clause eliminates the need for separate entries for repeated data items and supplies information required for the applying subscripts and indexes.

Format 1 (Fixed Length Tables):



In format 1, the value of integer-2 specifies the exact number of occurrences.

Format 2 (Variable Length Tables):



This format specifies that the subject of this entry has a variable number of occurrences. The current value of the data item referenced by data-name-1 represents the maximum number of occurrences and the value of integer-1 the minimum.

The value of the data item referenced by data-name-1 must fall within the range integer-1 through integer-2.

The ASCENDING/DESCENDING KEY option (both formats) specifies that the repeated data is arranged in ascending or descending order according to the values contained in data-name-2, data-name-3 etc. (The order is determined according to the rules for comparison of operands - see Comparison of Numeric and Nonnumeric operands under Conditional Statements in the Procedure Division description.)

The data-names are listed in their descending order of significance. Integer-1, when used, must be less than integer-2. All integers must be positive.

If the data-name, which is the subject of this entry, or an entry subordinate to it is to be referred to by indexing, then the INDEXED BY clause is required.

The OCCURS clause cannot be specified in a data description entry that:

a) has a level-number of 01, 77, or 88

b) described an item of variable size, i.e., if any subordinate item contains an OCCURS DEPENDING ON clause.

8.2.2 The USAGE Clause

The USAGE IS INDEX clause specifies that the data item has an index format.

Format

[USAGE IS] INDEX

The data item is an *index data item* and is treated as computational it will occupy 2 bytes in storage.

An index data item can be referenced explicity only in a SEARCH or SET statement, a relation condition, the USING phrase of a Procedure Division header, or the USING phrase of a CALL statement.

The USAGE clause can be written at any level, if written at group level it applies to every elementary item in the group. (The USAGE clause of a elementary item cannot contradict the USAGE clause of a group to which the item belongs.)

An index data item can be part of a group referred to in a MOVE or INPUT-OUTPUT statement, in which case no conversion will take place.

8.3 TABLE HANDLING - PROCEDURE DIVISION

In the Procedure Division, Table Handing makes use of the SEARCH and SET statements. Also, comparisons may be made between index-names and/or index data items as described under 'Relation Conditions' below.

Relation Conditions:

Comparison tests may be made between:

1) Two index-names. This is equivalent to comparing their occurrence numbers.

2) An index-name and a data item. The occurrence number corresponding to the value of the index-name is compared to the data item or literal.

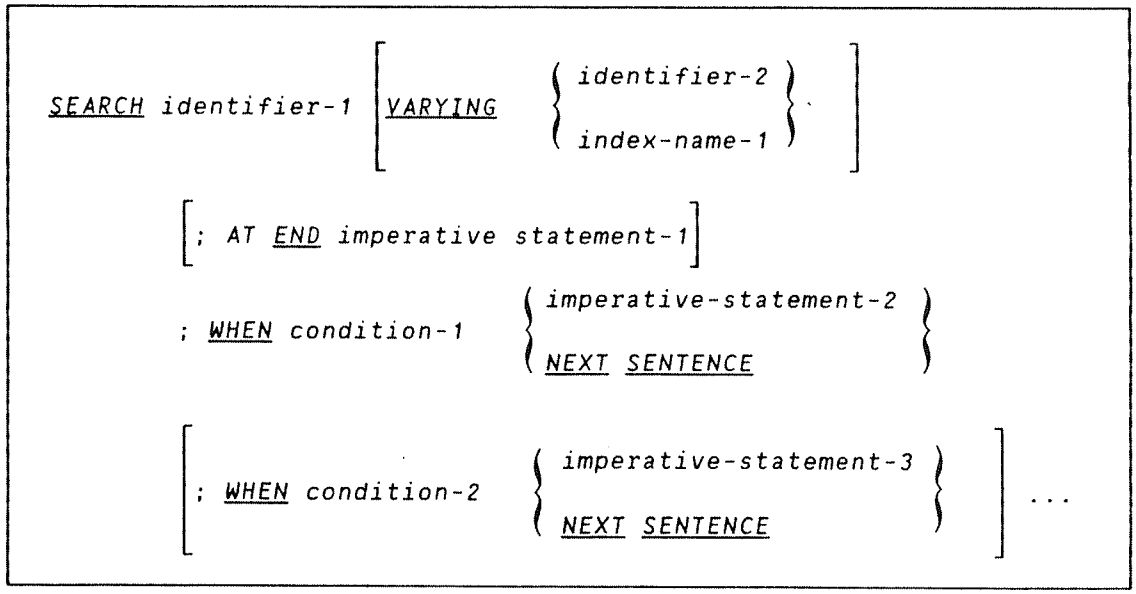
3) An index data item and an index name or another index data item. The actual values are compared without conversion.

4) The results of any other comparison involving an index data item are undefined.

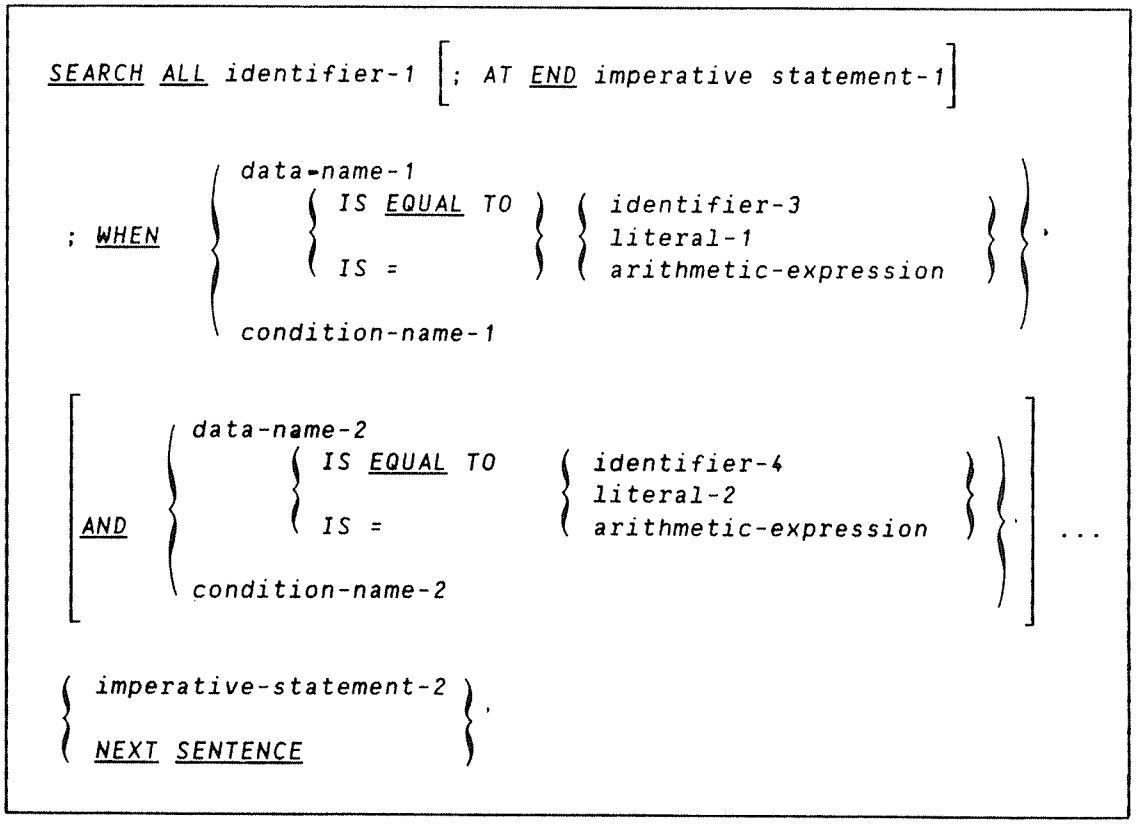
8.3.1 The SEARCH Statement

Data that has been arranged in the form of a table is very often searched. In COBOL, the SEARCH statement provides facilities, through its two options, for producing serial and non-serial searches. In using the SEARCH statement, the programmer may vary an associated index-name or data-name. This statement also provides facilities for executing imperative statements when certain conditions are true.

Format 1:



Format 2:



Note: The required relational character '=' is not underlined to avoid confusion with other symbols.

The SEARCH statement searches a table for an element that satisfies the specified condition, and adjusts the associated index to indicate that element.

In both formats, identifier-1 must not be subscripted or indexed, but its description in the Data Division must contain an OCCURS clause and an INDEXED BY clause.

Identifier-2, if present, must be described as USAGE IS INDEX or as a numeric elementary item without any positions to the right of the assumed decimal point.

Format 1:

1) The search operation begins at the current index setting. If, at this point, the value of the index-name associated with identifier-1 is not greater than the highest possible occurrence number, the following takes place:

   a) The conditions in the WHEN option are evaluated in the order in which they are written, making use of the index settings wherever specified.

   b) If none of the conditions are satisfied, the index-name for identifier-1 is incremented to correspond to the next table element. Then above process a) is repeated.

   c) If one of the conditions is satisfied upon evalutation, the search terminates immediately and the imperative statement associated with that condition is executed. The index-name remains pointing to the table element that caused the condition.

   d) If, however, the incremented index-name value is greater than the highest possible occurrence number (i.e., the end of the table has been reached), the search terminates. If the AT END phrase is specified, imperative-statement-1 is now executed. Otherwise, control passes to the next executable sentence.

2) At the beginning of the search operation, if the value of the index-name associated with identifier-1 is greater than the highest possible occurrence number, then the search terminates as explained above in step d).

3) When the VARYING phrase is not used, the index that is used for the search operation is the first (or only) index-name given in the INDEXED BY phrase of identifier-1.

4) If the VARYING index-name-1 option appears, then one of the following applies:

   a) When index-name-1 is the index for identifier-1, then this index is used for the search. If this is not the case (or the VARYING identifier-2 is present), the first - or only - index-name is used.

   b) If index-name-1 is an index for another table element, then the first (or only) index-name for identifier-1 will be used for the search. The occurrence number represented by index-name is incremented by the same amount as the search index-name, and at the same time.

5) If the VARYING identifier-2 option appears and identifier-2 is an index data item, then this item is incremented by the same amount as the search index, and at the same time. If identifier-2 is not an index data item, then it is incremented by the value one (1) at the same time as the search index is incremented.

A flowchart of a Format 1 type SEARCH operation containing two WHEN phrases follows:

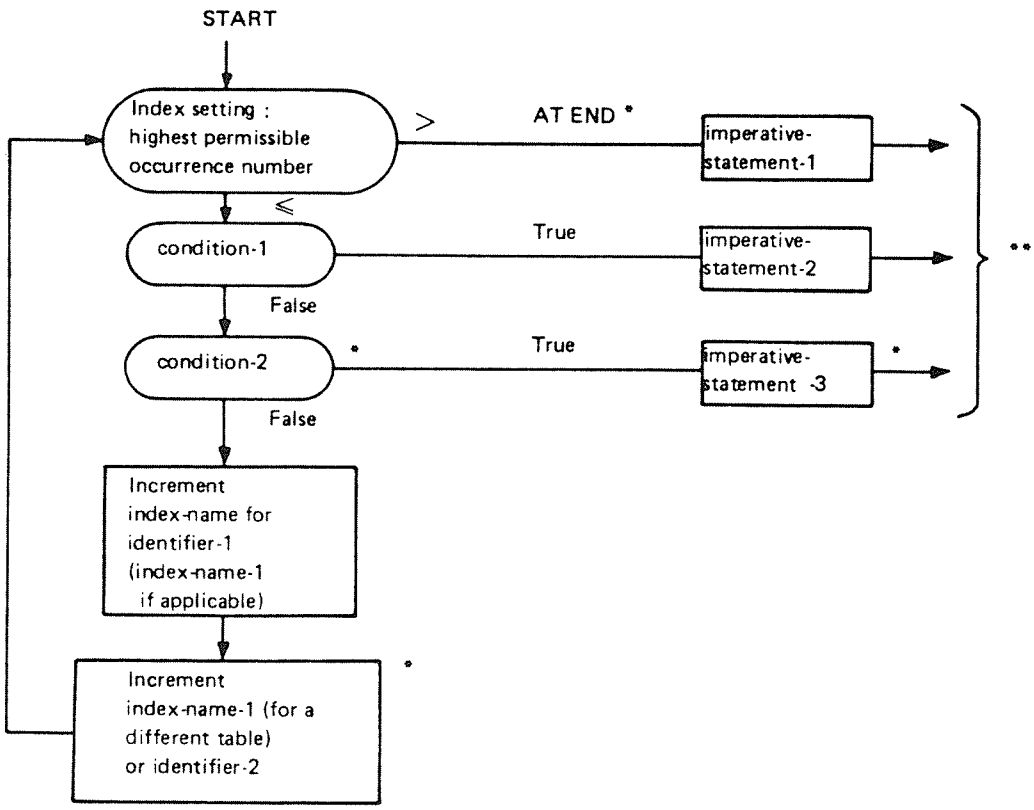


Figure 8.1:

* These operations are options included only when specified in the SEARCH statement.

** Each of these control transfers is to the next executable sentence unless the imperative-statement ends with a GO TO statement.

Format 2:

If the format 2, SEARCH ALL, is used, a non-serial search operation may take place. It is a more simple type of search than for format 1, commencing at the beginning of the table.

The initial setting of the index-name for identifier-1 is ignored (i.e., need not be initialized with the SET statement).

The index is the same as that associated with the first index-name specified in the OCCURS clause.

The following rules apply:

1) If the WHEN option cannot be satisfied by any setting of the index within the permitted range, then control is passed to imperative-statement-1 of the AT END phrase if present, or to the next executable sentence if this phrase is not present. In either case, the final setting of the index is not predictable.

2) If the WHEN option can be satisfied, control passes to imperative-statement-2 and the index will indicate an occurrence that allows the conditions to be satisfied.

8.3.1.1 Notes on Multidimensional Tables

Identifier-1 can be a data item subordinate to a data item containing an OCCURS clause. That is, it can be part of a two or three-dimensional table. In this case, the data description entry must specify an INDEXED BY option for each dimension.

To search an entire two or three-dimensional table it is necessary to execute a SEARCH statement several times, since execution of this statement modifies the setting of the index-name associated with identifier-1 only (and, if present, index-name-1 or identifier-2). Prior to each execution, SET statements must be executed to reinitialize the associated index-names.

Example:

```
 1        IDENTIFICATION DIVISION.
 2        PROGRAM-ID.
 3            SEARCH-EX.
 4       ****************************************************************
 5       * SHOWS USAGE OF SIMPLE TABLE "LOOK-UP" VIA SEARCH VERB.
 6       ****************************************************************
 7        DATA DIVISION.
 8        WORKING-STORAGE SECTION.
 9        77  TABLE-LENGTH        COMP VALUE 16.
10        77  FIND-NAME           PIC X(20).
11       ****************************************************************
12       * SET UP THE TABLE ELEMENTS, NORMALLY ONE WOULD READ DATA FROM
13       * A "REFERENCE" FILE AND PLACE INTO TABLE FOR PROCESSING.
14       ****************************************************************
15        01  NAMES-TABLE.
16            02  PIC X(20) VALUE "BRABANT             ".
17            02  PIC 9(5)  VALUE  310.
18            02  PIC X(20) VALUE "CISALPIN            ".
19            02  PIC 9(5)  VALUE  822.
20            02  PIC X(20) VALUE "ERASMUS             ".
21            02  PIC 9(5)  VALUE  481.
22            02  PIC X(20) VALUE "ETOILE DU NORD      ".
23            02  PIC 9(5)  VALUE  554.
24            02  PIC X(20) VALUE "GOTTARDO            ".
25            02  PIC 9(5)  VALUE  381.
26            02  PIC X(20) VALUE "ILE DE FRANCE       ".
27            02  PIC 9(5)  VALUE  544.
28            02  PIC X(20) VALUE "IRIS                ".
29            02  PIC 9(5)  VALUE  666.
30            02  PIC X(20) VALUE "LE CATALAN TALGO    ".
31            02  PIC 9(5)  VALUE  870.
32            02  PIC X(20) VALUE "LE CAPITOLE         ".
33            02  PIC 9(5)  VALUE  373.
34            02  PIC X(20) VALUE "LE MISTRAL          ".
35            02  PIC 9(5)  VALUE  683.
36            02  PIC X(20) VALUE "LEMANO              ".
37            02  PIC 9(5)  VALUE  595.
38            02  PIC X(20) VALUE "LIGURE              ".
39            02  PIC 9(5)  VALUE  322.
40            02  PIC X(20) VALUE "MEDIOLANUM          ".
41            02  PIC 9(5)  VALUE  889.
42            02  PIC X(20) VALUE "OISEAU BLEU         ".
43            02  PIC 9(5)  VALUE  1039.
44            02  PIC X(20) VALUE "REMBRANDT           ".
45            02  PIC 9(5)  VALUE  713.
46            02  PIC X(20) VALUE "RHEINGOLD           ".
47            02  PIC 9(5)  VALUE  1088.
48       ****************************************************************
49       * REDEFINE THE ELEMENTS FOR ACCESS WITH THE SEARCH VERB,
50       * NOTE THAT THE DATA-NAME WITH THE *OCCURS* CLAUSE IS USED IN
51       * SEARCH AND NOT THE REDEFINES DATA-NAME (WHICH MAY BE "FILLER").
52       ****************************************************************
53        01  FILLER  REDEFINES NAMES-TABLE.
54            02  TRAIN-TABLE OCCURS 16 TIMES INDEXED BY TABINDEX.
55                03  NAME        PIC X(20).
56                03  DISTANCE    PIC 9(5).
57
58        PROCEDURE DIVISION.
59        A000.
60       ****************************************************************
61       * LIST OUT ALL THE TABLE ENTRIES.
62       ****************************************************************
63            PERFORM LIST-TABLE-ENTRY
64                    VARYING TABINDEX FROM 1 BY 1 UNTIL
65                    TABINDEX = TABLE-LENGTH.
66        A002.
67       ****************************************************************
68       * REQUEST A NAME TO FIND.
69       ****************************************************************
70            DISPLAY "ENTER NAME TO FIND: ".
71            ACCEPT FIND-NAME.
72       ****************************************************************
73       * START AT TOP OF TABLE(1).
```

```
74      ****************************************************************
75           SET TABINDEX TO 1.
76      ****************************************************************
77      * LOOK FOR REQUESTED NAME.
78      ****************************************************************
79           SEARCH TRAIN-TABLE AT END DISPLAY "NAME NOT FOUND",
80                   WHEN FIND-NAME = NAME(TABINDEX),
81                           PERFORM LIST-TABLE-ENTRY.
82           GO TO A002.
83      ****************************************************************
84      * NOTE THE WAY THAT THE LIST ROUTINE IS USED BY EITHER THE
85      * PERFORM OR THE SEARCH VERB.
86      ****************************************************************
87       LIST-TABLE-ENTRY.
88           DISPLAY "TRAIN: ", NAME(TABINDEX), " TRAVELS: ",
89                   DISTANCE(TABINDEX), " KM.".
```

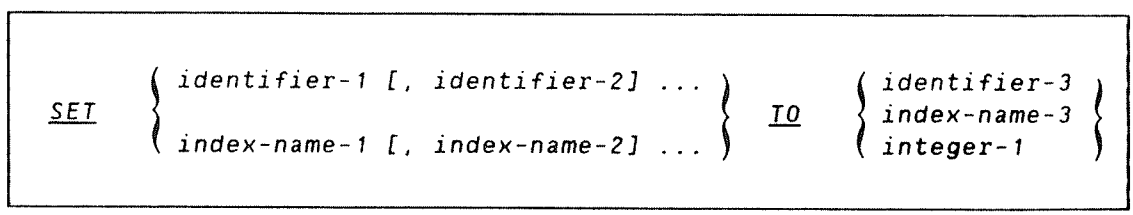
This program makes the same output whether we execute it on an ND-100 or an ND-500. The following shows execution on an ND-500 from SINTRAN III:

```
@nd (cobol-exam)c-8-13

TRAIN: BRABANT                TRAVELS: 00310 KM.
TRAIN: CISALPIN               TRAVELS: 00822 KM.
TRAIN: ERASMUS                TRAVELS: 00481 KM.
TRAIN: ETOILE DU NORD         TRAVELS: 00554 KM.
TRAIN: GOTTARDO               TRAVELS: 00381 KM.
TRAIN: ILE DE FRANCE          TRAVELS: 00544 KM.
TRAIN: IRIS                   TRAVELS: 00666 KM.
TRAIN: LE CATALAN TALGO       TRAVELS: 00870 KM.
TRAIN: LE CAPITOLE            TRAVELS: 00373 KM.
TRAIN: LE MISTRAL             TRAVELS: 00683 KM.
TRAIN: LEMANO                 TRAVELS: 00595 KM.
TRAIN: LIGURE                 TRAVELS: 00322 KM.
TRAIN: MEDIOLANUM             TRAVELS: 00889 KM.
TRAIN: OISEAU BLEU            TRAVELS: 01039 KM.
TRAIN: REMBRANDT              TRAVELS: 00713 KM.
ENTER NAME TO FIND: IRIS
TRAIN: IRIS                   TRAVELS: 00666 KM.
ENTER NAME TO FIND: ETOILE
NAME NOT FOUND
ENTER NAME TO FIND: ETOILE DU NORD
TRAIN: ETOILE DU NORD         TRAVELS: 00554 KM.
ENTER NAME TO FIND: LEMANO
TRAIN: LEMANO                 TRAVELS: 00595 KM.
ENTER NAME TO FIND:
```

8.3.2 The SET Statement

The SET statement establishes reference points for table handling operations by setting index-names associated with table elements.

Format 1:



Format 2:

```
SET index-name-4 [, index-name-5] ...

                    { UP BY   }   { identifier-4 }
                    { DOWN BY }   { integer-2    }
```

Identifier-1 and identifier-3 must name either index data items, or elementary items described as an integer.

Identifier-4 must be described as an elementary numeric integer.

Integer-1 and integer-2 may be signed. Integer-1 must be positive.

Index-names are related to a given table through the INDEXED BY option of the OCCURS clause which constitutes their definition.

Format 1 - TO Option

When this form of the SET statement is executed, the value of the sending field replaces the current value of the *receiving field*. If the receiving field specifies index-name-1, then, either:

a) If the sending field is an index data item, then the value of this item is placed in the index name without change.

b) Otherwise, the receiving field is converted to a displacement value corresponding to the occurrence number indicated by the sending field.

If the receiving field specifies an index data item, then this is set equal to the contents of the sending field (which must be an index-name or an index data item), and no conversion takes place.

If the receiving field specifies an integer data item, then it is set to an occurrence number that corresponds to the occurrence number associated with the sending field (which must be an index name).

The above processes are repeated for indentifier-2, index-name-2, etc.

If index-name-3 is specified, the value of the index before execution of the SET statement must correspond to an occurrence number of an element in the associated table.

Any subscripting or indexing associated with identifier-1, etc., is evaluated immediately before the value of the respective data item is changed.

Format 2 - UP/DOWN BY Option

When this form of the SET option is executed, the value of the receiving field, index-name-4, is incremented (UP BY) or decremented (DOWN BY) by a value corresponding to the value in the sending field. The process is repeated for index-name 5, etc.

Sending Item	Receiving Item		
	Integer Data Item	Index-name	Index Data Item
Integer Literal	No	Valid	No
Integer Data Item	No	Valid	No
Index-name	Valid	Valid	Valid*
Index Data Item	No	Valid*	Valid*

* No conversion takes place.

9 INTER-PROGRAM COMMUNICATION

Complex data processing problems are frequently solved by the use of separately compiled but logically coordinated programs, which, at execution time, form logical and physical subdivisions of a single run unit. This approach lends itself to dividing a large problem into smaller, more manageable segments which can be programmed and debugged independently. During execution, control is transferred from program to program by the used of CALL and EXIT PROGRAM statements.

9.1 BASIC CONCEPTS

In COBOL terminology, a program is either a source program or an object program depending on context; a source program is a syntactically correct set of COBOL statements; an object program is the set of instructions, constants, and other machine-oriented data resulting from the operation of a compiler on a source program; and a run unit is the total machine language necessary to solve a data processing problem. It includes one or more object programs as defined above, and it may include machine language from sources other than a COBOL compiler.

When the statement of a problem is subdivided into more than one program, the constituent programs must be able to communicate with each other. This communication may take two forms: transfer of control and reference to common data.

9.1.1 Transfer of Control

The CALL statement provides the means whereby control can be passed from one program to another within a run unit. A program that is activated by a CALL statement may itself contain CALL statements. However, results are unpredictable where circulaity of control is initiated; i.e., where program A calls program B, then program B calls program A or another program that calls program A.

When control is passed to a called program, execution proceeds in the normal way from procedure statement to procedure statement beginning with the first nondeclarative statement. If control reaches a STOP RUN statement, this signals the logical end of the run unit. If control reaches an EXIT PROGRAM statement, this signals the logical end of the called program only, and control then reverts to the point immediately following the CALL statement in the calling program. Stated briefly, the EXIT PROGRAM statement terminates only the program in which it occurs, and the STOP RUN statement terminates the entire run unit.

If the called program is not COBOL, then the termination of the run unit or the return to the calling program must be programmed in accordance with the language of the called program.

9.1.2 Reference to Common Data

Because of program interaction, it may be necessary for one or more programs to have access to the same data.

In a calling program, the common data items are described together with all other data items in the File, Working-Storage, or Linkage Sections. In the called program, common data items are described in the Linkage Section.

At object time, memory is allocated for the whole Data Division in the calling program but not for the Linkage Section of the called program. Communication between the called program and the common data items stored in the calling program is through USING clauses contained in both programs. The USING clause in the calling program is contained in the CALL statement and the operands are common data items described in its Data Division. The USING clause in the called program has operands which are data items appearing in its Linkage Section.

The sequence of appearance of the identifiers in both lists of operands is significant. They must match in both programs. While the called program is being executed, every reference to an operand whose identifier appears in the called program's USING clause is treated as if it were a reference to the corresponding operand in the USING clause of the active CALL statement.

(A calling program may itself be a called program, in this case, common data items can be described in the calling program's Linkage Section. Storage will not be allocated for these items in the calling program itself but reather in the program which calls the calling program.)

An example of a called and a calling program is outlined below:

CALLING PROGRAM (PROG-A)	*CALLED PROGRAM (PROG-B)*
WORKING-STORAGE SECTION.	*LINKAGE SECTION.*
01 A-LIST. *02 HEADING PIC X(10).* *02 YEAR PIC 9(2).* *02 MONTH PIC 9(2).* *02 CODE-NO PIC X(4).* .	*01 B-LIST.* *05 HEADING PIC X(10).* *05 DATE PIC 9(4).* *05 CODE-ID PIC X(4).* .
PROCEDURE DIVISION. .	*PROCEDURE DIVISION USING B-LIST.*
CALL PROG-B USING A-LIST.	

Note that the names of the data items need not correspond and that parts of data items can be referred to separately (DATE in the called program is subdivided into YEAR and MONTH in the calling program).

9.1.3 Interprogram Communication - Data Division

In the Data Division of a called program, all file description entries may be assigned a value of an integral literal (using a VALUE OF FILE-ID IS clause) which is the same as that defined in the main program, refer to section 5.3.1.

Note: In the present version of ND COBOL, it is no longer necessary to include a VALUE OF FILE-ID IS clause. If you *do* include this clause, its contents are treated as comments only.

The programmer specifies in the Linkage Section those data items that the called program has in common with the calling program.

