File No. S360-24
Order No. GC28-6396-5

OS

IBM Systems Reference Library

IBM OS Full American National Standard COBOL

Program Numbers: (Version 2) 360S-CB-545

(Version 3) 5734-CB1

(Version 4) 5734-CB2 (Compiler and Library)

5734-LM2 (Library Only)

OS/VS COBOL 5740-CB1 (Compiler and Library)
5740-LM1 (Library only)

This publication gives the programmer the rules for writing programs that are to be compiled by the IBM OS/VS COBOL and IBM Full American National Standard COBOL compilers under the Operating System. It is meant to be used as a reference manual in the writing of IBM American National Standard COBOL programs.

COBOL (COmmon Business Oriented Language) is a programming language, similar to English, that is used for commercial data processing. It was developed by the Conference On DAta SYstems Languages (CODASYL). The standard of the language is American National Standard COBOL X3.23-1968, as approved by the American National Standards Institute (ANSI). American National Standard COBOL is compatible with, and identical to, international standard ISO/R 1989-1972 Programming Language COBOL.

IBM OS/VS COBOL and IBM OS Full American National Standard COBOL, Versions 2, 3, and 4, incorporate the eight processing modules defined in the highest level of the American National Standard. These modules include:

- Nucleus
- Table Handling
- Sequential Access
- Random Access
- Sort
- Report Writer
- Segmentation
- Library

A significant number of IBM extensions are implemented as well; these extensions are printed on a shaded background.

PREFACE

COBOL (COmmon Business Oriented Language) is a programming language, similar to English, that is used for commercial data processing. It was developed by the Conference On DAta SYstems Languages (CODASYL). The standard of the language is American National Standard COBOL X3.23-1968 as approved by the American National Standards Institute (ANSI). American National Standard COBOL is compatible with, and identical to, the international standard ISO/R 1989-1972 Programming Language COBOL.

IBM OS/VS COBOL and IBM OS Full American National Standard COBOL incorporate the eight processing modules defined in the highest level of the American National Standard. These modules include:

- Nucleus
- Table Handling
- Sequential Access
- Random Access
- Sort
- Report Writer
- Segmentation
- Library

A significant number of IBM extensions are implemented as well.

This manual describes IBM OS/VS COBOL and all current versions of IBM OS Full American National Standard COBOL -- Versions 2, 3, and 4. Information relating only to the Version 3 and Version 4 compilers is presented within separate paragraphs. Such paragraphs begin with the heading "Program Product Information", followed by the Version number of the compiler. Paragraphs following these headings that contain program product information are indented. Information relating only to the OS/VS COBOL compiler is presented in the separate chapter "OS/VS COBOL Considerations".

This publication gives the programmer the rules for writing programs that are to be compiled by the IBM OS/VS COBOL and IBM OS Full American National COBOL compilers under the Operating System. It is meant to be used as a reference manual in the writing of IBM American National Standard COBOL programs.

In this publication, the term standard COBOL means American National Standard COBOL; the terms IBM Full American National Standard COBOL and this compiler mean the IBM implementation of the highest level of American National Standard COBOL and all extensions. There are two types of extensions:

1. Those that represent features not approved by American National Standard COBOL.
2. Those that represent an easing of the strict American National Standard COBOL rules for greater programming convenience.

Sixth Edition (June 1975)

This edition is a reprint of SGC28-6396-3 and GC28-6396-4 incorporating changes released in technical newsletters GN28-1002 (dated July 15, 1972) and GN28-1048 (dated May 15, 1974).

This edition applies to the IBM OS Full American National Standard COBOL, Version 2 at the Release 21 level of the Operating System and language for Version 3, Version 4, and OS/VS COBOL.

Information in this publication is subject to significant change. Any such changes will be published in new editions or technical newsletters. Before using the publication, consult the latest IBM System/360 and System/370 Bibliography, GA22-6822, and the technical newsletters that amend the bibliography, to learn which editions and technical newsletters are applicable and current.

Requests for copies of IBM publications should be made to the IBM branch office that serves you.

A form for readers' comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, System Development Division, LDF Publishing Department J04, 1501 California Avenue, Palo Alto, California 94304.

All such extensions are printed on a shaded background for the convenience of users who wish strict conformance with the standard. Use of features that are extensions may result in incompatibilities between the implementation represented by this document and other implementations. If a complete chapter is an extension, only the page heading is shaded. These chapters are:

OS/VS COBOL Considerations

Subprogram Linkage Statements

Debugging Language

Format Control of the Source Program Listing

Sterling Currency

Teleprocessing (Version 4)

String Manipulation (Version 4)

For the less experienced programmer, the introduction summarizes the general principles of COBOL, highlights features of American National Standard COBOL and, through an example, illustrates the logical sequence and interrelationship of commonly used elements of a COBOL program. The balance of the publication gives the specific rules for correct programming in IBM Full American National Standard COBOL, as implemented for the IBM Operating System. Appendixes provide supplemental information useful in writing IBM American National Standard COBOL programs. Appendix A describes the use of intermediate results in arithmetic operations. Appendix B contains several sample programs showing the use of mass storage files. Appendix C gives a summary of all formats and reserved words used in the IBM implementation of Full American National Standard COBOL. This appendix can be torn out, folded, and used as a pocket-size reference card. Appendix D gives a summary of the applicable statements and clauses for each file-processing technique. Appendix E gives considerations for the use of ASCII encoded files. Appendix F explains the symbolic debugging features of the Version 4 Compiler. Appendix G describes COBOL processing for the 3505 and 3525 devices.

Compiler output and restrictions, programming examples, and information about running an IBM American National Standard COBOL program are found in the publications:

IBM OS Full American National Standard COBOL Compiler and Library, Version 2, Programmer's Guide, Order No. GC28-6399

IBM OS Full American National Standard COBOL Compiler and Library, Version 3, Programmer's Guide, Order No. SC28-6437

IBM OS Full American National Standard COBOL Compiler and Library, Version 4, Programmer's Guide, Order No. SC28-6456

The appropriate programmer's guide and this language reference manual are corequisite publications.

A knowledge of basic data processing techniques is mandatory for the understanding of this publication. Such information can be found in the following publications:

Introduction to IBM Data Processing Systems, Order No. GC20-1684

Introduction to IBM System/360 Direct Access Stcrage Devices and Organization Methods, Order No. GC20-1649

The reader should also have a general knowledge of COBOL before using this manual. Useful background information can be found in the following publications:

American National Standard COBOL Coding:

Card And Tape Applications Text, Order No. SR29-0283

Coding Techniques And Disk Applications Text, Order No. SR29-0284

Illustrations, Order No. SR29-0285

Student Reference Guide, Order No. SR29-0286

Where information in the foregoing publications conflicts with information in this publication, the contents herein supersede any other in the writing of COBOL programs. Any violation of the rules defined in this publication for using the Operating System is considered an error.

A general knowledge of the IBM Operating System is desirable, although not mandatory. The following publication gives such information:

IBM System/360 Operating System: Introduction, Order No. GC28-6534

ACKNOWLEDGMENT

The following extract from Government Printing Office Form Number 1965-0795689 is presented for the information and guidance of the user:

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

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

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

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

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

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

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

Date of Publication: May 15, 1974
Form of Publication: TNL GN28-1048 to GC28-6396-3 and -4

IBM OS/VS COBOL

New: Programming Features

- WHEN-COMPILED special register
- OBJECT-COMPUTER paragraph -- automatic System/370 instruction generation
- VSAM file processing
- Merge facility
- 3886 OCR support
- FIPS Flagger support
- Miscellaneous processing considerations
  (plus all features of IBM OS Full American National Standard COBOL, Version 4)

Miscellaneous changes for OS/VS COBOL and IBM OS Full American National Standard COBOL, Versions 2, 3, and 4

Maintenance: Documentation Only

Minor technical changes and corrections

Editorial changes that have no technical significance are not noted here.

Specific changes to the text made as of this publishing date are indicated by a vertical bar to the left of the text. These bars will be deleted at any subsequent republication of the page affected.

Date of Publication: July 15, 1972
Form of Publication: TNL GN28-1002 to GC28-6396-3

IBM OS Full American National Standard COBOL, Version 4

New: Programming Changes

- EGI (End Of Group Indicator) substituted for ETI (End Of Transmission Indicator) in the SEND statement.
- Changes in implementation for the UNSTRING statement, as well as clarifications in documentation.

IBM OS Full American National Standard COBOL, Versions 2, 3, and 4

Maintenance: Documentation only
Minor technical changes and corrections.

Editorial changes that have no technical significance are not noted here.

Specific changes to the text made as of this publishing date are indicated by a vertical bar to the left of the text. These bars will be deleted at any subsequent republication of the page affected.

Summary of Amendments — Number 6

Date of Publication: May 1972
Form of Publication: Revision, GC28-6396-3

IBM OS Full American National Standard COBOL, Version 4

New: Programming Features

- Special Registers DATE, DAY, and TIME.
- ASSIGN clause device field as comments.
- Dynamic Subprogram Linkage (dynamic CALL statement and CANCEL statement).
- Teleprocessing Feature.
- String Manipulation Feature.
- Symbolic Debugging Feature and Example.
- 3525 Combined Function Processing.

IBM OS Full American National Standard COBOL, Versions 3 and 4

Maintenance: Documentation Only

ASCII tape file processing clarifications.

Miscellaneous Changes for Versions 2, 3, and 4

Maintenance: Documentation only

Minor technical changes and corrections.

Editorial changes that have no technical significance are not noted here.

Specific changes to the text made as of this pullishing date are indicated by a vertical bar to the left of the text. These bars will be deleted at any subsequent replublication of the page affected.

Summary of Amendments — Number 4

Date of Publication: June 1, 1971
Form of Publication: TNL GN28-0439 to GC28-6393-2

IBM OS Full American National Standard COBOL, Version 3

New: Programming Features

- ASSIGN clause specification of new device numbers.
- USAGE COMPUTATIONAL-4 (system-independent binary).

IBM OS Full American National Standard COBOL, Version 2

New: Documentation Only

- ASSIGN clause specification for new device numbers.

Miscellaneous Changes for Version 2 and Version 3

Minor technical changes and corrections.

Summary of Amendments — Number 5

Date of Publication: January 15, 1972
Form of Publication: TNL GN28-0478 to GC28-6396-2

IBM OS Full American National Standard COBOL, Version 3

New: Programming Features

- SORT-MESSAGE special register implementation.
- SORT-CORE-SIZE and SORT-RETURN special registers—added processing capabilities.

Summary of Amendments — Number 3

Date of Publication: December 30, 1970
Form of Publication: TNL GN28-0428 to GC28-6396-2

IBM OS Full American National Standard COBOL, Version 3

New: Programming Features

- SIGN clause implementation.
- ASCII tape file processing.
- OBJECT-COMPUTER paragraph requests System/370 instructions.
- RERUN at end-of-volume.
- RECORD CONTAINS 0 CHARACTERS implementation.
- Error Declarative GIVING option enhancement.
- START statement with generic key.
- ON statement enhancement.

Miscellaneous Changes for Versions 2 and 3

Maintenance: Documentation only

- BLOCK CONTAINS 0 CHARACTERS description.
- LABEL RECORDS clause clarification.
- PICTURE clause description and table of precedence.
- USAGE clause description.
- Added examples.
- Minor technical changes and corrections.

Summary of Amendments — Number 1

Date of Publication: March 1969
Form of Publication: Revision, GC28-6396-1

Implementation Change

New: Programming change.

FREE statement deleted.

Miscellaneous Changes

New: Documentation only

- Table Handling clarifications.
- Table Handling sample program

Maintenance: Documentation only

Minor technical changes and corrections.

Summary of Amendments — Number 2

Date of Publication: June 1970
Form of Publication: Revision, GC28-6396-2

IBM OS Full American National Standard COBOL, Version 2

New: Programming Features

- S-mode (spanned) record processing.
- TOTALING/TOTALED option for LABEL RECORDS clause.
- CLOSE statement change in implementation.

Miscellaneous Changes

Maintenance: Documentation only

Minor technical changes and corrections.

FIGURES

Special OS/VS COBOL considerations are discussed in the following pages. Implementation areas described are:

- WHEN-COMPILED Special Register
- The Configuration Section
- VSAM (Virtual Storage Access Method) Processing
- Merge Facility
- 3886 OCR (Optical Character Reader) Processing
- FIPS (Federal Information Processing Standard) Flagger
- Miscellaneous Processing Considerations

OS/VS COBOL supports all of the additional features described in this chapter. Support for these features is provided through a subset of the complete COBOL language as documented in CODASYL COBOL Journal Of Development. IBM-specified language capabilities are also implemented. All features of OS Full American National Standard COBOL, Versions 3 and 4, continue to he supported.

The OS/VS COBOL Compiler and Library Program Product operates under control of OS/VS1 or OS/VS2 (with or without TSO), and the CMS component of VM/370. OS/VS1 and OS/VS2 can operate as independent systems or under control of VM/370. To execute OS/VS COBOL object programs the OS/VS COBOL Subroutine Library is required.

Additional information on OS/VS can be found in the following publications:

Introduction to Virtual Storage In System/370, Order No. GR20-4260

OS/VS1 Planning and Use Guide, Order No. GC24-5090

OS/VS2 Planning and Use Guide, Order No. GC28-0600

OS/VS Data Management for Systems Programmers, Order No. GC28-0631

OS/VS Virtual Storage Access Method (VSAM) Planning Guide, Order No. GC26-3799

WHEN-COMPILED SPECIAL REGISTER

The WHEN-COMPILED special register makes available to the object program the date-and-time-compiled constant carried in the object module.

WHEN-COMPILED is a 20-byte alphanumeric field valid only as the sending field in a MOVE statement. The format of these twenty bytes is hh.mm.ssMMM DD, YYYY (hour.minute.secondmonth day, year).

For example, if the compilation began at 4:31 PM on June 10, 1973, WHEN-COMPILED would contain the value

16.31.00JUN 10, 1973

This special register is a programmer aid that provides a means of associating a compilation listing with both the object program and the output produced at execution time.

CONFIGURATION SECTION

The Configuration Section describes the computer on which the source program is compiled, the computer on which the object program is executed, and, optionally, SPECIAL-NAMES, which relate function-names used by the compiler with user-specified mnemonic-names.

General Format

```
CONFIGURATION SECTION.

SOURCE-COMPUTER.  computer-name.

OBJECT-COMPUTER.  computer-name

    [MEMORY SIZE  integer  { WORDS      }  ]
                           { CHARACTERS }
                           { MODULES    }

    [SEGMENT-LIMIT IS  priority-number].

SPECIAL-NAMES.  [function-name  IS  mnemonic-name] ...

    [CURRENCY SIGN IS  literal]  [DECIMAL-POINT IS COMMA].
```

The Configuration Section and its associated paragraphs are optional in a COBOL source program.

SOURCE-COMPUTER PARAGRAPH

The SOURCE-COMPUTER paragraph describes the computer upon which the source program is to be compiled. This paragraph is treated as documentation.

Computer-name is a word in the form IBM-370[-model-number].

OBJECT-COMPUTER PARAGRAPH

The OBJECT-COMPUTER paragraph describes the computer upon which the object program is to be executed.

Computer-name must be the first entry in the OBJECT-COMPUTER paragraph. Computer-name is a word in the form IBM-370[-model-number].

System/370 instructions are provided automatically by OS/VS COBOL. (When IBM-360 is specified, the compiler generates System/370 instructions and issues a warning message.) The Compiler generates instructions from the System/370 set -- including Move Long (MVCL), Compare Logical Long (CLCL), and Shift And Round Decimal (SRP) -- that are particularly useful to COBOL. These System/370 instructions replace object-time subroutines and instructions that former COBOL Compilers generated under System/360 including routines and instructions to handle decimal arithmetic scaling (where operands have a different number of decimal places) and rounding. System/370 support also gives much improved processing of variable length fields.

Since System/370 does not require boundary alignment for COMPUTATIONAL, COMPUTATIONAL-1, and COMPUTATIONAL-2 items, no moves are generated for items that are not SYNCHRONIZED.

Performance Considerations:: Space occupied by an OS/VS COBOL program is decreased, particulary when calls to object-time subroutines, are no longer necessary. Such calls are always generated in System/360 for variable-length moves and comparisons. If there is at least one variable-length alphanumeric move in the source program, System/370 support reduces the size of the object program by at least 484 bytes; if there is at least one variable-length alphanumeric comparison, System/370 support reduces the size of the object program by at least an additional 498 bytes.

Number of Bytes in Each Move or Comparison	For Each Alphanumeric Move: Object-program Instructions		For Each Comparison (in a conditional expression): Object-program Instructions	
	System/360 Bytes Needed	System/370 Bytes Needed	System/360 Bytes Needed	System/370 Bytes Needed
Variable length	26+480*	14-22	26+496*	16-24
fixed length				
1-256	6-16	6-16	8-26	8-26
257-512	12-22	12-22	16-36	16-24
513-768	18-28	14-22	24-46	16-24
769-1024	24-34	14-22	32-56	16-24
1025-1280	30-40	14-22	40-66	16-24
1281-1536	36-46	14-22	48-76	16-24
...	...	...	...	...
>4096	26+480*	14-22	26+496*	16-24

*Bytes needed to invoke object-time subroutine, plus size of subroutine itself.

Figure I. Moves and Comparisons -- System/360 vs. System/370

Figure I gives comparative figures without right justification for fixed-length and variable-length MOVE statements, and for fixed-length and variable-length comaprisons.

Figure II gives comparative figures for Shift And Round Decimal generation; the savings shown are made for each such operation in the object program.

The MEMORY SIZE clause can be used to document the actual equipment configuration needed to run the object program.

The SEGMENT-LIMIT clause is discussed in the Segmentation Chapter.

Except for the SEGMENT-LIMIT clause, the OBJECT-COMPUTER paragraph is treated as documentation.

Function	System/360 Bytes Needed	System/370 Bytes Needed
Rounding	39 + literal*	6
Left Scaling	6 + literal*	6
Right Scaling	12	6

*As used for decimal point alignment the literal varies in length with size of data-item, number of decimal positions defined, and/or scaling positions defined.

Figure II. Shift And Round Decimal (SRP) -- System/360 vs. System/370

SPECIAL-NAMES PARAGRAPH

The SPECIAL-NAMES paragraph as discussed in the Environment Division chapter applies to OS/VS COBOL without change.

VSAM FILE PROCESSING

VSAM (Virtual Storage Access Method) is a high-performance access method of OS/VS for use with direct access storage. VSAM provides high-speed retrieval and storage of data, more reliability, flexible data organization, ease of conversion from other access methods, and ease of use -- including simplified job control statements, data protection against unauthorized access, central control of data management functions, device independence (freedom from consideration of block sizes, control information, record deblocking, etc.), and cross-system compatibility.

Access Method Services, a multi-function utility program is used to define a VSAM data set, and optionally load records into it, convert an existing indexed or sequential data set to VSAM format, and perform other tasks as well. Access Method Services is described in OS/VS Virtual Storage Access Method (VSAM) Planning Guide, Order No. GC26-3799.

VSAM allows key-sequenced and entry-sequenced data sets; records can be fixed or variable in length.

In a key-sequenced data set (KSDS), records are stored in the ascending collating sequence of some embedded key field. For indexed files of this type, records can be retrieved sequentially in key sequence; they can also be retrieved randomly according to the particular value of the key.

In an entry-sequenced data set (ESDS), the records are stored in the order in which they are presented for inclusion in the data set. New records are stored at the end of the data set. In COBOL, record retrieval for sequential files of this type must be sequential.

VSAM files may be written on the following mass storage devices: 2314, 2319, 3330, 3340.

For VSAM file processing in COBOL, there are special language considerations in the Environment, Data, and Procedure Divisions.

ENVIRONMENT DIVISION -- FILE-CONTROL PARAGRAPH

The File-Control paragraph names the VSAM file, associates it with an external medium, and allows specification of other file-related information.

General Format 1 -- Sequential VSAM Files

```
FILE-CONTROL.

    {SELECT [OPTIONAL] file-name

     ASSIGN TO system-name-1 [system-name-2] ...

    [RESERVE integer [AREA ]]
                     [AREAS]

    [ORGANIZATION IS SEQUENTIAL]

    [ACCESS MODE IS SEQUENTIAL]

    [PASSWORD IS data-name-1]

    [FILE STATUS IS data-name-2].} ...
```

General Format 2 -- Indexed VSAM Files

```
FILE-CONTROL.

    {SELECT file-name

     ASSIGN TO system-name-1 [system-name-2] ...

    [RESERVE integer [AREA ]]
                     [AREAS]

     ORGANIZATION IS INDEXED

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

     RECORD KEY IS data-name-3

    [PASSWORD IS data-name-1]

    [FILE STATUS IS data-name-2].} ...
```

Each file described by an FD entry or SD entry in the Data Division must be described in one and only one File-Control entry.

The key word FILE-CONTROL may appear only once, at the beginning of the File-Control paragraph. The word FILE-CONTROL must begin in Area A, and be followed by a period followed by a space.

Each File-Control entry must begin with a SELECT clause followed immediately by an ASSIGN clause. The order in which the other clauses

appear is not significant, except that for indexed VSAM files the PASSWORD clause, if specified, must immediately follow the RECORD KEY clause. Each File-Control entry must end with a period followed by a space.

Each data-name in the File-Control entry may be qualified; it may not be subscripted or indexed. Each data-name must be at a fixed displacement from the beginning of the data description entry in which it appears; that is, it must not appear in the entry after an OCCURS DEPENDING ON clause.

SELECT Clause

The SELECT clause is used to name each file in the program. Each file described with an FD entry or SD entry in the Data Division must be named once and only once as a file-name following the key word SELECT.

FORMAT 1: The OPTIONAL clause must be specified for input files that are not necessarily present each time the object program is executed.

If file-name represents a sort file, only the ASSIGN clause may be written following the SELECT clause.

ASSIGN Clause

The ASSIGN clause associates the file with an external storage medium.

System-name specifies the external-name, and, optionally, a device class, a device number, and the file organization. System-name has the following structure.

```
[SYSnnn-][class-][device-][organization-]ddname
```

The SYSnnn, class, and device fields are included for compatibility only; these fields are treated as documentation.

The organization field is required for sequential VSAM files. The entry must be AS.

The organization field must not be specified for indexed VSAM files.

The ddname field is required. It is a 1-character to 8-character field, specifying the external-name by which the file is known to the system.

RESERVE Clause

The RESERVE clause is treated as documentation.

ORGANIZATION Clause

The ORGANIZATION clause specifies the logical structure of the file. The file organization is established at the time the file is defined and cannot subsequently be changed.

FORMAT 1: If the ORGANIZATION clause is omitted, ORGANIZATION SEQUENTIAL is assumed.

When ORGANIZATION SEQUENTIAL is specified or assumed, the records in the file are positioned sequentially in the order they were created. Once established, the position of the file records does not change.

FORMAT 2: When ORGANIZATION INDEXED is specified, each logical record in the file contains an embedded RECORD KEY which is associated with an index, and each record is identified through its RECORD KEY value. After records have been updated, or have been added to or deleted from the file, the position of the records may have changed.

ACCESS MODE Clause

The ACCESS MODE clause specifies the manner in which records in the file are to be processed.

When the ACCESS MODE clause is omitted, ACCESS MODE SEQUENTIAL is assumed.

When ACCESS MODE SEQUENTIAL is specified or assumed, the records are processed sequentially. That is, the next logical record in the file is the next processed.

ACCESS SEQUENTIAL can be specified for sequential or indexed VSAM files.

When ORGANIZATION IS SEQUENTIAL is specified or assumed, the records in the file are processed in the sequence established when the file was created or extended.

When ORGANIZATION IS INDEXED is specified, the records in the file are processed in the sequence of ascending record key values.

FORMAT 2: For indexed VSAM files, ACCESS MODE RANDOM and ACCESS MODE DYNAMIC can also be specified.

When ACCESS MODE RANDOM is specified, the sequence in which records are processed is determined by the sequence in which keys are presented. The desired record is accessed by placing the value of its key in the RECORD KEY data item before the associated input/output statement is executed.

When ACCESS MODE DYNAMIC is specified, records in the file are processed sequentially and/or randomly. The form of the specific input/output request determines the access mode.

RECORD KEY Clause (Format 2)

The RECORD KEY clause specifies the data item within the record which contains the key for that record; each RECORD KEY value in the file must be unique. A RECORD KEY must be specified for an indexed VSAM file.

Data-name-3 is the RECORD KEY data item. Data-name-3 must be defined as a fixed length alphanumeric or external-decimal unsigned integer data item within a record description entry associated with file-name. Data-name-3 is treated as an alphanumeric item.

The data description of data-name-3 and its relative location in the record must be the same as that specified when the file was defined.

PASSWORD Clause

The PASSWORD clause controls object-time access to the file.

Data-name-1 is the password data item; it must be defined in the Working-Storage Section as an alphanumeric item. The first 8 characters are used as the password; a shorter field is padded with blanks to 8 characters. The password data item must be equivalent to the one externally specified.

When the PASSWORD clause is specified, at object time the password data item must contain the valid password for this VSAM file before the file can be successfully opened. (See "Status Key" in the following Common Processing Facilities description.)

FILE STATUS Clause

The FILE STATUS clause allows the user to monitor the execution of each input/output request for the file.

Data-name-2 is the Status Key data item. Data-name-2 must be defined in the Data Division as a two-character alphanumeric or external-decimal unsigned integer item. Data-name-2 is treated as an alphanumeric item. Data-name-2 must not be defined in the File Section or the Report Section.

When the FILE STATUS clause is specified, a value is moved into the Status Key by the system after each input/output request that explicitly or implicitly refers to this file. The value indicates the status of the execution of the statement. (See "Status Key" in the following Common Processing Facilities description.)

ENVIRONMENT DIVISION -- I-O-CONTROL PARAGRAPH

The I-O-CONTROL paragraph specifies the special input/output techniques to be used in the program. The I-O-CONTROL paragraph and its associated clauses are optional.

General Format -- VSAM Files

```
I-O-CONTROL.
     [RERUN ON system-name EVERY integer RECORDS
          OF file-name-1] ...
     [SAME [RECORD] AREA
          FOR file-name-2 [file-name-3] ...] ... .
```

The key word I-O-CONTROL must begin in Area A and be followed by a period and a space.

RERUN Clause

The RERUN clause specifies that checkpoint records are to be taken.

System-name identifies the checkpoint file, and is specified as described in the Environment Division chapter. The checkpoint file must be a standard sequential file (it may not be a sequential VSAM file).

File-name represents the file for which checkpoint records are to be written. File-name may specify a VSAM file.

SAME Clause

The SAME RECORD AREA clause for VSAM files is implemented as described in the Environment Division chapter.

The SAME AREA clause, when specified for VSAM files, has the same meaning as the SAME RECORD AREA clause.

DATA DIVISION -- FD ENTRY

In the FD entry for a VSAM file, the RECORD CONTAINS clause is implemented as described in the Data Division chapter.

The BLOCK CONTAINS, DATA RECORDS, and VALUE OF clauses, are treated as documentation for VSAM files.

The RECORDING MODE and REPORT clauses must not be specified for VSAM files.

There are special considerations for the LABEL RECORDS clause.

LABEL RECORDS Clause

For VSAM files, the LABEL RECORDS clause specifies whether standard labels are present or omitted.

Format

```
LABEL { RECORD  IS } { STANDARD }
      { RECORDS ARE } { OMITTED  }
```

For VSAM files, either the STANDARD or the OMITTED option may be specified. Either option is treated as documentation.

The LABEL RECORDS clause is required in every FD entry.

PROCEDURE DIVISION

For VSAM files, there are several Common Processing Facilities that apply to more than one input/output statement. These Common Processing

Facilities are discussed before the descriptions of the separate input/output verbs.

Common Processing Facilities

CURRENT RECORD POINTER: Conceptually, the Current Record Pointer specifies the next record to be accessed by a sequential request. The setting of the Current Record Pointer is affected only by the OPEN, START, and READ statements. The concept of the Current Record Pointer has no meaning for random access or for output files.

STATUS KEY: If the FILE STATUS clause is specified in the File-Control entry, a value is placed into the specified Status Key (the 2-character data item named in the FILE STATUS clause) during execution of any request on that file; the value indicates the status of that request. The value is placed in the Status Key before execution of any Error Declarative or INVALID KEY/AT END option associated with the request.

The first character of the Status Key is known as Status Key 1; the second character is known as Status Key 2. Combinations of possible values and their meanings are shown in Figure III.

Status Key 1 Value	Meaning	Status Key 2 Value	Meaning
0	Successful Completion	0	No Further Information
1	At End (no next logical record, or an OPTIONAL file not available at OPEN time)	0	No Further Information
2	Invalid Key	1	Sequence Error
		2	Duplicate Key
		3	No Record Found
		4	Boundary Violation (indexed VSAM file)
3	Permanent Error (data check, parity check, transmission error)	0	No Further Information
		4	Boundary Violation (sequential VSAM file)
9	Other Errors	1	Password Failure
		2	Logic Error
		3	Resource Not Available
		4	No Current Record Pointer For Sequential Request
		5	Invalid or Incomplete File Information
		6	no DD card

Figure III. Status Key Values and Their Meanings

INVALID KEY CONDITION: The INVALID KEY condition can occur during execution of a START, READ, WRITE, REWRITE, OR DELETE statement. (For details of the causes for the condition, see documentation for those statements.) When the INVALID KEY condition is recognized, the following actions are taken in the following order:

1. If the FILE-STATUS clause is specified, a value is placed into the Status Key to indicate an INVALID KEY condition.

2. If the INVALID KEY option is specified in the statement causing the condition, control is transferred to the INVALID KEY imperative-statement. Any EXCEPTION/ERROR declarative procedure specified for this file is not executed.

3. If the INVALID KEY option is not specified, but an EXCEPTION/ERROR declarative procedure is specified for the file, the EXCEPTION/ERROR procedure is executed.

When an INVALID KEY condition occurs, the input/output statement which caused the condition is unsuccessful.

INTO/FROM IDENTIFIER OPTION: This option is valid for READ, REWRITE, and WRITE statements.

The INTO identifier option makes a READ statement equivalent to

```
READ  file-name

MOVE  record-name TO identifier
```

After successful execution of the READ statement, the current record becomes available both in the record-name and identifier.

The FROM identifier option makes a REWRITE or WRITE statement equivalent to

```
MOVE  identifier TO record-name

{REWRITE}
{WRITE  } record-name
```

After successful execution of the WRITE or REWRITE statement, the current record may no longer be available in record-name, but is still available in identifier.

In all cases, identifier must be the name of an entry in the Working-Storage Section, the Linkage Section, or of a record description for another previously opened file. Record-name/file-name and identifier must not refer to the same storage area.

EXCEPTION/ERROR Declarative

The EXCEPTION/ERROR Declarative specifies procedures for input/output exception or error handling that are to be executed in addition to the standard system procedures.

Format

```
USE AFTER STANDARD {EXCEPTION} PROCEDURE
                   {ERROR    }

        ON {file-name-1 [file-name-2] ...} .
           {INPUT                        }
           {OUTPUT                       }
           {I-O                          }
           {EXTEND                       }
```

A USE statement, when present, must immediately follow a section header in the Declaratives Section (see "Declaratives" in the Procedure Division chapter). A USE statement must be followed by a period followed by a space. The remainder of the section must consist of one or more procedural paragraphs that specify the procedures to be used.

The USE statement itself is not an executable statement; it merely defines the conditions for execution of the procedural paragraphs.

The words EXCEPTION and ERROR are synonymous and may be used interchangeably.

When the file-name option is specified, the procedure is executed only for the file(s) named. Appearance of a file-name must not cause simultaneous requests for the execution of more than one EXCEPTION/ERROR procedure. No file-name can refer to a sort file.

When the INPUT option is specified, the procedure is executed for all files opened in INPUT mode.

When the OUTPUT option is specified, the procedure is executed for all files opened in the OUTPUT mode.

When the I-O option is specified, the procedure is executed for all files opened in I-O mode.

When the EXTEND option is specified, the procedure is executed for all files opened in EXTEND mode.

The EXCEPTION/ERROR procedure is executed:

- Either after completing the standard system input/output error routine, or
- Upon recognition of an INVALID KEY or AT END condition when an INVALID KEY or AT END option has not been specified in the input/output statement, or
- Upon recognition of an IBM-defined condition which causes status key 1 to be set to 9.

After execution of the EXCEPTION/ERROR procedure, control is returned to the invoking routine.

The EXCEPTION/ERROR procedures are activated when an input/output error occurs during execution of a READ, WRITE, REWRITE, START, or DELETE statement.

If an OPEN statement is issued for a file already in the open status, the EXCEPTION/ERROR procedures are activated; when the execution of an OPEN statement is unsuccessful due to any other cause, the EXCEPTION/ERROR procedures are not activated.

If a file is in the OPEN status, and the execution of a CLOSE statement is unsuccessful, the EXCEPTION/ERROR procedures are activated. If the file is in a closed status and a CLOSE statement is issued, the EXCEPTION/ERROR procedures are not activated.

Within a declarative procedure, there must be no references to nondeclarative procedures. In nondeclarative procedures, there must be no references to declarative procedures, except that PERFORM statements may refer to procedure-names associated with a declarative procedure.

OPEN Statement

The OPEN statement initiates the processing of VSAM files.

Format

```
                 ⎧INPUT   file-name-1  [file-name-2] ...⎫
OPEN             ⎪OUTPUT  file-name-1  [file-name-2] ...⎬ ...
                 ⎪I-O     file-name-1  [file-name-2] ...⎪
                 ⎩EXTEND  file-name-1  [file-name-2] ...⎭
```

At least one of the options INPUT, OUTPUT, I-O, or EXTEND must be specified; there may be not more than one instance of each option specified in one OPEN statement, although more than one file-name may be specified with each option. The INPUT, OUTPUT, I-O, and EXTEND options may appear in the any order.

Each file-name designates a file upon which the OPEN statement is to operate. Each file-name must be defined in an FD entry in the Data Division, and must not name a sort file. The FD entry for the file must be equivalent to the information specified when the file was defined.

The successful execution of an OPEN statement determines the availability of the file and results in that file being in open mode. Before successful execution of the OPEN statement for a given file, no statement can be executed which refers explicitly or implicitly to that file. The successful execution of the OPEN statement makes the associated record area available to the program; it does not obtain or release the first data record.

The INPUT option permits opening the file for input operations.

The I-O option permits opening the file for both input and output operations.

The INPUT and I-O options are valid only for files which contain or which have contained records, whether or not the files still contain any records when the OPEN statement is executed. (That is, even if all the records in a file have been deleted, that file can still be opened INPUT or I-O.) The INPUT and I-O options must not be specified when the file has not been already created.

The OUTPUT option permits opening the file for output operations. This option can be specified when the file is being created. (The OUTPUT option must not be specified for a file that contains records, or which has contained records that have been deleted.)

The EXTEND option permits opening the file for output operations. ACCESS MODE SEQUENTIAL must be explicitly or implicitly specified. When EXTEND is specified, execution of the OPEN statement prepares the file for the addition of records immediately following the last record in the file. Subsequent WRITE statements add records to the file, as if the file had been opened OUTPUT. The EXTEND option can be specified when the file is being created; it can also be specified for a file which contains records, or which has contained records that have been deleted.

The OPEN mode, the ACCESS MODE, and the file ORGANIZATION determine the valid input/output statements for a given VSAM file. Figure IV shows permissible combinations.

File Organization and OPEN mode / ACCESS mode and I/O verb		INDEXED				SEQUENTIAL			
		INPUT	OUTPUT	I-O	EXTEND	INPUT	OUTPUT	I-O	EXTEND
SEQUENTIAL	OPEN	P	P	P	P	P	P	P	P
	READ	P	-	P	-	P	-	P	-
	WRITE	-	P	-	P	-	P	-	P
	REWRITE	-	-	P	-	-	-	P	-
	START	P	-	P	-	-	-	-	-
	DELETE	-	-	P	-	-	-	-	-
RANDOM	OPEN	P	P	P					
	READ	P	-	P					
	WRITE	-	P	P					
	REWRITE	-	-	P					
	START	-	-	-					
	DELETE	-	-	P					
DYNAMIC	OPEN	P	P	P					
	READ	P	-	P					
	WRITE	-	P	P					
	REWRITE	-	-	P					
	START	P	-	P					
	DELETE	-	-	P					

P indicates that this input/output statement is permissible for this combination of File Organization, Access Mode and OPEN Mode

- indicates that this input/output statement is not permissible for this combination of File Organization, Access Mode, and OPEN Mode

Figure IV. OPEN Statement Options and Permissible I/O Statements

A file may be opened for INPUT, OUTPUT, I-O, or EXTEND in the same program. After the first execution of an OPEN statement for a given file, each subsequent execution of an OPEN statement must be preceded by the successful execution of a CLOSE statement without the LOCK option.

Execution of an OPEN INPUT or OPEN I-O statement sets the Current Record Pointer to the first record existing in the file. For indexed files, the record with the lowest key value is considered the first record in the file. If no records exist in the file, the Current Record Pointer is set so that the first Format 1 READ statement executed results in an AT END condition.

If the PASSWORD clause is specified in the File-Control entry, the password data item must contain the valid password before the OPEN statement is executed. If the valid password is not present, the OPEN statement is unsuccessful.

If the FILE STATUS clause is specified in the File-Control entry, the associated Status Key is updated when the OPEN statement is executed.

If an OPEN statement is issued for a file already in the OPEN status, the EXCEPTION/ERROR procedure (if specified) for this file is executed.

START Statement

The START statement provides a means for logical positioning within an indexed file for subsequent sequential retrieval of records.

Format

```
START file-name [KEY IS { EQUAL TO                } data-name]
                        {  =                      }
                        {  GREATER THAN           }
                        {  >                      }
                        {  NOT LESS THAN          }
                        {  NOT <                  }
    [INVALID KEY imperative-statement]
```

When the START statement is executed, the associated file must be open in INPUT or I-O mode.

File-name must name an indexed VSAM file with sequential or dynamic access. File-name must be defined in an FD entry in the Data Division. File-name must not be the name of a sort file.

When the KEY option is not specified, the EQUAL TO relational operator is implied. When the START statement is executed, the EQUAL TO comparison is made between the current value in the RECORD KEY and the corresponding key field in the file's records. The Current Record Pointer is positioned to the logical record in the file whose key field satisfies the comparison.

When the KEY option is specified, data-name may be either

- The RECORD KEY for this file, or
- Any alphanumeric data item subordinate to the RECORD KEY whose leftmost character position corresponds to the leftmost character position of the RECORD KEY (that is, a generic key).

When the START statement is executed, the comparison specified in the KEY relational operator is made between data-name and the key field in the file's records. If the operands are of unequal size, the comparison proceeds as if the key field were truncated on the right to the length of the data-name. All other numeric and nonnumeric comparison rules apply. The Current Record Pointer is positioned to the first 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, and the position of the Current Record Pointer is undefined. (See "INVALID KEY Condition" in the preceding Common Processing Facilities Section.)

If the FILE STATUS clause is specified in the File-Control entry, the associated Status Key is updated when the START statement is executed.

READ Statement

For sequential access, the READ statement makes available the next logical record from a VSAM file. For random access, the READ statement makes available a specified record from a VSAM file.

```
┌──────────────────────────────────────────────────────────────┐
│                          Format 1                            │
├──────────────────────────────────────────────────────────────┤
│                                                              │
│   READ file-name  [NEXT] RECORD [INTO identifier]            │
│                                                              │
│       [AT END imperative-statement]                          │
│                                                              │
└──────────────────────────────────────────────────────────────┘
```

```
┌──────────────────────────────────────────────────────────────┐
│                          Format 2                            │
├──────────────────────────────────────────────────────────────┤
│                                                              │
│   READ file-name RECORD [INTO identifier]                    │
│                                                              │
│       [INVALID KEY imperative-statement]                     │
│                                                              │
└──────────────────────────────────────────────────────────────┘
```

When the READ statement is executed, the associated file must be open in INPUT or I-O mode.

File-name must be defined in an FD entry in the Data Division. File-name must not be the name of a sort file.

The INTO identifier option is described in the preceding Common Processing Facilities Section.

Following the unsuccessful execution of a READ statement, the contents of the associated record area and the position of the Current Record Pointer are undefined.

If the FILE STATUS clause is specified in the File-Control entry, the associated Status Key is updated when the READ statement is executed.

FORMAT 1: When ACCESS MODE SEQUENTIAL is specified or assumed for a VSAM file, this format must be used. For such files the statement makes available the next logical record from the file. For indexed VSAM files, the NEXT option need not be specified; for sequential VSAM files, the NEXT option must not be specified.

When ACCESS MODE DYNAMIC is specified for indexed VSAM files, the NEXT option must be specified for sequential retrieval. For such files, the READ NEXT statement makes available the next logical record from the file.

Before a Format 1 READ statement is executed, the Current Record Pointer must be positioned by the successful prior execution of an OPEN START, or READ statement. When the Format 1 READ statement is executed the record indicated by the Current Record Pointer is made available. For sequential VSAM files, the next record is the succeeding record in logical sequence. For a sequentially accessed indexed VSAM file, the next record is that one having the next higher RECORD KEY in collating sequence.

If the position of the Current Record Pointer is undefined when a Format 1 READ statement is issued, the execution of the statement is unsuccessful.

If, when a Format 1 READ statement is executed, no next logical record exists in the file, the AT END condition exists. The execution of the READ statement is considered unsuccessful.

When the AT END condition is recognized, the following actions are taken in the following order:

1. If the FILE-STATUS clause is specified in the File-Control entry, the Status Key is updated to indicate the AT END condition.
2. If the AT END option of the READ statement is specified, control is transferred to the AT END imperative-statement.
3. If the AT END option is not specified, and a USE AFTER EXCEPTION/ERROR procedure is specified, either explicitly or implicitly, that procedure is executed.

For files with SEQUENTIAL organization, when the AT END condition has been recognized, a READ statement for this file must not be executed until a successful CLOSE statement followed by a successful OPEN statement have been executed for this file.

For files with INDEXED organization, when the AT END condition is recognized, a Format 1 READ statement for this file must not be executed until one of the following has been successfully executed:

- A CLOSE statement followed by an OPEN statement
- A Format 2 READ statement (dynamic access)
- A START statement

If a sequential VSAM file with the OPTIONAL clause is not present at the time the file is opened, execution of the first READ statement causes the AT END condition to occur. Standard end-of-file procedures are not performed.

FORMAT 2: This format must be used for indexed VSAM files in random access mode, and for random record retrieval in the dynamic access mode.

Execution of a Format 2 READ statement causes the value in the RECORD KEY to be compared with the values contained in the corresponding key field in the file's records until a 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, an INVALID KEY condition exists, and execution of the READ statement is unsuccessful. (See "INVALID KEY Condition" in the preceding Common Processing Facilities Section.)

WRITE Statement

The WRITE statement releases a logical record to an OUTPUT, I-O, or EXTEND file.

Format
WRITE record-name [FROM identifier] [INVALID KEY imperative-statement]

When the WRITE statement is executed, the associated file must be open in OUTPUT, I-O, or EXTEND mode.

Record-name must be the name of a logical record in the File Section of the Data Division. Record-name may be qualified. Record-name must not be associated with a sort file.

The maximum record size for the file is established at the time the file is created, and cannot subsequently be changed.

Execution of the WRITE statement releases a logical record to the file associated with record-name.

After the WRITE statement is executed, the logical record is no longer available in record-name, unless:

- The associated file is named in a SAME RECORD AREA clause (in which case the record is also available as a record of the other files named in the SAME RECORD AREA clause), or
- The WRITE statement is unsuccessful due to a boundary violation.

In either of these two cases, the logical record is still available in record-name.

If the FROM identifier option is specified, then after the WRITE statement is executed, the information is still available in identifier, even though it may not be in record-name. (See "INTO/FROM Identifier Option" in the preceding Common Processing Facilities Section.)

The Current Record Pointer is not affected by execution of the WRITE statement.

The number of character positions required to store the record in a VSAM file may or may not be the same as the number of character positions defined by the logical description of that record in the COBOL program.

If the FILE STATUS clause is specified in the File-Control entry, the associated Status Key is updated when the WRITE statement is executed.

SEQUENTIAL VSAM FILES: The INVALID KEY option must not be specified.

When an attempt is made to write beyond the externally-defined boundaries of the file, the execution of the WRITE statement is unsuccessful, and an EXCEPTION/ERROR condition exists. The contents of record-name are unaffected. If an explicit or implicit EXCEPTION/ERROR procedure is specified for the file, the procedure is then executed; if no such procedure is specified, the results are undefined.

INDEXED VSAM FILES: Before the WRITE statement is executed, the contents of the RECORD KEY must be set to the desired value. Note that the value contained in any specific RECORD KEY must be unique within the records in the file.

When the WRITE statement is executed, the contents of the RECORD KEY are utilized so that subsequent access to the record can be based on the RECORD KEY.

If sequential access mode is specified or implied, records must be released to the file in ascending order of RECORD KEY.

If random or dynamic access is specified, records may be released in any program-specified order.

INVALID KEY Option: The INVALID KEY condition exists when any of the following conditions occur:

- For an OUTPUT or EXTEND file in sequential access mode, when the value of the RECORD KEY is not greater than the value of the RECORD KEY for the previous record.

- For an I-O or OUTPUT file in random or dynamic access mode, when the value of the RECORD KEY is equal to the value of a RECORD KEY for an already existing record.

- When an attempt is made to write beyond the externally-defined boundaries of the file.

When the INVALID KEY condition is recognized, the execution of the WRITE statement is unsuccessful, the contents of record-name are unaffected, and the Status Key, if specified, is set to a value to indicate the cause of the condition. (See "INVALID KEY Condition" and "Status Key" in the preceding Common Processing Facilities Section.)

REWRITE Statement

The REWRITE statement logically replaces an existing record in a VSAM file.

Format
REWRITE record-name [FROM identifier] [INVALID KEY imperative-statement]

When the REWRITE statement is executed, the associated file must be open in I-O mode.

Record-name must be the name of a logical record in the File Section of the Data Division. Record-name must not be associated with a sort file. Record-name may be qualified.

Execution of the REWRITE statement replaces an existing record in the file with the information contained in record-name. For a sequential VSAM file, the number of character positions in record-name must equal the number of character positions in the record being replaced. For an indexed VSAM file, the number of character positions in record-name need not equal the number of character positions in the record being replaced.

After successful execution of a REWRITE statement, the logical record is no longer available in record-name unless the associated file is named in a SAME RECORD AREA clause (in which case the record is also available as a record of the other files named in the SAME RECORD AREA clause).

If the FROM identifier option is specified, then after the REWRITE statement is executed, the information is still available in identifier, even though it may not be in record-name. (See "INTO/FROM Identifier Option" in the preceding Common Processing Facilities section.)

The Current Record Pointer is not affected by execution of the REWRITE statement.

If the FILE STATUS clause is specified in the File-Control entry, the associated Status Key is updated when the REWRITE statement is executed.

For files in the sequential access mode, the last prior input/output statement executed for this file must be a successfully executed READ statement. When the REWRITE statement is executed, the record retrieved by that READ statement is logically replaced.

SEQUENTIAL FILES: The INVALID KEY option must not be specified for this type of file. An EXCEPTION/ERROR declarative procedure may be specified.

INDEXED FILES: For an indexed file in the sequential access mode, the record to be replaced by the REWRITE statement is identified by the current value of the RECORD KEY. When the REWRITE statement is executed, the RECORD KEY must contain the value of the RECORD KEY for the last-retrieved record from the file.

For an indexed file in random or dynamic access mode, the record to be replaced is the record identified by the value of the RECORD KEY.

The INVALID KEY condition exists when:

- The access mode is sequential, and the value contained in the RECORD KEY of the record to be replaced does not equal the RECORD KEY of the last-retrieved record from the file.
- The value contained in the RECORD KEY does not equal that of any record in the file.

If either condition exists, the execution of the REWRITE statement is unsuccessful, the updating operation does not take place, and the data in record-name is unaffected. (See "INVALID KEY Condition" in the preceding Common Processing Facilities Section.)

DELETE Statement

The DELETE statement logically removes a record from an indexed VSAM file.

Format
DELETE file-name RECORD [INVALID KEY imperative-statement]

When the DELETE statement is executed, the associated file must be open in I-O mode.

File-name must be defined in an FD entry in the Data Division and must be the name of an indexed VSAM file.

For a file in sequential access mode, the INVALID KEY option must not be specified.

For a file in random or dynamic access mode, the INVALID KEY option may be specified.

For a file in sequential access mode, the last prior input/output statement must be a successfully executed READ statement. When the DELETE statement is executed, the system logically removes the record retrieved by that READ statement. The current record pointer is not affected by execution of the DELETE statement.

For a file in random or dynamic access mode, when the DELETE statement is executed, the system logically removes the record identified by the contents of the associated RECORD KEY data item. If the file does not contain the record specified by the key, an INVALID KEY condition exists. (See "INVALID KEY Condition" in the preceding Common Processing Facilities section.)

After successful execution of a DELETE statement, the record is logically removed from the file and can no longer be accessed. Execution of the DELETE statement does not affect the contents of the record area associated with file-name.

If the FILE STATUS clause is specified in the File-Control entry, the associated Status Key is updated when the DELETE statement is executed.

CLOSE Statement

The CLOSE statement terminates the processing of VSAM files.

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

A CLOSE statement may be executed only for a file in an open mode. After successful execution of a CLOSE statement, the record area associated with the file-name is no longer available. Unsuccessful execution of a CLOSE statement leaves availability of the record area undefined.

Each file-name designates a file upon which the CLOSE statement is to operate.

When the WITH LOCK option is not specified, standard system closing procedures are performed. This file may be opened again during this execution of the object program.

When the WITH LOCK option is specified, standard system closing procedures are performed; the compiler ensures that this file cannot be opened again during this execution of this object program.

After a CLOSE statement is successfully executed for the file, an OPEN statement for that file must be executed before any other input/output statement can refer explicitly or implicitly to the file.

If a CLOSE statement is not executed for an open file before the job is terminated (by, for example, execution of a STOP RUN or GOBACK statement), results are unpredictable.

If an input sequential VSAM file is described in the File-Control entry as OPTIONAL and the file is not present during this execution of the object program, standard end-of-file processing is not performed.

If the FILE STATUS clause is specified in the File-Control entry, the associated Status Key is updated when the CLOSE statement is executed.

If the file is in an OPEN status and the execution of the CLOSE statement is unsuccessful, the EXCEPTION/ERROR procedure (if specified) for this file is executed.

MERGE FACILITY

The Merge Facility gives the COBOL user access to the merging capabilities of the Program Product OS/VS Sort-Merge (Program Number 5740-SM1). Through COBOL, the user can combine two or more identically ordered input files into one output file according to key(s) contained in each record. More than one merge operation can be performed during one execution of the COBOL program. Special processing of output records can also be specified.

There are special considerations in the Environment Division, the Data Division, and the Procedure Division for the Merge Facility.

ENVIRONMENT DIVISION

Each input file and the resulting merged output file must be described in a separate File-Control entry, and each must be a standard sequential file, or a VSAM file with sequential access. The merge file must have a separate File-Control entry, as described in the following paragraphs.

File-Control Entry for Merge Files

The File-Control entry names the merge file and associates it with a storage medium.

```
┌──────────────────────────────────────────────────────────────────┐
│                      General Format                              │
├──────────────────────────────────────────────────────────────────┤
│                                                                  │
│    {SELECT file-name                                             │
│                                                                  │
│     ASSIGN TO system-name-1 [system-name-2] ... .}...            │
│                                                                  │
└──────────────────────────────────────────────────────────────────┘
```

Each File-Control entry for a merge file must begin with a SELECT clause, and be immediately followed by an ASSIGN clause. There may be no other clauses.

SELECT Clause: The SELECT clause names each merge file in the program. Each file described by an SD entry in the Data Division must be named once and only once as a file-name following the key word SELECT.

ASSIGN Clause: The ASSIGN clause is required. System-name can serve as documentation to describe the work units. However, since the system obtains this information at execution time, the compiler treates the ASSIGN clause as documentation.

If an ASCII-collated merge is to be performed, C must be specified in the organization field. (See "Appendix E: ASCII Considerations.")

I-O-Control Paragraph

The optional I-O-Control Paragraph specifies the storage area to be shared by different files.

SAME SORT-MERGE AREA Clause (OS/VS)

General Format

```
SAME { SORT       } AREA FOR
     { SORT-MERGE }
     { RECORD     }

    file-name-1 [file-name-2] ... .
```

When the SAME SORT AREA or SORT-MERGE AREA clause is specified, at least one file-name specified must name a sort or merge file. Files that are not sort or merge files may also be specified. The following rules apply:

- More than one SAME SORT AREA or SORT-MERGE AREA clause may be specified; however, a sort or merge file must not be named in more than one such clause.
- If a file that is not a sort or merge file is named in both a SAME AREA clause and in one or more SAME SORT AREA or SORT-MERGE AREA clauses, all of the files in the SAME AREA clause must also appear in all of the SAME SORT AREA or SORT-MERGE AREA clauses.
- Files named in a SAME SORT AREA or SORT-MERGE AREA clause need not have the same organization or access.
- Files named in a SAME SORT AREA or SORT-MERGE AREA clause that are not sort or merge files do not share storage with each other unless the user names them in a SAME AREA or SAME RECORD AREA clause.

The SAME SORT AREA or SORT-MERGE AREA clause specifies one storage area available for merge operations by each named merge file. That is, the storage area allocated for one merge operation is available for reuse in another merge operation.

The function of the SAME SORT AREA or SORT-MERGE AREA clause is to optimize the assignment of storage areas to a given MERGE statement. The system handles storage assignment automatically; hence, the SORT AREA or SORT-MERGE AREA options, if specified, are treated as documentation.

When the SAME RECORD AREA option is specified, the named files, including any sort or merge files, share only the area in which the current logical record is processed. Several of the files may be open at the same time, but the logical record of only one of these files can exist in the record area at one time.

DATA DIVISION

In the Data Division, the user must include file description entries for each merge input file and for the merged output file, merge-file description entries for each merge file, and record description entries for each.

Merge-File Description Entry

A merge-file description entry must appear in the File Section for each merge file named in a File-Control entry.

General Format

```
SD  merge-file-name

    [RECORD CONTAINS [integer-1 TO] integer-2 CHARACTERS]

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

The level indicator SD identifies the beginning of the merge-file description, and must precede the merge-file-name.

The clauses following merge-file-name are optional, and their order of appearance is not significant.

One or more record description entries must follow the merge-file description entry, but no input/output statements may be executed for the merge file.

Merge-File-Name: The merge-file-name must be the same as that specified in the merge file File-Control entry. It is also the name specified as the first operand in the MERGE statement.

RECORD CONTAINS Clause: The size of each data record is completely defined in the record description entry; therefore, this clause is never required. When it is specified, the same considerations apply as in its Data Division description.

DATA RECORDS Clause: This clause names the 01-level data records associated with this SD entry. This clause is never required, and the compiler treats it as documentation. When it is specified, the same considerations apply as in its Data Division description.

PROCEDURE DIVISION

The Procedure Division contains a MERGE statement describing the merge operation, and optional output procedures. The procedure-names of the output procedures are specified within the MERGE statement. More than one MERGE statement can be specified, appearing anywhere except in the declaratives section or in an input or output procedure for a SORT or MERGE statement.

MERGE Statement

The MERGE statement combines two or more identically sequenced files using specified key(s), and makes records available to an output file in merged order.

Format

```
MERGE  file-name-1

     ON { ASCENDING  } KEY  data-name-1 [data-name-2] ...
        { DESCENDING }

    [ON { ASCENDING  } KEY data-name-3 [data-name-4] ... ] ...
        { DESCENDING }

     USING file-name-2 file-name-3 [file-name-4] ...