Format:

```
LINKAGE SECTION.

level-number { data-name-1 }
             { FILLER      }

    [REDEFINES Clause ]
    [BLANK WHEN ZERO Clause ]
    [JUSTIFIED Clause ]
    [OCCURS Clause ]
    [PICTURE Clause ]
    [SIGN Clause ]
    [SYNCHRONIZED Clause ]
    [USAGE Clause ]
    [IMPORT Clause ]
    [IMPORT [COMMON] Clause]

[88 condition-name VALUE Clause ]
```

The Linkage Section in a program is meaningful if and only if the object program is to function under the control of a CALL statement, and the CALL statement in the calling program contains a USING phrase.

The IMPORT clause must specify the same data item as in the corresponding EXPORT clause (see section 5.4.2.13 for a description of the rules which apply to both clauses).

The IMPORT COMMON clause is used to specify a FORTRAN common block IMPORT.

The VALUE clause must not be specified in the Linkage Section except in condition-name entries (level 88).

The Linkage Section is used for describing data that is available through the calling program but is to be referred to in both the calling and the called program. No space is allocated in the program for data items referenced by data-names in the Linkage Section of that program. Procedure Division references to these data items are resolved at object time by equating the reference in the called program to the location used in the calling program.

Data items defined in the Linkage Section of the called program may be referenced within the Procedure Division of the called program only if they are specified as operands of the USING phrase of the Procedure Division header or are subordinate to such operands, and the object program is under the control of a CALL statement that specifies a USING phrase.

The structure of the Linkage Section is the same as that previously described for the Working-Storage Section, beginning with a section header, and followed by data description entries for noncontiguous data items and/or record description entries.

Each Linkage Section record name and noncontiguous item name must be unique within the called program since it cannot be qualified.

Of those items defined in the Linkage Section, only data-name-1, data-name-2, ... in the USING phrase of the Procedure Division header, data items subordinate to these data-names, and condition names and/or index-names associated with such data-names and/or subordinate data items, may be referenced in the Procedure Division.

9.1.3.1 Data Item Description Entries

Items in the Linkage Section that bear no hierarchic relationship to one another need not be grouped into records and are classified and defined as noncontiguous elementary items. Each of these data items is defined in a separate data description entry which begins with the special level-number 77.

The Following data clauses are required in each data description entry:

a) level-number 77

b) data-name

c) the PICTURE clause or the USAGE IS INDEX clause

Other data description clauses are optional and can be used to complete the description of the item if necessary.

9.1.3.2 Record Description Entries

Data elements in the Linkage Section which bear a definite hierarchic relationship to one another must be grouped into 01-level records according to the rules for formation of record descriptions. Any clause which is used in an input or output record description can be used in a Linkage Section.

9.1.4 Inter-Program Communication - Procedure Division

In the Procedure Division, control is transferred between programs by means of the CALL statement.

Reference to common data is provided by the USING option which can appear in the CALL statement and in the called program's Procedure Division header.

Format of Procedure Division Header:

```
PROCEDURE DIVISION [USING data-name-1 [, data-name-2] ...] .
```

The USING phrase is present if and only if the object program is to function under the control of a CALL statement, and the CALL statement in the calling program contains a USING phrase.

For a description of the data-names, see the details of the USING option in the CALL statement. The USING option is common to several Inter-Program Communication elements.

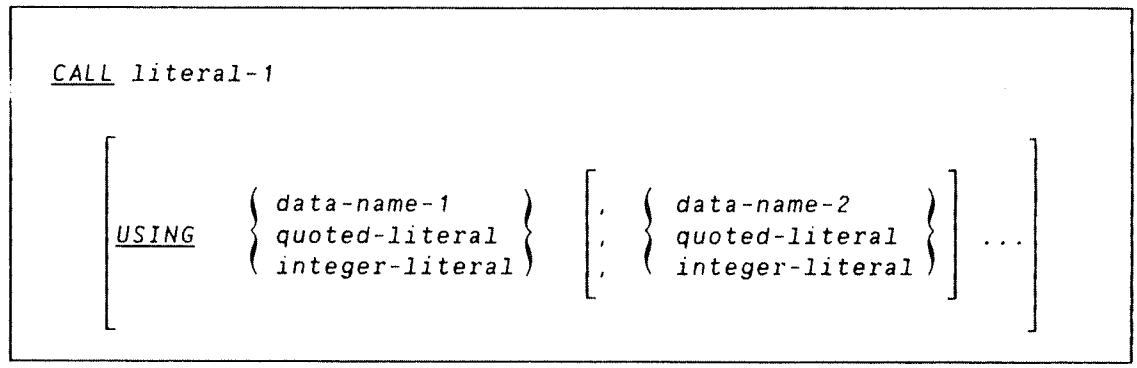
Each of the operands in the USING phrase of the Procedure Divison header must be defined as a data item in the Linkage Section of the program in which this header occurs, and it must have 01 or 77 level-number.

Within a called program, Linkage Section data items are processed according to their data descriptions given in the called program.

9.1.4.1 The CALL Statement

The CALL statement causes control to be transferred from one object program to another within the run unit.

Format:



Literal-1 must be a non-numeric literal and conform to the rules for formation of a program name (see PROGRAM-ID paragraph in the Identification Division chapter).

Called programs may contain CALL statements.

CALL statement execution causes control to pass to the called subprogram. The first time a called program is entered its state is that of an original copy of the program. Each subsequent time a called program is entered, the state is as it was upon the last exit from that program.

Reinitialization of GO TO statements that have been altered etc., are the responsibility of the programmer.

This option makes data items in a calling program available to the called program.

The USING option is specified if, and only if, the called subprogram is to operate under control of a CALL statement and that CALL statement itself contains a USING option. That is, for each CALL USING statement in a calling program there must be a corresponding USING option specified in a called subprogram.

The data-name, or quoted-literal, or integer-literal, specified by the USING option indicate the data items available to a calling program that may also be referred to in the called program. The order of appearance of these data-names is critical. Corresponding data-names refer to a single set of data equally available to both programs. Their description must define an equal number of character positions but their correspondence is positional and not by name. (In the case

of index names no such correspondence is established, and separate indices are referred to in the called and calling programs).

The integer-literal must be in the range -32768 to +32767 on both the ND-100 and the ND-500.

9.1.4.2 The EXIT PROGRAM Statement

The EXIT PROGRAM statement marks the logical end of a called program.

Format

EXIT PROGRAM.

The EXIT PROGRAM statement must appear in a sentence by itself.

The EXIT PROGRAM sentence must be the only sentence in the paragraph.

General Rule:

1) An execution of an EXIT PROGRAM statement in a called program causes control to be passed to the calling program. During execution, an EXIT PROGRAM statement in a program which is not called behaves as if it were an EXIT statement (see under Procedure Branching Statements in the Procedure Division description).

Example:

```
 1        IDENTIFICATION DIVISION.
 2        PROGRAM-ID.
 3            GENSEQ.
 4       ***********************************************************************
 5       *    CREATES SQ-FILES AND LISTS.
 6       ***********************************************************************
 7        ENVIRONMENT DIVISION.
 8        INPUT-OUTPUT SECTION.
 9        FILE-CONTROL.
10            SELECT SQ-FILE ASSIGN "COB1:DATA",
11                ORGANIZATION IS SEQUENTIAL,
12                ACCESS IS SEQUENTIAL.
13        DATA DIVISION.
14        FILE SECTION.
15        FD  SQ-FILE.
16        01  M-REC.
17            02          PIC X(10).
18            02 SEQNUM   PIC 9(5), BLANK WHEN ZERO.
19            02          PIC X(5).
20            02          PIC X(40).
21        WORKING-STORAGE SECTION.
22        01  RANDNO       PIC  9(4) PACKED-DECIMAL, VALUE ZERO.
23        01  MAXRAND      PIC S9(4) PACKED-DECIMAL, VALUE 1000.
24        01  NORECS       PIC  9(4) PACKED-DECIMAL.
25        01  RECCNT       PIC  99, COMP, VALUE 0.
26
27        PROCEDURE DIVISION.
28        INIT-01.
29            OPEN OUTPUT SQ-FILE.
30            DISPLAY "CREATE RECORDS?".
31            PERFORM GET-NORECS.
32            PERFORM CRE-SQ-FILE NORECS TIMES.
33       * BUILDS THE INPUT FILE.
34            CLOSE SQ-FILE.
35            DISPLAY "FILE SQ-FILE CREATED.", RECCNT, " RECORDS.".
36            OPEN INPUT SQ-FILE.
37        LIST-FILE-0.
38            MOVE 0 TO RECCNT.
39        LIST-FILE-1.
40            READ SQ-FILE AT END GO TO LIST-END.
41            ADD 1 TO RECCNT.
42            DISPLAY "REC ", RECCNT, ", SEQNUM = ", SEQNUM.
43            GO TO LIST-FILE-1.
44        LIST-END.
45            CLOSE SQ-FILE.
46            DISPLAY "JOB FINISH".
47            STOP RUN.
48        CRE-SQ-FILE.
49            CALL "RND" USING RANDNO, MAXRAND.
50            MOVE ALL "*" TO M-REC.
51            MOVE RANDNO TO SEQNUM.
52            ADD 1 TO RECCNT.
53            DISPLAY "UT REC = ", RECCNT, " KEY = ", SEQNUM.
54            WRITE M-REC.
55        GET-NORECS.
56            ACCEPT NORECS.
57            IF NORECS NOT NUMERIC,
58                DISPLAY "*** NOT NUMERIC DATA ***",
59                GO TO GET-NORECS
60            END-IF.
```

Example:

```
 1        IDENTIFICATION DIVISION.
 2        PROGRAM-IO.
 3            RND.
 4       ***************************************************************
 5       * GENERATES RANDOM NUMBERS FOR USE IN OTHER EXAMPLES THROUGHOUT
 6       * THE MANUAL. NOTE THAT THIS ROUTINE IS NOT VERY EFFICIENT COMPARED
 7       * TO A RANDOM NUMBER GENERATORS IN ASSEMBLY LANGUAGE, FOR
 8       * INSTANCE. BUT IT CAN BE COMPILED AND LINKED INTO ANY ND COBOL
 9       * PROGRAM.
10       ***************************************************************
11        DATA DIVISION.
12        WORKING-STORAGE SECTION.
13        01  SEED     PIC  9(4)  PACKED-DECIMAL, VALUE 29533.
14        01  CONST    PIC  9(8)  PACKED-DECIMAL, VALUE 34764391.
15        01  VAR      PIC  9(8)  PACKED-DECIMAL.
16        01  X        PIC  9(4)  PACKED-DECIMAL.
17        01  XMAX     PIC S9(4)  PACKED-DECIMAL, VALUE 1000.
18        LINKAGE SECTION.
19        01  RANDNO   PIC  9(4)  PACKED-DECIMAL.
20        01  MAXRAND  PIC S9(4)  PACKED-DECIMAL.
21
22        PROCEDURE DIVISION USING RANDNO, MAXRAND.
23        MAXRAND-SWITCH.
24            IF MAXRAND > 0 THEN PERFORM GENERATE-NUMBER
25            ELSE-IF MAXRAND = 0 THEN PERFORM RETURN-SEED
26            ELSE PERFORM SET-NEW-SEED
27            END-IF.
28        GENERATE-NUMBER.
29            MOVE RANDNO TO VAR.
30            COMPUTE VAR = (VAR + SEED) * CONST.
31            MOVE MAXRAND TO XMAX.
32            DIVIDE VAR BY XMAX GIVING VAR REMAINDER X.
33            MOVE X TO RANDNO.
34            EXIT PROGRAM.
35        RETURN-SEED.
36            MOVE SEED TO RANDNO.
37            EXIT PROGRAM.
38        SET-NEW-SEED.
39            MOVE RANDNO TO SEED.
40            EXIT PROGRAM.
```

10 DEBUGGING

The Symbolic Debugger is an ND product that lets you test your ND-100 and ND-500 programs. Here are some of the things you can do with it. After each item, the Debugger command(s) you use are listed.

- Stop execution of your program at a given line, label, routine, or program address.
  Use BREAK or BREAK-ADDRESS.

- Define multiple step points where your program will automatically stop.
  Use LOG-LINES one or more times, followed by STEP, STEP, STEP, etc. Otherwise use LOG-CALLS and STEP. To see what has been logged, use DUMP-LOG.

- Stop execution of your program when the value of a variable changes.
  Use LOG-LINES or LOG-CALLS, followed by GUARD.

- Inspect the values of variables during program execution.
  Use DISPLAY or LOOK-AT-DATA.

- Find out which lines in your program get executed and which do not.
  Use STEP 0, which will list lines until a BREAK is reached.

- Debug screen-oriented programs by using two terminals.
  Use RESERVE-TERMINAL.

- Change the value of COMPUTATIONAL variables in main programs.
  Use SET.

All of these facilities, as well as others, are described in detail in the Symbolic Debugger User Guide, ND-60.144.

The Debugger has about 30 commands that are identical in the ND-100 and ND-500 versions. In addition, there are 3 commands that are only on the ND-100 Debugger, and 5 commands that are only on the ND-500 Debugger.

10.1 USING THE ND-100

Here is how you compile two files, MAIN:SYMB and SUB:SYMB, so that they can be debugged on the ND-100. In this section, and elsewhere in this chapter, the sign "↵" is used to indicate where you must press the Carriage Return key.

```
@CREATE-FILE MAIN:BRF 0 ↵
@CREATE-FILE SUB:BRF 0 ↵
@COBOL-100 ↵
DEBUG-MODE ↵
COMPILE MAIN,TERMINAL,MAIN ↵
DEBUG-MODE ↵
COMPILE SUB,TERMINAL,SUB ↵
EXIT ↵
```

Note: Type DEBUG-MODE before every COMPILE line, because DEBUG is turned off automatically.

Load your :BRF files in the normal way. You may use NRL or the BRF-LINKER. If you use overlay, however, you must use the BRF-LINKER.

If your program is called MAIN:PROG, start the debugger like this:

```
@DEBUGGER MAIN ↵
```

or like this:

```
@DEBUGGER ↵
*PLACE MAIN ↵
```

10.2 USING THE ND-500

Here is how you compile two files, MAIN:SYMB and SUB:SYMB, so that they can be debugged on the ND-500. In this section, and elsewhere in this chapter, the sign "↵" is used to indicate where you must press the Carriage Return key.

```
@CREATE-FILE MAIN:NRF 0 ↵
@CREATE-FILE SUB:NRF 0 ↵
@ND COBOL-500 ↵
DEBUG-MODE ↵
COMPILE MAIN,TERMINAL,MAIN ↵
DEBUG-MODE ↵
COMPILE SUB,TERMINAL,SUB ↵
EXIT ↵
```

Note: Type DEBUG-MODE before every COMPILE line, because DEBUG is turned off automatically.

Load your :NRF files in the normal way in the LINKAGE-LOADER.

If you call your domain MAIN, start the debugger like this:

```
@ND DEBUGGER MAIN ↵
```

10.3 DEBUGGING EXAMPLES

In this section, we will show how to get information about the execution of the following little COBOL program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DEBUG.
AUTHOR.     IBO.

DATA DIVISION.
WORKING-STORAGE SECTION.
01  F1  PIC 9(12) USAGE IS COMP VALUE IS ZERO.
01  F2  PIC 9(12) USAGE IS COMP VALUE IS ZERO.
01  S   PIC 9(12) USAGE IS COMP VALUE IS ZERO.
01  T   PIC X(12) VALUE IS SPACES.
01  ANSWER PIC X VALUE IS SPACE.

PROCEDURE DIVISION.

READ-NUMBER-1.
    DISPLAY 'FIRST NUMBER: '
        ACCEPT F1.
    IF F1 IS EQUAL TO 0 THEN GO TO READ-NUMBER-1.

READ-NUMBER-2.
    DISPLAY 'SECOND NUMBER: '
        ACCEPT F2.
    IF F2 IS EQUAL TO 0 THEN GO TO READ-NUMBER-2.

MULTIPLICATE.
    MULTIPLY F1 BY F2 GIVING S.
    MOVE S TO T.
    INSPECT T REPLACING LEADING '0' BY ' '.
    DISPLAY 'MULTIPLICATION GIVES ', T.
    DISPLAY 'CONTINUE? (N TO STOP) '
        ACCEPT ANSWER.
    IF ANSWER IS NOT EQUAL TO 'N' THEN GO TO READ-NUMBER-1.

FINI.
    STOP RUN.
```

Example 1

Here is an example of the Debugger being used on the little program above compiled and loaded as an ND-100 program. The letters A, B, C, etc., to the left refer to comments that appear at the bottom of each page.

```
          @DEBUG TEST ↵
          COBOL PROGRAM.  DEBUG.3
A         *BREAK,25 F1=5 ↵
B         *RUN ↵

          FIRST NUMBER: 5 ↵
          SECOND NUMBER: 5 ↵

          CONDITIONAL BREAK AT DEBUG.25
          *DISPLAY F1 ↵
          F1=5
C         *SET F1=11 ↵
          *DISPLAY F2 ↵
          F2=5
          *FORMATS-LOOK-AT ↵
          FORMATS (A,D,F,H,I,O OR COMBINATIONS): I ↵
D         *LOOK-AT-PROGRAM ↵
          PROGRAM ADDRESS: ↵
          P 000070B: LDD I * 135
          P 000071B: SKP DA UEQ ; ↵
E         *LOOK-AT-DATA ADDR(F1) ↵
          D 000002B: 000000B       0
          D 000003B: 0000138      11
          D 000004B: 000000B       0
          D 000005B: 000005B       5     ; ↵
F         *ACTIVE-ROUTINES ↵
          DEBUG.3
```

NOTES:

A) This is a conditional break. You will stop at line 25 if the value of the variabel F1 equals 5.

B) The program will run until F1 equals 5 in line 25.

C) SET can be used to change the values of both numeric and string variables.

D) You can inspect the instructions that your program consists of.

E) This inspects the address at which F1 is stored. Evidently F1 is stored in 2 bytes.

F) This will list the hierarchy of calls. It is useful to know in program with many subroutines.

```
          *LOG-LINES ↵
          PROGRAM AREA:
A         *GUARD,F1 ↵
          *RUN ↵
          MULTIPLICATION GIVES              55
          CONTINUE? (N TO STOP) Y ↵
          FIRST NUMBER: 10 ↵

          GUARD VIOLATION AT DEBUG.20
          *DISPLAY F1 ↵
          F1=10
B         *RESET-BREAKS ↵
          *RUN ↵
          SECOND NUMBER: 9 ↵
          MULTIPLICATION GIVES              90
          CONTINUE? (N TO STOP) N ↵
          ∂
```

NOTES:

A) This removes all break and step points.

B) The program will stop when the value of F1 changes anywhere in the program.

The above program could have been debugged in the same way if it had been a ND-500 program. Only the output from the LOOK-AT commands would have appeared differently.

Example 2

Here is another example using the same program. Note the use of multiple step points:

```
        @ND DEBUGGER TEST ↵
        ND-500 SYMBOLIC DEBUGGER.
        COBOL PROGRAM.  DEBUG.3
        *MACRO ↵
A       NAME: X ↵
        BODY: DISPLAY F1;DISPLAY F2;STEP ↵
        *LOG-LINES READ-NUMBER-1 ↵
        *LOG-LINES 23 ↵
B       *LOG-LINES 27 ↵
        *STEP ↵
C       DEBUG.18      *X ↵
        F1=0
        F2=0

        FIRST NUMBER: 5 ↵
        DEBUG.23      *X ↵
        F1=5
        F2=0
        SECOND NUMBER: 20 ↵
        DEBUG.28      *X ↵
        F1=5
        F2=20
        MULTIPLICATION GIVES          100
        CONTINUE? (N TO STOP) N ↵

        @__
```

NOTES:

A) The command X will perform the 3 commands listed after body.

B) 3 step points are defined. Each STEP command will bring you to the next step point. Each RUN command will bring you to the nearest breakpoint.

C) READ-NUMBER-1 starts at line 18.

The above program could have been debugged in the same way if it had been a ND-100 program. Only the output from the LOOK-AT commands would have appeared differently.

To learn more about how to use the Debugger, see the Symbolic Debugger User Guide, which is manual ND-60.144.

11 PROGRAMMING EXAMPLES

11.1 EXECUTING A SIMPLE PROGRAM

Here are some examples of how to compile, load and run a simple program on the ND-100 and ND-500 computers. The program also demonstrates some of the features of the ND COBOL.

To try out the example, you must first write the following program onto the file "X-001:symb", using one of the ND editors. And by the way, note a convenient feature when using PED and NOTIS: if you write *% tab c* on the first positions in the first line of the file, then the editor will set tabulator stops suitable for COBOL programs for you when initializing.

```
% tab c
   IDENTIFICATION DIVISION.
   PROGRAM-ID.X-001.

   DATA DIVISION.
   WORKING-STORAGE SECTION.
   01 NAME    PIC X(30).
   01 I       COMP.

   PROCEDURE DIVISION.
   1000. BLANK SCREEN.
         DISPLAY (10, 1) "Your name:"
         ACCEPT (10, 12) NAME WITH PROMPT.
         BLANK LINE 10.
         DISPLAY (1, 1) FRAME 18 * 75 WITH HEADING.
         DISPLAY (2, 28) "My  name  is".
         DO FOR I FROM 4 to 17
            DISPLAY (I, 3) NAME WITH BLINK
            DISPLAY (I, 42) NAME WITH INVERSE-VIDEO
         END-DO.
         DISPLAY (22, 10)
            "You have now used the ND COBOL System"
              WITH UNDERLINE.
   STOP RUN.
```

11.1.1 Running the Example on an ND-100 Computer

The following listing of a terminal session shows how to compile, load and execute the program on an ND-100. The underlined parts of the listing is what you have to type in when trying the example.

```
@COBOL-H

ND-100 COBOL COMPILER - ND-10176H

*COMPILE X-001,1,X-001

ND-100 COBOL COMPILER - ND-10176H  TIME: 16.17.21  DATE: 85.02.19

  SOURCE FILE: X-001
  OBJECT FILE: X-001
  MODES:       2-BANK

      1 % tab c
      2         IDENTIFICATION DIVISION.
      3         PROGRAM-ID.  X-001.
      4
      5         DATA DIVISION.
      6         WORKING-STORAGE SECTION.
      7         01  NAME              PIC X(30).
      8         01  I                 COMP.
      9
     10         PROCEDURE DIVISION.
     11         1000. BLANK SCREEN.
     12               DISPLAY (10,  1) "Your name:".
     13               ACCEPT  (10, 12) NAME WITH PROMPT.
     14               BLANK LINE 10.
     15               DISPLAY ( 1,  1) FRAME 18 * 75 WITH HEADING.
     16               DISPLAY ( 2, 28) "M y   n a m e   i s".
     17               DO FOR I FROM 4 TO 17
     18                  DISPLAY (I,  3) NAME WITH BLINK
     19                  DISPLAY (I, 42) NAME WITH INVERSE-VIDEO
     20               END-DO.
     21               DISPLAY (22, 10)
     22                  "You have now used the ND COBOL System"
     23                     WITH UNDERLINE.
     24               STOP RUN.
     25

--- END OF COMPILATION ----------
NUMBER OF ERRORS FOUND:          0
NUMBER OF WARNINGS GIVEN:        0
NUMBER OF SOURCE LINES:         25
LINES/MINUTE (CPU TIME):      2571
---------------------------------

*EXIT
@NRL
RELOCATING LOADER LDR-1935I
*PROG-FILE X-001
*LOAD X-001
FREE: 000223-177777 .... FREE DATA AREA: 000464-177777
*LOAD COBOL-2B
FREE: 020173-177777 .... FREE DATA AREA: 006320-177777
*EXIT
@X-001
```

.. and watch the screen.

if you know how to use mode-files (and if you don't, you should know that they are very handy when compiling and loading programs on ND computers), then try executing a :MODE-file with the following contents:

```
@COBOL-H
COMPILE X-001,1,"X-001"
EXIT

@NRL
PROG-FILE "X-001"
LOAD X-001
LOAD COBOL-2BANK
EXIT
```

Then, try executing X-001.

11.1.2 Running the Example on an ND-500 Computer

The following listing of a terminal session shows how to compile, load and execute the program on an ND-500. The underlined parts of the listing is what you have to type in when trying the example.

```
@ND

ND-500 MONITOR  Version F00 84.11.27 / REV.-F01
N500: COBOL-H

ND-500 COBOL COMPILER - ND-10177H

*COMPILE X-001,1,X-001

ND-500 COBOL COMPILER - ND-10177H  TIME: 16.18.17  DATE: 85.02.19

 SOURCE FILE: X-001
 OBJECT FILE: X-001

    1 % tab c
    2        IDENTIFICATION DIVISION.
    3        PROGRAM-ID.  X-001.
    4
    5        DATA DIVISION.
    6        WORKING-STORAGE SECTION.
    7        01  NAME                PIC X(30).
    8        01  I                   COMP.
    9
   10        PROCEDURE DIVISION.
   11        1000. BLANK SCREEN.
   12              DISPLAY (10,  1) "Your name:".
   13              ACCEPT  (10, 12) NAME WITH PROMPT.
   14              BLANK LINE 10.
   15              DISPLAY ( 1,  1) FRAME 18 * 75 WITH HEADING.
   16              DISPLAY ( 2, 28) "M y   n a m e   i s".
   17              DO FOR I FROM 4 TO 17
   18                 DISPLAY (I,  3) NAME WITH BLINK
   19                 DISPLAY (I, 42) NAME WITH INVERSE-VIDEO
   20              END-DO.
   21              DISPLAY (22, 10)
```

```
    22                "You have now used the ND COBOL System"
    23                   WITH UNDERLINE.
    24             STOP RUN.
    25
--- END OF COMPILATION ----------
NUMBER OF ERRORS FOUND:         0
NUMBER OF WARNINGS GIVEN:       0
NUMBER OF SOURCE LINES:        24
LINES/MINUTE (CPU TIME):     2571

---------------------------------
*EXIT

N500: LINK-LOAD

ND-Linkage-Loader -  F          10. September  1983 Time: 00:07
Nll entered:                    19. February   1985 Time: 16:18
Nll: SET-DOMAIN X-001
Nll: LOAD X-001
Program:.........634 P     Data:...........1130 D01
Nll: LOAD COBOL-LIB
COBOL-LIB-H850101
Program:.......72605 P01   Data:..........23530 D01
Nll: LOAD EXCEPT-LIB
EXCEPTION-LIB-2041578
Program:.......72605 P01   Data:..........23530 D01
Nll: EXIT


N500: X-001
```

Then, see what happens.

If you know how to use MODE-files (and if you don't, you should know that they are very handy when compiling and loading programs on ND computers), then try executing a :MODE-file with the following contents:

```
@ND
COBOL
COMPILE X-001,1,"X-001"
EXIT

LINK-LOAD
SET-DOMAIN "X-001"
LOAD X-001
LOAD COBOL-LIB
LOAD EXCEPTION-LIB
EXIT
```

Then, try executing X-001:

```
@ND
ND-500: X-001
```

etc.

11.2 OVERLAY SYSTEMS

Sometimes a program cannot be run in the ND-100 because it is too large to fit into the one-bank address space of 64 pages, or the 64 pages for the program and 64 pages for data allowed when compiling in the two-bank mode. (The ND-500 computers do not have this limitation, thus the present section on overlays is *not relevant to ND-500 installations.*)

A common solution to this problem is to divide the program into reasonably small parts which can be run one at a time, and in such a way that one part (or subprogram) can use the space freed when another subprogram has finished. Thus, the program will only need space for those subprograms that have to be in memory at the same time.

The sets of different subprograms to be loaded one at a time are called ***overlays*** or ***links***, and the process of loading an overlay to replace an existing set of subprograms is called ***overlaying*** these subprograms.

Building overlays with the BRF-Linker or the NRL is a convenient way of bypassing the problem of large programs not being able to fit into the address space because:

1) Programs built as overlay systems do not need source code modification.

2) The Symbolic Debugger is available for overlays.

An overlay structure cannot be made into a reentrant subsystem.

11.2.1 The Multilevel Overlay System

When using the overlay capability on the ND-100, you must understand how your program operates, and especially the relationship between the modules within it.

There are significant differences between the overlay systems built by the NRL and those of the BRF-Linker. If you build your system with the BRF-Linker, the :SYMB-files can be compiled, loaded and executed on an ND-500 computer without modification. If you build it with the NRL, you will have to modify the symbolic code before you can make it run on an ND-500. This is because the NRL makes necessary the inclusion of special subprogram calls in the code.

You are advised to organize your overlay structure (described below) so as to retain in memory the links containing commonly used subprograms, and place the infrequently used subprograms in links which can overlay one another. For example, a special error recovery subprogram would only need to be brought into memory when the corresponding error occurred.

When you load an overlay system and want it to be available to other users, be sure to specify to the linker you are using not only the name of the program file, but also the *name of the user area where it is to be found*. Otherwise, the operating system may not find the overlay subprograms on the default user areas when the overlay system calls them, with errors as a result. And remember, the sum of the length of the user name and the length of the overlay program file name must not exceed 16 characters.

Each link should be a collection of functionally related modules and be as self-contained as possible, calling other links as infrequently as possible. In particular, references to links which would overlay other links should be kept to a minimum.

A tree structure, called an ***overlay structure***, can be used to illustrate the internal dependencies of the overlay links. In a tree structure, each link has only one immediate ancestor, but it may have more than one immediate descendant. The ***root link*** contains the parts of the program which must always be in memory during execution. . Since the root link receives control when the program starts, it does not have an ancestor. The remaining links branch away from the root link and are structured according to their interdependencies.

Links which do not have to be in memory at the same time are termed ***independent links***, whereas links which must be in memory at the same time are termed ***dependent links***. For example, two modules which do not refer to each other or pass data directly to each other, are independent links. When such links are no longer required in memory, they can be overlaid by other links that are brought in. On the other hand, a link must have all the links upon which it depends in memory at the same time and cannot therefore overlay them. Every link is dependent on its ancestor and, consequently, on the root link.

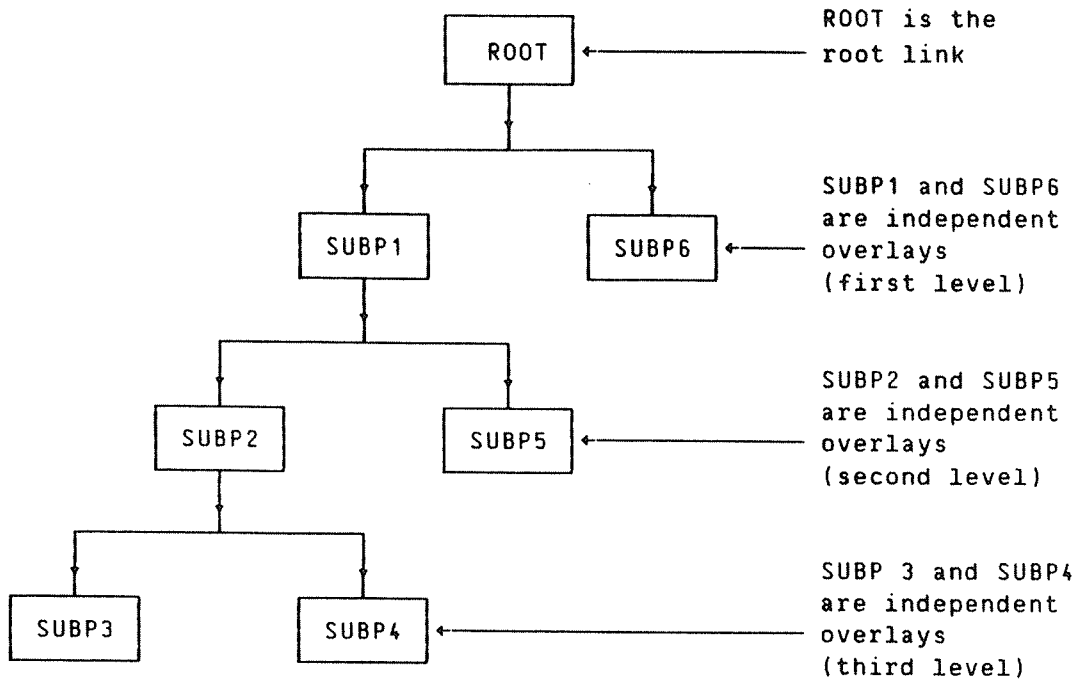
As an illustration, assume we have a program consisting of a main program ROOT and six subprograms SUBP1, SUBP2, SUBP3, SUBP4, SUBP5 and SUBP6. The subprograms are related as follows:

1) SUBP1 and SUBP6 are called directly from ROOT and are independent of each other.

2) SUBP2 and SUBP5 are called directly from SUBP1 and are independent of each other.

3) SUBP3 and SUBP4 are called directly from SUBP2 and are also independent of each other.

The following tree structure illustrates the subprogram dependencies:



SUBP4 depends on SUBP1 and SUBP2, consequently, SUBP1 and SUBP2 must be in memory in order to execute SUBP4. The chain of links which a link depends on is referred to as the ***path*** of the link. The action of bringing a link into memory is termed ***path loading***. The chain of links branching away from a link is known as the ***extended path*** of that link. In the previous example, the path of SUBP4 is ROOT, SUBP1, and SUBP2. There are three extended paths of SUBP1:

1) SUBP2, SUBP3

2) SUBP2, SUBP4

3) SUBP5

A link may communicate with other links that lie in its own path or one of its extended paths. The communication is through references to global symbols. A reference from the current link to a global symbol in another link in the path is called a ***backward reference***, while a reference from the current link to a global symbol in another link on one of its extended paths is called a ***forward reference***. Since all links on the path of the current link must be in memory, a backward reference does not cause any links to be brought into memory. With a forward reference, however, the link referred to may not be in memory. Then it must be fetched, possibly overlaying a link already there.

11.2.2 Designing an Overlay Structure

The first step to be taken when designing an overlay structure is to draw a diagram showing the functional relationships between the modules within the program. The tree begins with the root link which contains the main program and remains in memory throughout execution. The remainder of the program is contained in the overlay links.

The user should remember several points when drawing his overlay structure:

1) References that will overlay existing links should be minimized.

2) Independent links cannot reference each other; communication is by way of a common link.

3) As a general rule, calls to subprograms on other links should be forward references, while returns from subprograms should be backward references.

4) If data is modified during execution, the modification is destroyed once the link is overlaid. Therefore, if data required by another link is modified, then the data must be returned to this other link before the link containing the changed data is overlaid.

5) When a link is to be overlaid, no addresses or references to it should remain.

6) Modules, subprograms or data areas used by several links should be explicitly loaded into a link that is common to all links using these modules or data areas. For example, a FORTRAN COMMON data area should be in a link in the path of all links referencing it. Moreover, COMMON should be positioned in such a way that it never gets reinitialized after the first call. In other programming languages using the distinction between local and global data, similar considerations must be made for the data which are global to several link paths.

7) The Symbolic Debugger should be used with some care on overlays. Debugger commands affecting program/data in an overlay should not be given until a breakpoint is reached on that particular overlay. Moreover, these commands are effective only while the overlay resides in memory. In other words, overlays are always brought fully initialized into memory.

Tree-structured overlay systems can be several levels deep. The amount of memory required for an overlay system is at least equivalent to the size of the "longest" path. This is not the minimum requirement, however, since special tables are needed when a program is divided into links.

The root link (and the COMMON areas defined within it) reside in memory throughout the entire execution, while the overlays (and the COMMON areas defined within them) reside on a random read-only file. This file is specified with the ***PROGRAM-FILE*** command.

11.2.3 Commands for Overlay Loading with BRF-Linker

Here, the overlay linking commands in the BRF-linker are shown. For further details, see the manuals BRF-Linker User Manual (ND-60.196) or ND Relocating Loader (ND-60.066).

Overlay structures are loaded using the same BRF-Linker commands as for normal loading. However, we also need to specify that we are loading a new link in the overlay structure. This is done by the command:

Brl: ***OVERLAY*** *<level>,<entry name 1>[,...,<entry name n>]*

This command specifies that a new overlay link is to be generated. The parameter <level> is the overlay level, and <entry name 1> to <entry name n> give the names of the subprograms that may be called from the previous level. After this command has been given, the specified subprograms can be loaded from one or more BRF files. It is recommended that the overlay subprograms be kept on a separate BRF file compiled in library mode. In this way, the specified set of subprograms may be selected and put into the overlay independently of the compilation sequence.

The level number in an OVERLAY command must not be more than 1 higher than the level number in the previous OVERLAY command.

The special form:

Brl: ***OVERLAY 0,,***

is used to indicate the start of the root link. This should be the first command following the PROGRAM-FILE command.

In 2-bank programs, the special form:

Brl: ***OVERLAY -1,,***

will append the last overlaid data part to the previously appended one. This permits all data to be placed consecutively with no data overlay. Make sure that no previous data overlays share this area with the current data overlay.

To dump the root link, the COMMON area, and the last overlay link onto the file specified in the ***PROGRAM-FILE*** command, use either the ***EXIT*** or the ***RUN*** commands. If you use the ***RUN*** command, the execution of the overlay system will start immediately. Otherwise the execution of the overlay system must be started by a separate command (RECOVER).

11.2.4 Example: Creating an Overlay System with the BRF-Linker

This simple example of an overlay system is built according to the overlay tree structure shown in section 11.2.1.

```
@COBOL
ND-100 COBOL COMPILER - ND-10176H

*COMPILE OVERLAY-PROGRAM:S,TERM,OVERLAY-PROGRAM

ND-100 COBOL COMPILER - ND-10176H     TIME: 09.03.10     DATE: 84.11.20

  SOURCE FILE: OVERLAY-PROGRAM:S
  OBJECT FILE: OVERLAY-PROGRAM
  MODES:       2-BANK

      1 % TAB C
      2       * Previous line sets PED/NOTIS tabulators automatically
      3        IDENTIFICATION DIVISION.
      4        PROGRAM-ID. ROOT.
      5        AUTHOR. IBO.
      6
      7        DATA DIVISION.
      8        WORKING-STORAGE SECTION.
      9        01 OVERLAY-LEVEL COMP.
     10
     11        PROCEDURE DIVISION.
     12
     13        START-ROOT.
     14            MOVE 0 TO OVERLAY-LEVEL.
     15            DISPLAY "MAIN-PROGRAM: OVERLAY LEVEL ", OVERLAY-LEVEL.
     16            CALL "SUBP1" USING OVERLAY-LEVEL.
     17            DISPLAY "MAIN-PROGRAM: OVERLAY LEVEL ", OVERLAY-LEVEL.
     18            CALL "SUBP6" USING OVERLAY-LEVEL.
     19            DISPLAY "MAIN-PROGRAM: OVERLAY LEVEL ", OVERLAY-LEVEL.
     20
     21        END-ROOT.
     22            DISPLAY "END OF OVERLAY PROGRAM EXECUTION. HAVE A NICE DAY!".
     23            STOP RUN.

--- END OF COMPILATION ----------
NUMBER OF ERRORS FOUND:          0
NUMBER OF WARNINGS GIVEN:        0
NUMBER OF SOURCE LINES:         23
LINES/MINUTE (CPU TIME):      3219
---------------------------------

  *COMPILE OVERLAY-SUBP1:S,TERM,OVERLAY-SUBP1

  ND-100 COBOL COMPILER - ND-10176H     TIME: 09.03.13     DATE: 84.11.20

    SOURCE FILE: OVERLAY-SUBP1:S
    OBJECT FILE: OVERLAY-SUBP1
    MODES:       2-BANK
```

```
 1 % TAB C
 2
 3        IDENTIFICATION DIVISION.
 4        PROGRAM-ID. SUBP1.
 5        AUTHOR. IBO.
 6
 7        DATA DIVISION.
 8        LINKAGE SECTION.
 9        01 OVERLAY-LEVEL COMP.
10
11        PROCEDURE DIVISION USING OVERLAY-LEVEL.
12        START-SUBP.
13            ADD 1 TO OVERLAY-LEVEL.
14            DISPLAY "SUBP1 : OVERLAY LEVEL ", OVERLAY-LEVEL.
15            CALL "SUBP2" USING OVERLAY-LEVEL.
16            DISPLAY "SUBP1 : OVERLAY LEVEL ", OVERLAY-LEVEL.
17            CALL "SUBP5" USING OVERLAY-LEVEL.
18            DISPLAY "SUBP1 : OVERLAY LEVEL ", OVERLAY-LEVEL.
19            SUBTRACT 1 FROM OVERLAY-LEVEL.
20
21        END-SUBP.
22            DISPLAY "END OF EXECUTION OF SUBP1.".
23            EXIT PROGRAM.

--- END OF COMPILATION ----------
NUMBER OF ERRORS FOUND:          0
NUMBER OF WARNINGS GIVEN:        0
NUMBER OF SOURCE LINES:         23
LINES/MINUTE (CPU TIME):      2571
--------------------------------
```

```
  *COMPILE OVERLAY-SUBP2:S,TERM,OVERLAY-SUBP2

  ND-100 COBOL COMPILER - ND-10176H     TIME: 09.03.17     DATE: 84.11.20

    SOURCE FILE: OVERLAY-SUBP2:S
    OBJECT FILE: OVERLAY-SUBP2
    MODES:       2-BANK

        1 % TAB C
        2
        3       IDENTIFICATION DIVISION.
        4       PROGRAM-ID. SUBP2.
        5       AUTHOR. IBO.
        6
        7       DATA DIVISION.
        8       LINKAGE SECTION.
        9       01 OVERLAY-LEVEL COMP.
       10
       11       PROCEDURE DIVISION USING OVERLAY-LEVEL.
       12       START-SUBP.
       13           ADD 1 TO OVERLAY-LEVEL.
       14           DISPLAY "SUBP2 : OVERLAY LEVEL ", OVERLAY-LEVEL.
       15           CALL "SUBP3" USING OVERLAY-LEVEL.
       16           DISPLAY "SUBP2 : OVERLAY LEVEL ", OVERLAY-LEVEL.
       17           CALL "SUBP4" USING OVERLAY-LEVEL.
       18           DISPLAY "SUBP2 : OVERLAY LEVEL ", OVERLAY-LEVEL.
       19           SUBTRACT 1 FROM OVERLAY-LEVEL.
       20
       21       END-SUBP.
       22           DISPLAY "END OF EXECUTION OF SUBP2.".
       23           EXIT PROGRAM.

--- END OF COMPILATION ----------
NUMBER OF ERRORS FOUND:         0
NUMBER OF WARNINGS GIVEN:       0
NUMBER OF SOURCE LINES:        23
LINES/MINUTE (CPU TIME):     2490
--------------------------------

  *COMPILE OVERLAY-SUBP6:S,TERM,OVERLAY-SUBP6

  ND-100 COBOL COMPILER - ND-10176H     TIME: 09.03.20     DATE: 84.11.20

    SOURCE FILE: OVERLAY-SUBP6:S
    OBJECT FILE: OVERLAY-SUBP6
    MODES:       2-BANK

        1 % TAB C
        2
        3       IDENTIFICATION DIVISION.
        4       PROGRAM-ID. SUBP6.
        5       AUTHOR. IBO.
        6
        7       DATA DIVISION.
        8       LINKAGE SECTION.
        9       01 OVERLAY-LEVEL COMP.
       10
       11       PROCEDURE DIVISION USING OVERLAY-LEVEL.
       12       START-SUBP.
       13           ADD 1 TO OVERLAY-LEVEL.
       14           DISPLAY "SUBP6 : OVERLAY LEVEL ", OVERLAY-LEVEL.
       15           SUBTRACT 1 FROM OVERLAY-LEVEL.
       16
       17       END-SUBP.
       18           DISPLAY "END OF EXECUTION OF SUBP6.".
       19           EXIT PROGRAM.

--- END OF COMPILATION ----------
NUMBER OF ERRORS FOUND:         0
NUMBER OF WARNINGS GIVEN:       0
NUMBER OF SOURCE LINES:        19
LINES/MINUTE (CPU TIME):     2714
--------------------------------

  *COMPILE OVERLAY-SUBP3:S,TERM,OVERLAY-SUBP3
```

```
  ND-100 COBOL COMPILER - ND-10176H     TIME: 09.03.23     DATE: 84.11.20

    SOURCE FILE: OVERLAY-SUBP3:S
    OBJECT FILE: OVERLAY-SUBP3
    MODES:       2-BANK

        1 % TAB C
        2
        3         IDENTIFICATION DIVISION.
        4         PROGRAM-ID. SUBP3.
        5         AUTHOR. IBO.
        6
        7         DATA DIVISION.
        8         LINKAGE SECTION.
        9         01 OVERLAY-LEVEL COMP.
       10
       11         PROCEDURE DIVISION USING OVERLAY-LEVEL.
       12         START-SUBP.
       13             ADD 1 TO OVERLAY-LEVEL.
       14             DISPLAY "SUBP3 : OVERLAY LEVEL ", OVERLAY-LEVEL.
       15             SUBTRACT 1 FROM OVERLAY-LEVEL.
       16
       17         END-SUBP.
       18             DISPLAY "END OF EXECUTION OF SUBP3.".
       19             EXIT PROGRAM.

--- END OF COMPILATION ----------
NUMBER OF ERRORS FOUND:          0
NUMBER OF WARNINGS GIVEN:        0
NUMBER OF SOURCE LINES:         19
LINES/MINUTE (CPU TIME):      3005
---------------------------------
```

```
*COMPILE OVERLAY-SUBP4:S,TERM,OVERLAY-SUBP4

ND-100 COBOL COMPILER - ND-10176H     TIME: 09.03.25     DATE: 84.11.20

  SOURCE FILE: OVERLAY-SUBP4:S
  OBJECT FILE: OVERLAY-SUBP4
  MODES:       2-BANK

      1 % TAB C
      2
      3       IDENTIFICATION DIVISION.
      4       PROGRAM-ID. SUBP4.
      5       AUTHOR. IBO.
      6
      7       DATA DIVISION.
      8       LINKAGE SECTION.
      9       01 OVERLAY-LEVEL COMP.
     10
     11       PROCEDURE DIVISION USING OVERLAY-LEVEL.
     12       START-SUBP.
     13           ADD 1 TO OVERLAY-LEVEL.
     14           DISPLAY "SUBP4 : OVERLAY LEVEL ", OVERLAY-LEVEL.
     15           SUBTRACT 1 FROM OVERLAY-LEVEL.
     16
     17       END-SUBP.
     18           DISPLAY "END OF EXECUTION OF SUBP4.".
     19           EXIT PROGRAM.

--- END OF COMPILATION ----------
NUMBER OF ERRORS FOUND:          0
NUMBER OF WARNINGS GIVEN:        0
NUMBER OF SOURCE LINES:         19
LINES/MINUTE (CPU TIME):      2877
--------------------------------

*COMPILE OVERLAY-SUBP5:S,TERM,OVERLAY-SUBP5

ND-100 COBOL COMPILER - ND-10176H     TIME: 09.03.28     DATE: 84.11.20

  SOURCE FILE: OVERLAY-SUBP5:S
  OBJECT FILE: OVERLAY-SUBP5
  MODES:       2-BANK

      1 % TAB C
      2
      3       IDENTIFICATION DIVISION.
      4       PROGRAM-ID. SUBP5.
      5       AUTHOR. IBO.
      6
      7       DATA DIVISION.
      8       LINKAGE SECTION.
      9       01 OVERLAY-LEVEL COMP.
     10
     11       PROCEDURE DIVISION USING OVERLAY-LEVEL.
     12       START-SUBP.
     13           ADD 1 TO OVERLAY-LEVEL.
     14           DISPLAY "SUBP5 : OVERLAY LEVEL ", OVERLAY-LEVEL.
     15           SUBTRACT 1 FROM OVERLAY-LEVEL.
     16
     17       END-SUBP.
     18           DISPLAY "END OF EXECUTION OF SUBP5.".
     19           EXIT PROGRAM.

--- END OF COMPILATION ----------
NUMBER OF ERRORS FOUND:          0
NUMBER OF WARNINGS GIVEN:        0
NUMBER OF SOURCE LINES:         19
LINES/MINUTE (CPU TIME):      2685
--------------------------------

*EXIT
```

```
@BRF-LINKER
- BRF Linker -  10721A
Brl: PROGRAM-FILE OVERLAY-PROGRAM
Brl: OVERLAY 0,,
Brl: LOAD OVERLAY-PROGRAM,COBOL-2B
FREE: P 000160-177777     D 000455-177777
FREE: P 003355-177777     D 002644-177777
Brl: OVERLAY 1,SUBP1
Brl: LOAD OVERLAY-SUBP1,COBOL-2B
FREE: P 003560-177777     D 003300-177777     DEBUG 000004
FREE: P 003560-177777     D 003300-177777     DEBUG 000004
Brl: OVERLAY 2,SUBP2
Brl: LOAD OVERLAY-SUBP2,COBOL-2B
FREE: P 003762-177777     D 003734-177777     DEBUG 000010
FREE: P 003762-177777     D 003734-177777     DEBUG 000010
Brl: OVERLAY 3,SUBP3
Brl: LOAD OVERLAY-SUBP3,COBOL-2B
FREE: P 004101-177777     D 004370-177777     DEBUG 000014
FREE: P 004101-177777     D 004370-177777     DEBUG 000014
Brl: OVERLAY 3,SUBP4
OVERLAY COMPLETED. BLOCK NO: 2001    4006-4101/3734-4370
SUBP3.....4011 P  *.........4101 P
*.........4370 D

Brl: LOAD OVERLAY-SUBP4,COBOL-2B
FREE: P 004101-177777     D 004370-177777     DEBUG 000024
FREE: P 004101-177777     D 004370-177777     DEBUG 000024
Brl: OVERLAY 2,SUBP5
OVERLAY COMPLETED. BLOCK NO: 2004    4006-4101/3734-4370
SUBP4.....4011 P  *.........4101 P
*.........4370 D

OVERLAY COMPLETED. BLOCK NO: 2007    3604-4006/3300-3734
SUBP2.....3607 P  *.........4006 P
*.........3734 D

Brl: LOAD OVERLAY-SUBP5,COBOL-2B
FREE: P 003677-177777     D 003734-177777     DEBUG 000040
FREE: P 003677-177777     D 003734-177777     DEBUG 000040
Brl: OVERLAY 1,SUBP6
OVERLAY COMPLETED. BLOCK NO: 2012    3604-3677/3300-3734
SUBP5.....3607 P  *.........3677 P
*.........3734 D

OVERLAY COMPLETED. BLOCK NO: 2015    3402-3604/2644-3300
SUBP1.....3405 P  *.........3604 P
*.........3300 D

Brl: LOAD OVERLAY-SUBP6,COBOL-2B
FREE: P 003475-177777     D 003300-177777     DEBUG 000054
FREE: P 003475-177777     D 003300-177777     DEBUG 000054
Brl: EXIT
OVERLAY COMPLETED. BLOCK NO: 2020    3402-3475/2644-3300
SUBP6.....3405 P  *.........3475 P
*.........3300 D
```

```
@OVERLAY-PROGRAM

MAIN-PROGRAM: OVERLAY LEVEL 00000+
SUBP1 : OVERLAY LEVEL 00001+
SUBP2 : OVERLAY LEVEL 00002+
SUBP3 : OVERLAY LEVEL 00003+
END OF EXECUTION OF SUBP3.
SUBP2 : OVERLAY LEVEL 00002+
SUBP4 : OVERLAY LEVEL 00003+
END OF EXECUTION OF SUBP4.
SUBP2 : OVERLAY LEVEL 00002+
END OF EXECUTION OF SUBP2.
SUBP1 : OVERLAY LEVEL 00001+
SUBP5 : OVERLAY LEVEL 00002+
END OF EXECUTION OF SUBP5.
SUBP1 : OVERLAY LEVEL 00001+
END OF EXECUTION OF SUBP1.
MAIN-PROGRAM: OVERLAY LEVEL 00000+
SUBP6 : OVERLAY LEVEL 00001+
END OF EXECUTION OF SUBP6.
MAIN-PROGRAM: OVERLAY LEVEL 00000+
END OF OVERLAY PROGRAM EXECUTION. HAVE A NICE DAY!
```

11.2.5 subprograms and Commands for Building an Overlay System with the NRL

When building an overlay system to be loaded with the NRL, the following system included subprograms *must* be called:

```
CALL "OVLINIT".
CALL "OVERLAY" USING "<sub-name>" [, <parameter>] ... .
```

where *<sub-name>* is the program-id of the subprogram to be overlaid and *[, <parameter>] ...* denotes the possible parameters to be submitted to that subprogram. *Note:* The *<sub-name>* must be *precisely* seven characters long. If the program-id of the program to be overlaid is too short, blanks must be filled in to make it long enough.

CALL "OVLINIT" is used in the root node to initiate the node for linking to the overlays.

Use CALL "OVERLAY" in the source code where you want to read in and execute an overlaid subprogram. This call is completely analogous to the subprogram calls used in non-overlaid systems, except that the overlay has to be read into your logical memory before you can transfer control to it.

The subprogram "OVRECAL" can be used as an alternative to "OVERLAY". OVERLAY *forces* the overlaid subprogram to be read in. OVRECAL *does not* read the overlaid program if it has been read before, but has nato been overlaid by another program.

The commands given to the NRL during loading are explained in the following example.

11.2.6 Example: Creating an Overlay System with the NRL

This simple example of an overlay system is built according to the overlay tree structure shown in section 11.2.1.

```
@COBOL-H
ND-100 COBOL COMPILER - ND-10176H
*COMPILE OVER-NRL-PROG:S,TERM,OVER-NRL-PROGRAM
ND-100 COBOL COMPILER - ND-10176H     TIME: 15.47.03     DATE: 85.02.01
  SOURCE FILE: OVER-NRL-PROG:S
  OBJECT FILE:-OVER-NRL-PROGRAM
  MODES:       2-BANK

     1 % TAB C
     2       * Previous line instructs PED to set COBOL tabulators.
     3        IDENTIFICATION DIVISION.
     4        PROGRAM-ID. ROOT.
     5        AUTHOR. IBO.
     6
     7        DATA DIVISION.
     8        WORKING-STORAGE SECTION.
     9        01 OVERLAY-LEVEL COMP.
    10
    11        PROCEDURE DIVISION.
    12
    13        START-ROOT.
    14       ******************************************************************
    15       * Note the following lines. When using NRL to load overlay sys-
    16       * tems, special subprograms must be used to handle the overlay
    17       * calls. This makes code conversion necessary when the program
    18       * system is transferred to an ND-500 or loaded with the BRF-Lin-
    19       * ker.
    20       ******************************************************************
    21            CALL "OVLINIT".
    22       *          :......... "OVLINIT" initializes the overlay system.
    23            MOVE 0 TO OVERLAY-LEVEL.
    24            DISPLAY "MAIN-PROGRAM: OVERLAY LEVEL ", OVERLAY-LEVEL.
    25       ******************************************************************
    26       * Beware of the following trap: The number of characters inside
    27       * the " (quote) signs in the parameter transmitting the name
    28       * of the overlaid subprogram must be exactly 7, like this one:
    29       *                                  =========
    30       *               "SUBP1  "
    31       *               -1234567-
    32       ******************************************************************
    33            CALL "OVERLAY" USING "SUBP1  " OVERLAY-LEVEL.
    34       *          :          This is the proper call to the
    35       *          :          overlaid subprogram when
    36       *          :......... loading with the NRL.
    37            DISPLAY "MAIN-PROGRAM: OVERLAY LEVEL ", OVERLAY-LEVEL.
    38            CALL "OVERLAY" USING "SUBP6  " OVERLAY-LEVEL.
    39       *          :......... Another overlay call.
    40            DISPLAY "MAIN-PROGRAM: OVERLAY LEVEL ", OVERLAY-LEVEL.
    41       ******************************************************************
    42       * The call to "OVERLAY" with the subprogram name as a parameter
    43       * is also used for further overlay calls in the following
    44       * subprograms.
    45       ******************************************************************
    46
    47        END-ROOT.
```

```
    48            DISPLAY "END OF OVERLAY PROGRAM EXECUTION. HAVE A NICE DAY!".
    49            STOP RUN.