     { GIVING file-name-5                                          }
     { OUTPUT PROCEDURE IS section-name-1 [THRU section-name-2]    }
```

No file-name specified in the MERGE statement may be open at the time the statement is executed. The files are automatically opened and closed by the merge operation; all implicit functions are performed, such as execution of system procedures or any associated declarative procedures.

No file-name may be specified more than once in one MERGE statement.

Only one file-name from a multiple file reel may appear in one MERGE statement.

FILE-NAME-1: This file-name represents the merge file, and must be described in an SD entry in the Data Division.

ASCENDING/DESCENDING KEY Option: These options specify whether records are to be merged in ascending or descending sequence, based on one or more merge keys.

Each data-name represents a KEY data item, and must be described in the record description(s) associated with the SD entry for file-name-1, the merge work file. The following rules apply:

- if file-name-1 has more than one associated record description entry, the KEY data items need be described in only one such record description
- each data-name may be qualified; it may not be subscripted or indexed (that is, it may not contain or be contained in an entry that contains an OCCURS clause)
- KEY data items must be at a fixed displacement from the beginning of the record (that is, no KEY data item may follow an OCCURS DEPENDING ON clause in the record description)
- a maximum of 12 keys may be specified; the total length of all keys must not exceed 4092 bytes
- all key fields must be located within the first 4092 bytes of the logical record

The KEY data items are listed in order of decreasing significance, no matter how they are divided into KEY phrases. Using the format as an example, data-name-1 is the most significant key, and records are merged in ascending or descending order on that key; data-name-2 is the next most significant key; within data-name-1, records are merged on

data-name-2 in ascending or descending order. Within data-name-2, records are merged on data-name-3 in ascending or descending order; within data-name-3, records are merged on data-name-4 in ascending or descending order, etc.

When ASCENDING is specified, the merged sequence is from the lowest to the highest value of the contents in the KEY data item according to the collating sequence used.

When DESCENDING is specified, the merged sequence is from the highest to the lowest value of the contents in the KEY data item according to the collating sequence used.

Figure V gives the collating sequence used for each category of KEY data item.

KEY Category	Collating Sequence
Alphabetic Alphanumeric Alphanumeric Edited Numeric Edited	EBCDIC (non-algebraic and unsigned)
Numeric	Algebraic (signed)

Figure V. KEY Item Categories and Collating Sequences

The rules for comparison are those for the relation condition (see "Relation Condition" in the Conditions chapter of the Procedure Division). If two or more KEY data items test as equal, the merge operation makes the records available in the order that the input file-names are specified in the USING option.

USING Option: All file-names listed in the USING option represent identically ordered input files that are to be merged. Two through eight file-names may be specified.

GIVING Option: File-name-5 is the name of the merged output file. When this option is specified, all merged records made available from the merge operation are automatically written on the output file.

OUTPUT PROCEDURE Option: When this option is specified, all output records from the merge operation are made available to the user (through a RETURN statement) for further processing.

When an output procedure is specified, control passes to the procedure during execution of the MERGE statement. Before entering the output procedure, the merge operation reaches a point at which it can provide the next merged record when requested. The RETURN statement in the output procedure is a request for the next merged record. (See the RETURN statement description in the Sort Feature chapter.) An output procedure must contain at least one RETURN statement to make merged records available for further processing.

Control may be passed to an output procedure only when a related MERGE statement is being executed.

The output procedure must not form part of any other procedure.

If section-name-1 alone is specified, the output procedure must consist of one contiguous Procedure Division section.

If section-name-1 THRU section-name-2 is specified, the output procedure consists of two or more contiguous Procedure Division sections; section-name-1 specifies the first such section; section-name-2 specifies the last such section.

Control must not be passed to the output procedure unless a related MERGE or SORT statement is being executed, because RETURN statements in the output procedure have no meaning unless they are controlled by a MERGE or SORT statement. The output procedure may consist of the processing requests necessary to select, modify, or copy the records being made available, one at a time, from the merge operation. The following restrictions apply:

- There may be no explicit transfers of control outside the output procedure. ALTER, GO TO, and PERFORM statements within the procedure must not refer to procedure-names outside the output procedure. However, an implicit transfer of control to a declarative procedure is allowed.
- No SORT or MERGE statements are allowed.
- The remainder of the Procedure Division must not transfer control to points inside the output procedure; that is, ALTER, GO TO, and PERFORM statements in the remainder of the Procedure Division must not specify procedure-names within an output procedure.

The compiler inserts an end-of-processing transfer at the end of the last output procedure section. When end-of-processing is recognized, the merge operation is terminated, and control is transferred to the next statement following the MERGE statement.

SEGMENTATION RESTRICTIONS: The MERGE statement may be specified in a segmented program. However, the following restrictions apply:

- If the MERGE statement appears in the fixed portion, then any associated output procedure must be:
  - completely within the fixed portion, or
  - completely within one independent segment
- If the MERGE statement appears in an independent segment, then any associated output procedure must be:
  - completely within the fixed portion, or
  - completely within the same independent segment as the MERGE statement

3886 OCR PROCESSING

The IBM 3886 OCR (Optical Character Reader) Model 1 is a general purpose online unit record device that satisfies a broad range of data entry requirements. The 3886 OCR can significantly reduce time and cost factors, by eliminating input steps in both new and existing applications; a keying process is no longer necessary, since the 3886 OCR can read and recognize data created by numeric hand printing, high-speed computer printing, typewriters, and preprinted forms.

The IBM 3886 OCR uses several new technologies which make it a compact, highly reliable, modular device. A powerful microprogrammed recognition and control processor performs all machine control and character recognition functions, and enables the 3886 OCR to perform sophisticated data and blank editing.

The 3886 OCR accepts documents from 3 x 3 to 9 x 12 inches in size. Under program control, it can read documents line-by-line, transmitting their contents line-by-line to the CPU. Additional facilities, all under program control, include: document marking, line marking, document ejecting (with stacker selection), and line reading (of current line).

OS/VS COBOL support for the 3886 OCR is through an object-time subroutine in the COBOL library, invoked through COBOL CALL statements. By means of parameters passed to the subroutine, the following operations are provided: open and close the file, read a line, wait for read completion, mark a line, mark the current document, eject the current document, and load a format record. After each operation, a status indicator is passed back to the COBOL program, so that any exceptional condition can be tested.

Through a fixed format OCR file information area in the Working-Storage or Linkage Section, the COBOL user defines storage for the OCR parameters. Of these parameters, the COBOL programmer is responsible for providing a file identifier, a format record identifier, an operation code, and (depending on the operation) a line number, line format number, mark code, and stacker number. After completion of each operation a status indicator is returned; after completion of a read operation, header and data records are also returned.

OS/VS provides two macro instructions for defining documents. The DFR macro instruction defines attributes common to a group of line types. The DLINT macro instruction defines specific attributes of an individual line type. The DFR and associated DLINT macro instructions are used in one assenbly to build a format record module. The format record must be link-edited into a system library so that it can be loaded into the 3886 OCR when the file is to be processed. The format record indicates the line types to be read, attributes of the fields in the lines, and the format of the data records to be processed.

Additional information on the IBM 3886 OCR can be found in the following publications:

IBM 3886 Optical Character Reader

General Information Manual, Order No. GA21-9146

Input Document Design Guide and Specifications, Order No. GA21-9148

OS/VS Program Planning Guide for IBM 3886 Optical Character Reader Model 1, Order No. GC21-5069

FIPS FLAGGER

The FIPS (Federal Information Processing Standard) is a compatible subset of full American National Standard COBOL, X3.23-1968. The FIPS itself is subdivided into four levels: low, low-intermediate, high-intermediate, and full. Any program written to conform to the FIPS must conform to one of those levels of FIPS processing. Processing modules included in full American National Standard COBOL, and those included in the four level of the FIPS are shown in Figure VI.

American National Standard COBOL Processing Modules	Full FIPS Processing Modules	High-intermediate FIPS Processing Modules	Low-intermediate FIPS Processing Modules	Low FIPS Processing Modules
2NUC 1,2 (Nucleus)	2NUC 1,2	2NUC 1,2	2NUC 1,2	1NUC 1,2
3TBL 1,3 (Table Handling)	3TBL 1,3	2TBL 1,3	2TBL 1,3	1TBL 1,3
2SEQ 1,2 (Sequential Access)	2SEQ 1,2	2SEQ 1,2	2SEQ 1,2	1SEQ 1,2
2RAC 0,2 (Random Access)	2RAC 0,2	2RAC 0,2	2RAC 0,2	
2SRT 0,2 (Sort)	2SRT 0,2	1SRT 0,2		
2RPW 0,2 (Report Writer)				
2SEG 0,2 (Segment-ation)	2SEG 0,2	1SEG 0,2	1SEG 0,2	
2LIB 0,2 (Library)	2LIB 0,2	1LIB 0,2	1LIB 0,2	

Figure VI. The Four Levels of FIPS Processing

The FIPS Flagger identifies source clauses and statements that do not conform to the Federal standard. Four levels of flagging, to conform to the four levels of the FIPS, are provided. The following lists identify COBOL source elements flagged for each level.

FULL FIPS FLAGGING: When flagging for the full FIPS level is specified, the following elements of the COBOL source, if specified, are identified.

GLOBAL ITEMS

Single quote instead of double
Floating Point Literals

Special Register LINE-COUNTER
Special Register PAGE-COUNTER
Special Register CURRENT-DATE
Special Register TIME-OF-DAY
Special Register RETURN-CODE
Special Register SORT-RETURN
Special Register SORT-FILE-SIZE
Special Register SORT-CORE-SIZE
Special Register SORT-MODE-SIZE
Special Register SORT-MESSAGE
Special Register LABEL-RETURN
Special Register WHEN-COMPILED

Comment Lines with * in Column 7
The SUPPRESS option of the COPY statement

IDENTIFICATION DIVISION Items

ID abbreviation for IDENTIFICATION
Accepting Identification Division Paragraphs in any order
Accepting Program Name in quotes

ENVIRONMENT DIVISION Items

Optional CONFIGURATION SECTION and Paragraphs
S01 and S02 Function-names in SPECIAL-NAMES paragraph

Allowing any order for optional SELECT clauses
W, R, or I as Organization indicator in System-name
Optional omission of IS in ACCESS MODE IS Clause
Optional omission of IS in ACTUAL KEY IS Clause
ACTUAL-KEY clause for sequential access of a direct file
ACTUAL-KEY clause for sequential creation of a direct File
NOMINAL KEY Clause in FILE-CONTROL Paragraph
RECORD KEY Clause in FILE-CONTROL Paragraph
TRACK-AREA Clause in FILE-CONTROL Paragraph
The COPY statement in the FILE-CONTROL paragraph

Short form of RERUN ON Clause
Interchangeable use of REEL and UNIT in RERUN ON Clause
APPLY Clause in I-O-CONTROL paragraph
Allowing I-O-CONTROL paragraph clauses in any order

RESERVE integer AREAS clause (as distinguished from the RESERVE
ALTERNATE AREAS clause)
ORGANIZATION clause
ACCESS MODE DYNAMIC clause
PASSWORD clause
FILE STATUS clause

DATA DIVISION Items

REPORT SECTION of DATA DIVISION
RD level indicator
The DATA RECORDS clause for a REPORT FD
LINKAGE SECTION of DATA DIVISION
COMMUNICATION SECTION
CD level indicator

Allowing unequal level numbers to belong to the same group
RECORDING MODE Clause of FD entry.
REPORT Clause of FD Entry
LABEL RECORDS CLAUSE on Sort File Description
Optional BLOCK CONTAINS for DIRECT Files when RECORDING MODE IS S
Integer-2 of zero for BLOCK CONTAINS clause
Integer-2 of zero for RECORD CONTAINS clause
TOTALING and TOTALED AREA option of the LABEL RECORDS clause
Accepting name of preceding entry when using multiple redefinitcn
External Floating-point picture
The SIGN Clause
Allowing the SYNCHRONIZED Clause at the 01 level
COMPUTATIONAL-1 option of the USAGE Clause
COMPUTATIONAL-2 option of the USAGE Clause
COMPUTATIONAL-3 option of the USAGE Clause
COMPUTATIONAL-4 option of the USAGE Clause
Nested OCCURS DEPENDING ON clauses
Allowing SYNCHRONIZED with USAGE IS INDEX
The COPY statement in the Working-Storage Section
DISPLAY-ST option of the USAGE Clause and associated PICTURE
Use of VALUE Clause as Comments in File Section for other than
Condition-name entries
COPY REDEFINES in Working-Storage Section

PROCEDURE DIVISION Items

USING clause on PROCEDURE DIVISION
THEN used to separate statements
Allowing omission of section header at beginning of Procedure Division
The START statement
The REWRITE statement
The TRANSFORM statement

The GENERATE statement
The INITIATE statement
The TERMINATE statement

The DEBUG statement
The READY TRACE statement
The RESET TRACE statement
The ON statement
The EXHIBIT statement
The CALL statement
The ENTRY statement
The GOBACK statement
The EXIT PROGRAM statement

The USE AFTER STANDARD EXCEPTION sentence
The READ NEXT statement
The DELETE statement
The EXTEND option for the OPEN statement and Error Procedures
The SERVICE RELOAD statement

The unary plus operator
Allowing omission of the space following the unary operator
OTHERWISE in IF statements
The message condition
The GO TO MORE-LABELS statement – p. 171
GIVING option of USE sentence
USE BEFORE REPORTING sentence
Allowing REVERSED option of OPEN with multiple reel files
LEAVE, REREAD, and DISP options of the OPEN statement
Allowing omission of the AT END option for READ statements
Allowing omission of the INVALID KEY option for READ and WRITE statements
The AT END-OF-PAGE or EOP option of the WRITE statement
The WRITE AFTER POSITIONING statement
The FROM SYSIN or CONSOLE option of the ACCEPT statement
The DATE/DAY/TIME format of the ACCEPT statement
The UPON CONSOLE, SYSPUNCH, or SYSOUT option of the DISPLAY statement
The WITH DISP or POSITIONING option of the CLOSE statement

The BASIS statement
The INSERT statement
The DELETE statement
The RECEIVE Statement
The SEND Statement
The STRING Statement
The UNSTRING Statement
The CANCEL Statement

The EJECT statement
The SKIP1 statement
The SKIP2 statement
The SKIP3 statement

HIGH-INTERMEDIATE FIPS FLAGGING: When flagging for the high-intermediate FIPS level is specified, all elements included in the preceding list are flagged, plus the following additional COBOL source elements:

GLOBAL ITEMS

The REPLACING option of the COPY statement

ENVIRONMENT DIVISION

SEGMENT-LIMIT clause in OBJECT-COMPUTER paragraph
SORT option of SAME Clause

DATA DIVISION

The ASCENDING and DESCENDING KEY option of the OCCURS clause
The DEPENDING ON option of the OCCURS clause

PROCEDURE DIVISION

All sections with the same priority number must be together
All segments with priority numbers 1-49 must be together
The SEARCH statement
More than one SORT statement
The FROM option of the RELEASE statement
The INTO option of the RETURN statement

LOW-INTERMEDIATE FIPS FLAGGING: When flagging for the low-intermediate FIPS level is specified, all elements included in the preceding lists are flagged, plus the following additional COBOL source elements:

ENVIRONMENT DIVISION

The OR option of the SELECT sentence

DATA DIVISION

SD level indicator

PROCEDURE DIVISION

One or more SORT statements
Only one STOP RUN statement in the non-declarative portion
The RETURN statement
The RELEASE statement

LOW FIPS FLAGGING: When flagging for the low FIPS level is specified, all elements included in the preceding lists are flagged, plus the following additional COBOL source elements:

GLOBAL ITEMS

The COPY statement
Comma and semicolon as punctuation
Data-names which begin with non-alphabetic character
Continuation of words and numeric literals

Figurative constant ZEROES
Figurative constant ZEROS
Figurative constant SPACES
Figurative constant HIGH-VALUES
Figurative constant LOW-VALUES
Figurative constant QUOTES
Figurative constant ALL literal

IDENTIFICATION DIVISION

DATE-COMPILED Paragraph

ENVIRONMENT DIVISION

RESERVE ALTERNATE AREAS Clause in File-Control Paragraph (SELECT sentence)
OPTIONAL in SELECT Clause
ACTUAL KEY Clause in File-Control Paragraph
FILE-LIMITS ARE Clause
Data-name instead of literal in FILE-LIMIT IS clause
Multiple extents in FILE-LIMIT IS clause
RANDOM option in ACCESS MODE IS Clause

RECORD and file-name option of SAME Clause
MULTIPLE FILE TAPE Clause in I-O-CONTROL Paragraph

DATA DIVISION

Level numbers 11 - 49
Level numbers 1-9 (1 digit)
Level number 66 RENAMES clause
Level number 88 Condition Name
Nesting of REDEFINES Clause
VALUE Clause as Condition-name entry
Integer-1 TO option of BLOCK CONTAINS (RECORD or CHARACTER) Clause
Data-name option on LABEL RECORDS Clause
Data-name option of VALUE OF Clause
Multiple Index-names for OCCURS clause

PROCEDURE DIVISION

+, -, *, /, and **
>, <, and = in relationals
Connectives OF, IN, ', ', AND, OR, and NOT
DECLARATIVES, END DECLARATIVES and USE sentence
Qualification of names
Priority number on Section header
The COMPUTE verb
The SEEK Statement
The Sign condition (POSITIVE, NEGATIVE, or ZERO)
Condition-name condition
Compound conditions
Nested IF statements
CORRESPONDING option (ADD, SUBTRACT, and MOVE)
Multiple results of ADD and SUBTRACT statements
REMAINDER option of DIVIDE statement
GO TO without object (used with ALTER)
Multiple operands of ALTER statement
UNTIL Condition and VARYING form of PERFORM
REVERSED and NO REWIND options of OPEN statement
Multiple file-names in OPEN statement
INTO option of READ statement
INVALID KEY option of READ statement
FROM option of WRITE statement
ADVANCING identifier LINES/mnemonic/name form of WRITE
The FROM option of the ACCEPT statement
The UPON option of the DISPLAY statement
The WITH NO REWIND or LOCK option of the CLOSE statement
Multiple file-names in a CLOSE statement
Three levels of subscripting
Multiple Index-names/identifier in SET statement
The UP BY and DOWN BY option of the SET statement

MISCELLANEOUS PROCESSING CONSIDERATIONS

The following items, concerning the Environment Division ASSIGN clause, and the Procedure Division WRITE ADVANCING and SORT statements, apply only to OS/VS COBOL.

ASSIGN Clause

In the ASSIGN clause system-name, the class and device fields are treated as documentation. At execution time, any valid device can be associated with the file through the DD statement. See the documentation for the Version 4 ASSIGN clause for further considerations.

WRITE ADVANCING Statement

A compile-time option allows the user to specify WRITE BEFORE/AFTER ADVANCING without reserving the first character in the output record as the control character.

SORT Statement

The SORT statement implementation has been enhanced to allow up to eight input files when the USING option is specified.

The input files, as well as the output file, can be standard sequential files, or sequentially accessed VSAM files.

FIGURES

CHARTS

TABLES

FEATURES OF THE PROGRAM PRODUCT COMPILERS

OS/VS COBOL

The Program Product OS/VS COBOL Compiler and Library includes the following features:

VSAM (Virtual Storage Access Method) Support -- which provides fast storage and retrieval of records, password protection, centralized and simplified data and space management, advanced error recovery facilities, plus system and user catalogs. COBOL supports indexed (key-sequenced) files and sequential (entry-sequenced) files. Records can be fixed or variable in length.

Merge Support -- The MERGE verb enables two or more identically-sequenced input files to be combined into a single output file by specifying a set of keys. Both standard sequential and sequentially accessed VSAM files can be designated as input or output.

Lister Option -- provides specially formatted listing with embedded cross references for increased intelligibility and ease of use. Reformatted source deck is available as an option.

Verb Profiles -- facilitates identifying and locating verbs in the COBOL source program. Options provide verb summary or verb cross reference listing which includes verb summary.

Execution Time Statistics -- maintains a count of the number of times each verb in the COBOL source program is executed during an individual program execution.

FIPS (Federal Information Processing Standard) Flagger -- which issues messages identifying nonstandard elements in a COBOL source program. The FIPS Flagger makes it possible to ensure that COBOL clauses and statements in an OS/VS COBOL source program conform to the Federal Information Processing Standard.

WHEN-COMPILED Special Register -- which is a programmer aid that provides a means of associating a compilation listing with both the object program and the output produced at execution time.

System/370 Device Support -- any valid OS/VS device can be used with an OS/VS COBOL program. In most cases, support is transparent to the OS/VS COBOL program. There are special considerations for the following devices:

> 3886 OCR (Optical Character Reader) -- this device reads multiline alphanumeric or numeric machine-printed documents or numeric hand-printed documents, with stacker selection. OS/VS COBOL support is through an object-time library subroutine.
>
> 3330, 3340 Disk Facilities -- these devices are high-speed large-capacity disks, with the RPS (rotational positional sensing) feature. Use of the fixed block standard option, which can be specified at object time, results in much improved performance.
>
> Multifunction Card Devices -- OS/VS COBOL supports the combined function processing available through these devices.

Combined functions available are: read/punch, read/print, punch/print, and read/punch/print.

All features of OS Full American National Standard COBOL Version 4 continue to be supported by IBM OS/VS COBOL. (See the following section.) IBM OS/VS COBOL is packaged as a single Program Product, Program Number 5740-CB1; the Library is also available as a separate Program Product, Program Number 5740-LM1.

OS FULL AMERICAN NATIONAL COBOL VERSION 4

The Version 4 Compiler and Library is a Program Product that contains all of the features of the Version 2 and Version 3 Compilers, and also contains a number of added features, as well as improved functions.

The Version 4 Compiler contains the following features:

Advanced Symbolic Debugging provides faster and easier debugging for the COBOL programmer. At abnormal termination a formatted dump, using COBOL source data-names, is produced. Execution-time dynamic dumps at user-specified points in the Procedure Division can also be obtained. When the symbolic debugging feature is requested, optimized object code is automatically provided. (Appendix F gives an example of Symbolic Debugging output.)

Optimized Object Code can be requested, resulting in considerably smaller object programs than are produced without optimization. For COBOL programs that are not I/O bound, execution time is reduced.

COBOL Teleprocessing (TP) programs can now be written, using IBM extensions to American National Standard COBOL. Such programs are device-independent, and can be created more easily than Assembler TP programs. The source language for such programs is a subset of the CODASYL specifications for COBOL TP language.

COBOL Library Management Facility allows installations running with multiple COBOL regions/partitions to save considerable main storage by sharing some or all of the COBOL library subroutine modules.

Dynamic Subprogram Linkage gives the user object-time control of main storage. At object time, COBOL subprograms can be loaded under program control; when such a subprogram is no longer needed, the calling program can free the storage it occupies for other use.

Syntax-checking Compilation can be requested to save machine time and money while debugging source syntax errors. When unconditional syntax checking is requested, the source program is scanned for syntax errors and such error messages are generated, but no object code is produced. When conditional syntax checking is requested, a full compilation is produced if no messages or only W-level or C-level messages are generated; if one or more E-level or D-level messages are generated, no object code is produced. Selected test cases have shown that when object code is not generated, compilation time may be reduced by as much as 70%.

String Manipulation, providing for more flexible data manipulation, can now be specified in COBOL. Contiguous data can be separated into multiple logical subfields; two or more separate subfields can be concatenated into a single field.

All of the features of OS Full American National Standard COBOL Version 3 continue to be supported by Version 4. The Version 4 Compiler and Library is packaged as a single Program Product (Program Number 5734-CB2); the Library is also available as a separate Program Product (Program Number 5734-LM2).

OS FULL AMERICAN NATIONAL STANDARD COBOL VERSION 3

The Version 3 Compiler and Library is a Program Product that contains a number of improvements in function and performance over Version 2. The compiler may be used with or without the Time Sharing Option (TSO) of the IBM Operating System.

FEATURES DEPENDENT ON TSO

With TSO, the terminal user may choose options to determine the characteristics of compiler output to the terminal. He may direct to the terminal:

- Compilation progress and diagnostic messages.
- The compiler's entire listing data set.

The user may suppress either category or may suppress all output to the terminal.

In addition, if the user has recorded line numbers in the input data set, the compiler may be instructed to substitute these numbers for internal statement numbers in any diagnostic messages printed on the terminal. Also, when diagnostic messages are printed on the terminal, a message stating the total number of statements in error can be included at the request of the user.

FEATURES NOT DEPENDENT ON TSO

With or without TSO, programmers can use the following features:

Optional alphabetically ordered cross-reference listings. Significant performance improvement has been made to the current cross-reference option which preserves source statement order.

A flow trace option, which prints a formatted trace of the last procedures executed before an abnormal termination of execution. The number of procedures to be traced is specified by the user.

A statement number option, which provides the user with the number of the COBOL statement, and of the verb within the statement, being executed when an abnormal termination of execution occurs.

Expansion of the functions of the CLIST and DMAP compiler options. In addition to the condensed listing (CLIST) and the glossary (DMAP), global tables, literal pools, and register assignments are included.

The ability to batch compile more than one program or subprogram with a single invocation of the compiler, resulting in a reduction in compilation time.

A separately signed numeric data type. The programmer can use the SIGN clause to specify the position and the mode of representation of the operation sign of numeric data items.

The ability to specify record size at object time for an input QSAM or QISAM data set.

Generic key for Indexed Files. The programmer can request a record from an ISAM data set by using a search key which is comprised of a user-specified number of the high-order characters of the key. The user does not have to specify the full or exact search key.

A checkpoint-rerun provision at end-of-volume for sequentially accessed files with any file organization.

Expansion of the ON statement to permit use of identifiers as well as literals as the count-conditional operands.

Enhancement of error declaratives so that a GIVING option can bereferenced when the declarative specifies a file-name list or the INPUT, OUTPUT, or I-O options.

Installation default options separately located from other coding to improve maintainability and serviceability.

Implementation of ASCII, the American National Standard Code for Information Interchange, X3.4-1968, which provides the user with the capability at object time of accepting and creating magnetic tape files written in this code.

Support for American National Standard Magnetic Tape Labels for Information Interchange, X3.27-1969. These labels may be used only with tape files written in the American National Standard Code for Information Interchange.

The ability to sort files using the ASCII collating sequence for character data on a per sort basis.

System/370 Support can be requested, to take advantage of the System/370 instruction set. When such support is specified, certain System/370 instructions are generated to replace the equivalent object-time subroutines and instructions needed when running under System/360.

OPEN Statement improvement -- generated code for the OPEN statement has been modified to give substantial savings in object program space.

Additional Device Support -- The following mass-storage devices are now supported: 2305-1, 2305-2, 2319, 3330. The 3211 printer is also supported.

In 1959, a group of computer professionals, representing the U.S. Government, manufacturers, universities, and users, formed the Conference On DAta SYstems Language (CODASYL). At the first meeting, the conference agreed upon the development of a common language for the programming of commercial problems. The proposed language would be capable of continuous change and development, it would be problem-oriented and machine-independent, and it would use a syntax closely resembling English, avoiding the use of special symbols as much as possible. The COmmon Business Oriented Language (COBOL) which resulted met most of these requirements.

As its name implies, COBOL is especially efficient in the processing of business problems. Such problems involve relatively little algebraic or logical processing; instead, they usually manipulate large files of similar records in a relatively simple way. Thus, COBOL emphasizes the description and handling of data items and input/output records.

In the years since 1959, COBOL has undergone considerable refinement and standardization. Now, an extensive subset for a standard COBOL has been approved by ANSI (the American National Standards Institute), an industry-wide association of computer manufacturers and users; this standard is called American National Standard COBOL (formerly known as USA Standard COBOL).

This publication explains IBM OS Full American National Standard COBOL, which is compatible with the highest level of American National Standard COBOL and includes a number of IBM extensions to it as well. The compiler supports the processing modules defined in the standard. These processing modules include:

NUCLEUS -- which defines the permissible character set and the basic elements of the language contained in each of the four COBOL divisions: Identification Division, Environment Division, Data Division, and Procedure Division.

TABLE HANDLING -- which allows the definition of tables and making reference to them through subscripts and indexes. A convenient method for searching a table is provided.

SEQUENTIAL ACCESS -- which allows the records of a file to be read or written in a serial manner. The order of reference is implicitly determined by the position of the logical record in the file.

RANDOM ACCESS -- which allows the records of a file to be read or written in a manner specified by the programmer. Programmer-specified keys control successive references to the file.

SORT -- which provides the capability of sorting files in ascending and/or descending order. This feature also includes procedures for handling such files both before and after they have been sorted.

REPORT WRITER -- which allows the programmer to describe the format of a report in the DATA DIVISION, thereby minimizing the amount of PROCEDURE DIVISION coding necessary.

SEGMENTATION -- which allows large problem programs to be split into segments to be designated as permanent or overlayable core storage. This assures more efficient use of core storage at object time.

LIBRARY -- which supports the retrieval and updating of pre-written source program entries from a user's library, for inclusion in a COBOL

program at compile time. The effect of the compilation of library text is as though the text were actually part of the source program.

In this publication, the features included in the NUCLEUS, SEQUENTIAL ACCESS, and RANDOM ACCESS modules are presented as part of the discussion of "Language Considerations" and of the four divisions of a COBOL program. The other five modules -- TABLE HANDLING, SORT, REPORT WRITER, LIBRARY, and SEGMENTATION -- are presented as separate features of American National Standard COBOL.

This manual describes all versions of IBM OS American National Standard COBOL. All information relating to the Program Product Version 3 and Version 4 compilers is presented within separate paragraphs. Such paragraphs begin with the heading "Program Product Information", followed by the Version numbers and all following paragraphs pertaining to such information are indented. All information relating only to the OS/VS COBOL Compiler and Library Program Product is included in the separate chapter, "OS/VS COBOL Considerations."

This chapter gives the reader a general understanding of the principles of IBM OS Full American National Standard COBOL (hereinafter simply termed "COBOL"). It introduces the reader to COBOL and demonstrates some of the ways in which the language can be used in the solution of commercial problems. This discussion does not define the rules for using COBOL, but rather attempts to explain the basic concepts of the language through relatively simple examples.

The reader who has an understanding of the principles of currently implemented versions of COBOL may wish to go directly to "Language Considerations." Other readers will find many concepts discussed in this chapter of help in using the detailed instructions throughout the rest of this manual.

PRINCIPLES OF COBOL

COBOL is one of a group of high-level computer languages. Such languages are problem oriented and relatively machine independent, freeing the programmer from many of the machine oriented restrictions of assembler language, and allowing him to concentrate instead upon the logical aspects of his problem.

COBOL looks and reads much like ordinary business English. The programmer can use English words and conventional arithmetic symbols to direct and control the complicated operations of the computer. The following are typical COBOL sentences:

```
ADD DIVIDENDS TO INCOME.
MULTIPLY UNIT-PRICE BY STOCK-ON-HAND
     GIVING STOCK-VALUE.
IF STOCK-ON-HAND IS LESS THAN ORDER-POINT
      MOVE ITEM-CODE TO REORDER-CODE.
```

Such COBOL sentences are easily understandable, but they must be translated into machine language -- the internal instruction codes -- before they can actually be used.

A special systems program, known as a compiler, is first entered into the computer. The COBOL program (referred to as the source program) is then entered into the machine, where the compiler reads it and analyzes it. The COBOL language contains a basic set of reserved words and symbols. Each combination of reserved words and symbols is transformed by the compiler into a definite set of usable machine instructions. In effect, the programmer has at his disposal a whole series of "prefabricated" portions of the machine-language program he wishes the compiler to construct.

When he writes a COBOL program, he is actually directing the compiler to bring together, in the proper sequence, the groups of machine instructions necessary to accomplish the desired result. From the programmer's instructions, the compiler creates a new program in machine language. This program is known as an object program.

Once the object program has been produced, it may be used at once, or it may be recorded on some external medium and stored for future use. When it is needed, it can then be called upon again and again to process data.

Every COBOL program is processed first when the compiler translates the COBOL program into machine language (compile time), then when the machine language program actually processes the data (execution time).

A simple example illustrates the basic principles of translating a COBOL sentence. To increase the value of an item named INCOME by the value of an item named DIVIDENDS, the COBOL programmer writes the following sentence:

```
ADD DIVIDENDS TO INCOME.
```

Before the compiler can interpret this sentence, it must be given certain information. The programmer describes the data represented by the names DIVIDENDS and INCOME in such a way that the compiler can recognize it, obtain it when needed, and treat it in accordance with its special characteristics.

First, the compiler examines the word ADD. It determines whether or not ADD is one of the COBOL reserved words, that is, words that have clearly defined meanings in COBOL (rather than a word like DIVIDENDS, which is defined by the programmer). ADD is a special kind of reserved word -- a COBOL key word. Therefore, the compiler generates the machine instructions necessary to perform an addition and inserts them into the object program.

The compiler next examines the word DIVIDENDS. Because the programmer has supplied data information about DIVIDENDS, the compiler knows where and how DIVIDENDS information is to be placed in core storage, and it inserts into the object program the instructions needed in order to locate and obtain the data.

When the compiler encounters the word TO, it again determines whether or not this is a COBOL reserved word. It is such a word, and the compiler interprets it to mean that the value represented by the name following the word TO, in this case INCOME, must be increased as a result of the addition.

The compiler next examines the word INCOME. Again, it has access to data information about the word. As a result, it is able to place in the object program the instructions necessary to locate and use INCOME data.

The programmer placed a period after the word INCOME. The effect of the period on the COBOL compiler is similar to its effect in the English language. The period tells the compiler that it has reached the last word to which the verb ADD applies, the end of the sentence.

The logical steps we have described are performed by the compiler in creating the object program, although they might not be performed in exactly this sequence. All these preparatory steps are required only in creating the object program. Once created, the object program is used for the actual processing and may be saved for future reference. The source program is not required further, unless the programmer makes a change in it; in that case, it must be compiled again to create a new object program.

When the machine-language instruction for ADD is actually performed at execution time, the instruction is executed in either of two ways, depending on the format of the data:

1. It directly adds the value of DIVIDENDS to the value of the data representing INCOME, thus giving the new value of INCOME.

   or

2. It moves the data representing INCOME into a special work area or register; then DIVIDENDS is added to it to create the sum, after which the new value of INCOME is returned to the proper area in storage.

In this simple example, the object program could add the two specified items with very few machine instructions. In actual practice, however, some complex COBOL sentences produce dozens of machine instructions. Then, too, a computer can be instructed to repeat a procedure any number of times. A few COBOL sentences can start the computer on operations that could process millions of data records rapidly and accurately.

A SAMPLE COBOL PROGRAM

COBOL is based on English; it uses English words and certain syntax rules derived from English. However, because it is a computer language, it is much more precise than English. The programmer must, therefore, learn the rules that govern COBOL and follow them exactly. These rules are detailed later, beginning in the next chapter. The rest of this chapter gives a general picture of how a COBOL program is put together.

The basic unit of COBOL is the word -- which may be a COBOL reserved word or a programmer-defined word. Reserved words have a specific syntactical meaning to the COBOL compiler, and must be spelled exactly as shown in the reserved word list (see Appendix C). Programmer-defined words are assigned by the user to such items as data-names and procedure-names; they must conform to the COBOL rules for the formation of names.

Reserved words and programmer-defined words are combined by the programmer into clauses (in the Environment and Data Divisions) and statements (in the Procedure Division); clauses and statements must be formed following the specific syntactical rules of COBOL. A clause or a statement specifies only one action to be performed, one condition to be analyzed, or one description of data. Clauses and statements can be combined into sentences. Sentences may be simple (one statement or one clause), or they may be compound (a combination of statements or a combination of clauses). Sentences can be combined into paragraphs, which are named units of logically related sentences, and paragraphs can be further combined into named sections. In the Procedure Division, both paragraphs and sections can be referred to as procedures, and their names can be referred to as procedure names.

There are four divisions in each COBOL program. Each is placed in its logical sequence, each has its necessary logical function in the program, and each uses information developed in the divisions preceding it. The four divisions and their sequence are:

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
```

To illustrate how a COBOL program is written, let us create a simplified procedure to record changes in the stocks of office furniture

offered for sale by a manufacturer. We will need such data items as an item code to identify each type of product, an item name corresponding to the code, the unit price of each item of stock, the reorder point at which the manufacturer replaces each item, and the amount of stock on hand plus its value for each item. Our procedure will update a MASTER-FILE of all stocks the manufacturer carries by reading a DETAIL-FILE of current transactions, performing the necessary calculations, and placing the updated values in the MASTER-FILE. We will also create an ACTION-FILE of items to be reordered. The MASTER-FILE resides on a direct-access (mass storage) disk device; the DETAIL-FILE and ACTION-FILE reside on tape devices.

Many of the examples used in the following discussion have been simplified for greater clarity. Figure 2, at the end of this chapter, shows how the entire UPDATING program would actually be written.

Identification Division

First we must assign a name to our program, presenting the information like this:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. UPDATING.
```

PROGRAM-ID informs the compiler that we have chosen the unique name UPDATING for the program we have written.

In addition to the name of the program, the Identification Division allows us to list the name of the programmer, the date the program was written, and other information that will serve to document the program.

Environment Division

Although COBOL is, to a large degree, machine independent, there are some aspects of any program that depend on the particular computer being used and on its associated input/output devices. In the Environment Division, the characteristics of the computer used may be identified. The location of each file referenced in the program, and how each one of them will be used, must be described.

First we will describe the source computer (the one the compiler uses) and the object computer (the one the object program uses) as follows:

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.  IBM-360-H50.
OBJECT-COMPUTER.  IBM-360-H50.
```

This tells us that both computers will be an IBM System/360 Model H50.

Next we must identify the files to be used in our program, and assign them to specific input/output devices. This is done in the Input-Output Section.

```
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT MASTER-FILE, ASSIGN TO ...
        ACCESS MODE IS RANDOM
        ACTUAL KEY IS FILEKEY.
    SELECT DETAIL-FILE, ASSIGN TO ...
        ACCESS MODE IS SEQUENTIAL.
    SELECT ACTION-FILE, ASSIGN TO ...
        .
        .
        .
```

The ellipses (...) in the three foregoing ASSIGN clauses indicate the omission of system-name, an item too complex to illustrate here. System-name is in a special format, and it tells the compiler on which symbolic unit the file will be found and in what way the data is organized within the file.

Our MASTER-FILE resides on a disk pack, which is a mass storage device. Access for these devices can be either RANDOM or SEQUENTIAL. If ACCESS MODE IS RANDOM, then each record within the file can be located directly through the use of a key (identified in the statement ACTUAL KEY IS FILEKEY). For our program we have named this key FILEKEY, and later in the Data Division we will describe it fully. During the processing of our object program, each record will be made available to the user in the sequence that the keys are presented to the system.

Our DETAIL-FILE and our ACTION-FILE reside on tape. This means that ACCESS MODE must be sequential. On tape it is necessary to refer to each successive record in the file in order to find any individual record we might wish to access. Since the compiler assumes that the ACCESS MODE is sequential unless specified otherwise, the ACCESS MODE clause is never needed in describing a tape file.

Data Division

The Data Division of the COBOL program gives a detailed description of all the data to be used in the program -- whether to be read into the machine, used in intermediate processing, or written as output. To simplify this discussion, we will describe only the two most important aspects of data description.

1. We will inform the compiler that we intend to work with one kind of input record, our detail record; one kind of update record, our master record; and one kind of output record, our action record.

2. We will assign data-names to each of the items of data to be used.

First, we must organize the two input records -- a MASTER-RECORD and a DETAIL-RECORD. The MASTER-RECORD will be derived from ledger records that look like those shown in Table 1.

Table 1. Typical Ledger Records Used for MASTER-RECORD

Item Code	Item Name	Stock on Hand	Unit Price ($)	Stock Value ($)	Order Point
A10	2-drawer file cabinets	100	50	5,000	50
A11	3-drawer file cabinets	175	80	14,000	80
A12	4-drawer file cabinets	200	110	22,000	150
B10	Secretarial desks	150	200	30,000	120
B11	Salesmen's desks	50	175	8,750	50
B12	Executive desks	75	500	37,500	60
C10	Secretarial posture chairs	125	50	6,250	140
C11	Side chairs	50	40	2,000	60
C12	Executive swivel chairs	25	150	3,750	20

There will be a MASTER-RECORD for each item in this list. In defining the data for the compiler, we will make sure that each record is in the same format as all the others. Thus, if we specify the characteristics of a single record, we will have specified the characteristics of the whole set. In this way, all of the master records can be organized into a data set, or file, that we will name MASTER-FILE. Each complete record within the file we will name the MASTER-RECORD, with the individual items of data grouped within it. Accordingly, we will begin our Data Division as follows:

```
DATA DIVISION.
FILE SECTION.
FD  MASTER-FILE DATA RECORD IS MASTER-RECORD...
        .
        .
        .
01  MASTER-RECORD.
    05  ITEM-CODE...
    05  ITEM-NAME...
    05  STOCK-ON-HAND...
    05  UNIT-PRICE...
    05  STOCK-VALUE...
    05  ORDER-POINT...
        .
        .
        .
```

The FILE SECTION entry informs the COBOL compiler that the items that follow will describe the format of each file and of each record within each file to be used in the program. The level indicator FD (File Description) introduces the MASTER-FILE itself, and tells the compiler that each entry within MASTER-FILE will be referred to as MASTER-RECORD. The entry with level number 01 identifies the MASTER-RECORD itself, and the subordinate entries with level number 05 describe the subdivisions within the complete MASTER-RECORD. The concept of levels is a basic attribute of COBOL. The highest level is the FD, the next highest level is 01. Level numbers from 02 through 49 may subdivide the record, and the subdivisions themselves can be further subdivided if need be. The smaller the subdivision, the larger the level number must be.

Each of the data items would actually be described more fully than is shown here. In an actual program, for example, we would inform the compiler that each of the items identified as STOCK-ON-HAND, UNIT-PRICE, STOCK-VALUE, and ORDER-POINT would represent positive numeric values of a specific size in a specific form, and so forth. At this point, we need not concern ourselves with these details.

The MASTER-FILE is the main record of current inventory. Changes to this record are made by entering the details of individual transactions or groups of transactions. Thus, receipts of new stocks and shipments to customers will change both STOCK-ON-HAND and STOCK-VALUE. These changes are summarized in the detail record for each item. A typical record would appear in a ledger as shown in Table 2.

Table 2. Typical DETAIL-RECORD

Item Code	Item Name	Receipts	Shipments
B11	Salesmen's desks	25	15

We will therefore organize a DETAIL-FILE, made up of individual items to be referred to as DETAIL-RECORD. DETAIL-FILE will be arranged by ITEM-CODE in ascending numerical order.

```
FD  DETAIL-FILE DATA RECORD IS DETAIL-RECORD...
01  DETAIL-RECORD.
    05  ITEM-CODE...
    05  ITEM-NAME...
    05  RECEIPTS...
    05  SHIPMENTS...
```

The ACTION-FILE will contain a list of items to be reordered, plus relevant data:

```
FD  ACTION-FILE DATA RECORD IS ACTION-RECORD...
01  ACTION-RECORD.
    05  ITEM-CODE...
    05  ITEM-NAME...
    05  STOCK-ON-HAND...
    05  UNIT-PRICE...
    05  ORDER-POINT...
```

This completes the description of the files we will use.

Note that the names of data items contained within the files are in many cases identical. Yet each name within each file must be unique, or ambiguities in references to them will occur. Since identical names are used in our data descriptions, we must use a special means of distinguishing between them. The COBOL naming system, with its concept of levels, allows us to make this distinction by reference to some larger group of data of which the item is a part. Thus, ITEM-CODE OF MASTER-RECORD, and ITEM-CODE OF DETAIL-RECORD, and ITEM-CODE OF ACTION-RECORD can be clearly differentiated from each other. The use of a higher level name in this way is called qualification. Qualification is required in making distinctions between otherwise identical names.

Now we must construct the Working-Storage Section of our Data Division. This section describes records and data items that are not part of the files, but are used during the processing of the object program.

For our program, we will need several entries in our Working-Storage Section. Among them will be several items constructed with level numbers, similar to those used to describe the file records.

```
WORKING-STORAGE SECTION.
        .
        .
        .
77  QUOTIENT...
        .
        .
        .
01  FILEKEY...
    05  TRACK-ID...
    05  RECORD-ID...
01  ERROR-MESSAGE.
    05  ERROR-MESSAGE-1...
    05  ERROR-MESSAGE-2...
    05  ERROR-MESSAGE-3...
```

We will use the FILEKEY record in constructing the FILEKEY. We will use the ERROR-MESSAGE record to create warning messages when errors are encountered during object time processing. We have assigned the level number 77 to the data item named QUOTIENT. This level number informs the compiler that QUOTIENT is a noncontiguous data item -- that is, that this item has no relationship to any other data item described in the Working-Storage Section. Note that the data items related to each other must be listed after all the noncontiguous data items.

Procedure Division

The Procedure Division contains the instructions needed to solve our problem. To accomplish this, we will use several types of COBOL statements. In constructing our sample program, we will discover how each type of statement can be used to obtain the results we want.

Beginning the Program -- Input Operations

Our first step in building the Procedure Division is to make the records contained in the MASTER-FILE and the DETAIL-FILE available for processing. If we write the statements:

```
PROCEDURE DIVISION.
    .
    .
    .
    OPEN INPUT DETAIL-FILE.
    OPEN I-O MASTER-FILE.
```

the system establishes a line of communication with each file, checks to make sure that each is available for use, brings the first record of the DETAIL-FILE file into special areas of internal storage known as buffers, and does other housekeeping.

The files can now be accessed. Our next statements will therefore be:

```
READ DETAIL-FILE AT END GO TO END-ROUTINE.
.
.
.
READ MASTER-FILE INVALID KEY PERFORM INPUT-ERROR
GO TO ERROR-ROUTINE-1.
```

At this point in our program, these two statements make available for processing the first record from each file. (Note that the AT END phrase and the INVALID KEY phrase are necessary in these sentences. Their use will be explained later.) We are now able to begin arithmetic operations upon the data.

Arithmetic Statements

We have already seen that the COBOL language contains the verb ADD. Using this verb, we can add RECEIPTS to STOCK-ON-HAND by writing the COBOL statement:

```
ADD RECEIPTS TO STOCK-ON-HAND.
```

This instructs the program to find the value of RECEIPTS in the DETAIL-RECORD and add it to the value of STOCK-ON-HAND in the MASTER-RECORD. (For the sake of brevity, this example and the ones following have been simplified by omitting the name qualification which would be necessary in actual coding. Figure 2, at the end of this chapter, shows the actual coding necessary.)

Next we must reduce the new value of STOCK-ON-HAND by the amount of SHIPMENTS. The COBOL verb SUBTRACT will accomplish this result for us, and so we write:

```
SUBTRACT SHIPMENTS FROM STOCK-ON-HAND.
```

These two statements, carried out in succession, will produce a current value for STOCK-ON-HAND.

Actually, there is a more concise way to perform this particular calculation. We have broken it into two steps, but COBOL provides another verb which allows us to specify more than one arithmetic operation in a single statement. This is the verb COMPUTE.

```
COMPUTE STOCK-ON-HAND = STOCK-ON-HAND + RECEIPTS - SHIPMENTS.
```

A COMPUTE statement is always interpreted to mean that the value to the left of the equal sign will be changed to equal the value resulting from the calculation specified to the right. The calculation to the right of the equal sign is evaluated from left to right. That is, in our example, the addition is performed first and then the subtraction.

The name STOCK-ON-HAND occurs twice in this sentence, but this causes no difficulty. The expression to the right is calculated first; thus, it is the current value of STOCK-ON-HAND that is used as the basis for computing the new value. When this new value has been calculated, it replaces the old value of STOCK-ON-HAND in the MASTER-RECORD.

So far we have brought only the value of STOCK-ON-HAND up to date, but a change in this value will also cause a change in STOCK-VALUE. We will assume that this figure does not include allowances for quantity discounts, damage to stock, or other such factors, and that STOCK-VALUE is nothing more than the unit price multiplied by the number of items currently in stock. COBOL provides us with a MULTIPLY verb, which permits us to accomplish this:

```
MULTIPLY STOCK-ON-HAND BY UNIT-PRICE GIVING STOCK-VALUE.
```

The result of the multiplication will be placed in the MASTER-RECORD as the new value of STOCK-VALUE. Within the program, this statement must be executed after the COMPUTE statement we wrote earlier, since STOCK-ON-HAND must be the updated, not the original, value.

Conditional Statements

There are instructions in COBOL that examine data to determine whether or not some condition is present and, depending on what is found, to carry out an appropriate course of action.

The MASTER-RECORD contains an item called ORDER-POINT. An item is to be reordered when its stock has been reduced either to or below its order point. Let us assume that we have written a procedure for initiating such an order, and that we have given the name REORDER-ROUTINE to this procedure. We then write the following two sentences:

```
IF STOCK-ON-HAND IS LESS THAN ORDER-POINT
   PERFORM REORDER-1...
IF STOCK-ON-HAND IS EQUAL TO ORDER-POINT
   PERFORM REORDER-1...
```

in order to compare the present value of STOCK-ON-HAND with the value of ORDER-POINT. If STOCK-ON-HAND is a smaller value, the COBOL verb PERFORM causes a transfer of control to the paragraph named REORDER-1. If STOCK-ON-HAND is not less than ORDER-POINT, our next instruction is evaluated. If the values are equal, control is transferred to REORDER-1. If the values are not equal, control is transferred to the next instruction.

It is permissible, in COBOL, to combine the two tests into one:

```
IF STOCK-ON-HAND IS LESS THAN ORDER-POINT OR EQUAL TO
     ORDER-POINT PERFORM REORDER-1...
```

Here we are writing a compound condition with an implied subject. STOCK-ON-HAND, the subject of the first condition, is understood to be the subject of the second condition as well. Compound conditions increase the flexibility of COBOL and make the handling of many kinds of problems easier.

In this example, we tested successively for two conditions out of three. Unless the programmer has some need to distinguish between these two conditions (and he might), it would be simpler to test for the third condition instead:

```
IF STOCK-ON-HAND IS GREATER THAN ORDER-POINT NEXT SENTENCE
     ELSE PERFORM REORDER-1...
```

The words NEXT SENTENCE have a special meaning in COBOL. When IF STOCK-ON-HAND IS GREATER THAN ORDER-POINT is true, NEXT SENTENCE takes effect. Every instruction in the balance of the IF sentence is ignored, and control is transferred to the sentence following.

The test can be simplified even further, since COBOL allows us to express negation:

```
IF STOCK-ON-HAND IS NOT GREATER THAN ORDER-POINT
    PERFORM REORDER-1...
```

If the value of STOCK-VALUE is less than or equal to that of ORDER-POINT, control is transferred to REORDER-1. If the value is greater, control automatically passes to the next successive sentence.

The actual rules for specifying tests and comparisons will be given in a subsequent chapter.

Handling Possible Errors

Let us write one more conditional statement:

```
IF STOCK-ON-HAND IS LESS THAN ZERO...
     GO TO ERROR-WRITE.
```

One would expect that the smallest value STOCK-ON-HAND could assume would be zero. If a negative record were processed, the values found would probably be completely erroneous. To prevent this, the programmer could anticipate the possibility of error and write a special routine to be executed whenever the value of STOCK-ON-HAND was found to be negative. Such a routine could stop the processing of this record, print out the erroneous data, and proceed automatically to process the following records. The more comprehensive a programmer makes his error checking, the less likely it is that inaccurate information will pass through without being marked for special attention.

Data-Manipulation Statements

We saw in the preceding paragraph that if the value of STOCK-ON-HAND fell below a certain point, control would be passed to a special sequence of instructions named REORDER-1. Our output ACTION-FILE has been set up for just this purpose. The bulk of REORDER-1 could consist of data-manipulation statements; that is, instructions which move the necessary data items from the MASTER-RECORD area in storage to that area reserved for the ACTION-FILE records. The COBOL verb MOVE can be used to accomplish this. We must explain here that the verb MOVE does <u>not</u> mean an actual physical movement of data. Instead, it means that the data items from MASTER-RECORD are copied into ACTION-RECORD. Items within MASTER-RECORD are not destroyed when a MOVE statement is executed, and are available for further processing. Individual items contained in ACTION-RECORD before the operation, however, are replaced when the statement is executed. Our MOVE statements will be written:

```
MOVE ITEM-CODE OF MASTER-RECORD TO ITEM-CODE
     OF ACTION-RECORD.
MOVE ITEM-NAME OF MASTER-RECORD TO ITEM-NAME
     OF ACTION-RECORD.
MOVE STOCK-ON-HAND OF MASTER-RECORD TO
     STOCK-ON-HAND OF ACTION-RECORD.
MOVE UNIT-PRICE OF MASTER-RECORD TO UNIT-PRICE
     OF ACTION-RECORD.
MOVE ORDER-POINT OF MASTER-RECORD TO ORDER-POINT
     OF ACTION-RECORD.
```

With these five statements, we have set up the ACTION-RECORD to be written in the ACTION-FILE. However, there is another and easier method for the programmer to specify the five MOVE operations by taking advantage of the qualification system in naming:

```
MOVE CORRESPONDING MASTER-RECORD TO ACTION-RECORD.
```

The word CORRESPONDING indicates that those data items with names which are identical in both records are to be copied from MASTER-RECORD into ACTION-RECORD. Thus, five MOVE statements are replaced by one.

Output Operations

When all arithmetic and data-manipulation statements have been executed, we will write the results in some form. COBOL allows us to do this with a WRITE instruction.

```
WRITE MASTER-RECORD INVALID KEY ...
     GO TO ERROR-WRITE.
```

Or, if we were to indicate that an item was to be reordered, we could write the following:

```
WRITE ACTION-RECORD.
```

In either case, the record would be recorded on the output device specified for the file in the Environment Division; its format would be determined by the Data Division description of the file.

Procedure-Branching Statements

In our inventory problem, there will be as many master records as there are kinds of furniture in stock, and there will be a varying number of detail records. We must read each successive DETAIL-RECORD in DETAIL-FILE, until every one of the records in the file has been processed.

Each time a DETAIL-RECORD is read, we will perform calculations upon its ITEM-CODE in order to produce our FILEKEY. FILEKEY will then be used to find a matching record in MASTER-RECORD. If a matching record cannot be found, either the DETAIL-RECORD is in error, or the MASTER-RECORD is missing from the file and we must mark that record for special processing. Consider the series of statements in Figure 1.

You will note that several new elements have been added to the arithmetic statements and conditional phrases we have already discussed. First, there are the elements that extend to the left of the other statements. These elements are the procedure-names we described earlier. Each procedure-name indicates the beginning of a paragraph or a section within the program, and each indicates a reference point for programmer-specified transfer of control. When a procedure is entered, each logically successive instruction is processed in turn.

```
NEXT-DETAIL-RECORD-ROUTINE.
    READ DETAIL-FILE AT END GO TO END-ROUTINE-1.
         .
         .
         .
    READ MASTER-FILE INVALID KEY PERFORM INPUT-ERROR
         GO TO ERROR-WRITE.
COMPUTATION-ROUTINE.
         .
         .
         .
    IF STOCK-ON-HAND IN MASTER-RECORD IS LESS THAN ZERO
         PERFORM DATA-ERROR GO TO ERROR-WRITE.
         .
         .
         .
    IF STOCK-ON-HAND IN MASTER-RECORD IS NOT GREATER THAN
         ORDER-POINT IN MASTER-RECORD PERFORM REORDER-1
         THRU REORDER-2.
WRITE-MASTER-ROUTINE.
         .
         .
         .
    GO TO NEXT-DETAIL-RECORD-ROUTINE.
REORDER-1.
    GO TO SWITCH-ROUTINE.
SWITCH-ROUTINE.
    ALTER REORDER-1 TO  REORDER-2
         END-ROUTINE-1 TO  END-ROUTINE-3.
    OPEN OUTPUT ACTION-FILE.
REORDER-2.
    MOVE CORRESPONDING MASTER-RECORD TO ACTION-RECORD.
    WRITE ACTION-RECORD.
ERROR-WRITE.
         .
         .
         .
    GO TO NEXT-DETAIL-RECORD-ROUTINE.
INPUT-ERROR.
    MOVE " KEY ERROR ON INPUT " TO ERROR-MESSAGE-1.
         .
         .
         .
DATA-ERROR.
    MOVE "DATA ERROR ON INPUT " TO ERROR-MESSAGE-1.
         .
         .
         .
END-ROUTINE-1.
    GO TO END-ROUTINE-2.
END-ROUTINE-3.
    CLOSE ACTION-FILE.
END-ROUTINE-2.
    CLOSE DETAIL-FILE.
    CLOSE MASTER-FILE.
    STOP RUN.
```

Figure 1. Illustration of Procedure Branching

The procedure-names give us a means of controlling the processing of successive items in our DETAIL-FILE. If, for example, we have finished processing one complete DETAIL-RECORD and wish to begin processing the next, control must be transferred to NEXT-DETAIL-RECORD-ROUTINE. This is accomplished through the use of the COBOL verb GO TO, which transfers control to the procedure indicated, as in the statement:

```
GO TO NEXT-DETAIL-RECORD-ROUTINE.
```

Processing then continues with the first sentence following the procedure-name NEXT-DETAIL-RECORD-ROUTINE. Note the many other examples of the GO TO statement in our program. Each gives us the means of transferring control from one procedure to another.

Another way in which to control the processing of a series of records is through the use of the COBOL verb PERFORM. Like the verb GO TO, the verb PERFORM specifies a transfer to the first sentence of a routine. In addition, PERFORM provides various ways of determining the manner in which the procedure is to be processed.

Within the COMPUTATION-ROUTINE, there is a statement which uses the COBOL verb PERFORM:

```
IF STOCK-ON-HAND IN MASTER-RECORD IS LESS THAN ZERO
    PERFORM DATA-ERROR GO TO ERROR-WRITE.
```

When STOCK-ON-HAND is computed to be less than zero, an error condition has occurred. First, the compiler is instructed to transfer control to a procedure named DATA-ERROR. Within DATA-ERROR, there is a MOVE statement which copies the characters within quotation marks ("DATA ERROR ON INPUT ") into the area of storage reserved for ERROR-MESSAGE-1. (The characters within quotation marks are what is known as a literal -- because they literally mean themselves. When ERROR-MESSAGE is displayed, these words will be an actual part of the error message.) Control is now transferred back to the next statement following the PERFORM statement, which is the GO TO ERROR-WRITE statement.

Note that within COMPUTATION-ROUTINE there is another PERFORM statement that is processed in a similar manner:

```
IF STOCK-ON-HAND IN MASTER-RECORD IS NOT GREATER THAN
    ORDER-POINT IN MASTER-RECORD
    PERFORM REORDER-1 THRU REORDER-2.
```

This time, the PERFORM statement instructs the object program to process several paragraphs before returning control to the next successive statement. Thus, when this PERFORM statement is executed, control is transferred to REORDER-1. This paragraph is executed, the next paragraph, SWITCH-ROUTINE, is also executed, and then all the statements contained in REORDER-2 are executed, at which point control is returned to the first statement in WRITE-MASTER-ROUTINE -- the next successive statement after the PERFORM statement.

A PERFORM statement may specify that a single section or paragraph be processed, or, if the desired procedure consists of more than one section or paragraph, it can specify two names that identify the beginning and the end of the procedure.

GO TO and PERFORM statements may seem to do much the same job. Yet there are specific reasons that will cause the programmer to choose one over the other. On the one hand, the programmer may wish to transfer control to the same procedure from two entirely different sections of

the program. In this case, PERFORM offers the most convenient method of returning to the point from which the transfer was made. On the other hand, if the programmer wishes to proceed to a portion of the program without specifying a return to the current routine, a GO TO statement will provide the best method of making the transfer.

In addition to the GO TO and PERFORM statements, there is another COBOL statement that affects procedure branching: the ALTER statement.

In any given execution of our object program, we may or may not use our ACTION-FILE. Only if some item in STOCK-ON-HAND has fallen below REORDER-POINT will it be necessary to create an ACTION-RECORD. Therefore, depending upon the data that is being processed, we will open ACTION-FILE only if and when such an operation is necessary.

Suppose that for the first time in a particular execution of our object program we have encountered a value for STOCK-ON-HAND that indicates it must be reordered. The statement:

```
IF STOCK-ON-HAND IN MASTER-RECORD IS NOT GREATER THAN
    ORDER-POINT IN MASTER-RECORD
    PERFORM REORDER-1 THRU REORDER-2.
```

instructs the compiler, when STOCK-ON-HAND is not greater than ORDER-POINT, to transfer control to the first sentence in REORDER-1. REORDER-1 consists of but one statement:

```
GO TO SWITCH-ROUTINE.
```

SWITCH-ROUTINE, as it happens, is the next paragraph, and it contains an ALTER statement:

```
ALTER REORDER-1 TO  REORDER-2
      END-ROUTINE-1 TO  END-ROUTINE-3.
```

This statement instructs the compiler to substitute the words REORDER-2 for SWITCH-ROUTINE (within REORDER-1), and END-ROUTINE-3 for END-ROUTINE-2 (within END-ROUTINE-1). Since, at the time the ALTER statement is executed, we are already beyond the point at which the substitution is to be made in REORDER-1, we continue processing each sequential statement until we reach the end of REORDER-2. We open ACTION-FILE, and so forth, until we return control to the next statement following the PERFORM statement.

However, in this execution of our object program, the next time we must reorder an item, a different sequence of statements is performed. The program transfers control to REORDER-1, but now the GO TO statement within REORDER-1 has a different operand. Instead of SWITCH-ROUTINE, the program is now instructed to transfer control to the paragraph named REORDER-2. Through use of the ALTER statement, we have created a switch that bypasses the OPEN ACTION-FILE statement in subsequent processing of reordered items, since the OPEN statement need be executed but once in any execution of our object program.

Similarly, if ACTION-FILE was never opened in this execution of our object program, it is not necessary to close it. Therefore, the second part of the ALTER statement:

```
END-ROUTINE-1 TO END-ROUTINE-3
```

allows alternate paths of program flow, depending on whether or not this ALTER statement was ever executed. The precise rules for programming the ALTER statement are given later in this publication; note, however, the increased programming flexibility it offers.

Ending the Program

One last step in the logic of our inventory program must now be taken. We have obtained the update information from a record, performed the needed arithmetic calculations, moved the data from one area of storage to another, and written the decision-making and procedure-branching instructions necessary to take care of special cases and to process each succeeding record. Then we have written the updated information into the MASTER-FILE and, when necessary, have written the ACTION-FILE. We must now terminate the program after all records have been acted upon. Remember that we wrote our first READ statement as follows:

```
READ DETAIL-FILE AT END GO TO END-ROUTINE-1.
```

END-ROUTINE-1 will consist of the few instructions necessary to terminate operations for this program.

Just as the programmer must make all the files available to the system with a set of OPEN instructions, he must now disconnect these same files with another series:

```
END-ROUTINE-1.
    GO TO END-ROUTINE-2.
END-ROUTINE-3.
    CLOSE ACTION-FILE.
END-ROUTINE-2.
    CLOSE DETAIL-FILE.
    CLOSE MASTER-FILE.
```

These instructions initiate necessary housekeeping routines. (Note here that, in our program, ACTION-FILE will be closed only if REORDER-1 THRU REORDER-2 has been performed and the ALTER statement has been executed.) Once a file has been closed, it cannot be accessed by the program again. The programmer now writes one last COBOL instruction, and it must be at the logical end of his processing:

```
STOP RUN.
```

At this point, COBOL ending procedures are initiated, and the execution of the program is halted.