 --- END OF COMPILATION ----------
 NUMBER OF ERRORS FOUND:          0
 NUMBER OF WARNINGS GIVEN:        0
 NUMBER OF SOURCE LINES:         49
 LINES/MINUTE (CPU TIME):      2571
 --------------------------------

*COMPILE OVER-NRL-SUBP1:S,TERM,OVER-NRL-SUBP1

ND-100 COBOL COMPILER - ND-10176H     TIME: 15.47.07     DATE: 85.02.01

  SOURCE FILE: OVER-NRL-SUBP1:S
  OBJECT FILE: OVER-NRL-SUBP1
  MODES:       2-BANK

     1 % TAB C
     2
     3        IDENTIFICATION DIVISION.
     4        PROGRAM-ID. SUBP1.
     5        AUTHOR. IBO.
     6
     7        DATA DIVISION.
     8        LINKAGE SECTION.
     9        01 OVERLAY-LEVEL COMP.
    10
    11        PROCEDURE DIVISION USING OVERLAY-LEVEL.
    12        START-SUBP.
    13            ADD 1 TO OVERLAY-LEVEL.
    14            DISPLAY "SUBP1 : OVERLAY LEVEL ", OVERLAY-LEVEL.
    15            CALL "OVERLAY" USING "SUBP2  " OVERLAY-LEVEL.
    16            DISPLAY "SUBP1 : OVERLAY LEVEL ", OVERLAY-LEVEL.
    17            CALL "OVERLAY" USING "SUBP5  " OVERLAY-LEVEL.
    18            DISPLAY "SUBP1 : OVERLAY LEVEL ", OVERLAY-LEVEL.
    19            SUBTRACT 1 FROM OVERLAY-LEVEL.
    20
    21        END-SUBP.
    22            DISPLAY "END OF EXECUTION OF SUBP1.".
    23            EXIT PROGRAM.

 --- END OF COMPILATION ----------
 NUMBER OF ERRORS FOUND:          0
 NUMBER OF WARNINGS GIVEN:        0
 NUMBER OF SOURCE LINES:         23
 LINES/MINUTE (CPU TIME):      2571
 --------------------------------

*COMPILE OVER-NRL-SUBP2:S,TERM,OVER-NRL-SUBP2

ND-100 COBOL COMPILER - ND-10176H     TIME: 15.47.10     DATE: 85.02.01

  SOURCE FILE: OVER-NRL-SUBP2:S
  OBJECT FILE: OVER-NRL-SUBP2
  MODES:       2-BANK

     1 % TAB C
     2
     3        IDENTIFICATION DIVISION.
     4        PROGRAM-ID. SUBP2.
     5        AUTHOR. IBO.
     6
     7        DATA DIVISION.
     8        LINKAGE SECTION.
     9        01 OVERLAY-LEVEL COMP.
    10
    11        PROCEDURE DIVISION USING OVERLAY-LEVEL.
    12        START-SUBP.
    13            ADD 1 TO OVERLAY-LEVEL.
    14            DISPLAY "SUBP2 : OVERLAY LEVEL ", OVERLAY-LEVEL.
    15            CALL "OVERLAY" USING "SUBP3  " OVERLAY-LEVEL.
    16            DISPLAY "SUBP2 : OVERLAY LEVEL ", OVERLAY-LEVEL.
    17            CALL "OVERLAY" USING "SUBP4  " OVERLAY-LEVEL.
    18            DISPLAY "SUBP2 : OVERLAY LEVEL ", OVERLAY-LEVEL.
    19            SUBTRACT 1 FROM OVERLAY-LEVEL.
    20
```

```
    21          END-SUBP.
    22              DISPLAY "END OF EXECUTION OF SUBP2.".
    23              EXIT PROGRAM.

 --- END OF COMPILATION ----------
 NUMBER OF ERRORS FOUND:          0
 NUMBER OF WARNINGS GIVEN:        0
 NUMBER OF SOURCE LINES:         23
 LINES/MINUTE (CPU TIME):      2604
 --------------------------------

*COMPILE OVER-NRL-SUBP6:S,TERM,OVER-NRL-SUBP6

ND-100 COBOL COMPILER - ND-10176H     TIME: 15.47.13     DATE: 85.02.01

  SOURCE FILE: OVER-NRL-SUBP6:S
  OBJECT FILE: OVER-NRL-SUBP6
  MODES:       2-BANK

     1 % TAB C
     2
     3          IDENTIFICATION DIVISION.
     4          PROGRAM-ID. SUBP6.
     5          AUTHOR. IBO.
     6
     7          DATA DIVISION.
     8          LINKAGE SECTION.
     9          01 OVERLAY-LEVEL COMP.
    10
    11          PROCEDURE DIVISION USING OVERLAY-LEVEL.
    12          START-SUBP.
    13              ADD 1 TO OVERLAY-LEVEL.
    14              DISPLAY "SUBP6 : OVERLAY LEVEL ", OVERLAY-LEVEL.
    15              SUBTRACT 1 FROM OVERLAY-LEVEL.
    16
    17          END-SUBP.
    18              DISPLAY "END OF EXECUTION OF SUBP6.".
    19              EXIT PROGRAM.

 --- END OF COMPILATION ----------
 NUMBER OF ERRORS FOUND:          0
 NUMBER OF WARNINGS GIVEN:        0
 NUMBER OF SOURCE LINES:         19
 LINES/MINUTE (CPU TIME):      3163
 --------------------------------

*COMPILE OVER-NRL-SUBP3:S,TERM,OVER-NRL-SUBP3

ND-100 COBOL COMPILER - ND-10176H     TIME: 15.47.15     DATE: 85.02.01

  SOURCE FILE: OVER-NRL-SUBP3:S
  OBJECT FILE: OVER-NRL-SUBP3
  MODES:       2-BANK

     1 % TAB C
     2
     3          IDENTIFICATION DIVISION.
     4          PROGRAM-ID. SUBP3.
     5          AUTHOR. IBO.
     6
     7          DATA DIVISION.
     8          LINKAGE SECTION.
     9          01 OVERLAY-LEVEL COMP.
    10
    11          PROCEDURE DIVISION USING OVERLAY-LEVEL.
    12          START-SUBP.
    13              ADD 1 TO OVERLAY-LEVEL.
    14              DISPLAY "SUBP3 : OVERLAY LEVEL ", OVERLAY-LEVEL.
    15              SUBTRACT 1 FROM OVERLAY-LEVEL.
    16
    17          END-SUBP.
    18              DISPLAY "END OF EXECUTION OF SUBP3.".
    19              EXIT PROGRAM.

 --- END OF COMPILATION ----------
 NUMBER OF ERRORS FOUND:          0
 NUMBER OF WARNINGS GIVEN:        0
```

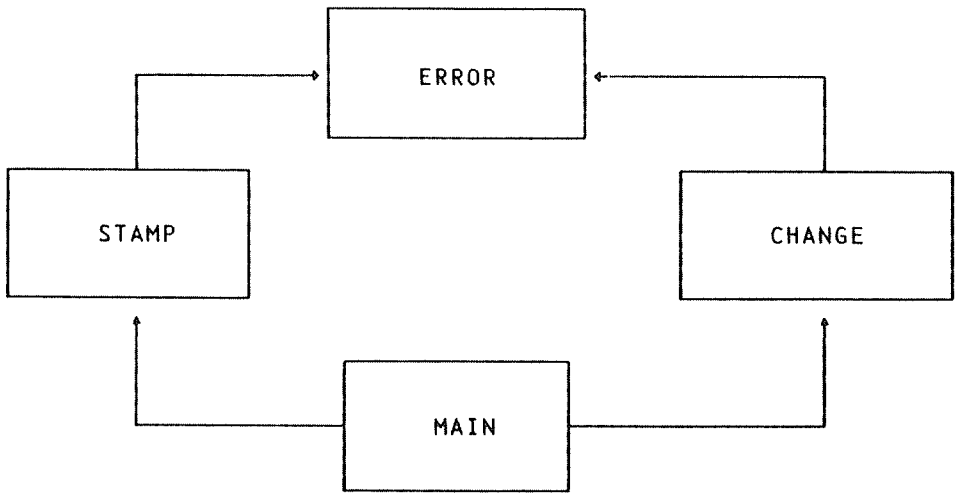
```
 NUMBER OF SOURCE LINES:        19
 LINES/MINUTE (CPU TIME):     2690
 --------------------------------

*COMPILE OVER-NRL-SUBP4:S,TERM,OVER-NRL-SUBP4

ND-100 COBOL COMPILER - ND-10176H     TIME: 15.47.18     DATE: 85.02.01

  SOURCE FILE: OVER-NRL-SUBP4:S
  OBJECT FILE: OVER-NRL-SUBP4
  MODES:       2-BANK

     1 % TAB C
     2
     3        IDENTIFICATION DIVISION.
     4        PROGRAM-ID. SUBP4.
     5        AUTHOR. IBO.
     6
     7        DATA DIVISION.
     8        LINKAGE SECTION.
     9        01 OVERLAY-LEVEL COMP.
    10
    11        PROCEDURE DIVISION USING OVERLAY-LEVEL.
    12        START-SUBP.
    13            ADD 1 TO OVERLAY-LEVEL.
    14            DISPLAY "SUBP4 : OVERLAY LEVEL ", OVERLAY-LEVEL.
    15            SUBTRACT 1 FROM OVERLAY-LEVEL.
    16
    17        END-SUBP.
    18            DISPLAY "END OF EXECUTION OF SUBP4.".
    19            EXIT PROGRAM.

 --- END OF COMPILATION ----------
 NUMBER OF ERRORS FOUND:         0
 NUMBER OF WARNINGS GIVEN:       0
 NUMBER OF SOURCE LINES:        19
 LINES/MINUTE (CPU TIME):     2361
 --------------------------------

*COMPILE OVER-NRL-SUBP5:S,TERM,OVER-NRL-SUBP5

ND-100 COBOL COMPILER - ND-10176H     TIME: 15.47.20     DATE: 85.02.01

  SOURCE FILE: OVER-NRL-SUBP5:S
  OBJECT FILE: OVER-NRL-SUBP5
  MODES:       2-BANK

     1 % TAB C
     2
     3        IDENTIFICATION DIVISION.
     4        PROGRAM-ID. SUBP5.
     5        AUTHOR. IBO.
     6
     7        DATA DIVISION.
     8        LINKAGE SECTION.
     9        01 OVERLAY-LEVEL COMP.
    10
    11        PROCEDURE DIVISION USING OVERLAY-LEVEL.
    12        START-SUBP.
    13            ADD 1 TO OVERLAY-LEVEL.
    14            DISPLAY "SUBP5 : OVERLAY LEVEL ", OVERLAY-LEVEL.
    15            SUBTRACT 1 FROM OVERLAY-LEVEL.
    16
    17        END-SUBP.
    18            DISPLAY "END OF EXECUTION OF SUBP5.".
    19            EXIT PROGRAM.

 --- END OF COMPILATION ----------
 NUMBER OF ERRORS FOUND:         0
 NUMBER OF WARNINGS GIVEN:       0
 NUMBER OF SOURCE LINES:        19
 LINES/MINUTE (CPU TIME):     3007
 --------------------------------

*EXIT
```

@CC The the SET-MODE commands given to the NRL on the following

```
@CC command lines are necessary with all COBOL programs which
@CC are not compiled with the *1-BANK-MODE option on.

@NRL
RELOCATING LOADER LDR-1935I
*PROG-FILE OVER-NRL-PROGRAM
*SET-MODE DATA
*OVERLAY-GENERATION 7
*SET-MODE PROG
*LOAD OVER-NRL-PROGRAM COBOL-2B
FREE: 003475-177777 .... FREE DATA AREA: 002742-177777
*OVERLAY-ENTRY (1) SUBP1
*LOAD OVER-NRL-SUBP1 COBOL-2B
OVERLAY 1 LEVEL  1 COMPLETED. AREA: 003475-003704   002742-003411
  SUBP1=P 003500
*OVERLAY-ENTRY (2) SUBP2
*LOAD OVER-NRL-SUBP2 COBOL-2B
OVERLAY 2 LEVEL  2 COMPLETED. AREA: 003705-004114   003412-004061
  SUBP2=P 003710
*OVERLAY-ENTRY (3) SUBP3
*LOAD OVER-NRL-SUBP3 COBOL-2B
OVERLAY 3 LEVEL  3 COMPLETED. AREA: 004115-004207   004062-004515
  SUBP3=P 004120
*OVERLAY-ENTRY (3) SUBP4
*LOAD OVER-NRL-SUBP4 COBOL-2B
OVERLAY 4 LEVEL  3 COMPLETED. AREA: 004115-004207   004062-004515
  SUBP4=P 004120
*OVERLAY-ENTRY (2) SUBP5
*LOAD OVER-NRL-SUBP5 COBOL-2B
OVERLAY 5 LEVEL  2 COMPLETED. AREA: 003705-003777   003412-004045
  SUBP5=P 003710
*OVERLAY-ENTRY (1) SUBP6
*LOAD OVER-NRL-SUBP6 COBOL-2B
OVERLAY 6 LEVEL  1 COMPLETED. AREA: 003475-003567   002742-003375
  SUBP6=P 003500
*EXIT
@OVER-NRL-PROGRAM

MAIN-PROGRAM: OVERLAY LEVEL 00000+
SUBP1 : OVERLAY LEVEL 00001+
SUBP2 : OVERLAY LEVEL 00002+
SUBP3 : OVERLAY LEVEL 00003+
END OF EXECUTION OF SUBP3.
SUBP2 : OVERLAY LEVEL 00002+
SUBP4 : OVERLAY LEVEL 00003+
END OF EXECUTION OF SUBP4.
SUBP2 : OVERLAY LEVEL 00002+
END OF EXECUTION OF SUBP2.
SUBP1 : OVERLAY LEVEL 00001+
SUBP5 : OVERLAY LEVEL 00002+
END OF EXECUTION OF SUBP5.
SUBP1 : OVERLAY LEVEL 00001+
END OF EXECUTION OF SUBP1.
MAIN-PROGRAM: OVERLAY LEVEL 00000+
SUBP6 : OVERLAY LEVEL 00001+
END OF EXECUTION OF SUBP6.
MAIN-PROGRAM: OVERLAY LEVEL 00000+
END OF OVERLAY PROGRAM EXECUTION. HAVE A NICE DAY!
```

11.3 BUILDING A NON-OVERLAY FILE-HANDLING PROGRAM SYSTEM

The purpose of this section is to show how program systems other than overlay systems are built in ND COBOL. Points to remember when building this and similar program systems are included as comments in the symbolic source code listings and as annotations to the terminal sessions which are printed together with the sessions. The program system has the following call structure:



MAIN is the main program, and will eventually be the name of the loaded program system. It consists of paragraphs for listing errors (logged by the ERROR subprogram) and for calling the subprogram CHANGE which edits the indexed file where the time and date of the most recent activity is kept, in addition to the entry and exit paragraphs.

Each time one of these paragraphs are performed, the STAMP subprogram is called from the paragraph in question, and writes the time and date the paragraph was executed on the appropriate record in the indexed file ISAM-EX. If an error condition or exception occurs during the use of the indexed file in the STAMP or CHANGE subprograms, the ERROR subprogram is called from the DECLARATIVES sections of those programs. ERROR then identifies the error/exception from the variables passed to it, and writes the error number together with an explanatory text, and time and date of occurrence, on the sequential file ERROR-LOG:DATA. It is this error list that can be viewed by users executing the MAIN program.

In the examples below, the finished program systems are built in the simplest possible ways. The "system supervision" programs MAIN and CHANGE are used to monitor the functioning of the system, while STAMP and ERROR could be loaded together with other systems to record the activities in these systems. Then the programmer might choose to open the ISAM-EX file in the MULTI-USER MODE, and to make the other systems reentrant or public in other ways, while use of the MAIN program system would be limited. If that is the case, more advanced loading procedures would be required. Please consult the relevant loader manuals for details.

This is what the program looks like when it is executed. See if you can find out what goes on from the description above!

```
@MAIN

STAMP called with key = ENTRY
----------------------------------------.
USE CAPITALS AS RESPONSES IN THIS PROGRAM.
----------------------------------------.
L(ist errors), C(hange keys), E(xit) ? C
Calling CHANGE subprogram to see and change ISAM-EX.
C(hange), D(elete), N(ew), L(ist), else exit: L
Key CHANGE message: CHANGEing the ISAM-EX file     DATE: 850228 TIME: 16033324
Key ENTRY  message: Entering the program system    DATE: 850228 TIME: 16035554
Key EXIT   message: EXIT from the program system   DATE: 850228 TIME: 16033364
Key FORTRA message: A FORTRAN program called       DATE: *NEW*  TIME: *NEW*
Key LI-ERR message: Last time errors were listed   DATE: 850228 TIME: 16035864
C(hange), D(elete), N(ew), L(ist), else exit: E
Leaving CHANGE subprogram
STAMP called with key = CHANGE
L(ist errors), C(hange keys), E(xit) ? L
94 Error flag set                                  STAMP  day:850228 time:15532706
97 File access violation                           STAMP  day:850228 time:15532774
97 File access violation                           STAMP  day:850228 time:15532794
94 Error flag set                                  CHANGE day:850228 time:15533522
97 File access violation                           CHANGE day:850228 time:15533998
97 File access violation                           CHANGE day:850228 time:15534382
94 Error flag set                                  STAMP  day:850228 time:15535792
97 File access violation                           STAMP  day:850228 time:15535814
97 File access violation                           STAMP  day:850228 time:15535832
94 Error flag set                                  CHANGE day:850228 time:15541526
99 SINTRAN file error                              CHANGE day:850228 time:15542004
97 File access violation                           CHANGE day:850228 time:15542714
97 File access violation                           CHANGE day:850228 time:15551130
98 Wrong file description                          CHANGE day:850228 time:15551490
99 SINTRAN file error                              STAMP  day:850228 time:15551522
97 File access violation                           STAMP  day:850228 time:15551546
97 File access violation                           STAMP  day:850228 time:15551574
98 Wrong file description                          STAMP  day:850228 time:15551684
99 SINTRAN file error                              STAMP  day:850228 time:15551954
97 File access violation                           STAMP  day:850228 time:15551978
97 File access violation                           STAMP  day:850228 time:15551996
98 Wrong file description                          STAMP  day:850228 time:15552018
STAMP called with key = LI-ERR
L(ist errors), C(hange keys), E(xit) ? E
STAMP called with key = EXIT
JOB DONE - EXIT TO OPERATING SYSTEM.
```

11.3.1 Sample Programs - Source Listings

Here is the source code for the programs in the system.

```
% TAB C
      ****************************************************************
      * Previous line instructs PED to set COBOL tabulators.
      ****************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. MAIN-PROGRAM.
       AUTHOR. IBO.
      ****************************************************************
      * This is the main program in a system consisting of this
      * program and several subprograms which is constructed for
      * demonstration purposes.
      *
      * The programs illustrate how to call subprograms from
      * other COBOL programs, how to handle file errors in the
      * DECLARATIVES section, and how files should be specified
      * in relationship to where they are used.
      *
      * Note that all files to be used in a subprogram must be
      * closed upon entry of that subprogram. The subprogram must
      * close all its files before returning control to the calling
      * program.
      *
      * An indexed and a sequential file are used, but since the
      * indexed file is not used in the MAIN program, it is not
      * defined here, whereas the sequential error-logging file is
      * defined below, since current errors can be listed by the
      * MAIN program.
      *
      * The file "ERROR-LOG:DATA" must also exist before it is used.
      *
      * For the indexed file, the prime record keys must be unique, and
      * "ISAM-EX:DATA" must exist and be empty, while "ISAM-EX:ISAM"
      * must not exist or be empty before the first run. The first
      * time the program is executed, there are no keys in the indexed
      * file: they must be entered by the user. During this process,
      * plenty errors will be logged on the error file to show how the
      * system works. The keys which MUST be present in the indexed
      * file if no errors shall occur are:
      *    - ENTRY  ( program system is started.     )
      *    - EXIT   ( program system is stopped.     )
      *    - CHANGE ( the ISAM file is edited.       )
      *    - LI-ERR ( the current errors are listed. )
      *
      * These keys can be put on the ISAM-EX file by repeated use
      * of the "C(hange)" command in this program and the "N(ew)"
      * command in the CHANGE subprogram.
      *
      * Other keys may be added cover other uses of the system,
      * like when the subprograms that stamp the ISAM-EX file are
      * loaded into other program systems.
      ****************************************************************

       ENVIRONMENT DIVISION.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT ERROR-FILE ASSIGN "ERROR-LOG:DATA"
                  ORGANIZATION IS SEQUENTIAL
                  ACCESS IS SEQUENTIAL.

       DATA DIVISION.

       FILE SECTION.
       FD ERROR-FILE.
       01 ERROR-REC.
```

```
    02 E-STATUS PIC XX.
    02 E-TEXT.
        03 E-TYPE PIC X(40).
        03 E-NUMBER PIC X(3).
        03 E-PROGRAM PIC X(6).
    02 E-DATE PIC X(6).
    02 E-TIME PIC X(8).

WORKING-STORAGE SECTION.
01 ANSWER PIC X.
01 TRANSMIT-KEY PIC X(6).
01 ISAM-STATUS PIC XX.
01 TRANSMIT-STATUS PIC XX.

PROCEDURE DIVISION.
*****************************************************************
* These procedures monitor the activities of the users
* and makes extensive notes of their activities.
*****************************************************************
STAMP-ON-ENTRY.
    MOVE "ENTRY" TO TRANSMIT-KEY.
    CALL "STAMP-ISAM" USING ISAM-STATUS, TRANSMIT-KEY.
    DISPLAY "----------------------------------------".
    DISPLAY "USE CAPITALS AS RESPONSES IN THIS PROGRAM.".
    DISPLAY "----------------------------------------".

CHOICE.
    DISPLAY
        "L(ist errors), C(hange keys), E(xit) ? "
            ACCEPT ANSWER.
    IF ANSWER = "E" THEN PERFORM FINI
    ELSE-IF ANSWER = "L" THEN PERFORM LIST-ERRORS
    ELSE-IF ANSWER = "C" THEN PERFORM CHANGE
    ELSE GO TO CHOICE
    END-IF.

CHANGE.
    DISPLAY
        "Calling CHANGE subprogram    see and change ISAM-EX.".
    CALL "CHANGE-ISAM" USING TRANSMIT-KEY.
    MOVE "CHANGE" TO TRANSMIT-KEY.
    CALL "STAMP-ISAM" USING ISAM-STATUS, TRANSMIT-KEY.
    GO TO CHOICE.

LIST-ERRORS.
    OPEN INPUT ERROR-FILE.
    DO
            PERFORM READ-ERROR-REC
    END-DO.

READ-ERROR-REC.
    READ ERROR-FILE AT END PERFORM END-ERROR-LIST.
    DISPLAY           E-STATUS,
            " ",      E-TYPE,
            " ",      E-NUMBER,
            " ",      E-PROGRAM,
            " day:",  E-DATE,
            " time:", E-TIME.

END-ERROR-LIST.
    CLOSE ERROR-FILE.
    MOVE "LI-ERR" TO TRANSMIT-KEY.
    CALL "STAMP-ISAM" USING ISAM-STATUS, TRANSMIT-KEY.
    PERFORM CHOICE.

FINI.
    MOVE "EXIT" TO TRANSMIT-KEY.
    CALL "STAMP-ISAM" USING ISAM-STATUS, TRANSMIT-KEY.
    DISPLAY "JOB DONE - EXIT TO OPERATING SYSTEM.".
    STOP RUN.
```

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. STAMP-ISAM.
 AUTHOR. IBO.
*****************************************************************
* This is a subprogram to be called from the MAIN program,
* and from programs in other languages than COBOL.
* See comments to the MAIN for an explanation of the
* details concerning definitions of common files.
*
* Note that the VALUE OF FILE-ID clause in the FILE SECTION of
* the DATA DIVISION is no longer needed. Also note that in this
* version of the program, all special ND COBOL screen handling
* facilities have been removed. That is, there are no position
* clauses to the ACCEPT and DISPLAY sentences in this program,
* and the BLANK SCREEN sentence is not used at all. This is done
* to get the same kind of screen handling as the calling programs,
* and to avoid having the COBOL subprogram disable the ESCAPE key,
* as it does when these special sentences are first used.
*
* See comments to the COBOL main program for an explanation of the
* details concerning definitions of common files.
*****************************************************************

 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT ISAM-FILE ASSIGN TO "ISAM-EX:DATA",
            ORGANIZATION IS INDEXED,
            ACCESS MODE IS DYNAMIC,
            RECORD KEY IS ISAM-KEY,
            FILE STATUS IS ISAM-STATUS.

 DATA DIVISION.
 FILE SECTION.
 FD ISAM-FILE.
 01 ISAM-RECORD.
     02 ISAM-KEY PIC X(6).
     02 ISAM-TEXT.
         03 ISAM-MESSAGE PIC X(30).
         03 ISAM-DATE PIC X(6).
         03 ISAM-TIME PIC X(8).
 WORKING-STORAGE SECTION.
 01 ISAM-STATUS PIC XX.
 01 ERROR-NUMBER COMP.
*****************************************************************
* Note that the area ISAM-STATUS is for storing file-status
* after operations on files. It might be tempting to place this
* in the LINKAGE SECTION for easy transmission to the calling
* program; however, this is not allowed.
*****************************************************************
 LINKAGE SECTION.
 01 TRANSMIT-STATUS PIC XX.
 01 TRANSMIT-KEY PIC X(6).

 PROCEDURE DIVISION USING TRANSMIT-STATUS, TRANSMIT-KEY.

 DECLARATIVES.
 S1 SECTION.
     USE AFTER ERROR PROCEDURE ON ISAM-FILE.
 ISAM-ERROR.
     IF ISAM-STATUS = "99" THEN
         CALL "ISERR" USING ERROR-NUMBER
         CALL "CBERMSG" USING ERROR-NUMBER
     END-IF.
     MOVE "STAMP" TO TRANSMIT-KEY.
     CALL "ERROR-ISAM" USING ISAM-STATUS, ERROR-NUMBER,
         TRANSMIT-KEY.
 END DECLARATIVES.

 SUBP-ACTIONS SECTION.
```

```
 OPEN-FILE.
     OPEN I-O ISAM-FILE.
*****************************************************************
* Note that files must be opened and closed in each subpro-
* gram where they are used, even if these operations are also
* performed in the calling program.
*****************************************************************

 WRITE-STAMP.
     MOVE TRANSMIT-KEY TO ISAM-KEY.
     DISPLAY "STAMP called with key = ", ISAM-KEY.
     READ ISAM-FILE KEY IS ISAM-KEY
         INVALID KEY
         PERFORM INVALID-KEY.
     ACCEPT ISAM-DATE FROM DATE.
     ACCEPT ISAM-TIME FROM TIME.
     REWRITE ISAM-RECORD.
     PERFORM EXIT-PARAGRAPH.

 INVALID-KEY.
     DISPLAY "<STAMP> TRIED TO STAMP A NONEXISTENT KEY!".

 EXIT-PARAGRAPH.
     MOVE ISAM-STATUS TO TRANSMIT-STATUS.
     CLOSE ISAM-FILE.
     EXIT PROGRAM.
*****************************************************************
* All shared files must be closed before exit from subprogram.
*****************************************************************
```

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. CHANGE-ISAM.
 AUTHOR. IBO.
*****************************************************************
* This is a subprogram to be called from the MAIN program.
* See comments to that program for an explanation of the
* details concerning definitions of common files.
*****************************************************************

 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT ISAM-FILE ASSIGN TO "ISAM-EX:DATA",
            ORGANIZATION IS INDEXED,
            ACCESS MODE IS DYNAMIC,
            RECORD KEY IS ISAM-KEY,
            FILE STATUS IS ISAM-STATUS.

 DATA DIVISION.
 FILE SECTION.
 FD ISAM-FILE.
 01 ISAM-RECORD.
     02 ISAM-KEY PIC X(6).
     02 ISAM-TEXT.
         03 ISAM-MESSAGE PIC X(30).
         03 ISAM-DATE PIC X(6).
         03 ISAM-TIME PIC X(8).

 WORKING-STORAGE SECTION.
 01 ANSWER PIC X.
 01 ISAM-STATUS PIC XX.
 01 CHANGE-KEY PIC X(6).
 01 TRANSMIT-CALLING PIC X(6).
 01 ERROR-NUMBER COMP.
*****************************************************************
* Note that ERROR-NUMBER will be used for transmission of
* SINTRAN error numbers according to the initial DECLARATIVES
* of the PROCEDURE DIVISION and related procedures.
*****************************************************************
 LINKAGE SECTION.
 01 TRANSMIT-STATUS PIC XX.

 PROCEDURE DIVISION USING TRANSMIT-STATUS.

*****************************************************************
* The following paragraph shows how error-messages pertaining to
* the file used throughout this program can be recorded and
* displayed. It is entered each time an error in the use of
* the file is detected.
*
* If the file status bytes (ISAM-STATUS in this program) are set
* to '99', then the error is a SINTRAN error. Such errors will
* cause an exit from the program, with an error message to the
* display from SINTRAN if the program does not contain a
* DECLARATIVES section.
*
* It may be desirable that the program can handle these and
* other errors itself, without any exit to the operating system.
* Here, all errors are recorded on a sequential file called
* ERROR-LOG. In the case of SINTRAN errors, the SINTRAN error
* number is recovered through using the ISERR subprogram.
* Subsequently, the system monitor call MON64 is called by the
* subprogram CBERMSG to get the error-message displayed.
*
* All monitor calls which can be called as subprograms from
* COBOL have names consisting of the names found in the SINTRAN
* Reference Manual, prefixed with the letters CB.
*
* Note that the actions taken by these routines (including
* CBERMSG) are not ruled by the COBOL screen handling verbs;
* therefore they will disturb the screen picture these verbs
* generate.
*****************************************************************
```

```
DECLARATIVES.
S1 SECTION.
    USE AFTER ERROR PROCEDURE ON ISAM-FILE.

ISAM-ERROR.
    IF ISAM-STATUS = "99" THEN
        CALL "ISERR" USING ERROR-NUMBER
        CALL "CBERMSG" USING ERROR-NUMBER
    END-IF.
    MOVE "CHANGE" TO TRANSMIT-CALLING.
    CALL "ERROR-ISAM" USING ISAM-STATUS, ERROR-NUMBER,
        TRANSMIT-CALLING.
END DECLARATIVES.
*****************************************************************
* Note that if a DECLARATIVES SECTION is used, the program must
* be subdivided into sections.
*****************************************************************

MAIN-ACTIONS SECTION.

OPEN-FILE.
    OPEN I-O ISAM-FILE.
    MOVE SPACES TO ISAM-STATUS.

CHOICE.
    DISPLAY "C(hange), D(elete), N(ew), L(ist), else exit: "
        ACCEPT ANSWER.
    IF ANSWER = "C" THEN PERFORM CHANGE-REC
    ELSE-IF ANSWER = "D" THEN PERFORM DELETE-REC
    ELSE-IF ANSWER = "N" THEN PERFORM NEW-REC
    ELSE-IF ANSWER = "L" THEN PERFORM LIST
    ELSE PERFORM EXIT-PARAGRAPH.

*****************************************************************
* The following paragraph lists the contents of the ISAM-EX file.
* Note how the primary key, ISAM-KEY, is initialized.
*
* Previous versions of ND-COBOL permitted statements such as
* "START ISAM-FILE."; however, this non-standard format is now
* removed.
*****************************************************************
LIST.
    MOVE SPACES TO ISAM-KEY.
    START ISAM-FILE KEY > ISAM-KEY.
    DO
        READ ISAM-FILE NEXT AT END GO TO CHOICE.
        DISPLAY "Key ",       ISAM-KEY,
                " message: ", ISAM-MESSAGE,
                " DATE: ",    ISAM-DATE,
                " TIME: ",    ISAM-TIME.
    END-DO.

CHANGE-REC.
    DISPLAY "KEY? " ACCEPT ISAM-KEY.
    READ ISAM-FILE KEY IS ISAM-KEY
        INVALID KEY
        PERFORM INVALID-KEY.
    DISPLAY "Current key is     -> ", ISAM-KEY.
    DISPLAY "Current message is -> ", ISAM-MESSAGE.
    DISPLAY "Stamped at: ", ISAM-DATE, " : ", ISAM-TIME.
    DISPLAY "New message -> " ACCEPT ISAM-MESSAGE.
    REWRITE ISAM-RECORD.
    GO TO CHOICE.

DELETE-REC.
    DISPLAY "KEY? " ACCEPT ISAM-KEY.
    READ ISAM-FILE KEY IS ISAM-KEY
        INVALID KEY
        PERFORM INVALID-KEY.
    DISPLAY "Current record is -> ", ISAM-KEY, " : ",
        ISAM-MESSAGE, " : ", ISAM-DATE, " : ", ISAM-TIME.
    DISPLAY "DELETE OK? (Y/N) " ACCEPT ANSWER.
    IF ANSWER = "Y" THEN DELETE ISAM-FILE
        ELSE DISPLAY "Record not deleted.".

NEW-REC.
```

```
           DISPLAY "KEY? " ACCEPT ISAM-KEY.
           READ ISAM-FILE KEY IS ISAM-KEY.
           IF ISAM-STATUS = "00" THEN PERFORM RECORD-EXISTED.
           PERFORM ENTER-KEY.
           GO TO CHOICE.

       RECORD-EXISTED.
           DISPLAY "This key exists already!".
           GO TO CHOICE.

       ENTER-KEY.
           DISPLAY "Message? -> " ACCEPT ISAM-MESSAGE.
           MOVE "*NEW* " TO ISAM-DATE, ISAM-TIME.
           DISPLAY "New record is -> ", ISAM-KEY, " : ",
               ISAM-MESSAGE, " : ", ISAM-DATE, " : ", ISAM-TIME.
           WRITE ISAM-RECORD.

       INVALID-KEY.
           DISPLAY "Did not find the key you asked for.".
           GO TO CHOICE.

       EXIT-PARAGRAPH.
           MOVE ISAM-STATUS TO TRANSMIT-STATUS.
           CLOSE ISAM-FILE.
           DISPLAY "Leaving CHANGE subprogram".
           EXIT PROGRAM.
      ******************************************************************
      * All shared files must be closed before exit from subprogram.
      ******************************************************************
```

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. ERROR-ISAM.
 AUTHOR. IBO.
*****************************************************************
* Logs ISAM file errors.
*****************************************************************

 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT ERROR-FILE ASSIGN "ERROR-LOG:DATA"
         ORGANIZATION IS SEQUENTIAL
         ACCESS IS SEQUENTIAL.

 DATA DIVISION.
 FILE SECTION.
 FD ERROR-FILE.
 01 ERROR-REC.
     02 E-STATUS PIC XX.
     02 E-TEXT.
         03 E-TYPE PIC X(40).
         03 E-NUMBER PIC X(3).
         03 E-PROGRAM PIC X(6).
     02 E-DATE PIC X(6).
     02 E-TIME PIC X(8).

 WORKING-STORAGE SECTION.
 01  ANSWER PIC X.

 LINKAGE SECTION.
     01 ISAM-STATUS PIC XX.
     01 ERROR-CODE COMP.
     01 ERROR-SOURCE PIC X(6).

 PROCEDURE DIVISION USING ISAM-STATUS, ERROR-CODE, ERROR-SOURCE.

*****************************************************************
* The ERROR-FILE is opened in append mode with the EXTEND
* option, so that each new record is written after the end of
* the file.
*****************************************************************

 INITIATION.
     OPEN EXTEND ERROR-FILE.

 COMPOSE-REC.
     MOVE ISAM-STATUS TO E-STATUS.
     IF ERROR-CODE NOT EQUAL ZERO THEN
         MOVE ERROR-CODE TO E-NUMBER.
     MOVE ERROR-SOURCE TO E-PROGRAM.
     ACCEPT E-DATE FROM DATE.
     ACCEPT E-TIME FROM TIME.
     IF ISAM-STATUS = '99' THEN MOVE
         'SINTRAN file error' TO  E-TYPE
     ELSE-IF ISAM-STATUS = '98' THEN MOVE
         'Wrong file description' TO E-TYPE
     ELSE-IF ISAM-STATUS = '97' THEN MOVE
         'File access violation' TO E-TYPE
     ELSE-IF ISAM-STATUS = '95' THEN MOVE
         'File not initialised or opened' TO E-TYPE
     ELSE-IF ISAM-STATUS = '94' THEN MOVE
         'Error flag set' TO E-TYPE
     ELSE-IF ISAM-STATUS = '78' THEN MOVE
         'Record modified by another program' TO E-TYPE
     ELSE-IF ISAM-STATUS = '68' THEN MOVE
         'Record locked by another program' TO E-TYPE
     ELSE-IF ISAM-STATUS = '23' THEN MOVE
         'Record not in ISAM-EX' TO E-TYPE
     ELSE-IF ISAM-STATUS = '22' THEN MOVE
         'Duplicates not allowed' TO E-TYPE
     ELSE-IF ISAM-STATUS = '21' THEN MOVE
         'Wrong sequence of words' TO E-TYPE
     ELSE MOVE
         'Unknown error type on file ISAM-EX' TO E-TYPE.
```

```
WRITE-REC.
    DISPLAY
        'ERROR is called by the DECLARATIVES SECTION of ',
            E-PROGRAM.
    DISPLAY E-STATUS, " ", E-TYPE, "/", E-NUMBER,
        "/", E-PROGRAM, " DAY ", E-DATE, " TIME ", E-TIME.
    WRITE ERROR-REC.
    CLOSE ERROR-FILE.

FINI.
    EXIT PROGRAM.
```

11.3.2 Compiling and Loading the Program System on an ND-100

The following example shows how the previous programs can be compiled and loaded on an ND-100. Here, the program listings are sent to the LINE-PRINTER. They could of course go to any other printing device available, to the terminal (just write TERMINAL instead of LINE-PRINTER), or to a file. If using a file, type the name of the file you want to keep the listing on (here, you can use MAIN:LIST or anything else) - within quotes if the file does not exist already (thus: "MAIN:LIST").

Also note that here, the files MAIN:BRF, SUBP:BRF, CHANGE:BRF and ERROR:BRF exist before the compiling session starts. If they did not exist, you would have to create them by typing their names in quotes ("). When creating relocatable code files like this, the compiler knows that their extension shall be :BRF, and you can skip typing the extensions.

```
@COBOL

ND-100 COBOL COMPILER - ND-10177H

*COMPILE MAIN,LINE-PRINTER,MAIN

--- END OF COMPILATION ----------
NUMBER OF ERRORS FOUND:          0
NUMBER OF WARNINGS GIVEN:        0
NUMBER OF SOURCE LINES:        132
---------------------------------
*COMPILE SUBP,LINE-PRINTER,SUBP

--- END OF COMPILATION ----------
NUMBER OF ERRORS FOUND:          0
NUMBER OF WARNINGS GIVEN:        0
NUMBER OF SOURCE LINES:        102
---------------------------------
*COMPILE CHANGE,LINE-PRINTER,CHANGE

--- END OF COMPILATION ----------
NUMBER OF ERRORS FOUND:          0
NUMBER OF WARNINGS GIVEN:        0
NUMBER OF SOURCE LINES:        180
---------------------------------
```

```
*COMPILE ERROR,LINE-PRINTER.ERROR

--- END OF COMPILATION ----------
NUMBER OF ERRORS FOUND:          0
NUMBER OF WARNINGS GIVEN:        0
NUMBER OF SOURCE LINES:         88
-------------------------------
*EXIT
```

The following lines show how these programs are linked together to form an executable program system. The sequence the programs are loaded in is arbitrary, as long as you remember to load the COBOL library last.

There are two COBOL libraries, each containing subprograms that your programs will need to communicate with terminals, files and so on. If you compile your programs in the 1-BANK mode (see appendix 6) then you must specify COBOL-1BANK as your library. Otherwise the programs are compiled in 2-BANK mode, and you must specify COBOL-2BANK as your library.

Also note that here, the file MAIN:PROG exists before the loading session starts. If this hadn't been the case, you would have to create it by typing its name within quotes ("). When creating executable code files with quotes, the BRF-Linker knows that their extension shall be :PROG, so you can save some finger energy by leaving that out.

To get a precise description of how the BRF-Linker works, see the BRF-Linker User Manual, ND-60.196.01.

```
@BRF-LINKER
- BRF Linker - 10721A
Brl: PROG-FILE MAIN
Brl: LOAD MAIN
FREE: P 000552-177777     D 002721-177777
Brl: LOAD SUBP-ISAM
FREE: P 001162-177777     D 003511-177777
Brl: LOAD CHANGE-ISAM
FREE: P 002474-177777     D 004517-177777
Brl: LOAD ERROR-ISAM
FREE: P 003475-177777     D 007527-177777
Brl: LOAD COBOL-2B
FREE: P 054721-177777     D 043106-177777
Brl: EXIT
```

Optionally, COBOL programs can be loaded on the ND-100 with the NRL (ND Relocating Loader). A loading session with the NRL is almost exactly the same as with the BRF-Linker, as long as you are not building an overlay system. Therefore, the above remarks concerning the BRF-Linker also hold for the NRL.

The following session builds the same program as the previous one.

```
@NRL
RELOCATING LOADER LDR-1935I
*PROG-FILE MAIN
*LOAD MAIN
FREE: 000552-177777 .... FREE DATA AREA: 002721-177777
*LOAD SUBP-ISAM
FREE: 001162-177777 .... FREE DATA AREA: 003511-177777
*LOAD CHANGE-ISAM
FREE: 002474-177777 .... FREE DATA AREA: 004517-177777
*LOAD ERROR-ISAM
FREE: 003475-177777 .... FREE DATA AREA: 007527-177777
*LOAD COBOL-2B
FREE: 054721-177777 .... FREE DATA AREA: 043106-177777
*EXIT
```

11.3.3 Compiling and Loading the Program System on an ND-500

The following example shows how the previous programs can be compiled and loaded on an ND-500. Here, the program listings are sent to the LINE-PRINTER. They could of course go to any other printing device available, to the terminal (just write TERMINAL instead of LINE-PRINTER) or to a file. If using a file, type the name of the file you want to keep the listing on (here, you can use MAIN:LIST or anything else), in quotes if it does not exist already (thus: "MAIN:LIST").

Also note that here, the files MAIN:NRF, SUBP:NRF, CHANGE:NRF and ERROR:NRF exist before the compiling session starts. If this hadn't been the case, you would have to create them by typing their names in quotes (thus: "MAIN"). When creating relocatable code files like this, the compiler knows that their extension shall be :NRF, so you do not have to type the extensions.

Note that the name of the compiler is prefixed with 'ND-500' in this example. ND-500 is the name of the program in the ND-100 which provides the operating environment for and monitors the activities of the ND-500 part of the computer system. It is possible to do most of the work on the ND-500 computer in this environment - for details, see the manual 'ND-500 Loader/Monitor', ND-60.136.

```
@ND-500 COBOL

ND-500 COBOL COMPILER - ND-10177H

*COMPILE MAIN,LINE-PRINTER,MAIN

--- END OF COMPILATION ----------
NUMBER OF ERRORS FOUND:          0
NUMBER OF WARNINGS GIVEN:        0
NUMBER OF SOURCE LINES:        132
LINES/MINUTE (CPU TIME):      6947
---------------------------------
*COMPILE SUBP,LINE-PRINTER,SUBP

--- END OF COMPILATION ----------
```

```
NUMBER OF ERRORS FOUND:          0
NUMBER OF WARNINGS GIVEN:        0
NUMBER OF SOURCE LINES:        102
LINES/MINUTE (CPU TIME):      8500
--------------------------------
*COMPILE CHANGE,LINE-PRINTER,CHANGE

--- END OF COMPILATION ----------
NUMBER OF ERRORS FOUND:          0
NUMBER OF WARNINGS GIVEN:        0
NUMBER OF SOURCE LINES:        180
LINES/MINUTE (CPU TIME):      5400
--------------------------------
*COMPILE ERROR,LINE-PRINTER,ERROR

--- END OF COMPILATION ----------
NUMBER OF ERRORS FOUND:          0
NUMBER OF WARNINGS GIVEN:        0
NUMBER OF SOURCE LINES:         88
LINES/MINUTE (CPU TIME):      4400
--------------------------------
*EXIT
```

When loading the program system on the ND-500, the appropriate loader is the Linkage-Loader. For some introductory remarks on ND-500 programs and points to remember, see p. 13 in this manual. For a full explanation, see the manual 'ND-500 Loader/Monitor', ND-60.136. The following loading session shows the simplest possible way of loading our program system.

Note the inclusion of the EXCEPT-LIB. This library file contains routines which a COBOL program will require if it uses indexed or relative files.

```
@ND-500 LINKAGE-LOADER

ND-Linkage-Loader - F          10. September  1983 Time: 00:07
Nil entered:                   28. February   1985 Time: 18: 0
Nil: SET-DOMAIN MAIN
Nil: OPEN-SEGMENT MAIN,,,
Nil: LOAD MAIN
Program:.........1167 P01   Data:...........6150 D01
Nil: LOAD SUBP
Program:.........2147 P01   Data:..........11264 D01
Nil: LOAD CHANGE-ISAM
Program:........10134 P01   Data:..........21610 D01
Nil: LOAD ERROR-ISAM
Program:........11511 P01   Data:..........30174 D01
Nil: LOAD COBOL-LIB
COBOL-LIB-H850101
COBOL-LIB-H850101
Program:.......236046 P01   Data:.........145144 D01
Nil: LOAD EXCEPT-LIB
EXCEPTION-LIB-204157B
EXCEPTION-LIB-204157B
EXCEPTION-LIB-204157B
Program:.......252054 P01   Data:.........171556 D01
Nil: EXIT
```

11.3.4 Calling COBOL Subprograms from FORTRAN on the ND-100

In the following example, the subprograms STAMP and ERROR (described in the previous sections of this chapter) are loaded together with a FORTRAN main program. The main program does nothing more than to call STAMP to leave a mark on the indexed file saying which program called and when it called. The STAMP program will log all errors and exceptional conditions on the ERROR-LOG file by calling the ERROR subprogram from its DECLARATIVES section.

```
@FORTRAN-100
ND-100/NORD-10 ANSI 77 FORTRAN COMPILER - 203053D
FTN: SEPARATE
FTN: COMPILE FORTRAN-PROGRAM,1,FORTRAN-PROGRAM


ND-100/NORD-10 ANSI 77 FORTRAN COMPILER - 203053D     13:59   4 MAR 1985
SOURCE FILE:   FORTRAN-PROGRAM:SYMB

    1*      % TAB F
    2*      C The line above sets PED tabulator stops to suit FORTRAN.
    3*            PROGRAM FOROUT
    4*      C*************************************************************************
    5*      C                                                                        *
    6*      C              CALLING COBOL FROM FORTRAN ON AN ND-100                   *
    7*      C              ======================================                    *
    8*      C                                                                        *
    9*      C This FORTRAN code shows how a COBOL program may be called from a       *
   10*      C FORTRAN program. The COBOL subprogram is part of another program sys-  *
   11*      C tem  used as an example elsewhere in this manual. It stamps a record   *
   12*      C in an ISAM file according to the ISAMKY which is passed to it, and     *
   13*      C writes an error message on an error log if something is wrong with     *
   14*      C the file system or else. The keys must be entered into the ISAM file   *
   15*      C before this program makes use of it, otherwise the ISAM file cannot    *
   16*      C be stamped.                                                            *
   17*      C                                                                        *
   18*      C Note the way the characters "FORTRA" are passed to the COBOL program   *
   19*      C  - they cannot  be passed as a CHARACTER substring.                    *
   20*      C                                                                        *
   21*      C The next three lines define the variables that will be passed to       *
   22*      C the COBOL subprogram.                                                  *
   23*      C*************************************************************************
   24*            INTEGER ISAMST
   25*            INTEGER ISAMKY
   26*            DIMENSION ISAMKY(3)
   27*            WRITE (1,100)
   28*      C*************************************************************************
   29*      C Now, the variables ISAMST and ISAMKY must have values assigned to      *
   30*      C them - note how the ASCII characters are passed to the ISAMKY array.   *
   31*      C Since the ND-100 has 16 bit word length, each word can hold two bytes, *
   32*      C and these have to be assigned as follows.                              *
   33*      C*************************************************************************
   34*            ISAMST = "00"
   35*            ISAMKY(1) = "FO"
   36*            ISAMKY(2) = "RT"
   37*            ISAMKY(3) = "RA"
   38*      C*************************************************************************
   39*      C The COBOL subprogram is then called.                                   *
   40*      C*************************************************************************
```

```
   41*            CALL STAMP(ISAMST,ISAMKY)
   42*            WRITE (1,200) ISAMST
   43*      100   FORMAT (///' FORTRAN main program here ... '///)
   44*      200   FORMAT (///' Returns from COBOL with ISAM-STATUS ',/1X,A2//)
   45*            END

- CPU TIME USED: 2.6 SECONDS.  45 LINES COMPILED.
- NO MESSAGES
- PROGRAM SIZE=64 DATA SIZE=106 COMMON SIZE=0
FTN: EXIT

@BRF-LINKER
- BRF Linker -  10721A
Brl: PROG-FILE FORTRAN-PROG
Brl: LOAD FORTRAN-PROGRAM
FREE: P 000100-177777     D 000152-177777
Brl: LOAD SUBP
FREE: P 000510-177777     D 000742-177777
Brl: LOAD ERROR
FREE: P 001511-177777     D 003752-177777
Brl: LOAD COBOL-2B
FREE: P 042572-177777     D 037260-177777
Brl: LOAD FORTRAN-2B
FREE: P 067704-177777     D 045740-177777
Brl: LIST-ENTRIES-UNDEFINED
Brl: EXIT
```

11.3.5 Calling COBOL Subprograms from FORTRAN on the ND-500

In the following example, the subprograms STAMP and ERROR are loaded together with a FORTRAN main program. The main program does nothing more than to call STAMP to leave a mark on the indexed file saying which program called and when it called. The STAMP program will log all errors and exceptional conditions on the ERROR-LOG file by calling the ERROR subprogram from its DECLARATIVES section.

```
@ND-500 FORTRAN-500

ND-500 ANSI 77 FORTRAN COMPILER - 203054H
FTN:
FTN: COMPILE FORTRAN-ND-PROG,1,FORTRAN-ND-PROG


ND-500 ANSI 77 FORTRAN COMPILER - 203054H     14:00   4 MAR 1985
SOURCE FILE:   FORTRAN-ND-PROG:SYMB

    1*      % TAB F
    2*      C `font=1;
    3*            PROGRAM FOROUT
    4*      C*************************************************************************
    5*      C                                                                        *
    6*      C              CALLING COBOL FROM FORTRAN ON AN ND-500                   *
    7*      C              ======================================                    *
    8*      C                                                                        *
    9*      C                                                                        *
   10*      C This FORTRAN code shows how a COBOL program may be called from a       *
   11*      C FORTRAN program. The COBOL subprogram is similar to another program    *
   12*      C system  used as an example elsewhere in this manual. It stamps a rec-  *
```

```
   13*      C ord in an ISAM file according to the ISAMKEY which is passed to it,    *
   14*      C and writes an error message on an error log if something is wrong with *
   15*      C the file system or else. The keys must be entered into the ISAM file   *
   16*      C before this program makes use of it, otherwise the ISAM file cannot    *
   17*      C be stamped.                                                            *
   18*      C                                                                        *
   19*      C In the corresponding COBOL subprograms, all special screen handling    *
   20*      C facilities have been removed - i. e., all position clauses have been   *
   21*      C removed from the DISPLAY and ACCEPT sentences, and all BLANK SCREEN    *
   22*      C sentences have been removed. This has been done for two reasons:       *
   23*      C     - to avoid having the ESCAPE key disabled (this is done automa-    *
   24*      C       tically once the special COBOL screen handling facilities are    *
   25*      C       used                                                             *
   26*      C     - to make the screen handling of the COBOL subprograms consistent  *
   27*      C       with that of the FORTRAN program                                 *
   28*      C                                                                        *
   29*      C Note the way the characters "FORTRA" are passed to the COBOL program   *
   30*      C by using an integer array called ISAMKY  - they cannot  be passed      *
   31*      C as a CHARACTER substring.                                              *
   32*      C                                                                        *
   33*      C The next three lines define the variables that will be passed to       *
   34*      C the COBOL subprogram.                                                  *
   35*      C*************************************************************************
   36*            INTEGER ISAMST
   37*            INTEGER ISAMKY
   38*            DIMENSION ISAMKY(2)
   39*      C*************************************************************************
   40*      C Now, the variables ISAMST and ISAMKY must have values assigned to      *
   41*      C them - note how the ASCII characters are passed to the ISAMKY array.   *
   42*      C                                                                        *
   43*      C Since the ND-500 can store four bytes in each word, the first four     *
   44*      C bytes must be stored in the first word of the integers which hold the  *
   45*      C text to be passed, and the next two bytes must be held in the next     *
   46*      C word.                                                                  *
   47*      C*************************************************************************
   48*            ISAMST = "00"
   49*            ISAMKY(1) = "FORT"
   50*            ISAMKY(2) = "RA"
   51*      C*************************************************************************
   52*      C The COBOL subprogram is then called.                                   *
   53*      C*************************************************************************
   54*            WRITE (1,100)
   55*            CALL STAMP(ISAMST,ISAMKY)
   56*            WRITE (1,200) ISAMST
   57*      100   FORMAT (///' FORTRAN main program here ... '///)
   58*      200   FORMAT (///' Returns from COBOL with ISAM-STATUS ',/1X,A2//)
   59*            END

- CPU TIME USED: 0.5 SECONDS.  59 LINES COMPILED.
- NO MESSAGES
- PROGRAM SIZE=113 DATA SIZE=248 COMMON SIZE=0
FTN: EXIT


@ND-500 LINKAGE-LOADER


ND-Linkage-loader -  F          10. September  1983 Time: 00:07
Nll entered:                     4. March      1985 Time: 14: 1
Nll: SET-DOMAIN FORTRAN-ND-PROG
Nll: OPEN-SEGMENT FORTRAN-ND-PROG,,
Nll: LOAD-SEGMENT FORTRAN-ND-PROG,,
Program:.........165 P      Data:............374 D01
Nll: LOAD SUBP
Program:.........1145 P01   Data:............3510 D01
Nll: LOAD ERROR
Program:.........2605 P01   Data:...........12364 D01
Nll: LOAD COBOL-LIB
COBOL-LIB-H850101
COBOL-LIB-H850101
Program:.......116531 P01   Data:.........106750 D01
Nll: LOAD FORTRAN-LIB
FORTRAN-LIB-203101H
FORTRAN-LIB-203101H
FORTRAN-LIB-203101H
FORTRAN-LIB-203101H
Program:.......162346 P01   Data:.........135560 D01
```

```
Nil: LOAD EXCEP-LIB
EXCEPTION-LIB-204157B
EXCEPTION-LIB-204157B
EXCEPTION-LIB-204157B
Program:......200273 P01   Data:.........162402 D01
Nil: CLOSE
Nil: EXIT
```

A P P E N D I X 1

COMPOSITE LANGUAGE SKELETON

This appendix contains the complete syntax of ND COBOL. It is intended to display complete and syntactically correct formats used throughout this manual.