This is only a general picture of the way in which a COBOL program works. The following chapters in this manual give detailed descriptions of all four divisions within a COBOL program, with explicit instructions for correct programming in IBM Full American National Standard COBOL.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. UPDATING.
REMARKS. THIS IS A SIMPLIFIED UPDATE PROGRAM, USED AS AN
    EXAMPLE OF BASIC COBOL TECHNIQUES. THE PROGRAM IS
    EXPLAINED IN DETAIL IN THE INTRODUCTION TO THIS MANUAL.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-360-H50.
OBJECT-COMPUTER. IBM-360-H50.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT MASTER-FILE ASSIGN TO DA-2311-D-MASTER
        ACCESS MODE IS RANDOM
        ACTUAL KEY IS FILEKEY.
    SELECT DETAIL-FILE ASSIGN TO UT-2400-S-INFILE
        ACCESS IS SEQUENTIAL.
    SELECT ACTION-FILE ASSIGN TO UT-2400-S-OUTFILE.
DATA DIVISION.
FILE SECTION.
FD  MASTER-FILE LABEL RECORDS ARE STANDARD
    DATA RECORD IS MASTER-RECORD.
01  MASTER-RECORD.
    05  ITEM-CODE        PICTURE X(3).
    05  ITEM-NAME        PICTURE X(29).
    05  STOCK-ON-HAND    PICTURE S9(6)      USAGE COMP SYNC.
    05  UNIT-PRICE       PICTURE S999V99    USAGE COMP SYNC.
    05  STOCK-VALUE      PICTURE S9(9)V99   USAGE COMP SYNC.
    05  ORDER-POINT      PICTURE S9(3)      USAGE COMP SYNC.
FD  DETAIL-FILE LABEL RECORDS ARE OMITTED
    DATA RECORD IS DETAIL-RECORD.
01  DETAIL-RECORD.
    05  ITEM-CODE        PICTURE X(3).
    05  ITEM-NAME        PICTURE X(29).
    05  RECEIPTS         PICTURE S9(3)      USAGE COMP SYNC.
    05  SHIPMENTS        PICTURE S9(3)      USAGE COMP SYNC.
FD  ACTION-FILE LABEL RECORDS ARE OMITTED
    DATA RECORD IS ACTION-RECORD.
01  ACTION-RECORD.
    05  ITEM-CODE        PICTURE X(3).
    05  ITEM-NAME        PICTURE X(29).
    05  STOCK-ON-HAND    PICTURE S9(6)      USAGE COMP SYNC.
    05  UNIT-PRICE       PICTURE S999V99    USAGE COMP SYNC.
    05  ORDER-POINT      PICTURE S9(3)      USAGE COMP SYNC.
WORKING-STORAGE SECTION.
77  SAVE                 PICTURE S9(10)     USAGE COMP SYNC.
77  QUOTIENT             PICTURE S9999      USAGE COMP SYNC.
01  FILEKEY.
    05  TRACK-ID         PICTURE S9(5)      USAGE COMP SYNC.
    05  RECORD-ID        PICTURE X(29).
01  ERROR-MESSAGE.
    05  ERROR-MESSAGE-1 PICTURE X(20).
    05  ERROR-MESSAGE-2 PICTURE X(36).
    05  ERROR-MESSAGE-3 PICTURE X(46).
```

Figure 2. Complete UPDATING Program (Part 1 of 2)

```
PROCEDURE DIVISION.
OPEN-FILES-ROUTINE.
    OPEN INPUT DETAIL-FILE.
    OPEN I-O MASTER-FILE.
NEXT-DETAIL-RECORD-ROUTINE.
    READ DETAIL-FILE AT END GO TO END-ROUTINE-1.
NEXT-MASTER-RECORD-ROUTINE.
    MOVE ITEM-CODE IN DETAIL-RECORD TO SAVE.
    DIVIDE 19 INTO SAVE GIVING QUOTIENT
        REMAINDER TRACK-ID.
    MOVE ITEM-NAME IN DETAIL-RECORD TO RECORD-ID.
    READ MASTER-FILE INVALID KEY
        PERFORM INPUT-ERROR GO TO ERROR-WRITE.
COMPUTATION-ROUTINE.
    COMPUTE STOCK-ON-HAND IN MASTER-RECORD = STOCK-ON-HAND
        IN MASTER-RECORD + RECEIPTS - SHIPMENTS.
    IF STOCK-ON-HAND IN MASTER-RECORD IS LESS THAN ZERO
        PERFORM DATA-ERROR GO TO ERROR-WRITE.
    MULTIPLY STOCK-ON-HAND IN MASTER-RECORD BY UNIT-PRICE
        IN MASTER-RECORD GIVING STOCK-VALUE
        IN MASTER-RECORD.
    IF STOCK-ON-HAND IN MASTER-RECORD IS NOT GREATER THAN
        ORDER-POINT IN MASTER-RECORD PERFORM REORDER-1
        THRU REORDER-2.
WRITE-MASTER-ROUTINE.
    WRITE MASTER-RECORD INVALID KEY
        PERFORM OUTPUT-ERROR GO TO ERROR-WRITE.
    GO TO NEXT-DETAIL-RECORD-ROUTINE.
REORDER-1. GO TO SWITCH-ROUTINE.
SWITCH-ROUTINE.
    ALTER REORDER-1 TO  REORDER-2
       END-ROUTINE-1 TO  END-ROUTINE-3.
    DISPLAY "ACTION FILE UTILIZED".
    OPEN OUTPUT ACTION-FILE.
REORDER-2.
    MOVE CORRESPONDING MASTER-RECORD TO ACTION-RECORD.
    WRITE ACTION-RECORD.
ERROR-WRITE.
    MOVE DETAIL-RECORD TO ERROR-MESSAGE-2.
    DISPLAY ERROR-MESSAGE.
    GO TO NEXT-DETAIL-RECORD-ROUTINE.
INPUT-ERROR.
    MOVE " KEY ERROR ON INPUT " TO ERROR-MESSAGE-1.
    MOVE SPACES TO ERROR-MESSAGE-3.
DATA-ERROR.
    MOVE "DATA ERROR ON INPUT " TO ERROR-MESSAGE-1.
    MOVE MASTER-RECORD TO ERROR-MESSAGE-3.
OUTPUT-ERROR.
    MOVE "KEY ERROR ON OUTPUT " TO ERROR-MESSAGE-1.
    MOVE SPACES TO ERROR-MESSAGE-3.
END-ROUTINE-1.
    GO TO END-ROUTINE-2.
END-ROUTINE-3.
    CLOSE ACTION-FILE.
END-ROUTINE-2.
    CLOSE DETAIL-FILE.
    CLOSE MASTER-FILE.
    STOP RUN.
```

Figure 2. Complete UPDATING Program (Part 2 of 2)

PART I -- LANGUAGE CONSIDERATIONS

- STRUCTURE OF THE LANGUAGE
- ORGANIZATION OF THE COBOL PROGRAM
- METHODS OF DATA REFERENCE
- USE OF THE COBOL CODING FORM
- FORMAT NOTATION

The COBOL language is so structured that the programmer can write his individual problem program within a framework of words that have particular meaning to the COBOL compiler. The result is the performance of a standard action on specific units of data. For example, in a COBOL statement such as MOVE NET-SALES TO CURRENT-MONTH, the words MOVE and TO indicate standard actions to the COBOL compiler. NET-SALES and CURRENT-MONTH are programmer-defined words which refer to particular units of data being processed by his problem program.

COBOL CHARACTER SET

The complete character set for COBOL consists of the following 51 characters:

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

Note: This compiler's default option for the quotation mark is the apostrophe ('). Unless the default option is overridden, the quotation mark (") may not be used. If conformance with the standard COBOL character set is desired, the programmer must specify the quotation mark (") through an EXEC card at compile time. If the quotation mark is thus specified, the apostrophe (') may not be used.

Characters Used in Words

The characters used in words in a COBOL source program are the following:

0 through 9
A through Z
- (hyphen)

A word is composed of a combination of not more than 30 characters chosen from the character set for words. The word cannot begin or end with a hyphen.

Characters Used for Punctuation

The following characters are used for punctuation:

Character	Meaning
	space
,	comma
;	semicolon
.	period
" or '	quotation mark
(	left parenthesis
)	right parenthesis

The following general rules of punctuation apply in writing a COBOL source program:

1. When any punctuation mark is indicated in a format in this publication, it is required in the program.

2. A period, semicolon, or comma, when used, must not be preceded by a space, but must be followed by a space.

3. A left parenthesis must not be followed immediately by a space; a right parenthesis must not be preceded immediately by a space.

4. At least one space must appear between two successive words and/or parenthetical expressions and/or literals. Two or more successive spaces are treated as a single space, except within nonnumeric literals.

5. An arithmetic operator or an equal sign must always be preceded by a space and followed by a space. A unary operator may be preceded by a left parenthesis.

6. A comma may be used as a separator between successive operands of a statement. An operand of a statement is shown in a format as a lower-case word.

7. A comma or a semicolon may be used to separate a series of clauses. For example, DATA RECORD IS TRANSACTION, RECORD CONTAINS 80 CHARACTERS.

8. A semicolon may be used to separate a series of statements. For example, ADD A TO B; SUBTRACT B FROM C.

9. The word THEN may be used to separate a series of statements. For example, ADD A TO B THEN SUBTRACT B FROM C.

Characters Used for Editing

Editing characters are single characters or specific two-character combinations belonging to the following set:

Character	Meaning
B	space
0	zero
+	plus
-	minus
CR	credit
DB	debit
Z	zero suppression
*	check protection
$	currency sign
,	comma
.	period (decimal point)

(For applications, see the discussion of alphanumeric edited and numeric edited data items in "Data Division.")

Characters Used in Arithmetic Expressions

The characters used in arithmetic expressions are as follows:

Character	Meaning
+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

Arithmetic expressions are used in the COMPUTE statement and in relation conditions (see "Procedure Division" for more details).

Characters Used for Relation Conditions

A relation character is a character that belongs to the following set:

Character	Meaning
>	greater than
<	less than
=	equal to

Relation characters are used in relation conditions (discussed in "Procedure Division").

TYPES OF WORDS

A word is composed of a combination of not more than 30 characters chosen from the character set for words. The word cannot begin or end with a hyphen.

The space (blank) is not an allowable character in a word; the space is a word separator. Wherever a space is used as a word separator, more than one may be used.

A word is terminated by a space, or by a period, right parenthesis, comma, or semicolon.

Reserved Words

Reserved words exist for syntactical purposes and must not appear as user-defined words. However, reserved words may appear as nonnumeric literals, i.e., a reserved word may be enclosed in quotation marks. When used in this manner, they do not take on the meaning of reserved words and violate no syntactical rules.

There are three types of reserved words:

1. Key Words. A key word is a word whose presence is required in a COBOL entry. Such words are upper case and underlined in the formats given in this publication.

   Key words are of three types:

   a. Verbs such as ADD, READ, and ENTER.

   b. Required words, which appear in statement and entry formats, such as the word TO in the ADD statement.

   c. Words that have a specific functional meaning, such as ZERO, NEGATIVE, SECTION, TALLY, etc.

2. Optional Words. Within each format, upper case words that are not underlined are called optional words because they may appear at the user's option. The presence or absence of each optional word in the source program does not alter the compiler's translation. Misspelling of an optional word, or its replacement by another word of any kind, is not allowed.

3. Connectives. There are three types of connectives:

   a. Qualifier connectives, which are used to associate a data-name or paragraph-name with its qualifier. The qualifier connectives are OF and IN (see "Methods of Data Reference").

   b. Series connectives, which link two or more consecutive operands. The series connective is the comma (,).

   c. Logical connectives that are used in compound conditions. The logical connectives are AND, OR, AND NOT, and OR NOT (see "Conditions").

Names

There are three types of names used in a COBOL program:

1. A data-name is a word that contains at least one alphabetic character and identifies a data item in the Data Division. The following are formed according to the rules for data-names:

   file-names
   index-names
   mnemonic-names
   record-names
   report-names
   sort-file-names
   sort-record-names

Program Product Information (Version 4)

cd-names are formed following the rules for formation of a data-name.

2. A condition-name is a name given to a specific value, set of values, or range of values within the complete set of values that a particular data item may assume. The data item itself is called a conditional variable. The condition-name must contain at least one alphabetic character (see "Data Division" and the discussion of "Special-names" in "Environment Division").

3. A procedure-name is either a paragraph-name or a section-name. A procedure-name may be composed solely of numeric characters. Two numeric procedure-names are equivalent if, and only if, they are composed of the same number of digits and have the same value (see "Procedure Division"). The following are formed according to the rules for procedure-names:

   library-names
   program-names

Note: Abbreviations (such as PIC for PICTURE) are allowed for some reserved words; the abbreviation is the equivalent of the complete word. For the formats in which they are allowable, such abbreviations are shown in the format. The reserved words THRU and THROUGH are equivalent. In statement formats, wherever the reserved word THRU appears, the word THROUGH is also allowed.

Special-names

Special-names are used in the SPECIAL-NAMES paragraph of the Environment Division. The term special-name refers to a mnemonic-name. A mnemonic-name is a programmer-defined word that is associated in the Environment Division with a function-name: function-names are names with a fixed meaning, defined by IBM.

In the Procedure Division, mnemonic-name can be written in place of its associated function-name in any format where such substitution is valid. The formation of a mnemonic-name follows the rules for formation of a data-name (see "Special-names" in "Environment Division").

CONSTANTS

A constant is a unit of data whose value is not subject to change. There are two types of constants: literals and figurative constants.

Literals

A literal is a string of characters whose value is determined by the set of characters of which the literal is composed. Every literal belongs to one of two categories, numeric and nonnumeric.

NUMERIC LITERALS: There are two types of numeric literals: fixed-point and floating-point.

A fixed-point numeric literal is defined as a string of characters chosen from the digits 0 through 9, the plus sign, the minus sign, and the decimal point. Every fixed-point numeric literal:

1. Must contain from 1 through 18 digits.
2. Must not contain more than one sign character. If a sign is used, it must appear as the leftmost character of the literal. If the literal is unsigned, the literal is positive.
3. Must not contain more than one decimal point. The decimal point is treated as an assumed decimal point, and may appear anywhere in the literal except as the rightmost character. If the literal contains no decimal point, the literal is an integer.

(See the discussion of fixed-point numeric items in "Data Division.")

A floating-point numeric literal is a data item whose potential range of value is too great for fixed-point representation. A floating-point literal must have the form:

[±]mantissa E[±]exponent

A floating-point literal must appear as a continuous string of characters with no intervening spaces. The plus or minus signs preceding the mantissa and exponent are the only optional characters within the format. The mantissa consists of from 1 through 16 digits with a required decimal point.

The exponent is represented immediately to the right of the mantissa by the symbol E, followed by a plus or minus sign (if a sign is given) and one or two digits. The magnitude of the number represented by a floating-point literal must not exceed .72 x (10^{76}). A zero exponent must be written as 0 or 00. It is assumed that an unsigned exponent is positive.

The value of the literal is the product of the mantissa and ten raised to the power given by the exponent. For example, the literal

+.72E+76

has the value

.72 x 10^{76}

(See the discussion of floating-point numeric items in "Data Division.")

If the literal conforms to the rules for the formation of numeric literals, but is enclosed in quotation marks, it is a nonnumeric literal.

NONNUMERIC LITERALS: A nonnumeric literal is defined as a string of any allowable characters in the Extended Binary Coded Decimal Interchange Code (EBCDIC) set, excluding the quotation mark character. A nonnumeric literal may be composed of from 1 through 120 characters enclosed in quotation marks. Any spaces within the quotation marks are part of the nonnumeric literal and, therefore, are part of the value. All nonnumeric literals are in the alphanumeric category.

Figurative Constants

A figurative constant is a constant to which a specific data-name has been assigned. These data-names are reserved words. Such a data-name must not be enclosed in quotation marks when used as a figurative constant. The singular and plural forms of a figurative constant are equivalent and may be used interchangeably.

A figurative constant may be used in place of a literal wherever a literal appears in a format. There is one exception to this rule: if the literal is restricted to numeric characters, only the figurative constant ZERO (ZEROES, ZEROS) is allowed.

The fixed data-names and their meanings are as follows:

ZERO ZEROES ZEROS	Represents the value 0, or one or more occurrences of the character 0, depending on context.
SPACE SPACES	Represents one or more blanks or spaces.
HIGH-VALUE HIGH-VALUES	Represents one or more occurrences of the character that has the highest value in the computer's collating sequence. The character for HIGH-VALUE is the hexadecimal 'FF'.
LOW-VALUE LOW-VALUES	Represents one or more occurrences of the character that has the lowest value in the computer's collating sequence. The character for LOW-VALUE is the hexadecimal '00'.
QUOTE QUOTES	Represents one or more occurrences of the quotation mark character. The word QUOTE (QUOTES) cannot be used in place of a quotation mark to enclose a nonnumeric literal.
ALL literal	Represents one or more occurrences of the string of characters composing the literal. The literal must be either a nonnumeric literal or a figurative constant other than the ALL literal. When a figurative constant is used, the word ALL is redundant and is used for readability only.

SPECIAL REGISTERS

The compiler generates storage areas that are primarily used to store information produced with the use of special COBOL features; these storage areas are called special registers.

TALLY

The word TALLY is the name of a special register whose implicit description is that of an integer of five digits without an operational sign, and whose implicit USAGE is COMPUTATIONAL. The primary use of the TALLY register is to hold information produced by the EXAMINE statement. References to TALLY may appear wherever an elementary data item of integral value may appear (see the "EXAMINE Statement" in "Procedure Division").

LINE-COUNTER

LINE-COUNTER is a numeric counter that is generated by the Report Writer. (For a complete discussion, see "Report Writer.")

PAGE-COUNTER

PAGE-COUNTER is a numeric counter that is generated by the Report Writer. (For a complete discussion, see "Report Writer.")

CURRENT-DATE

CURRENT-DATE is an 8-byte alphanumeric field, valid only as the sending field in a MOVE statement. The format of these eight bytes is MM/DD/YY (month/day/year).

TIME-OF-DAY

TIME-OF-DAY is a 6-byte external-decimal field, valid only as the sending field in a MOVE statement. The format is HHMMSS (hour, minute, second).

RETURN-CODE

RETURN-CODE is a binary item whose PICTURE is S9999. It can be set by the user to pass a return code to the operating system or the invoking program when executing a STOP RUN, GOBACK, or EXIT PROGRAM statement (see "Subprogram Linkage Statements" in "Procedure Division"). The return code may be used by the operating system to determine subsequent job or job step execution flow. When control is returned to an invoking program, the return code passed by the called program is stored in RETURN-CODE. The compiler initializes the field to 0 (zero), which is the normal return code for a successful completion; other

values returned are conventionally in multiples of 4. However, the maximum value the field can contain is 4095.

(The use of CURRENT-DATE, TIME-OF-DAY, and RETURN-CODE is explained in the Programmer's Guide.)

LABEL-RETURN

LABEL-RETURN is an alphanumeric item whose PICTURE is X. It may be used to indicate the validity of nonstandard labels. At the completion of a USE BEFORE STANDARD LABEL PROCEDURE for an input file, the programmer must set LABEL-RETURN to indicate the validity of the nonstandard label. It must be set to nonzero if the label is not correct.

The following registers are used by the Sort feature and are described under "Sort:"

SORT-FILE-SIZE

SORT-CORE-SIZE

SORT-MODE-SIZE

SORT-RETURN

Program Product Information (Version 3 and Version 4)

For Versions 3 and 4 the following additional Sort special register is also available:

SORT-MESSAGE

Program Product Information (Version 4)

For Version 4, the special registers DATE, DAY, and TIME may be used only as sending fields in conjunction with the ACCEPT statement to make this system information available to the COBOL program.

DATE

DATE is an unsigned external decimal item with PICTURE 9(6). Within DATE the sequence of data elements (from left to right) is: 2 digits for year of century, 2 digits for month of year, 2 digits for day of month. Therefore, July 1, 1971 is expressed as 710701.

DAY

DAY is an unsigned external decimal item with PICTURE 9(5). Within DAY the sequence of data elements (from left to right) is: 2 digits for year of century, 3 digits for day of year. Thus, July 1, 1971 is expressed as 71183.

TIME

TIME is an unsigned external decimal item with PICTURE 9(8). Within TIME the sequence of data elements (from left to right) is: 2 digits for hour of day, 2 digits for minute of hour, 2 digits for second of minute, 2 digits for hundredths of seconds. Thus 2:41 PM is expressed as 14410000.

See the description of the ACCEPT statement in "Procedure Division" for additional information.

Note: The special registers DATE, DAY, and TIME are valid only as sending fields in the ACCEPT statement, as opposed to the special registers CURRENT-DATE and TIME-OF-DAY which are valid only as sending fields in the MOVE statement.

ORGANIZATION OF THE COBOL PROGRAM

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

The four divisions, listed in sequence, and their functions are:

- IDENTIFICATION DIVISION, which names the program.
- ENVIRONMENT DIVISION, which indicates the machine equipment and equipment features to be used in the program.
- DATA DIVISION, which defines the nature and characteristics of data to be processed.
- PROCEDURE DIVISION, which consists of statements directing the processing of data in a specified manner at execution time.

Note: In all formats within this publication, the required clauses and optional clauses (when written) must appear in the sequence given in the format, unless the associated rules explicitly state otherwise.

Structure of the COBOL Program

```
{ IDENTIFICATION DIVISION.}
{ ID 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.

[CONFIGURATION SECTION.

SOURCE-COMPUTER. entry

OBJECT-COMPUTER. entry

[SPECIAL-NAMES. entry]]

[INPUT-OUTPUT SECTION.

FILE-CONTROL. {entry}...

[I-O-CONTROL. entry]]
```

```
DATA DIVISION.
[FILE SECTION.
{file description entry
{record description entry}...}...]
[WORKING-STORAGE SECTION.
[data item description entry]...
[record description entry]...]
[LINKAGE SECTION.
[data item description entry]...
[record description entry]...]
[COMMUNICATION SECTION.                    (Version 4)
{communication description entry           (Version 4)
[record description entry]...}...]         (Version 4)
[REPORT SECTION.
{report description entry
{report group description entry}...}...]
PROCEDURE DIVISION [USING identifier-1 [identifier-2]...]
[[DECLARATIVES.
{section-name SECTION.  USE Sentence.
{paragraph-name.  {sentence}...}...}...
END DECLARATIVES.]
{section-name SECTION [priority].]
{paragraph-name.  {sentence}...}...}...
```

Every name used in a COBOL source program must be unique, either because no other name has the identical spelling, or because it is made unique through qualification, subscripting, or indexing.

An identifier is a data-name, unique in itself, or made unique by the syntactically correct combination of qualifiers, subscripts, and/or indexes.

QUALIFICATION

A name may be made unique if the name exists within a hierarchy of names and the name can be singled out by mentioning one or more of the higher levels of the hierarchy. The higher levels are called qualifiers. Qualification is the process by which such a name is made unique.

Qualification is applied by placing after a data-name or a paragraph-name one or more phrases, each composed of a qualifier preceded by IN or OF. IN and OF are logically equivalent. Only one qualifier is allowed for a paragraph-name.

Enough qualification must be mentioned to make the name unique; however, it may not be necessary to mention all levels of the hierarchy. For example, if there is more than one file whose records contain the field EMPLOYEE-NO, yet there is but one file whose records are named MASTER-RECORD, EMPLOYEE-NO OF MASTER-RECORD would sufficiently qualify EMPLOYEE-NO. EMPLOYEE-NO OF MASTER-RECORD OF MASTER-FILE is valid but unnecessary (see the discussion of level indicators and level numbers in "Data Division").

The name associated with a level indicator is the highest level qualifier available for a data-name. (A level indicator (FD, SD, RD) specifies the beginning of a file description, sort file description, or report description.) A section-name is the highest (and the only) qualifier available for a procedure-name (see the discussion of procedure-names in "Procedure Division"). Thus, level indicator names and section-names must be unique in themselves since they cannot be qualified.

Program Product Information (Version 4)

In the Communication Section, the level indicator CD specifies the beginning of a communication description.

The name of a conditional variable can be used as a qualifier for any of its condition-names. In addition, a conditional variable may be qualified to make it unique.

The rules for qualification follow:

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

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

3. If a data-name or a condition-name is assigned to more than one data item in a source program, the data-name or condition-name must

be qualified each time reference is made to it in the Procedure, Environment, or Data Division (except in the REDEFINES clause where, by definition, qualification is unnecessary). (See the REDEFINES clause in "Data Division.")

4. 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 paragraph-name need not be qualified when referred to within the section in which it appears.

5. A data-name cannot be subscripted when it is being used as a qualifier.

6. A name can be qualified even though it does not need qualification; if there is more than one combination of qualifiers that ensures uniqueness, then any of these combinations can be used.

Although user-defined data-names can be duplicated within the Data Division and Procedure Division, the following rules should be noted:

1. No duplicate section-names are allowed.

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

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

SUBSCRIPTING

Subscripts can be used only when reference is made to an individual element within a list or table of elements that have not been assigned individual data-names (see "Table Handling").

INDEXING

References can be made to individual elements within a table of elements by specifying indexing for that reference. An index is assigned to a given level of a table by using an INDEXED BY clause in the definition of the table. A name given in the INDEXED BY clause is known as an index-name and is used to refer to the assigned index (see "Table Handling").

USE OF THE COBOL CODING FORM

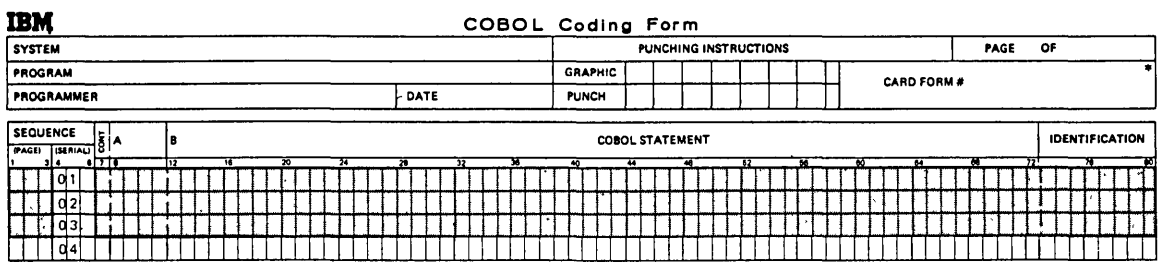
The reference format provides a standard method for writing COBOL source programs. The format is described in terms of character positions in a line on an input/output medium. Punched cards are the initial input medium to the COBOL compiler. The compiler accepts source programs written in reference format (see Figure 3) and produces an output listing of the source program in the same reference format.

The rules for spacing given in the following discussion of the reference format take precedence over any other specifications for spacing given in this publication.

SEQUENCE NUMBERS

A sequence number, consisting of six digits in the sequence number area, is used to identify numerically each card image to be compiled by the COBOL compiler. The use of sequence numbers is optional.

If sequence numbers are present, they must be in ascending order. An error message is issued when source language input is out of sequence. Sequence checking can be suppressed at compile time by overriding the compiler's default option of checking.



Columns 1-6 represent the sequence number area.
Column 7 is the continuation area.
Columns 8-11 represent Area A } Used for writing COBOL source statements.
Columns 12-72 represent Area B }
Columns 73-80 are used to identify the program.

Figure 3. Reference Format

AREA A AND AREA B

Area A, columns 8 through 11, is reserved for the beginning of division headers, section-names, paragraph-names, level indicators, and certain level numbers. Area B occupies columns 12 through 72.

Division Header

The division header must be the first line in a division. The division header starts in Area A with the division-name, followed by a space and the word DIVISION, and a period. If this program is to be called, a space and a USING clause may follow the words PROCEDURE DIVISION. No other text may appear on the same line as the division header.

Section Header

The name of a section starts in Area A of any line following the division header. The section-name is followed by a space, the word SECTION, and a period. If program segmentation is desired, a space and a priority number may follow the word SECTION. No other text may appear on the same line as the section-header, except USE and COPY sentences.

Note: Although USE and COPY may appear in the Declaratives portion of the Procedure Division, only USE is restricted to the Declaratives portion. COPY may be used elsewhere in the COBOL program.

Paragraph-names and Paragraphs

The name of a paragraph starts in Area A of any line following the division header. It is followed by a period followed by a space.

A paragraph consists of one or more successive sentences. The first sentence in a paragraph begins anywhere in Area B of either the same line as paragraph-name or the immediately following line. Each successive line in the paragraph starts anywhere in Area B.

Level Indicators and Level Numbers

In those Data Division entries that begin with a level indicator, the level indicator begins in Area A, followed in Area B by its associated file-name and appropriate descriptive information.

In those data description entries that begin with a level number 01 or 77, the level number begins in Area A, followed in Area B by its associated data-name and appropriate descriptive information.

In those data description entries that begin with level numbers 02 through 49, 66, or 88, the level number may begin anywhere in Area A or Area B, followed in Area B by its associated data-name and descriptive information.

CONTINUATION OF LINES

Any sentence or entry that requires more than one line is continued by starting subsequent line(s) in Area B. These subsequent lines are called continuation lines. The line being continued is called the continued line. If a sentence or entry occupies more than two lines, all lines other than the first and last are both continuation and continued lines.

CONTINUATION OF NONNUMERIC LITERALS

When a nonnumeric literal is continued from one line to another, a hyphen is placed in column 7 of the continuation line, and a quotation mark preceding the continuation of the literal may be placed anywhere in Area B. All spaces at the end of the continued line and any spaces following the quotation mark of the continuation line and preceding the final quotation mark are considered part of the literal.

CONTINUATION OF WORDS AND NUMERIC LITERALS

When a word or numeric literal is continued from one line to another, a hyphen must be placed in column 7 of the continuation line to indicate that the first nonblank character in Area B of the continuation line is to follow the last nonblank character on the continued line, without an intervening space.

BLANK LINES

A blank line is one that contains nothing but spaces from column 7 through column 72, inclusive. A blank line may appear anywhere in the source program, except immediately preceding a continuation line.

COMMENT LINES

Explanatory comments may be inserted on any line within a source program by placing an asterisk in column 7 of the line. Any combination of the characters from the EBCDIC set may be included in Areas A and B of that line. The asterisk and the characters will be produced on the source listing but serve no other purpose (see the NOTE statement in "Compiler Directing Statements" in "Procedure Division.")

FORMAT NOTATION

Throughout this publication, basic formats are prescribed for various elements of COBOL. These generalized descriptions are intended to guide the programmer in writing his own statements. They are presented in a uniform system of notation, explained in the following paragraphs. Although it is not part of COBOL, this notation is useful in describing COBOL.

1. All words printed entirely in capital letters are reserved words. These are words that have preassigned meanings in COBOL. In all formats, words in capital letters represent an actual occurrence of those words. If any such word is incorrectly spelled, it will not be recognized as a reserved word and may cause an error in the program.

2. All underlined reserved words are required unless the portion of the format containing them is itself optional. These are key words. If any such word is missing or is incorrectly spelled, it is considered an error in the program. Reserved words not underlined may be included or omitted at the option of the programmer. These words are used only for the sake of readability; they are called optional words and, when used, must be correctly spelled.

3. The characters +, -, <, >, =, when appearing in formats, although not underlined, are required when such formats are used.

4. All punctuation and other special characters (except those symbols cited in the following paragraphs) represent the actual occurrence of those characters. Punctuation is essential where it is shown. Additional punctuation can be inserted, according to the rules for punctuation specified in this publication.

5. Words that are printed in lower-case letters represent information to be supplied by the programmer. All such words are defined in the accompanying text.

6. In order to facilitate references to them in text, some lower-case words are followed by a hyphen and a digit or letter. This modification does not change the syntactical definition of the word.

7. Certain entries in the formats consist of a capitalized word(s) followed by the word "Clause" or "Statement." These designate clauses or statements that are described in other formats, in appropriate sections of the text.

8. Square brackets ([ ]) are used to indicate that the enclosed item may be used or omitted, depending on the requirements of the particular program. When two or more items are stacked within brackets, one or none of them may occur.

9. Braces ({ }) enclosing vertically stacked items indicate that one of the enclosed items is required.

10. The ellipsis (...) indicates that the immediately preceding unit may occur once, or any number of times in succession. A unit means either a single lower-case word, or a group of lower-case words and one or more reserved words enclosed in brackets or braces. If a term is enclosed in brackets or braces, the entire unit of which it is a part must be repeated when repetition is specified.

11. Comments, restrictions, and clarifications on the use and meaning of every format are contained in the appropriate portions of the text.

PART II -- IDENTIFICATION AND ENVIRONMENT DIVISIONS

- IDENTIFICATION DIVISION
- ENVIRONMENT DIVISION -- FILE PROCESSING SUMMARY
- ORGANIZATION OF THE ENVIRONMENT DIVISION
- ENVIRONMENT DIVISION -- CONFIGURATION SECTION
- ENVIRONMENT DIVISION -- INPUT-OUTPUT SECTION

IDENTIFICATION DIVISION

The Identification Division is the first division of a COBOL program. It identifies the source program and the object program. A source program is the initial problem program; an object program is the output from a compilation.

In addition, the user may include the date the program is written, the date the compilation of the source program is accomplished, etc., in the paragraphs shown.

Structure of the Identification Division

```
{IDENTIFICATION DIVISION.}
{ID 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]...]
```

Specific paragraph-names identify the type of information contained in the paragraph. The name of the program must be given in the first paragraph, which is the PROGRAM-ID paragraph. The other paragraphs are optional. If included, they must be presented in the order shown. However, this compiler will accept them in any order.

The Identification Division must begin with the reserved words IDENTIFICATION DIVISION followed by a period. Each comment-entry may be any combination of characters from the EBCDIC set, organized to conform to sentence and paragraph structure. This compiler will accept ID DIVISION followed by a period as a substitute for the standard division header.

PROGRAM-ID Paragraph

The PROGRAM-ID paragraph gives the name by which a program is identified.

Format
PROGRAM-ID. program-name.

PROGRAM-ID/DATE-COMPILED Paragraphs

The PROGRAM-ID paragraph contains the name of the program and must be present in every program.

Program-name identifies the object program to the control program. Program-name must conform to the rules for formation of a procedure-name. However, this compiler accepts program-name written within quotation marks. The first eight characters of program-name are used as the identifying name of the program and should therefore be unique as a program-name.

Since the system expects the first character of program-name to be alphabetic, the first character, if it is numeric, will be converted as follows:

0 to J

1-9 to A-I

Since the system does not include the hyphen as an allowable character, the hyphen is converted to zero if it appears as the second through eighth character of the name.

Note: For additional information concerning program-name when using the Sort or Segmentation features, see the Programmer's Guide.

DATE-COMPILED Paragraph

The DATE-COMPILED paragraph provides the compilation date on the source program listing.

Format
DATE-COMPILED. [comment-entry]

The paragraph-name DATE-COMPILED causes the current date to be inserted during program compilation. If a comment-entry is present, even though it spans lines, it is replaced in its entirety with the current date.

ENVIRONMENT DIVISION -- FILE PROCESSING SUMMARY

In COBOL, all aspects of the total data processing problem that depend on the physical characteristics of a specific computer are given in one portion of the source program known as the Environment Division. Thus, a change in computers entails major changes in this division only. The primary functions of the Environment Division are to describe the computer system on which the object program is run and to establish the necessary links between the other divisions of the source program and the characteristics of the computer.

The exact contents of the Environment Division depend on the method used to process files in the COBOL program. Before the language elements used in the Environment Division can be discussed meaningfully, some background in the file processing techniques available to the COBOL user must be given.

Each combination of data organization and access method specified in the COBOL language is defined as a file-processing technique. The file-processing technique to be used for a particular file is determined by the data organization of that file and whether the access method is sequential or random. Table 3, at the end of this chapter, summarizes the file-processing techniques.

DATA ORGANIZATION

Four types of data organization are made available to Operating System COBOL users: sequential, direct, relative, and indexed. The means of creating or retrieving logical records in a file differ, depending on which type of data organization exists (organization being the structure of data on a physical file). Each type of data organization is incompatible with the others. Organization of an input file must be the same as the organization of the file when it was created.

Sequential Data Organization

When sequential data organization is used, the logical records in a file are positioned sequentially in the order in which they are created and are read sequentially in the order in which they were created (or in sequentially reversed order if the REVERSED option of the OPEN statement is written for tape files). Such a file organization is referred to in this publication as standard sequential organization.

This type of data organization must be used for tape or unit-record files and may be used for files assigned to mass storage devices. No key is associated with records on a sequentially organized file.

Direct Data Organization

Direct data organization is characterized by the use of the relative track addressing scheme. When this addressing scheme is used, the positioning of the logical records in a file is determined by an ACTUAL KEY supplied by the user in the Environment Division. ACTUAL KEY is a key which is used to locate a logical record of the file. The first portion is the track identifier, which specifies the track (relative to the first track for a file) on which space to place a record is sought, or at which the search for a record is to begin. The second portion is the record identifier, which is a symbolic identifier for the record. Files with direct data organization must be assigned to mass storage devices.

Relative Data Organization

Relative data organization is characterized by the use of the relative record addressing scheme. When this addressing scheme is used, the position of the logical records in a file is determined relative to the first record of the file starting with the initial value of zero. A NOMINAL KEY is used to identify randomly accessed records. Files with relative data organization must be assigned to mass storage devices.

Indexed Data Organization

When indexed data organization is used, the position of each logical record in a file is determined by indexes created with the file and maintained by the system. The indexes are based on keys provided by the user. Indexed files must be assigned to mass storage devices.

ACCESS METHODS

Two access methods are available to users of Operating System COBOL: sequential access and random access.

Sequential access is the method of reading and writing records of a file in a serial manner; the order of reference is implicitly determined by the position of a record in the file.

Random access is the method of reading and writing records in a programmer-specified manner; the control of successive references to the file is expressed by specifically defined keys supplied by the user.

ACCESSING A SEQUENTIAL FILE

A standard sequential file may only be accessed sequentially, i.e., records are read or written in order. Records can be created and retrieved; for standard sequential files on mass storage devices, records can also be updated.

ACCESSING A DIRECT FILE

Direct files may be accessed both sequentially and randomly. Records can be created and retrieved sequentially; they can be created, retrieved, updated, and added randomly.

Sequential Access

When a direct file is being read sequentially, records are retrieved in logical sequence; this logical sequence corresponds exactly to the physical sequence of the records. Dummy records, if present, are also made available.

When a direct file is being read sequentially, the ACTUAL KEY clause may be specified. The track identifier (representing the relative track number) is not changed. The symbolic identifier for the record is placed in the record-identifier portion of ACTUAL KEY, except when an input/output error occurs.

A direct file may be created sequentially, and the ACTUAL KEY clause is required for this type of processing. Data is written sequentially. When the user wishes to switch tracks, he must add a number equal to the number of the tracks to be advanced to the track number portion of the ACTUAL KEY field.

COBOL will add dummy (recording mode F) or capacity (recording mode U, V, or S) records to complete the previous track(s). A relative track address of zero in the ACTUAL KEY field corresponds to the first track assigned to the file. If the initial value is not zero, COBOL will complete the intervening tracks with dummy or capacity records and write the first record on the track indicated by the ACTUAL KEY. When no more space is available on the specified track, the compiler generates coding to advance to the next track by adding one to the track address portion of the ACTUAL KEY. Data management will automatically replace the dummy or capacity records when additions are made to the file. At the time that the file is closed, dummy or capacity records are added to the current track and all following tracks, as determined by the TRACK-LIMIT clause (see "TRACK-LIMIT Clause" in "Input-Output Section"). When a unit of a multivolume file is closed, the tracks which have been allocated on the current unit are initialized with dummy or capacity records before the next unit is made available.

After a WRITE, CLOSE, or CLOSE UNIT, the relative track number for the last WRITE (of a data, dummy, or capacity record) is placed in the first four bytes of the ACTUAL KEY by the compiler.

Dummy records are identified by the figurative constant HIGH-VALUE in the fifth position of the ACTUAL KEY. If no ACTUAL KEY is specified, dummy records are not identifiable.

Random Access

When a direct file is accessed randomly, the ACTUAL KEY clause is required.

When records are being retrieved from a direct file randomly, the ACTUAL KEY is used to determine the track and to locate a particular record on that track. When a match is found, the data portion of the record is read, or, for a rewrite operation, replaced by a new record. The specified track is the only one searched for the desired record. If the desired record cannot be found on the specified track, the search can be extended to a specific number of tracks, or to the entire file by a DD card option (see the Programmer's Guide).

For a WRITE operation, after locating the track, the system searches for the last record on the track, and writes the new record (with control fields including a key field equal to the identifier found within the ACTUAL KEY field) after the last record. If the required space cannot be found on the specified track, the search can be extended to include a specific number of tracks, or to include the entire file by a DD card option.

When a direct file is being created, all the tracks of the file are initialized at open time with capacity records (mode U, V, or S) or dummy records (mode F). The number of tracks to be initialized is determined by the TRACK-LIMIT clause, or by the SPACE parameter in the DD card if the clause is omitted. Therefore, a WRITE statement issued for an output file is processed in the same manner as a WRITE statement that adds a record to an I-O file.

Appendix B contains an example of a program to create a direct file; Figure 2, in the Introduction, contains an example of a program to update a direct file.

ACCESSING A RELATIVE FILE

A relative file may be accessed either sequentially or randomly. Records can be created, retrieved, updated, and added sequentially; they can be retrieved, updated, and added randomly.

Sequential Access

A relative file may be created sequentially only. When a relative file is being created, the NOMINAL KEY clause may be specified. The compiler adds dummy records to complete the last track of the file when it is closed and to initialize the allocated tracks on the current volume when a CLOSE UNIT is executed. The relative block number of the last record written is placed in the NOMINAL KEY after a WRITE, CLOSE, or CLOSE UNIT, if the key is specified. If the NOMINAL KEY is specified, and the value in the NOMINAL KEY for a WRITE is greater than the next sequential relative block number, the necessary number of dummy records is written by the compiler so that the actual record is written in the specified relative block position.

Dummy records are identified by the presence of the figurative constant HIGH-VALUE in the first position of the record. The user can add dummy records by writing a record with HIGH-VALUE in the first position of the record. When the key is not specified, the user must write dummy records himself, except for those written by the compiler during the execution of a CLOSE or CLOSE UNIT statement.

When a relative file is being read sequentially, the records are made available in the order in which the records were written. Dummy records are also made available.

Random Access

To retrieve or update a relative file randomly, the NOMINAL KEY clause is required in the Environment Division. (The NOMINAL KEY contains the position of the record relative to the beginning of the file, starting with an initial value of zero.)

The records are retrieved on the basis of the NOMINAL KEY. Records can be updated by reading a record into an area, updating it in that area, and rewriting it from the same area. Records may not be added to the file except by replacement of dummy records created by the user or the compiler.

ACCESSING AN INDEXED FILE

An indexed file may be accessed either sequentially or randomly. Records can be created, retrieved, updated, and added sequentially; they can be retrieved, updated, and added randomly.

Sequential Access

An indexed file may be created sequentially only. When creating an indexed file, the RECORD KEY clause must be specified. It is used to indicate the location of the key within the record itself. Records appear in the file in the order in which they are written.

Room may be reserved for the insertion of new records by writing records with HIGH-VALUE in the first byte. These records are not made available in a sequential retrieval.

To retrieve or update an indexed file sequentially, the RECORD KEY clause must be specified. If record retrieval is to begin with other than the first record, the NOMINAL KEY clause must be specified, and a START statement must be executed before the first READ statement. Records are read in the order in which they were placed on the file previously. Logically, this corresponds to the sequence of keys, which must be in collating sequence at the time the file is created. The START statement can also be used to initiate the access of a segment of the file when processing sequentially. More than one START statement may be used in a program.

Random Access

To retrieve or update an indexed file randomly, both the NOMINAL KEY and RECORD KEY clauses are required. A record is considered "found" when the NOMINAL KEY is equal to the value of the RECORD KEY for the record. When adding or updating a record in a randomly accessed indexed file, the value in the RECORD KEY position must be identical to that of the NOMINAL KEY field.

Dummy records (records with HIGH-VALUE in the first byte) are made available in a random retrieval.

Dummy records forced off the primary area by random addition of records are physically deleted and are not written in the overflow area. (For a discussion of primary and overflow areas, see the Programmer's Guide.)

Appendix B contains examples of programs to create, retrieve, and update indexed files.

Table 3. File-processing Techniques

Data Management Technique	Device Type	Access	Organization
QSAM	Reader	[SEQUENTIAL]	standard sequential
QSAM	Punch	[SEQUENTIAL]	standard sequential
QSAM	Printer	[SEQUENTIAL]	standard sequential
QSAM	Tape	[SEQUENTIAL]	standard sequential
QSAM	Mass storage	[SEQUENTIAL]	standard sequential
BSAM	Mass storage	[SEQUENTIAL]	direct
BDAM	Mass storage	RANDOM	direct
QISAM	Mass storage	[SEQUENTIAL]	indexed
BISAM	Mass storage	RANDOM	indexed
BSAM	Mass storage	[SEQUENTIAL]	relative
BDAM	Mass storage	RANDOM	relative

ORGANIZATION OF THE ENVIRONMENT DIVISION

The Environment Division must begin in Area A, with the heading ENVIRONMENT DIVISION followed by a period.

The Environment Division is divided into two sections: the Configuration Section and the Input-Output Section. The sections and paragraphs, when written, must appear in the sequence shown.

Structure of the Environment Division

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER paragraph

OBJECT-COMPUTER paragraph

[SPECIAL-NAMES paragraph]

[INPUT-OUTPUT SECTION.

FILE-CONTROL paragraph

[I-O-CONTROL paragraph]]

ENVIRONMENT DIVISION -- CONFIGURATION SECTION

The Configuration Section deals with the overall specifications of computers. It is divided into three paragraphs: the SOURCE-COMPUTER paragraph, which describes the computer on which the source program is compiled; the OBJECT-COMPUTER paragraph, which describes the computer on which the program is executed; and, optionally, the SPECIAL-NAMES paragraph, which relates the function-names used by the compiler to mnemonic-names specified in the source program by the user.

General Format

```
CONFIGURATION SECTION.
SOURCE-COMPUTER.  source-computer-entry
OBJECT-COMPUTER.  object-computer-entry
[SPECIAL-NAMES.  special-names-entry]
```

Section-names and paragraph-names must begin in Area A.

The Configuration Section and its associated paragraphs are optional within a COBOL source program.

SOURCE-COMPUTER Paragraph

The SOURCE-COMPUTER paragraph serves only as documentation and describes the computer upon which the program is to be compiled.

General Format

```
SOURCE-COMPUTER.  computer-name.
```

Computer-name is IBM-360[-model-number] or IBM-370[-model-number].

The SOURCE-COMPUTER paragraph is treated as comments by the COBOL compiler.

OBJECT-COMPUTER Paragraph

The OBJECT-COMPUTER paragraph describes the computer on which the program is to be executed.

General Format

```
OBJECT-COMPUTER.  computer-name

    [MEMORY SIZE integer {WORDS     }  ]
                         {CHARACTERS}
                         {MODULES   }

    [SEGMENT-LIMIT IS priority-number].
```

Computer-name is IBM-360[-model-number]. Computer-name must be the first entry in the OBJECT-COMPUTER paragraph.

If the configuration implied by computer-name comprises more or less equipment than is actually needed by the object program, the MEMORY SIZE clause permits the specification of the actual subset (or superset) of the configuration.

With the exception of the SEGMENT-LIMIT clause, both the SOURCE-COMPUTER and OBJECT-COMPUTER paragraphs are treated as comments by the COBOL compiler.

The SEGMENT-LIMIT clause is discussed in "Segmentation."

Program Product Information (Version 3 and Version 4)

Computer-name may also be specified as IBM-370[-model-number]. If IBM-370 is specified, System/370 instructions are generated by the compiler. When IBM/370 is specified, the object program must be executed on a System/370 machine.

SPECIAL-NAMES Paragraph

The SPECIAL-NAMES paragraph provides a means of relating function-names to user-specified mnemonic-names. The SPECIAL-NAMES paragraph can also be used to exchange the functions of the comma and the period in the PICTURE character string and in numeric literals. In addition, the user may specify a substitution character which then must be used in place of the currency sign ($) in the PICTURE character string.

```
┌──────────────────────────────────────────────────────────────────┐
│                          General Format                          │
├──────────────────────────────────────────────────────────────────┤
│                                                                  │
│   SPECIAL-NAMES.                                                 │
│                                                                  │
│       [function-name IS mnemonic-name] ...                       │
│                                                                  │
│       [CURRENCY SIGN IS literal]                                 │
│                                                                  │
│       [DECIMAL-POINT IS COMMA].                                  │
│                                                                  │
└──────────────────────────────────────────────────────────────────┘
```

When the SPECIAL-NAMES paragraph is specified, the comma or the semicolon may optionally be used to separate successive entries; there must be one and only one period, placed at the end of the paragraph.

Function-name may be chosen from the following list:

SYSOUT
SYSIN
SYSPUNCH
CONSOLE
C01 through C12
CSP
S01
S02
literal

If SYSIN, SYSOUT, SYSPUNCH, or CONSOLE is specified, the associated mnemonic-name may be used in ACCEPT and DISPLAY statements.

If C01 through C12, CSP, S01, or S02 is specified, the associated mnemonic-names may be used in a WRITE BEFORE/AFTER ADVANCING statement. These function-names are the carriage control characters shown in Table 4.

Table 4. Choices of Function-name and Action Taken

Function-name	Action Taken
CSP	Suppress spacing.
C01 through C09	Skip to channel 1 through 9, respectively.
C10 through C12	Skip to channel 10, 11, 12, respectively.
S01, S02	Pocket select 1 or 2, on the IBM 1442, and P1 and P2 on the IBM 2540.

The choice of literal indicates that function-name is to be used to identify Report Writer output. The mnemonic-name should appear in a CODE clause in a Report Description entry (RD) (see "Report Writer"). One such special-name entry may be given for each Report defined in a source program. The literal must be a one-character nonnumeric literal.

The CURRENCY SIGN clause specifies the literal that is used in the PICTURE clause to represent the currency symbol. The literal must be

nonnumeric and is limited to a single character which must not be any of the following:

1. Digits 0 through 9
2. Alphabetic characters A, B, C, D, P, R, S, V, X, Z, or the space.
3. Special characters * - , . ; ( ) + " or '

If the CURRENCY SIGN clause is not present, only the $ can be used as the currency symbol ($) in the PICTURE clause.

The DECIMAL-POINT IS COMMA clause means that the function of the comma and the period are exchanged in the PICTURE character string and in numeric literals. When this clause is written, the user must represent the decimal point, when required in a numeric literal or in the PICTURE clause, by a comma (,); the period must be used for the functions ordinarily served by the comma.

ENVIRONMENT DIVISION -- INPUT-OUTPUT SECTION

The Input-Output Section deals with the definition of each file, the identification of its external storage media, the assignment of the file to one or more input/output devices, and also deals with information needed for the most efficient transmission of data between the media and the object program. The section is divided into two paragraphs: the FILE-CONTROL paragraph, which names and associates the files used in the program with the external media; and the I-O-CONTROL paragraph, which defines special input/output techniques.

General Format

```
[INPUT-OUTPUT SECTION.
FILE-CONTROL.  {file-control-entry}
[I-O-CONTROL.  input-output-control-entry]]
```

FILE-CONTROL PARAGRAPH

Information that is used or developed by the program may be stored externally. File description entries in the Data Division name the files into which the information is arranged and specify their physical characteristics. The FILE-CONTROL paragraph assigns the files (by the names given in the file description entries) to input/output devices.

General Format

```
FILE-CONTROL.
   {SELECT Clause
   ASSIGN Clause
   [RESERVE Clause]
   [FILE-LIMIT Clause]
   [ACCESS MODE Clause]
   [PROCESSING MODE Clause]
   [ACTUAL KEY Clause]
   [NOMINAL KEY Clause]
   [RECORD KEY Clause]
   [TRACK-AREA Clause]
   [TRACK-LIMIT Clause].} ...
```

Each SELECT sentence must begin with a SELECT clause followed immediately by an ASSIGN clause; the order in which the optional clauses are written is not significant.

SELECT Clause

The SELECT clause is used to name each file in a program.

Format

```
SELECT [OPTIONAL] file-name
```

Each file described in the Data Division must be named once and only once as a file-name following the key word SELECT. Each file named in a SELECT clause must have a file description (FD) entry or sort-file description (SD) entry in the Data Division of the source program.

The key word OPTIONAL may be specified only for input files accessed sequentially. It is required for input files that are not necessarily present each time the object program is executed. When a file is not present at object time, the first READ statement for that file causes control to be passed to the imperative-statement following the key words AT END. However, OPTIONAL need not be specified and will be treated as a comment, since this function is performed by the operating system through the DD statement with the DUMMY or NULLFILE parameter.

ASSIGN Clause

The ASSIGN clause is used to assign a file to an external medium.

Format

```
ASSIGN TO [integer-1] system-name-1
          [system-name-2]...
            [FOR MULTIPLE { REEL } ]
                          { UNIT }
```

Integer-1 indicates the number of input/output units of a given medium assigned to file-name. However, since the number of units is automatically determined by the operating system, the integer-1 option need not be specified. When specified, it is treated as comments (see IBM System/360 Operating System: Job Control Language, Form GC28-6539).

System-name specifies a device class, a particular input/output device, the organization of data upon this device, and the external-name of the file. All files used in a program must be assigned to an input/output medium. Any system-name beyond the first for a file will be treated as comments.

FOR MULTIPLE REEL/UNIT is applicable whenever the number of tape units or mass storage devices assigned might be less than the number of reels or units in the file. The operating system will automatically

handle volume switching for sequentially processed files. All volumes must be mounted for randomly accessed files. Therefore, when this clause is specified, it is treated as comments.

System-name has the following structure:

class[-device]-organization-name

Class is a 2-character field that specifies the device class;

DA (mass storage)
UT (utility)
UR (unit-record)

Files assigned to UT or UR must have standard sequential organization and can be accessed only sequentially. Files assigned to DA may have standard sequential or direct organization. When organization is direct, access may be either sequential or random.

Files assigned to DA may also have relative or indexed organization. When organization is relative or indexed, access may either be sequential or random.

Device is used to specify a particular device within a device class. It can be a 4- to 6-character field. If device independence for a file is desired, the device class must be UT, no device number may be specified, and no END-OF-PAGE clauses may be associated with the file. At execution time, such a file may be assigned to any device class (including unit-record).

The allowable system devices for any given class are as follows:

Mass storage (DA) 2301, 2302, 2303, 2311, 2314, 2321.

Utility (UT) 2301, 2302, 2311, 2314, 2321, 2400.

Unit-record (UR) 1403, 1404 (for continuous forms only), 1442R, 1442P, 1443, 1445, 2501, 2520R, 2520P, 2540R, 2540P. (R indicates reader; P indicates punch.)

Note: Sort input, output, or work files may be assigned to any utility device except device number 2321 (see "Sort").

Program Product Information (Version 3)

For Version 3 only, the following additional system devices are allowable:

Mass Storage (DA) 2305-1, 2305-2, 2319, 3330

Utility (UT) 2305-1, 2305-2, 2319, 3330

Unit Record (UR) 3211

Note: For the Version 1 and Version 2 Compilers, these devices (2305-1, 2305-2, 2319, 3330, or 3211) can be used, if the device field in system-name is omitted. At execution time, any of these devices can be specified through the UNIT subparameter of the file's DD statement. Note, however, that except for files containing spanned records the device field is treated as comments. For files containing spanned records, the block length for the file is checked against the maximum block length allowed for the device specified, and the smaller of the two becomes the block size that is used.

Program Product Information (Version 4)

The device field in system-name is treated as comments by the Version 4 Compiler. At execution time, any valid device can be specified through the UNIT subparameter of the file's DD statement. The following considerations apply:

- If an invalid device number is specified, no error diagnostic is produced.
- For an ASCII file, if 2400 (or other compatible tape device) is not specified in the device field, no error diagnostic is produced.
- For a direct file with spanned records, the Version 4 Compiler always calculates buffer size from the COBOL record description.

Organization is a 1-character field that indicates the file organization. The following characters must be used:

S for files with standard sequential organization
D for files with direct organization
W for files with direct organization when REWRITE is used. When the file is opened as INPUT or OUTPUT, however, W is the equivalent of D.
R for files with relative organization
I for files with indexed organization

Table 5 can be used to determine the correct choice for the organization field in system-name.

Name is a 1- to 8-character field specifying the external-name by which the file is known to the system. It is the name that appears in the name field of the DD card for the file.

Note: ASCII considerations for the ASSIGN clause are given in Appendix E.

Table 5. Values for the Organization Field for System-name

Device Type	ACCESS MODE Clause	File Organization	Organization Field
tape, punch reader, printer	[SEQUENTIAL]	standard sequential	S
mass storage device	[SEQUENTIAL]	standard sequential	S
mass storage device	[SEQUENTIAL]	direct	D
mass storage device	RANDOM	direct	D
mass storage device	RANDOM	direct (REWRITE)	W
mass storage device	[SEQUENTIAL]	relative	R
mass storage device	RANDOM	relative	R
mass storage device	[SEQUENTIAL]	indexed	I
mass storage device	RANDOM	indexed	I

RESERVE Clause

The RESERVE clause allows the user to modify the number of input/output areas (buffers) allocated by the compiler.

Format

```
RESERVE { NO      } ALTERNATE [ AREA  ]
        { integer }           [ AREAS ]
```

This clause specifies that the number of buffers represented by integer be reserved for a standard sequential file or an indexed file that is accessed sequentially, in addition to the one required buffer which is reserved automatically.

This clause must not be specified for direct or relative files; if specified, the clause is ignored and the one required buffer is reserved.

The number of additional buffers is represented by the value of integer which must not exceed 254. If NO is written, no additional buffers are reserved, aside from the standard minimum of one.

If the RESERVE clause is omitted, and the SAME AREA clause is used for the file, two areas are reserved. If the RESERVE clause is omitted, and the SAME AREA clause is not used for the file, the number of buffers assigned at execution time is taken from the DD card for the file. If no buffers are specified on the DD card, two areas are reserved.

FILE-LIMIT Clause

The FILE-LIMIT clause serves only as documentation, and is used to specify the logical beginning and the logical end of a file on a mass storage device.

Format

```
{FILE-LIMIT IS  }  {data-name-1}        {data-name-2}
{FILE-LIMITS ARE}  {literal-1  }  THRU  {literal-2  }

    [{data-name-3}        {data-name-4}    ]...
     {literal-3 }  THRU   {literal-4 }
```

The logical beginning of a mass storage file is the address specified by the first operand of the FILE-LIMIT clause; the logical end of a mass storage file is the address specified as the last operand of the FILE-LIMIT clause. Because file boundaries are determined at execution time from the operating system's control cards, this clause need not be specified and will be treated as comments.

ACCESS MODE Clause

The ACCESS MODE clause defines the manner in which records of a file are to be accessed.

Format

```
ACCESS MODE IS  {SEQUENTIAL}
                {RANDOM    }
```

If this clause is not specified, ACCESS IS SEQUENTIAL is assumed. For ACCESS IS SEQUENTIAL, records are placed or obtained sequentially. That is, the next logical record is made available from the file when a READ statement is executed, or the next logical record is placed into

the file when a WRITE statement is executed. ACCESS IS SEQUENTIAL may be applied to files assigned to tape, unit record, or mass storage devices.

For ACCESS IS RANDOM, storage and retrieval are on the basis of an ACTUAL or NOMINAL KEY associated with each record. When the RANDOM option is specified, the file must be assigned to a mass storage device. ACCESS IS RANDOM may be specified when file organization is direct, relative, or indexed.

The key word IS must be specified. However, this compiler allows the key word IS to be omitted.

PROCESSING MODE Clause

The PROCESSING MODE clause serves only as documentation, and indicates the order in which records are processed.

Format
PROCESSING MODE IS SEQUENTIAL

This clause may be omitted. When specified, it is treated as comments.

ACTUAL KEY Clause

When creating or retrieving records from a randomly accessed file, the programmer is responsible for providing the ACTUAL KEY for each record to be processed.

An ACTUAL KEY is a key that can be directly used by the system to locate a logical record on a mass storage device. The ACTUAL KEY is made up of two components:

1. The track identifier, which contains the relative track number at which the search is to start for a record or for a space in which to place a new record.
2. The record identifier, which serves as a unique symbolic identifier for the record and is associated with the record itself.

Format
ACTUAL KEY IS data-name

The ACTUAL KEY clause must be specified for direct files when ACCESS IS RANDOM is specified. The ACTUAL KEY field must be set to a desired value before the execution of the READ and WRITE statements.

When a READ statement is executed for a file, a specific logical record (located using the contents of data-name) is made available from that file.

When a WRITE statement is executed, a logical record is placed in the file at a location found through the use of the contents of data-name.

The ACTUAL KEY clause may be specified when reading direct files sequentially.

The ACTUAL KEY clause must be specified when creating a direct file with sequential access.

Data-name may be any fixed item from 5 through 259 bytes in length. It must be defined in the File, Working-Storage, or Linkage Section. However, if data-name is specified in the File Section, it may not be contained in the file for which it is the key. The following considerations apply:

1. The first four bytes of data-name are the track identifier and must be defined as a 5-integer binary data item whose maximum value does not exceed 65,535.

2. The remainder of data-name -- 1 through 255 bytes in length -- represents the record identifier. It is the user's responsibility to select from 1 through 255 bytes for the symbolic portion of the ACTUAL KEY field. Data within these bytes will be treated exactly as specified.

The key word IS must be specified. However, this compiler allows the key word IS to be omitted.

Sample coding to represent the data-name specified in the ACTUAL KEY clause would be as follows:

```
ENVIRONMENT DIVISION.
       .
       .
       .
   ACTUAL KEY IS THE-ACTUAL-KEY.
       .
       .
       .
DATA DIVISION.
       .
       .
       .
WORKING-STORAGE SECTION.
01  THE-ACTUAL-KEY.
       05  RELATIVE-TRACK-KEY USAGE COMPUTATIONAL PICTURE IS S9(5)
           VALUE IS 10 SYNCHRONIZED.
       05  EMPLOYEE-NO PICTURE IS X(6) VALUE IS LOW-VALUE.
```

RELATIVE-TRACK-KEY contains the relative track address at which the record is to be placed, or at which to search for the record. EMPLOYEE-NO serves as a unique identifier associated with the record itself and represents the record identifier of the key field.

NOMINAL KEY Clause

The NOMINAL KEY clause is used with indexed and relative files. For indexed files, the clause specifies a symbolic identity for a specific logical record. For relative files, the NOMINAL KEY clause specifies the relative record number for a specific logical record, relative to the beginning of the file.

Format
NOMINAL KEY IS data-name

A NOMINAL KEY clause is required when an indexed or relative file is accessed randomly. It is also required when an indexed file is accessed sequentially and a START statement is used. Tne NOMINAL KEY clause may also be specified when creating a relative file.

When the NOMINAL KEY clause is specified for an indexed file that is accessed randomly:

- Data-name may specify any fixed-length Working-Storage item from 1 through 255 bytes in length.
- Data-name must be at a fixed displacement from the beginning of the record description in which it appears; that is, it may not appear in the entry subsequent to an OCCURS DEPENDING ON clause.
- The symbolic identity of the record must be placed in data-name before the execution of the READ, WRITE, or REWRITE statement.
- The symbolic identity is used when retrieving or updating a record to locate the logical record with a matching RECORD KEY or, when adding a record, to create the key that will be associated with the record.
- When a READ statement is executed, a specific logical record is made available from the file using the contents of data-name.
- When a WRITE or REWRITE statement is executed, the symbolic identity of the record specified by data-name is used to determine the physical location at which the record is written.
- If the TRACK-AREA clause is not specified for the file, then after a WRITE statement is executed, the contents of the NOMINAL KEY field are unpredictable.

When the NOMINAL KEY clause is specified for an indexed file that is accessed sequentially:

- Data-name may specify any fixed-length Working-Storage item from 1 through 255 bytes in length.
- Data-name must be at a fixed displacement from the beginning of the record description in which it appears; that is, it may not appear in the entry subsequent to an OCCURS DEPENDING ON clause.
- The NOMINAL KEY clause must be specified if a Format 1 START statement is used. When the START statement is executed, the contents of data-name are used to locate the record at which processing is to begin. The next READ statement accesses this record.

When the NOMINAL KEY clause is used for a relative file that is either created or accessed randomly:

- Data-name may specify any 8-integer binary item in Working-Storage whose maximum value does not exceed 15,728,640.
- Data-name must be at a fixed displacement from the beginning of the record description in which it appears; that is, it may not appear in the entry subsequent to an OCCURS DEPENDING ON clause.
- The relative record number must be placed in data-name before the execution of the READ, WRITE, or REWRITE statement.
- When a READ statement is executed, a specific logical record is made available from the file using the contents of data-name.
- When a WRITE or REWRITE statement is executed, the relative record number is used to determine the physical location, relative to the beginning of the file, at which the record is written.

RECORD KEY Clause

A RECORD KEY is used to access an indexed file. It specifies the item within the data record that contains the key for the record.

Format
RECORD KEY IS data-name

The RECORD KEY clause must be specified for an indexed file.

Data-name may be any fixed-length item within the record. It must be less than 256 bytes in length.

When two or more record descriptions are associated with a file, a similar field must appear in each description, and must be in the same relative position from the beginning of the record, although the same data-name need not be used for both fields.

Data-name must be defined to exclude the first byte of the record in the following cases:

1. Files with unblocked records
2. Files from which records are to be deleted
3. Files whose keys might start with a delete-code character (HIGH-VALUE).

With these exceptions, the item specified by data-name may appear anywhere within the record.

TRACK-AREA Clause

This clause may be used optionally when records are to be added to an indexed file in the random access mode. Efficiency in adding a record is improved when the TRACK-AREA clause is specified.

Format

```
TRACK-AREA IS { data-name } CHARACTERS
              { integer   }
```

When records are to be added to random access files with indexed organization, this clause specifies either an area (data-name) or the size of an area (integer). The area is used to hold all the blocks on a track, including their count and key fields, plus one logical record.

The area defined by the TRACK-AREA clause must be a multiple of 8 and must not exceed 32,760 bytes.

When the integer option is specified, an area of integer bytes is obtained from the system when the file is opened. It is released to the system when the file is closed.

When the data-name option is specified, data-name must specify an item described with an 01 or 77 level number in the Working-Storage Section.

If a record is added to an indexed file, and the TRACK-AREA clause was not specified for the file, the contents of the NOMINAL KEY field are unpredictable after a WRITE statement is executed.

TRACK-LIMIT Clause

The TRACK-LIMIT clause indicates the relative number of the last track to be initialized for the creation of files with direct organization.

Format

```
TRACK-LIMIT IS integer [ TRACK  ]
                       [ TRACKS ]
```

This clause does not cause track allocation, which is the function of a DD card parameter.

Sequential Access -- When used in conjunction with ACCESS IS SEQUENTIAL and a file opened as OUTPUT, if the last relative track number used by the file when it is closed is less than that specified in the TRACK-LIMIT clause, the unused portion(s) of the track(s) is filled with capacity records (mode U, V, or S) or with dummy records (mode F). If the last relative track number used by the file is equal to or

greater than that specified in the TRACK-LIMIT clause, or if the clause is omitted, a capacity record or dummy records is written for the current track of the file, and the remaining allocated tracks are not initialized. Note that since the first relative track is track 0, at least integer plus one track will be initialized.

Random Access -- When used in conjunction with ACCESS IS RANDOM, the TRACK-LIMIT clause specifies the last relative track number to be initialized at open time; the tracks are initialized with dummy (mode F) or capacity (modes U, V, or S) records. This defines the total size of the file; that is, no additional tracks may be used by the file, and any references to tracks outside this area will result in an INVALID KEY condition. If this clause is omitted, the number of tracks initialized is determined from the SPACE and VOLUME count parameters of the DD card. The first volume will be initialized according to the primary allocation quantity, and succeeding volumes (if any) will be initialized from the secondary quantity (one quantity per volume).

I-O-CONTROL PARAGRAPH

The I-O-CONTROL paragraph defines some of the special techniques to be used in the program. It specifies the points at which checkpoints are to be established, the core storage area which is to be shared by different files, the location of files on multiple-file reels, and optimization techniques. The I-O-CONTROL paragraph and its associated clauses are an optional part of the Environment Division.

Format

```
I-O-CONTROL.
     [RERUN Clause] ...
     [SAME AREA Clause] ...
     [MULTIPLE FILE TAPE Clause] ...
     [APPLY Clause] ...  .
```

The order in which the I-O-CONTROL paragraph clauses are written is not significant.

RERUN Clause

The presence of a RERUN clause specifies that checkpoint records are to be taken. A checkpoint record is a recording of the status of a problem program and main storage resources at desired intervals. The contents of core storage are recorded on an external storage device at the time of the checkpoint, and can be read back into core storage to restart the program from that point.

Format 1

```
RERUN ON system-name

                 ( integer    RECORDS )
      EVERY      (                    ) OF file-name
                 ( [END OF]  ( REEL  ) )
                 (           ( UNIT  ) )
```

Format 2

```
RERUN ON system-name.
```

Checkpoint records are written sequentially and must be assigned to tape or mass storage devices.

System-name specifies the external medium for the checkpoint file, the file upon which checkpoint records are written. It must not be the same as any system-name used in a File-Control ASSIGN clause, but it follows the same rules of formation. System-name must specify a tape or mass storage device.

File-name represents the file for which checkpoint records are to be written. It must be described with a file description entry in the Data Division.

FORMAT 1: More than one Format 1 RERUN clause may be specified in a program. If multiple RERUN clauses are specified, they may be specified either for the same or for different checkpoint files.

There are two options of the Format 1 RERUN clause. Each may be specified once for any given file-name.

RECORDS Options: This option is valid for sequentially or randomly accessed files. It specifies that a checkpoint record is to be written for every integer records of file-name processed.

The value of integer must not exceed 16,777,215.

Program Product Information (Version 3 and Version 4)

END OF REEL/UNIT Option: This option is valid only for sequentially accessed files with any organization. It specifies that a checkpoint record is to be written whenever end-of-volume for file-name occurs. Normal volume closing procedures are also performed. END OF REEL is valid only for tape files; END OF UNIT is valid only for sequentially accessed files residing on mass storage devices.

However, in order to achieve device independence, this compiler allows the terms REEL and UNIT to be used interchangeably.

FORMAT 2: Format 2 is used for taking checkpoint records for sort files, and is described in "I-O-CONTROL Paragraph" in the chapter on the Sort Feature.

Note: ASCII considerations for the RERUN clause are given in Appendix E.

SAME Clause

The SAME clause specifies that two or more files are to use the same core storage area during processing.

Format

```
SAME [ RECORD ] AREA FOR file-name-1 {file-name-2} ...
     [ SORT   ]
```

The discussion that follows pertains only to SAME AREA and SAME RECORD AREA. The SAME clause with the SORT option is discussed in "Sort."

The SAME RECORD AREA clause specifies that two or more files are to use the same main storage area for processing the current logical record. All of the files may be open at the same time. A logical record in the shared storage area is considered to be:

- a logical record of each opened output file in this SAME RECORD AREA clause, and
- a logical record of the most recently read input file in this SAME RECORD AREA clause.

If the SAME AREA clause does not contain the RECORD option, the area being shared includes all storage areas assigned to the files; therefore, it is not valid to have more than one of these files open at one time.

More than one SAME clause may be included in a program; however:

1. A specific file-name must not appear in more than one SAME AREA clause.
2. A specific file-name must not appear in more than one SAME RECORD AREA clause.
3. If one or more file-names of a SAME AREA clause appear in a SAME RECORD AREA clause, all of the file-names in that SAME AREA clause must appear in that SAME RECORD AREA clause. However, that SAME RECORD AREA clause may contain additional file-names that do not appear in that SAME AREA clause.

Note: For a direct file with mode S records, the program is device-dependent if both the SAME AREA clause and the device field of system-name are specified. The compiler then determines the segment

work area as either the track capacity of the device specified, or as 8 + logical-record-length, whichever is smaller.

If the SAME AREA clause is specified, and the device field of system-name is not specified, the compiler calculates the segment work area as 8 + logical-record-length, no matter which device is used.

If neither the SAME AREA clause nor the device field is specified, then at execution time the segment work area is calculated as either the track area of the device assigned, or 8 + logical-record-length, whichever is smaller.

Program Product Information (Version 3 and Version 4)

If the BLOCK CONTAINS 0 and/or the RECORD CONTAINS 0 clauses are specified, then the SAME AREA clause may not be specified.

MULTIPLE FILE TAPE Clause

The MULTIPLE FILE TAPE clause is used for documentation purposes and indicates that two or more files share the same physical reel of tape.

Format
MULTIPLE FILE TAPE CONTAINS file-name-1 [POSITION integer-1] [file-name-2 [POSITION integer-2]]...

The MULTIPLE FILE TAPE clause is required when more than one file shares the same physical reel of tape.

However, this compiler treats the MULTIPLE FILE TAPE clause as comments, since this function is performed by the system through the LABEL parameter of the DD statement (see the Programmer's Guide).

APPLY Clause

There are several options of the APPLY clause. More than one of each option may appear.

Format for Option 1
APPLY WRITE-ONLY ON file-name-1 [file-name-2] ...

This option is used to make optimum use of buffer and device space allocated when creating a file whose recording mode is V. Normally, a buffer is truncated when there is not enough space remaining in it to accommodate the maximum size record. Use of this option will cause a buffer to be truncated only when the next record does not fit in the unused remainder of the buffer. This option has meaning only when the file is opened as OUTPUT.

The files named in this option must have standard sequential organization.

Every WRITE statement associated with the file must use the WRITE record-name FROM identifier option. None of the subfields of record-name may be referred to in procedural statements, nor may any of the subfields be the object of an OCCURS DEPENDING ON clause.

However, if the same file is opened for INPUT or I-O, the subfields of the record-name may be referred to. When the same file is opened for I-O, the WRITE statement must not be used; the REWRITE statement must be used in its place.

Format for Option 2
APPLY CORE-INDEX ON file-name-1 [file-name-2]...

This option may be specified only for an indexed file whose access mode is random. It is used to specify that the highest level index is to be processed in core. The area will be obtained at open time and released at close time.

Format for Option 3
APPLY RECORD-OVERFLOW ON file-name-1 [file-name-2] ...

If the record overflow feature is available for the mass storage device being used, the amount of unused space on a volume may be reduced by specifying this option for files on that volume. If the option is used, a block that does not fit on the track is partially written on that track and continued on the next available track.

This option may be specified only for a standard sequential file (with F, U, or V mode records) assigned to a mass storage device, or a direct file with fixed-length records.

Format for Option 4
APPLY REORG-CRITERIA TO data-name ON file-name

If the "reorganization criteria" feature was specified on the DD card for an indexed file when it was created, this option may be specified for the file when ACCESS IS RANDOM is specified.

The reorganization statistics maintained by the system will be placed in data-name when a CLOSE statement is executed for the file. Data-name must be composed of three COMPUTATIONAL items of 2, 2, and 4 bytes in length, respectively.

The first 2 bytes will contain the number of cylinder overflow areas that are full. The second 2 bytes will contain the number of tracks (partial or whole) remaining in the independent overflow area. The last 4 bytes will contain the number of READ or WRITE statements that accessed overflow records that are not the first in the chain of such records.

PART III -- DATA DIVISION

- DATA DIVISION -- INTRODUCTION
- ORGANIZATION OF THE DATA DIVISION
- FILE DESCRIPTION ENTRY -- DETAILS OF CLAUSES
- DATA DESCRIPTION
- DATA DESCRIPTION -- DETAILS OF CLAUSES

DATA DIVISION -- INTRODUCTION

The Data Division of a COBOL source program contains the description of all information to be processed by the object program. Two types of data may be processed by a COBOL program: information recorded externally on files and information created internally. The second type, which exists only during the execution of a program, will be discussed later in this chapter in "Working-Storage Section."

ORGANIZATION OF EXTERNAL DATA

A file is a collection of records. There are two types of records: physical records and logical records. A physical record is a group of characters or records which is treated as an entity when moved into or out of core storage. A logical record is a number of related data items. It may itself be a physical record, i.e., contained within a single physical unit, it may be one of several logical records contained within a single physical record, or it may extend across physical units.

COBOL source language statements provide the means of describing the relationship between physical and logical records. Once this relationship is established, only logical records are made available to the COBOL programmer. Hence, in this manual, a reference to records means logical records unless the term "physical records" is used.

DESCRIPTION OF EXTERNAL DATA

In the discussion of data description, a distinction must first be made between a record's external description and its internal content.

External description refers to the physical aspects of a file, i.e., the way in which the file appears on an external medium. For example, the number of logical records per physical record describes the grouping of records in the file. The physical aspects of a file are specified in file description entries.

A COBOL record usually consists of groups of related information that are treated as an entity. The explicit description of the contents of each record defines its internal characteristics. For example, the type of data to be contained within each field of a logical record is an internal characteristic. This type of information about each field of a particular record is grouped into a record description entry.

ORGANIZATION OF THE DATA DIVISION

The Data Division is divided into four sections: the File Section, the Working-Storage Section, the Linkage Section, and the Report Section.

All data that is stored externally, for example, on magnetic tape, must be described in the File Section before it can be processed by a COBOL program. Information that is developed for internal use must be described in the Working-Storage Section. Information passed from one program to another must be described in the Linkage Section. The content and format of all reports that are to be generated by the Report Writer feature must be described in the Report Section.

The Data Division is identified by, and must begin with, the header DATA DIVISION. The File Section is identified by, and must begin with, the header FILE SECTION. The header is followed by one or more file description entries and one or more associated record description entries. The Working-Storage Section is identified by, and must begin with, the header WORKING-STORAGE SECTION. The header is followed by data item description entries for noncontiguous items, followed by record description entries. The Linkage Section is identified by, and must begin with, the header LINKAGE SECTION. The header is followed by noncontiguous data item description entries, followed by record description entries. The Report Section is identified by, and must begin with, the header REPORT SECTION. The header is followed by one or more report description entries, and one or more report group description entries.

Program Product Information (Version 4)

In Version 4, a fifth section, the Communication Section, contains information about the interface between the COBOL TP program and the user written TCAM Message Control Program. The Communication Section is identified by, and must begin with the header COMMUNICATION SECTION. The header is followed by one or more communication description entries, each optionally followed by one or more record description entries. In the Data Division, the Communication Section must be written after the Linkage Section and before the Report Section. (See the Teleprocessing chapter.)

For the proper formats of Division and Section headers, see "Use of the COBOL Coding Form" in "Language Considerations."

Structure of the Data Division

```
DATA DIVISION.