1.1. NOTATION USED IN FORMATS

1.1.1. Definition of a General Format

A general format is the specific arrangement of the elements of a clause or a statement. (1) A clause or a statement consists of elements as defined below. Throughout this manual, a format is shown adjacent to information defining the clause or statement. When more than one specific arrangement is permitted, the General Format is separated into numbered formats. Clauses must be written in the sequence given in the General Format. (Clauses that are optional must, if they are used, appear in the sequence shown.) In certain cases, stated explicitly in the rules associated with a given format, clauses may appear in sequences other than shown. Applications, requirements or restrictions are shown as rules.

1.1.1.1. Elements

Elements which make up a clause or a statement consist of uppercase words, lowercase words, level-numbers, brackets, braces, connectives, and special characters.

1.1.1.2. Words

All underlined uppercase words are called key words and are required when the functions of which they are a part are used. Uppercase words which are not underlined are optional to the user and need not be written in the source program. Uppercase words, whether underlined or not, must be spelled correctly.

Lowercase words, in a General Format, are generic terms used to represent COBOL words, literals, PICTURE character-strings, or a complete syntactical entry that must be supplied by the user. Where generic terms are repeated in a General Format, a number or letter appendage to the term serves to identify that term for explanation or discussion.

---

(1) These definitions are identical to those of the CODASYL COBOL committee.

1.1.1.3. Level-Numbers

When specific level-numbers appear in Data Description entry formats, those specific level-numbers are required when such entries are used in a COBOL program. In this document, the form 01, 02...09 is used to indicate level-numbers 1 through 9.

1.1.1.4. Brackets, Braces and Choice Indicators

When brackets, [ ], enclose a portion of a General Format, one of the options contained within the brackets may be explicitly specified or that portion of the General Format may be omitted.

When braces, { }, enclose a portion of a General Format, one of the options contained within the braces must either be explicitly specified or implicitly selected. If one and only one of the options contains only reserved words which are not key words, that option is the default option and is implicitly selected unless one of the options is explicitly specified.

When choice indicators, {[ ]}, enclose a portion of the General Format, one or more of the unique options contained within the choice indicators must be specified, but a single option may be specified only once.

Options are indicated in a General Format or a portion of a General Format by vertically stacking alternative possibilities, by a series of brackets, braces or choice indicators or by a combination of both. An option is selected by specifying one of the possibilities, from a stack of alternative possibilities, or by specifying a unique combination of possibilities from a series of brackets, braces or choice indicators.

1.1.1.5. The Ellipsis

In text, other than the General Formats, the ellipsis indicates that one or more words have been omitted. This is allowed only if comprehension is not imaired. This is the conventional meaning of the ellipsis, and this use becomes apparent in context.

In a General Format, the ellipsis represents optional repetition of a portion of a format which is determined as follows:

Given '...'(the ellipsis) in a format, scanning right to left, determine the ']' (right bracket) or } (right brace) delimiter immediately to the left of the '...'; continue scanning right to left and determine the logically matching '[' (left bracket) or '{' (left brace) delimiter; the '...' applies to the portion of the format between the determined pair of delimiters.

1.1.1.6. Format Punctuation

The separator '.' (period), when used in formats, has the status of a required word.

1.1.1.7. Use of Special Characters in Formats

Special characters, when appearing in formats, although not underlined, are required when such portions of the formats are used.

Identification Division

GENERAL FORMAT FOR IDENTIFICATION DIVISION

```
IDENTIFICATION DIVISION

PROGRAM-ID. program-name.

[AUTHOR. [comment-entry] ...]

[INSTALLATION. [comment-entry] ...]

[DATE-WRITTEN. [comment-entry] ...]

[DATE-COMPILED. [comment-entry] ...]

[SECURITY. [comment-entry] ...]

[REMARKS. [comment-entry] ...]
```

Environment Division

GENERAL FORMAT FOR ENVIRONMENT DIVISION

```
ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER.   computer-name  [WITH DEBUGGING MODE] .

OBJECT-COMPUTER.   computer-name
        [, SEGMENT-LIMIT IS  segment-number] .

[SPECIAL-NAMES.

        [, CURRENCY SIGN IS  literal]
        [, DECIMAL-POINT IS COMMA] ] .

[
INPUT-OUTPUT SECTION.
]

  FILE-CONTROL.
        {file-control-entry} ...
  I-O-CONTROL.
        [SAME AREA FOR  file-name-1 {, file-name-2} ...] ...]
```

GENERAL FORMAT FOR FILE CONTROL ENTRY

FORMAT 1:

```
SELECT  [OPTIONAL]  file-name
        ASSIGN TO  assignment-name-1
        [
        [; RESERVE  integer-1 [AREA ]]
                              [AREAS]
        ]
        [; ORGANIZATION IS SEQUENTIAL]
        [; ACCESS MODE IS SEQUENTIAL]
        [; FILE STATUS IS  data-name-1] .
```

FORMAT 2:

```
SELECT  [OPTIONAL]  file-name
        ASSIGN TO  assignment-name-1
        [
        [; RESERVE  integer-1 [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-3] .
```

Environment Division

GENERAL FORMAT FOR FILE CONTROL ENTRY

FORMAT 3:

```
SELECT  [OPTIONAL]  file-name
        ASSIGN TO  assignment-name-1
        [
        [; RESERVE  integer-1 [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] .
```

FORMAT 4:

```
SELECT  file-name  ASSIGN TO  assignment-name-1 .
```

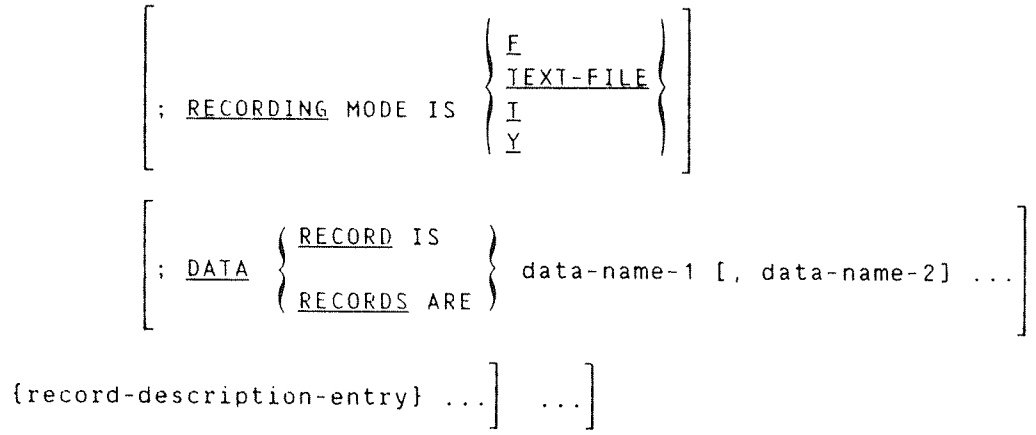
Data Division

GENERAL FORMAT FOR DATA DIVISION

```
DATA DIVISION.
[FILE SECTION.
[FD  file-name
        [; BLOCK CONTAINS [integer-1 TO] integer-2 {RECORDS    }]
                                                   {CHARACTERS}
        [; RECORD CONTAINS [integer-3 TO] integer-4 CHARACTERS
                [DEPENDING ON  identifier] ]
        [; LABEL {RECORD IS  } {STANDARD}]
                 {RECORDS ARE} {OMITTED }
        [VALUE OF FILE-ID IS integer]
        [; RECORDING MODE IS {F          }]
                             {TEXT-FILE  }
                             {I          }
                             {Y          }
        [; DATA {RECORD IS  } data-name-3 [, data-name-4] ...] .
                {RECORDS ARE}
[record-description-entry] ...] ...
[SD file-name
        [; RECORD CONTAINS [integer-1 TO] integer-2 CHARACTERS
                [DEPENDING ON  identifier] ]
```

Data Division

GENERAL FORMAT FOR DATA DIVISION



Data Division

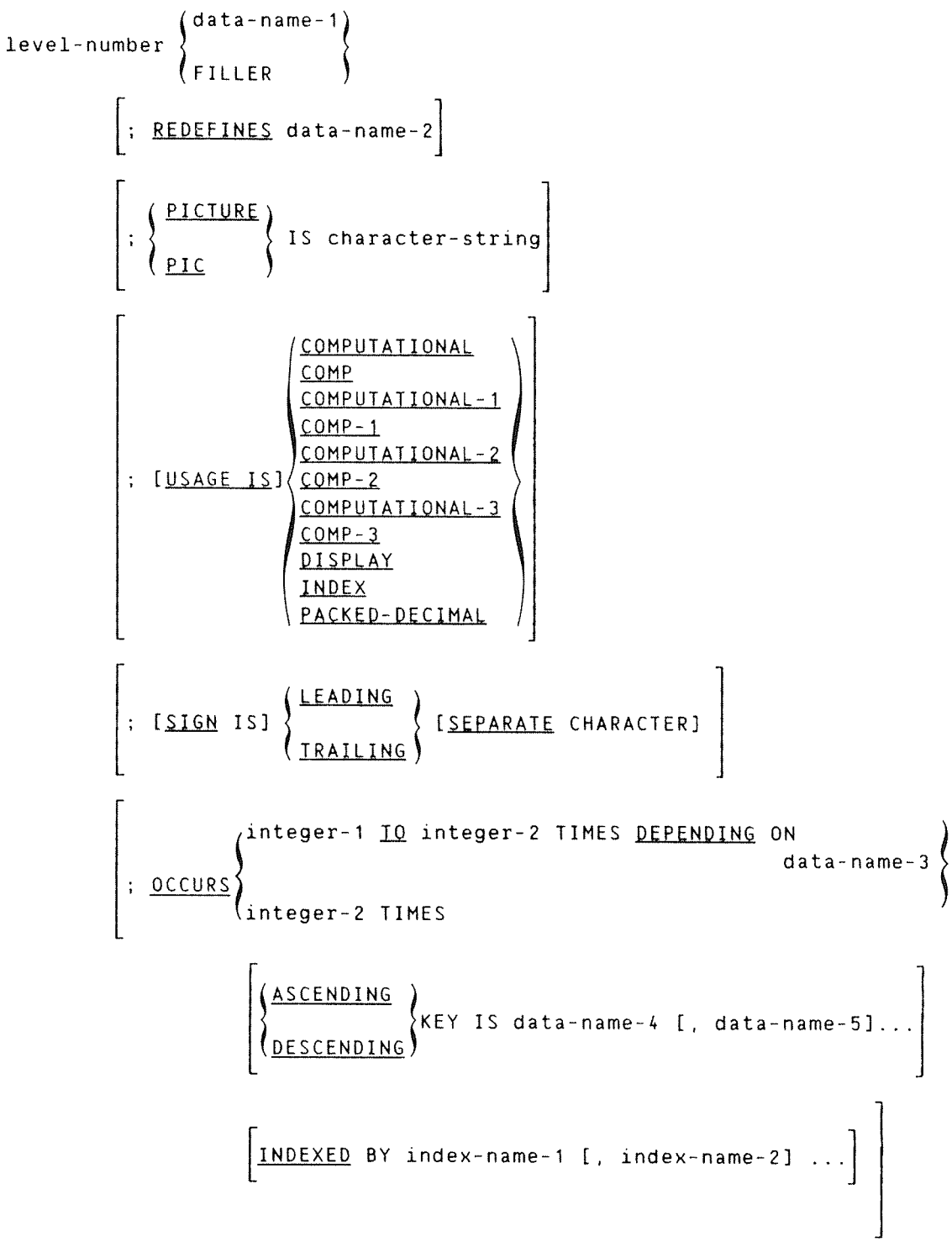
GENERAL FORMAT FOR DATA DIVISION

```
[WORKING-STORAGE SECTION.

 [ 77-level-description-entry ]
 [                            ] ...]
 [ record-description-entry   ]

[LINKAGE SECTION.

 [ 77-level-description-entry ]
 [                            ] ...]
 [ record-description-entry   ]
```

Data Division

GENERAL FORMAT FOR DATA DESCRIPTION ENTRY

FORMAT 1



GENERAL FORMAT FOR DATA DESCRIPTION ENTRY

```
[ ; { SYNCHRONIZED } [ LEFT  ] ]
    { SYNC         } [ RIGHT ]

[ ; ( JUSTIFIED ) RIGHT ]
    ( JUST      )

[ ; BLANK WHEN ZERO ]

[ ; VALUE IS literal ]

[ ; IMPORT [COMMON] ]

[ ; EXPORT ] .
```

FORMAT 2:

```
88 condition-name; ( VALUE IS  ) literal-1 [ ( THROUGH ) literal-2 ]
                   ( VALUES ARE )           [ ( THRU    )         ]

[ , literal-3 [ ( THROUGH ) literal-4 ] ] ... .
              [ ( THRU    )           ]
```

GENERAL FORMAT FOR PROCEDURE DIVISION

FORMAT 1:

```
PROCEDURE DIVISION [ USING data-name-1 [, data-name-2] ...] .

[DECLARATIVES.

{section-name SECTION [segment-number] .
     [USE sentence]
[paragraph-name. [sentence] ...] ...} ...

END DECLARATIVES.]

{section-name SECTION [segment-number] .
[paragraph-name. [sentence] ...] ...} ...
```

FORMAT 2:

```
PROCEDURE DIVISION [USING data-name-1 [, data-name-2] ...] .
{paragraph-name. [sentence] ...} ...
```

COBOL Verb Format

GENERAL FORMAT FOR VERBS

```
ACCEPT identifier [FROM  mnemonic-name]

ACCEPT identifier FROM { DATE     }
                       { DAY      }
                       { TIME     }
                       { CPU-TIME }

ACCEPT ( { identifier [{+} integer] } { identifier [{+} integer] } )
       ( {            integer       } {            integer       } )
         identifier [WITH [BEEP]
                          [SPACE-FILL]
                          [LENGTH-CHECK]
                          [AUTO-SKIP]
                          [PROMPT]
                          [BLANK-WHEN-ZERO]
                          [MUST]
                          [UPDATE]
                          [JUSTIFIED-RIGHT]
                          [INVISIBLE]
                          [INVERSE-VIDEO]
                          [BLINK]
                          [UNDERLINE]
                          [UPPER-CASE]
                          [LOW-INTENSITY]
                          [NORMAL]
                          [HELP Label]
                          [RE-DISPLAY Label]
                          [CANCEL Label]
                          [F1-F8 Label]
                          [UP Label]
                          [DOWN Label]
                          [HOME Label]
                          [EXIT Label]
                          [LEFT Label]
                          [RIGHT Label]
                          [CONTROL Label] ]
```

COBOL Verb Format

GENERAL FORMAT FOR VERBS

ACCEPT-ERROR

ACCEPT-RETURN

COBOL Verb Format

GENERAL FORMAT FOR VERBS

```
ADD { identifier-1 } [ , identifier-2 ] ... TO identifier-m [ROUNDED]
    { literal-1    } [ , literal-2    ]
      [ , identifier-n [ROUNDED] ] ...
      [ ; ON SIZE ERROR imperative-statement ]

ADD { identifier-1 } , { identifier-2 } [ , identifier-3 ] ...
    { literal-1    }   { literal-2    } [ , literal-3    ]
      GIVING identifier-m [ROUNDED] [ , identifier-n [ROUNDED] ] ...
      [ ; ON SIZE ERROR imperative-statement ]

ALTER procedure-name-1 TO [PROCEED TO] procedure-name-2
      [ , procedure-name-3 TO [PROCEED TO] procedure-name-4 ] ...

BLANK SCREEN

BLANK [ LINE  ] n1 [ TO n2 ] [ COLUMN n3 TO n4 ]
      [ LINES ]

CALL literal-1 [ USING { data-name-1 } [ , data-name-2 ] ... ]
               { quoted-literal }      [ , quoted-literal ]
               { integer        }      [ , integer        ]
```

COBOL Verb Format

GENERAL FORMAT FOR VERBS

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

CLOSE file-name-3 [ [ [REEL]  [WITH NO REWIND] ]
                  [ [UNIT]                     ]
                  [ WITH { NO REWIND } ]
                  [      { LOCK      } ]
                  [ , file-name-4 [ [ [REEL]  [WITH NO REWIND] ] ] ... ]
                  [               [ [UNIT]                     ] ]
                  [               [ WITH { NO REWIND } ]        ]
                  [               [      { LOCK      } ]        ]

COMPUTE identifier-1 [ROUNDED] [, identifier-2 [ROUNDED]] ...
        = arithmetic-expression
        [; ON SIZE ERROR imperative-statement]

CONTINUE

COPY file-name

DELETE file-name RECORD [; INVALID KEY imperative-statement]
```

COBOL Verb Format

GENERAL FORMAT FOR VERBS

```
DISPLAY { identifier-1 } [, identifier-2 ] ... [UPON mnemonic-name]
        { literal-1    } [, literal-2    ]

        [WITH NO ADVANCING]

DISPLAY( { identifier-1 [{+} integer-1] } { identifier-2 [{+} integer-3] } )
         { integer-2                    } { integer-4                    }

         { identifier-3 } [, identifier-4 ] ... [WITH [BEEP]
         { literal-1    } [, literal-2    ]       [SPACE-FILL]
                                                  [INVERSE-VIDEO]
                                                  [BLINK]
                                                  [UNDERLINE]
                                                  [LOW-INTENSITY]
                                                  [NORMAL]
                                                  [AUTO-ERASE]
                                                  [PROMPT]
                                                  [BLANK-WHEN-ZERO] ]

DISPLAY( { identifier-1 [{+} integer-1] } { identifier-2 [{+} integer-3] } )
         { integer-2                    } { integer-4                    }

       FRAME { identifier-3 } * { identifier-4 }
             { literal-1    }   { literal-2    }

             [WITH [SPACE-FILL] [HEADING] [REMARKS] [AUTO-ERASE]  ]

DISPLAY( { identifier-1 [{+} integer-1] } { identifier-2 [{+} integer-3] } )
         { integer-2                    } { integer-4                    }

       { FULL-BAR   } { identifier-3 } * { identifier-4 }
       { SPARSE-BAR } { literal-1    }   { literal-2    }
```

COBOL Verb Format

GENERAL FORMAT FOR VERBS

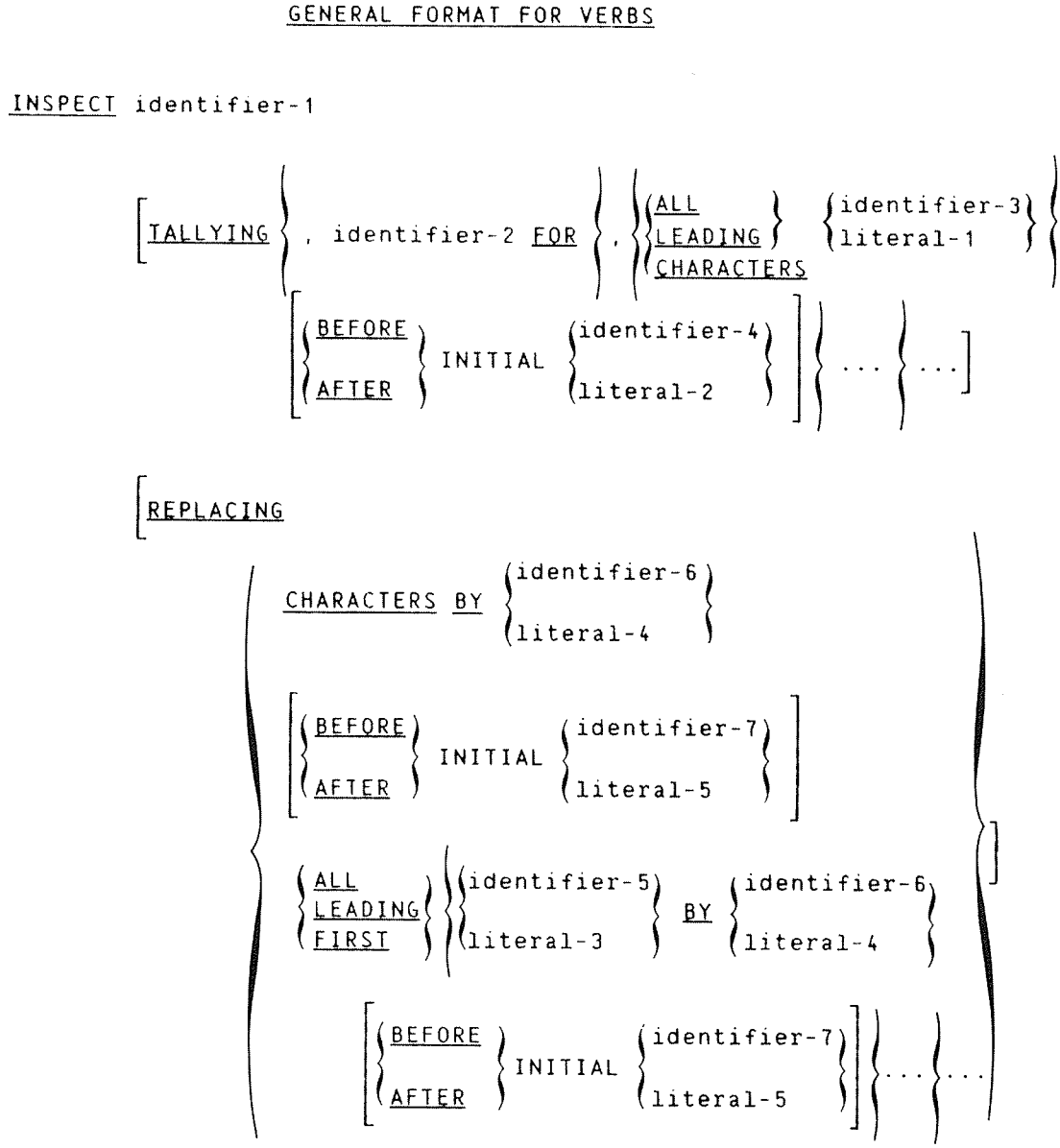
```
DIVIDE { identifier-1 } INTO identifier-2 [ROUNDED]
       { literal-1    }
        [, identifier-3 [ROUNDED] ]
        [; ON SIZE ERROR imperative-statement]

DIVIDE { identifier-1 } { INTO } { identifier-2 }
       { literal-1    } { BY   } { literal-2    }
        GIVING identifier-3 [ROUNDED] [, identifier-4 [ROUNDED] ] ...
        [; ON SIZE ERROR imperative-statement]

DIVIDE { identifier-1 } { INTO } { identifier-2 }
       { literal-1    } { BY   } { literal-2    }
        GIVING identifier-3 [ROUNDED]
        REMAINDER identifier-4
        [; ON SIZE ERROR imperative-statement]

DO sentence [WHILE condition sentence] ... END-DO

DO FOR identifier-1 FROM { identifier-2 [{+} integer-2] }
                         { integer-3                    }
   [ UP   ]
   [ DOWN ] [BY integer-4] TO { identifier-5 [{+} integer-5] }
                              { integer-6                    }
        sentence
        [WHILE condition sentence] ... END-DO
```

COBOL Verb Format

GENERAL FORMAT FOR VERBS

```
EXHIBIT NAMED  { identifier } ...
               { literal    }

EXIT [PROGRAM]

EXIT-DO

EXIT-ALL-DO

GO TO [procedure-name-1]

GO TO procedure-name-1 [, procedure-name-2] ... [, procedure-name-n]
      DEPENDING ON identifier

IF condition { statement-1 }  [ELSE { statement-2 }]
             { NEXT SENTENCE }      { NEXT SENTENCE }

IF condition THEN { statement-3 }  [ELSE { statement-2 }] [END-IF]
                  { NEXT SENTENCE }      { NEXT SENTENCE }

IF condition THEN { statement-5 }
                  { NEXT SENTENCE }

    [ELSE-IF condition-2  THEN { statement-6 }] ...
                               { NEXT SENTENCE }

    [ELSE { statement-7 }] [END-IF]
          { NEXT SENTENCE }
```

GENERAL FORMAT FOR VERBS



COBOL Verb Format

GENERAL FORMAT FOR VERBS

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

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

```
MOVE { CORRESPONDING } identifier-1 TO identifier-2
     { CORR          }
```

COBOL Verb Format

GENERAL FORMAT FOR VERBS

```
MULTIPLY { identifier-1 } BY identifier-2 [ROUNDED]
         { literal-1    }
         [, identifier-3 [ROUNDED] ] ...
         [; ON SIZE ERROR imperative-statement]

MULTIPLY { identifier-1 } BY { identifier-2 }
         { literal-1    }   { literal-2    }
         GIVING identifier-3 [ROUNDED]
         [, identifier-4  [ROUNDED] ]...
         [; ON SIZE ERROR imperative-statement]
```

COBOL Verb Format

GENERAL FORMAT FOR VERBS

```
OPEN { INPUT file-name-1 [WITH NO REWIND]
                [, file-name-2 [WITH NO REWIND]] ...

     { OUTPUT file-name-3 [WITH NO REWIND]
                [, file-name-4 [WITH NO REWIND]] ...

     { I-O file-name-5 [, file-name-6] ...

     { EXTEND file-name-7 [, file-name-6] ...
```

```
OPEN { INPUT  } file-name [WITH { MULTI-USER-MODE  }]
     { OUTPUT }                { IMMEDIATE-WRITE }
     { I-O    }                { MANUAL-UNLOCK   }

    [, [INPUT ] file-name [WITH { MULTI-USER-MODE  }]] ...
       [OUTPUT]               { IMMEDIATE-WRITE }
       [I-O   ]               { MANUAL-UNLOCK   }
```

COBOL Verb Format

GENERAL FORMAT FOR VERBS

```
PERFORM range

PERFORM range { identifier-1 } TIMES
              { integer      }

PERFORM range UNTIL condition-1

PERFORM range VARYING { identifier-5 } FROM { identifier-6 }
                      { index-name-3 }      { index-name-4 }
                                            { literal-3    }

                   BY { identifier-7 } UNTIL condition-1
                      { literal-4    }

          [ AFTER { identifier-8 } FROM { identifier-9 }
                  { index-name-5 }      { index-name-6 }
                                        { literal-5    }

                   BY { identifier-10 } UNTIL condition-2
                      { literal-6     }

          [ AFTER { identifier-11 } FROM { identifier-12 }
                  { index-name-8  }      { index-name-8  }
                                         { literal-6     }

                   BY { identifier-13 } UNTIL condition-3 ] ]
                      { literal-7     }
```

where range is the construct:

```
procedure-name-1 [ { THROUGH } procedure-name-2 ]
                   { THRU    }
```

COBOL Verb Format

GENERAL FORMAT FOR VERBS

```
READ file-name [ [ [ NEXT ] ] RECORD ] [INTO identifier] [WITH LOCK]
                [ PREVIOUS ]
     [; AT END imperative statement]

READ file-name RECORD [INTO identifier] [WITH LOCK]
     [; KEY IS data-name] [; INVALID KEY imperative-statement]

READ file-name RECORD [INTO identifier] [WITH LOCK]
     [; INVALID KEY imperative-statement]

RELEASE record-name [FROM identifier]
```

COBOL Verb Format

GENERAL FORMAT FOR VERBS

RESET SCREEN

```
RETURN file-name RECORD [INTO identifier]
        [; AT END imperative statement]

REWRITE record-name [FROM identifier]

REWRITE record-name [FROM identifier]
        [; INVALID KEY imperative-statement]
```

COBOL Verb Format

GENERAL FORMAT FOR VERBS

```
SEARCH identifier-1 [ VARYING { identifier-2   } ]
                              { index-name-1   }

     [ ; AT END imperative statement-1 ]

     ; WHEN condition-1 { imperative-statement-2 }
                        { NEXT SENTENCE          }

     [ ; WHEN condition-2 { imperative-statement-3 } ] ...
                          { NEXT SENTENCE          }
```

```
SEARCH ALL identifier-1 [ ; AT END imperative statement-1 ]

     ; WHEN { data-name-1      { IS EQUAL TO } { identifier-3           } } }
            {                  {             } { literal-1              } }
            {                  { IS =        } { arithmetic-expression  } }
            { condition-name-1                                            }

     [ AND { data-name-2      { IS EQUAL TO } { identifier-4           } } } ] ...
           {                  {             } { literal-2              } }
           {                  { IS =        } { arithmetic-expression  } }
           { condition-name-2                                            }

     { imperative-statement-2 }
     { NEXT SENTENCE          }
```

```
SET { identifier-1 [, identifier-2] ... } TO { identifier-3 }
    { index-name-1 [, index-name-2] ...  }    { index-name-3 }
                                              { integer-1    }
```

```
SET index-name-4 [, index-name-5] ... { UP BY   } { identifier-4 }
                                      { DOWN BY } { integer-2    }
```

COBOL Verb Format

GENERAL FORMAT FOR VERBS

```
SORT file-name-1 ON { ASCENDING  } KEY data-name-1 [, data-name-2] ...
                    { DESCENDING }
    [ ON { ASCENDING  } KEY data-name-3 [, data-name-4] ... ] ...
         { DESCENDING }

    { INPUT PROCEDURE IS section-name-1 [ { THROUGH } section-name-2 ] }
    { USING file-name-2                   { THRU    }                 }

    { OUTPUT PROCEDURE IS section-name-3 [ { THROUGH } section-name-4 ] }
    { GIVING file-name-2                   { THRU    }                  }

START file-name [ KEY { IS EQUAL TO                } data-name ]
                      { IS =                       }
                      { IS GREATER THAN            }
                      { IS >                       }
                      { IS NOT LESS THAN           }
                      { IS NOT <                   }
                      { IS LESS THAN               }
                      { IS <                       }
                      { IS NOT GREATER THAN        }
                      { IS NOT >                   }

    [; INVALID KEY imperative-statement]

STOP { RUN     }
     { literal }
```

COBOL Verb Format

GENERAL FORMAT FOR VERBS

```
STRING { identifier-1 } [ , identifier-2 ]
       { literal-1    } [ , literal-2   ] ... DELIMITED BY { identifier-3 }
                                                           { literal-3    }
                                                           { SIZE         }

     [ , { identifier-4 } [ , identifier-5 ]
         { literal-4    } [ , literal-5   ] ...

                 DELIMITED BY { identifier-6 } ]
                              { literal-6    }
                              { SIZE         } ...

       INTO identifier-7 [WITH POINTER identifier-8]

       [; ON OVERFLOW imperative-statement]
```

```
SUBTRACT { identifier-1 } [ , identifier-2 ]
         { literal-1    } [ , literal-2   ] ...

         FROM identifier-m [ROUNDED] [ , identifier-n [ROUNDED] ]

         [; ON SIZE ERROR imperative-statement]
```

```
SUBTRACT { identifier-1 } [ , identifier-2 ]
         { literal-1    } [ , literal-2   ] ... FROM { identifier-m }
                                                    { literal-m    }

         GIVING identifier-n [ROUNDED] [ , identifier-o [ROUNDED] ] ...

         [; ON SIZE ERROR imperative-statement]
```

COBOL Verb Format

GENERAL FORMAT FOR VERBS

```
UNLOCK file-name

UNSTRING identifier-1  [ DELIMITED BY [ALL] { identifier-2 }
                                            { literal-1    }
                       [ , OR [ALL] { identifier-3 } ] ... ]
                                    { literal-2    }
         INTO identifier-4 [ , DELIMITER IN identifier-5 ]
                           [ , COUNT IN identifier-6 ]
                           [ , identifier-7 [ , DELIMITER IN identifier-8 ]
                                            [ , COUNT IN identifier-9 ] ]
         [WITH POINTER identifier-10]
         [TALLYING IN identifier-11]
         [ ; ON OVERFLOW imperative-statement ]

USE AFTER STANDARD { EXCEPTION } PROCEDURE ON { file-name-1                  }
                   { ERROR     }              { [, filename-2] ...          }
                                              { INPUT                       }
                                              { OUTPUT                      }
                                              { I-O                         }
                                              { EXTEND                      }
```

COBOL Verb Format

GENERAL FORMAT FOR VERBS

```
WRITE record-name FROM identifier-1
      [ { BEFORE } ADVANCING { { { identifier-3 } [ LINE ] } }
      [ { AFTER  }           { { { integer      } [ LINES ] } } ]
                             {   PAGE                       }

WRITE record-name [FROM identifier-1] [WITH LOCK]

      [; INVALID KEY imperative-statement]
```

Condition Formats

GENERAL FORMAT FOR CONDITIONS

RELATION CONDITION:

```
{ identifier-1                  }  { IS [NOT] GREATER THAN }  { identifier-2                  }
{ literal-1                     }  { IS [NOT] LESS THAN    }  { literal-2                     }
{ arithmetic-expression-1       }  { IS [NOT] EQUAL TO     }  { arithmetic-expression-2       }
{ index-name-1                  }  { IS [NOT] >            }  { index-name-2                  }
                                   { IS [NOT] <            }
                                   { IS [NOT] =            }
```

CLASS CONDITION:

```
identifier IS [NOT] { NUMERIC      }
                    { ALPHABETIC   }
```

SIGN CONDITION:

```
arithmetic-expression IS [NOT] { POSITIVE }
                               { NEGATIVE }
                               { ZERO     }
```

CONDITION-NAME CONDITION:

```
condition-name
```

NEGATED SIMPLE CONDITION:

```
NOT simple condition
```

COMBINED CONDITION:

```
condition { { AND } condition } ...
          { { OR  }           }
```

ABBREVIATED COMBINED RELATION CONDITION:

```
relation-condition { { AND } [NOT] [relational-operator]  object } ...
                   { { OR  }                                      }
```

MISCELLANEOUS FORMATS

QUALIFICATION:

```
{data-name-1     }  [ {OF} data-name-2 ] ...
{condition-name  }    {IN}
```

```
paragraph-name  [ {OF} section-name ]
                  {IN}
```

```
file-name
```

SUBSCRIPTING:

```
{data-name     }  ( subscript-1 [, subscript-2 [, subscript-3]] )
{condition-name}
```

INDEXING:

```
{data-name     }  ( {index-name-1  [{+} literal-2]}
{condition-name}    {literal-1                    }

      [ , {index-name-2  [{+} literal-4]}
          {literal-3                   }

            [ , {index-name-3  [{+} literal-6]} ] ] )
                {literal-5                   }
```

MISCELLANEOUS FORMATS

IDENTIFIER: FORMAT 1

```
data-name-1 [ { OF } data-name-2 ] ...
            [ { IN }             ]

                [ ( subscript-1 [, subscript-2 [, subscript-3] ] ) ]
```

IDENTIFIER: FORMAT 2

```
data-name-1 [ { OF } data-name-2 ] ...
            [ { IN }             ]

                [ ( { index-name-1 [ {+} literal-2 ] }
                    { literal-1                      }

                    [ , { index-name-2 [ {+} literal-4 ] } ]
                        { literal-3                       }

                    [ , { index-name-3 [ {+} literal-6 ] } ] ] ) ]
                        { literal-5                       }
```

A P P E N D I X 2

ASCII CHARACTER SET

CHAR	Byte Position Left	Right	Dec	CHAR	Byte Position Left	Right	Dec
NUL	000000	000000	0	0	030000	000060	48
SOH	000400	000001	1	1	030400	000061	49
STX	001000	000002	2	2	031000	000062	50
ETX	001400	000003	3	3	031400	000063	51
EOT	002000	000004	4	4	032000	000064	52
ENQ	002400	000005	5	5	032400	000065	53
ACK	003000	000006	6	6	033000	000066	54
BEL	003400	000007	7	7	033400	000067	55
BS	004000	000010	8	8	034000	000070	56
HT	004400	000011	9	9	034400	000071	57
LF	005000	000012	10	:	035000	000072	58
VT	005400	000013	11	;	035400	000073	59
FF	006000	000014	12	<	036000	000074	60
CR	006400	000015	13	=	036400	000075	61
SO	007000	000016	14	>	037000	000076	62
SI	007400	000017	15	?	037400	000077	63
DLE	010000	000020	16	@	040000	000100	64
DC1	010400	000021	17	A	040400	000101	65
DC2	011000	000022	18	B	041000	000102	66
DC3	011400	000023	19	C	041400	000103	67
DC4	012000	000024	20	D	042000	000104	68
NAK	012400	000025	21	E	042400	000105	69
SYN	013000	000026	22	F	043000	000106	70
ETB	013400	000027	23	G	043400	000107	71
CAN	014000	000030	24	H	044000	000110	72
EM	014400	000031	25	I	044400	000111	73
SUB	015000	000032	26	J	045000	000112	74
ESC	015400	000033	27	K	045400	000113	75
FS	016000	000034	28	L	046000	000114	76
GS	016400	000035	29	M	046400	000115	77
RS	017000	000036	30	N	047000	000116	78
US	017400	000037	31	O	047400	000117	79
SPACE	020000	000040	32	P	050000	000120	80
!	020400	000041	33	Q	050400	000121	81
"	021000	000042	34	R	051000	000122	82
#	021400	000043	35	S	051400	000123	83
$	022000	000044	36	T	052000	000124	84
%	022400	000045	37	U	052400	000125	85
&	023000	000046	38	V	053000	000126	86
'	023400	000047	39	W	053400	000127	87
(	024000	000050	40	X	054000	000130	88
)	024400	000051	41	Y	054400	000131	89
*	025000	000052	42	Z	055000	000132	90
+	025400	000053	43	[	055400	000133	91
,	026000	000054	44	\	056000	000134	92
-	026400	000055	45	]	056400	000135	93
.	027000	000056	46	^	057000	000136	94
/	027400	000057	47				

CHAR	Byte Position Left	Right	Dec	CHAR	Byte Position Left	Right	Dec
_	057400	000137	95	o	067400	000157	111
`	060000	000140	96	p	070000	000160	112
a	060400	000141	97	q	070400	000161	113
b	061000	000142	98	r	071000	000162	114
c	061400	000143	99	s	071400	000163	115
d	062000	000144	100	t	072000	000164	116
e	062400	000145	101	u	072400	000165	117
f	063000	000146	102	v	073000	000166	118
g	063400	000147	103	w	073400	000167	119
h	064000	000150	104	x	074000	000170	120
i	064400	000151	105	y	074400	000171	121
j	065000	000152	106	z	075000	000172	122
k	065400	000153	107		075400	000173	123
l	066000	000154	108		076000	000174	124
m	066400	000155	109		076400	000175	125
n	067000	000156	110		077000	000176	126
				DEL	077400	000177	127

A P P E N D I X 3

RUNTIME MESSAGES

	Reason	*Explanation*
600B	I-O - ERR nnn where nnn is a standard SINTRAN III File System Error Code	An error in an I-O operation has arisen without the possibility of user reaction due to omission of any of the following applicable to the file in question: - AT END clause - INVALID KEY clause - USE AFTER STANDARD ERROR (Declarative) If any relevant element above is available, the runtime library routines allow that element to process the data, and THIS ERROR DOES NOT ARISE.
601B	INDEX FILE ERROR nn where nn is a status returned from ISAM	An error has arisen in using the Indexed or relative file system, without the possibility of user reaction due to omission of any of the following clauses or to other index file errors: - AT END clause - INVALID KEY clause - USE AFTER STANDARD ERROR (Declarative) If any relevant element above is available, the runtime library routines allow that element to process the data, and THIS ERROR DOES NOT ARISE.
602B	FILE NOT OPEN	Non-addressable data due to an attempt to use data in a file that is not open.
603B	FILE NOT OPEN IN CORRECT MODE	Attempt to use data in a file whioch is not open in the correct mode.
604B	IMPROPER RECORD LENGTH	Incoming record size incorrect when using the REWRITE statement.
605B	ILLEGAL USE OF REWRITE	Sequence error when using REWRITE. Previous I-O statement not a READ.

	Reason	*Explanation*
606B	OPEN MODE I-O CANNOT BE USED FOR MAGNETIC TAPE	Magnetic tape files cannot be opened for I-O.
607B	SORT ERROR	Issued by the SORT system.
610B	SORT ERROR - FILE TOO BIG	Issued by the SORT system.
611B	SORT ERROR - TOTAL KEY TOO LONG	Issued by the SORT system.
612B	SORT ERROR IN RECORD SIZE	Issued by the SORT system.
613B	COMPILER/LIBRARY INCOMPATIBILITY	Different versions of the compiler and runtime library cannot be used simultaneously.

APPENDIX 4

RESERVED WORD LIST

ACCEPT
ACCEPT-ERROR
ACCEPT-RETURN
ACCESS
ADD
ADVANCING
AFTER
ALL
ALPHABETIC
ALTER
ALTERNATE
AND
ARE
AREA
AREAS
ASCENDING
ASSIGN
AT
AUTHOR
AUTO-ERASE
AUTO-SKIP
BEEP
BEFORE
BLANK
BLANK-WHEN-ZERO
BLINK
BLOCK
BOX
BY
CALL
CANCEL
CHARACTER
CHARACTERS
CLOSE
COLUMN
COMMA
COMMON
COMP
COMP-1
COMP-2
COMP-3
COMPUTATIONAL
COMPUTATIONAL-1
COMPUTATIONAL-2
COMPUTATIONAL-3
COMPUTE
CONFIGURATION
CONTAINS
CONTINUE
CONTROL
COPY
CORR
CORRESPONDING
COUNT
CPU-TIME
CURRENCY
DATA
DATE
DATE-COMPILED
DATE-WRITTEN
DAY
DEBUGGING
DECIMAL-POINT
DECLARATIVES
DELETE
DELIMITED
DELIMITER
DEPENDING
DESCENDING
DISPLAY
DIVIDE
DIVISION
DO
DOWN
DUPLICATES
DYNAMIC
ELSE
ELSE-IF
END
END-DO
END-IF
ENVIRONMENT
EQUAL
ERASE
ERROR
EXCEPTION
EXHIBIT
EXIT
EXIT-ALL-DO
EXIT-DO
EXPORT
EXTEND
F1-F8
FD
FILE
FILE-CONTROL
FILE-ID
FILLER
FIRST
FOR
FRAME
FROM
FULL-BAR
GIVING
GO
GOBACK
GREATER
HEADING
HELP
HIGH-VALUE
HIGH-VALUES
HOME
I-O
I-O-CONTROL
ID
IDENTIFICATION
IF
IMMEDIATE-WRITE
IMPORT
IN
INDEX
INDEXED
INITIAL
INPUT
INPUT-OUTPUT
INSPECT
INSTALLATION
INTO
INVALID
INVERSE-VIDEO
INVISIBLE
IS
JUST
JUSTIFIED
JUSTIFIED-RIGHT
KEY
LABEL
LEADING
LEFT
LENGTH-CHECK
LESS
LINE
LINES
LINKAGE
LISTEN
LOCK
LOW-INTENSITY
LOW-VALUE
LOW-VALUES
MANUAL-UNLOCK
MERGE
MODE
MONITOR-CALL
MOVE
MULTI-USER-MODE
MULTIPLY
MUST
NAMED
NEGATIVE
NEXT
NO
NORMAL

NOT
NUMERIC
OBJECT-COMPUTER
OCCURS
OF
OFF
OMITTED
ON
OPEN
OPTIONAL
OR
ORGANIZATION
OUTPUT
OVERFLOW
PACKED-DECIMAL
PAGE
PERFORM
PIC
PICTURE
POINTER
POSITIVE
PREVIOUS
PROCEDURE
PROCEED
PROGRAM
PROGRAM-ID
PROMPT
QUOTE
QUOTES
RANDOM
RE-DISPLAY
READ
RECORD
RECORDING
RECORDS
REDEFINES
REEL
RELATIVE
RELEASE
REMAINDER
REMARKS
REMOVAL
RENAMES
REPLACING
REPORT
RESERVE
RESET
RETURN
REWIND
REWRITE
RIGHT
ROUNDED
RUN
SAME
SCREEN
SD
SEARCH
SECTION
SECURITY
SELECT
SENTENCE
SEPARATE
SEQUENTIAL
SET
SIGN
SIZE
SORT
SOURCE-COMPUTER
SPACE
SPACE-FILL
SPACES
SPARSE-BAR
SPECIAL-NAMES
STANDARD
START
STATUS
STOP
STRING
SUBTRACT
SYNC
SYNC-2
SYNCHRONIZED
SYNCHRONIZED-2
TALLYING
TEXT-FILE
THAN
THEN
THROUGH
THRU
TIME
TIMEOUT
TIMES
TO
TRAILING
UNDERLINE
UNIT
UNLOCK
UNSTRING
UNTIL
UP
UPDATE
UPON
UPPER-CASE
USAGE
USE
USER-DEFINED-SIZE
USING
VALUE
VALUES
VARYING
WHEN
WHILE
WITH
WORKING-STORAGE
WRITE
ZERO
ZEROES
ZEROS

A P P E N D I X 5

CROSS REFERENCE EXAMPLE

To obtain a cross reference listing with a compilation the command

XREF file-name

must be issued at compile-time where 'file-name' is the name of a work file.

```
ND-100 COBOL COMPILER                    TIME: 13.08.32     DATE: 85.02.19
  SOURCE FILE: X
  OBJECT FILE: X
  MODES:       2-BANK

     1        IDENTIFICATION DIVISION.
     2        PROGRAM-ID.
     3            CROSS-REFERENCE-EXAMPLE.
     4        DATA DIVISION.
     5        WORKING-STORAGE SECTION.
     6        01  PERSON.
     7            03 NAME              PIC X(30)       VALUE "NORSK DATA A/S".
     8            03 ADDRESS           PIC X(30)       VALUE "OSLO, NORWAY".
     9            03 TELEPHONE         PIC 9(11)       VALUE 023090330.
    10            03 INCOME            PIC S9(9)V99    COMP-3.
    11            03 COUNTRY           PIC S9(2)       COMP VALUE 1.
    12                                 88  NORWAY      VALUE 1.
    13                                 88  SWEDEN      VALUE 2.
    14                                 88  DENMARK     VALUE 3.
    15                                 88  ENGLAND     VALUE 4.
    16                                 88  GERMANY     VALUE 5.
    17                                 88  SWITZERLAND VALUE 6.
    18
    19        PROCEDURE DIVISION.
    20        TEST SECTION.
    21        0000.
    22            IF NORWAY           PERFORM 1000.
    23            IF SWEDEN           PERFORM 2000.
    24            IF DENMARK          PERFORM 3000.
    25            IF ENGLAND          PERFORM 4000.
    26            IF GERMANY          PERFORM 5000.
    27            IF SWITZERLAND      PERFORM 6000.
    28            STOP RUN.
    29        1000.
    30            DISPLAY NAME "IS NORWEGIAN".
    31        2000.
    32            DISPLAY NAME "IS SWEDISH".
    33        3000.
    34            DISPLAY NAME "IS DANISH".
    35        4000.
    36            DISPLAY NAME "IS ENGLISH".
    37        5000.
    38            DISPLAY NAME "IS GERMAN".
    39        6000.
    40            DISPLAY NAME "IS SWISS".
```

```
N O R D   C O B O L   C R O S S   R E F E R E N C E   L I S T

PROGRAM-ID: CROSS-REFERENCE-EXAMPLE

0000 . . . .(PARAGRAPH)      21
1000 . . . .(PARAGRAPH)      22      29
2000 . . . .(PARAGRAPH)      23      31
3000 . . . .(PARAGRAPH)      24      33
4000 . . . .(PARAGRAPH)      25      35
5000 . . . .(PARAGRAPH)      26      37
6000 . . . .(PARAGRAPH)      27      39
ADDRESS. . .(X       30)      8
COUNTRY. . .(COMP     2)     11
DENMARK. . .(88       2)     14      24
ENGLAND. . .(88       2)     15      25
GERMANY. . .(88       2)     16      26
INCOME . . .(COMP-3   6)     10
NAME . . . .(X       30)      7      30      32      34      36      38      40
NORWAY . . .(88       2)     12      22
PERSON . . .(X       80)      6
SWEDEN . . .(88       2)     13      23
SWITZERLAND.(88       2)     17      27
TELEPHONE. .(NUM     11)      9
TEST . . . .(SECTION  )     20
```

APPENDIX 6

COMPILER COMMANDS

COMPILER COMMANDS: ND-100

HELP
Lists available commands.

EXIT
Exit to SINTRAN III.

COMPILE
<source-file><list-file><object-file>

Defines I/O files for the COBOL compiler.

XREF-LIST
<work-file>

A cross reference list will be output onto the list file, not the work-file. The parameter <work-file> provides a working file for XREF. Default file type is :XREF. Example: See Appendix 5.

DEBUG-MODE
Debug information will be generated and the Symbolic Debugger can be used. See Symbolic Debugger User Guide, ND-60.158.

LIBRARY-MODE
The object file will be a library file.

ND100-EXTENDED-MODE
Turns ON the use of the commercial instruction set (COM) in the compiler.

If the computer has a commercial instruction set, this command increases the speed of execution.

Note: on ND-10 the commercial instruction set must be installed in order to run COBOL programs.

1-BANK-MODE
If this command is not present, the default is 2-bank mode. Normally the code and data are separated, but the use of this command ensures that they are together. The runtime library COBOL-1BANK must be loaded.

TPS-MODE
The compilation will take place under the TPS system.

LOAD

file-name [,file-name]...

To complete the executable program, libraries or other object files may be added by using the above command, where file name is the name of an object file or library.

The default type of the file loaded will be :BRF on the ND-100.

LOAD commands will be ignored if they are placed in the source file, or if no PROG-FILE command has been given.

Any error messages which appear while the LOAD command is being executed can be found in the ND Relocating Loader manual (ND-60.066).

COMPILER COMMANDS: ND-500

HELP
Lists available commands.

EXIT
Exit to SINTRAN III.

COMPILE
<source-file><list-file><object-file>

Defines I/O files for the COBOL compiler. The default type for the source-file is :SYMB or :COB. For list-file it is :SYMB, and for the object-file it is :NRF.

XREF-LIST
<work-file>

A cross reference list will be output to the list file, not on the work-file. The parameter <work-file> provides a working file for XREF. Default file type is :XREF.

DEBUG-MODE
Debug information will be generated and the Symbolic Debugger can be used. See Symbolic Debugger User Guide, ND-60.158.

LIBRARY-MODE
The object file will be a library file.

APPENDIX 7

INDEXED/RELATIVE I-O STATUS SUMMARY

The following table summarizes what values the I-O status may have after one of the Indexed/Relative I-O verbs has been used:

STATUS		VERB							
Code Pic-ture XX	Meaning	OPEN	CLOSE	READ NEXT	READ	WRITE	REWRITE	DELETE	START
"00"	OK	x	x	x	x	x	x	x	x
"10"	End of file			x					
"21"	Wrong sequence of words			x			x	x	
"22"	Duplicates not allowed					x	x		
"23"	Record not found				x			x	x
"24"	No more space on file					x			
"68"	Record locked by another program			x1	x1		x	x	
"78"	Record modified by another program						x	x	
"90"	Multiuser Supervisor not started	x							
"93"	Too many keys	x2							
"94"	Error flag set	x							
"95"	File not initialized or opened	x	x						
"97"	File access violation	x		x	x	x	x	x	x
"98"	Wrong file description	x	x3						
"99"	SINTRAN III file system error	x	x						

x = Verbs where this I-O status may occur
x1 = May occur only on read with lock
x2 = Check the COBOL system variable CB70
x3 = File already closed

A P P E N D I X 8

COBOL SYSTEM VARIABLES

ND COBOL will allow application programs to access a set of COBOL system variables for inspection and/or modification. These variables must be IMPORTed in the LINKAGE SECTION.

Name	Contents	Modification allowed
CB50	Field Termination Character (See ACCEPT statement)	No
CB60	Number of significant characters in an entered or edited ACCEPT field.	No
CB70	Maximum number of keys in an indexed sequential file (initial value = 6)	Yes

The system variable *CB50* is of special interest when making programs with screen handling statements, such as DISPLAY and ACCEPT. It holds information on which of the keys <CR>, ↓, ↑, ←, →, \, HELP, CANCEL etc. the user pressed when in a field. CB50 must be imported as a COMP-1 variable in the LINKAGE SECTION of the application program.

The following table shows possible values of CB50:

Keys pressed	*CB50 value*
F1 *F1 shifted* . . . . *F8* *F8 shifted*	*1* *9* *8* *16*
HOME ( ) *RIGHT* ( ) *LEFT* ( ) *UP* ( ) *DOWN* ( )	*41* *42* *43* *44* *45*
CR ( ) *AUTO-SKIP*	*51* *52*
EXIT <*Control*> *L* *CANCEL* *HELP* *TIMEOUT*	*61* *61* *62* *63* *64*

Example:

```
       LINKAGE SECTION.

       01  CB50     COMP IMPORT.
       01  CB60     COMP IMPORT.
       01  CB70     COMP IMPORT.

       PROCEDURE DIVISION.

      * Allow up to 12 keys in an indexed sequential file
      * (this must be done before the first open of an
      * indexed sequential file):

                .
                .
                .
           MOVE 12 TO CB70.
                .
                .
                .
```

APPENDIX 9

HANDLING SINTRAN ERRORS

The system subroutine "CBERMSG" and "ISERR" make SINTRAN error numbers and texts available. Thus, SINTRAN error detection facilities can guide the program flow, without having them cause the program to abort or disturb the secreen picture.

The DECLARATIVES part of the procedure division is the appropriate place for use of these subroutines. Consider this example:

```
IDENTIFICATION DIVISION.
          .
          .

WORKING-STORAGE SECTION.
          .
          .

01 ERROR-NUMBER          COMP.
          .
          .

PROCEDURE DIVISION.