FILE SECTION.

{file description entry

{record description entry}...}...

WORKING-STORAGE SECTION.

[data item description entry]...

[record description entry]...
```

```
LINKAGE SECTION.

[data item description entry]...

[record description entry]...

COMMUNICATION SECTION.              (Version 4)

{communication description entry    (Version 4)

[record description entry]...}...   (Version 4)

REPORT SECTION.

{report description entry

{report group description entry}...}...
```

Each of the sections of the Data Division is optional and may be omitted from the source program when the section is unnecessary. When used, the sections must appear in the foregoing sequence.

ORGANIZATION OF DATA DIVISION ENTRIES

Each Data Division entry begins with a level indicator or a level number, followed by one or more spaces, followed by the name of a data item (except in the Report Section), followed by a sequence of independent clauses describing the data item. The last clause is always terminated by a period followed by a space.

There are two types of Data Division entries: those that begin with a level indicator and those that begin with a level number.

Level Indicator

The level indicator FD is used to specify the beginning of a file description entry. When the file is a sort-file, the level indicator SD must be used instead of FD (see "Sort"). When a report is to be generated by the Report Writer feature, the level indicator RD, specifying the beginning of a report description entry, must be provided for each report, in addition to the FD for the file from which the report is generated (see "Report Writer"). Figure 4 summarizes the level indicators.

Indicator	Use
FD	file description entries
SD	sort-file description entries
CD	communication description entries (Version 4)
RD	report description entries

Figure 4. Level Indicator Summary

Each level indicator must begin in Area A and be followed in Area B by its associated name and appropriate descriptive information.

Level indicators are illustrated in the sample COBOL programs found in Appendix B.

Level Number

Level numbers are used to structure a logical record to satisfy the need to specify subdivisions of a record for the purpose of data reference. Once a subdivision has been specified, it may be further subdivided to permit more detailed data reference.

The basic subdivisions of a record, that is, those not further subdivided, are called elementary items; consequently, a record may consist of a sequence of elementary items, or the record itself may be an elementary item.

In order to refer to a set of elementary items, the elementary items are combined into groups. A group item consists of a named sequence of one or more elementary items. Groups, in turn, may be combined into larger groups. Thus, an elementary item may belong to more than one group. In the following example, the group items MARRIED and SINGLE are themselves part of a larger group named RETIRED-EMPLOYEES:

```
02  RETIRED-EMPLOYEES.
    03  MARRIED.
        04  NO-MALE PICTURE 9(8).
        04  NO-FEMALE PICTURE 9(8).
    03  SINGLE.
        04  NO-MALE PICTURE 9(8).
        04  NO-FEMALE PICTURE 9(8).
```

A system of level numbers shows the organization of elementary items and group items. Since records are the most inclusive data items, the level number for a record must be 1 or 01. Less inclusive data items are assigned higher (not necessarily successive) level numbers not greater than 49. There are special level numbers -- 66, 77, and 88 -- which are exceptions to this rule. Separate entries are written in the source program for each level number used.

A group includes all group and elementary items following it until a level number less than or equal to the level number of that group is encountered. The level number of an item which immediately follows the last elementary item of the previous group must be equal to the level number of one of the groups to which a prior elementary item belongs.

Standard

```
01  A.
    05  C-1.
        10  D PICTURE X.
        10  E PICTURE X.
    05  C-2.
     .
     .
     .
```

Nonstandard

```
01  A.
    05  C-1.
        10  D PICTURE X.
        10  E PICTURE X.
    04  B-1.
     .
     .
     .
```

In the foregoing example, the compiler will accept the nonstandard use of 04 and treat it as though it had been written as an 05.

Level numbers 01 and 77 must begin in Area A, followed in Area B by associated data-names and appropriate descriptive information. All other level numbers may begin in either Area A or in Area B, followed in Area B by associated data-names and appropriate descriptive information.

A single-digit level number is written either as a space followed by a digit or as a zero followed by a digit. At least one space must separate a level number from the word following the level number.

Special Level Numbers

Three types of data exist whose level numbers are not intended to structure a record:

66: Names of elementary items or groups described by a RENAMES clause for the purpose of regrouping data items have been assigned the special level number 66. For an example of the function of the RENAMES clause, see "Data Description."

77: Noncontiguous Working-Storage items, which are not subdivisions of other items and are not themselves subdivided, have been assigned the special level number 77.

88: Entries that specify condition-names to be associated with particular values of a conditional variable have been assigned the special level number 88. For an example of level-88 items, see "Data Description."

Indentation

Successive data description entries may have the same format as the first such entry or may be indented according to level number. Indentation is useful for documentation purposes and does not affect the action of the compiler.

FILE SECTION

The File Section contains a description of all externally stored data (FD), and a description of each sort-file (SD) used in the program.

The File Section must begin with the header FILE SECTION followed by a period. The File Section contains file description entries and sort-file description entries, each one followed by its associated record description entry (or entries).

General Format

```
FILE SECTION.
{file description entry
{record description entry} ...}...
```

File Description Entry

In a COBOL program, the File Description Entries (FD and SD) represent the highest level of organization in the File Section. The file description entry provides information about the physical structure and identification of a file, and gives the record-name(s) associated with that file.

For a complete discussion of the sort-file-description entry, see "Sort."

Record Description Entry

The Record Description Entry consists of a set of data description entries which describe the particular record(s) contained within a particular file. For a full discussion of the format and the clauses required within the record description entry, see "Data Description."

WORKING-STORAGE SECTION

The Working-Storage Section may contain descriptions of records which are not part of external data files but are developed and processed internally.

The Working-Storage Section must begin with the section header WORKING-STORAGE SECTION followed by a period. The Working-Storage Section contains data description entries for noncontiguous items and record description entries, in that order.

General Format

```
WORKING-STORAGE SECTION.

[data item description entry] ...

[record description entry] ...
```

Data Item 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 of these items is defined in a separate data item description entry that begins with the special level number 77.

Record Description Entries

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

LINKAGE SECTION

The Linkage Section describes data made available from another program. It is also used to describe data from the PARM field of the EXEC statement, which is made available to a main program at execution time (see "Subprogram Linkage").

General Format
LINKAGE SECTION. [data item description entry] ... [record description entry] ...

Data item description entries and record description entries in the Linkage Section provide names and descriptions, but storage within the program is not reserved since the data area exists elsewhere. Any data description clause may be used to describe items in the Linkage Section, with one exception: the VALUE clause may not be specified for other than level-88 items. In the Linkage Section, the compiler assumes that each level-01 item starts on a doubleword boundary.

Note: The combined total number of level-77 and level-01 items in the Linkage Section may not exceed 255.

Program Product Information (Version 4)

COMMUNICATION SECTION

The Communication Section contains Communication Description entries for input and/or for output, and optional record description entries. The Communication Section is discussed in "Teleprocessing".

REPORT SECTION

The Report Section contains Report Description entries and report group description entries for every report named in the REPORT clause. The Report Section is discussed in "Report Writer."

FILE DESCRIPTION ENTRY -- DETAILS OF CLAUSES

The file description entry consists of level indicator (FD), followed by file-name, followed by a series of independent clauses. The entry itself is terminated by a period.

General Format

```
FD file-name
     [BLOCK CONTAINS Clause]
     [RECORD CONTAINS Clause]
     [RECORDING MODE Clause]
      LABEL RECORDS Clause
     [VALUE OF Clause]
     [DATA RECORDS Clause]
     [REPORT Clause].
```

The level indicator FD identifies the beginning of a file description entry and must precede the file-name. The clauses that follow the name of the file are optional in many cases, and their order of appearance is not significant.

BLOCK CONTAINS Clause

The BLOCK CONTAINS clause is used to specify the size of a physical record.

Format

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

The BLOCK CONTAINS clause is unnecessary when a physical record contains one and only one complete logical record. In all other instances, this clause is required.

The BLOCK CONTAINS clause need not be specified for:

- Direct files with F, U, or V-mode records.
- Direct files when the RECORDING MODE clause is specified for S-mode records.
- Relative files.
- Files containing U-mode records.

For these types of files, the compiler accepts the clause and treats it as comments, issuing a diagnostic message.

The RECORDS option may be used unless one of the following situations exists, in which case the CHARACTERS option should be used:

1. The physical record contains padding (areas not contained in a logical record).
2. Logical records are grouped in such a manner that an inaccurate physical record size would be implied. Such would be the case where the user describes a mode V record of 100 characters, yet each time he writes a block of 4, he writes a 50-character record followed by three 100-character records. Had he used the RECORDS option, the compiler would have calculated the block length as 420.
3. Logical records extend across physical records; that is, recording mode is S (spanned).

When the RECORDS option is used, the compiler assumes that the block size provides for integer-2 records of maximum size and then provides additional space for any required control bytes.

When the CHARACTERS option is used, the physical record size is specified in Standard Data Format, i.e., in terms of the number of bytes occupied internally by its characters, regardless of the number of characters used to represent the item within the physical record. The number of bytes occupied internally by a data item is included as part of the discussion of the USAGE clause. Integer-1 and integer-2 must include slack bytes and control bytes contained in the physical record.

When the CHARACTERS option is used, and if only integer-2 is shown and is not zero, integer-2 represents the exact size of the physical record. If both integer-1 and integer-2 are shown, they refer to the minimum and maximum size of the physical record, respectively.

When both integer-1 and integer-2 appear, they must be positive integers.

When only integer-2 is shown, and it is specified as zero, the block size is determined at object time from the DD parameters or the data set label for the file. If integer-2 is specified as zero, the file-name may not appear in a SAME AREA clause. The file must either be a standard sequential file or an indexed file whose ACCESS MODE is sequential.

When the BLOCK CONTAINS clause is omitted, it is assumed that records are not blocked. When neither the CHARACTERS nor the RECORDS option is specified, the CHARACTERS option is assumed.

Program Product Information (Version 3 and Version 4)

If the BLOCK CONTAINS clause is omitted and the RECORD CONTAINS 0 CHARACTERS clause is specified, then the block size is determined at object time from the DD parameters or the data set label for the file.

Note: When an indexed file is opened for INPUT or I-O, the blocking factor must equal the blocking factor used when the file was created. This restriction also holds if the block size is specified at object time with a DD card, rather than at compile time with an FD entry.

For an indexed file, neither the [integer-1 TO] nor the CHARACTERS option may be used.

Note: ASCII considerations for the BLOCK CONTAINS clause are given in Appendix E.

RECORD CONTAINS Clause

The RECORD CONTAINS clause is used to specify the size of a file's data records.

Format
RECORD CONTAINS [integer-1 TO] integer-2 CHARACTERS

Since the size of each data record is completely defined within the record description entry, this clause is never required. When the clause is specified, the following notes apply:

1. If both integer-1 and integer-2 are shown, they refer to the number of characters in the smallest data record and the number in the largest data record, respectively.

2. Integer-2, when nonzero, should not be used by itself unless all the data records in the file have the same size. In this case, integer-2 represents the exact number of characters in the data record.

3. The size of the record must be specified in Standard Data Format, i.e., in terms of the number of bytes occupied internally by its characters, regardless of the number of characters used to represent the item within the record. The number of bytes occupied internally by a data item is discussed in the description of the USAGE clause. The size of a record is determined according to the rules for obtaining the size of a group item.

When both integer-1 and integer-2 appear, they must be positive integers.

Program Product Information (Version 3 and Version 4)

When only integer-2 is shown, it may be specified as zero. When it is specified as zero, the record size is determined at object time from either the data set label or the DD card; the recording mode may be fixed, variable, or spanned. If integer-2 is specified as zero, the associated file-name may not appear in a SAME AREA or SAME RECORD AREA clause. The file must be an input file whose organization is standard sequential or indexed and whose ACCESS MODE is sequential.

If, at object time, the actual record (that is, the record actually read) is larger than the 01 data record description, only the record length specified by the record description is accessible to

the user; if the actual record is smaller then the record description, references to areas beyond the actual record produce unpredictable results.

If the RECORD CONTAINS clause specifies integer-2 as zero and the BLOCK CONTAINS 0 CHARACTERS clause is used (or if the BLOCK CONTAINS clause is omitted), then the block size is determined at object time from the DD parameters or the data set label for the file.

Whether this clause is specified or omitted, the record lengths are determined by the compiler from the record descriptions. When one or more of the data item description entries within a record contains an OCCURS DEPENDING ON clause, the compiler uses the maximum value of the variable to calculate the record length.

However, if more than one entry in a given record description contains an OCCURS DEPENDING ON clause, and the maximum values of the variables in these OCCURS clauses do not occur simultaneously, integer-2, as specified by the user, may indicate a maximum record size other than the size calculated by the compiler from the maximum values of the OCCURS clause variables. In this case, the user-specified value of integer-2 determines the amount of storage set aside to contain the data record.

For example, in a school whose total enrollment is 500, an unblocked file of collective attendance records is being created, each record of which is described as follows:

```
01  ATTENDANCE-RECORD.
    05  DATE PICTURE X(6).
    05  NUMBER-ABSENT PICTURE S999 COMPUTATIONAL SYNCHRONIZED.
    05  NUMBER-PRESENT PICTURE S999 COMPUTATIONAL SYNCHRONIZED.
    05  NAMES-OF-ABSENT OCCURS 0 TO 500 TIMES DEPENDING ON
            NUMBER-ABSENT PICTURE A(20).
    05  NAMES-OF-PRESENT OCCURS 0 TO 500 TIMES DEPENDING ON
            NUMBER-PRESENT PICTURE A(20).
```

The programmer can save storage by taking advantage of the fact that NUMBER-ABSENT plus NUMBER-PRESENT will never exceed the school's total enrollment. Unless the programmer writes RECORD CONTAINS 10,010 CHARACTERS in the FD entry for the file, the compiler calculates the record size to be almost twice as large.

Recording Mode

When the RECORDING MODE clause is not used to specify the recording mode of the records in the file, the COBOL compiler scans each record description entry to determine it. The recording mode may be F (fixed), U (unspecified), V (variable), or S (spanned).

Recording Mode F -- All of the records in a file are the same length and each is wholly contained within one block. Blocks may contain more than one record, and there is usually a fixed number of records per block. In this mode, there are no record-length or block-descriptor fields.

Recording Mode U -- The records may be either fixed or variable in length. However, there is only one record per block. There are no record-length or block-descriptor fields.

Recording Mode V -- The records may be either fixed or variable in length, and each must be wholly contained in one block. Blocks may contain more than one record. Each data record includes a record-length

field and each block includes a block-descriptor field. These fields are not described in the Data Division; provision is automatically made for them. These fields are not available to the user.

Recording Mode S -- The records may be either fixed or variable in length, and may be larger than a block. If a record is larger than the remaining space in a block, a segment of the record is written to fill the block. The remainder of the record is stored in the next block (or blocks, if required). Only complete records are made available to the user. Each segment of a record in a block, even if it is the entire record, includes a segment-descriptor field, and each block includes a block-descriptor field. These fields are not described in the Data Division; provision is automatically made for them. These fields are not available to the user.

For standard sequential files, the compiler determines the recording mode for a given file to be:

F if all the records are defined as being the same size and the size is smaller than or equal to the block size.

V if the records are defined as variable in size, or if the RECORD CONTAINS clause specifies variable size records and the longest record is less than or equal to the maximum block size.

S if the maximum block size is smaller than the largest record size.

For direct files, the compiler determines the recording mode for a given file to be:

F if all the records are defined as being the same size, and the size is smaller than or equal to the block size.

U if the records are defined as variable in size, or if the RECORD CONTAINS clause specifies variable size records and the longest record is less than or equal to the maximum block size.

S if the maximum block size is smaller than the largest record size.

Files with indexed or relative organization must have F-mode records.

Note: ASCII considerations for compiler calculation of recording mode are given in Appendix E.

RECORDING MODE Clause

The RECORDING MODE clause is used to specify the format of the logical records in the file.

Format
RECORDING MODE IS mode

Mode is specified as either F, V, U or S. If this clause is not specified, the recording mode is determined as described in "Recording Mode."

F mode (fixed-length format) may be specified when all the logical records in a file are the same length and each is wholly contained within one physical block. This implies that no OCCURS DEPENDING ON clause is associated with an entry in any record description for the file. If more than one record description entry is given following the FD entry, all record lengths calculated from the record descriptions must be equal.

V mode (variable-length format) may be specified for any combination of record descriptions if each record is wholly contained in one physical block. A mode V logical record is preceded by a control field containing the length of the logical record. Blocks of variable-length records include a block-descriptor control field. V mode may not be specified for files with indexed or relative organization.

U mode (unspecified format) may be specified for any combination of record descriptions, if each record is wholly contained in one physical block, and the block contains only one physical record. It is comparable to V mode with the exception that U-mode records are not blocked and have no preceding control field. U mode may not be specified for files with indexed or relative organization.

S mode (spanned format) may be specified for any combination of record descriptions. If a record is larger than the remaining space in a block, a segment of the record is written to fill the block. The remainder of the record is stored in the next block (or blocks, if required). Only complete records are made available to the user. Each segment of a record in a block, even if it is the entire record, includes a segment-descriptor field, and each block includes a block-descriptor field. These fields are not described in the Data Division; provision is automatically made for them. These fields are not available to the user. S mode may be specified for standard sequential files or for direct files.

Note: ASCII considerations for the RECORDING MODE clause are given in Appendix E.

LABEL RECORDS Clause

The LABEL RECORDS clause specifies whether labels are present, and if present, identifies the labels.

Format

```
LABEL { RECORD IS  }  { OMITTED                                          }
      { RECORDS ARE}  { STANDARD                                         }
                      { data-name-1 [data-name-2]...[TOTALING AREA       }
                      {    IS data-name-3 TOTALED AREA IS                }
                      {    data-name-4]                                  }
```

The LABEL RECORDS clause is required in every FD.

The OMITTED option specifies that either no explicit labels exist for the file or that the existing labels are nonstandard and the user does not want them to be processed by a label declarative (i.e., they will be processed as data records). The OMITTED option must be specified for files assigned to unit record devices. It may be specified for files

assigned to magnetic tape units. Use of the OMITTED option does not result in automatic bypassing of nonstandard labels on input. It is the user's responsibility to either process or bypass nonstandard labels on input and create them on output.

The STANDARD option specifies that labels exist for the file and that the labels conform to system specifications. The system will bypass user labels appearing in the file if the STANDARD option is specified.

The STANDARD option must be specified for files with indexed organization.

Note: ASCII considerations for the LABEL RECORDS clause are given in Appendix E.

In the discussion that follows, all references to data-name-1 apply equally to data-name-2.

The data-name-1 option indicates either the presence of user labels in addition to standard labels, or the presence of nonstandard labels. Data-name-1 specifies the name of a user label record. Data-name-1 must appear as the subject of a record description entry associated with the file, and must not appear as an operand of the DATA RECORDS clause for the file.

If user labels are to be processed, data-name-1 may be specified for direct files, relative files, or for standard sequential files with the exception of files assigned to unit-record devices.

A user label is 80 characters in length. A user header label must have UHL in character positions 1 through 3. A user trailer label must have UTL in character positions 1 through 3. Both header and trailer labels may be grouped, and each label must show the relative position (1, 2, ...) of the label within the user label group, in character position 4. The remaining 76 characters are formatted according to the user's choice. User header labels, follow standard beginning file labels but precede the first data record; user trailer labels follow standard closing file labels.

If nonstandard labels are to be processed, data-name-1 may be specified only for standard sequential files, with the exception of files assigned to unit-record devices. The length of a nonstandard label may not exceed 4,095 character positions.

All Procedure Division references to data-name-1, or to any item subordinate to data-name-1, must appear within label processing declaratives.

Note: In the discussion that follows, the term volume applies to all input/output devices. Treatment of a mass storage device in the sequential access mode is logically equivalent to the treatment of a tape file.

The TOTALING and TOTALED AREA option may be specified when the programmer wishes to create a sequential file with user labels. With this option he is able to obtain exact information about each volume of a multivolume file, so that the information can be recorded in the user trailer label each time a volume switch occurs.

(A WRITE statement executed before a volume switch may cause a record to be written after the volume switch occurs (i.e., the record is written on the new volume), because the volume switch took place between the time the WRITE statement was initiated, and the time the record was actually written. Thus, information accumulated as current records are processed does not, at label processing time, necessarily reflect the output on that volume. The TOTALING/TOTALED AREA option can be used to rectify this situation.)

Data-name-3, the TOTALING AREA, is defined in the Working-Storage Section. Data-name-3 is used by the programmer to store information to be used in constructing the user labels -- information such as accumulated totals for records, identification fields within the current record, etc. Before each WRITE statement he issues, he must store information associated with the current record in data-name-3. There are two exceptions. If he has specified the SAME RECORD AREA and/or the APPLY WRITE-ONLY clauses, then he must store the current record information in data-name-3 after issuing the WRITE statement. The information in data-name-3 is always associated by the system with the current WRITE statement.

Data-name-4, the TOTALED AREA, must be defined at the 01 level in the Linkage Section, and must contain fields described as identical with those within data-name-3. The system allocates the space for data-name-4 and uses it to save user label information (obtained from data-name-3) associated with the most recent record actually written on the current volume. Thus, when a volume switch occurs, data-name-4 contains the user label information for the last record actually written on the current volume, and the programmer can use data-name-4 to construct an accurate trailer label for the current volume, and an accurate header label for the next.

For both data-name-3 and data-name-4, the user must define the first two bytes of each record for use by the system.

The TOTALING and TOTALED AREA option may not be specified for S-mode records.

VALUE OF Clause

The VALUE OF clause particularizes the description of an item in the label records associated with a file, and serves only as documentation.

Format

```
VALUE OF data-name-1 IS  {literal-1  }
                         {data-name-2}

          [data-name-3 IS {literal-2  }]...
                          {data-name-4}
```

To specify the required values of identifying data items in the label records for the file, the programmer must use the VALUE OF clause.

However, this compiler treats the VALUE OF clause as comments, since this function is performed by the system through the LABEL parameter of the DD statement for the file (see the Programmer's Guide).

DATA RECORDS Clause

The DATA RECORDS clause serves only as documentation, and identifies the records in the file by name.

Format
DATA { RECORD IS / RECORDS ARE } data-name-1 [data-name-2]...

The presence of more than one data-name indicates that the file contains more than one type of data record. That is, two or more record descriptions for a given file occupy the same storage area. These records need not have the same description. The order in which the data-names are listed is not significant.

Data-name-1, data-name-2, etc., are the names of data records, and each must be preceded in its record description entry by the level number 01. This clause is never required.

REPORT Clause

The REPORT clause is used in conjunction with the Report Writer feature. A complete description of the REPORT clause can be found in "Report Writer."

In COBOL, the terms used in connection with data description are:

Data Description Entry -- the clause, or clauses, that specify the characteristics of any particular noncontiguous data item, or of any data item that is a portion of a record. The data description entry consists of a level number, a data-name (or condition-name), plus any associated data description clauses.

Data Item Description Entry -- a data description entry that defines a noncontiguous data item. It consists of a level number (77), a data-name plus any associated data description entries. Data item description entries are valid in the Working-Storage Section and in the Linkage Section.

Record Description Entry -- the term used in connection with a record. It consists of a hierarchy of data description entries. Record description entries are valid in the File, Working-Storage, and Linkage Sections.

Program Product Information (Version 4)

For Version 4 record description entries are valid in the Communication Section.

The maximum length for a data description entry is 32,767 bytes, except for a fixed-length Working-Storage or Linkage Section group item, which may be as long as 131,071 bytes.

General Format 1

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

    [REDEFINES Clause]
    [BLANK WHEN ZERO Clause]
    [JUSTIFIED Clause]
    [OCCURS Clause]
    [PICTURE Clause]
    [SIGN Clause] (Versions 3 and 4)
    [SYNCHRONIZED Clause]
    [USAGE Clause]
    [VALUE Clause]
```

General Format 2

```
66 data-name-1 RENAMES Clause.
```

General Format 3
88 condition-name VALUE Clause.

General Format 1 is used for record description entries in the File, Working-Storage, and Linkage Sections and for data item description entries in the Working-Storage and Linkage Sections. The following rules apply:

1. Level number may be any number from 1 through 49 for record description entries, or 77 for data item description entries.

2. The clauses may be written in any order, with one exception: the REDEFINES clause, when used, must immediately follow the data-name.

3. The PICTURE clause must be specified for every elementary item, with the exception of index data items and internal floating-point items. Index data items are described in "Table Handling."

4. Each entry must be terminated by a period.

5. Semicolons or commas may be used as separators between clauses.

Program Product Information (Version 4)

For Version 4, General Format 1 is also used for record description entries in the Communication Section.

General Format 2 is used for the purpose of regrouping data items. The following rules apply:

1. A level-66 entry cannot rename another level-66 entry, nor can it rename a level-77, level-88, or level-01 entry.

2. All level-66 entries associated with a given logical record must immediately follow the last data description entry in the record.

3. The entry must be terminated by a period.

The RENAMES clause is discussed in detail later in this chapter.

General Format 3 is used to describe entries that specify condition-names to be associated with particular values of a conditional variable. A condition-name is a name assigned by the user to a specific value a data item may assume during object program execution. The following rules apply:

1. The condition-name entries for a particular conditional variable must immediately follow the conditional variable.

2. A condition-name can be associated with any elementary data description entry except another condition-name, or an index data item.

3. A condition-name can be associated with a group item data description entry. In this case:

   - The condition value must be specified as a nonnumeric literal or figurative constant.
   - The size of the condition value must not exceed the sum of the sizes specified by the pictures in all the elementary items within the group.
   - No element within the group may contain a JUSTIFIED or SYNCHRONIZED clause.
   - No USAGE other than USAGE IS DISPLAY may be specified within the group.

4. The specification of a condition-name at the group level does not restrict the specification of condition-names at levels subordinate to that group.

5. The relation test implied by the definition of a condition-name at the group level is performed in accordance with the rules for comparison of nonnumeric operands, regardless of the nature of elementary items within the group.

6. Each entry must be terminated by a period.

Examples of both group and elementary condition-name entries are given in the description of the VALUE clause.

DATA DESCRIPTION ENTRY -- DETAILS OF CLAUSES

The data description entry consists of a level number, followed by a data-name, followed by a series of independent clauses. The clauses may be written in any order, with one exception: the REDEFINES clause, when used, must immediately follow the data-name. The entry must be terminated by a period.

Data-name or FILLER Clause

A data-name specifies the name of the data being described. The word FILLER specifies an elementary or group item of the logical record that is never referred to and therefore need not be named.

Format

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

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

Program Product Information (Version 4)

For Version 4, record description entries are allowed in the Communication Section. A data-name or the key word FILLER must be the first word following each level number in such an entry.

A data-name is a name assigned by the user to identify a data item used in a program. A data-name refers to a kind of data, not to a particular value; the item referred to may assume a number of different values during the course of a program.

The key word FILLER is used to specify an elementary item or a group item that is never referred to in the program, and therefore need not be named. Under no circumstances may a FILLER item be referred to directly. In a MOVE, ADD, or SUBTRACT statement with the CORRESPONDING option, FILLER items are ignored.

Note: Level-77 and level-01 entries in the Working-Storage Section and Linkage Section must be given unique data-names, since neither can be qualified. Subordinate data-names, if they can be qualified, need not be unique.

REDEFINES Clause

The REDEFINES clause allows the same computer storage area to contain different data items or provides an alternative grouping or description of the same data. That is, the REDEFINES clause specifies the redefinition of a storage area, not of the data items occupying the area.

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

The level numbers of data-name-1 and data-name-2 must be identical, but must not be 66 or 88. Data-name-2 is the name associated with the previous data description entry. Data-name-1 is an alternate name for the same area. When written, the REDEFINES clause must be the first clause following data-name-1.

The REDEFINES clause must not be used in level-01 entries in the File Section. Implicit redefinition is provided when more than one level-01 entry follows a file description entry.

Program Product Information (Version 4)

For Version 4, the REDEFINES clause must not be used in level-01 entries in the Communication Section. Implicit redefinition is provided when more than one level-01 entry follows a communication description entry.

Redefinition starts at data-name-2 and ends when a level number less than or equal to that of data-name-2 is encountered. Between the data descriptions of data-name-2 and data-name-1, there may be no entries having lower level numbers (numerically) than the level number of data-name-2 and data-name-1.

Example:

```
05  A.
    10  A-1 PICTURE X.
    10  A-2 PICTURE XXX.
    10  A-3 PICTURE 99.
05  B REDEFINES A PICTURE X(6).
```

In this case, B is data-name-1, and A is data-name-2. When B redefines A, the redefinition includes all of the items subordinate to A (A-1, A-2, and A-3).

The data description entry for data-name-2 cannot contain an OCCURS clause, nor can data-name-2 be subordinate to an entry which contains an OCCURS clause. An item subordinate to data-name-2 may contain an OCCURS clause without the DEPENDING ON option. Data-name-1 or any items subordinate to data-name-1 may contain an OCCURS clause without the DEPENDING ON option. Neither data-name-2 nor data-name-1 nor any of their subordinate items may contain an OCCURS clause with the DEPENDING ON option.

When data-name-1 has a level number other than 01, it must specify a storage area of the same size as data-name-2.

However, this compiler will allow the size of the redefining area (data-name-1) to be less than the size of the redefined area (data-name-2).

If data-name-1 contains an OCCURS clause, its size is computed by multiplying the length of one occurrence by the number of occurrences.

Note: In the discussion which follows, the term "computational" refers to COMPUTATIONAL, COMPUTATIONAL-1, and COMPUTATIONAL-2 items.

When the SYNCHRONIZED clause is specified for an item that also contains a REDEFINES clause, the data item that is redefined must have the proper boundary alignment for the data item that REDEFINES it. For example, if the programmer writes:

```
05  A PICTURE X(4).
05  B REDEFINES A  PICTURE S9(9) COMP SYNC.
```

he must ensure that A begins on a fullword boundary.

When the SYNCHRONIZED clause is specified for a computational item that is the first elementary item subordinate to an item that contains a REDEFINES clause, the computational item must not require the addition of slack bytes.

Except for condition-name entries, the entries giving the new description of the storage area must not contain any VALUE clauses.

The entries giving the new description of the storage area must follow the entries describing the area being redefined, without intervening entries that define new storage areas. Multiple redefinitions of the same storage area should all use the data-name of the entry that originally defined the area. However, this compiler will accept as valid the data-name of the preceding entry when multiple redefinition is used. For example, both of the following are valid uses of the REDEFINES clause:

```
05  A PICTURE 9999.
05  B REDEFINES A PICTURE 9V999.
05  C REDEFINES A PICTURE 99V99.
```

```
05  A PICTURE 9999.
05  B REDEFINES A PICTURE 9V999.
05  C REDEFINES B PICTURE 99V99.
```

Data items within an area can be redefined without their lengths being changed; the following statements result in the storage layout shown in Figure 5.

```
05  NAME-2.
    10  SALARY PICTURE XXX.
    10  SO-SEC-NO PICTURE X(9).
    10  MONTH PICTURE XX.
05  NAME-1 REDEFINES NAME-2.
    10  WAGE PICTURE XXX.
    10  MAN-NO PICTURE X(9).
    10  YEAR PICTURE XX.
```

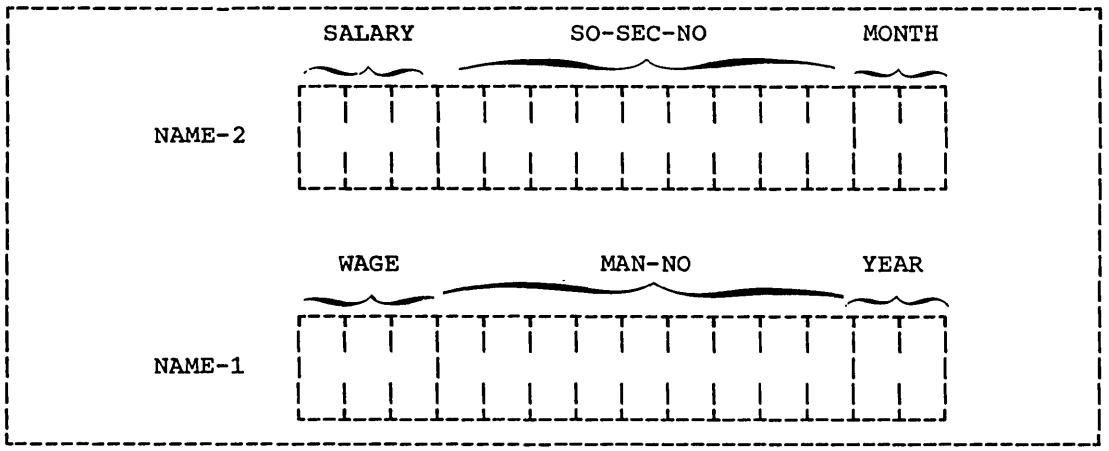


Figure 5. Areas Redefined Without Changes in Length

Data items can also be rearranged within an area; the following statements result in the storage layout shown in Figure 6.

```
05  NAME-2.
    10  SALARY  PICTURE XXX.
    10  SO-SEC-NO  PICTURE X(9).
    10  MONTH  PICTURE XX.
05  NAME-1  REDEFINES  NAME-2.
    10  MAN-NO  PICTURE X(6).
    10  WAGE PICTURE 999V999.
    10  YEAR  PICTURE XX.
```

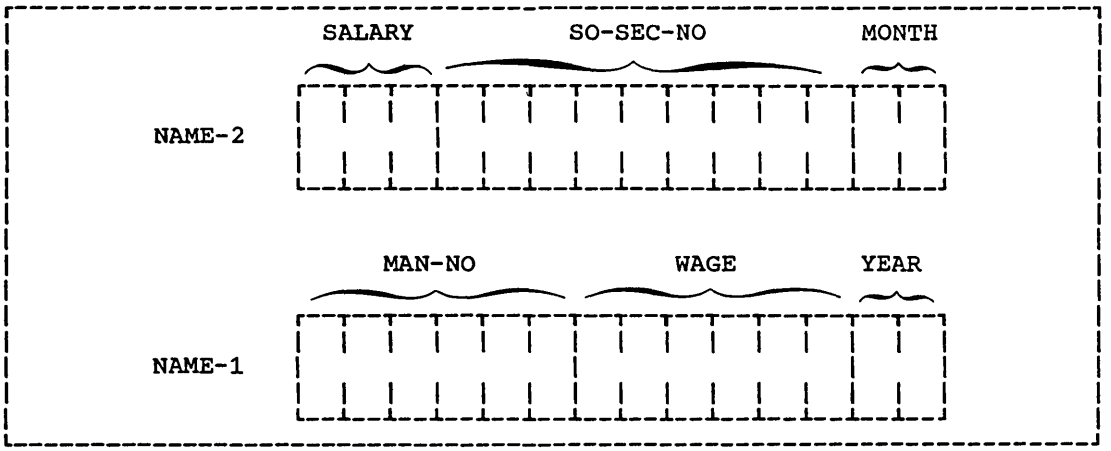


Figure 6. Areas Redefined and Rearranged

When an area is redefined, all descriptions of the area remain in effect. Thus, if B and C are two separate items that share the same storage area due to redefinition, the procedure statements MOVE X TO B or MOVE Y TO C could be executed at any point in the program. In the first case, B would assume the value of X and take the form specified by the description of B. In the second case, the same physical area would receive Y according to the description of C. It should be noted, however, that if both of the foregoing statements are executed successively in the order specified, the value Y will overlay the value X. However, redefinition in itself does not cause any data to be erased and does not supersede a previous description.

The usage of data items within an area can be redefined.

Altering the USAGE of an area through redefinition does not cause any change in existing data. Consider the example:

```
05  B  PICTURE 99 USAGE DISPLAY  VALUE IS 8.
05  C  REDEFINES B  PICTURE S99  USAGE COMPUTATIONAL.
05  A  PICTURE S9999  USAGE COMPUTATIONAL.
```

Assuming that B is on a halfword boundary, the bit configuration of the value 8 is 1111 0000 1111 1000, because B is a DISPLAY item. Redefining B does not change its appearance in storage. Therefore, a great difference results from the two statements ADD B TO A and ADD C TO A. In the former case, the value 8 is added to A, because B is a display item. In the latter case, the value -3,848 is added to A, because C is a binary item (USAGE IS COMPUTATIONAL), and the bit configuration appears as a negative number.

Moving a data item to a second data item that redefines the first one (for example, MOVE B TO C when C redefines B), may produce results that are not those expected by the programmer. The same is true of the reverse (MOVE B TO C when B redefines C).

A REDEFINES clause may be specified for an item within the scope of an area being redefined, that is, an item subordinate to a redefined item. The following example would thus be a valid use of the REDEFINES clause:

```
05  REGULAR-EMPLOYEE.
    10  LOCATION  PICTURE A(8).
    10  STATUS  PICTURE X(4).
    10  SEMI-MONTHLY-PAY PICTURE 9999V99.
    10  WEEKLY-PAY  REDEFINES SEMI-MONTHLY-PAY
        PICTURE 999V999.

05  TEMPORARY-EMPLOYEE REDEFINES REGULAR-EMPLOYEE.
    10  LOCATION  PICTURE A(8).
    10  FILLER  PICTURE X(6).
    10  HOURLY-PAY  PICTURE 99V99.
```

REDEFINES clauses may also be specified for items subordinate to items containing REDEFINES clauses. For example:

```
05  REGULAR-EMPLOYEE.
    10  LOCATION  PICTURE A(8).
    10  STATUS  PICTURE X(4).
    10  SEMI-MONTHLY-PAY  PICTURE 999V999.

05  TEMPORARY-EMPLOYEE  REDEFINES REGULAR-EMPLOYEE.
    10  LOCATION  PICTURE A(8).
    10  FILLER  PICTURE X(6).
    10  HOURLY-PAY  PICTURE 99V99.
    10  CODE-H REDEFINES  HOURLY-PAY  PICTURE 9999.
```

BLANK WHEN ZERO Clause

This clause specifies that an item is to be set to blanks whenever its value is zero.

Format
BLANK WHEN ZERO

When the BLANK WHEN ZERO clause is used, the item will contain only blanks if the value of the item is zero.

The BLANK WHEN ZERO clause may be specified only at the elementary level for numeric edited or numeric items. When this clause is used for an item whose PICTURE is numeric, the category of the item is considered to be numeric edited.

This clause may not be specified for level-66 and level-88 data items.

JUSTIFIED Clause

The JUSTIFIED clause is used to override normal positioning of data within a receiving alphabetic or alphanumeric data item.

Format
{ JUSTIFIED } RIGHT { JUST }

Normally, the rule for positioning data within a receiving alphanumeric or alphabetic data item is:

- The data is aligned in the receiving field, beginning at the leftmost character position within the receiving field. Unused character positions to the right are filled with spaces. If truncation occurs, it will be at the right.

The JUSTIFIED clause affects the positioning of data in the receiving field as follows:

1. When the receiving data item is described with the JUSTIFIED clause and the data item sent is larger than the receiving data item, the leftmost characters are truncated.

2. When the receiving data item is described with the JUSTIFIED clause and is larger than the data item sent, the data is aligned at the rightmost character position in the data item. Unused character positions to the left are filled with spaces.

The JUSTIFIED clause may only be specified for elementary items.

This clause must not be specified for level-66 or level-88 data items.

OCCURS Clause

The OCCURS clause is used to define tables and other homogeneous sets of data, whose elements can be referred to by subscripting or indexing. The OCCURS clause is described in "Table Handling."

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 can be used only at the elementary level.

The character string consists of certain allowable combinations of characters in the COBOL character set. The maximum number of characters allowed in the character string is 30. The allowable combinations determine the category of the elementary item.

There are five categories of data that can be described with a PICTURE clause. They are:

1. Alphabetic
2. Numeric
3. Alphanumeric
4. Alphanumeric edited
5. Numeric edited

The Three Classes of Data

The five categories of data items are grouped into three classes: alphabetic, numeric, and alphanumeric. For alphabetic and numeric, the class and the category are synonymous. The alphanumeric class includes the categories of alphanumeric (without editing), alphanumeric edited, and numeric edited.

Every elementary item belongs to one of the three classes and to one of the five categories. The class of a group item is treated at object time as alphanumeric regardless of the class of the elementary items subordinate to that group item.

Table 6 shows the relationship of the class and category for elementary and group data items.

Table 6. Class and Category of Elementary and Group Data Items

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

Character String and Item Size

In the processing of data through COBOL statements, the size of an elementary item is determined through the number of character positions specified in its PICTURE character string. In core storage, however, the size is determined by the actual number of bytes the item occupies, as determined by its PICTURE character string, and also by its USAGE (see "USAGE Clause").

Normally, when an arithmetic item is moved from a longer field into a shorter one, this compiler will truncate the data to the number of characters represented in the PICTURE character string of the shorter item.

For example, if a sending field with PICTURE S99999, and containing the value +12345, is moved to a COMPUTATIONAL receiving field with PICTURE S99, the data is truncated to +45.

As a compile time option, however, this compiler may be instructed, in such an operation, to truncate only such digits as would overflow the receiving field. If this option is used, the result of the move in the foregoing example is +2345, since a COMPUTATIONAL item two bytes in length can contain up to four decimal digits of data. Note that care must be used when using this option, since there are times when the data may contain a negative sign.

Repetition of Symbols

An integer which is enclosed in parentheses following one of the symbols

A , X 9 P Z * B 0 + - $

indicates the number of consecutive occurrences of the symbol. For example, if the programmer writes

```
A(40)
```

the four characters (40) indicate forty consecutive appearances of the symbol A. The number within parentheses may not exceed 32,767.

Note: The following symbols may appear only once in a given PICTURE clause:

S V . CR DB E

Symbols Used in the PICTURE Clause

The functions of the symbols used to describe an elementary item are:

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

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

E The E in the character string represents the exponent in an external floating-point item. The E occupies one byte of storage, and is counted in determining the size of the elementary item. The E is included in any representation upon external media.

P The P indicates an assumed decimal scaling position, and is used to specify the location of an assumed decimal point when the point is not within the number that appears in the data item. The scaling position character P is not counted in the size of the data item. Scaling position characters are counted in determining the maximum number of digit positions (18) in numeric edited items or in items that appear as operands in arithmetic statements.

The scaling position character P may appear only to the left or right of the other characters in the string as a continuous string of P's within a PICTURE description. The sign character S and the assumed decimal point V are the only characters which may appear to the left of a leftmost string of P's. Since the scaling position character P implies an assumed decimal point (to the left of the P's if the P's are leftmost PICTURE characters and to the right of the P's if the P's are rightmost PICTURE characters), the assumed decimal point symbol V is redundant as either the leftmost or rightmost character within such a PICTURE description.

S The symbol S is used in a PICTURE character string to indicate the presence (but not the representation nor, necessarily, the position) of an operational sign, and must be written as the leftmost character in the PICTURE string. An operational sign indicates whether the value of an item involved in an operation is positive or negative. The symbol S is not counted in determining the size of the elementary item, unless an associated SIGN clause specifies the SEPARATE CHARACTER option.

V The V is used in a character string to indicate the location of the assumed decimal point and may appear only once in a character string. The V does not represent a character position and, therefore, is 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 may contain any allowable character from the EBCDIC set.

Z Each Z in the character string represents a leading numeric character position; when that position contains a zero, the zero is replaced by a space character. Each Z is counted in the size of the item.

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

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

, Each comma in the character string represents a character position into which a comma will be inserted. This character is counted in the size of the item. The comma insertion character cannot be the last character in the PICTURE character string.

. When a period appears in the character string, it is an editing symbol that represents the decimal point for alignment purposes. In addition, it represents a character position into which a period will be inserted. This character is counted in the size of the item. The period insertion character cannot be the last character in the PICTURE character string.

Note: For a given 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 this exchange, the rules for the period apply to the comma and the rules for the comma apply to the period wherever they appear in a PICTURE clause.

+, -, CR, DB These symbols are used as editing sign control symbols. When used, each represents the character position into which the editing sign control symbol will be placed. The symbols are mutually exclusive in one character string. Each character used in the symbol is counted in determining the size of the data item.

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

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

Table 7 shows the order of precedence of the symbols used in the PICTURE clause.

The Five Categories of Data

The following is a detailed description of the allowable combinations of characters for each category of data.

ALPHABETIC ITEMS: An alphabetic item is one whose PICTURE character string contains only the symbol A. Its contents, when represented in Standard Data Format, must be any combination of the 26 letters of the Roman alphabet and the space from the COBOL character set. Each alphabetic character is stored in a separate byte.

If a VALUE clause is specified for an alphabetic item, the literal must be nonnumeric.

Table 7. Precedence of Symbols Used in the PICTURE Clause

SECOND SYMBOL \ FIRST SYMBOL		NON-FLOATING INSERTION SYMBOLS								FLOATING INSERTION SYMBOLS						OTHER SYMBOLS						
		B	0	,	.	+ -	+ -	CR DB	cs[1]	Z *	Z *	+ -	+ -	cs[1]	cs[1]	9	A X	S	V	P	P	E[2]
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						
	+ or -																					X
	+ or -	X	X	X	X				X	X	X			X	X	X			X	X	X	
	CR or DB	X	X	X	X				X	X	X			X	X	X			X	X	X	
	cs[1]					X																
FLOATING INSERTION SYMBOLS	Z or *	X	X	X		X			X	X												
	Z or *	X	X	X	X	X			X	X	X								X		X	
	+ or -	X	X	X					X			X										
	+ or -	X	X	X	X				X			X	X						X		X	
	cs[1]	X	X	X		X								X								
	cs[1]	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	X
	A X	X	X													X	X					
	S																					
	V	X	X	X		X			X	X		X		X		X		X		X		
	P	X	X	X		X			X	X		X		X		X		X		X		
	P					X			X									X	X		X	
	E[2]				X	X										X			X			

[1] cs is the abbreviation for the currency symbol.
[2] See the description of external floating-point items for the specific combination of symbols that is valid.

At least one of the symbols A, X, Z, 9, or *, or at least two of the symbols +, -, or cs must be present in a PICTURE string.
An X at an intersection indicates that the symbol(s) at the top of the column may, in a given character-string, appear anywhere to the left of the symbol(s) at the left of the row.
Non-floating insertion symbols + and -, floating insertion symbols Z, *, +, -, and cs, and other symbol P appear twice in the above PICTURE character precedence table. The leftmost column and uppermost row for each symbol represents its use to the left of the decimal point position. The second appearance of the symbol in the table represents its use to the right of the decimal point position.
Braces ({}) indicate items that are mutually exclusive.

ALPHANUMERIC ITEMS: An alphanumeric item is one whose PICTURE character string is restricted to combinations of the symbols A, X, and 9. The item is treated as if the character string contained all X's. Its contents, when represented in Standard Data Format, are allowable characters from the EBCDIC set.

A PICTURE character string which contains all A's or all 9's does not define an alphanumeric item.

If a VALUE clause is specified for an alphanumeric item, the literal must be nonnumeric.

NUMERIC ITEMS: There are two types of numeric items: fixed-point items and floating-point items.

Fixed-Point Numeric Items: There are three types of fixed-point numeric items: external decimal, binary, and internal decimal. See the discussion of the USAGE clause for details concerning each.

The PICTURE of a fixed-point numeric item may contain a valid combination of the following characters:

9 V P S

Examples of fixed-point numeric items:

PICTURE	Valid Range of Values
9999	0 through 9999
S99	-99 through +99
S999V9	-999.9 through +999.9
PPP999	0 through .000999
S999PPP	-1000 through -999000 and +1000 through +999000 or zero

The maximum size of a fixed-point numeric item is 18 digits.

The contents of a fixed-point numeric item, when represented in Standard Data Format, must be a combination of the Arabic numerals 0 through 9; the item may contain an operational sign. If the PICTURE contains an S, the contents of the item are treated as positive or negative values, depending on the operational sign; if the PICTURE does not contain an S, the contents of the item are treated as absolute values.

If a VALUE clause is specified for an elementary numeric item, the literal must be numeric. If a VALUE clause is specified for a group item consisting of elementary numeric items, the group is considered alphanumeric, and the literal must therefore be nonnumeric.

Note: ASCII considerations for the PICTURE clause are given in Appendix E.

Floating-Point Numeric Items: These items define data whose potential range of value is too great for fixed-point presentation. The magnitude of the number represented by a floating-point item must be greater than 5.4×10^{-79} but must not exceed $.72 \times 10^{76}$.

There are two types of floating-point items: internal floating-point and external floating-point. See the discussion of the USAGE Clause for details concerning each.

No PICTURE clause may be associated with an internal floating-point item.

An external floating-point item has a PICTURE character string in the following form:

{±}mantissaE{±}exponent

where each element of the string is composed according to the following rules:

±
A plus sign or a minus sign must immediately precede both the mantissa and the exponent in the PICTURE character string

+ indicates that a plus sign in the data represents positive values and that a minus sign represents negative values.

- indicates that a space character in the data represents positive values and that a minus sign represents negative values.

The plus sign, the space character, and the minus sign occupy a byte of storage.

mantissa
The mantissa immediately follows the first sign character, and is represented using the following three symbols:

9 Each 9 in the mantissa character string represents a digit position into which a numeric character will be placed. From one to sixteen 9's may be present in the string. Each digit position occupies a byte of storage.

. indicates an actual decimal point. It occupies a byte of storage.

V indicates an assumed decimal point. It does not take up any storage.

One actual or assumed decimal point must be present in the mantissa as a leading, embedded, or trailing symbol.

E
indicates the exponent, and immediately follows the mantissa. It occupies one byte of storage.

exponent
The exponent immediately follows the second sign character. It is represented by two consecutive 9's. Each occupies a byte of storage.

External data must conform to the representation specified in the PICTURE clause.

Examples of external floating-point items:

PICTURE	Format of External Data	Value Expressed
-9V99E-99	b540E-79	$+5.40 \times 10^{-79}$
+999.99E+99	+123.45E-14	$+123.45 \times 10^{-14}$
-V9(6)E+99	b565656E+45	$+.565656 \times 10^{45}$
+.9(10)E-99	+.7200000000E 76	$+.72 \times 10^{76}$

(Note that any of the above PICTURE representations can express the full range of possible values.)

No VALUE clause may be associated with an external floating-point item.

ALPHANUMERIC EDITED ITEMS: An alphanumeric edited item is one whose PICTURE character string is restricted to certain combinations of the following symbols:

A X 9 B 0

To qualify as an alphanumeric edited item, one of the following conditions must be true:

1. The character string must contain at least one B and at least one X.
2. The character string must contain at least one 0 and at least one X.
3. The character string must contain at least one 0 (zero) and at least one A. Its contents, when represented in Standard Data Format, are allowable characters chosen from the EBCDIC set.

USAGE IS DISPLAY is used in conjunction with alphanumeric edited items.

If a VALUE clause is specified for an alphanumeric edited item, the literal must be nonnumeric. The literal is treated exactly as specified; no editing is performed.

Editing Rules: Alphanumeric edited items are subject to only one type of editing: simple insertion using the symbols 0 and B.

Examples of alphanumeric edited items:

PICTURE	Value of Data	Edited Result
000X(12)	ALPHANUMER01	000ALPHANUMER01
BBBX(12)	ALPHANUMER01	ALPHANUMER01
000A(12)	ALPHABETIC	000ALPHABETIC
X(5)BX(7)	ALPHANUMERIC	ALPHA NUMERIC

NUMERIC EDITED ITEMS: A numeric edited item is one whose PICTURE character string is restricted to certain combinations of the symbols:

B P V Z 0 9 , . * + - CR DB $

The allowable combinations are determined from the order of precedence of symbols and editing rules.

The maximum number of digit positions that may be represented in the character string is 18.

The contents of the character positions that represent a digit, in Standard Data Format, must be one of the numerals.

USAGE IS DISPLAY is used in conjunction with numeric edited items.

If a VALUE clause is specified for a numeric edited item the literal must be nonnumeric. The literal is treated exactly as specified; no editing is performed.

The maximum length of a numeric edited item is 127 characters.

Editing Rules: All types of editing are valid for numeric edited items.

Types of Editing

There are two general methods of performing editing in the PICTURE clause: by insertion or by suppression and replacement.

There are four types of insertion editing:

1. Simple insertion
2. Special insertion
3. Fixed insertion
4. Floating insertion

There are two types of suppression and replacement editing:

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

Insertion Editing

Simple insertion editing is performed using the following insertion characters:

(comma) B (space) 0 (zero)

The insertion characters are counted in the size of the item and represent the position in the item into which the character will be inserted.

Examples of simple insertion editing:

PICTURE	Value of Data	Edited Result
99,999	12345	12,345
9,999,000	12345	2,345,000
99B999B000	1234	01 234 000
99B999B000	12345	12 345 000
99BBB999	123456	23 456

Special insertion editing is performed using the period (.) as the insertion character. The result of special insertion editing is the appearance of the insertion character in the item in the same position as shown in the character string.

In addition to being an insertion character, the period represents a decimal point for alignment purposes. The insertion character used for the actual decimal point is counted in the size of the item.

The use of both the assumed decimal point, represented by the symbol V, and the actual decimal point, represented by the period insertion character, in one PICTURE character string is not allowed.

Examples of special insertion editing:

PICTURE	Value of Data	Edited Result
999.99	1.234	001.23
999.99	12.34	012.34
999.99	123.45	123.45
999.99	1234.5	234.50

Fixed insertion editing is performed by using the following insertion characters:

currency symbol	$
editing sign control symbols	+ - CR DB

Only one currency symbol and only one of the editing sign control symbols can be used in a given PICTURE character string.

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.

$ — The currency symbol must be the leftmost character position to be counted in the size of the item, unless it is preceded by either a + or a - symbol.

+ or - — When either symbol is used, it must represent the leftmost or rightmost character position to be counted in the size of the item.

CR or DB — When either symbol is used, it represents two character positions in determining the size of the item and must represent the rightmost character positions that are counted in the size of the item.

Editing sign control symbols produce results depending upon the value of the data item as shown in Table 8.

Table 8. Editing Sign Control Symbols and Their Results

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 of fixed insertion editing:

PICTURE	Value of Data	Edited Result
999.99+	+6555.556	555.55+
+9999.99	-5555.555	-5555.55
9999.99-	+1234.56	1234.56
$999.99	-123.45	$123.45
-$999.99	-123.456	-$123.45
$9999.99CR	+123.45	$0123.45
$9999.99DB	-123.45	$0123.45DB

Floating insertion editing is indicated in a PICTURE character string by using a string of at least two of the allowable insertion characters $ + or - to represent the leftmost numeric character positions into which the insertion characters can be floated.

The currency symbol ($) and the editing sign symbols (+ or -) are mutually exclusive as floating insertion characters in a given PICTURE character string.

Any of the simple insertion characters (, B 0) embedded in the string of floating insertion characters, or to the immediate right of this string, are part of the floating string.

In a PICTURE character string, there are only two ways of representing floating insertion editing:

1. Any or all leading numeric character positions to the left of the decimal point are represented by the insertion character.

2. All of the numeric character positions in the PICTURE character string are represented by the insertion character.

The result of floating insertion editing depends upon the representation in the PICTURE character string:

1. If the insertion characters are only to the left of the decimal point, a single insertion character is placed into the character position immediately preceding the first nonzero digit in the data represented by the insertion symbol string or the decimal point, whichever is farther to the left of the PICTURE character string.

2. 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 characters are only to the left of the decimal point.

To avoid truncation when using floating insertion editing, the programmer must specify the minimum size of the PICTURE character string for the receiving data item to be:

1. The number of characters in the sending item, plus

2. The number of insertion characters (other than floating insertion characters) being edited into the receiving data item, plus

3. One character for the floating insertion character.

Examples of floating insertion editing:

PICTURE	Value of Data	Edited Result
$$$$.99	.123	$.12
$$$9.99	.12	$0.12
$$,$$$,999.99	-1234.56	$1,234.56
++,+++,999.99	-123456.789	-123,456.78
$$,$$$,$$$.99CR	-1234567	$1,234,567.00CR
$$,$$$,$$$.99DB	+1234567	$1,234,567.00
++,+++,+++.+++	0000.00	

Zero Suppression and Replacement Editing

Zero-suppression and replacement editing means the suppression of leading zeros in numeric character positions and is indicated by the use of the alphabetic character Z or the character * in the PICTURE character string. If Z is used, the replacement character will be the space; if * is used, the replacement character will be *.

The symbols + - * Z and $ are mutually exclusive as floating replacement characters in a given PICTURE character string.

Each suppression symbol is counted in determining the size of an item.

Zero suppression and replacement editing is indicated in a PICTURE character string by using a string of one or more of either allowable symbol to represent leading numeric character positions, which are to be replaced when the associated character position in the data contains a zero. Any of the simple insertion characters embedded in the string of symbols or to the immediate right of this string are part of the string. Simple insertion or fixed insertion editing characters to the left of the string are not included.

In a PICTURE character string, there are only two ways of representing zero suppression:

1. Any or all of the leading numeric character positions to the left of the decimal point are represented by suppression symbols.

2. All of the numeric character positions in the PICTURE character string are represented by suppression symbols.

If the suppression symbols appear only to the left of the decimal point, any leading zero in the data which appears in a character position corresponding to a suppression symbol in the string is replaced by the replacement character. Suppression terminates at the first nonzero digit in the data or at the decimal point, whichever is encountered first.

If all numeric character positions in the PICTURE character string are represented by suppression symbols, and the value of the data is not zero, the result is the same as if the suppression characters were only to the left of the decimal point.

If the value of the data is zero, the entire data item will be spaces if the suppression symbol is Z, or it will be asterisks (except for the actual decimal point) if the suppression symbol is *.

If the value of the data is zero and the asterisk is used as the suppression symbol, zero suppression editing overrides the function of the BLANK WHEN ZERO clause, if specified.

Examples of zero suppression and replacement editing:

PICTURE	Value of Data	Edited Result
ZZZZ.ZZ	0000.00	
****.**	0000.00	****.**
ZZZZ.99	0000.00	.00
****.99	0000.00	****.00
ZZ99.99	0000.00	00.00
Z,ZZZ.ZZ+	+123.456	123.45+
*,***.***+	-123.45	**123.45-
,*,***.***+	+12345678.9	*2,345,678.90+
$Z,ZZZ,ZZZ.ZZCR	+12345.67	$ 12,345.67
$B*,***,***.**BBDB	-12345.67	$ ***12,345.67 DB

Program Product Information (Version 3 and Version 4)

SIGN Clause

The SIGN clause specifies the position and mode of representation of the operational sign for a numeric data description entry.

Format (Version 3 and Version 4)
[SIGN IS] { LEADING / TRAILING } [SEPARATE CHARACTER]

The SIGN clause is required only when an explicit description of the properties of the operational sign is necessary.

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

Only one SIGN clause may apply to any given numeric data description entry.

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

When specified, the SIGN clause defines the position and mode of representation of the operational sign for the numeric data description entry to which it applies, or for each signed numeric data description entry subordinate to the group to which it applies.

If the SEPARATE CHARACTER option is not specified, then:

- The operational sign is presumed to be associated with the LEADING or TRAILING digit position, whichever is specified, of the elementary numeric data item. (In this instance, specification of SIGN IS TRAILING is the equivalent of the standard action of the compiler.)
- The character S in the PICTURE character string is not counted in determining the size of the item (in terms of Standard Data Format characters).

If the SEPARATE CHARACTER option is specified, then:

- The operational sign is presumed to be the LEADING or TRAILING character position, whichever is specified, of the elementary numeric data item. This character position is not a digit position.
- The character S in the PICTURE character string is counted in determining the size of the data item (in terms of Standard Data Format characters).
- + is the character used for the positive operational sign.
- - is the character used for the negative operational sign.

- At object time if one of the characters + or - is not present in the data an error occurs, and the program will terminate abnormally.

Every numeric data description entry whose PICTURE contains the character S is a signed numeric data description entry. If the SIGN clause applies to such an entry, and conversion is necessary for purposes of computation, or for comparisons, conversion takes place automatically.

If no SIGN clause applies to a numeric data description entry whose PICTURE character string contains the character S, then the position of the operational sign is determined as explained in the description of the USAGE clause.

Note: ASCII considerations for the SIGN clause are given in Appendix E.

SYNCHRONIZED Clause

The SYNCHRONIZED clause specifies the alignment of an elementary item on one of the proper boundaries in core storage.

Format

```
{SYNCHRONIZED} [LEFT ]
{SYNC        } [RIGHT]
```

The SYNCHRONIZED clause is used to ensure efficiency when performing arithmetic operations on an item.

The SYNCHRONIZED clause may appear only at the elementary level or at level-01. When used at level-01, every elementary item within this level-01 item is synchronized.

If either the LEFT or the RIGHT option is specified, it is treated as comments.

The length of an elementary item is not affected by the SYNCHRONIZED clause.

When the SYNCHRONIZED clause is specified for an item within the scope of an OCCURS clause, each occurrence of the item is synchronized.

When the item is aligned, the character positions between the last item assigned and the current item are known as "slack bytes." These unused character positions are included in the size of any group to which the synchronized elementary item belongs.

The proper boundary used to align the item to be synchronized depends on the format of the item as defined by the USAGE clause.

When the SYNCHRONIZED clause is specified, the following actions are taken:

For a COMPUTATIONAL item:

1. If its PICTURE is in the range of S9 through S9(4), the item is aligned on a halfword (even) boundary.

2. If its PICTURE is in the range of S9(5) through S9(18), the item is aligned on a fullword (multiple of 4) boundary.

For a COMPUTATIONAL-1 item, the item is aligned on a fullword boundary.

For a COMPUTATIONAL-2 item, the item is aligned on a doubleword (multiple of 8) boundary.

For a DISPLAY or COMPUTATIONAL-3 item, the SYNCHRONIZED clause is treated as comments.

Note: In the discussion which follows, the term "computational" refers to COMPUTATIONAL, COMPUTATIONAL-1, and COMPUTATIONAL-2 items.

When the SYNCHRONIZED clause is specified for an item that also contains a REDEFINES clause, the data item that is redefined must have the proper boundary alignment for the data item that REDEFINES it. For example, if the programmer writes:

```
02 A PICTURE X(4).
02 B REDEFINES A  PICTURE S9(9) COMP SYNC.
```

he must ensure that A begins on a fullword boundary.

When the SYNCHRONIZED clause is specified for a computational item that is the first elementary item subordinate to an item that contains a REDEFINES clause, the computational item must not require the addition of slack bytes.

When SYNCHRONIZED is not specified for binary or internal floating-point items, no space is reserved for slack bytes. However, when computation is done on these fields, the compiler generates the necessary instructions to move the items to a work area which has the correct boundary necessary for computation.

In the File Section, the compiler assumes that all level-01 records containing SYNCHRONIZED items are aligned on a doubleword boundary in the buffer. The user must provide the necessary inter-record slack bytes to ensure alignment.

In the Working-Storage Section, the compiler will align all level-01 entries on a doubleword boundary.

For the purposes of aligning COMPUTATIONAL, COMPUTATIONAL-1, and COMPUTATIONAL-2 items in the Linkage Section, all level-01 items are assumed to begin on doubleword boundaries. Therefore, if the user issues a CALL statement he must ensure that such operands of any USING clause within it are correspondingly aligned.

Slack Bytes

There are two types of slack bytes: intra-record slack bytes and inter-record slack bytes.

Intra-record slack bytes are unused character positions preceding each synchronized item in the record.

Inter-record slack bytes are unused character positions added between blocked logical records.

INTRA-RECORD SLACK BYTES: For an output file, or in the Working-Storage Section, the compiler inserts intra-record slack bytes to ensure that all SYNCHRONIZED items are on their proper boundaries. For an input file, or in the Linkage Section, the compiler expects intra-record slack

bytes to be present when necessary to assure the proper alignment of a SYNCHRONIZED item.

Because it is important for the user to know the length of the records in a file, the algorithm the compiler uses to determine whether slack bytes are required and, if they are required, the number of slack bytes to add, is as follows:

- The total number of bytes occupied by all elementary data items preceding the computational item are added together, including any slack bytes previously added.
- This sum is divided by m, where:

  m = 2 for COMPUTATIONAL items of 4-digit length or less

  m = 4 for COMPUTATIONAL items of 5-digit length or more

  m = 4 for COMPUTATIONAL-1 items

  m = 8 for COMPUTATIONAL-2 items

- If the remainder (r) of this division is equal to zero, no slack bytes are required. If the remainder is not equal to zero, the number of slack bytes that must be added is equal to m - r.

These slack bytes are added to each record immediately following the elementary data item preceding the computational item. They are defined as if they were an item with a level number equal to that of the elementary item that immediately precedes the SYNCHRONIZED computational item, and are included in the size of the group which contains them.

For example:

```
01  FIELD-A.
    05  FIELD-B  PICTURE X(5).
    05  FIELD-C.
        10  FIELD-D  PICTURE XX.
       [10  Slack-Bytes  PICTURE X.  Inserted by compiler]
        10  FIELD-E COMPUTATIONAL PICTURE S9(6) SYNCHRONIZED.

01  FIELD-L.
    05  FIELD-M  PICTURE X(5).
    05  FIELD-N  PICTURE XX.
   [05  Slack-Bytes  PICTURE X.  Inserted by compiler]
    05  FIELD-O.
        10  FIELD-P COMPUTATIONAL PICTURE S9(6) SYNCHRONIZED.
```

Slack bytes may also be added by the compiler when a group item is defined with an OCCURS clause and contains within it a SYNCHRONIZED data item with USAGE defined as COMPUTATIONAL, COMPUTATIONAL-1, or COMPUTATIONAL-2. To determine whether slack bytes are to be added, the following action is taken:

- The compiler calculates the size of the group, including all the necessary intra-record slack bytes.
- This sum is divided by the largest m required by any elementary item within the group.
- If r is equal to zero, no slack bytes are required. If r is not equal to zero, m - r slack bytes must be added.

The slack bytes are inserted at the end of each occurrence of the group item containing the OCCURS clause. For example, if a record is defined as follows:

```
01  WORK-RECORD.
    05  WORK-CODE  PICTURE X.
    05  COMP-TABLE OCCURS 10 TIMES.
        10  COMP-TYPE  PICTURE X.
       [10  Ia-Slack-Bytes  PIC XX.  Inserted by compiler]
        10  COMP-PAY PICTURE S9(4)V99 COMP SYNC.
        10  COMP-HRS PICTURE S9(3) COMP SYNC.
        10  COMP-NAME  PICTURE X(5).
```

The record will appear in storage as shown in Figure 7.

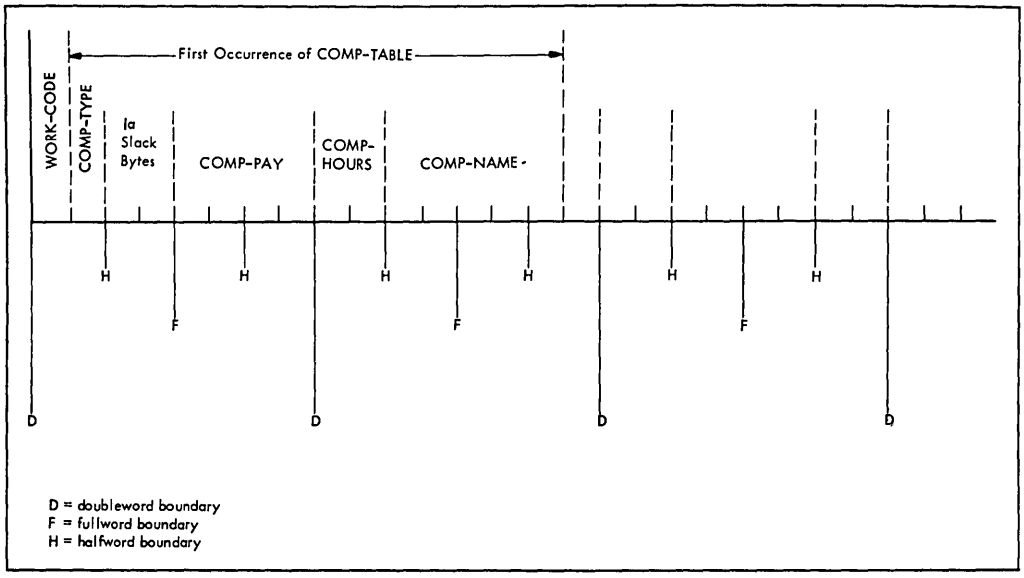


Figure 7. Insertion of Intra-occurrence Slack Bytes

In order to align COMP-PAY and COMP-HRS upon their proper boundaries, the compiler has added two intra-occurrence slack bytes (shown above as Ia-Slack-Bytes).

However, without further adjustment, the second occurrence of COMP-TABLE would now begin one byte before a doubleword boundary, and the alignment of COMP-PAY and COMP-HRS would not be valid for any occurrence of the table after the first. Therefore, the compiler must add inter-occurrence slack bytes at the end of the group, as though the record had been written:

```
01  WORK-RECORD.
    05  WORK-CODE.  PICTURE X.
    05  COMP-TABLE OCCURS 10 TIMES.
        10  COMP-TYPE PICTURE X.
       [10  Ia-Slack-Bytes  PIC XX.  Inserted by compiler]
        10  COMP-PAY PICTURE S9(4)V99 COMP SYNC.
        10  COMP-HRS PICTURE S9(3)  COMP SYNC.
        10  COMP-NAME PICTURE X(5).
       [10  Ie-Slack-Bytes  PIC XX.  Inserted by compiler]
```

so that the second (and each succeeding) occurrence of COMP-TABLE begins one byte beyond a doubleword boundary. The storage layout for the first occurrences of COMP-TABLE will now appear as shown in Figure 8.

Each succeeding occurrence within the table will now begin at the same relative position as the first.

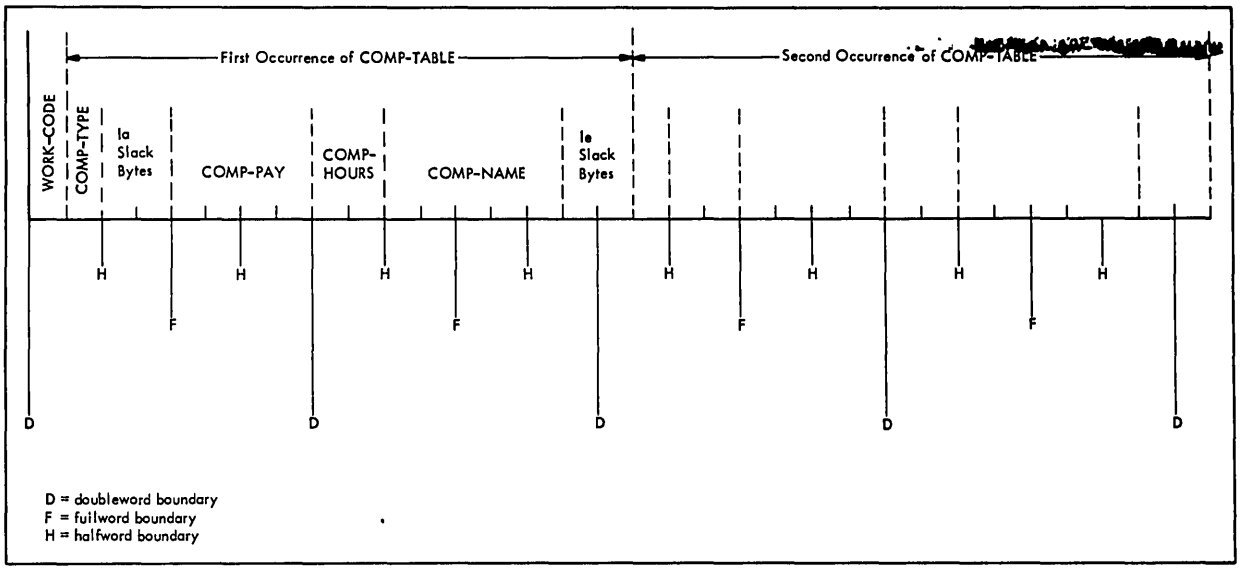


Figure 8. Insertion of Inter-occurrence Slack Bytes

Where SYNCHRONIZED data items defined as COMPUTATIONAL, COMPUTATIONAL-1, or COMPUTATIONAL-2 follow an entry containing an OCCURS DEPENDING ON clause, slack bytes are added on the basis of the field occurring the maximum number of times. If the length of this field is not divisible by the m required for the data, only certain values of the data-name that is the object of the DEPENDING ON option will give proper alignment of the fields. These values are those for which the length of the data field multiplied by the number of occurrences plus the slack bytes that have been calculated based on the maximum number of occurrences is divisible by m.

For example:

```
01  FIELD-A.
    05  FIELD-B  PICTURE 99.
    05  FIELD-C  PICTURE X OCCURS 20 TO 99 TIMES
          DEPENDING ON FIELD-B.
   [05  Slack-Bytes  PICTURE X.  Inserted by compiler]
    05  FIELD-D  COMPUTATIONAL  PICTURE S99 SYNCHRONIZED.
```

In this example, when references to FIELD-D are required, FIELD-B is restricted to odd values only.

```
01  FIELD-A.
    05  FIELD-B  PICTURE 999.
    05  FIELD-C  PICTURE XX OCCURS 20 TO 99 TIMES
          DEPENDING ON FIELD-B.
   [05  Slack-Bytes  PICTURE X.  Inserted by compiler]
    05  FIELD-D  COMPUTATIONAL  PICTURE S99 SYNCHRONIZED.
```

In this example all values of FIELD-B give proper references to FIELD-D.

INTER-RECORD SLACK BYTES: If the file contains blocked logical records that are to be processed in a buffer, and any of the records contain entries defined as COMPUTATIONAL, COMPUTATIONAL-1, or COMPUTATIONAL-2, for which the SYNCHRONIZED clause is specified, the user must add any inter-record slack bytes needed for proper alignment.

The lengths of all the elementary data items in the record, including all intra-record slack bytes, are added. For mode V records, it is necessary to add four bytes for the count field. The total is then divided by the highest value of m for any one of the elementary items in the record.

If r (the remainder) is equal to zero, no inter-record slack bytes are required. If r is not equal to zero, m - r slack bytes are required. These slack bytes may be specified by writing a level-02 FILLER at the end of the record.

If mode U records are being read backwards, doubleword boundary alignment of the input/output buffer will be obtained only if the lengths of the logical records are divisible by eight.

Example: The following example shows the method of calculating both intra-record and inter-record slack bytes. Consider the following record description:

```
01  COMP-RECORD.
    05  A-1      PICTURE X(5).
    05  A-2      PICTURE X(3).
    05  A-3      PICTURE X(3).
    05  B-1      PICTURE S9999  USAGE COMP SYNCHRONIZED.
    05  B-2      PICTURE S99999  USAGE COMP SYNCHRONIZED.
    05  B-3      PICTURE S9999  USAGE COMP SYNCHRONIZED.
```

The number of bytes in A-1, A-2, and A-3 total 11. B-1 is a 4-digit COMPUTATIONAL item and therefore one intra-record slack byte must be added before B-1. With this byte added, the number of bytes preceding B-2 total 14. Since B-2 is a COMPUTATIONAL item of five digits in length, two intra-record slack bytes must be added before it. No slack bytes are needed before B-3.

The revised record description entry now appears as:

```
01  COMP-RECORD.
    05  A-1      PICTURE X(5).
    05  A-2      PICTURE X(3).
    05  A-3      PICTURE X(3).
   [05  Slack-Byte-1  PICTURE X.  Inserted by compiler]
    05  B-1      PICTURE S9999 USAGE COMP SYNCHRONIZED.
   [05  Slack-Byte-2  PICTURE XX.  Inserted by compiler]
    05  B-2      PICTURE S99999 USAGE COMP SYNCHRONIZED.
    05  B-3      PICTURE S9999  USAGE COMP SYNCHRONIZED.
```

There is a total of 22 bytes in COMP-RECORD, but from the rules given in the preceding discussion, it appears that m = 4 and r = 2. Therefore, to attain proper alignment for blocked records, the user must add two inter-record slack bytes at the end of the record.

The final record description entry appears as:

```
01  COMP-RECORD.
    05  A-1      PICTURE X(5).
    05  A-2      PICTURE X(3).
    05  A-3      PICTURE X(3).
   [05  Slack-Byte-1  PICTURE X.  Inserted by compiler]
    05  B-1      PICTURE S9999 USAGE COMP SYNCHRONIZED.
   [05  Slack-Byte-2  PICTURE XX.  Inserted by compiler]
    05  B-2      PICTURE S99999 USAGE COMP SYNCHRONIZED.
    05  B-3      PICTURE S9999 USAGE COMP SYNCHRONIZED.
    05  FILLER PICTURE XX.  [Inter-record slack bytes added by user]
```

USAGE Clause

The USAGE clause specifies the manner in which a data item is represented in core storage.

Format 1

```
[USAGE IS]  { DISPLAY                  }
            { COMPUTATIONAL   }        }
            { COMP            }        }
            { COMPUTATIONAL-1 }        }
            { COMP-1          }        }
            { COMPUTATIONAL-2 }        }
            { COMP-2          }        }
            { COMPUTATIONAL-3 }        }
            { COMP-3          }        }
            { DISPLAY-ST               }
            { INDEX                    }
```

Format 2 (Version 3 and Version 4)

```
[USAGE IS]  { COMPUTATIONAL-4 }
            { COMP-4          }
```

The USAGE clause can be specified at any level of data description. However, if the USAGE clause is written at a group level, it applies to each elementary item in the group. The usage of an elementary item cannot contradict the usage of a group to which an elementary item belongs.

This clause specifies the manner in which a data item is represented in core storage. However, the specifications for some statements in the Procedure Division may restrict the USAGE clause of the operand referred to.

If the USAGE clause is not specified for an elementary item, or for any group to which the item belongs, it is assumed that the usage is DISPLAY.

Note: ASCII considerations for the USAGE Clause are given in Appendix E.

DISPLAY Option

The DISPLAY option can be explicit or implicit. It specifies that the data item is stored in character form, one character per eight-bit byte. This corresponds to the form in which information is represented for initial card input or for final printed or punched output. USAGE IS DISPLAY is valid for the following types of items:

- alphabetic
- alphanumeric
- alphanumeric edited
- numeric edited
- external decimal
- external floating-point

Alphabetic, alphanumeric, alphanumeric edited, and numeric edited items are discussed in the description of the PICTURE clause.

External Decimal Items: These items are sometimes referred to as zoned decimal items. Each digit of a number is represented by a single byte. The four high-order bits of each byte are zone bits; the four high-order bits of the low-order byte represent the sign of the item. The four low-order bits of each byte contain the value of the digit. When external decimal items are used for computations, the compiler performs the necessary conversions.

The maximum length of an external decimal item is 18 digits.

Examples of external decimal items and their internal representation are shown in Table 8.1.

External Floating-Point Items: The PICTURE of an external floating-point item is in the following form:

{±}mantissaE{±}exponent

(See the discussion of the PICTURE clause for the valid combination of symbols.)

The mantissa is the decimal part of the number.

The exponent specifies a power of ten that is used as a multiplier.

The value of an external floating-point number is the mantissa multiplied by the power of ten expressed by the exponent. The magnitude of a number represented by a floating-point item must be greater than $5.4 \times (10^{-79})$ but must not exceed $.72 \times (10^{76})$.

When used as a numeric operand an external floating-point number is scanned at object time, and converted to the equivalent internal floating-point values. In this form, the number is used in arithmetic operations. (See COMPUTATIONAL-1 and COMPUTATIONAL-2 options.)

An example of an external floating-point number and its internal representation is shown in Table 9.

The Computational Options

A COMPUTATIONAL, COMPUTATIONAL-1, COMPUTATIONAL-2, COMPUTATIONAL-3, or COMPUTATIONAL-4 item represents a value to be used in arithmetic operations and must be numeric. If the USAGE of any group item is described with any of these options, it is the elementary items within this group which have that USAGE. The group item itself cannot be used in computations.

COMPUTATIONAL OPTION: This option is specified for binary data items. Such items have a decimal equivalent consisting of the decimal digits 0 through 9, plus a sign.

The amount of storage occupied by a binary item depends on the number of decimal digits defined in its PICTURE clause:

Digits in PICTURE Clause	Storage Occupied
1 through 4	2 bytes (halfword)
5 through 9	4 bytes (fullword)
10 through 18	8 bytes (2 fullwords -- not necessarily a doubleword)

The leftmost bit of the storage area is the operational sign.

The PICTURE of a COMPUTATIONAL item may contain only 9's, the operational sign character S, the implied decimal point V, and one or more P's.

Note: The COMPUTATIONAL option is system dependent and normally is assigned to representations that yield the greatest efficiency when performing arithmetic operations on that system; for this compiler, the COMPUTATIONAL option is binary.

An example of a binary item is shown in Table 9.

COMPUTATIONAL-1, COMPUTATIONAL-2 OPTIONS: These options are specified for internal floating-point items. Such an item is equivalent to an external floating-point item in capability and purpose. Such items occupy either 4 or 8 bytes of storage.

The sign of the fraction (mantissa) is the leftmost bit in either format.

The exponent appears in bit positions 1 through 7.

The fraction -- equivalent to the mantissa -- appears in the rightmost bytes.

COMPUTATIONAL-1 is specified for short-precision internal floating point items. Such items are four bytes in length, aligned on a fullword boundary. The fraction occupies the rightmost three bytes of the item.

COMPUTATIONAL-2 is specified for long-precision floating-point items. Such items are eight bytes in length, and are aligned on a doubleword boundary. The fraction occupies the rightmost seven bytes of the item.

No PICTURE clause may be associated with an internal floating-point item.

If a VALUE clause is associated with an internal floating-point item, the literal must be a floating-point literal (for example, VALUE IS 7.14E+2).

Examples of internal floating-point items, and their internal representation, are shown in Table 9.

COMPUTATIONAL-3 OPTION: This option is specified for internal decimal items. Such an item appears in storage in packed decimal format. There are two digits per byte, with the sign contained in the low-order four bits of the rightmost byte. Such an item may contain any of the digits 0 through 9, plus a sign, representing a value not exceeding 18 decimal digits.

For internal decimal items whose PICTURE does not contain an S, the sign position is occupied by a bit configuration that is interpreted as positive, but that does not represent an overpunch.

The PICTURE of a COMPUTATIONAL-3 item may contain only 9's, the operational sign character S, the assumed decimal point V, and one or more P's.

Examples of internal decimal items and their internal representation are shown in Table 9.

Program Product Information (Version 3 and Version 4)

COMPUTATIONAL-4 OPTION: This option (Format 2) is specified for system-independent binary items. For this compiler, it is the equivalent of COMPUTATIONAL.

USAGE DISPLAY-ST is discussed in the chapter on Sterling Currency.

USAGE INDEX is discussed in the chapter on Table Handling.

Table 9. Internal Representation of Numeric Items (Part 1 of 2)

Item	Value	Description	Internal Representation*
External Decimal	-1234	DISPLAY PICTURE 9999	|Z1|Z2|Z3|F4| byte
		DISPLAY PICTURE S9999	|Z1|Z2|Z3|D4| byte Note that, internally, the D4, which represents -4, is the same bit configuration as the EBCDIC character M.
		(Version 3 & 4) DISPLAY PICTURE S9999 SIGN TRAILING SEPARATE	|Z1|Z2|Z3|Z4|60| byte
		(Version 3 & 4) DISPLAY PICTURE S9999 SIGN LEADING	|D1|Z2|Z3|Z4| byte Note that internally the D1, which represents -1, is the same bit configuration as the EBCDIC character J.

*Codes used in this column are as follows:
Z = zone, equivalent to hexadecimal F, bit configuration 1111

Hexadecimal numbers and their equivalent meanings are:
F = non-printing plus sign (treated as an absolute value)
C = internal equivalent of plus sign, bit configureation 1100
D = internal equivalent of minus sign, bit configuration 1101

S = sign position of a numeric field; internally,
1 in this position means the number is negative
0 in this position means the number is positive

b = a blank

60 = minus sign, bit configuration 0110 0000

Table 9. Internal Representation of Numeric Items (Part 2 of 2)

Item	Value	Description	Internal Representation*
Binary	-1234	COMPUTATIONAL PICTURE S9999	|1111|1011|0010|1110| S byte Note that, internally, negative binary numbers appear in two's complement form.
Internal Decimal	+1234	COMPUTATIONAL-3 PICTURE 9999	|01|23|4F| byte
		COMPUTATIONAL-3 PICTURE S9999	|01|23|4C| byte
External Floating-point	+12.34E+2	DISPLAY PICTURE +99.99E-99	|+|1|2|.|3|4|E|b|0|2| byte
Internal Floating-point		COMPUTATIONAL-1	|S|Exponent|Fraction| 0 1 7 8 31
		COMPUTATIONAL-2	|S|Exponent|Fraction| 0 1 7 8 63

*Codes used in this column are as follows:
Z = zone, equivalent to hexadecimal F, bit configuration 1111

Hexadecimal numbers and their equivalent meanings are:
F = non-printing plus sign (treated as an absolute value)
C = internal equivalent of plus sign, bit configuration 1100
D = internal equivalent of minus sign, bit configuration 1101

S = sign position of a numeric field; internally,
1 in this position means the number is negative
0 in this position means the number is positive

b = a blank

VALUE Clause

The VALUE clause is used to define the initial value of a Working-Storage item or the value associated with a condition-name.

There are two formats of the VALUE clause:

```
Format 1

VALUE IS literal
```

```
Format 2

{VALUE IS  }
{VALUES ARE}    literal-1 [THRU literal-2]

      [literal-3 [THRU literal-4]]...
```

The VALUE clause must not be stated for any item whose size, explicit or implicit, is variable.

A figurative constant may be substituted wherever a literal is specified.

Rules governing the use of the VALUE clause differ with the particular section of the Data Division in which it is specified.

1. In the File Section and the Linkage Section, the VALUE clause must be used only in condition-name entries. However, this compiler will accept the VALUE clause in other File Section and Linkage Section entries, and treat it as comments.

Program Product Information (Version 4)

For Version 4, in the Communications Section the VALUE clause should be used only in condition-name entries. When it is used in other entries, the Version 4 Compiler accepts the clause and treats it as comments.

2. In the Working-Storage Section, the VALUE clause must be used in condition-name entries, and it may also be used to specify the initial value of any data item. It causes the item to assume the specified value at the start of execution of the object program. If the VALUE clause is not used in an item's description, the initial value is unpredictable.

3. In the Report Section, the VALUE clause causes the report data item to assume the specified value each time its report group is presented. This clause may be used only at an elementary level in the Report Section. The Report Section is discussed in detail in the "Report Writer" chapter.

The VALUE clause must not be specified in a data description entry that contains an OCCURS clause or in an entry that is subordinate to an entry containing an OCCURS clause. This rule does not apply to condition-name entries.

Within a given record description, the VALUE clause must not be used in a data description entry that is subsequent to a data description entry which contains an OCCURS clause with a DEPENDING ON phrase.

The VALUE clause must not be specified in a data description entry that contains a REDEFINES clause or in an entry that is subordinate to an entry containing a REDEFINES clause. This rule does not apply to condition-name entries.

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 USAGE of the individual elementary or group items contained within this group. The VALUE clause then cannot be specified at subordinate levels within this group.

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

The VALUE clause must not be specified for external floating-point items.

The following rules apply:

1. If the item is numeric, all literals in the VALUE clause must be numeric literals. If the literal defines the value of a Working-Storage item, the literal is aligned according to the rules for numeric moves, except that the literal must not have a value that would require truncation of nonzero digits.

   If the item is internal floating-point, the literal must be a floating-point literal (for example, VALUE IS 7.14E+2).

2. If the item is alphabetic or alphanumeric (elementary or group), all literals in the VALUE clause must be nonnumeric literals. The literal is aligned according to the alignment rules (see "JUSTIFIED Clause"), except that the number of characters in the literal must not exceed the size of the item.

3. All numeric literals in a VALUE clause of an item must have a value that is within the range of values indicated by the PICTURE clause for that item. For example, for PICTURE 99PPP, the literal must be within the range 1000 through 99000 or zero. For PICTURE PPP99, the literal must be within the range .00000 through .00099.

4. The function of the editing characters in a PICTURE clause is ignored in determining the initial appearance of the item described. However, editing characters are included in determining the size of the item.

5. A maximum of 65,535 bytes may be initialized by means of a single VALUE clause.

Format 1 of the VALUE clause must not conflict with other clauses either in the data description of the item or in the data descriptions within the hierarchy of this term.

Format 2 of the VALUE clause is used to describe a condition-name. Each condition-name requires a separate level-88 entry. A Format 2 VALUE clause associates a value, values, or range of values with the condition-name.

In a condition-name entry, the VALUE clause is required and is the only clause permitted in the entry. A condition-name is a name assigned by the user to the values a data item may assume during object program execution. A condition-name must be formed according to the rules for data-name formation. The condition-name entries for a

particular conditional variable must follow the conditional variable. Hence, a level-88 entry must always be preceded either by the entry for the conditional variable or by another level-88 entry (in the case of several consecutive condition-names pertaining to a given item).

The THRU option assigns a range of values to a condition-name. Wherever used, literal-1 must be less than literal-2, literal-3 less than literal-4, etc.

The type of literal in a condition-name entry must be consistent with the data type of the conditional variable. In the following example, CITY-COUNTY-INFO, COUNTY-NO, and CITY are conditional variables; the associated condition-names immediately follow the level-number 88. The PICTURE associated with COUNTY-NO limits the condition-name value to a 2-digit numeric literal. The PICTURE associated with CITY limits the condition-name value to a 3-character nonnumeric literal. Any values for the condition-names associated with CITY-COUNTY-INFO cannot exceed 5 characters, and the literal (since this is a group item) must be nonnumeric:

```
05  CITY-COUNTY-INFO.
        88  BRONX                VALUE "03NYC".
        88  BROOKLYN             VALUE "24NYC".
        88  MANHATTAN            VALUE "31NYC".
        88  QUEENS               VALUE "41NYC".
        88  STATEN-ISLAND        VALUE "43NYC".
    10  COUNTY-NO           PICTURE 99.
        88  DUTCHESS             VALUE 14.
        88  KINGS                VALUE 24.
        88  NEW-YORK             VALUE 31.
        88  RICHMOND             VALUE 43.
    10  CITY                PICTURE X(3).
        88  BUFFALO              VALUE "BUF".
        88  NEW-YORK-CITY        VALUE "NYC".
        88  POUGKEEPSIE          VALUE "POK".
05  POPULATION...
```

Every condition-name pertains to an item in such a way that the condition-name may be qualified by the name of the item and the item's qualifiers. The use of condition-names in conditions is described in "Conditions."

A condition-name may pertain to an item (a conditional variable) requiring subscripts. In this case, the condition-name, when written in the Procedure Division, must be subscripted according to the requirements of the associated conditional variable.

A condition-name can be associated with any elementary or group item except the following:

1. A level-66 item.

2. A group containing items with descriptions which include JUSTIFIED, SYNCHRONIZED, or USAGE other than DISPLAY.

3. An index data item (see "Table Handling").

RENAMES Clause

The RENAMES clause permits alternate, possibly overlapping, groupings of elementary data.

Format
66 data-name-1 RENAMES data-name-2 [THRU data-name-3]

One or more RENAMES entries can be written for a logical record.

All RENAMES entries associated with a given logical record must immediately follow its last data description entry.

Data-name-2 and data-name-3 must be the names of elementary items or groups of elementary items in the associated logical record and cannot be the same data-name. Data-name-3 cannot be subordinate to data-name-2.

When data-name-3 is not specified, data-name-2 can be either a group item or an elementary item. When data-name-2 is a group item, data-name-1 is treated as a group item, and when data-name-2 is an elementary item, data-name-1 is treated as an elementary item.

When data-name-3 is specified, data-name-1 is a group item that includes all elementary items:

1. Starting with data-name-2 (if it is an elementary item); or starting with the first elementary item within data-name-2 (if it is a group item), and
2. Ending with data-name-3 (if it is an elementary item); or ending with the last elementary item within data-name-3 (if it is a group item).

A level-66 entry cannot rename another level-66 entry nor can it rename a level-77, level-88, or level-01 entry.

Data-name-1 cannot be used as a qualifier and can be qualified only by the names of the level-01 or FD entries.

Program Product Information (Version 4)

In the Communication Section, data-name-1 can be qualified only by the names of the level-01 or CD entries.

Both data-name-2 and data-name-3 may be qualified.

Neither data-name-2 nor data-name-3 may have an OCCURS clause in its data description entry nor may either of them be subordinate to an item that has an OCCURS clause in its data description entry.

Data-name-2 must precede data-name-3 in the record description; after any associated redefinition, the beginning point of the area described by data-name-3 must logically follow the beginning point of the area described by data-name-2.

For example, the following Working-Storage record is incorrect:

```
 01  ERR-REC.
     05  GROUP-A.
         10  FIELD-1A.
             15  ITEM-1A  PICTURE XXXX.
             15  ITEM-2A  PICTURE XXX.
         10  FIELD-2A.
             15  ITEM-3A  PICTURE XXX.
             15  ITEM-4A  PICTURE XXX.
     05  GROUP-B REDEFINES GROUP-A.
         10  FIELD-1B.
             15  ITEM-1B  PICTURE XX.
             15  ITEM-2B  PICTURE XXX.
             15  ITEM-3B  PICTURE XX.
         10  FIELD-2B.
             15  ITEM-4B  PICTURE XX.
             15  ITEM-5B  PICTURE XX.
             15  ITEM-6B  PICTURE XX.
 66  NEW-ERR-REC RENAMES ITEM-3A THRU ITEM-2B.
```

Although ITEM-3A precedes ITEM-2B in the record description, ITEM-2B logically precedes ITEM-3A in storage. Thus this example is incorrect.

The following shows the corrected Working-Storage record:

```
 01  CORRECTED-RECORD.
     05  GROUP-A.
         10  FIELD-1A.
             15  ITEM-1A  PICTURE XX.
             15  ITEM-2A  PICTURE XXX.
             15  ITEM-3A  PICTURE XX.
         10  FIELD-2A.
             15  ITEM-4A  PICTURE XX.
             15  ITEM-5A  PICTURE XX.
             15  ITEM-6A  PICTURE XX.
     05  GROUP-B REDEFINES GROUP-A.
         10  FIELD-1B.
             15  ITEM-1B  PICTURE XXXX.
             15  ITEM-2B  PICTURE XXX.
         10  FIELD-2B.
             15  ITEM-3B  PICTURE XXX.
             15  ITEM-4B  PICTURE XXX.
 66  NEW-REC RENAMES ITEM-2A THRU ITEM-3B.
```

In this example ITEM-2A precedes ITEM-3B both in the record description and logically in storage.

The following example shows how the RENAMES clause might be used in an actual program:

```
01  OUT-REC.
    05  FIELD-X.
        10  SUMMARY-GROUPX.
            15  FILE-1 PICTURE X.
            15  FILE-2 PICTURE X.
            15  FILE-3 PICTURE X.
    05  FIELD-Y.
        10  SUMMARY-GROUPY.
            15  FILE-1 PICTURE X
            15  FILE-2 PICTURE X.
            15  FILE-3 PICTURE X.
    05  FIELD-Z.
        10  SUMMARY-GROUPZ.
            15  FILE-1 PICTURE X.
            15  FILE-2 PICTURE X.
            15  FILE-3 PICTURE X.
66  SUM-X RENAMES FIELD-X.
66  SUM-XY RENAMES FIELD-X THRU FIELD-Y.
66  SUM-XYZ RENAMES FIELD-X THRU FIELD-Z.
```

In the Procedure Division, the programmer may wish, for example, to do a complete tally of files in each field of the foregoing record. If all active files are represented by an A and all inactive files are represented by an I, the statement

```
EXAMINE SUM-XYZ TALLYING ALL "A"
```

would accomplish this purpose. The two additional RENAMES entries (SUM-X and SUM-XY) allow a less inclusive tally, if desired. (The EXAMINE statement is discussed in "Procedure Division.")

- ORGANIZATION OF THE PROCEDURE DIVISION
- ARITHMETIC EXPRESSIONS
- CONDITIONS
- CONDITIONAL STATEMENTS
- DECLARATIVES
- ARITHMETIC STATEMENTS
- PROCEDURE BRANCHING STATEMENTS
- DATA-MANIPULATION STATEMENTS
- INPUT/OUTPUT STATEMENTS
- SUBPROGRAM LINKAGE STATEMENTS
- COMPILER-DIRECTING STATEMENTS

ORGANIZATION OF THE PROCEDURE DIVISION

The Procedure Division contains the specific instructions for solving a data processing problem. COBOL instructions are written in statements, which may be combined to form sentences. Groups of sentences may form paragraphs, and paragraphs may be combined to form sections.

The Procedure Division must begin with the header PROCEDURE DIVISION followed by a period and a space unless Subprogram Linkage is used. In this case, the Procedure Division header in a called program may optionally contain a USING clause preceding the period (see "Subprogram Linkage").

The Procedure Division header is followed, optionally, by Declarative Sections, which are in turn followed by procedures, each made up of statements, sentences, paragraphs, and/or sections, in a syntactically valid format. The end of the Procedure Division (and the physical end of the program) is that physical position in a COBOL source program after which no further procedures appear.

The statement is the basic unit of the Procedure Division. A statement is a syntactically valid combination of words and symbols beginning with a COBOL verb. There are three types of statements: conditional statements containing conditional expressions (that is, tests for a given condition), imperative statements consisting of an

consisting of a compiler-directing verb and its operands.

A sentence is composed of one or more statements. The statements may optionally be separated by semicolons or the word THEN. A sentence must be terminated by a period followed by a space.

Several sentences that convey one idea or procedure may be grouped to form a paragraph. A paragraph must begin with a paragraph-name followed by a period. A paragraph may be composed of one or more successive sentences. A paragraph ends immediately before the next paragraph-name or section-name, at the end of the Procedure Division, or, in the Declarative portion, at the key words END DECLARATIVES.

One or more paragraphs form a section. A section must begin with a section header (section-name followed by the word SECTION, followed by a period; if program segmentation is desired, a space and a priority number followed by a period may be inserted after the word SECTION). The general term procedure-name may refer to both paragraph-names and section-names.

The Procedure Division may contain both declaratives and procedures.

Declarative sections must be grouped at the beginning of the Procedure Division, preceded by the key word DECLARATIVES followed by a period and a space. Declarative sections are concluded by the key words END DECLARATIVES followed by a period and a space. (For a more complete discussion of declarative sections, see "Declaratives.")

A procedure is composed of a paragraph or group of successive paragraphs, or a section or group of successive sections within the Procedure Division. Paragraphs need not be grouped into sections.

If sections are used within the Procedure Division, a section header should immediately follow the Procedure Division header, except when a declarative section is included, in which case the section header should immediately follow END DECLARATIVES. However, if a section header is missing at the beginning of the nondeclarative portion of the

Procedure Division, this compiler processes the source program as though a section head had been written.

A section ends immediately before the next section-name or at the end of the Procedure Division, or, in the Declarative portion of the Procedure Division, immediately before the next section-name or at the words END DECLARATIVES, where END must appear in Area A.

If program segmentation is used, the programmer must divide the entire Procedure Division into named sections. Program segmentation is discussed in "Segmentation."

Execution begins with the first statement of the Procedure Division, excluding declaratives. Statements are then executed in the order in which they are presented for compilation, except where the rules in this chapter indicate some other order.

Structure of the Procedure Division

```
PROCEDURE DIVISION [USING identifier-1 [identifier-2]...].

[[DECLARATIVES.

{section-name SECTION.  USE Sentence.

{paragraph-name.  {sentence}...}...}...

END DECLARATIVES.]

{section-name SECTION [priority].]

{paragraph-name.  {sentence}...}...}...
```

CATEGORIES OF STATEMENTS

There are three categories of statements used in COBOL: conditional statements, imperative statements, and compiler-directing statements.

A conditional statement is a statement containing a condition that is tested (see "Conditions") to determine which of the alternate paths of program flow is to be taken.

An imperative statement specifies that an unconditional action is to be taken by an object program. An imperative statement may also consist of a series of imperative statements.

A compiler-directing statement directs the compiler to take certain actions at compile time.

CONDITIONAL STATEMENTS

COBOL statements used as conditional statements are:

IF	
ON	
ADD, COMPUTE, SUBTRACT, MULTIPLY, DIVIDE	(ON SIZE ERROR)
GO TO	(DEPENDING ON)
READ, SEARCH, RETURN	(AT END)
WRITE	(AT END-OF-PAGE)
READ, START, WRITE, REWRITE	(INVALID KEY)
PERFORM	(UNTIL)
SEARCH	(WHEN)
EXHIBIT	(CHANGED)

Program Product Information (Version 4)

For Version 4, the following additional statements are used as conditional statements:

RECEIVE	(NO DATA)
STRING	(ON OVERFLOW)
UNSTRING	(ON OVERFLOW)

The options in parentheses cause otherwise imperative statements to be treated as conditionals at execution time. A discussion of these options is included as part of the description of the associated imperative statement.

IMPERATIVE STATEMENTS

COBOL verbs used in imperative statements can be grouped into the following categories and subcategories:

A. DECLARATIVES
   - USE

B. PROCEDURAL

   1. Arithmetic
      - ADD
      - COMPUTE
      - DIVIDE
      - MULTIPLY
      - SUBTRACT

   2. Procedure-Branching
      - GO TO
      - ALTER
      - PERFORM
      - STOP
      - EXIT

3. Data-Manipulation
   MOVE
   EXAMINE
   TRANSFORM

Program product Information (Version 4)

For Version 4, the following additional data manipulation statements are allowed:

STRING
UNSTRING

Note: The STRING and UNSTRING statements are described in the separate chapter "String Manipulation".

4. Input/Output
   OPEN
   START
   SEEK
   READ
   WRITE
   REWRITE
   ACCEPT
   DISPLAY
   CLOSE

Program Product Information (Version 4)

For Version 4, the following input/output statements are valid for teleprocessing programs:

RECEIVE
SEND

Note: The RECEIVE and SEND statements are described in the separate chapter "Teleprocessing".

5. Report Writer
   GENERATE
   INITIATE
   TERMINATE

6. Table Handling
   SEARCH
   SET

7. Sort
   SORT
   RETURN
   RELEASE

8. Debugging
   READY TRACE
   RESET TRACE
   EXHIBIT

Note: Debugging, Report Writer, Table Handling, and Sort statements are discussed in separate chapters.

C. SUBPROGRAM LINKAGE

CALL
ENTRY
GOBACK
EXIT (PROGRAM)

Program Product Information (Version 4)

For Version 4, the following additional Subprogram Linkage statement is valid:

CANCEL

COMPILER-DIRECTING STATEMENTS

COBOL verbs used in compiler-directing statements are:

COPY
ENTER
NOTE
DEBUG

Note: The COPY statement is discussed in "Source Program Library Facility." The statements listed in "Format Control of the Source also be considered as compiler-directing statements.

ARITHMETIC EXPRESSIONS

Arithmetic expressions are used as operands of certain conditional and arithmetic statements.

An arithmetic expression may consist of any of the following:

1. An identifier described as a numeric elementary item
2. A numeric literal
3. Identifiers and literals, as defined in items 1 and 2, separated by arithmetic operators
4. Two arithmetic expressions, as defined in items 1, 2, and/or 3, separated by an arithmetic operator
5. An arithmetic expression, as defined in items 1, 2, 3, and/or 4, enclosed in parentheses

Any arithmetic expression may be preceded by a unary + or a unary -.

ARITHMETIC OPERATORS

There are five arithmetic operators that may be used in arithmetic expressions. Each is represented by a specific character or character combination that must be preceded by a space and followed by a space, except that a unary operator must not be preceded by a space when it follows a left parenthesis. This compiler allows the space following the unary operator to be omitted.

Arithmetic Operator	Meaning
+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

Parentheses may be used in arithmetic expressions to specify the order in which elements are to be evaluated.

Expressions within parentheses are evaluated first. When expressions are contained within a nest of parentheses, evaluation proceeds from the least inclusive to the most inclusive set.

When parentheses are not used, or parenthesized expressions are at the same level of inclusiveness, the following hierarchical order is implied:

1. unary + or unary -
2. **
3. * and /
4. + and -

When the order of consecutive operations on the same hierarchical level is not completely specified by parentheses, the order of operation is from left to right.

Table 10 shows permissible symbol pairs. A symbol pair in an arithmetic expression is the occurrence of two symbols that appear in sequence.

Table 10. Permissible Arithmetic Symbol Pairs

First Symbol \ Second Symbol	Variable (identifier or literal)	* / ** + -	unary + unary -	(	)
Variable (identifier or literal)	-	p		-	p
* / ** + -	p	-	p	p	-
unary + or unary -	p	-	-	p	-
(	p	-	p	p	-
)	-	p		-	p

p indicates a permissible pairing
- indicates that the pairing is not permitted

An arithmetic expression may begin only with a left parenthesis, a unary +, a unary -, or a variable, and may end only with a right parenthesis or a variable.

There must be a one-to-one correspondence between left and right parentheses of an arithmetic expression.

CONDITIONS

A condition causes the object program to select between alternate paths of control depending on the truth value of a test. Conditions are used in IF, PERFORM, and SEARCH statements.

A condition is one of the following:

- Class condition
- Condition-name condition
- Relation condition
- Sign condition

Program Product Information (Version 4)

For Version 4 Teleprocessing Programs, the following condition is also valid:

- Message Condition

Note: The Message Condition is discussed in the separate chapter "Teleprocessing".

In addition, there are two constructions that affect the evaluation of conditions. These are:

1. (condition)

   Parentheses may be used to group conditions (see "Compound Conditions").

2. NOT condition

   The construction -- NOT condition -- (where condition is one of the four conditions listed above) is not permitted if the condition itself contains a NOT.

Conditions may be combined through the use of logical operators to form compound conditions (for a full discussion, see "Compound Conditions").

TEST CONDITIONS

A test condition is an expression that, taken as a whole, may be either true or false, depending on the circumstances existing when the expression is evaluated.

There are four types of simple conditions which, when preceded by the word IF, constitute one of the four types of tests: class test, condition-name test, relation test, and sign test.

The construction -- NOT condition -- may be used in any simple test condition to make the relation specify the opposite of what it would express without the word NOT. For example, NOT (AGE GREATER THAN 21) is the opposite of AGE GREATER THAN 21.

Each of the previously mentioned tests, when used within the IF statement, constitutes a conditional statement (see "Conditional Statements").

Class Condition

The class test determines whether data is alphabetic or numeric.

Format
identifier IS [NOT] { NUMERIC / ALPHABETIC }

The operand being tested must be described implicitly or explicitly as USAGE DISPLAY or USAGE COMPUTATIONAL-3.

A numeric data item consists of the digits 0 through 9, with or without an operational sign.

The identifier being tested is determined to be numeric only if the contents consist of any combination of the digits 0 through 9. If the PICTURE of the identifier being tested does not contain an operational sign, the identifier being tested is determined to be numeric only if the contents are numeric and an operational sign is not present. If its PICTURE does contain an operational sign, the identifier being tested is determined to be numeric only if the contents are numeric and a valid operational sign is present. Valid operational signs are hexadecimal F, C, and D.

Program Product Information (Version 3 and Version 4)

For numeric data items described with the SEPARATE SIGN clause, valid operational signs are hexadecimal 4E and 60.

The NUMERIC test cannot be used with an identifier described as alphabetic.

An alphabetic data item consists of the space character and the characters A through Z.

The identifier 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.

The ALPHABETIC test cannot be used with an identifier described as numeric.

Table 11 shows valid forms of the class test.

Table 11. Valid Forms of the Class Test

Type of Identifier	Valid Forms of the Class Test	
Alphabetic	ALPHABETIC	NOT ALPHABETIC
Alphanumeric, Alphanumeric Edited, Numeric Edited	ALPHABETIC NUMERIC	NOT ALPHABETIC NOT NUMERIC
External-Decimal or Internal-Decimal	NUMERIC	NOT NUMERIC

Condition-name Condition

The condition-name condition causes a conditional variable to be tested to determine whether or not its value is equal to one of the values associated with condition-name.

Format
condition-name

An example of the use of the condition-name condition follows:

```
 05  MARITAL-STATUS  PICTURE 9.
     88  SINGLE VALUE 1.
     88  MARRIED VALUE 2.
     88  DIVORCED VALUE 3.
```

MARITAL-STATUS is the conditional variable; SINGLE, MARRIED, and DIVORCED are condition-names. Only one of the conditions specified by condition-name can be present for individual records in the file. To determine the marital status of the individual whose record is being processed, IF SINGLE... can be coded, and its evaluation as true or false determines the subsequent path the object program takes.

A condition-name is used in conditions as an abbreviation for the relation condition, since the associated condition-name is equal to only one of the values (or ranges of values) assigned to that conditional variable. That is, to determine whether the condition SINGLE is present, IF MARITAL-STATUS = 1... would have the same effect as using the condition-name test IF SINGLE ....

If the condition-name is associated with a range of values (or with several ranges of values), the conditional variable is tested to determine whether or not its value falls within the range(s), 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.

(An example of both group and elementary condition-name entries is given in the description for the VALUE clause in "Data Division".)

Relation Condition

A relation condition causes a comparison of two operands, either of which may be an identifier, a literal, or an arithmetic expression.

Format

```
{identifier-1          }
{literal-1             }  relational-operator
{arithmetic-expression-1}

     {identifier-2          }
     {literal-2             }
     {arithmetic-expression-2}
```

The first operand is called the subject of the condition; the second operand is called the object of the condition.

The subject and object may not both be literals.

The subject and object must have the same USAGE, except when two numeric operands are compared.

A relational-operator specifies the type of comparison to be made in a relation condition. The meaning of the relational operators is shown in Table 12.

Table 12. Relational-Operators and Their Meanings

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

The word TO in the EQUAL TO relational operator is required; however, this compiler allows the word TO to be omitted.

The relational-operator must be preceded by, and followed by, a space.

COMPARISON OF NUMERIC OPERANDS: For operands whose class is numeric, a comparison is made with respect to the algebraic value of the operands.

Zero is considered a unique value, regardless of sign.

Comparison of numeric 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 for one numeric and one nonnumeric operand, a comparison is made with respect to the binary collating sequence of the characters in the EBCDIC set.

The EBCDIC collating sequence, in ascending order, is:

1.		(space)
2.	.	(period or decimal point)
3.	<	("less than" symbol)
4.	(	(left parenthesis)
5.	+	(plus sign)
6.	$	(currency symbol)
7.	*	(asterisk)
8.	)	(right parenthesis)
9.	;	(semicolon)
10.	-	(hyphen or minus symbol)
11.	/	(stroke, virgule, slash)
12.	,	(comma)
13.	>	("greater than" symbol)
14.	'	(apostrophe or single quotation mark)
15.	=	(equal sign)
16.	"	(quotation mark)
17-42.	A through Z	
43-52.	0 through 9	

(The complete EBCDIC collating sequence is given in the publication IBM System/360 Reference Data, Form X20-1703.)

If one of the operands is described as numeric, it is treated as though it were moved to an alphanumeric data item of the same size and the contents of this alphanumeric data item were then compared to the nonnumeric operand (see "MOVE Statement").

The size of an operand is the total number of characters in the operand. All group items are treated as nonnumeric operands.

Numeric and nonnumeric operands may be compared only when their USAGE is the same, implicitly or explicitly.

However, this compiler allows a group item to be compared to a numeric item even if the USAGE of the numeric item is other than DISPLAY.

There are two cases of nonnumeric comparison to consider: operands of equal size and operands of unequal size.

1. Comparison of Operands of Equal Size

   Characters in corresponding character positions of the two operands are compared from the high-order end through the low-order end. The high-order end is the leftmost position; the low-order end is the rightmost character position.

   If all pairs of characters compare equally through the last pair, the operands are considered equal when the low-order end is reached.

   If a pair of unequal characters is encountered, the two characters are compared to determine their relative position in the collating sequence. The operand that contains the character higher in the collating sequence is considered to be the greater operand.

2. Comparison of Operands of Unequal Size

   If the operands are of unequal size, comparison proceeds as though the shorter operand were extended on the right by a sufficient number of spaces to make the operands of equal size.

COMPARISONS INVOLVING INDEX-NAMES AND/OR INDEX DATA ITEMS: The comparison of two index-names is equivalent to the comparison of their corresponding occurrence numbers.

In the comparison of an index data item with an index-name or with another index data item, the actual values are compared without conversion.

The comparison of an index-name with a numeric item is permitted if the numeric item is an integer. The numeric integer is treated as an occurrence number. All other comparisons involving an index-name or index data item are not allowed. (For a discussion of indexing, see "Table Handling.")

Permissible comparisons are shown in Table 13.

Table 13. Permissible Comparisons

First Operand \ Second Operand	GR	AL	AN	ANE	NE	FC* NNL	ZR NL	ED	BI	ID	EF	IF	SR	SN	IN	IDI
Group (GR)	NN	NN	NN	NN	NN	NN	NN	NN	NN	NN	NN	NN	NN	NN		
Alphabetic (AL)	NN	NN	NN	NN	NN	NN	NN	NN			NN		NN	NN		
Alphanumeric (AN)	NN	NN	NN	NN	NN	NN	NN	NN			NN		NN	NN		
Alphanumeric Edited (ANE)	NN	NN	NN	NN	NN	NN	NN	NN			NN		NN	NN		
Numeric Edited (NE)	NN	NN	NN	NN	NN	NN	NN	NN			NN		NN	NN		
Figurative Constant (FC)* and Nonnumeric Literal (NNL)	NN	NN	NN	NN	NN			NN			NN		NN	NN		
Fig. Constant ZERO (ZR) and Numeric Literal (NL)	NN	NN	NN	NN	NN			NU	NU	NU	NU	NU	NN	NU	IO[1]	
External Decimal (ED)	NN	NN	NN	NN	NN	NN	NU	NU	NU	NU	NU	NU	NN	NU	IO[1]	
Binary (BI)	NN						NU	NU	NU	NU	NU	NU		NU	IO[1]	
Internal Decimal (ID)	NN						NU	NU	NU	NU	NU	NU		NU	IO[1]	
External Floating Point (EF)	NN	NN	NN	NN	NN	NN	NU	NU	NU	NU	NU	NU	NN	NU		
Internal Floating Point (IF)	NN						NU	NU	NU	NU	NU	NU		NU		
Sterling Report (SR)	NN	NN	NN	NN	NN	NN	NN	NN			NN		NN	NN		
Sterling Nonreport (SN)	NN	NN	NN	NN	NN	NN	NU	NU	NU	NU	NU	NU	NN	NU		
Index Name (IN)							IO[1]	IO[1]	IO[1]	IO[1]					IO	IV
Index Data Item (IDI)															IV	IV

*FC includes all Figurative Constants except ZERO.
[1]Valid only if the numeric item is an integer.

Notes:
NN = comparison as described for nonnumeric operands
NU = comparison as described for numeric operands
IO = comparison as described for two index-names
IV = comparison as described for index data items

Sign Condition

The sign condition determines whether or not the algebraic value of a numeric operand (i.e., an item described as numeric) is less than, greater than, or equal to zero.

Format

```
{identifier            }          {POSITIVE}
{                      } IS [NOT] {NEGATIVE}
{arithmetic-expression }          {ZERO    }
```

An operand is positive if its value is greater than zero, negative if its value is less than zero, and zero if its value is equal to zero. An unsigned field is always positive or zero.

COMPOUND CONDITIONS

Two or more simple conditions can be combined to form a compound condition. Each simple condition is separated from the next by one of the logical operators AND or OR.

The logical operators must be preceded by a space and followed by a space. The meaning of the logical operators is as follows:

Logical Operator	Meaning
OR	Logical inclusive OR, i.e., either or both are true
AND	Logical conjunction, i.e., both are true
NOT	Logical negation

Figure 9 shows the relationships between the logical operators and simple conditions A and B, where A and B have the following values:

Values for Condition A	Values for Condition B
True	True
False	True
True	False
False	False

A AND B	A OR B	NOT A	NOT (A AND B)	NOT A AND B	NOT (A OR B)	NOT A OR B
True	True	False	False	False	False	True
False	True	True	True	True	False	True
False	True	False	True	False	False	False
False	False	True	True	False	True	True

Figure 9. Logical Operators and the Resulting Values Upon Evaluation

EVALUATION RULES

Logical evaluation begins with the least inclusive pair of parentheses and proceeds to the most inclusive.

If the order of evaluation is not specified by parentheses, the expression is evaluated in the following order:

1. Arithmetic expressions
2. Relational-operators
3. [NOT] condition
4. AND and its surrounding conditions are evaluated first, starting at the left of the expression and proceeding to the right.
5. OR and its surrounding conditions are then evaluated, also proceeding from left to right.

Consider the expression:

```
A IS NOT GREATER THAN B OR A + B IS EQUAL TO C AND D IS POSITIVE
```

This will be evaluated as if it were parenthesized as follows:

```
(A IS NOT GREATER THAN B) OR (((A + B) IS EQUAL TO C) AND (D IS
POSITIVE)).
```

The order of evaluation is as follows:

1. (A + B) is evaluated, giving some intermediate result, for example, x.
2. (A IS NOT GREATER THAN B) is evaluated, giving some intermediate truth value, for example, t1.
3. (x IS EQUAL TO C) is evaluated, giving some intermediate truth value, for example, t2.
4. (D IS POSITIVE) is evaluated, giving some intermediate truth value, for example, t3.
5. (t2 AND t3) is evaluated, giving some intermediate truth value, for example, t4.
6. (t1 OR t4) is evaluated, giving the final truth value, and the result of the expression.

Table 14 shows permissible symbol pairs. A symbol pair in a compound condition is the occurrence of two symbols appearing in sequence.

Table 14. Permissible Symbol Pairs -- Compound Conditions

First Symbol \ Second Symbol	Condition	OR	AND	NOT	(	)
Condition	-	p	p	-	-	p
OR	p	-	-	p	p	-
AND	p	-	-	p	p	-
NOT	p	-	-	-	p	-
(	p	-	-	p	p	-
)	-	p	p	-	-	p

p indicates a permissible pairing
- indicates that the pairing is not permitted

IMPLIED SUBJECTS AND RELATIONAL-OPERATORS

When relation conditions are written in a consecutive sequence, any relation condition except the first may be abbreviated by:

1. The omission of the subject of the relation condition, or
2. The omission of the subject and relational-operator of the relation condition.

Within a sequence of relation conditions, both forms of abbreviation may be used. The effect of using such abbreviations is as if the omitted subject was taken from the most recently stated subject, or the omitted relational-operator was taken from the most recently stated relational-operator.

Format of Implied Subject:

```
...subject  relational-operator   object

{AND}
{OR } [NOT] relational-operator   object...
```

Format of Implied Subject and Relational-operator:

```
...subject   relational-operator  object  { AND } [NOT] object...
                                          { OR  }
```

Ambiguity may result from using NOT in conjunction with abbreviations. When only the subject is implied, NOT is interpreted as a logical operator rather than as part of the relational operator. For example, A NOT > B AND < C is equivalent to A NOT > B AND A < C. When both the subject and the relational operator are implied, NOT is interpreted as part of the relational operator. For example, A NOT > B AND C is equivalent to A NOT > B AND A NOT > C.

The following are examples of implied subjects, and relational-operators. Each example consists of two equivalent statements:

Implied Subject

A = B OR NOT > C	(The subject, A, is implied.)
A = B OR NOT A > C	(The subject, A, is explicit.)

Implied Subject and Relational-Operator

A = B AND C	(Subject and relational-operator, A = , are implied.)
A = B AND A = C	(Subject and relational-operator, A = , are explicit.)
A NOT = B AND C	(Subject and relational-operator, A NOT = , are implied.)
A NOT = B AND A NOT = C	(Subject and relational-operator, A NOT = , are explicit.)

Implied Subject, and Subject and Relational-Operator

A > B AND NOT < C AND D	(Subject, A, is implied in the second condition. Subject, A, and relational-operator, <, are implied in the third condition.)
A > B AND NOT A < C AND A < D	(Subject, A, and relational-operator, <, are explicit.)

The omitted subject is taken from the most recently stated subject, that is, A.

The omitted relational-operator is taken from the most recently stated relational-operator, that is, <.

CONDITIONAL STATEMENTS

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 on this truth value. Conditional statements are listed in "Categories of Statements."

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

Only the IF statement is discussed in this section. Discussion of the other conditional statements is included as part of the description of the associated imperative statements.

IF Statement

The IF statement causes a condition to be evaluated. The subsequent action of the object program depends upon whether the condition is true or false.