DECLARATIVES.

FILE-ERROR SECTION.
    USE AFTER ERROR-PROCEDURE ON ISAM-FILE.

ISAM-ERROR.
    IF ISAM-STATUS = "99" THEN
        CALL "ISERR" USING ERROR-NUMBER
        CALL "CBERMSG" USING ERROR-NUMBER
    END-IF.
    CALL "ERROR-ISAM" USING ERROR-NUMBER.
          .
          .

END DECLARATIVES.

MAIN SECTION.
          .
          .

    OPEN I-O ISAM-FILE.
          .
          .

    CLOSE ISAM-FILE.
          .
          .

    STOP RUN.
```

If anything goes wrong during opening and closing of the ISAM-FILE, the ISAM-ERROR paragraph is performed. If it is a SINTRAN-error, the ISAM-STATUS will become equal to "99". Then the subroutine ISERR (which is a standard library routine) with ERROR-NUMBER as a parameter will return with the SINTRAN error number in ERROR-NUMBER. The user may either leave it at that, or use the subroutine CBERMSG (which is also a standard library routine) to display the appropriate SINTRAN error message.

Note: If CBERMSG is used, it will disturb any screen picture developed before it was called.

Also note that these routines may be changed in future implementations of ND-COBOL.

APPENDIX 10

EXECUTING SINTRAN COMMANDS

The system subroutine "CBCOMND" executes ordinary SINTRAN commands. It is used as follows:

```
CALL "CBCOMND" USING "<SINTRAN command> ' ".
```

Here, note how the parameter is passed. Since the SINTRAN commands may involve the creation of new files, it is ***mandatory*** to end the SINTRAN command string with a single quote , ', before the finishing double-quote, ". For example:

```
CALL "CBCOMND" USING "LIST-FILES :BRF,"BRF:LIST"'".
```

Also note that this routine may be changed in future implementations of ND-COBOL.

APPENDIX 11

SIZE OF TEMPORARY FIELDS

Execution by the compiler of certain arithmetic statements or operations can generate intermediate results which will be stored in *temporary fields*.

Intermediate results can be obtained when:

1) A COMPUTE statement assigns the value of an arithmetic expression to more than one data item.

2) ADD or SUBTRACT statements are encountered which have multiple operands immediately following the verb.

3) IF or PERFORM statements containing arithmetic expressions are executed.

Using the COMPUTE statement as an example, the size of temporary fields can be ascertained as follows.

Each numeric item within the arithmetic expression is examined. A temporary field is formed which can contain the maximum number of digit positions before the decimal point in any examined item, linked together with the maximum number of digit positions of any examined item following a decimal point.

Example: If we have:

```
COMPUTE X = A * B
```

where A is declared as PIC S9(5)V99999
and B is declared as PIC S9(7)V9999

Then the temporary field will have a size of :

```
S9(7)V99999
```

If the total number of positions after the figures have been linked together is greater than 18, the number of digit positions will be truncated from the right.

COMPUTATIONAL DATA ITEMS

The size of a field for a COMPUTATIONAL item is a single word with four or less integer positions before the decimal point, and a double word with five or more such positions.

However, for the purposes of calculating the sizes of temporary fields, a COMPUTATIONAL item occupying a single word and *not* having a picture definition, is treated as if it had *five* places before the decimal point, and a double word item as if it had*ten*. COMPUTATIONAL items are integers and have no places after the decimal point.

A COMPUTATIONAL item with a picture definition will have a temporary field formed, containing the maximum number of digit positions, plus a sign position.

For example, an item declared as:

PIC S9(2) COMPUTATIONAL

with a value of 11, will have a temporary field of 3 positions.

A P P E N D I X 12

GLOSSARY

Abbreviated Combined Relation Condition
The combined condition that results from the explicit omission of a common subject or a common subject and a common relational operator in a consecutive sequence of relation conditions.

Access Mode
The manner in which records are to be operated upon within a file.

Actual Decimal point
The physical representation, using either of the decimal point characters period (.) or comma (,), of the decimal point position in a data item.

Alphabetic Character
A character that belongs to the following set of characters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z and the space.

Alphanumeric Character
Any character in the computer's character set.

Alternate Record Key
A key, other than the prime key, whose contents identify a record within an indexed file.

Arithmetic Expression
An arithmetic expression can be:

1) an identifier or a numeric elementary item

2) a numeric literal

3) such identifiers and literals separated by arithmetic operators

4) two arithmetic expressions separated by an arithmetic operator

5) an arithmetic operation enclosed in a parenthesis.;

Arithmetic Operator

A single character, or a fixed two character combination, that belongs to the following set:

Character	Meaning
+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

Ascending Key

A key upon whose values data is ordered, starting with the lowest value of key up to the highest value of key in accordance with the rules for comparing data items.

Assumed Decimal Point

A decimal point which does not involve the existence of an actual character in a data item. The assumed decimal point has logical meaning but no physical representation.

At End Condition

A condition caused:

1) During the execution of a READ statement for a sequentially accessed file.

2) During the execution of a RETURN statement, when no next logical record exists for the associated sort file or merge file.

3) During the execution of a SEARCH statement, when the search operation terminates without satisfying the condition specified in any of the associated WHEN phrases.

Block

A physical unit of data that is normally composed of one or more logical records. For mass storage files, a block may contain a portion of a logical record. The size of the block has no direct relationship to the size of the file within which the block has not direct relationship to the size of the file with which the block is contained or tp the size of the logical record(s) that are either continued within the block or that overlap the block. The term is synonymnous with physical record.

Called Program
A program which is the object of a CALL statement combined at object time with the calling program to produce a run unit.

Calling Program
A program which executes a CALL to another program.

Character
The basic, indivisible unit of the language.

Character Position
A character position is the amount of physical storage required to store a single standard data format described as USAGE IS DISPLAY.

Characterstring
A sequence of contiguous characters which form a COBOL word, a literal, a PICTURE characterstring, or a comment entry

Class Condition
The proposition, for which a truth value can be determined, that the content of an item is wholly alphabetic or is wholly numeric.

Clause
A clause is an ordered set of consecutive COBOL characterstrings whose purpose is to specify an attribute of an entry.

COBOL Word
See Word.

Collating Sequence
The sequence in which the characters that are acceptable to a computer are ordered for the purpose of sorting, merging or comparing.

Column
A character position within a print line. The columns are numbered from 1, by 1, starting at the leftmost character position of the print line and extending to the rightmost position of the print line.

Combined Condition
A condition that is the result of connecting two or more conditions with the 'AND' or the 'OR' logical operators.

Comment Line

A source program line represented by an asterisk in the indicator area of the line and any characters from the computer's character set in area A and area B of that line. The comment line serves only for documentation in a program. A special form of comment line represented by a stroke (/) in the indicator area of the line and any characters from the computer's character set in area A and B of that line causes page ejection prior to printing the comment.

Compile Time

The time at which a COBOL source program is translated, by a COBOL compiler, to a COBOL object program.

Compiler Directing Statement

A statement, beginning with a compiler directing verb, that causes the compiler to take a specific action during compilation.

Complex Condition

A condition in which two or more logical operators act upon one or more conditions. (See Negated Simple Condition, Combined Condition and Negated Combined Condition.)

Computername

A systemname that identifies the computer upon which the program is to be compiled or run.

Condition

A status of a program at execution time for which a truth value can be determined. Where the term 'condition' (condition-1, condition-2, ...) appears in these language specifications or in reference to 'condition' (condition-1, condition-2, ...) of a general format, we have a conditional expression condition. It consists of either a simple condition optionally parenthesized, or a combined condition consisting of the syntactically correct combination of simple conditions, logical operators, and parenteses, for which a truth value can be determined.

Condition Name

A user defined word assigned to a specific value, set of values, or range of values, within the complete set of values that a conditional variable may posess.

Conditionname Condition

The proposition, for which a truth value can be determined, that the value of a conditional variable is a member of the set of values attributed to a conditionname associated with the conditional variable.

Conditional Expression
A simple condition or a complex condition specified in an IF, PERFORM or SEARCH statement. (See Simple Condition and Complex Condition.)

Conditional Statement
A conditional statement specifies that the truth value of a condition is to be determined and that the subsequent action of the object program is dependent of this truth value.

Conditional Variable
A data item of which one or more values has a condition name assigned to it.

Configuration Section
A section of the Environment Division that describes overall specifications of source and object computers.

Connective
A reserved word that is used to:

1) Associate a dataname, paragraphname, conditionname or textname with its qualifier.

2) Link two or more operands within a series.

3) Form conditions (logical connectives)(see Logical Operator).

Contiguous Items
Items that are described by consecutive entries in the Data Division and bear a definite hierarchic relation to each other.

Currency Sign
The character '$' of the COBOL character set.

Currency Symbol
The character defined by the CURRENCY SIGN clause in the SPECIAL NAMES paragraph. If no CURRENCY SIGN clause is present in a COBOL source program, the currency symbol is identical to the currency sign.

Current Record
The record which is available in the record area associated with the file.

Current Record Pointer
A conceptual entity that is used in the selection of the next record.

Data Clause
A clause that appears in a data description entry in the Data Division and provides information describing a particular attribute of a data item.

Data Description Entry
An entry in the Data Division that is composed of a level number followed by a data name, if required, and then followed by a set of data clauses, as required.

Data Item
A character or a set of contiguous characters (excluding in either case literals) defined as a unit of data by the COBOL program.

Dataname
A user defined word that names a data item described in a data description entry in the Data Division. When used in general formats, 'dataname' represents a word which can neither be subscripted or indexed, nor qualified unless specifically permitted by the rules of the format.

Debugging Line
A debugging line is any line with 'D' in column 7. It is only compiled when the compiler has been directed to provide output for use with the Symbolic Debugger.

Declaratives
A set of one or more special purpose sections, written at the beginning of the Procedure Division, the first of which is preceded by the key word DECLARATIVES and the last of which is followed by the key words END DECLARATIVES. A declarative is composed of a section header, followed by a USE compiler directing sentence, followed by a set of zero, one or more associated paragraphs.

Declarative Sentence
A compiler directing sentence consisting of a single USE statement terminated by the separator period (.).

Delimiter

A character or a sequence of contiguous characters that identify the end of a string of characters and separates that string of characters from the following string of characters. A delimiter is not part of the string of characters that it delimits.

Descending Key

A key, upon whose values data is ordered, starting with the highest value of key down to the lowest value of key, according to the rules for comparing data items.

Digit Position

A digit position is the amount of physical storage required to store a single digit. This amount may vary, depending on the usage of the physical storage defined by the implementor.

Division

A set of zero, one or more sections of paragraphs, called the division body, that are formed and combined according to a specific set of rules. There are four (4) division in a COBOL program: Identification, Environment, Data and Procedure.

Division Header

A combination of words followed by a period and a space that indicates the beginning of a division. The division headers are:

IDENTIFICATION DIVISION
ENVIRONMENT DIVISION
DATA DIVISION
PROCEDURE DIVISION [USE sentence]

Dynamic Access

An access mode in which specific logic records can be obtained from or placed into a mass storage file in a nonsequential manner (see Random Access) and obtained from a file in a sequential manner (see Sequential Acces) during the scope of the same OPEN statement.

Editing Character

A single character or a fixed two character combination belonging to the following set:

Character:	Meaning:
B	space
0	zero
+	plus
-	minus
CR	credit
DB	debit
Z	zero suppress
*	check protect
$	currency sign
,	comma (decimal point)
.	period (decimal point)
/	stroke (virgule, slash)

Elementary Item

A data item that is not described as further logically subdivided.

End of Procedure Division

The physical position in a COBOL source program after which no further procedures appear.

Entry

Any descriptive set of consecutive clauses terminated by a period and written in the Identification Division, Environment Division or Data Division of a COBOL source program.

Environment Clause

A clause that appears as a part of an Environment Division entry.

Execution Time

See Object Time.

Extend Mode

The state of a sequential file after execution of an OPEN statement, with the EXTEND phrase specified for that file, and before the execution of a CLOSE statement for that file.

Figurative Constant

A compiler generated value referenced through the use of certain reserved words.

File
A collection of records.

FILE-CONTROL
The name of an Environment Division paragraph in which the data files for a given source program are declared.

File Description Entry
An entry in the File Section of the Data Division that is composed of the level indicator FD, followed by a file name, and then followed by a set of file clauses as required.

File Name
A user defined word that names a file described in a file description entry or a SORT/MERGE file description entry within the File Section of the Data Division.

File Organization
The permanent logical file structure established when a file is created.

File Section
The section of the Data Division that contains file description entries and SORT/MERGE file description entries together with their associated record descriptions.

Format
A specific arrangement of a set of data.

Group Item
A named contiguous set of elementary items or group items.

High Order End
The leftmost character of a string of characters.

I-O-Control
The name of an Environment Division paragraph in which object program requirements for specific input-output techniques, rerun points, sharing of the same areas of several data files and multiple file storage on a single input-output device are specified.

I-O Mode
The state of a file after execution of an OPEN statement, with the I-O phrase specified for that file, and before the execution of a CLOSE statement for that file.

Identifier
A data name, followed as required by the syntactically correct combinations of qualifiers, subscripts and indices necessary to make unique reference to a data item.

Imperative Statement
A statement that begins with an imperative verb and specifies an unconditional action to be taken. An imperative statement may consist of a sequence of imperative verbs.

Index
A computer storage position or register, the content of which represents the identification of a particular element in the table.

Index Data Item
A data item in which the value associated with an index name can be stored in a form specified by the implementor.

Index Name
A user defined word that names an index associated with a specific table.

Indexed Data Name
An identifier that is composed of a data name, followed by one or more index names enclosed in parentheses.

Indexed File
A file with indexed organization.

Indexed Organization
The permanent logical file structure in which each record is identified by a value of one or more keys within that record.

Input File
A file that is opened in the input mode.

Input Mode
The state of a file after execution of an OPEN statement, with the INPUT phrase specified for that file, and before the execution of a CLOSE statement for that file.

Input-Output File
A file that is opened in the I-O mode.

Input-Output Section
The section of the Environment Division that names the files and the external media required by an object program and which provides information required for transmission and handling of data during the execution of the object program.

Input Procedure
A set of statements that is executed each time a record is released to the sort file.

Integer
A numeric literal or a numeric data item that does not include any character positions to the right of the assumed decimal point. Where the term 'integer' appears in general formats, integer can not be a numeric data item, neither can it be signed, nor can it bezero unless this is explicitly allowed by the rules of that format.

Invalid Key Condition
A condition, at object time, caused when a specific value of the key associated with an indexed or relative file is determined to be invalid.

Key
A data item which identifies the location of a record, or a set of data items which serve to identify the ordering of data.

Key Word
A Reserved Word whose presence is required when the format in which the word appears is used in a source program.

Level Indicator
Two alphabetic characters that identify a specific type of file or a position in the hierarchy.

Level Number
A user defined word which either indicates the position of a data item in the hierarchical structure of a logical record or the special properties of a data description entry. A level number is expressed as a single- or double-digit number. Level numbers in the range 1 through 49 indicate the position of a data item in the hierarchical structure of a logical record. Level numbers in the range 1 to 9 may be written either as a single digit or as a zero followed by a significant digit. Level numbers 77 and 88 identify special properties of a data description entry.

Library Name
A user defined word that names a COBOL library that is to be used by the compiler for a given source program compilation.

Library Text
A sequence of character strings and/or separators in a COBOL library.

Linkage Section
The section in the Data Division of the called program that describes data items available from the calling program. These data items may be referred to by both the calling and called program.

Literal
A characterstring whose value is implied by the ordered set of characters comprising the string.

Logical Operator
One of the reserved words AND, OR or NOT. In the formation of a condition, either of one or both AND and OR can be used as logical connectives. NOT can be used for logical negation.

Logical Record
The most inclusive data item. The level number for a record is 01.

Low Order End
The rightmost character of a string of characters.

Mass Storage
A storage medium on which data may be organized and maintained in both a sequential and a nonsequential manner.

Mass Storage Control System (MSCS)
An input-output control system that directs, or controls, the processing of mass storage files.

Mass Storage File
A collection of records that is assigned to a mass storage medium.

Merge File
A collection of records to be merged by a MERGE statement. The merge file is created and can be used only by the merge function.

Mnemonic Name
A user defined word that is associated in the Environment Division with a specified implementor name.

MSCS
See Mass Storage Control System.

Negated Combined Condition
The 'NOT' logical operator immediately followed by a parenthesized combined condition.

Negated Simple Condition
The 'NOT' logical operator immediately followed by a simple condition.

Next Executable Statement
The next statement to which control will be transferred after execution of the current statement is complete.

Next Record
The record which logically follows the current record of a file.

Noncontiguous Items
Elementary data items, in the Working-Storage and Linkage sections, which bear no hierarchic relation to other data items.

Nonnumeric Item
A data item whose description permits its contents to be composed of any combination of characters taken from the computer's character set. Certain categories of nonnumeric items may be formed from more restricted character sets.

Nonnumeric Literal
A characterstring within quotation marks. The string of characters may include any character in the computer's character set. To represent a single quotation mark character within a nonnumeric literal, two contiguous quotation marks must be used.

Numeric Character
A character that belongs to the following set of digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Numeric Item

A data item whose description restricts its contents to a value represented by characters chosen from the digits '0' to '9'. If signed, the item may also contain a '+', '-' or any other representation of an operational sign.

Numeric Literal

A literal composed of one or more numeric characters that also may contain either a decimal , or an algebraic sign, or both. The decimal point must not be the rightmost character. The algebraic sign, if present, must be the leftmost character.

Object Computer

The name of an Environment Division paragraph in which the computer environment, where the object program is executed, is described.

Object Program

A set or group of executable macine language instructions and other material designed to interact with data to provide problem solutions. In this context, an object program is generally the machine language result of the operation of a COBOL compiler on a source program. Where there is no danger of ambiguity, the word 'program' alone may be used instead of the phrase 'object program'.

Object Time

The time at which an object program is executed.

Open Mode

The state of a file after execution of an OPEN statement for that file, and before the execution of a CLOSE statement for that file. The particular open mode is specified in the OPEN statement as either INPUT, OUTPUT, I-O or EXTEND.

Operand

Whereas the general definition of operand is 'that component which is operated upon', for the purpose of this publication, any lower case word (or words) that appears in a statement or entry format may be considered to be an operand and, as such, an implied reference to the data indicated by the operand.

Operational Sign

An algebraic sign, associated with a numeric data item or a numeric literal, to indicate whether its value is positive or negative.

Option
A phrase in which a choice can be made between alternate wordings.

Optional Word
A reserved word that is included in a specific format only to improve the readability of the language, and whose presence is optional to the user when the format in which the word apppears is used in a source program.

Output File
A file that is opened in either the output mode or extend mode.

Paragraph Header
A reserved word, followed by a period and a space that indicates the beginning of a paragraph in the Identification and Environment Divisions.

Paragraph Name
A user defined word that identifies and begins a paragraph in the procedure division.

Phrase
A phrase is an ordered set of one or more consecutive COBOL character strings that form a portion of a COBOL procedural statement or of a COBOL clause.

Physical Record
See Block.

Prime Record Key
A key whose contents uniquely identify a record within an indexed file.

Procedure
A paragraph or a group of logically successive paragraphs or a section or a group of logically consecutive sections, within the Procedure Division.

Procedure Name
A user defined word used to name a paragraph or section in the Procedure Division. It consists of a paragraph name (which may be qualified), or a section name.

Program Name
A user defined word that identifies a COBOL source program.

Punctuation Character
A character that belongs to the following set:

Character:	*Meaning:*
,	*comma*
;	*semicolon*
.	*period*
"	*quotation mark (double)*
'	*quotation mark (single) or apostrophe*
(	*left parenthesis*
)	*right parenthesis*
	space
=	*equal sign*

Qualified Dataname
An identifier composed of a dataname, followed by one or more sets of either of the connectives OF and IN, followed by a dataname qualifier.

Qualifier

1) A dataname used in a reference together with another data name at a lower level in the same hierarchy.

2) A section name used in a reference together with a paragraph name specified in that section.

3) A library name used in a reference together with a text name associated with that library.

Random Access
An access mode in which the program specified value of a key data item identifies the logical record that is obtained from, deleted from or placed into a relative or indexed file.

Record
See Logical Record.

Record Area
A storage area allocated to the purpose of processing the record described in a record description entry in the file section.

Record Description
See Record Description Entry.

Record Description Entry
The total set of data description entries associated with a particular record.

Record Key
A key, either the prime record key or an alternate record key, whose contents identify a record within an indexed file.

Record Name
A user defined word that names a record described in a record description entry in the Data Division.

Reference Format
A format that provides a standard method for describing COBOL source programs.

Relation
See Relational Operator.

Relation Character
A character belonging to the following set:

Character:	*Meaning:*
>	*greater than*
<	*less than*
=	*equal to*

Relation Condition
The proposition, for which a truth value can be determined, that the value of an arithmetic expression or data item has a specific relationship to the value of another arithmetic expression or data item (see Relational Operator).

Relational Operator
A reserved word, a relation character, a group of consecutive reserved words or a group of consecutive reserved words and relation characters used in the construction of a relation condition. The permissible operators and their meanings are:

Relational Operator:	*Meaning:*
IS [NOT] GREATER THAN *IS [NOT] >*	*Greater than or not greater than*
IS [NOT] LESS THAN *IS [NOT] <*	*Less than or not less than*
IS [NOT] EQUAL TO *IS [NOT] =*	*Equal to or not equal to*

Relative File
A key whose contents identify a logical record in a relative file.

Relative Organization
The permanent logical file structure in which each record is uniquely identified by an integer value greater than zero, which specifies the record's logical ordinal position in the file.

Reserved Word
A COBOL word specified in the list of words which may be used in COBOL source programs, but cannot appear in the programs as user defined words or system names.

Routine Name
A user defined word that identifies a procedure written in a language other than COBOL.

Run Unit
A set of one or more object programs which function, at object time, as a unit to provide problem solutions.

Section
A set of zero, one, or more paragraphs or entries called a section body, the first of which is preceded by a section header. Each section consists of the section header and the related section body.

Section Header
A combination of words followed by a period and a space that indicates the beginning of a section in the Environment, Data and Procedure Division.

In the Environment and Data Divisions, a section header is composed of reserved words followed by a period and a space. The permissible section headers are:

In the Environment Division:
CONFIGURATION SECTION
INPUT-OUTPUT SECTION
In the Data Division:
FILE SECTION
WORKING-STORAGE SECTION
LINKAGE SECTION

In the Procedure Division, a section header is composed of a section name, followed by the reserved word SECTION, followed by a period and a space.

Section Name
A user defined word which names a section in the Procedure Division.

Sentence
A sequence of one or more statements, the last of which is terminated by a period followed by a space.

Separator
A Punctuation Character used to delimit character strings.

Sequential Access
An access mode in which logical records are obtained from or placed into a file in a consecutive predecessor to successor logical record sequence determined by the order of records in the file.

Sequential File
A file with sequential organization.

Sequential Organization
The permanent logical file structure in which a record is identified by a predecessor successor relationship established when the record is placed into the file.

Sign Condition
The proposition, for which one of the following truth values can be determined: that the algebraic value of a data item or an arithmetic expression is either less than, greater than, or equal to zero.

simple condition
any single condition chosen from the set:

relation condition
class condition
conditionname condition
switch status condition
sign condition
(simple condition)

Sort File
A collection of records to be sorted by a SORT statement. The sort file is created and can be used by the sort function only.

Sort/Merge File Description Entry
An entry in the File Section of the Data Division that is composed of tha level indicator SD, followed by a file name, and then followed by a set of file clauses as required.

SOURCE-COMPUTER
The name of an Environment Division paragraph in which the computer environment where the source program is compiled, is described.

Source Program
Although it is recognized that a source program may be represented by other forms and symbols, in this document it always refers to a syntactically correct set of COBOL statements beginning with an Identification Division and ending with the end of a Procedure Division. In contexts where there is no danger of ambiguity, the word "program" alone may be used in place of the phrase "source program".

Special Character
A character that belongs to the following set:

Character:	*Meaning:*
+	*plus sign*
-	*minus sign*
*	*asterisk*
/	*stroke (virgule, slash)*
=	*equal sign*
$	*currency sign*
,	*comma (decimal point)*
;	*semicolon*
.	*period (decimal point)*
"	*quotation mark (double)*
'	*quotation mark (single)*
(	*left parenthesis*
)	*right parenthesis*
>	*greater than symbol*
<	*less than symbol*

Special Character Word
A reserved word which is an arithmetic operator or a relation character

SPECIAL-NAMES
The name of an Environment Division paragraph in which implementor names are related to user specified mnemonic names.

Special Registers
Compiler generated storage areas whose primary use is to store information in conjunction with the use of specific COBOL features.

Statement
A syntactically valid combination of words and symbols, beginning with a verb, and written in the Procedure Division.

Subprogram
See Called Program.

Subscript
An integer whose value identifies a particular element in a table.

Subscripted Data Name
An identifier that is composed of a data name followed by one or more subscripts enclosed in parentheses.

System Name
A COBOL word which is used to communicate with the operating environment.

Table
A set of logically consecutive items of data that are defined in the Data Division by means of the OCCURS clause.

Table Element
A data item that belongs to the set of repeated items comprising a table.

Truth Value
The representation of the result of the evaluation of a condition in terms of one of two values:*true* or *false*

Unary Operator
A plus (+) or a minus (-) sign, which precedes a variable or a left parenthesis in an arithmetic expression and which has the effect of multiplying the expression by +1 or -1, respectively.

User Defined Word
A COBOL word that must be supplied by the user to satisfy the format of a clause or statement.

Variable
A data item whose value may be changed by the execution of the object program. A variable used in an arithmetic expression must be a numeric elementary item.

Verb
A word that expresses an action to be taken by a COBOL compiler or object program.

Word
A characterstring of not more than 30 characters which forms a user defined word, a system name or a reserved word.

Working-Storage Section
The section of the Data Division that describes Working-Storage data items, composed either of non-contiguous items or of Working-Storage records or of both.

77 Level Description Entry

A data description entry that describes a non-contiguous data item with the level number 77.

Index

********** SEND US YOUR COMMENTS!!! **********

Are you frustrated because of unclear information in this manual? Do you have trouble finding things? Why don't you join the Reader's Club and send us a note? You will receive a membership card — and an answer to your comments.

Please let us know if you

* find errors
* cannot understand information
* cannot find information
* find needless information

Do you think we could improve the manual by rearranging the contents? You could also tell us if you like the manual!

********** HELP YOURSELF BY HELPING US!! **********

Manual name: ND COBOL Reference Manual

Manual number: 60.144.3 EN

What problems do you have? (use extra pages if needed) _____

_____

_____

_____

_____

Do you have suggestions for improving this manual ? _____

_____

_____

_____

_____

_____

Your name: _____ Date: _____

Company: _____ Position: _____

Address: _____

_____

What are you using this manual for ? _____

_____

NOTE!

This form is primarily for documentation errors. Software and system errors should be reported on Customer System Reports.

Send to:

Norsk Data A.S
Documentation Department
P.O. Box 25, Bogerud
0621 Oslo 6, Norway

Norsk Data's answer will be found on reverse side

Answer from Norsk Data _____

Answered by _____ Date _____

Norsk Data A.S
Documentation Department
P.O. Box 25, Bogerud
0621 Oslo6, Norway

Systems that put people first

NORSK DATA A.S OLAF HELSETS VEI 5 P.O. BOX 25 BOGERUD 0621 OSLO 6 NORWAY
TEL.: 02 - 29 54 00 - TELEX: 18284 NDN