Format

```
IF condition THEN { statement-1      } { ELSE       } { statement-2      }
                  { NEXT SENTENCE    } { OTHERWISE  } { NEXT SENTENCE    }
```

The phrase ELSE/OTHERWISE NEXT SENTENCE may be omitted if and only if it immediately precedes the period for the sentence.

When an IF statement is executed, the following action is taken:

1. If the condition is true, the statement immediately following the condition or THEN (statement-1) is executed. (If ELSE/OTHERWISE is omitted, then all imperative statements following the condition and preceding the period for the sentence are considered to be part of statement-1.) Control is then passed implicitly to the next sentence unless GO TO procedure-name is specified in statement-1. If the condition is true and NEXT SENTENCE is written, control passes explicitly to the next sentence.

2. If the condition is false, either the statement following ELSE or OTHERWISE (statement-2) is executed, or, if the ELSE or OTHERWISE option is omitted, the next sentence is executed. If the condition is false and NEXT SENTENCE is written following ELSE, control passes explicitly to the next sentence.

When IF statements are not nested, statement-1 and statement-2 must represent imperative statements.

Nested IF Statements

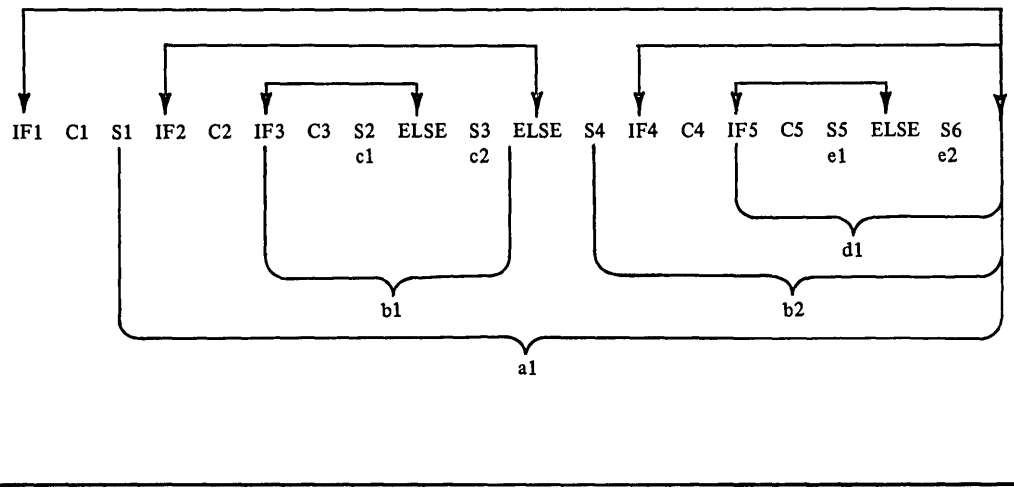
The presence of one or more IF statements within the initial IF statement constitutes a "nested IF statement."

Statement-1 and statement-2 in IF statements may consist of one or more imperative statements and/or a conditional statement. If a conditional statement appears as statement-1 or as part of statement-1, it is said to be nested. Nesting statements is much like specifying subordinate arithmetic expressions enclosed in parentheses and combined in larger arithmetic expressions.

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

In the conditional statement in Figure 10, C stands for condition; S stands for any number of imperative statements; and the pairing of IF and ELSE is shown by the lines connecting them.

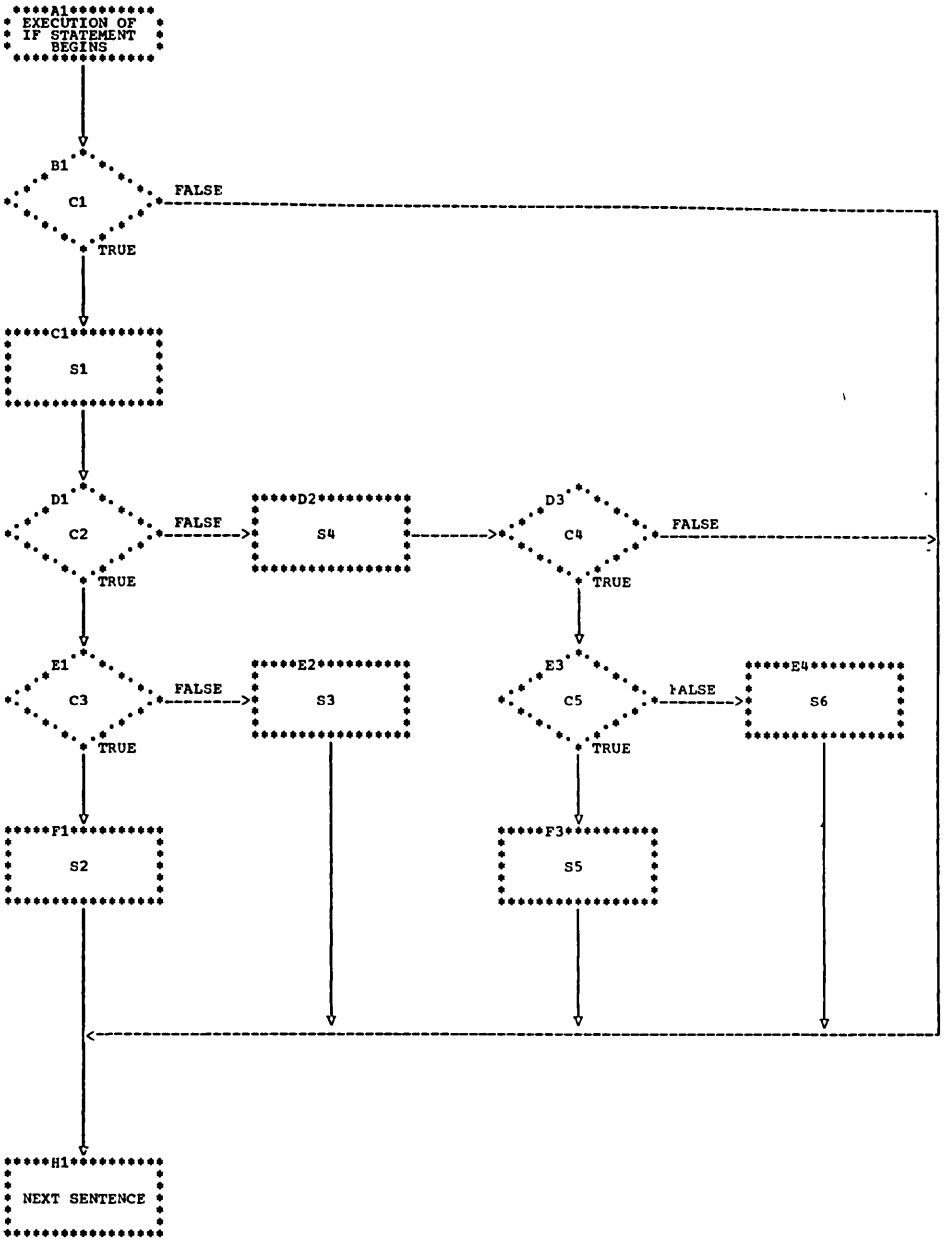
Chart 1 is a flowchart indicating the logical flow of the conditional statement in Figure 10.



a1 – Statement-1 for IF1 (If C1 is false, the next sentence is executed, since there is no ELSE for it.)
b1 – Statement-1 for IF2
b2 – Statement-2 for IF2
c1 – Statement-1 for IF3
c2 – Statement-2 for IF3
d1 – Statement-1 for IF4 (If C4 is false, the next sentence is executed, since there is no ELSE for it.)
e1 – Statement-1 for IF5
e2 – Statement-2 for IF5

Figure 10. Conditional Statements With Nested IF Statements

Chart 1. Logical Flow of Conditional Statement With Nested IF Statements



DECLARATIVES

The Declaratives Section provides a method of including procedures that are invoked non-synchronously; that is, they are executed not as part of the sequential coding written by the programmer, but rather when a condition occurs which cannot normally be tested by the programmer.

Although the system automatically handles checking and creation of standard labels and executes error recovery routines in the case of input/output errors, additional procedures may be specified by the COBOL programmer. The Report Writer feature also uses declarative procedures.

Since these procedures are executed only when labels of a file are to be processed, or at the time an error in reading or writing occurs or when a report group is to be produced, they cannot appear in the regular sequence of procedural statements. They must be written at the beginning of the Procedure Division in a subdivision called DECLARATIVES. A group of declarative procedures constitutes a declarative section. Related procedures are preceded by a USE sentence that specifies their function. A declarative section ends with the occurrence of another section-name with a USE sentence or with the key words END DECLARATIVES.

The key words DECLARATIVES and END DECLARATIVES must each begin in Area A and be followed by a period. No other text may appear on the same line.

General Format

```
PROCEDURE DIVISION.

DECLARATIVES.

{section-name SECTION.  USE sentence.

{paragraph-name.  {sentence} ...  } ...  } ...

END DECLARATIVES.
```

The USE sentence identifies the type of declarative. There are three formats of the USE sentence. Each is associated with one of the following types of procedures:

1. Input/output label handling
2. Input/output error-checking procedures
3. Report writing procedures

A USE sentence, when present, must immediately follow a section header in the Declarative portion of the Procedure Division and must be followed by a period followed by a space. The remainder of the section must consist of one or more procedural paragraphs that define the procedures to be used. The USE sentence itself is never executed; rather, it defines the conditions for the execution of the USE procedure.

Format 1 is used to provide user label-handling procedures. There are two options of Format 1.

Format 1

Option 1

```
USE { BEFORE } STANDARD [BEGINNING] [ REEL ]
    { AFTER  }                      [ FILE ]
                                    [ UNIT ]

    LABEL PROCEDURE ON { {file-name}... }
                       { OUTPUT         } .
                       { INPUT          }
                       { I-O            }
```

Option 2

```
USE { BEFORE } STANDARD [ENDING]    [ REEL ]
    { AFTER  }                      [ FILE ]
                                    [ UNIT ]

    LABEL PROCEDURE ON { {file-name}... }
                       { OUTPUT         } .
                       { INPUT          }
                       { I-O            }
```

When BEFORE is specified, it indicates that nonstandard labels are to be processed. Nonstandard labels may be specified only for tape files.

When AFTER is specified, it indicates that user labels follow standard file labels, and are to be processed.

Note: ASCII considerations for user label-handling procedures are given in Appendix E.

The labels must be listed as data-names in the LABEL RECORDS clause in the file description entry for the file, and must be described as level-01 data items subordinate to the file entry.

If neither BEGINNING nor ENDING is specified, the designated procedures are executed for both beginning and ending labels.

If UNIT, REEL, or FILE is not included, the designated procedures are executed both for REEL or UNIT, whichever is appropriate, and for FILE labels. The REEL option is not applicable to mass storage files. The UNIT option is not applicable to files in the random access mode, since only FILE labels are processed in this mode.

If FILE is specified, the designated procedures are executed at beginning-of-file (on first volume) and/or at end-of-file (on last volume) only. If REEL or UNIT is specified, the designated procedures are executed at beginning-of-volume (on each volume but the first) and/or at end-of-volume (on each volume but the last.) Both BEGINNING and ENDING label processing is executed if BEGINNING or ENDING has not been specified.

The same file-name may appear in different specific arrangements of Format 1. However, appearance of a file-name in a USE statement must

not cause the simultaneous request for execution of more than one USE declarative.

If the file-name option is used, the file description entry for file-name must not specify a LABEL RECORDS ARE OMITTED clause.

The file-name must not represent a sort-file.

The user label procedures are executed as follows when the OUTPUT, INPUT, or I-O options are specified:

- When OUTPUT is specified, only for files opened as output.
- When INPUT is specified, only for files opened as input.
- When I-O is specified, only for files opened as I-O.

If the INPUT, OUTPUT, or I-O option is specified, and an input, output, or input-output file, respectively, is described with a LABEL RECORDS ARE OMITTED clause, the USE procedures do not apply.

The standard system procedures are performed:

1. Before or after the user's beginning or ending input label check procedure is executed.
2. Before the user's beginning or ending output label is created.
3. After the user's beginning or ending output label is created, but before it is written on tape.
4. Before or after the user's beginning or ending input-output label check procedure is executed.

Within the procedures of a USE declarative in which the USE sentence specifies an option other than file-name, references to common label items need not be qualified by a file-name. A common label item is an elementary data item that appears in every label record of the program, but does not appear in any data record of this program. Such items must have identical descriptions and positions within each label record.

Within a Format 1 declarative section there must be no reference to any nondeclarative procedure. Conversely, in the nondeclarative portion there must be no reference to procedure-names that appear in the declaratives section, except that PERFORM statements may refer to a USE procedure, or to procedures associated with it.

The exit from a Format 1 declarative section is inserted by the compiler following the last statement in the section. All logical program paths within the section must lead to the exit point.

There is one exception: a special exit may be specified by the statement GO TO MORE-LABELS. When an exit is made from a Format 1 declarative section by means of this statement, the system will do one of the following:

1. Write the current beginning or ending label and then re-enter the USE section at its beginning for further creating of labels. After creating the last label, the user must exit by executing the last statement of the section.
2. Read an additional beginning or ending label, and then re-enter the USE section at its beginning for further checking of labels. When processing user labels, the section will be re-entered only if there is another user label to check. Hence, there need not be a program path that flows through the last statement in the section. For nonstandard labels, the compiler does not know how many labels

exist. Therefore, the last statement in the section must be executed to terminate nonstandard label processing.

If a GO TO MORE-LABELS statement is not executed for a user label, the declarative section is not re-entered to check or create any immediately succeeding user labels.

When reading nonstandard header labels, it is the user's responsibility to read any tape marks that are used to terminate labels. The GO TO MORE-LABELS exit must be used, and the declarative must recognize that a tape mark rather than data is being read. The final exit from the declarative must not be taken until the file is positioned just before the first data record.

The programmer must set special register LABEL-RETURN to nonzero if the nonstandard header label of an input file is not correct.

After the nonstandard trailer labels are processed, the system determines from the DD statement if another reel is to be read or if the current reel is the end of the file. If the current reel is the last one for the file, the statement executed is the one specified in the AT END phrase of the READ statement that detected the end-of-reel condition. If the current reel is not the last, a volume-switch takes place, the header label is processed, and the first record on the reel is read.

SAMPLE LABEL DECLARATIVE PROGRAM

The following program creates a file with user labels. To create the labels, the program contains a DECLARATIVES section, with USE procedures for creating both header and trailer labels.

The program illustrates the following items:

(1) For the file requiring the creation of user labels, the LABEL RECORDS clause uses the data-name option.

(2) The USE AFTER BEGINNING/ENDING LABEL option is specified to create user labels.

(3) The program creates two user header labels, utilizing the special exit GO TO MORE LABELS to create the second label.

(4) The information to be inserted in the user labels comes from input file records. Therefore, records containing the information must be read and stored before the output file is opened, and the header label procedures are invoked.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. LABELPGM.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-360-F50.
OBJECT-COMPUTER. IBM-360-F50.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT NO-LBL ASSIGN TO UT-2400-S-INFILE.
    SELECT USER ASSIGN TO UT-2400-S-USRFILE.
DATA DIVISION.
FILE SECTION.
FD  NO-LBL
    RECORD CONTAINS 80 CHARACTERS
    LABEL RECORD IS OMITTED.
01  IN-REC.
    05  TYPEN PIC X(4).
    05  DEPT-ID PIC X(11).
    05  BIL-PERIOD PIC X(5).
    05  NAME PIC X(20).
    05  AMOUNT PIC 9(6).
    05  FILLER PIC X(15).
    05  SECUR-CODE PIC XX.
    05  FILLER PIC 9.
    05  ACCT-NUM PIC 9(10).
    05  FILLER PIC 9(6).
01  IN-LBL-HIST.
    05  FILLER PIC X(4).
    05  FILE-HISTORY PIC X(76).
FD  USER
    RECORD CONTAINS 80 CHARACTERS
    BLOCK CONTAINS 5 RECORDS
    LABEL RECORDS ARE USR-LBL USR-LBL-HIST.
01  USR-LBL.
    05  USR-HDR PIC X(4).
    05  DEPT-ID PIC X(11).
    05  USR-REC-CNT PIC 9(8) COMP-3.
    05  BIL-PERIOD PIC X(5).
    05  FILLER PIC X(53).
    05  SECUR-CODE PIC XX.
```

(1)

LABEL Declarative--Sample Program

```
01  USR-LBL-HIST.
    05  FILLER PIC X(4).
    05  LBL-HISTORY PIC X(76).
01  USR-REC.
    05  TYPEN PIC X(4).
    05  FILLER PIC X(5).
    05  NAME PIC X(20).
    05  FILLER PIC X(4).
    05  ACCT-NUM PIC 9(10).
    05  AMOUNT PIC 9(6) COMP-3.
    05  FILLER PIC X(25).
    05  U-SEQ-NUMB PIC 9(8).
WORKING-STORAGE SECTION.
77  U-REC-NUMB PIC 9(8) VALUE ZERO.
77  SAV-DEPT-ID PIC X(11).
77  LBL-SWITCH PIC 9 VALUE ZERO.
77  USER-SWITCH PIC 9 VALUE ZERO.
01  STOR-REC.
    05  DEPT-ID PIC X(11).
    05  BIL-PERIOD PIC X(5).
    05  SECUR-CODE PIC XX.
PROCEDURE DIVISION.

(2) DECLARATIVES.
USR-HDR-LBL SECTION. USE AFTER BEGINNING FILE
    LABEL PROCEDURE ON USER.
A.  IF LBL-SWITCH = 0
        MOVE SPACES TO USR-LBL
        MOVE ZEROES TO USR-REC-CNT
        MOVE 'UHL1' TO USR-HDR
        MOVE CORRESPONDING STOR-REC TO USR-LBL
(3)     ADD 1 TO LBL-SWITCH  GO TO MORE-LABELS
            ELSE MOVE 'UHL2' TO USR-HDR
                 MOVE FILE-HISTORY TO LBL-HISTORY.
USR-TRLR-LBL SECTION. USE AFTER ENDING FILE
    LABEL PROCEDURE ON USER.
B.  MOVE SPACES TO USR-LBL.
    MOVE 'UTL1' TO USR-HDR.
    MOVE SAV-DEPT-ID TO DEPT-ID IN USR-LBL.
    MOVE U-REC-NUMB TO USR-REC-CNT.
END DECLARATIVES.

    OPEN INPUT NO-LBL.
READ-IN.
(4) READ NO-LBL AT END GO TO END-JOB.
A.  IF USER-SWITCH = 1 NEXT SENTENCE
        ELSE ADD 1 TO USER-SWITCH
             MOVE CORRECPONDING IN-REC TO STOR-REC
             MOVE DEPT-ID OF IN-REC TO SAV-DEPT-ID
    PERFORM READ-IN
        OPEN OUTPUT USER
        GO TO READ-IN.
    MOVE SPACES TO USR-REC
    ADD 1 TO U-REC-NUMB
    MOVE CORRESPONDING IN-REC TO USR-REC
    MOVE U-REC-NUMB TO U-SEQ-NUMB
    WRITE USR-REC
    GO TO READ-IN.
END-JOB.
    CLOSE NO-LBL USER
    STOP RUN.
```

A Format 2 USE sentence specifies procedures to be followed if an input/output error occurs during file processing.

```
                              Format 2
-----------------------------------------------------------------------

  USE AFTER STANDARD ERROR PROCEDURE

      ON { file-name-1 [file-name-2] ... }
         { INPUT                         }
         { OUTPUT                        }
         { I-O                           }

        [GIVING data-name-1 [data-name-2]].

-----------------------------------------------------------------------
```

USE declaratives which specify error handling procedures are activated when an input/output error occurs during execution of a READ, WRITE, REWRITE, or START statement.

Automatic system error routines are executed before user-specified procedures.

User error handling procedures are executed for invalid key conditions if the INVALID KEY option is not specified in the statement causing the condition.

Within the error procedure, the allowable statements that may be executed depend on the organization and access specified for the file in error.

Within a USE procedure there must not be any reference to nondeclarative procedures except when an exit is taken with a GO TO statement. Conversely, in the nondeclarative portion, there must be no reference to procedure-names that appear in the declaratives portion, except that PERFORM statements may refer to a USE declarative or to procedures associated with such a declarative.

When the file-name option is used, error handling procedures are executed for input/output errors occurring for the named file(s) only.

A file-name must not be referred to, implicitly or explicitly, in more than one Format 2 USE sentence.

The user error procedures are executed, when the INPUT, OUTPUT, or I-O options are specified and an input/output error occurs, as follows:

- When INPUT is specified, only for files opened as INPUT.
- When OUTPUT is specified, only for files opened as OUTPUT.
- When I-O is specified, only for files opened as I-O.

When the GIVING option is used and an error declarative section is entered, data-name-1 will contain the information shown in Figure 11.

The GIVING option is valid only when the error procedure specifies a single file-name.

Program Product Information (Version 3 and Version 4)

The GIVING option may specify multiple file-names, or any one of the INPUT/ OUTPUT/I-O options.

Byte	Information
0-11	System use
12-13	Number of bytes in error block (in binary) or blank
14-48	Blanks
49	,
50-57	Jobname
58	,
59-66	Stepname
67	,
68-70	Unit address
71	,
72-73	Device-type
74	,
75-82	DDNAME
83	,
84-89	Operation attempted
90	,
91-105	Error description
106	,
107-127	The contents of this field depend on the type of input/output device in use, as follows:
	For unit record, 107-120 Asterisks
	121 ,
	122-127 Access method
	For magnetic tape, 107-113 Block count (in decimal)
	114 ,
	115-119 Access method
	120-127 Blanks
	For mass storage, 107-120 Last actual address, in the form BBCCHHR (in hexadecimal)
	121 ,
	122-127 Access method
128-135	System use

Notes:

1. Bytes 12-13 are in binary representation.

2. Bytes 49-127, unless otherwise indicated, are in EBCDIC representation.

3. Bytes 91-105 (Error Description) contain a brief description of the type of error that occurred. The description is provided by the system error analysis procedures, and is placed into bytes 91-105 upon entry into the Error Declarative.

   For example, if the FD for an input file described 120-character records, and if at execution time the file actually contained 100-character records, then when a read operation was attempted an error would occur, and the system would return the message "WRN.LEN.RECORD".

   Similarly, for a BISAM file opened I-O, an attempt to write a record without a preceding read operation would cause an error, and the system would return the message "INVALID REQUEST".

   Note that each release of the Operating System adds and deletes messages, so that the actual content of the messages is subject to change.

Figure 11. Information Supplied With the GIVING Option When an Error Declarative is Entered

Data-name-1 must be a 136-byte item. It must be defined in the Working-Storage Section. If no data was transmitted, bytes 12-13 of data-name-1 will contain blanks.

If specified, data-name-2 contains the block in error for a READ operation, if data was transmitted. For a WRITE, REWRITE, or START operation, data-name-2 must not be referred to.

Data-name-2 must be an item large enough to hold the largest physical block which exists or which will be processed. It must be defined in the Working-Storage or Linkage Section. If data-name-2 is defined in the Linkage Section, the block in error is referenced in the buffer area; no storage need be defined for the error block.

An exit from this type of declarative section can be effected by executing the last statement in the section (normal return), or by means of a GO TO statement. A summary of the facilities, precautions to be taken when using the GIVING option, and suggested user response associated with each file-processing technique when an error occurs, is given in the Programmer's Guide.

A Format 3 USE sentence specifies Procedure Division statements that are executed just before a report group named in the Report Section of the Data Division is produced (see "Report Writer").

Format 3

```
USE BEFORE REPORTING data-name.
```

ARITHMETIC STATEMENTS

The arithmetic statements are used for computations. Individual operations are specified by the ADD, SUBTRACT, MULTIPLY, and DIVIDE statements. These operations can be combined symbolically in a formula, using the COMPUTE statement.

Because there are several options common to the arithmetic statements, their discussion precedes individual statement descriptions.

CORRESPONDING Option

The CORRESPONDING option enables computations to be performed on elementary items of the same name simply by specifying the group item to which they belong. The word CORRESPONDING may be abbreviated as CORR.

Both identifiers following CORRESPONDING must refer to group items. For the purposes of this discussion, these identifiers will be called d_1 and d_2.

Elementary data items from each group are considered CORRESPONDING when both data items have the same name and qualification, up to but not including d_1 and d_2.

Neither d_1 nor d_2 may be a data item with level number 66, 77, or 88, nor may either be described with the USAGE IS INDEX clause. Neither d1 nor d2 may be a FILLER item.

Each data item subordinate to d_1 or d_2 that is described with a REDEFINES, RENAMES, OCCURS, or USAGE IS INDEX clause is ignored; any items subordinate to such data items are also ignored. However, d_1 or d_2 may themselves be described with REDEFINES or OCCURS clauses, or be subordinate to items described with REDEFINES or OCCURS clauses.

Each FILLER item subordinate to d1 or d2 is ignored. However, d1 or d2 may be subordinate to a FILLER item.

GIVING Option

If the GIVING option is specified, the value of the identifier that follows the word GIVING is set equal to the calculated result of the arithmetic operation. This identifier, since not itself involved in the computation, may be a numeric edited item.

ROUNDED Option

After decimal point alignment, the number of places in the fraction of the result of an arithmetic operation is compared with the number of places provided for the fraction of the resultant identifier.

When the size of the fractional result exceeds the number of places provided for its storage, truncation occurs unless ROUNDED is specified. When ROUNDED is specified, the least significant digit of the resultant identifier has its value increased by 1 whenever the most significant digit of the excess is greater than or equal to 5.

When the resultant identifier is described by a PICTURE clause containing P's and when the number of places in the calculated result exceeds this size, rounding or truncation occurs relative to the rightmost integer position for which storage is allocated.

Note: The preceding ROUNDED description does not apply when the resultant field is floating point, in which case rounding has no meaning. However, if at least one of the operands of an arithmetic operation is floating-point and the resultant field is fixed-point, rounding always takes place, whether or not ROUNDED is specified.

SIZE ERROR Option

If, after decimal point alignment, the 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 and does not apply to intermediate results. If the ROUNDED option 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 option is specified.

If the SIZE ERROR option is not specified and a size error condition occurs, the value of the resultant identifier affected may be unpredictable.

If the SIZE ERROR option is specified and a size error condition occurs, the value of the resultant identifier affected by the size error is not altered. After completion of the execution of the arithmetic operation, the imperative statement in the SIZE ERROR option is executed.

For COMPUTATIONAL-1 and COMPUTATIONAL-2 items, only division by zero causes the imperative statement in the SIZE ERROR option to be executed.

Overlapping Operands

When the sending and receiving operands of an arithmetic statement or a MOVE statement share a part of their storage (that is, when the operands overlap), the result of the execution of such a statement is unpredictable.

ADD Statement

The ADD statement causes two or more numeric operands to be summed and the result to be stored.

Format 1

```
ADD { identifier-1 } [ identifier-2 ]
    { literal-1    } [ literal-2    ]  ... TO identifier-m [ROUNDED]
   [identifier-n [ROUNDED]] ... [ON SIZE ERROR imperative-statement]
```

```
+----------------------------------------------------------------------+
|                              Format 2                                |
+----------------------------------------------------------------------+
|                                                                      |
|     {identifier-1} {identifier-2} [identifier-3]                     |
| ADD {            } {            } [            ] ...                 |
|     {literal-1   } {literal-2   } [literal-3   ]                     |
|                                                                      |
|   GIVING identifier-m [ROUNDED] [ON SIZE ERROR imperative-statement] |
|                                                                      |
+----------------------------------------------------------------------+
```

```
+----------------------------------------------------------------------+
|                              Format 3                                |
+----------------------------------------------------------------------+
|                                                                      |
|        {CORR         }                                               |
|   ADD  {             }   identifier-1 TO identifier-2                |
|        {CORRESPONDING}                                               |
|                                                                      |
|      [ROUNDED] [ON SIZE ERROR imperative-statement]                  |
|                                                                      |
+----------------------------------------------------------------------+
```

FORMAT 1 -- the values of the operands preceding the word TO are added together, and the sum is added to the current value of identifier-m (identifier-n), etc. The result is stored in identifier-m (identifier-n), etc.

FORMAT 2 -- when the GIVING option is used, there must be at least two operands preceding the word GIVING. The values of these operands are added together, and the sum is stored as the new value of identifier-m.

In Formats 1 and 2 each identifier must refer to an elementary numeric item, with the exception of identifiers appearing to the right of the word GIVING. These may refer to numeric edited data items.

Each literal must be a numeric literal.

The maximum size of each operand is 18 decimal digits. The maximum size of the resulting sum, after decimal point alignment, is 18 decimal digits.

FORMAT 3 -- when the CORRESPONDING option is used, elementary data items within identifier-1 are added to and stored in corresponding elementary data items within identifier-2. Identifier-1 and identifier-2 must be group items.

When ON SIZE ERROR is used in conjunction with CORRESPONDING, the size error test is made only after the completion of all the ADD operations. If any of the additions produces a size error condition, the resultant field for that addition remains unchanged, and the imperative statement specified in the SIZE ERROR option is executed.

COMPUTE Statement

The COMPUTE statement assigns to a data item the value of a data item, literal, or arithmetic expression.

Format

```
COMPUTE identifier-1 [ROUNDED] = {identifier-2
                                  {literal-1
                                  {arithmetic-expression}

     [ON SIZE ERROR imperative-statement]
```

Literal-1 must be a numeric literal.

Identifier-2 must refer to an elementary numeric item. Identifier-1 may describe a numeric edited data item.

The identifier-2 and literal-1 options provide a method for setting the value of identifier-1 equal to the value of identifier-2 or literal-1.

The arithmetic-expression option permits the use of a meaningful combination of identifiers, numeric literals, and arithmetic operators. Hence, the user can combine arithmetic operations without the restrictions imposed by the arithmetic statements ADD, SUBTRACT, MULTIPLY, and DIVIDE.

As in all arithmetic statements, the maximum size of each operand is 18 decimal digits.

DIVIDE Statement

The DIVIDE statement is used to find the quotient resulting from the division of one data item into another data item.

Format 1

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

     [ON SIZE ERROR imperative-statement]
```

Format 2

```
DIVIDE  {identifier-1}  {INTO}  {identifier-2}  GIVING identifier-3
        {literal-1   }  {BY  }  {literal-2   }

        [ROUNDED] [REMAINDER identifier-4]

        [ON SIZE ERROR imperative-statement]
```

When Format 1 is used, the value of identifier-1 (or literal-1) is divided into the value of identifier-2. The value of the dividend (identifier-2) is replaced by the value of the quotient.

When Format 2 is used, the value of identifier-1 (or literal-1) is divided into or by identifier-2 (or literal-2), the quotient is stored in identifier-3, and the remainder optionally is stored in identifier-4.

A remainder is defined as the result of subtracting the product of the quotient and the divisor from the dividend. When the REMAINDER option is specified, none of the identifiers may refer to floating-point items. If the ROUNDED option is also specified, the quotient is rounded after the remainder is determined.

Each identifier must refer to an elementary numeric item except the identifier following the word GIVING which may be a numeric edited item.

Each literal must be a numeric literal.

The maximum size of each operand is 18 decimal digits. The maximum size of the resulting quotient, after decimal point alignment, is 18 decimal digits. The maximum size of the resulting remainder (if specified), after decimal point alignment is 18 decimal digits.

Division by zero always results in a size error condition.

MULTIPLY Statement

The MULTIPLY statement is used to multiply one data item by another data item.

Format 1

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

        [ON SIZE ERROR imperative-statement]
```

```
Format 2

MULTIPLY {identifier-1}  BY  {identifier-2}  GIVING identifier-3
         {literal-1   }      {literal-2   }

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

When Format 1 is used, the value of identifier-1 (or literal-1) is multiplied by the value of identifier-2. The value of the multiplier (identifier-2) is replaced by the product.

When Format 2 is used, the value of identifier-1 (or literal-1) is multiplied by identifier-2 (or literal-2), and the product is stored in identifier-3.

Each identifier must refer to an elementary numeric item except the identifier following the word GIVING which may be a numeric edited item.

Each literal must be a numeric literal.

The maximum size of each operand is 18 decimal digits. The maximum size of the resulting product, after decimal point alignment, is 18 decimal digits.

SUBTRACT Statement

The SUBTRACT statement is used to subtract one, or the sum of two or more, numeric data items from another data item(s).

```
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 }  GIVING identifier-n
          { literal-m    }
    [ROUNDED]   [ON SIZE ERROR imperative-statement]
```

Format 3

```
SUBTRACT { CORR          }  identifier-1 FROM identifier-2
         { CORRESPONDING }
    [ROUNDED] [ON SIZE ERROR imperative-statement]
```

Format 1 -- all literals or identifiers preceding the word FROM are added together, and this total is subtracted from identifier-m, and identifier-n (if stated), etc. The result of the subtraction is stored as the new value of identifier-m, identifier-n, etc.

Format 2 -- all literals or identifiers preceding the word FROM are added together, and this total is subtracted from literal-m or identifier-m. The result of the subtraction is stored as the new value of identifier-n.

Format 3 -- data items in identifier-1 are subtracted from, and the difference stored into corresponding data items in, identifier-2. When the CORRESPONDING option is used in conjunction with ON SIZE ERROR and an ON SIZE ERROR condition arises, the result for SUBTRACT is analogous to that for ADD.

Each identifier must refer to an elementary numeric item except the identifier following the word GIVING which may be a numeric edited item.

Each literal must be a numeric literal.

The maximum size of each operand is 18 decimal digits. The maximum size of the resulting difference, after decimal point alignment, is 18 decimal digits.

Statements, sentences, and paragraphs in the Procedure Division are ordinarily executed sequentially. The procedure branching statements allow alterations in the sequence. These statements are ALTER, GO TO, PERFORM, STOP, and EXIT.

GO TO Statement

The GO TO statement allows a transfer from one part of the program to another.

```
Format 1

GO TO procedure-name-1
```

```
Format 2

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

```
Format 3

GO TO.
```

When Format 1 is specified, control is passed to procedure-name-1 or to another procedure name if the GO TO statement has been changed by an ALTER statement. (If the latter is the case, the GO TO statement must have a paragraph name, and the GO TO statement must be the only statement in the paragraph.)

If a GO TO statement represented by Format 1 appears in an imperative sentence, it must appear as the only or last statement in a sequence of imperative statements.

When Format 2 is used, control is transferred to one of a series of procedures, depending on the value of the identifier. For example, when identifier has a value of 1, control is passed to procedure-name-1; a value of 2 causes control to be passed to procedure-name-2, ...; a value of n causes control to be passed to procedure-name-n. For the GO TO statement to have effect, identifier must represent a positive or unsigned integer, i.e., 1, 2, ..., n. If the value of the identifier is anything other than a value within the range 1 through n, the GO TO statement is ignored. The number of procedure-names must not exceed 2031.

Identifier is the name of a numeric elementary item described as an integer. Its PICTURE must be of four digits or less. Its USAGE must be DISPLAY, COMPUTATIONAL, or COMPUTATIONAL-3.

When Format 3 is used, an ALTER statement, referring to the GO TO statement, must have been executed prior to the execution of the GO TO statement. The GO TO statement must immediately follow a paragraph name and must be the only statement in the paragraph.

ALTER Statement

The ALTER statement is used to change the transfer point specified in a GO TO statement.

Format
ALTER procedure-name-1 TO [PROCEED TO] procedure-name-2 [procedure-name-3 TO [PROCEED TO] procedure-name-4]...

Procedure-name-1, procedure-name-3, etc., must be the names of paragraphs that contain only one sentence consisting of a GO TO statement without the DEPENDING option.

Procedure-name-2, procedure-name-4, etc., must be the names of paragraphs or sections in the Procedure Division.

The effect of the ALTER statement is to replace the procedure-name operands of the GO TO statements with procedure-name-2, procedure-name-4, etc., of the ALTER statement, where the paragraph-name containing the GO TO statement is procedure-name-1, procedure-name-3, etc. For example:

```
PARAGRAPH-1.
    GO TO BYPASS-PARAGRAPH.
PARAGRAPH-1A.
          .
          .
          .
BYPASS-PARAGRAPH.
          .
          .
          .
    ALTER PARAGRAPH-1 TO PROCEED TO PARAGRAPH-2.
          .
          .
          .
PARAGRAPH-2.
          .
          .
          .
```

Before the ALTER statement is executed, when control reaches PARAGRAPH-1, the GO TO statement transfers control to BYPASS-PARAGRAPH. After execution of the ALTER statement, however, when control reaches PARAGRAPH-1, the GO TO statement transfers control to PARAGRAPH-2.

Segmentation Information: A GO TO statement in a section whose priority is greater than or equal to 50 must not be referred to by an ALTER statement in a section with a different priority. All other uses of the ALTER statement are valid and are performed even if the GO TO to which the ALTER refers is in an overlayable fixed segment (see "Segmentation").

PERFORM Statement

The PERFORM statement is used to depart from the normal sequence of procedures in order to execute a statement, or a series of statements, a specified number of times; or until a predetermined condition is satisfied. After the statements are executed, control is returned to the statement after the PERFORM statement.

Format 1

```
PERFORM procedure-name-1 [THRU procedure-name-2]
```

Format 2

```
PERFORM procedure-name-1 [THRU procedure-name-2]
    {identifier-1}
    {integer-1   }  TIMES
```

Format 3

```
PERFORM procedure-name-1 [THRU procedure-name-2]
    UNTIL condition-1
```

Format 4

```
PERFORM procedure-name-1 [THRU procedure-name-2]

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

        BY      {literal-3    } UNTIL condition-1
                {identifier-3 }

        [AFTER  {index-name-4 } FROM {index-name-5 }
                {identifier-4 }      {literal-5    }
                                     {identifier-5 }

        BY      {literal-6    } UNTIL condition-2
                {identifier-6 }

        [AFTER  {index-name-7 } FROM {index-name-8 }
                {identifier-7 }      {literal-8    }
                                     {identifier-8 }

        BY      {literal-9    } UNTIL condition-3]]
                {identifier-9 }
```

Each procedure-name must be the name of a section or paragraph in the Procedure Division.

Each identifier represents a numeric elementary item described in the Data Division. In Format 2, and Format 4 with the AFTER option, each identifier represents a numeric item described as an integer. However, when Format 4 with the AFTER option is used, this compiler allows each identifier to be described as a non-integral numeric item.

Each literal represents a numeric literal.

Whenever a PERFORM statement is executed, control is transferred to the first statement of the procedure named procedure-name-1. Control is always returned to the statement following the PERFORM statement. The point from which this control is passed is determined as follows:

1. If procedure-name-1 is a paragraph-name and procedure-name-2 is not specified, the return is made after the execution of the last statement of procedure-name-1.

2. If procedure-name-1 is a section name and procedure-name-2 is not specified, the return is made after the execution of the last sentence of the last paragraph in procedure-name-1.

3. If procedure-name-2 is specified and it is a paragraph name, the return is made after the execution of the last statement of that paragraph.

4. If procedure-name-2 is specified and it is a section name, the return is made after the execution of the last sentence of the last paragraph in the section.

When both procedure-name-1 and procedure-name-2 are specified, GO TO and PERFORM statements may appear within the sequence of statements within these paragraphs or sections. When procedure-name-1 alone is specified, PERFORM statements may appear within the procedure. GO TO

may also appear but may not refer to a procedure-name outside the range of procedure-name-1.

When a PERFORM statement includes within its range of procedures another PERFORM statement, this embedded PERFORM statement must have its range of procedures either totally included in or totally excluded from the range of procedures of the original PERFORM statement. That is, the exit point of the original PERFORM statement cannot be contained within the range of procedures of the embedded PERFORM statement, except as a common exit point. Embedded PERFORM or GO TO statements may have their exit point at the same point that the original PERFORM makes it exit. This common exit point must be the name of a paragraph consisting solely of an EXIT statement.

Control may be passed to a sequence of statements that lies between the entry and exit points of a PERFORM statement by means other than a PERFORM. In this case, control passes through the last statement of the procedure to the following statement as if no PERFORM statement referred to these procedures.

FORMAT 1: When Format 1 is used, the procedure(s) referred to are executed once, and control returns to the statement following the PERFORM statement.

FORMAT 2: When Format 2 is used, the procedure(s) are performed the number of times specified by identifier-1 or integer-1. Once the TIMES option is satisfied, control is transferred to the statement following the PERFORM statement.

The following rules apply to the use of a Format 2 PERFORM statement:

1. If integer-1 or identifier-1 is zero or a negative number at the time the PERFORM statement is initiated, control passes to the statement following the PERFORM statement.

2. Once the PERFORM statement has been initiated, any reference to identifier-1 has no effect in varying the number of times the procedures are initiated.

FORMAT 3: When Format 3 is used, the specified procedures are performed until the condition specified by the UNTIL option is true. At this time, control is transferred to the statement following the PERFORM statement. If the condition is true at the time that the PERFORM statement is encountered, the specified procedure(s) are not executed.

FORMAT 4: Format 4 is used to augment the value of one or more identifiers or index-names during the execution of a PERFORM statement.

When executing a Format 4 PERFORM statement, the initial values of identifier-2 (index-name-2) and identifier-5 (index-name-5) must be positive in order to conform with the standard. However, this compiler allows these initial values to be negative.

In the following discussion of Format 4, every reference to identifier-n also refers to index-name-n except when identifier-n is the object of the BY option. Also, when index-names are used, the FROM and BY clauses have the same effect as in a SET statement (see "Table Handling").

During execution of the PERFORM statement, reference to index-names or identifiers of the FROM option has no effect in altering the number of times the procedures are to be executed. Changing the value of index-names or identifiers of the VARYING option or identifiers of the BY option, however, will change the number of times the procedures are executed.

When one identifier is varied, the following is the sequence of events:

1. Identifier-1 is set equal to its starting value, identifier-2 or literal-2.

2. If condition-1 is false, the specified procedure(s) are executed once.

3. The value of identifier-1 is augmented by the specified increment or decrement, identifier-3 or literal-3, and condition-1 is evaluated again.

4. Steps 2 and 3 are repeated, if necessary, until the condition is true. When the condition is true, control passes directly to the statement following the PERFORM statement. If the condition is true for the starting value of identifier-1, the procedure(s) are not executed, and control passes directly to the statement following the PERFORM statement.

Chart 2 is a flowchart illustrating the logic of the PERFORM statement when one identifier is varied.

When two identifiers are varied, the following is the sequence of events:

1. Identifier-1 and identifier-4 are set to their initial values, identifier-2 (or literal-2) and identifier-5 (or literal-5), respectively.

2. Condition-1 is evaluated; if true, control is passed to the statement following the PERFORM statement; if false, condition-2 is evaluated.

3. If condition-2 is false, procedure-name-1 through procedure-name-2 (if specified) is executed once.

4. Identifier-4 is augmented by identifier-6 (or literal-6), and condition-2 is evaluated again.

5. If condition-2 is false, steps 3 and 4 are repeated.

6. If condition-2 is true, identifier-4 is set to its initial value, identifier-5.

7. Identifier-1 is augmented by identifier-3 (or literal-3).

8. Steps 2 through 7 are repeated until condition-1 is true.

At the termination of the PERFORM statement, if condition-1 was true when the PERFORM statement was encountered, identifier-1 and identifier-4 contain their initial values. Otherwise, identifier-1 has a value that differs from its last used setting by an increment or decrement, as the case may be.

Chart 3 is a flowchart illustrating the logic of the PERFORM statement when two identifiers are varied.

For three identifiers, the mechanism is the same as for two identifiers except that identifier-7 goes through the complete cycle each time that identifier-4 is augmented by identifier-6 or literal-6, which in turn goes through a complete cycle each time identifier-1 is varied.

Chart 4 is a flowchart illustrating the logic of the PERFORM statement when three identifiers are varied.

SEGMENTATION INFORMATION: A PERFORM statement appearing in a section whose priority is less than the segment limit can have within its range only one of the following:

1. Sections each of which has a priority number less than 50.
2. Sections wholly contained in a single segment whose priority number is greater than 49.

However, this compiler allows the PERFORM to have within its range sections with any priority numbers.

A PERFORM statement appearing in a section whose priority number is equal to or greater than the segment limit can have within its range only one of the following:

1. Sections with the same priority number as the section containing the PERFORM statement.
2. Sections with a priority number less than the segment limit.

However, this compiler allows the PERFORM to have within its range sections with any priority numbers.

When a procedure-name in an independent segment is referred to by a PERFORM statement contained in a segment with a different priority number, the segment referred to is made available in its initial state for each execution of the PERFORM statement. When a procedure-name in the fixed portion is referred to by a PERFORM statement in an independent segment, the independent segment is reinitialized upon exit from the PERFORM statement. (See "Segmentation.")

Chart 2. Logical Flow of Option 4 PERFORM Statement Varying One Identifier

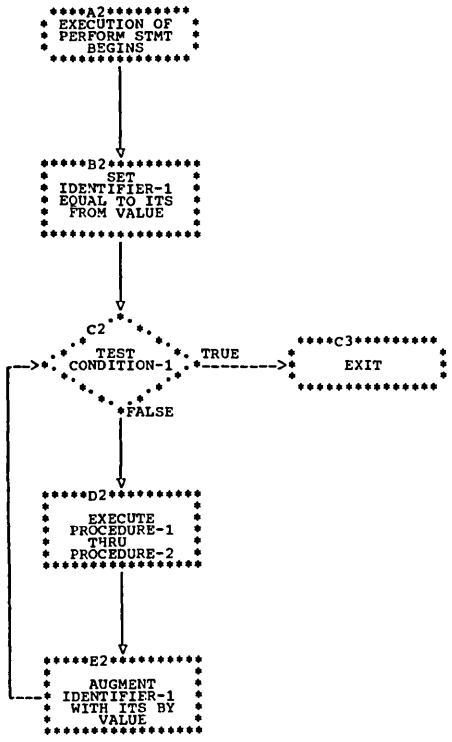


Chart 3. Logical Flow of Option 4 PERFORM Statement Varying Two Identifiers

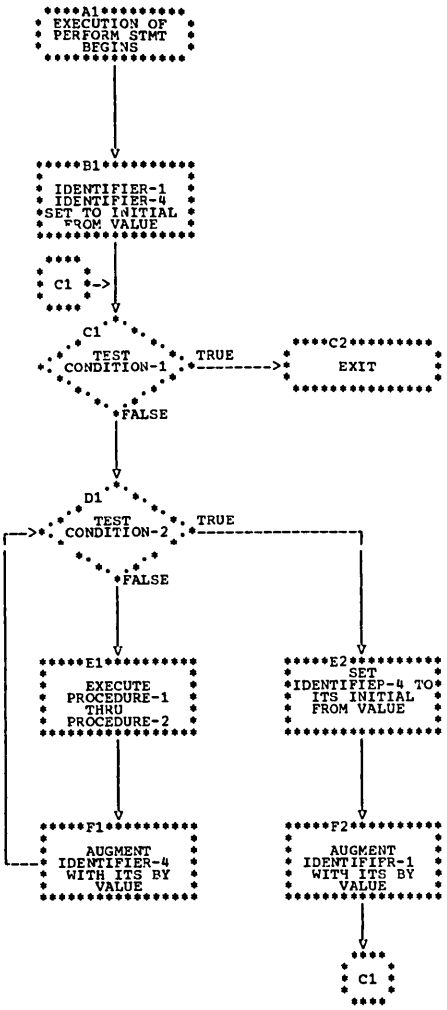
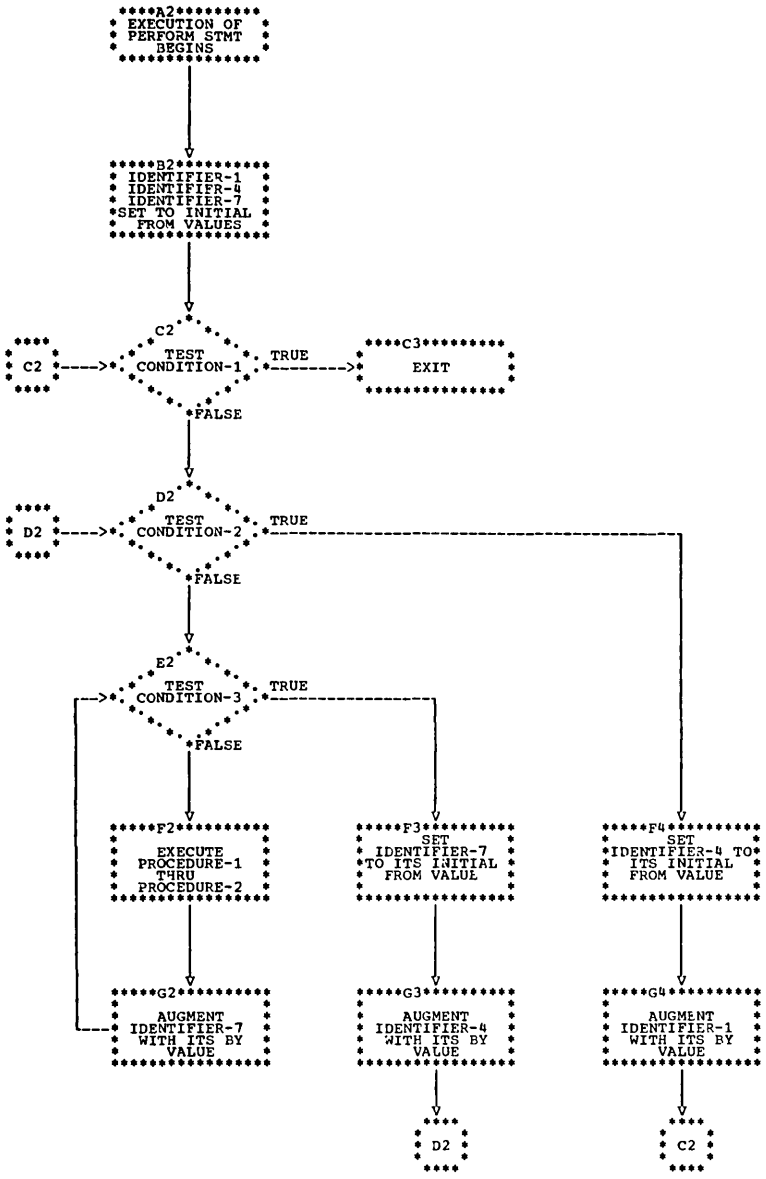


Chart 4. Logical Flow of Option 4 PERFORM Statement Varying Three Identifiers



STOP Statement

The STOP statement halts the object program either permanently or temporarily.

Format

```
STOP { RUN     }
     { literal }
```

When the RUN option is used, the execution of the object program is terminated, and control is returned to the system.

If a STOP statement with the RUN option appears in an imperative statement, it must appear as the only or last statement in a sequence of imperative statements. All files should be closed before a STOP RUN statement is issued.

If it is desired to pass a return code to the operating system or the invoking program, the special register RETURN-CODE must be set prior to [illegible] S9999. The normal return code for successful completion is zero; other values returned are conventionally in multiples of four. However, the maximum value the field can contain is 4095.

For the effect when STOP RUN is used in either a calling program or a called program, see "Subprogram Linkage."

When the literal option is used, the literal is communicated to the operator. The program may be resumed only by operator intervention. Continuation of the object program begins with the execution of the next statement in sequence.

The literal may be numeric or nonnumeric, or it may be any figurative constant except ALL.

EXIT Statement

The EXIT statement provides a common end point for a series of procedures.

Format

```
paragraph-name.  EXIT [PROGRAM].
```

It is sometimes necessary to transfer control to the end point of a series of procedures. This is normally done by transferring control to the next paragraph or section, but in some cases this does not have the required effect. For instance, the point to which control is to be transferred may be at the end of a range of procedures governed by a PERFORM or at the end of a declarative section. The EXIT statement is provided to enable a procedure-name to be associated with such a point.

If control reaches an EXIT paragraph and no associated PERFORM or USE statement is active, control passes through the EXIT point to the first sentence of the next paragraph.

The EXIT statement must be preceded by a paragraph-name and be the only statement in the paragraph.

The EXIT statement with the PROGRAM option is discussed in "Subprogram Linkage."

Movement and inspection of data are implicit in the functioning of several of the COBOL statements. These statements are: MOVE, EXAMINE, and TRANSFORM.

MOVE Statement

The MOVE statement is used to transfer data from one area of storage to one or more other areas.

Format 1

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

Format 2

```
MOVE  {CORRESPONDING}  identifier-1 TO identifier-2
      {CORR         }
```

An index data item cannot appear as an operand of a MOVE statement.

FORMAT 1: identifier-1 and literal represent the sending area; identifier-2, identifier-3, ... represent the receiving areas.

The data designated by literal or identifier-1 is moved first to identifier-2, then to identifier-3 (if specified), etc.

FORMAT 2: the CORRESPONDING option is used to transfer data between items of the same name simply by specifying the group items to which they belong.

Neither identifier may be a level-66, level-77, or level-88 data item.

Data items from each group are considered CORRESPONDING when they have the same name and qualification, up to but not including identifier-1 and identifier-2.

At least one of the data items of a pair of matching items must be an elementary data item.

Each subordinate item containing an OCCURS, REDEFINES, USAGE IS INDEX, or RENAMES clause is ignored. However, either identifier may

have a REDEFINES or OCCURS clause in its description or may be subordinate to a data item described with these clauses.

General Rules Applying to Any MOVE Statement:

1. Any move in which the sending and receiving items are both elementary items is an elementary move. Each elementary item belongs to one of the following categories: numeric, alphabetic, alphanumeric, numeric edited, or alphanumeric edited (see "PICTURE Clause" in "Data Division"). Numeric literals belong to the category numeric; nonnumeric literals belong to the category alphanumeric.

2. When an alpnanumeric edited, alphanumeric, or alphabetic item is a receiving item:

   a. Justification and any necessary filling of unused character positions takes place as defined under the JUSTIFIED clause. Unused character positions are filled with spaces.

   b. If the size of the sending item is greater than the size of the receiving item, the excess characters are truncated after the receiving item is filled.

   c. If the sending item has an operational sign, the absolute value is used.

3. When a numeric or numeric edited item is a receiving item:

   a. Alignment by decimal point and any necessary zero filling of unused character positions takes place, except when zeros are replaced because of editing requirements.

   b. The absolute value of the sending item is used if the receiving item has no operational sign.

   c. If the sending item has more digits to the left or right of the decimal point than the receiving item can contain, excess digits are truncated.

   d. The results at object time may be unpredictable if the sending item contains any nonnumeric characters.

4. Any necessary conversion of data from one form of internal representation to another takes place during the move, along with any specified editing in the receiving item.

5. Any move that is not an elementary move is treated exactly as though it were an alphanumeric elementary move, except that there is no conversion of data from one form of internal representation to another. In such a move, the receiving area is filled without consideration for the individual elementary or group items contained within either the sending or the receiving area.

6. When the sending and receiving operands of a MOVE statement share a part of their storage (that is, when the operands overlap), the result of the execution of such a statement is unpredictable.

There are certain restrictions on elementary moves. These are shown in Table 15.

Table 15. Permissible Moves

Source Field \ Receiving Field	GR	AL	AN	ED	BI	NE	ANE	ID	EF	IF	SN	SR
Group (GR)	Y	Y	Y	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]
Alphabetic (AL)	Y	Y	Y	N	N	N	Y	N	N	N	N	N
Alphanumeric (AN)	Y	Y	Y	Y[4]	Y[4]	Y[4]	Y	Y[4]	Y[4]	Y[4]	Y[4]	Y[4]
External Decimal (ED)	Y[1]	N	Y[2]	Y	Y	Y	Y[2]	Y	Y	Y	Y	Y
Binary (BI)	Y[1]	N	Y[2]	Y	Y	Y	Y[2]	Y	Y	Y	Y	Y
Numeric Edited (NE)	Y	N	Y	N	N	N	Y	N	N	N	N	N
Alphanumeric Edited (ANE)	Y	Y	Y	N	N	N	Y	N	N	N	N	N
ZEROS (numeric or alphanumeric)	Y	N	Y	Y[3]	Y[3]	Y[3]	Y	Y[3]	Y[3]	Y[3]	Y[3]	Y[3]
SPACES (AL)	Y	Y	Y	N	N	N	Y	N	N	N	N	N
HIGH-VALUE, LOW-VALUE, QUOTES	Y	N	Y	N	N	N	Y	N	N	N	N	N
ALL literal	Y	Y	Y	Y[5]	Y[5]	Y[5]	Y	Y[5]	N	N	N	N
Numeric Literal	Y[1]	N	Y[2]	Y	Y	Y	Y[2]	Y	Y	Y	Y	Y
Nonnumeric Literal	Y	Y	Y	Y[5]	Y[5]	Y[5]	Y	Y[5]	N	N	N	N
Internal Decimal (ID)	Y[1]	N	Y[2]	Y	Y	Y	Y[2]	Y	Y	Y	Y	Y
External Floating-point (EF)	Y[1]	N	N	Y	Y	Y	N	Y	Y	Y	Y	Y
Internal Floating-point (IF)	Y[1]	N	N	Y	Y	Y	N	Y	Y	Y	Y	Y
Sterling Nonreport (SN)	Y[1]	N	Y	Y	Y	Y	N	Y	Y	Y	Y	Y
Sterling Report (SR)	Y	N	Y	N	N	N	Y	N	N	N	N	N
Floating-point Literal	Y[1]	N	N	Y	Y	Y	N	Y	Y	Y	Y	Y

[1]Move without conversion (like AN to AN).
[2]Only if the decimal point is at the right of the least significant digit.
[3]Numeric move.
[4]The alphanumeric field is treated as an ED (integer) field.
[5]The literal must consist only of numeric characters and is treated as an ED integer field.

EXAMINE Statement

The EXAMINE statement is used to count the number of times a specified character appears in a data item and/or to replace a character with another character.

Format 1

```
EXAMINE identifier TALLYING  { UNTIL FIRST }
                             { ALL         }  literal-1
                             { LEADING     }

          [REPLACING BY literal-2]
```

Format 2

```
EXAMINE identifier REPLACING { ALL         }
                             { LEADING     }
                             { FIRST       }  literal-1
                             { UNTIL FIRST }

          BY literal-2
```

In all cases, the description of identifier must be such that its usage is display (explicitly or implicitly).

When identifier represents a nonnumeric data item, examination starts at the leftmost character and proceeds to the right. Each character in the data item is examined in turn. For purposes of the EXAMINE statement, external floating-point items are treated as nonnumeric data items.

When identifier represents a numeric data item, this data item must consist of numeric characters, and may possess an operational sign. Examination starts at the leftmost character and proceeds to the right. Each character is examined in turn.

If the letter 'S' is used in the PICTURE of the data item description to indicate the presence of an operational sign, the sign is ignored by the EXAMINE statement.

Each literal must consist of a single character belonging to a class consistent with that of the identifier; in addition, each literal may be any figurative constant except ALL. If identifier is numeric, each literal must be an unsigned integer or the figurative constant ZERO (ZEROES, ZEROS).

When Format 1 is used, an integral count is created which replaces the value of a special register called TALLY, whose implicit description is that of an unsigned integer of five digits (see "Language Considerations").

1. When the ALL option is used, this count represents the number of occurrences of literal-1.

2. When the LEADING option is used, this count represents the number of occurrences of literal-1 prior to encountering a character other than literal-1.

3. When the UNTIL FIRST option is used, this count represents all characters encountered before the first occurrence of literal-1.

Whether Format 2 is used, or the REPLACING option of Format 1, the replacement rules are the same. They are as follows:

1. When the ALL option is used, literal-2 is substituted for each occurrence of literal-1.

2. When the LEADING option is used, the substitution of literal-2 for each occurrence of literal-1 terminates as soon as a character other than literal-1 or the right-hand boundary of the data item is encountered.

3. When the UNTIL FIRST option is used, the substitution of literal-2 terminates as soon as literal-1 or the right-hand boundary of the data item is encountered.

4. When the FIRST option is used, the first occurrence of literal-1 is replaced by literal-2.

Specific EXAMINE statements showing the effect of each [illegible] the associated data item and the TALLY are shown in Table 16.

Table 16. Examples of Data Examination

EXAMINE Statement	ITEM-1 (Before)	Data (After)	Resulting Value of TALLY
EXAMINE ITEM-1 TALLYING ALL 0	101010	101010	3
EXAMINE ITEM-1 TALLYING ALL 1 REPLACING BY 0	101010	000000	3
EXAMINE ITEM-1 REPLACING LEADING "*" BY SPACE	**7000	7000	+
EXAMINE ITEM-1 REPLACING FIRST "*" by "$"	**1.94	$*1.94	+

+ unchanged

TRANSFORM Statement

The TRANSFORM statement is used to alter characters according to a transformation rule.

Format

```
TRANSFORM identifier-3 CHARACTERS FROM {figurative-constant-1}
                                       {nonnumeric-literal-1}
                                       {identifier-1        }

          TO {figurative-constant-2}
             {nonnumeric-literal-2}
             {identifier-2        }
```

Identifier-3 must represent an elementary alphabetic, alphanumeric, or numeric edited item, or a group item.

The combination of the FROM and TO options determines what the transformation rule is.

The following rules pertain to the operands of the FROM and TO options:

1. Nonnumeric literals require enclosing quotation marks.
2. Identifier-1 and identifier-2 must be elementary alphabetic, or alphanumeric items, or fixed length group items not over 255 characters in length.
3. A character may not be repeated in nonnumeric-literal-1 or in the area defined by identifier-1. If a character is repeated, the results will be unpredictable.
4. The allowable figurative constants are: ZERO, ZEROES, ZEROS, SPACE, SPACES, QUOTE, QUOTES, HIGH-VALUE, HIGH-VALUES, LOW-VALUE, and LOW-VALUES.

When either identifier-1 or identifier-2 appears as an operand of the specific transformation, the user can change the transformation rule at object time.

Examples of data transformation are given in Table 17; combinations of the FROM and TO options are shown in Table 18.

If any of the operands of a TRANSFORM statement share a part of their storage (that is, if the operands overlap), the result of the execution of such a statement is unpredictable.

Table 17. Examples of Data Transformation

Identifier-3 (Before)	FROM	TO	Identifier-3 (After)
1b7bbABC	SPACE	QUOTE	1"7""ABC
1b7bbABC	"17CB"	"QRST"	QbRbbATS
1b7bbABC	b17ABC	CBA71b	BCACC71b
1234WXY89	98YXW4321	ABCDEFGHI	IHGFEDCBA

Table 18. Combinations of FROM and TO Options (Part 1 of 2)

Operands	Transformation Rule
FROM figurative-constant-1 TO figurative-constant-2	All characters in the data item represented by identifier-3 equal to the single character figurative-constant-1 are replaced by the single character figurative-constant-2.
FROM figurative-constant-1 TO nonnumeric-literal-2	All characters in the data item represented by identifier-3 equal to the single character figurative-constant-1 are replaced by the single character nonnumeric-literal-2.
FROM figurative-constant-1 TO identifier-2	All characters in the data item represented by identifier-3 equal to the single character figurative-constant-1 are replaced by the single character represented by identifier-2.
FROM nonnumeric-literal-1 TO figurative-constant-2	All characters in the data item represented by identifier-3 that are equal to any character in nonnumeric-literal-1 are replaced by the single character figurative-constant-2.
FROM nonnumeric-literal-1 TO	Nonnumeric-literal-1 and nonnumeric-literal-2 must be equal in length or nonnumeric-literal-2 must be a single If the nonnumeric-literals are equal in length, any character in the data item represented by identifier-3 equal to a character in nonnumeric-literal-1 is replaced by the character in the corresponding position of nonnumeric-literal-2. If the length of nonnumeric-literal-2 is one, all characters in the data item represented by identifier-3 that are equal to any character appearing in nonnumeric-literal-1 are replaced by the single character given in nonnumeric-literal-2.

Table 18. Combinations of FROM and TO Options (Part 2 of 2)

Operands	Transformation Rule
FROM nonnumeric-literal-1 TO identifier-2	Nonnumeric-literal-1 and the data item represented by identifier-2 must be equal in length or identifier-2 must represent a single character item. If nonnumeric-literal-1 and identifier-2 are equal in length, any character represented by identifier-3 equal to a character in nonnumeric-literal-1 is replaced by the character in the corresponding position of the item represented by identifier-2. If the length of the data item represented by identifier-2 is one, all characters represented by identifier-3 that are equal to any character appearing in nonnumeric-literal-1 are replaced by the single character represented by identifier-2.
FROM identifier-1 TO figurative-constant-2	All characters represented by identifier-3 that are equal to any character in the data item represented by identifier-1 are replaced by the single character figurative-constant-2.
FROM identifier-1 TO nonnumeric-literal-2	The data item represented by identifier-1 and nonnumeric-literal-2 must be of equal length or nonnumeric-literal-2 must be one character. If identifier-1 and nonnumeric-literal-2 are equal in length, any character in identifier-3 equal to a character in identifier-1 is replaced by the character in the corresponding position of nonnumeric-literal-2. If the length of nonnumeric-literal-3 is one, all characters represented by identifier-3 that are equal to any character represented by identifier-1 are replaced by the single character given in nonnumeric-literal-2.
FROM identifier-1 TO identifier-2	Any character in the data item represented by identifier-3 equal to a character in the data item represented by identifier-1 is replaced by the character in the corresponding position of the data item represented by identifier-2. Identifier-1 and identifier-2 can be one or more characters, but must be equal in length.

The flow of data through the computer is governed by the Operating System. The COBOL statements discussed in this section are used to initiate the flow of data to and from files stored on external media and to govern low-volume information that is to be obtained from or sent to input/output devices such as a card reader or console typewriter.

The Operating System is a record processing system. That is, the unit of data made available by a READ or passed along by a WRITE is the record. The COBOL user need be concerned only with the use of individual records; provision is automatically made for such operations as the movement of data into buffers and/or internal storage, validity checking, error correction (where feasible), unblocking and blocking, and volume switching procedures.

Discussions in this section use the terms volume and reel. The term volume applies to all input and output devices. The term reel applies only to tape devices. Treatment of mass storage devices in the sequential access mode is logically equivalent to the treatment of tape files.

Note: The WRITE statement with the BEFORE/AFTER ADVANCING option is referred to in some of the discussions which follow as the WRITE BEFORE/AFTER ADVANCING statement. Similarly, the WRITE statement with the AFTER POSITIONING option is referred to in some discussions as the WRITE AFTER POSITIONING statement.

OPEN Statement

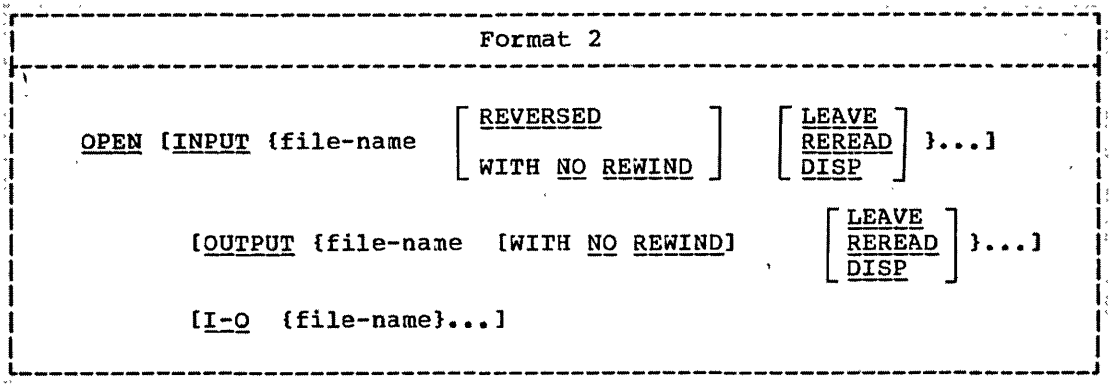
The OPEN statement initiates the processing of input, output, and input-output files. It performs checking and/or writing of labels and other input/output operations.

Format 1

```
OPEN [INPUT {file-name [REVERSED                ] }...]
                       [WITH NO REWIND          ]

     [OUTPUT {file-name [WITH NO REWIND]}...]

     [I-O {file-name}...]
```



The file-name must be defined by a file description entry in the Data Division.

At least one of the options INPUT, OUTPUT, or I-O must be specified. However, there must be no more than one instance of each option in the same statement, although more than one file-name may be used with each option. These options may appear in any order.

The I-O option pertains only to mass storage files.

The OPEN statement must not specify a sort-file, but an OPEN statement must be specified for all other files. The OPEN statement for a file must be executed prior to the first READ, START, REWRITE, or WRITE statement for that file. A file can be opened more than once. However, a second OPEN statement for a file cannot be executed prior to the execution of a CLOSE statement for that file. The OPEN statement does not obtain or release the first data record. A READ or WRITE statement must be executed to obtain or release, respectively, the first data record.

The OPEN statement causes the user's beginning label subroutine to be executed if one is specified by a USE sentence in the Declaratives Section.

The REVERSED and the NO REWIND options can be used only with a sequential single reel file. This compiler allows REVERSED to be used with a sequential multiple reel file. The REVERSED option cannot be used for a file containing mode V records. If the option is specified for a file containing mode U records, doubleword boundary alignment of the logical record is obtained only if the length of the logical record is divisible by eight. If there is no doubleword boundary alignment for a record containing SYNCHRONIZED items, the record cannot be properly processed.

Files with nonstandard labels should not be opened for reversed reading unless the last label is followed by a tape mark. Otherwise, the system reads labels as though they were data records. When the REVERSED option is specified, subsequent READ statements for the file make the data records of the file available in reversed order; that is, starting with the last record.

When the REVERSED option is specified, execution of the OPEN statement causes the file to be positioned at the end of the file.

When opening a file, the NO REWIND option has no effect on file positioning. It appears in the format for language consistency. When either NO REWIND or no option is specified, positioning of a file at OPEN time is controlled by the operating system (see the Programmer's Guide).

If a sequential input file is designated with the OPTIONAL clause in the File Control paragraph of the Environment Division, the clause is

treated as comments. The desired effect is achieved by specifying the DUMMY or NULLFILE parameter in the DD statement for the file. If the parameter is specified, the first READ statement for this file causes control to be passed to the imperative statement after the key words AT END.

The I-O option permits 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.

When the I-O option is used, the execution of the OPEN statement includes the following steps:

1. The label is checked.

2. The user's label subroutine, if one is specified by a USE sentence, is executed.

3. The label is written.

Format 2 may be specified only for standard sequential files. Since the positioning options are only applicable to tape files, they will be ignored if, at execution time, a mass storage device is assigned to the file.

In Format 2, when the subsequent volume is not to be mounted on the same device, the LEAVE, REREAD, and DISP options define the positioning of volumes at end of volume in two cases:

1. When automatic end of volume occurs (automatic end of volume occurs when an end-of-volume condition is detected during execution of a READ or WRITE statement).

2. When execution of a CLOSE REEL/UNIT WITH POSITIONING statement causes forced end of volume.

The LEAVE option causes each volume affected to be positioned at the end of the file on the volume, unless the REVERSED option is also specified. If the REVERSED option is specified, the tape is positioned at the beginning (i.e., the logical end) of the file on each volume affected.

The REREAD option causes each volume affected to be backspaced and positioned at the beginning of the file on the volume, unless the REVERSED option is specified. If the REVERSED option is specified, the tape is repositioned at the end (i.e., the logical beginning) of the file on each volume.

If the DISP option is specified, the action taken -- such as rewind, unload, etc. -- is a function of the DISP parameter of the associated DD statement for the file. The action is the same, whether or not the REVERSED option is specified.

If no volume positioning is specified, DISP is assumed.

START Statement

The START statement initiates processing of a segment of a sequentially accessed indexed file at a specified key. Processing may be specified to begin at a specific NOMINAL KEY that matches a RECORD KEY within the file, or it may be specified to start at the beginning of a specific generic class of records. Processing begins with the first record of the specified generic key class.

Format 1

```
START file-name [INVALID KEY imperative-statement]
```

Format 2 (Version 3 and Version 4)

```
START  file-name  USING KEY data-name { EQUAL TO } identifier
                                      {    =    }

       [INVALID KEY  imperative-statement]
```

Normally, an indexed file in the sequential access mode is processed sequentially from the first record until the last, or until the file is closed. If processing is to begin at other than the first record, or if processing is to continue at other than the next sequential record, then a START statement must be executed prior to the READ statement for the record desired. Processing then continues sequentially until a subsequent START or CLOSE statement is executed, or until end-of-file is reached.

If processing is to begin at the first record in the file, a START statement is not required before the first READ statement.

File-name: The file-name must be defined by a file description entry in the Data Division.

Format 1: When Format 1 is used, the contents of the NOMINAL KEY are used as the key value of the record at which processing is to begin. In this instance, this key value must be placed in the data-name specified by the NOMINAL KEY clause for this file before the START statement is issued.

When the INVALID KEY option is specified, control is passed to the imperative-statement following INVALID KEY when the contents of the NOMINAL KEY field are invalid. The key is considered invalid when the record is not found in the file.

In both Format 1 and Format 2, if the INVALID KEY option is not specified, an invalid key condition causes the execution of the USE AFTER STANDARD ERROR procedure, if specified, for the file. If neither is specified, abnormal termination may result.

Program Product Information (Version 3 and Version 4)

Format 2: When Format 2 is used, the programmer requests that processing begin with the first record of a specified generic key class.

Data-name must be the data-name specified in the RECORD KEY clause for the file.

Identifier contains the generic key value for the request, and may be any data item less than or equal in length to the RECORD KEY for the file. Identifier may not appear in the record description for this file.

The USAGE of data-name and identifier should be DISPLAY.

When the USING KEY option is specified, then before a START statement is issued, the user must place the desired value (the generic key) into identifier. When the START statement is executed, the contents of identifier are compared with the contents of the RECORD KEY data-name. The comparison is non-algebraic, from left to right. The length of the comparison is controlled by the length of identifier. Sequential processing of the file begins at the first record whose RECORD KEY contains a match with the contents of identifier.

Identifiers of different lengths may be specified for different START statements for the same file.

For example, if the data records in a file contain a 10-character RECORD KEY field, and the user wishes to process the file from the beginning of a generic class defined by the first five characters

identifier field. If he later wishes to begin processing from the beginning of another generic class defined by the first three characters within the RECORD KEY field, his next START statement may specify a 3-character identifier field.

Note that upon execution of a Format 2 START statement the contents of the NOMINAL KEY field associated with the file remain unchanged.

If identifier is greater in length than data-name, then the excess low-order characters of identifier are truncated.

In Format 2, when the INVALID KEY option is specified, control is passed to the imperative-statement following INVALID KEY when the contents of identifier are invalid. Identifier is considered invalid when the generic key class it contains is not found in the file.

SEEK/READ Statements

SEEK Statement

The SEEK statement serves only as documentation, and is meant to initiate the accessing of a mass storage data record for subsequent reading or writing.

Format
SEEK file-name RECORD

The file-name must be defined by a file description entry in the Data Division.

A SEEK statement pertains only to direct files in the random access mode and may be executed prior to the execution of a READ or WRITE statement.

The SEEK statement uses the contents of the data-name in the ACTUAL KEY clause for the location of the record to be accessed. If the key is invalid, when the next READ or WRITE statement for the associated file is executed, control will be passed to the imperative statement following the INVALID KEY option.

However, this statement (if specified) is treated as a comment.

READ Statement

The functions of the READ statement are:

1. For sequential file processing, to make available the next logical record from an input file and give control to a specified imperative statement when end-of-file is detected.
2. For random file processing, to make available a specific record from a mass storage file and give control to a specified imperative statement if the contents of the associated ACTUAL KEY or NOMINAL KEY data item are found to be invalid.

Format
READ file-name RECORD [INTO identifier] {AT END / INVALID KEY} imperative-statement

An OPEN statement must be executed for the file prior to the execution of the first READ for that file. When a READ statement is

executed, the next logical record in the named file becomes accessible in the input area defined by the associated record description entry.

The record remains in the input area until the next input/output statement for that file is executed. No reference can be made by any statement in the Procedure Division to information that is not actually present in the current record. Thus, it is not permissible to refer to the nth occurrence of data that appears fewer than n times. If such a reference is made, no assumption should be made about the results in the object program.

When a file consists of more than one type of logical record, these records automatically share the same storage area; this is equivalent to an implicit redefinition of the area. Only the information that is present in the current record is accessible.

FILE-NAME: The file-name must be defined by a file description entry in the Data Division.

INTO IDENTIFIER OPTION: The INTO identifier option makes the READ equivalent to a READ statement and a MOVE statement. Identifier must be the name of a Working-Storage or Linkage Section entry, or an output record of a previously opened file. When this option is used, the current record becomes available in the input area, as well as in the area specified by identifier. Data will be moved into identifier in accordance with the COBOL rules for the MOVE statement without the CORRESPONDING option.

Program Product Information (Version 4)

Communication Section.

AT END OPTION: The AT END option must be specified for all files in the sequential access mode. If, during the execution of a READ statement, the logical end of the file is reached, control is passed to the imperative statement specified in the AT END phrase. After execution of the imperative statement associated with the AT END phrase, a READ statement for that file must not be given without prior execution of a CLOSE statement and an OPEN statement for that file.

If a DD card for a sequential file specifies the DUMMY or NULLFILE parameter, on the first READ for the file, control will be passed to the imperative statement in the AT END phrase. For purposes of language consistency, the OPTIONAL clause should be specified for this type of file.

If, during the processing of a multivolume file in the sequential access mode, the end of tape reel or mass storage unit is recognized on a sequential READ, the following operations are carried out:

a. The standard ending volume label procedure and the user's ending volume label procedure if specified by the USE statement. The order of execution of these two procedures is specified by the USE statement. Positioning of the volume is performed as specified in the OPEN volume positioning option.

b. A volume switch.

c. The standard beginning volume label procedure and the user's beginning volume procedure if specified. The order of execution is again specified by the USE statement.

d. The first data record on the new volume is made available.

INVALID KEY OPTION: The INVALID KEY option must be specified for mass storage files in the random access mode. This compiler will allow the user to omit this option. If the INVALID KEY option is not specified, an invalid key condition will cause the execution of the USE AFTER STANDARD ERROR procedure if specified for the file. If no error processing declarative is specified for the file, the invalid key condition will be ignored.

If ACCESS IS RANDOM is specified for the file, the contents of the ACTUAL or NOMINAL KEY for the file must be set to the desired value before the execution of the READ statement.

Only the track specified in the ACTUAL KEY is searched for the record being read.

If the desired record cannot be found on the specified track, the search can be extended to include a specific number of tracks or to include the entire file, with the LIMCT parameter on the DD card.

Control is passed to the imperative statement following INVALID KEY when the contents of the ACTUAL KEY or NOMINAL KEY field are invalid.

The key is considered invalid under the following conditions:

1. For a direct file that is accessed randomly: when the record is not found within the search limits, or when the track address in the ACTUAL KEY field is outside the limits of the file.

2. For an indexed file that is accessed randomly: when no record exists whose RECORD KEY field matches the contents of the NOMINAL KEY field.

3. For a relative file that is accessed randomly: when the relative record number in the NOMINAL KEY field is outside the limits of the file.

When the execution of a READ statement for an indexed file causes an INVALID KEY condition, a REWRITE statement should not be executed for the record with that key.

WRITE Statement

The WRITE statement releases a logical record to an output file. It can also be used for vertical positioning of a print file. For sequentially accessed mass storage files, the WRITE statement passes control to a specified imperative statement if no space is available in which to write the record. For randomly accessed mass storage files, the WRITE statement passes control to a specified imperative statement if the contents of the associated ACTUAL or NOMINAL KEY data item are found to be invalid.

The WRITE statement can also be used for pocket selection for a card punch file.

Format 1

```
WRITE record-name [FROM identifier-1]

  [ { BEFORE } ADVANCING { identifier-2 LINES }          ]
    { AFTER  }           { integer LINES    }
                         { mnemonic-name    }

    [AT { END-OF-PAGE } imperative-statement]
        { EOP        }
```

Format 2

```
WRITE record-name  [FROM identifier-1]

    AFTER POSITIONING { identifier-2 }  LINES
                      { integer      }

    [AT { END-OF-PAGE } imperative-statement]
        { EOP        }
```

Format 3

```
WRITE record-name [FROM identifier-1]

    INVALID KEY imperative-statement
```

An OPEN statement for a file must be executed prior to executing the first WRITE statement for that file.

For files in both the sequential and random access modes, the logical record released is no longer available after the WRITE statement is executed.

RECORD-NAME: The record-name is the name of a logical record in the File Section of the Data Division and must not be part of a sort-file.

FROM OPTION: When the FROM option is written, it makes the WRITE equivalent to the statement MOVE identifier-1 TO record-name followed by the statement WRITE record-name. Data is moved into record-name in accordance with the COBOL rules for the MOVE statement without the CORRESPONDING option. Identifier-1 should be defined in the Working-Storage Section, the Linkage Section, or in another FD.

Program Product Information (Version 4)

For Version 4, identifier may be the name of an entry in the Communication Section.

FORMAT 1 AND FORMAT 2: Formats 1 and 2 are used only with standard sequential files.

The ADVANCING and POSITIONING options allow control of the vertical positioning of each record on the printed page. If the ADVANCING or POSITIONING option is not used, automatic advancing is provided to cause single spacing. If the ADVANCING or POSITIONING option is used, automatic advancing is overridden.

When the ADVANCING or POSITIONING option is written for a record in a file, every WRITE statement for records in the same file must also contain one of these options. The POSITIONING and ADVANCING options may not both be specified for a file.

When the ADVANCING or POSITIONING option is used, the first character in each logical record for the file must be reserved by the user for the control character. The compiler will generate instructions to insert the appropriate carriage control character as the first character in the record. If the records are to be punched, the first character is used for pocket selection. It is the user's responsibility to see that the appropriate channels are punched on the carriage control tape.

Format 1: In the ADVANCING option, when identifier-2 is used, it must be the name of a nonnegative numeric elementary item (less than 100) described as an integer. If identifier-2 is specified, the printer page is advanced the number of lines contained in the identifier.

When integer is used in the ADVANCING option, it must be nonnegative and less than 100. If integer is specified, the printer page is advanced the number of lines equal to the value of integer.

When the mnemonic-name option is used in the ADVANCING option, it must be defined as a function-name in the Special-Names paragraph of the Environment Division. It is used for a skip to channels 1-9, 10-12, and to suppress spacing. It is also used for pocket selection for a card punch file.

The action taken for each function-name is given in Table 19.

If the BEFORE ADVANCING option is used, the record is written before the printer page is advanced according to the preceding rules.

If the AFTER ADVANCING option is used, the record is written after the printer page is advanced according to the preceding rules.

Table 19. Action Taken for Function-names -- ADVANCING Option

Function-name	Action Taken
CSP	Suppress spacing
C01 through C09	Skip to channel 1 through 9, respectively
C10 through C12	Skip to channel 10, 11, and 12, respectively
S01, S02	Pocket select 1 or 2 on the IBM 1442, and P1 or P2 on the IBM 2540

Format 2: In the AFTER POSITIONING option, identifier-2 must be described as a one-character alphanumeric item, that is, with PICTURE X. Table 20 shows the valid values that identifier-2 may assume and their interpretations.

In Format 2, integer must be unsigned, and must be the value 0, 1, 2, or 3. The values assume the meanings given in Table 21.

Table 20. Values of Identifier-2 and Interpretations -- POSITIONING Option

Value of Identifier-2	Interpretation
b(blank)	Single-spacing
0	Double-spacing
-	Triple-spacing
+	Suppress spacing
1 - 9	Skip to channel 1 - 9, respectively
A, B, C	Skip to channel 10, 11, 12, respectively
V, W	Pocket select 1 or 2, respectively, on the IBM 1442, and P1 or P2 on the IBM 2540.

Table 21. Values of Integer and Interpretations -- POSITIONING Option

Integer	Interpretation
0	Skip to channel 1 of next page (carriage control "eject")
1	Single-spacing
2	Double-spacing
3	Triple-spacing

If the AFTER POSITIONING option is used, the record is written after the printer page is advanced according to the preceding rules.

END-OF-PAGE OPTION: The END-OF-PAGE condition exists when the channel 12 punch on the carriage control tape is sensed by an on-line printer. The printer file must be defined as an unblocked single buffered file. The programmer should ensure that every WRITE statement in the program (whether using the ADVANCING or the POSITIONING option) advances the printer only one line at a time; otherwise, the channel 12 punch may not be sensed and results may be unpredictable.

When an END-OF-PAGE condition exists, the writing and spacing operations are completed before the END-OF-PAGE imperative statement is executed. The END-OF-PAGE statement will be executed only for an on-line printer.

Note: DISPLAY, EXHIBIT, WRITE AFTER POSITIONING, and WRITE AFTER ADVANCING statements all cause the printer to space before printing. However, a simple WRITE statement without any option given, or a WRITE BEFORE ADVANCING statement both cause the printer to space after printing. Therefore, it is possible that mixed DISPLAY, EXHIBIT, and

simple WRITE statements or WRITE BEFORE ADVANCING statements within the same program may cause overprinting.

MULTIVOLUME SEQUENTIAL FILES: The discussion below applies to all multivolume tape files and mass storage files in the sequential access mode.

After the recognition of an end-of-volume on a multivolume OUTPUT file in the sequential access mode, the WRITE statement performs the following operations:

1. The standard ending volume label procedure and the user's ending volume label procedure if specified by a USE statement. The order of execution of these two procedures is specified by the USE statement. Positioning of the volume is performed as specified in the OPEN volume positioning option.

2. A volume switch.

3. The standard beginning volume label procedures and the user's beginning volume label procedure if specified by the USE statement. The order is specified by the USE statement.

FORMAT 3: Format 3 is used for randomly or sequentially accessed mass storage files.

For standard sequential files opened as OUTPUT, the WRITE statement can be specified only to create the file. For such files opened as I-O, a READ statement must be executed before the WRITE statement is issued; the WRITE statement updates the record retrieved by the previous READ statement.

For sequentially accessed direct files, the WRITE statement creates a record for an OUTPUT file. If a record with the same ACTUAL KEY already exists, the WRITE statement replaces that record; otherwise it creates a new record.

For sequentially accessed indexed or relative files, the WRITE statement creates a record for an OUTPUT file.

If ACCESS IS RANDOM is specified for the file, the contents of the ACTUAL or NOMINAL KEY field for the file must be set to the desired value before the execution of a WRITE statement. For a direct file, the track specified in the ACTUAL KEY field is searched for space for the record to be written.

If the required space cannot be found or if the record is not found on the specific track, the search can be extended to include a specific number of tracks, or to include the entire file, with the LIMCT parameter on the DD card.

INVALID KEY OPTION: The INVALID KEY option must be specified for a file that resides on a mass storage device. This compiler will allow the user to omit this option. If the INVALID KEY option is not specified, an invalid key condition causes the execution of the USE AFTER STANDARD ERROR procedure if specified for the file. If no error processing declarative is specified for the file, the invalid key condition is ignored.

Control is passed to the imperative statement following INVALID KEY where the following conditions exist:

1. For a mass storage file in the sequential access mode and opened as OUTPUT: when no space is available in which to write the record.

2. For a direct file in the random access mode and opened as I-O or OUTPUT: when a record is being added to the file, and any one of the following conditions occurs:

a. The track number specified in the ACTUAL KEY field is outside the limits of the file.

b. For files with mode F records, the figurative constant HIGH VALUE (or its equivalent) has been moved into the first character position of the symbolic portion of the ACTUAL KEY field.

3. For a direct file in the random access mode, opened as I-O, and a record is being updated: when the record is not found, or when the track number in the ACTUAL KEY field is outside the limits of the file.

4. For an indexed file in the sequential access mode, opened as OUTPUT, and either one of the following conditions occurs:

   a. The contents of the RECORD KEY field are not in ascending order when compared with the contents of the RECORD KEY field of the preceding record.

   b. The contents of the RECORD KEY field duplicate those of the preceding record.

5. For an indexed file in the random access mode, opened as I-O, and a record is being added to the file: when the contents of the NOMINAL KEY field associated with the record to be added duplicate the contents of a RECORD KEY field already in the file.

RANDOMLY ACCESSED DIRECT FILES: For a direct file in the random access mode that is opened I-O, the following considerations apply:

1. If D is specified in the ASSIGN clause system-name, then:

   a. a WRITE statement updates a record if the preceding [illegible] statement was for a record with the same ACTUAL KEY.

   b. a WRITE statement adds a new record to the file, whether or not a duplicate record exists, if the preceding READ statement was not for a record with the same ACTUAL KEY.

2. If W is specified in the ASSIGN clause system-name, then:

   a. a REWRITE statement searches for a record with a matching ACTUAL KEY, and updates it.

   b. a WRITE statement adds a new record to the file, whether or not a duplicate key exists.

REWRITE Statement

The function of the REWRITE statement is to replace a logical record on a mass storage device with a specified record, if the contents of the associated ACTUAL KEY or NOMINAL KEY are found to be valid.

Format
REWRITE record-name [FROM identifier] [INVALID KEY imperative-statement]

The record-name is the name of a logical record in the File Section of the Data Division and must not be part of a sort-file.

The READ statement for a file must be executed before a REWRITE statement for the file can be executed, except for a direct file accessed randomly. A REWRITE statement can be executed only for files opened as I-O; any file organization is valid.

When the FROM option is used, the REWRITE statement is equivalent to the statement MOVE identifier TO record-name followed by the statement REWRITE record-name. Identifier should be defined in the Working-Storage Section, Linkage Section, or in another FD.

For a direct file that is accessed randomly, control is passed to the imperative statement following INVALID KEY when the contents of the ACTUAL KEY field are invalid. The key is considered invalid when the record is not found, or the track number specified in ACTUAL KEY is outside the limits of the file.

For a relative file that is accessed randomly, control is passed to the imperative statement following INVALID KEY when the contents of the NOMINAL KEY field are invalid. The key is considered invalid when the relative record number in the NOMINAL KEY field is outside the limits of the file.

An INVALID KEY error will never be detected when updating a randomly accessed indexed file, and the results of a REWRITE statement are unpredictable. If, when randomly reading a record of an indexed file, an INVALID KEY condition occurs, the record should not be rewritten. If the INVALID KEY option is not specified, an invalid key condition will cause the execution of the USE AFTER STANDARD ERROR procedure, if specified for the file. If no error processing declarative is specified for the file, the invalid key condition will be ignored.

If ACCESS IS RANDOM is specified for the file, the ACTUAL or NOMINAL KEY must be set to the desired value prior to the execution of the REWRITE statement.

Note: For the relationship between the REWRITE statement and the ASSIGN clause system-name, see the paragraphs on Randomly Accessed Files in "Write Statement".

ACCEPT Statement

The function of the ACCEPT statement is to obtain data from the system logical input device (SYSIN), or from the CONSOLE.

Format 1

```
ACCEPT identifier [FROM  { SYSIN       }  ]
                         { CONSOLE     }
                         { mnemonic-name }
```

Format 2 (Version 4)

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

FORMAT 1: Identifier may be either a fixed-length group item or an elementary alphabetic, alphanumeric, external decimal, or external floating-point item. Identifier may not be any special register except TALLY. The data is read and the appropriate number of characters is moved into the area reserved for identifier. No editing or error checking of the incoming data is done.

If the input/output device specified by an ACCEPT statement is the same one designated for a READ statement, the results may be unpredictable.

Mnemonic-name may assume either the meaning SYSIN or CONSOLE. Mnemonic-name must be specified in the SPECIAL-NAMES paragraph of the Environment Division. If mnemonic-name is associated with CONSOLE, identifier must not exceed 114 character positions in length. If the FROM option is not specified, SYSIN is assumed.

When an ACCEPT statement with the FROM mnemonic-name for CONSOLE option or FROM CONSOLE is executed, the following actions are taken:

1. A system generated message code is automatically displayed,

2. Execution is suspended. When a console input message, preceded by the same message code as in point 1 above, is identified by the control program, execution of the ACCEPT statement is resumed and the message is moved to the specified identifier and left justified, regardless of the PICTURE. If the field is not filled, the low-order positions may contain invalid data.

If mnemonic-name is associated with SYSIN or if the FROM SYSIN option is specified, an input record size of 80 is assumed. If the size of the accepting data item is less than 80 characters, the data must appear as the first set of characters within the input record; any characters beyond the length of the accepting identifier are truncated. If the size of the accepting data item is greater than 80 characters, as many input records as necessary are read until the storage area allocated to the data item is filled. If the accepting data item is greater than 80 characters, but is not an exact multiple of 80, the remainder of the last input record is not accessible.

Program Product Information (Version 4)

FORMAT 2: This format makes the information in the specified Special Register (DATE, DAY, or TIME) available to the COBOL program in the specified identifier.

The identifier may be either a fixed-length group item, or an elementary alphanumeric, alphanumeric edited, numeric edited, external decimal, binary, internal decimal, or external floating-point item. The data is moved from the specified Special Register into the identifier, following the rules for the MOVE statement without the CORRESPONDING option.

DATE has the implicit PICTURE 9(6). The sequence of data elements (from left to right) is: 2 digits for year of century, 2

digits for month of year, 2 digits for day of month. Thus July 1, 1971 is expressed as 710701.

DAY has the implicit PICTURE 9(5). The sequence of data elements (from left to right) is: 2 digits for year of century, 3 digits for day of year. Thus July 1, 1971 is expressed as 71183.

TIME has the implicit PICTURE 9(8). The sequence of data elements (from left to right) is: 2 digits for hour of day, 2 digits for minute of hour, 2 digits for second of minute, 2 digits for hundredths of second. Thus 2:41 PM is expressed as 14410000.

DISPLAY Statement

The function of the DISPLAY statement is to write data on an output device.

Format

```
DISPLAY { literal-1    } [ literal-2    ] ... [UPON { CONSOLE    } ]
        { identifier-1 }   [ identifier-2 ]         { SYSPUNCH   }
                                                    { SYSOUT     }
                                                    { mnemonic-name }
```

Mnemonic-name must be specified in the SPECIAL-NAMES paragraph of the Environment Division. Mnemonic-name may be associated only with the reserved words CONSOLE, SYSPUNCH, or SYSOUT.

When the UPON option is omitted, the system logical output device (SYSOUT) is assumed.

A maximum logical record size is assumed for each device. For CONSOLE (the system logical console device), the maximum is 100 characters. For SYSOUT (the system logical output device), the maximum is 120 characters. For SYSPUNCH (the system punch device), the maximum is 72 characters, with positions 73-80 used for the PROGRAM-ID name.

If the total character count of all operands is less than the maximum (or 72 for SYSPUNCH), the remaining character positions are padded with blanks. If the count exceeds the maximum size, operands are continued in the next record. As many records as necessary are written to display all the operands specified. Those operands pending at the time of the break are split between lines if necessary.

Identifiers described as USAGE COMPUTATIONAL, COMPUTATIONAL-1, COMPUTATIONAL-2, or COMPUTATIONAL-3 are converted automatically to external format, as follows:

1. Internal-decimal and binary items are converted to external decimal. Negative signed values cause a low-order sign overpunch to be developed.

2. Internal floating-point items are converted to external floating-point.

3. No other data items require conversion.

For example, if three internal decimal items have values of -34, +34, and 34, they are displayed as 3M, 34, and 34, respectively.

If a figurative constant is specified as one of the operands, only a single occurrence of the figurative constant is displayed.

Identifier may not be any special register except TALLY.

When a DISPLAY statement contains more than one operand, the data contained in the first operand is stored as the first set of characters, and so on, until the output record is filled. This operation continues until all information is displayed. Data contained in an operand may extend into subsequent records.

Note: DISPLAY, EXHIBIT, WRITE AFTER POSITIONING, and WRITE AFTER ADVANCING statements all cause the printer to space before printing. However, a simple WRITE statement without any option given, or a WRITE BEFORE ADVANCING statement both cause the printer to space after printing. Therefore, it is possible that mixed DISPLAY, EXHIBIT, and simple WRITE statements or WRITE BEFORE ADVANCING statements within the same program may cause overprinting.

CLOSE Statement

The CLOSE statement terminates the processing of input/output reels, units, and files, with optional rewind and/or lock where applicable.

Format 1

```
CLOSE  file-name-1  [REEL]   [WITH {NO REWIND}]
                    [UNIT]        {LOCK     }

       [file-name-2 [REEL]   [WITH {NO REWIND}]] ...
                    [UNIT]        {LOCK     }
```

Format 2

```
CLOSE  file-name-1    [WITH {NO REWIND}]
                            {LOCK     }
                            {DISP     }

       [file-name-2   [WITH {NO REWIND}] ] ...
                            {LOCK     }
                            {DISP     }
```

Format 3

```
CLOSE file-name-1 { REEL } [WITH { NO REWIND }]
                  { UNIT }       { LOCK      }
                                 { POSITIONING }

      [file-name-2{ REEL } [WITH { NO REWIND } ]] ...
                  { UNIT }       { LOCK      }
                                 { POSITIONING }
```

Each file-name is the name of a file upon which the CLOSE statement is to operate; it must not be the name of a sort-file.

The file-name must be defined in a file description entry in the Data Division.

A file may be closed more than once, but each CLOSE statement (without the REEL/UNIT option) must logically be preceded by an OPEN statement for that file. A file that is opened within a run unit must be closed within that run unit.

The REEL, DISP, WITH POSITIONING, and WITH NO REWIND options are applicable only to tape files. The UNIT option is applicable only to mass storage files in sequential access mode. Since device assignments can be specified at execution time, the words REEL and UNIT are interchangeable. If a file is assigned to a mass storage device, the DISP, WITH POSITIONING, and NO REWIND options will be ignored.

For purposes of showing the effect of various CLOSE options as applied to various storage media, all input/output files are divided into the following categories:

1. Unit record volume. A file whose input or output medium is such that rewinding, units, and reels have no meaning.
2. Sequential single volume. A sequential file that is entirely contained on one volume. There may be more than one file on this volume.
3. Sequential multivolume. A sequential file that may be contained on more than one volume.
4. Random single volume. A file in the random access mode that may be contained on a single mass storage volume.
5. Random multivolume. A file in the random access mode that may be contained on more than one mass storage volume.

Note: See also "File Processing Summary" in the Environment Division, and "Appendix D: Summary of File Processing Techniques and Applicable Statements and Clauses."

Sequential File Processing

The results of executing each CLOSE option for each type of file are summarized in Table 22. The definitions of the symbols in the illustration are given below. Where the definition of the symbol depends on whether the file is an input or output file, alternate

definitions are given; otherwise, a definition applies to files opened as INPUT, OUTPUT, and I-O.

A -- Previous Volumes Unaffected

All volumes in the file prior to the current volume are processed according to standard volume switch procedures except those volumes controlled by a prior CLOSE REEL/UNIT statement. The standard switch procedure positions the volumes as specified by the volume positioning option of the OPEN statement.

B -- No Rewind of Current Reel

The current volume is positioned at the logical end of the file on the volume.

C -- Standard Close File

Files opened as INPUT and I-O: If the file is positioned at its end, and label records are specified, the standard ending label procedure and the user ending label procedure (if specified by the USE statement) are performed. The order of execution of these two procedures is specified by the USE statement. Standard system closing procedures are then performed.

If the file is positioned at its end, and label records are not specified for the file, standard system closing procedures are performed.

If the file is positioned other than at its end, the standard system closing procedures are performed. Even if label procedures are specified, no label processing is performed.

(An INPUT or I-O file is considered to be at its end if the AT END phrase of the READ statement has been executed, and no CLOSE statement has been executed.)

Files opened as OUTPUT: If label records are specified for the file, standard ending label procedures and user ending label procedures (if specified by the USE statement) are performed. The order of execution of these two procedures is specified by the USE statement. Standard system closing procedures are then performed.

If label records are not specified for the file, standard system closing procedures are performed.

D -- Standard Reel/Unit Lock

This feature has no meaning in this system and is treated as comments.

E -- Standard File Lock

The compiler ensures that this file cannot be opened again during this execution of the object program.

F -- Standard Close Volume

Files Opened as INPUT and I-O: The following operations are performed:

1. A volume switch.
2. The standard beginning volume label procedure and the user's beginning volume label procedure (if specified by the USE

statement). The order of execution of these two procedures is specified by the USE statement.

3. Makes the next data record on the new volume available to be read.

Files Opened as OUTPUT: The following operations are performed:

1. The standard ending volume label procedure and the user's ending volume label procedure (if specified by the USE statement). The order of execution of these two procedures is specified by the USE statement.

2. A volume switch.

3. The standard beginning volume label procedure and the user's beginning volume label procedure (if specified by the USE statement). The order of execution of these two procedures is specified by the USE statement.

G -- Rewind

The current volume is positioned at its beginning.

H -- POSITIONING of Current Reel

The current volume is positioned as specified by the volume positioning option of the OPEN statement.

J -- DISP

The positioning of the current volume (such as rewind, unload, etc.) is a function of the DISP parameter of the associated DD statement for the file. The action is the same, whether or not the file was opened REVERSED.

X -- Illegal

This is an illegal combination of a CLOSE option and a file type. The results at object time may be unpredictable.

Table 22. Relationship of Types of Sequential Files and the Options of the CLOSE Statement

CLOSE Option \ FILE Type	Unit Record	Sequential Single-Volume	Sequential Multivolume
CLOSE	C	C, G	C, G, A
CLOSE WITH LOCK	C, E	C, G, E	C, G, E, A
CLOSE WITH NO REWIND	X	C, B	C, B, A
CLOSE WITH DISP	X	C, J	C, J, A
CLOSE REEL	X	X	F, G
CLOSE REEL WITH LOCK	X	X	F, G, D
CLOSE REEL WITH NO REWIND	X	X	F, B
CLOSE REEL WITH POSITIONING	X	X	F, H
CLOSE UNIT	X	X	F
CLOSE UNIT WITH LOCK	X	X	F, D
CLOSE UNIT WITH POSITIONING	X	X	F

General Considerations: A file is designated as optional by specifying the DUMMY or NULLFILE parameter on the DD card for the file. If an optional file is not present, the standard end-of-file processing is not performed. For purposes of language consistency, the OPTIONAL phrase of the SELECT clause should be specified for this type of file.

If a CLOSE statement without the REEL or UNIT option has been executed for a file, the next input/output statement to be executed for that file must be an OPEN statement.

Random File Processing

The results of executing each CLOSE option for each type of file are summarized in Table 23. The definitions of the symbols in the figure are given below. Where the definition depends on whether the file is an input or output file, alternate definitions are given; otherwise, a definition applies to files opened as INPUT, OUTPUT and I-O.

K -- Standard Close File

The standard ending label procedure and the user ending label procedure (if specified by the USE statement) are performed. For I-O files and OUTPUT files the labels are written. Standard system closing procedures are then performed.

L -- Standard File Lock

The compiler ensures that this file cannot be opened again during this execution of this object program.

Table 23. Relationship of Types of Random Files and the Options of the CLOSE Statement

CLOSE Option \ FILE Type	Random Single-Volume	Random Multivolume
CLOSE	K	K
CLOSE WITH LOCK	K, L	K, L

Subprogram linkage statements are special statements that permit communication between object programs. These statements are CALL, ENTRY, GOBACK, and EXIT.

Program-Product Information (Version 4)

A new option of the CALL statement and the addition of the CANCEL statement permit dynamic loading and deletion of COBOL subprograms in the COBOL processing environment.

The CALL statement, as it has previously been specified for OS Full American National Standard COBOL, has been static. That is, the main COBOL program and all subprograms invoked with the CALL statement must have been part of the same load module. Thus, when a subprogram was called it was already core-resident, and a branch to it occurred. Subsequent execution of CALL statements entered that subprogram in its last-used state. If alternate entry points were specified, then any CALL to the subprogram could select any of the alternate ENTRY points at which to enter the subprogram. If the linking of all subprograms with the main program resulted in a load module that required more main storage than was available, then the user could utilize the Segmentation feature. Now, with the implementation of the dynamic CALL and CANCEL statements, the COBOL user can control dynamically the modules that are to be core-resident.

For the Version 4 Compiler, the CALL statement can also be specified as dynamic; that is, the called subprogram is not link edited with the main program, but is instead link edited into a separate load module, and at execution time is loaded only if and when it is required (that is, when it is called).

Each subprogram invoked with a dynamic CALL statement may be part of a different load module, which is a member of the system link library or of a user-supplied private library. The execution of the dynamic CALL statement to a subprogram that is not core-resident results in the loading of that subprogram from secondary storage into the region/partition containing the main program, and a branch to the subprogram.

Thus, the first dynamic CALL to a subprogram obtains a fresh copy of the subprogram. Subsequent calls to the same subprogram (either by the original caller or by any other subprogram within the same region/partition) result in a branch to the same copy of the subprogram in its last-used state. However, when a CANCEL statement is issued for that subprogram, the core storage occupied by the subprogram is freed, and a subsequent CALL to the subprogram will function as though it were the first. A CANCEL statement referring to a called subprogram may be issued by a program other than the original caller. In order for the CALL statement to function as defined by CODASYL, the user subprograms must be linkage edited as non-reentrant and non-serially-reusable.

The user can specify the mode (static or dynamic) of the CALL literal statement through new parameters of the EXEC job control statement. Static mode is the default option. Subprograms invoked through the CALL identifier statement are always dynamically loaded at object time.

When the dynamic CALL statement is used at object time, the COBOL Library Management Facility must be used by the main program

and all subprograms in one region/partition. Otherwise, multiple copies of library subroutines may be resident at one time and cause unpredictable results.

User subprograms that are to be invoked at object time with the dynamic CALL statement must be members of the system link library or of a user-supplied private library.

In the sections that follow, the language for both the static and dynamic CALL statement is described. The CANCEL statement, which functions only for programs that have been dynamically called, is also described.

(Additional information on the static and dynamic CALL statements, and the associated EXEC job control statement parameters, can be found in OS Full American National Standard COBOL Compiler and Library, Version 4, Programmer's Guide, Order No. SC28-6456.)

CALL Statement

The CALL statement permits communication between a COBOL object program and one or more COBOL subprograms or other language subprograms.

Format 1
CALL literal-1 [USING identifier-1 [identifier-2]...]

Format 2 (Version 4)
CALL identifier-1 [USING identifier-2 [identifier-3]...]

Literal-1 is a nonnumeric literal and is the name of the program that is being called, or the name of an entry point in the called program. The program in which the CALL statement appears is the calling program. Literal-1 must conform to the rules for formation of a program-name. The first eight characters of literal-1 are used to make the correspondence between the called and calling program.

When the called program is to be entered at the beginning of the Procedure Division, literal-1 must specify the program-name (in the PROGRAM-ID paragraph) of the called program. The called program must have a USING clause as part of its Procedure Division header if there is a USING clause in the CALL statement which invoked it.

When the called program is to be entered at entry points other than the beginning of the Procedure Division, these alternate entry points are identified by an ENTRY statement and a USING option corresponding to the USING option of the invoking CALL statement. In the case of a CALL with a corresponding ENTRY, literal-1 must be a name other than the program-name but follows the same rules as those for the formation of a program-name.

The identifiers specified in the USING option of the CALL statement indicate those data items available to a calling program that may be referred to in a called program. When the called subprogram is a COBOL program, each of the operands in the USING option of the calling program

must be defined as a data item in the File Section, Working-Storage Section, or Linkage Section. If the called subprogram is written in a language other than COBOL, the operands of the USING option may additionally be a file-name or a procedure-name. If the operand of the USING option is a file-name, the file with which the file-name is associated must be opened in the calling program.

Program Product Information (Version 4)

For Version 4, each of the operands of the USING option in the calling program may additionally be defined as a data item in the Communication Section.

Names in the two USING lists (that of the CALL in the main program and that of the Procedure Division header or the ENTRY in the subprogram) are paired in a one-to-one correspondence. In the case of index-names, no such correspondence is established.

There is no necessary relationship between the actual names used for such paired names, but the data descriptions must be equivalent. When a group data item is named in the USING list of a Procedure Division header or an ENTRY statement, names subordinate to it in the subprogram's Linkage Section may be employed in subsequent subprogram procedural statements.

When group items with level numbers other than 01 are specified, proper word-boundary alignment is required if subordinate items are described as COMPUTATIONAL, COMPUTATIONAL-1, or COMPUTATIONAL-2.

The USING option should be included in the CALL statement only if there is a USING option in the called entry point, which is either included in the Procedure Division header of the called program or included in an ENTRY statement in the called program. The number of operands in the USING option of the CALL statement should be the same as the number of operands in the USING option of the Procedure Division header, or ENTRY statement. If the number of operands in the USING option of the CALL statement is greater than the number in the USING option in the called program, only those specified in the USING option of the called program may be referred to by the called program.

The execution of a CALL statement causes control to pass to the called program. The first time a called program is entered, its state is that of a fresh 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. Thus, the reinitialization of the following items is the responsibility of the programmer:

GO TO statements which have been altered
TALLY
Data items
ON statements
PERFORM statements
EXHIBIT CHANGED statements
EXHIBIT CHANGED NAMED statements

EXHIBIT CHANGED and EXHIBIT CHANGED NAMED operands will be compared against the value of the item at the time of its last execution, whether or not that execution was during another CALL to this program. If a branch is made out of the range of a PERFOPM, after which an exit is made from the program, the range of that PERFORM is still in effect upon a subsequent entry.

Called programs may contain CALL statements. However, a called program must not contain a CALL statement that directly or indirectly calls the calling program.

A called program may not be segmented.

Program Product Information (Version 4)

For Version 4, the following additional considerations for the CALL statement and for the CANCEL statement apply.

FORMAT 1: When the literal-1 option is specified, then the CALL statement may be either static or dynamic.

If the CALL literal-1 statement is static, the following considerations apply:

- The programmer may specify literal-1 as a program-name or as an alternate entry point, in any order.
- The first time a called program is entered, its state is that of a fresh copy of the program. Each subsequent time the program is entered, the state is as it was upon the last exit from the program.
- The CANCEL literal statement may not be specified in this case. The CANCEL identifier statement is accepted; however, the compiler then options the COBOL Library Management Facility.

If the CALL literal-1 statement is dynamic, the following considerations apply:

- A called program is in its initial state the first time it is called within a run unit, and also the first time it is called after a CANCEL statement for the called program has been executed.
- On all other entries into the called program, the state of the called program remains unchanged from its state when last executed.
- Differing entry points for one subprogram should not be specified unless an intervening CANCEL statement has been executed. (See note after the Format 2 description.)

  (For example, if subprogram A has been called using its program-name as the entry point, then until a CANCEL statement for subprogram A has been executed, subsequent CALL statements for subprogram A should all use the program-name as the entry point. After a CANCEL statement has been executed, however, some alternate entry point for subprogram A may then be specified. That entry point should be the one entry point specified until yet another CANCEL statement has been executed.)
- Names prefixed by ILBO cannot be used as names of called subprograms, or as names of alternate entry points.

FORMAT 2: The contents of identifier-1 must conform to the rules for formation of a program-name. The first 8 characters of identifier-1 are used to make the correspondence between the calling and called program.

The CALL identifier-1 statement is always dynamic. The following considerations apply:

- A called program is in its initial state the first time it is called within a run unit, and also the first time it is called after a CANCEL statement for the called program has been executed.

- On all other entries into the called program, the state of the called program remains unchanged from its state when last executed.
- Differing entry points for one subprogram should not be specified unless an intervening CANCEL statement has been executed. (See Note at the end of this description.)
- Names prefixed by ILBO cannot be used as names of called subprograms, or as names of alternate entry points.

Note: Linking two load modules together results logically in a single program with a primary entry point and an alternate entry point, each with its own name. (Each name by which a subprogram is to be dynamically invoked must be known to the system; each such name must be specified in linkage editor control statements as either a NAME or an ALIAS of the load module containing the subprogram.) Only if user modules are link edited with the attribute of non-reentrant and non-serially-reusable will a CANCEL statement guarantee a fresh copy of the subprogram upon a subsequent CALL.

Static and dynamic CALL statements may both be specified in the same program. The CALL literal-1 statement results, in this case, in the subprogram so invoked being link-edited with the main program into one load module. The CALL identifier-1 statement results in the dynamic invocation of a separate load module. When a dynamic CALL statement and a static CALL statement to the same subprogram are issued within one program, a second copy of the subprogram is loaded. Therefore, care must be used to avoid duplicate load modules.

CANCEL Statement

The CANCEL statement releases the core storage occupied by a called subprogram.

Format (Version 4)

```
CANCEL { literal-1    } [ literal-2    ] ...
       { identifier-1 }   [ identifier-2 ]
```

Each literal specified in the statement must be a nonnumeric literal.

The contents of each identifier specified must conform to the rules for formation of a program-name. The first 8 characters of the identifier are used to make the correspondence between the calling and called program.

Each literal or identifier specified in the CANCEL statement must be the same as the literal or identifier specified in the associated CALL statement(s).

The CANCEL literal statement is invalid in a program in which the CALL literal statement is static. Under the same conditions, the CANCEL identifier statement is accepted, but the compiler then options the COBOL Library Management facility.

Subsequent to the execution of a CANCEL statement, the program referred to therein ceases to have any logical relationship to the program in which the CANCEL statement appears. A subsequently executed CALL statement by any program in the run unit naming the same program will result in that program being entered in its initial state.

A logical relationship to a cancelled subprogram is established only by execution of a subsequent CALL statement.

A called subprogram is cancelled either by being directly referred to as the operand of a CANCEL statement or by the termination of the run unit of which the program is a member.

No action is taken when a CANCEL statement is executed naming a program that has not been called in this run unit or has been called and is at present cancelled. Control passes to the next statement.

To guarantee the proper execution of the CANCEL statement, prior to the execution of a CANCEL statement for a subprogram, every CALL statement for that subprogram should name the same entry point. Following the execution of a CANCEL statement, a CALL statement may specify a different entry point.

Called subprograms may contain CANCEL statements. However, a called subprogram must not contain a CANCEL statement that directly or indirectly cancels the calling program itself, or any other program higher that itself in the calling hierarchy. In such a case the run unit is terminated.

A program named in a CANCEL statement must not refer to any program that has been called and has not yet executed an EXIT PROGRAM or GOBACK statement. A program may, however, CANCEL a program that it did not call, providing that in the calling hierarchy it is higher than or equal to the program it is cancelling. For example, A calls B, and B calls C; when A receives control it can cancel C; or A calls B, and A calls C; when C receives control it can then cancel B.

ENTRY Statement

The ENTRY statement establishes an entry point in a COBOL subprogram.

Format
ENTRY literal-1 [USING identifier-1 [identifier-2]...]

Control is transferred to the entry point by a CALL statement in an invoking program.

Literal-1 must not be the name of the called program, but is formed according to the same rules followed for program-names.

Literal-1 must not be the name of any other entry point or program-name in the run unit.

A called program, once invoked, is entered at that ENTRY statement whose operand literal-1 is the same as the literal-1 specified in the CALL statement that invoked it.

USING Option

The USING option makes data items defined in the calling program available to a called program. The number of operands in the USING option of a called program must be less than or equal to the number of operands in the corresponding CALL statement of the invoking program.

The USING option may also be used at execution time to pass parameters from the EXEC statement to a main program.

The USING option may be specified in the CALL statement, the ENTRY statement, or in the Procedure Division header. The three uses are shown in the following formats:

Format 1 (Within a Calling Program)

```
CALL literal-1 [USING identifier-1 [identifier-2]...]
```

Format 2 (Version 4 -- Within a Calling Program)

```
CALL identifier-1 [USING identifier-2 [identifier-3]...]
```

Format 3 (Within a Called Program)

Option 1

```
ENTRY literal-1 [USING identifier-1 [identifier-2]...]
```

Option 2

```
PROCEDURE DIVISION [USING identifier-1 [identifier-2]...].
```

When the USING option is specified in the CALL statement, it must appear on either the Procedure Division header of the called program, or in an ENTRY statement in the called program.

The USING option may be present on the Procedure Division header or in an ENTRY statement if the object program is to function under the control of a CALL statement, and the CALL statement contains a USING clause. It may also be present on the Procedure Division header when information is to be passed from the EXEC statement to the main program.

When a called program has a USING on its Procedure Division header and linkage was effected by a CALL statement where literal-1 is the name of the called program, execution of the called program begins with the first instruction in the Procedure Division after the Declaratives Section.

USING Option

When linkage to a called program is effected by a CALL statement where literal-1 is the name of an entry point specified in the ENTRY statement of the called program, that execution of the called program begins with the first statement following the ENTRY statement.

When the USING option is present, the object program operates as though each occurrence of identifier-1, identifier-2, etc., in the Procedure Division had been replaced by the corresponding identifier from the USING option in the CALL statement of the calling program. That is, corresponding identifiers refer to a single set of data which is available to the calling program. The correspondence is positional and not by name. In the case of index-names, no such correspondence is established.

At execution time, the USING option may be used to pass parameters from the EXEC job control statement to a main COBOL program. In this case, a USING option on the Procedure Division header of a main program may contain identifier-1 as its only operand. Information from the PARM field of the EXEC statement is then available in the Linkage Section at the location specified as identifier-1. The first two bytes of identifier-1 contain a count of the number of bytes of information in the PARM field; the two bytes are set to zero if the PARM field was omitted. This two-byte field is binary and should be defined with PIC S9(4) COMP. Immediately following these two bytes is the information in the PARM field. The maximum length of the field to be passed is 100 bytes.

Each of the operands in the USING option of the Procedure Division header or the ENTRY statement must have been defined as a data item in the Linkage Section of the program in which this header or ENTRY statement occurs, and must have a level number of 01 or 77. Since the compiler assumes that each level-01 item is aligned upon a doubleword boundary, it is the programmer's responsibility to ensure proper alignment.

The following is an example of a calling program with the USING option:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALLPROG.
     .
     .
     .
DATA DIVISION.
     .
     .
     .
WORKING-STORAGE SECTION.
01  RECORD-1.
    10  SALARY      PICTURE S9(5)V99.
    10  RATE        PICTURE S9V99.
    10  HOURS       PICTURE S99V9.
     .
     .
     .
PROCEDURE DIVISION.
     .
     .
     .
    CALL "SUBPROG" USING RECORD-1.
     .
     .
     .
    CALL "PAYMASTR" USING RECORD-1.
     .
     .
     .
    STOP RUN.
```

The following is an example of a called subprogram associated with the preceding calling program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SUBPROG.
     .
     .
     .
DATA DIVISION.
     .
     .
     .
LINKAGE SECTION.
01  PAYREC.
    05  PAY          PICTURE S9(5)V99.
    05  HOURLY-RATE  PICTURE S9V99.
    05  HOURS        PICTURE S99V9.
     .
     .
     .
PROCEDURE DIVISION USING PAYREC.
     .
     .
     .
    GOBACK.
    ENTRY "PAYMASTR" USING PAYREC.
     .
     .
     .
    GOBACK.
```

Processing begins in CALLPROG, which is the calling program. When the statement

```
CALL "SUBPROG" USING RECORD-1.
```

is executed, control is transferred to the first statement of the Procedure Division in SUBPROG, which is the called program. In the calling program, the operand of the USING option is identified as RECORD-1.

When SUBPROG receives control, the values within RECORD-1 are made available to SUBPROG; in SUBPROG, however, they are referred to as PAYREC. Note that the descriptions of the subfields of PAYREC (described in the Linkage Section of SUBPROG) are the same as those for RECORD-1.

When processing within SUBPROG reaches the first GOBACK statement, control is returned to CALLPROG at the statement immediately following the original CALL statement. Processing then continues in CALLPROG until the statement

```
CALL "PAYMASTR" USING RECORD-1.
```

is reached. Control is again transferred to SUBPROG, but this time processing begins at the statement following the ENTRY statement in SUBPROG. The values within RECORD-1 are again made available to SUBPROG through the matching USING operand PAYREC. When processing reaches the second GOBACK statement, control is returned to CALLPROG at the statement immediately following the second CALL statement.

In any given execution of these two programs, if the values within RECORD-1 are changed between the time of the first CALL and the second, the values passed at the time of the second CALL statement will be the changed, not the original, values. If the programmer wishes to use the original values, then he must ensure that they have been saved.

Program Product Information (Version 4)

The following example shows a program using Format 1 of the CALL statement with the USING option (the CALL statement is static).

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALLSTAT.
     .
     .
     .
DATA DIVISION.
     .
     .
     .
WORKING-STORAGE SECTION.
01   RECORD-1.
     05  SALARY        PICTURE S9(5)V99.
     05  RATE          PICTURE S9V99.
     05  HOURS         PICTURE S99V9.
     .
     .
     .
PROCEDURE DIVISION.
     .
     .
     .
     CALL "SUBPROG" USING RECORD-1.
     .
     .
     .
     CALL "PAYMASTR" USING RECORD-1.
     .
     .
     .
     STOP RUN.
```

The following example shows a program achieving the same results with Format 2 -- the CALL identifier-1 option (the CALL statement is dynamic):

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALLDYNA.
     .
     .
     .
DATA DIVISION.
     .
     .
     .
WORKING-STORAGE SECTION.
77   IDENT  PICTURE X(8).
     .
     .
     .
01   RECORD-1.
     05  SALARY        PICTURE S9(5)V99.
     05  RATE          PICTURE S9V99.
     05  HOURS         PICTURE S99V9.
     .
     .
     .
```

```
PROCEDURE DIVISION.
     .
     .
     .
    MOVE "SUBPROG" TO IDENT.
    CALL IDENT USING RECORD-1.
     .
     .
     .
    CANCEL IDENT.
     .
     .
     .
    MOVE "PAYMASTR" TO IDENT.
    CALL IDENT USING RECORD-1.
     .
     .
     .
    STOP RUN.
```

The following is an example of a called subprogram which can be associated with either of the preceding calling programs:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SUBPROG.
     .
     .
     .
DATA DIVISION.
     .
     .
     .
LINKAGE SECTION.
01  PAYREC.
    10  PAY           PICTURE S9(5)V99.
    10  HOURLY-RATE   PICTURE S9V99.
    10  HOURS         PICTURE S99V9.
     .
     .
     .
PROCEDURE DIVISION USING PAYREC.
     .
     .
     .
    GOBACK.
    ENTRY "PAYMASTR" USING PAYREC.
     .
     .
     .
    GOBACK.
```

Processing begins in the calling program -- which may be either CALLSTAT or CALLDYNA. When the first CALL statement is executed, control is transferred to the first statement of the Procedure Division in SUBPROG, which is the called program.

Note that in each of the calling programs the operand of the USING option is identified as RECORD-1.

When SUBPROG receives control, the values within RECORD-1 are made available to SUBPROG; in SUBPROG, however, they are referred to as PAYREC. Note that the PICTURE descriptions of the subfields within PAYREC (described in the Linkage Section of SUBPROG) are the same as those for RECORD-1.

When processing within SUBPROG reaches the first GOBACK statement, control is returned to the calling program. Processing continues in that program until the second CALL statement is issued.

Note that in CALLSTAT (statically linked) that the CANCEL statement is not valid. In CALLDYNA, however, since the second CALL statement refers to another entry point within SUBPROG, a CANCEL statement is issued before the second CALL statement.

With the second CALL statement in the calling program, control is again transferred to subprog, but this time processing begins at the statement following the ENTRY statement in SUBPROG. The values within RECORD-1 are again made available to SUBPROG through the matching USING operand PAYREC. When processing reaches the second GOBACK statement, control is returned to the calling program at the statement immediately following the second CALL statement.

In any given execution of these two programs, if the values within RECORD-1 are changed between the time of the first CALL and the second, the values passed at the time of the second CALL statement will be the changed, not the original, values. If the user wishes to use the original values, then he must ensure that they have been saved.

Program Termination Considerations

There are three ways in COBOL source language to terminate a program. They are:

1. EXIT PROGRAM
2. GOBACK
3. STOP RUN

Table 24 shows the effect of each program termination statement, based on whether it is issued within a main program or a subprogram.

A main program is the highest level COBOL program invoked in a step. A subprogram is a COBOL program that is invoked by another COBOL program. (Programs written in other languages that follow COBOL linkage conventions are considered COBOL programs in this sense.)

The use of the GOBACK statement allows any COBOL program to function either as a main program or as a subprogram.

Table 24. Effect of Program Termination Statements Within Main Programs and Subprograms

Termination Statement	Main Program	Subprogram
EXIT PROGRAM	Non-operational	Return to invoking program.
STOP RUN	Return to invoker* (may be system and cause end of job step)	Return directly to invoker of main program* (may be system and cause end of job step).
GOBACK	Return to invoker* (may be system and cause end of job step)	Return to invoking program.

*If main program is called by a program written in another language that does not follow COBOL linkage conventions, return will be to this calling program.

If it is desired to pass a return code to the operating system or the invoking program, the special register RETURN-CODE must be set by the user prior to the termination statement. RETURN-CODE is a binary item whose PICTURE is S9999. The compiler initializes RETURN-CODE to 0 (zero), the normal return code for a successful completion; other values returned are conventionally in multiples of four. However, the maximum value the field can contain is 4095.

EXIT PROGRAM Statement

This form of the EXIT statement marks the logical end of a called program.

Format
paragraph-name. EXIT PROGRAM.

The EXIT statement must be preceded by a paragraph-name, and be the only statement in the paragraph.

If control reaches an EXIT PROGRAM statement while operating under the control of a CALL statement, control returns to the point in the calling program immediately following the CALL statement.

If control reaches an EXIT PROGRAM statement, and no CALL statement is active, control passes through the exit point to the first sentence of the next paragraph.

GOBACK Statement

The GOBACK statement marks the logical end of a called program.

Format
GOBACK.

A GOBACK statement must appear as the only statement, or as the last of a series of imperative statements, in a sentence.

If control reaches a GOBACK statement while operating under the control of a CALL statement, control returns to the point in the calling program immediately following the CALL statement.

If control reaches a GOBACK statement, and no CALL statement is active, control will be returned to the invoking program, which may be the system and cause end of job.

STOP RUN Statement

For a discussion of the STOP statement with the RUN option, see "Procedure-Branching Statements."

COMPILER-DIRECTING STATEMENTS

Compiler-directing statements are special statements that provide instructions for the COBOL compiler. The compiler-directing statements are COPY, ENTER, and NOTE.

COPY Statement

Prewritten source program entries can be included in a COBOL program at compile time. Thus, an installation can utilize standard file descriptions, record descriptions, or procedures without having to repeat programming them. These entries and procedures are contained in user-created libraries. They are included in a source program by means of a COPY statement (see "Source Program Library Facility").

ENTER Statement

The ENTER statement serves only as documentation, and is intended to provide a means of allowing the use of more than one source language in the same source program. This compiler allows no other source language in the source program.

Format
ENTER language-name [routine-name].

The ENTER statement is accepted as comments.

NOTE Statement

The NOTE statement allows the programmer to write commentary which will be produced on the source listing but not compiled.

Format
NOTE character string

Any combination of the characters from the EBCDIC set may be included in the character string.

If a NOTE sentence is the first sentence of a paragraph, the entire paragraph is considered to be part of the character string. Proper format rules for paragraph structure must be observed.

If a NOTE sentence appears as other than the first sentence of a paragraph, the commentary ends with the first instance of a period followed by a space.

Explanatory comments may be inserted on any line within a source program by placing an asterisk in column 7 of the line. Any combination of the characters from the EBCDIC set may be included in Area A and Area B of that line. The asterisk and the characters will be produced on the listing, but serve no other purpose.

- SORT FEATURE
- REPORT WRITER FEATURE
- TABLE HANDLING FEATURE
- SEGMENTATION FEATURE
- SOURCE PROGRAM LIBRARY FACILITY
- DEBUGGING LANGUAGE
- FORMAT CONTROL OF THE SOURCE PROGRAM LISTING
- STERLING CURRENCY FEATURE
- TELEPROCESSING (TP) FEATURE (Version 4)
- STRING MANIPULATION FEATURE (Version 4)

The COBOL programmer can gain convenient access to the sorting capability of the OS Sort/Merge by including a SORT statement and other elements of the Sort Feature in his source program. The Sort Feature provides the capability for sorting files and including procedures for special handling of these files both before and after they have been sorted. Within the limits of object-time storage, a source program may have any number of SORT statements, and each SORT statement may have its own special procedures.

The basic elements of the COBOL Sort Feature are the SORT statement in the Procedure Division and the Sort-File-Description (SD) entry, with its associated record description entries, in the Data Division. A sorting operation is based on sort-keys named in the SORT statement. A sort-key specifies the field within a record on which the file is sorted. Sort-keys are defined in the record description associated with the SD entry. The records of a file may be sorted in ascending or descending order, or in a mixture of the two; that is, the sort-keys may be specified as ascending or descending, independent of one another, and the sequence of the sorted records will conform to the mixture specified.

For more information on using the Sort Feature, see the Programmer's Guide.

Note: Language considerations for an ASCII collated sort are given in Appendix E.

ELEMENTS OF THE SORT FEATURE

To use the Sort Feature, the COBOL programmer must provide additional information in the Environment, Data, and Procedure Divisions of the source program.

The SORT statement in the Procedure Division is the primary element of a source program that performs one or more sorting operations. The term "sorting operation" means not only the manipulation by the Sort Program of sort-work-files on the basis of the sort-keys designated by the COBOL programmer, but it also includes the method of making records available to, and retrieving records from, these sort-work-files. A sort-work-file is the collection of records that is involved in the sorting operation as it exists on an intermediate device(s). Records are made available either by the USING or INPUT PROCEDURE options of the SORT statement. Sorted records are retrieved either by the GIVING or OUTPUT PROCEDURE options of the SORT statement.

In the Environment Division, the programmer must write SELECT sentences for all files used as input to and output from the sort program and for the sort-file. If checkpoint records are to be taken during the sorting operation, a RERUN statement must also be included.

In the Data Division, the programmer must include file description entries (FD) for all files that are used to provide input to or output from the sort program. He must also write a Sort-File-Description (SD) entry and its associated record description entries to describe the records that are to be sorted, including their sort-key fields.

In the Procedure Division, the programmer specifies in the SORT statement the sort-file to be sorted, the sort-key names, whether the sort is to be in ascending or descending sequence by key, and whether

records are to have special processing. If there is to be such processing, he also includes in the Procedure Division the program sections that perform the processing. Special SORT registers, if used, are referenced in the Procedure Division.

ENVIRONMENT DIVISION CONSIDERATIONS FOR SORT

There are certain statements the programmer must use in the Environment Division to use the Sort Feature. Detailed descriptions of these statements follow.

INPUT-OUTPUT SECTION

The Input-Output Section is composed of two parts: the FILE-CONTROL paragraph and the I-O-CONTROL paragraph.

FILE-CONTROL Paragraph

The FILE-CONTROL paragraph is specified once in a COBOL program. within this paragraph, all files referred to in the source program must be named in a SELECT sentence.

Files used within input and output procedures, and files named in the USING and GIVING options of the SORT statement are named in the SELECT sentence as described in "Environment Division."

The file named in the GIVING option of the SORT statement can alternately be described in the following format:

Format

```
SELECT file-name
     ASSIGN TO [integer-1] system-name-1 [system-name-2] ...
          OR system-name-3 [FOR MULTIPLE { REEL } ]
                                         { UNIT }
          [RESERVE { integer-2 } ALTERNATE [ AREA ] ].
                   { NO        }           [ AREAS ]
```

The OR option is neither required nor used by this compiler and is treated as comments.

The MULTIPLE clause function is specified by object time control cards; hence, the MULTIPLE clause is neither required nor used by this compiler. The RESERVE clause is applicable as described in "Environment Division."

SELECT Sentence for Sort File

The following format for the SELECT sentence must be used for the sort-file.

Format
SELECT sort-file-name ASSIGN TO [integer-1] system-name-1 [system-name-2] ...

The SELECT clause may be specified for the sort-file. Sort-file-name identifies the sort-file to the compiler.

The ASSIGN clause must be specified, and may be used to describe the sort work files; the integer and system-names can serve as documentation to describe the number and class of work units. However, since the system obtains this information at execution time, the compiler treats the ASSIGN clause as comments.

I-O-CONTROL Paragraph

The I-O-CONTROL paragraph specifies when checkpoints are to be taken, as well as what core storage area is to be shared by different files. The I-O-CONTROL paragraph is coded once in the source program. The checkpoint interval associated with the standard RERUN format (specified in the "Environment Division") is determined by the number of records processed for the given file. Obtaining checkpoint records within the operation of a SORT statement is specified by a special format of the RERUN statement, as described below.

RERUN Clause

The format of the RERUN clause used in conjunction with the sorting operation is:

Format
RERUN ON system-name

The presence of this format of the RERUN clause indicates that checkpoint records are to be written, at logical intervals determined by the sort program, during the execution of all SORT statements that appear in a COBOL program. Its absence indicates that, within the execution of any SORT statement, checkpoint records are not to be taken.

System-name must not be the same as any system-name used in an ASSIGN clause, but must follow the same rules of formation.

At the time the checkpoint procedure of the SORT statement takes effect, the status of all open files, whether involved in the sorting operation or not, is recorded.

SAME RECORD/SORT AREA Clause

The SAME RECORD/SORT AREA clause specifies that two or more files are to use the same storage area during processing.

Format

```
SAME { RECORD } AREA FOR file-name-1 {file-name-2}...
     { SORT   }
```

When the RECORD option is used, the named files, including any sort-files, share only the area in which the current logical record is processed. Several of the files may be open at the same time, but the logical record of only one of these files can exist in the record area at one time.

The function of the SORT option is to optimize the assignment of storage areas to a given SORT statement. The system handles storage assignment automatically; hence, the SORT option, if given, is treated as comments.

DATA DIVISION CONSIDERATIONS FOR SORT

In the Data Division the programmer must include file description entries for files to be sorted, sort-file description entries for sort work files, and record description entries for each.

FILE SECTION

The File Section of a program which contains a sorting operation must furnish information concerning the physical structure, identification, and record names of the records to be sorted. This is provided in the sort-file-description entry.

Sort File Description

A sort-file-description entry must appear in the File Section for every file named as the first operand of a SORT statement.

Format

```
SD sort-file-name
     [RECORDING MODE IS mode]
          [DATA {RECORD IS  } data-name-1 [data-name-2]...]
                {RECORDS ARE}
          [RECORD CONTAINS [integer-1 TO] integer-2 CHARACTERS]
          [LABEL {RECORD IS  } {STANDARD} ].  (Version 4)
                 {RECORDS ARE} {OMITTED  }
```

Sort-file-name is the name given to describe the records to be sorted.

The RECORDING MODE clause is discussed in "Data Division." The recording mode must be F, V, or S.

The DATA RECORDS clause specifies the names of the records in the file to be sorted. Data-name-1, data-name-2,... of the DATA RECORDS clause refer to the records described in the record descriptions associated with this SD.

The RECORD CONTAINS clause specifies the size of data records in the file to be sorted. This clause is optional. The actual size and mode (fixed or variable) of the records to be sorted are determined from the level-01 descriptions associated with a given SD entry. When the USING and GIVING options of the SORT statement are used, the record length associated with the SD must be the same length as the record associated with the FD's for the USING and GIVING files. If any of the SD data record descriptions contains an OCCURS clause with the DEPENDING ON option, variable-length records are assumed. See "Data Division" for the format assumptions that are made by the compiler when the RECORDING MODE clause is not specified.

Note: Extreme caution should be used when sorting variable length records with embedded objects of the OCCURS DEPENDING ON clause. See the section on Sorting Variable Length Records in the Programmer's Guide chapter on Using The Sort Feature.

Program Product Information (Version 4)

The Version 4 Compiler accepts the LABEL RECORDS clause, if specified, and treats it as comments.

The DATA RECORDS, LABEL RECORDS, and RECORD CONTAINS clauses are described in "Data Division".

PROCEDURE DIVISION CONSIDERATIONS FOR SORT

The Procedure Division must contain a SORT statement to describe the sorting operation and, optionally, input and output procedures. The procedure-names constituting the input and output procedures are specified within the SORT statement.

The Procedure Division may contain more than one SORT statement appearing anywhere except in the declaratives portion or in the input and output procedures associated with a SORT statement.

SORT Statement

The SORT statement provides information that controls the sorting operation. This information directs the sorting operation to obtain records to be sorted either from an input procedure or the USING file, to sort the records on a set of specified sort keys, and in the final phase of the sorting operation to make each record available in sorted order, either to an output procedure or to the GIVING file.

Format

```
SORT file-name-1 ON { DESCENDING } KEY {data-name-1} ...
                    { ASCENDING  }

    [ON { DESCENDING } KEY {data-name-2}   ...]...
        { ASCENDING  }

   {INPUT  PROCEDURE IS section-name-1 [THRU section-name-2]}
   {USING  file-name-2                                      }

   {OUTPUT PROCEDURE IS section-name-3 [THRU section-name-4]}
   {GIVING file-name-3                                      }
```

File-name-1 is the name given in the sort-file-description entry that describes the records to be sorted.

ASCENDING and DESCENDING: The ASCENDING and DESCENDING options specify whether records are to be sorted into an ascending or descending sequence, respectively, based on one or more sort keys.

Each data-name represents a "key" data item and must be described in the records associated with the sort-file-name.

At least one ASCENDING or DESCENDING clause must be specified. Both options may be specified in the same SORT statement, in which case, records are sorted on data-name-1, in ascending or descending order; and then within data-name-1, they are sorted on the KEY data item represented by data-name-2, in ascending or descending order, etc.

Keys are always listed from left to right in order of decreasing significance, regardless of whether they are ascending or descending.

The direction of the sort depends on the use of the ASCENDING or DESCENDING clauses as follows:

1. When an ASCENDING clause is used, the sorted sequence is from the lowest value of the key to the highest value, according to the collating sequence for the COBOL character set.

2. When a DESCENDING clause is used, the sorted sequence is from the highest value of the key to the lowest value, according to the collating sequence of the COBOL character set.

Sort keys must be one of the types of data item listed in Figure 12. Corresponding to each type of data item is a collating sequence that is used with it for sorting.

A character in the EBCDIC collating sequence (used with alphabetic, alphanumeric, etc., data items) is interpreted as not being signed. For fixed-point and internal floating-point numeric data items characters are collated algebraically (that is, as being signed).

Type of Data Item Used for Sort Key	Collating Sequence
Alphabetic	EBCDIC
Alphanumeric	EBCDIC
Numeric Edited	EBCDIC
Group	EBCDIC
External Decimal	Zoned Decimal
Binary	Fixed-point
Internal Decimal	Fixed-point
Internal Floating-point	Floating-point
External Floating-point	EBCDIC

Figure 12. Collating Sequence Used for Sort Keys

The EBCDIC collating sequence for COBOL characters in ascending order is:

1.		(space)
2.	.	(period or decimal point)
3.	<	(less than)
4.	(	(left parenthesis)
5.	+	(plus symbol)
6.	$	(currency symbol)
7.	*	(asterisk)
8.	)	(right parenthesis)
9.	;	(semicolon)
10.	-	(hyphen or minus symbol)
11.	/	(stroke, virgule, slash)
12.	,	(comma)
13.	>	(greater than)
14.	'	(apostrophe or single quotation mark)
15.	=	(equal sign)
16.	"	(quotation mark)
17-42.	A through Z	
43-52.	0 through 9	

(The complete EBCDIC collating sequence is given in IBM System/360 Reference Data, Form X20-1703.)

The record description for every record that is listed in the DATA RECORDS clause of an SD description must contain the "key" items data-name-1, data-name-2, etc. These "key" items are subject to the following rules:

1. Keys must be physically located in the same position and have the same data format in every logical record of the sort-file. If there are multiple record descriptions in an SD, it is sufficient to describe a key in only one of the record descriptions.

2. Key items must not contain an OCCURS clause nor be subordinate to entries that contain an OCCURS clause.

3. A maximum of 12 keys may be specified. The total length of all the keys must not exceed 256 bytes.

4. All keys must be at a fixed displacement from the beginning of a record; that is, they cannot be located after a variable table in a record.

5. All key fields must be located within the first 4092 bytes of a logical record.

6. The data-names describing the keys may be qualified.

SECTION-NAME-1 AND SECTION-NAME-2: Section-name-1 is the name of an input procedure; section-name-2 is the name of the last section that contains the input procedure in the COBOL main program. Section-name-2 is required if the procedure terminates in a section other than that in which it is started.

INPUT PROCEDURE: The presence of the INPUT PROCEDURE option indicates that the programmer has written an input procedure to process records before they are sorted and has included the procedure in the Procedure Division as one or more distinct sections.

The input procedure must consist of one or more sections that are written consecutively and do not form a part of any output procedure. The input procedure must include at least one RELEASE statement in order to transfer records to the sort-file.

Control must not be passed to the input procedure except when a related SORT statement is being executed, because the RELEASE statement in the input procedure has no meaning unless it is controlled by a SORT statement. The input procedure can include any procedures needed to select, create, or modify records. There are three restrictions on the procedural statements within an input procedure:

1. The input procedure must not contain any SORT statements.

2. The input procedure must not contain any transfers of control to points outside the input procedure. The execution of a CALL statement to another program that follows standard linkage conventions, or the execution of USE declaratives for label handling and error processing are not considered transfers of control outside of an input procedure. Hence, they are allowed to be activated within these procedures.

   However, this compiler permits the ALTER, GO TO, and PERFORM statements in the input procedure to refer to procedure-names outside the input procedure. However, it is the user's responsibility to ensure a return to the input procedure after exiting through a GO TO or PERFORM statement.

3. The remainder of the Procedure Division must not contain any transfers of control to points inside the input procedure (with the exception of the return of control from a Declaratives Section).

However, this compiler allows ALTER, GO TO and PERFORM statements in the remainder of the Procedure Division to refer to procedure-names within the input procedure. If a SORT statement is active when the transfer of control is made, then all such transfers are valid. If a SORT statement is not active, however, then the user must ensure that such a transfer of control does not cause:

a. A RELEASE statement to be executed

b. Control to reach the end of the input procedure

If an input procedure is specified, control is passed to the input procedure when the SORT program input phase is ready to receive the first record. The compiler inserts a return mechanism at the end of the last section of the input procedure and when control passes the last statement in the input procedure, the records that have been released to file-name-1 are sorted. The RELEASE statement transfers records from the Input Procedure to the input phase of the sort operation (see "RELEASE Statement").

USING: If the USING option is specified, all the records in file-name-2 are transferred automatically to file-name-1. At the time of execution of the SORT statement, file-name-2 must not be open. File-name-2 must be a standard sequential file. For the USING option, the compiler will open, read, release, and close file-name-2 without the programmer specifying these functions. If the user specifies error handling and/or label processing declaratives for file-name-2, the compiler will make the necessary linkage to the appropriate Declaratives Section.

SECTION-NAME-3 AND SECTION-NAME-4: Section-name-3 represents the name of an output procedure; section-name-4 is the name of the last section that contains the output procedure in the COBOL main program. Section-name-4 is required if the procedure terminates in a section other than that in which it is started.

OUTPUT PROCEDURE: The output procedure must consist of one or more sections that are written consecutively and do not form a part of any input procedure. The output procedure must include at least one RETURN statement in order to make sorted records available for processing.

Control must not be passed to the output procedure except when a related SORT statement is being executed, because RETURN statements in the output procedure have no meaning unless they are controlled by a SORT statement. The output procedure may consist of any procedures needed to select, modify, or copy the records that are being returned one at a time, in sorted order, from the sort-file. There are three restrictions on the procedural statements within the output procedure.

1. The output procedure must not contain any SORT statements.

2. The output procedure must not contain any transfers of control to points outside the output procedure. The execution of a CALL statement to another program that follows standard linkage conventions, or the execution of USE declaratives for label handling or error processing are not considered transfers of control outside of an output procedure. Hence, they are allowed to be activated within these procedures.

   However, this compiler permits the ALTER, GO TO, and PERFORM statements in the output procedure to refer to procedure-names outside the output procedure. However, it is the user's responsibility to ensure a return to the output procedure after exiting through a GO TO or PERFORM statement.

3. The remainder of the Procedure Division must not contain any transfers of control to points inside the output procedure (with

the exception of the return of control from a Declaratives Section).

However, this compiler allows ALTER, GO TO and PERFORM statements in the remainder of the Procedure Division to refer to procedure-names within the output procedure. If a SORT statement is active when the transfer of control is made, then all such transfers are valid. If a SORT statement is not active, however, then the user must ensure that such a transfer of control does not cause:

a. A RETURN statement to be executed

b. Control to reach the end of the output procedure

If an output procedure is specified, control passes to it after file-name-1 has been placed in sequence by the SORT statement. The compiler inserts a return mechanism at the end of the last section in the output procedure. When control passes the last statement in the output procedure, the return mechanism provides for termination of the SORT and then passes control to the next statement after the SORT statement.

When all the records are sorted, control is passed to the output procedure. The RETURN statement in the output procedure is a request for the next record (see "RETURN Statement").

GIVING: If the GIVING option is used, all sorted records in file-name-1 are automatically transferred to file-name-3. At the time of execution of the SORT statement, file-name-3 must not be open. File-name-3 must name a standard sequential file. For the GIVING option, the compiler will open, return, write, and close file-name-3 without the programmer specifying these functions. If the user specifies error handling and/or label processing declaratives for file-name-3, the compiler will make the necessary linkage to the appropriate Declaratives Section.

CONTROL OF INPUT OR OUTPUT PROCEDURES: The INPUT or OUTPUT PROCEDURE options function in a manner similar to Option 1 of the PERFORM statement; for example, naming a section in an INPUT PROCEDURE clause causes execution of that section during the sorting operation to proceed as though that section had been the subject of a PERFORM statement. As with a PERFORM, the execution of the section is terminated after execution of its last statement. The procedures may be terminated by using an EXIT statement (see "EXIT Statement").

RELEASE Statement

The RELEASE statement transfers records from the input procedure to the input phase of the sort operation.

Format
RELEASE sort-record-name [FROM identifier]

A RELEASE statement may be used only within the range of an input procedure associated with a SORT statement.

If the INPUT PROCEDURE option is specified, the RELEASE statement must be included within the given set of procedures.

Sort-record-name must be the name of a logical record in the associated sort-file description.

When the FROM identifier option is used, it makes the RELEASE statement equivalent to the statement MOVE identifier TO sort-record-name, followed by the RELEASE statement.

Sort-record-name and identifier must not refer to the same storage area. A MOVE with the rules for group items is effected from identifier, using the length of the record-name associated with the SD entry.

After the RELEASE statement is executed, the logical record is no longer available. When control passes from the input procedure, the file consists of those records that were placed in it by the execution of the RELEASE statement.

RETURN Statement

The RETURN statement obtains individual records in sorted order from the final phase of the sort program.

Format
RETURN sort-file-name RECORD [INTO identifier] AT END imperative-statement

Sort-file-name is the name given in the sort-file-description entry that describes the records to be sorted.

All references to records retrieved by a RETURN statement must be in terms of the record description(s) associated with the SD entry, unless the INTO option is specified. The retrieved record may, optionally, be moved to the user's own area and be referenced as appropriate.

A RETURN statement may only be used within the range of an output procedure associated with a SORT statement for file-name-1.

The identifier must be the name of a working-storage area or an output record area. Use of the INTO option has the same effect as the MOVE statement for alphanumeric items.

The imperative-statement in the AT END phrase specifies the action to be taken when all the sorted records have been obtained from the sorting operation. After execution of the imperative-statement in the AT END phrase, no RETURN statements may be executed within the current output procedure.

EXIT Statement

The EXIT statement may be used as a common end point for input or output procedures executed as with programs executed through a PERFORM statement.

Format
paragraph-name. EXIT.

When used in this manner, the EXIT statement must appear as the only statement in the last paragraph of the input or output procedure.

SPECIAL REGISTERS FOR SORT

Four special registers are available to users of the Sort Feature. These registers provide a means of object time communication between the user and the Sort Feature, and they aid in the optimization of Sort performance.

Program Product Information (Version 3 and Version 4)

When the Version 3 or Version 4 Compiler is used in conjunction with the Program Product OS Sort/Merge (Program No. 5734-SM1), a fifth special register, SORT-MESSAGE, is available, as well as improved functions for the SORT-CORE-SIZE and SORT-RETURN special registers. When either the Version 3 or the Version 4 Compiler is used in conjunction with the Type 1 OS Sort/Merge (Program No. 360S-SM-023), these new features are not available, and a warning message is issued by the sort during execution (the sort executes properly).

Further information about the Sort Special Registers can be found in "Using the SORT Feature" in the Program Product Programmer's Guide.

The first three registers may have control information transferred to them at object time if the user specifies them as the receiving fields of statements such as MOVE. However, none of the sort special registers can be used as operands in ACCEPT, DISPLAY, or EXHIBIT statements. The information must be passed before the SORT statement is executed. The registers are initialized to zero by the compiler, but are not reset after a Sort procedure is executed. Thus, if a program has multiple SORT statements, any values in the registers at the completion of one sorting operation will be in the registers at the beginning of another, unless modified.

1. SORT-FILE-SIZE is the name of a binary data item whose PICTURE is S9(8). It is used for the estimated number of records in the file to be sorted. If SORT-FILE-SIZE is omitted, the Sort Feature assumes that the file contains the maximum number of records that can be processed with the available core size and number of work units. If the estimate exceeds the maximum, the estimate will be ignored.

2. SORT-CORE-SIZE is the name of a binary data item whose PICTURE is S9(8). It is used to specify the number of bytes of storage available to the sorting operation if it is different from the core size that the Sort Feature would normally use.

Program Product Information (Version 3 and Version 4)

By placing one of the following values in SORT-CORE-SIZE, the programmer can take advantage of the "maximum" main storage parameter supported by the OS Sort/Merge Program Product:

a. When +999999 is coded, COBOL specifies that Sort should use all available main storage, reserving 6K bytes of main storage for use by the data management routines.

b. When a negative number is specified, COBOL uses the absolute value of the number as the number of bytes of main storage to be reserved for data management routines and buffers. The value is only reserved if the OS Sort/Merge Program Product was installed with the "maximum" main storage parameter.

3. SORT-MODE-SIZE is the name of a binary data item whose PICTURE is S9(5). It is used for variable-length records. If the length of most records in the file is significantly different from the average record length, performance is improved by specifying the most frequently occurring record length. If SORT-MODE-SIZE is omitted, the average of the maximum and minimum record lengths is assumed. For example, if records vary in length from 20 to 100 bytes, but most records are 30 bytes long, the value 30 should be moved to SORT-MODE-SIZE. The maximum record length handled by the Sort Feature is 32,000 bytes.

4. SORT-RETURN is the name of a binary data item whose PICTURE is S9(4). It contains a return code of 0 or 16 at the completion of a sorting operation to signify the success or failure, respectively, of the sort operation. If the sort operation terminates abnormally and there is no reference to this special register anywhere in the program, a message is displayed on the console. The operator then may continue or cancel the job.

Program Product Information (Version 3 and Version 4)

The SORT-RETURN special register can also be used to terminate the OS Sort/Merge Program Product sorting operation. The programmer can place the value 16 in this special register at any point during an Input or Output Procedure to terminate the Sort immediately after execution of the next RELEASE or RETURN statement.

5. SORT-MESSAGE is the name of an alphanumeric data item whose PICTURE is X(8). If the OS Sort/Merge Program Product has been installed so that any of its messages are routed to the printer, then the programmer can place in SORT-MESSAGE the DDname which the Sort is to use in place of SYSOUT for its messages. For example, when the statement MOVE "SORTDDNM" TO SORT-MESSAGE is executed before the Sort is initiated, then the Sort writes its printer messages to SORTDDNM rather than to SYSOUT. If SORT-MESSAGE is not modified during the program, the default value is decided when the OS Sort/Merge Program Product is installed.

An example of how variable information can be passed to the Sort feature by use of a register is:

```
ACCEPT FILE-SIZE FROM CONSOLE.
MOVE FILE-SIZE TO SORT-FILE-SIZE.
```

SAMPLE PROGRAM USING THE SORT FEATURE

Figure 13 illustrates a sort based on a sales contest. The records to be sorted contain data on salesmen: name and address, employee number, department number, and pre-calculated net sales for the contest period.

The salesman with the highest net sales in each department wins a prize, and smaller prizes are awarded for second highest sales, third highest, etc. The order of the SORT is (1) by department, the lowest numbered first (ASCENDING KEY DEPT); and (2) by net sales within each department, the highest net sales first (DESCENDING KEY NET-SALES).

The records for the employees of departments 7 and 9 are eliminated in an input procedure (SCREEN-DEPT) before sorting begins. The remaining records are then sorted, and the output is placed on another file for use in a later job step.

```
000005 IDENTIFICATION DIVISION.
000010 PROGRAM-ID. CONTEST.
000015 ENVIRONMENT DIVISION.
000016 CONFIGURATION SECTION.
000017 SOURCE-COMPUTER.  IBM-360-H50.
000018 OBJECT-COMPUTER.  IBM-360-H50.
000019 SPECIAL-NAMES. SYSOUT IS PRINTER.
000020 INPUT-OUTPUT SECTION.
000025 FILE-CONTROL.
000030     SELECT NET-FILE-IN ASSIGN TO UT-2400-S-INFILE.
000035     SELECT NET-FILE-OUT ASSIGN TO UT-2400-S-SORTOUT.
000040     SELECT NET-FILE ASSIGN TO UT-2400-S-NETFILE.
000050 DATA DIVISION.
000055 FILE SECTION.
000060 SD  NET-FILE
000065     DATA RECORD IS SALES-RECORD.
000070 01  SALES-RECORD.
000075     05  EMPL-NO          PICTURE 9(6).
000080     05  DEPT             PICTURE 9(2).
000085     05  NET-SALES        PICTURE 9(7)V99.
000090     05  NAME-ADDR        PICTURE X(55).
000095 FD  NET-FILE-IN
00096      LABEL RECORDS ARE OMITTED
000100     DATA RECORD IS NET-CARD-IN.
000105 01  NET-CARD-IN.
000110     05  EMPL-NO-IN       PICTURE 9(6).
000115     05  DEPT-IN          PICTURE 9(2).
000120     05  NET-SALES-IN     PICTURE 9(7)V99.
000125     05  NAME-ADDR-IN     PICTURE X(55).
000130 FD  NET-FILE-OUT
000131     LABEL RECORDS ARE OMITTED
000135     DATA RECORD IS NET-CARD-OUT.
000140 01  NET-CARD-OUT.
000145     05  EMPL-NO-OUT      PICTURE 9(6).
000150     05  DEPT-OUT         PICTURE 9(2).
000155     05  NET-SALES-OUT    PICTURE 9(7)V99.
000160     05  NAME-ADDR-OUT    PICTURE X(55).
```

Figure 13. Sample Program Using the Sort Feature (Part 1 of 2)

```
000165 PROCEDURE DIVISION.
000170 ELIM-DEPT-7-9-NO-PRINTOUT.
000175     SORT NET-FILE
000180         ASCENDING KEY DEPT
000185         DESCENDING KEY NET-SALES
000190         INPUT PROCEDURE SCREEN-DEPT
000195         GIVING NET-FILE-OUT.
000200 CHECK-RESULTS SECTION.
000205 C-R-1.
000210     OPEN INPUT NET-FILE-OUT.
000215 C-R-2.
000220     READ NET-FILE-OUT AT END GO TO C-R-FINAL.
000225     DISPLAY EMPL-NO-OUT DEPT-OUT NET-SALES-OUT
000230         NAME-ADDR-OUT UPON PRINTER.
000235 C-R-3.
000240     GO TO C-R-2.
000245 C-R-FINAL.
000250     CLOSE NET-FILE-OUT.
000255     STOP RUN.
000260 SCREEN-DEPT SECTION.
000265 S-D-1.
000270     OPEN INPUT NET-FILE-IN.
000275 S-D-2.
000280     READ NET-FILE-IN AT END GO TO S-D-FINAL.
000285     DISPLAY EMPL-NO-IN DEPT-IN NET-SALES-IN
000290         NAME-ADDR-IN UPON PRINTER.
000295 S-D-3.
000300     IF DEPT-IN = 7 OR 9 GO TO S-D-2
000305         ELSE
000310             MOVE NET-CARD-IN TO SALES-RECORD
000315             RELEASE SALES-RECORD
000320             GO TO S-D-2.
000325 S-D-FINAL.
000330     CLOSE NET-FILE-IN.
000335 S-D-END.
000340     EXIT.
```

Figure 13. Sample Program Using the Sort Feature (Part 2 of 2)

REPORT WRITER FEATURE

The Report Writer Feature permits the programmer to specify the format of a printed report in the Data Division, thereby minimizing the amount of Procedure Division coding he would have to write to create the report.

A printed report consists of the information reported and the format in which it is printed. Several reports can be produced by one program.

In the Data Division, the programmer gives the name(s) and describes the format(s) of the report(s) he wishes produced. In the Procedure Division, he writes the statements that produce the report(s).

At program execution time, the report in the format defined is produced -- data to be accumulated is summed, totals are produced, counters are stepped and reset, and each line and each page is printed. Thus, the programmer need not concern himself with the details of these operations.

DATA DIVISION -- OVERALL DESCRIPTION

In the Data Division, the programmer must write an FD entry that names the output file upon which the report is to be written, and must also name the report itself. A report may be written on two files at the same time.

At the end of the Data Division, he must add a Report Section to define the format of each report named. In the Report Section, there are two types of entries:

1. The Report Description Entry (RD) which describes the physical aspects of the report format.

2. The report group description entries which describe the data items within the report and their relation to the report format.

In the report description entry, the programmer specifies the maximum number of lines per page, where report groups are to appear on the page, and which data items are to be considered as controls.

Controls govern the basic format of the report. When a control changes value -- that is, when a control break occurs -- special actions will be taken before the next line of the report is printed. Controls are listed in a hierarchy, proceeding from the most inclusive down to the least inclusive. Thus, by specifying HEADING and FOOTING controls, the programmer is able to instruct the Report Writer to produce the report in whatever format he desires.

For example, in the program at the end of this chapter, the hierarchy of controls proceeds from the highest (FINAL) to an intermediate control (MONTH) to the minor control (DAY). DAY is the minor control since if MONTH changes, DAY also must change. Whenever any control changes, special actions are performed by the Report Writer -- sum information is totaled, counters are reset, special information is printed, and so forth -- before the next detail line is printed.

The report group description entries describe the characteristics of all data items contained within the report group: the format of each data item present, its placement in relation to the other data items within the report group, and any control factors associated with the

group. Information to be presented within a report group can be described in three ways:

- As SOURCE information, which is information from outside the report.
- As SUM information, which is the result of addition operations upon any data present, whether SOURCE information or other SUM information.
- As VALUE information, which is constant information.

Through the RD and the report group description entries, the programmer has thus defined completely the content, the format, and the summing operations necessary to produce the desired report.

PROCEDURE DIVISION -- OVERALL DESCRIPTION

In the Procedure Division, the programmer instructs the Report Writer to produce the report through the use of three Report Writer statements: INITIATE, GENERATE, and TERMINATE.

The INITIATE statement performs functions in the Report Writer analogous to the OPEN statement for individual files.

The GENERATE statement automatically produces the body of the report. Necessary headings and footings are printed, counters are incremented and reset as desired, source information is obtained, and sum information is produced, data is moved to the data item(s) in the report group description entry, controls are tested, and when a control break occurs, the additional lines requested are printed, as well as the detail line that caused the control break. All of this is done automatically, thus relieving the programmer of the responsibility for writing detailed tests and looping procedures that would otherwise be necessary.

The TERMINATE statement completes the processing of a report. It is analogous to the CLOSE statement for individual files.

In the Declaratives portion of the Procedure Division, the programmer may also specify a USE BEFORE REPORTING procedure for report group. In this procedure, he is able to specify any additional processing he wishes done before a specific report group is printed.

Two special registers are used by the Report Writer feature:

LINE-COUNTER -- which is a numeric counter used by the Report Writer to determine when a PAGE HEADING and/or a PAGE FOOTING report group is to be presented. The maximum value of LINE-COUNTER is based on the number of lines per page as specified in the PAGE LIMIT(S) clause. LINE-COUNTER may be referred to in any Procedure Division statement.

PAGE-COUNTER -- which is a numeric counter that may be used as a SOURCE data item in order to present the page number on a report line. The maximum size of PAGE-COUNTER is based on the size specified in the PICTURE clause associated with an elementary item whose SOURCE IS PAGE-COUNTER. This counter may be referred to by any Procedure Division statement.

Figure 15, at the end of this chapter, gives an example of a Report Writer program for a manufacturer's quarterly report.

Figure 16, which follows the program, shows the report that would be produced.

DATA DIVISION CONSIDERATIONS FOR REPORT WRITER

The names of all the reports to be produced must be named in the File Section of the Data Division. An entry is required in the FD entry to list the names of the reports to be produced on that file. A Report Section must be added at the end of the Data Division to define the format of each report.

FILE DESCRIPTION

The File Description furnishes information concerning the physical structure, identification, and record-names pertaining to a given file.

General Format

```
FD file-name

     [BLOCK CONTAINS Clause]
     [RECORD CONTAINS Clause]
     [RECORDING MODE Clause]
     LABEL RECORDS Clause
     [VALUE OF Clause]
     [DATA RECORDS Clause]
     REPORT Clause.
```

A discussion of all the above-mentioned clauses appears in "Data Division." A description of the REPORT clause, the RECORDING MODE clause, the DATA RECORDS clause, and the RECORD CONTAINS clause for a file on which a report is produced follows.

REPORT Clause

Each unique report-name must appear in the REPORT clause of the FD entry (or entries) for the file(s) on which the report(s) is to be produced. The REPORT clause cross references the description of report description entries with their associated file description entry.

Format

```
{ REPORT IS  }
{ REPORTS ARE } report-name-1 [report-name-2]...
```

Each file description entry for standard sequential OUTPUT files within the File Section may include a REPORT clause containing the names

of one or more reports. These reports may be of different sizes, formats, etc., and the order in which their names appear in the clause is not significant.

Each unique report-name listed in an FD entry must be the subject of an RD entry in the Report Section. A given report-name may appear in a maximum of two REPORT clauses.

RECORDING MODE Clause

The RECORDING MODE clause is used to specify the format of the logical records within the file. If this clause is omitted, the recording mode is determined as described in "Data Division."

DATA RECORDS Clause

If the DATA RECORDS clause is specified, and the file is used for output, the AFTER ADVANCING option must be used when writing records named in this clause.

RECORD CONTAINS Clause

The RECORD CONTAINS clause enables the user to specify the maximum size of his report record.

Format
RECORD CONTAINS [integer-1 TO] integer-2 CHARACTERS

The specified size of each report record must include the carriage control/line spacing character, and the CODE character, if the CODE option is used. If the RECORD CONTAINS clause is omitted, the compiler assumes a default size of 133 characters.

For variable-length records, the size of each print line will be integer-2 characters, and the size of each blank line required for spacing will be 17 characters. For fixed-length records, the size of each print line and each blank line required for spacing will be integer-2 characters.

For further information on the RECORD CONTAINS clause, see "Data Division."

REPORT SECTION

The Report Section consists of two types of entries for each report; one describes the physical aspects of the report format, the other type describes conceptual characteristics of the items that make up the report and their relationship to the report format. These are:

1. Report Description entry (RD)
2. Report group description entries

The Report Section must begin with the header REPORT SECTION.

Report Description Entry

The Report Description entry contains information pertaining to the overall format of a report named in the File Section and is uniquely identified by the level indicator RD. The clauses that follow the name of the report are optional, and their order of appearance is not significant.

The entries in this section stipulate:

1. The maximum number of lines that can appear on a page.
2. Where report groups are to appear on a page.
3. Data items that act as control factors during presentation of the report.

General Format

```
REPORT SECTION.

RD report-name
   [CODE Clause]
   [CONTROL Clause]
   [PAGE LIMIT Clause].
```

RD is the level indicator.

Report-name is the name of the report and must be unique. The report-name must be specified in a REPORT clause in the file description entry for the file on which the report is to be written.

CODE Clause

The CODE clause is used to specify an identifying character added at the beginning of each line produced. The identification is meaningful when more than one report is written on a file.

```
Format
WITH CODE mnemonic-name
```

Mnemonic-name must be associated with a single character literal used as function-name-1 in the SPECIAL-NAMES paragraph in the Environment Division. The identifying character is appended to the beginning of the line, preceding the carriage control/line spacing character. This clause should not be specified if the report is to be printed on-line.

CONTROL Clause

The CONTROL clause indicates the identifiers that specify the control hierarchy for this report, that is, the control breaks.

```
Format
{CONTROL IS  } {FINAL                                  }
{CONTROLS ARE} {identifier-1 [identifier-2]...         }
               {FINAL  identifier-1 [identifier-2]...  }
```

A control is a data item that is tested each time a detail report group is generated. If the test indicates that the value of the data item (i.e., CONTROL) has changed, a control break is said to occur, and special action (described below) is taken before printing the detail line.

FINAL is the highest level control. (It is the one exception to the statement that controls are data items.) The identifiers specify the control hierarchy of the other controls. Identifier-1 is the major control, identifier-2 is the intermediate control, etc. The last identifier specified is the minor control. The levels of the controls are indicated by the order in which they are written. FINAL need not be specified in the CONTROL clause, even if a CONTROL HEADING FINAL or CONTROL FOOTING FINAL is specified.

The control identifiers must each specify different data items; that is, their descriptions must not be such that they occupy (partially or completely) the same area of storage, or overlap in any way.

When controls are tested, the highest level control specified is tested first, then the second highest level, etc. When a control break is found for a particular level, a control break is implied for each lower level as well. A control break for FINAL occurs only at the beginning and ending of a report (i.e., before the first detail line is printed and after the last detail is printed).

The action to be taken as a result of a control break depends on what the programmer defines. He may define a CONTROL HEADING report group and/or a CONTROL FOOTING group or neither for each control.

The control footings and headings that are defined are printed prior to printing the originally referenced detail. They are printed in the following order: lowest level control footing, next higher level control footing, etc., up to and including the control footing for the level at which the control break occurred; then the control heading for that level, then the next lower level control heading, etc., down to and including the minor control heading; then the detail is printed. If, in the course of printing control headings and footings, a page condition is detected, the current page is ejected and a new page begun. If the associated report groups are given, a page footing and/or a page heading are also printed.

The CONTROL clause is required when CONTROL HEADING or CONTROL FOOTING report groups other than FINAL are specified.

The identifiers specified in the CONTROL clause are the only identifiers referred to by the RESET and TYPE clauses in a report group description entry for this report. The identifiers must be defined in the File or Working-Storage Section of the Data Division.

Program Product Information (Version 4)

For Version 4, the identifiers in the CONTROL clause may also be defined in the Communication Section.

PAGE LIMIT Clause

The PAGE LIMIT clause indicates the specific line control to be maintained within the logical presentation of a page, i.e., it describes the physical format of a page of the report.

Format

```
PAGE [LIMIT IS ]  integer-1  {LINE }
     [LIMITS ARE]            {LINES}

    [HEADING integer-2]
    [FIRST DETAIL integer-3]
    [LAST DETAIL integer-4]
    [FOOTING integer-5]
```

If this clause is not specified, PAGE-COUNTER and LINE-COUNTER special registers are not generated.

The PAGE LIMIT clause is required when page format must be controlled by the Report Writer.

integer-1: The PAGE LIMIT integer-1 LINES clause is required to specify the depth of the report page; the depth of the report page may or may not be equal to the physical perforated continuous form often associated in a report with the page length. The size of the fixed data-name, LINE-COUNTER, is the maximum numeric size based on integer-1 lines required for the counter to prevent overflow.

integer-2: The first line number of the first heading print group is specified by integer-2. No print group will start preceding integer-2, i.e., integer-2 is the first line on which anything may be printed.

integer-3: The first line number of the first normal print group (body group) is specified by integer-3. No DETAIL, CONTROL HEADING, or CONTROL FOOTING print group will start before integer-3.

integer-4: The last line number of the last nonfooting body group is specified by integer-4. No DETAIL or CONTROL HEADING print group will extend beyond integer-4.

integer-5: The last line number of the last CONTROL FOOTING print group is specified by integer-5. No CONTROL FOOTING print group will extend beyond integer-5. PAGE FOOTING print groups will follow integer-5.

Using the parameters of the PAGE LIMIT clause, the Report Writer establishes the areas of the page where each type of report group is allowed to be printed. The following are the page areas for each type of report group:

1. A REPORT HEADING report group can extend from line integer-2 to line integer-1, inclusive. If the REPORT HEADING report group is not on a page by itself, the FIRST DETAIL integer-3 clause must be present in the PAGE LIMIT clause of the report.

2. A PAGE HEADING report group may extend from line integer-2 to line integer-3 minus 1, inclusive. If a PAGE HEADING report group is specified in the report description, the FIRST DETAIL integer-3 clause must be present in the PAGE LIMIT clause of the report. A PAGE HEADING report group that follows a REPORT HEADING report group on the same page must be able to be printed in the area of the page defined in this rule.

3. CONTROL HEADING report groups and DETAIL report groups must be printed in the area of the page that extends from line integer-3 to line integer-4, inclusive.

4. CONTROL FOOTING report groups must be printed in the area of the page extending from line integer-3 to line integer-5, inclusive.

5. A PAGE FOOTING report group may extend from line integer-5 plus 1 to line integer-1, inclusive. If PAGE FOOTING is specified in the report description, either the FOOTING integer-5 or LAST DETAIL integer-4 clause must be present in the PAGE LIMIT clause of the report.

6. A REPORT FOOTING report group can extend from line integer-2 to line integer-1, inclusive. If the REPORT FOOTING report group is not on a page by itself, either the FOOTING integer-5 or LAST DETAIL integer-4 clause must be present in the PAGE LIMIT clause of the report.

Figure 14 pictorially represents page format report group control when the PAGE LIMIT clause is specified.

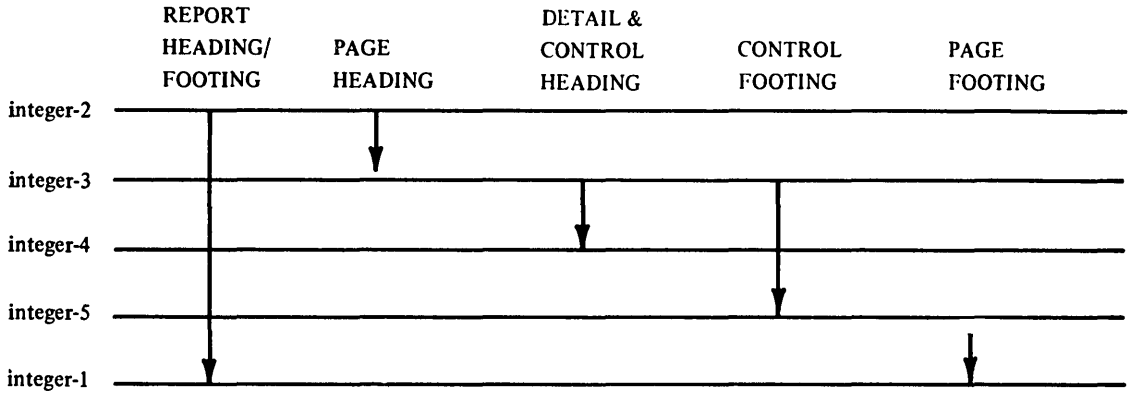


Figure 14. Page Format When the PAGE LIMIT Clause is Specified

The PAGE LIMIT clause may be omitted when no association is desired between report groups and the physical format of an output page. In this case, relative line spacing must be indicated for all report groups of the report.

If absolute line spacing is indicated for all the report groups, none of the integer-2 through integer-5 controls need be specified. If any of these limits are specified for a report that has only absolute line spacing, the limits are ignored.

If relative line spacing is indicated for any report group, all LINE NUMBER and NEXT GROUP spacing must be consistent with the control specified or implied in the PAGE LIMIT clause.

If PAGE LIMITS integer-1 is specified and some or all of the HEADING integer-2, FIRST DETAIL integer-3, LAST DETAIL integer-4, FOOTING integer-5 clauses are omitted, the following implicit control is assumed for all omitted specifications:

1. If HEADING integer-2 is omitted, integer-2 is considered to be equivalent to the value 1, that is, LINE NUMBER one.

2. If FIRST DETAIL integer-3 is omitted, integer-3 is considered to be equivalent to the value of integer-2.

3. If LAST DETAIL integer-4 is omitted, integer-4 is considered to be equivalent to the value of integer-5.

4. If FOOTING integer-5 is omitted, integer-5 is considered to be equivalent to the value of integer-4. If both LAST DETAIL integer-4 and FOOTING integer-5 are omitted, integer-4 and integer-5 are both considered to be equivalent to the value of integer-1.

Only one PAGE-LIMIT clause may be specified for a Report Description entry.

- Integer-1 through integer-5 must be positive integers.
- Integer-2 through integer-5 must be in ascending order. Integer-5 must not exceed integer-1.

Report Group Description Entry

A report comprises one or more report groups. Each report group is described by a hierarchy of entries similar to the description of a data record. There are three categories of report groups: heading groups, detail groups, and footing groups. A CONTROL HEADING, DETAIL, or CONTROL FOOTING report group may also be referred to as a body group.

The report group description entry defines the format and characteristics for a report group. The relative placement of a particular report group within the hierarchy of report groups, the format of all items, and any control factors associated with the group are defined in this entry.

Schematically, a report group is a line, a series of lines, or a null (i.e., nonprintable) group. A report group is considered to be one unit of the report. Therefore, the lines of a report group are printed as a unit.

A null group is a report group for which no LINE or COLUMN clauses have been specified (that is, a nonprintable report group).

The report group description entry defines the format and characteristics applicable to the type of report group.

1. For all report groups that are not null groups, the description entry indicates where and when the report group is to be presented.
2. For all report groups, the description entry indicates when the nonprinting functions of the report group, such as summation, are to be performed.
3. For all report groups except DETAIL, the description entry allows for the execution of a user-specified procedure prior to printing a report group. If a report group is a null group, the execution of the user procedure occurs in the same manner as though the report group were printed.
4. For CONTROL FOOTING report groups, the description entry indicates the user's summation algorithm.

Report group names are required when reference is made in the Procedure Division to:

- A DETAIL report group by a GENERATE statement
- A HEADING or FOOTING report group by a USE sentence

Report group names are required when reference is made in the Report Section to a DETAIL report group by a SUM UPON clause.

Except for the data-name clause which, when present, must immediately follow the level-number, the clauses may be written in any order.

General Format 1

```
01    [data-name-1]
      [LINE Clause]
      [NEXT GROUP Clause]
      TYPE Clause
      [USAGE Clause].
```

General Format 2

```
level number [data-name-1]
      [LINE clause]
      [USAGE clause].
```

General Format 3

```
level-number [data-name-1]
      [COLUMN Clause]
      [GROUP Clause]
      [JUSTIFIED Clause]
      [LINE Clause]
      [PICTURE Clause]
      [RESET Clause]
      [BLANK WHEN ZERO Clause]
      [{SOURCE}        ]
      [{SUM   } Clause ]
      [{VALUE }        ]
      [USAGE Clause].
```

General Format 4

```
01   [data-name-1]
      [BLANK WHEN ZERO Clause]
      [COLUMN Clause]
      [GROUP Clause]
      [JUSTIFIED Clause]
      [LINE Clause]
      [NEXT GROUP Clause]
      PICTURE Clause
      [RESET Clause]
      {SOURCE}
      {SUM   } Clause
      {VALUE }
      TYPE Clause
      [USAGE Clause].
```

Format 1 is used to indicate a report group. A report group description must contain a report group entry (level-01) and it must be the first entry. A report group extends from this entry either to the next report group level-01 entry or to the beginning of the next report description. A null report group may contain only a Format 1 report group entry.

Format 2 is used to indicate a group item. A group item entry may contain a level number from 02 through 48; this entry has the following functions:

- If a report group has more than one line and one of the lines contains more than one elementary item, a group item entry may be used to indicate the LINE number of the subordinate elementary items.
- If a group item entry contains no LINE clause and there are no SUM counters subordinate to it, its only function is documentation.

Format 3 is used to indicate an elementary item. An elementary item entry may contain a level number from 02 through 49; this entry has the following functions:

- An elementary item entry may be used to describe an item that is to be presented on a printed line. In this case, a COLUMN clause, a PICTURE clause, and either a SOURCE, SUM, or VALUE clause must be present.
- An elementary item entry in a DETAIL report group may be used to indicate to the Report Writer what operands are to be summed upon presentation of the DETAIL report group.
- An elementary item entry in a CONTROL FOOTING report group may be used to define a SUM counter (see the discussion of the SUM clause).

Format 4 is used to indicate a report group that consists of only one elementary item. If Format 4 is used to define the report group instead of Format 1, it must be the only entry in the group.

LINE Clause

The LINE clause indicates the absolute or relative line number of this entry in reference to the page or previous entry.

Format

```
LINE NUMBER IS  { integer-1          }
                { PLUS integer-2     }
                { NEXT PAGE          }
```

Each line of a report must have a LINE clause associated with it. For the first line of a report group, the LINE clause must be given either at the report group level or prior to or for the first elementary item in the line. For report lines other than the first in a report group, the LINE clause must be given prior to or for the first elementary item in the line. When a LINE clause is encountered, subsequent entries following the entry with the LINE clause are

implicitly presented on the same line until either another LINE clause or the end of the report group is encountered.

Integer-1 and integer-2 must be positive integers.

LINE NUMBER IS integer-1 is an absolute LINE clause. It indicates the fixed line of the page on which this line is to be printed. LINE-COUNTER is set to the value of integer-1 and is used for printing the items in this and the following entries within the report group until a different value for the LINE-COUNTER is specified.

LINE NUMBER IS PLUS integer-2 is a relative LINE clause. The line is printed relative to the previous line either printed or skipped. LINE-COUNTER is incremented by the value of integer-2 and is used for printing the items in this and the following entries within the report group until a different value for the LINE-COUNTER is specified. Exceptions to this rule are discussed later.

LINE NUMBER IS NEXT PAGE indicates that this report group is to be printed on the next page, not on the current page. This LINE clause may appear only in a report group entry or may be the LINE clause of the first line of the report group.

Within any report group, absolute LINE NUMBER entries must be indicated in ascending order, and an absolute LINE NUMBER cannot be preceded by a relative LINE NUMBER. If the first line of the first body group that is to be printed on a page contains either a relative LINE clause or a LINE NUMBER IS NEXT PAGE clause, the line is printed on line FIRST DETAIL integer-3. However, if the LINE-COUNTER contains a value that is greater than or equal to FIRST DETAIL integer-3, the line is printed on line LINE-COUNTER plus 1. This value of LINE-COUNTER was set by an absolute NEXT GROUP clause in the previously printed body group (see rules for NEXT GROUP).

If the report group entry of a body group contains a LINE NUMBER IS NEXT PAGE clause and the first line contains a relative LINE clause, the first line is printed relative to either FIRST DETAIL integer-3 or LINE-COUNTER, whichever is greater. This value of LINE-COUNTER was set by an absolute NEXT GROUP clause in the previously printed body group.

The following are the rules for the LINE clause by report group type:

1. REPORT HEADING

   - LINE NUMBER IS NEXT PAGE cannot be specified in the report group.
   - The first line of the report group may contain an absolute or relative LINE clause.
   - If the first line contains a relative line clause, it is relative to HEADING integer-2.

2. PAGE HEADING

   - LINE NUMBER IS NEXT PAGE cannot be specified in the report group.
   - The first line may contain either an absolute or relative LINE clause.
   - If the first line contains a relative LINE clause, it is relative to either HEADING integer-2 or the value of LINE-COUNTER, whichever is greater. The value in LINE-COUNTER that is greater than HEADING integer-2 can only result from a REPORT HEADING report group being printed on the same page as the PAGE HEADING report group.

3. CONTROL HEADING, DETAIL, and CONTROL FOOTING

   - LINE NUMBER IS NEXT PAGE may be specified in the report group.
   - The first line of the report group may contain either an absolute or relative LINE clause.

4. PAGE FOOTING

   - LINE NUMBER IS NEXT PAGE cannot be specified in the report group.
   - The first line of the report group may contain an absolute or relative LINE clause.
   - If the first line contains a relative LINE clause, it is relative to FOOTING integer-5.

5. REPORT FOOTING

   - If the report group is to be printed on a page by itself, LINE NUMBER IS NEXT PAGE must be specified.
   - If LINE NUMBER IS NEXT PAGE is the only LINE clause in the report group description, the line will be printed on line HEADING integer-2.
   - If the report group description does not contain a LINE NUMBER IS NEXT PAGE clause, the first line must contain an absolute or relative LINE clause. If it contains a relative LINE clause, the line is relative to either FOOTING integer-5 or the value of LINE-COUNTER, whichever is greater. The value in LINE-COUNTER that is greater than FOOTING integer-5 can only result from the printing of the PAGE FOOTING report group.

NEXT GROUP Clause

The NEXT GROUP clause indicates the spacing condition following the last line of the report group.

Format

```
NEXT GROUP IS  { integer-1      }
               { PLUS integer-2 }
               { NEXT PAGE      }
```

The NEXT GROUP clause can appear only in a report group entry. integer-1 and integer-2 must be positive integers.

The following are the rules for the NEXT GROUP clause by report group type:

1. REPORT HEADING

   - If the report group is to be printed on a page by itself, NEXT GROUP IS NEXT PAGE must be specified in the report group description.

- Integer-1 indicates an absolute line number which sets the LINE-COUNTER to this value after printing the last line of the report group.

- Integer-2 indicates a relative line number which increments the LINE-COUNTER by the integer-2 value after printing the last line of the report group.

- An absolute or relative NEXT GROUP clause must not cause the LINE-COUNTER to be set to a value greater than FIRST DETAIL integer-3 minus 1.

2. PAGE HEADING, PAGE FOOTING, and REPORT FOOTING

   - A NEXT GROUP clause cannot be specified in the report group.

3. CONTROL HEADING, DETAIL, and CONTROL FOOTING

   - If a NEXT GROUP clause implies a page change, the change occurs only when the next body group is to be printed.

   - The NEXT GROUP IS NEXT PAGE clause indicates that no more body groups are to be printed on this page.

   - An absolute or relative NEXT GROUP clause may cause the LINE-COUNTER to be set to a value greater than or equal to FIRST DETAIL integer-3 and less than or equal to FOOTING integer-5. This is an exception to the rule which defines the page area of CONTROL HEADING and DETAIL report groups.

   - If a NEXT GROUP IS integer-1 clause causes a page change, the value of LINE-COUNTER is set to the value of integer-1 before the formatting of the first line of the next body group to be printed. This implies that if the first line of the next body group to be printed contains a relative LINE NUMBER clause, the line will be printed on line LINE-COUNTER plus 1; if the first line contains an absolute LINE NUMBER clause that is less than or equal to integer-1, a page will be printed which contains only PAGE HEADING and FOOTING report groups, and the following page will contain the body group.

   - When the NEXT GROUP clause is specified for a CONTROL FOOTING report group, the NEXT GROUP clause functions are performed only when a control break occurs for the control that is associated with this report group.

If the USE procedure for a report group moves a 1 to PRINT-SWITCH, the NEXT GROUP clause functions are not performed for this report group.

TYPE Clause

The TYPE clause specifies the particular type of report group that is described by this entry and indicates the time at which the report group is to be generated.

Format

```
                  ⎧REPORT HEADING⎫
                  ⎩RH            ⎭
                  ⎧PAGE HEADING  ⎫
                  ⎩PH            ⎭
                  ⎧CONTROL HEADING⎫      ⎧identifier-n⎫
                  ⎩CH             ⎭      ⎩FINAL       ⎭
TYPE IS           ⎧DETAIL        ⎫
                  ⎩DE            ⎭
                  ⎧CONTROL FOOTING⎫      ⎧identifier-n⎫
                  ⎩CF             ⎭      ⎩FINAL       ⎭
                  ⎧PAGE FOOTING  ⎫
                  ⎩PF            ⎭
                  ⎧REPORT FOOTING⎫
                  ⎩RF            ⎭
```

The TYPE clause in a particular report group entry indicates the point in time at which this report group will be generated as output.

If the report group is described as TYPE DETAIL or DE, then a GENERATE statement in the Procedure Division directs the Report Writer to produce this report group. Each DETAIL report group must have a unique data-name at level-01 in a report.

If the report group is described as other than TYPE DETAIL or DE, the generation of this report group is an automatic feature of the Report Writer, as detailed in the following paragraphs.

The REPORT HEADING or RH entry indicates a report group that is produced only once at the beginning of a report during the execution of the first GENERATE statement. There can be only one report group of this type in a report. SOURCE clauses used in REPORT HEADING report groups refer to the values of data items at the time the first GENERATE statement is executed.

The PAGE HEADING or PH entry indicates a report group that is produced at the beginning of each page according to PAGE condition rules as specified below. There can be only one report group of this type in a report.

The CONTROL HEADING or CH entry indicates a report group that is produced at the beginning of a control group for a designated identifier, or, in the case of FINAL, is produced once before the first control group during the execution of the first GENERATE statement. There can be only one report group of this type for each identifier and for the FINAL entry specified in a report. In order to produce any CONTROL HEADING report groups, a control break must occur. SOURCE clauses used in CONTROL HEADING FINAL report groups refer to the values of the items at the time the first GENERATE statement is executed.

The CONTROL FOOTING or CF entry indicates a report group that is produced at the end of a control group for a designated identifier or is produced once at the termination of a report ending a FINAL control group. There can be only one report group of this type for each

identifier and for the FINAL entry specified in a report. In order to produce any CONTROL FOOTING report groups, a control break must occur. SOURCE clauses used in CONTROL FOOTING FINAL report groups refer to the values of the items at the time the TERMINATE statement is executed.

The PAGE FOOTING or PF entry indicates a report group that is produced at the bottom of each page according to PAGE condition rules as specified below. There can be only one report group of this type in a report.

The REPORT FOOTING or RF entry indicates a report group that is produced only at the termination of a report. There can be only one report group of this type in a report. SOURCE clauses used in TYPE REPORT FOOTING report groups refer to the value of items at the time the TERMINATE statement is executed.

Identifier-n, as well as FINAL, must be one of the identifiers described in the CONTROL clause in the report description entry.

A FINAL type control break may be designated only once for CONTROL HEADING or CONTROL FOOTING entries within a particular report description.

Nothing precedes a REPORT HEADING entry and nothing follows a REPORT FOOTING entry within a report.

The HEADING or FOOTING report groups occur in the following Report Writer sequence if all exist for a given report:

```
REPORT HEADING (one occurrence only)
PAGE HEADING
     .
     .
     .
CONTROL HEADING
DETAIL
CONTROL FOOTING
     .
     .
     .
PAGE FOOTING
REPORT FOOTING (one occurrence only)
```

CONTROL HEADING report groups are presented in the following hierarchical arrangement:

```
Final Control Heading (one occurrence only)
Major Control Heading
     .
     .
     .
Minor Control Heading
```

CONTROL FOOTING report groups are presented in the following hierarchical arrangement:

```
Minor Control Footing
     .
     .
     .
Major Control Footing
Final Control Footing (one occurrence only)
```

CONTROL HEADING report groups appear with the current values of any indicated SOURCE data items before the DETAIL report groups of the CONTROL group are produced. CONTROL FOOTING report groups appear with the previous values of any indicated SOURCE data items specified in the CONTROL clause, just after the DETAIL report groups of that CONTROL group have been produced.

The USE procedures specified for a CONTROL FOOTING report group that refer to source data items that are specified in the CONTROL clause affect the previous value of the items. If the CONTROL FOOTING report refers to source data items that are not specified in the CONTROLS clause, the USE procedures affect the current value of the items. These report groups appear whenever a control break occurs. LINE NUMBER determines the absolute or relative position of the CONTROL report groups, exclusive of the other HEADING and FOOTING report groups.

USAGE Clause

DISPLAY is the only option that may be specified for group and elementary items in a Report Group Description entry (see "USAGE Clause").

COLUMN Clause

The COLUMN clause indicates the absolute column number on the printed page of the high-order (leftmost) character of an elementary item.

Format
COLUMN NUMBER IS integer-1

The COLUMN clause indicates that the leftmost character of the elementary item is placed in the position specified by integer. If the COLUMN clause is not specified, the elementary item, though included in the description of the report group, is suppressed when the report group is produced at object time.

Integer-1 must be a positive integer.

The COLUMN number clause is given at the elementary level within a report group even if the elementary level is a single level-01 entry, which alone constitutes the report group.

Within a report group and a particular LINE NUMBER specification, COLUMN number entries need not be indicated from left to right.

GROUP INDICATE Clause

The GROUP INDICATE clause specifies that this elementary item is to be produced only on the first occurrence of the item after any control or page break.

Format
GROUP INDICATE

The GROUP INDICATE clause must be specified only at the elementary item level within a DETAIL report group.

An elementary item is not only group indicated in the first DETAIL report group containing the item after a control break, but is also group indicated in the first DETAIL report group containing the item on a new page, even though a control break did not occur.

JUSTIFIED Clause

The JUSTIFIED clause is applicable in report group description entries as described in "Data Division."

PICTURE Clause

The PICTURE clause is applicable in Report Group Description entries as described in "Data Division."

RESET Clause

The RESET clause indicates the CONTROL identifier that causes the SUM counter in the elementary item entry to be reset to zero on a CONTROL break.

Format
RESET ON { identifier-1 / FINAL }

After presentation of the CONTROL FOOTING report group, the counters associated with the report group are reset automatically to zero, unless an explicit RESET clause is given specifying reset based on a higher level control than the associated control for the report group.

The RESET clause may be used for progressive totaling of identifiers where subtotals of identifiers may be desired without automatic resetting upon producing the report group.

Identifier-1 must be one of the identifiers described in the CONTROL clause in the Report Description entry (RD). Identifier-1 must be a higher level CONTROL identifier than the CONTROL identifier associated with the CONTROL FOOTING report group in which the SUM and RESET clauses appear.

The RESET clause may be used only in conjunction with a SUM clause.

BLANK WHEN ZERO Clause

The BLANK WHEN ZERO clause is applicable here as discussed in "Data Division."

SOURCE, SUM, or VALUE Clause

The SOURCE, SUM, or VALUE clause defines the purpose of this elementary item within the report group.

Format
SOURCE IS { TALLY / identifier-1 }
SUM { TALLY / identifier-2 } [TALLY / identifier-3] ...[UPON data-name]
VALUE IS literal-1

SOURCE: The SOURCE clause indicates a data item that is to be used as the source for this report item. The item is presented according to the PICTURE clause and the COLUMN clause in this elementary item entry.

The SOURCE clause has two functions:

1. To specify a data item that is to be printed
2. To specify a data item that is to be summed in a CONTROL FOOTING report group (see the discussion of the SUM clause)

TALLY may be used in place of identifier-1 in a SOURCE clause.

Program Product Information (Version 4)

For Version 4, a SOURCE data item may also be an elementary numeric data item appearing in the Communication Section.

SUM: The SUM clause is used to cause automatic summation of data and may appear only in an elementary item entry of a CONTROL FOOTING report group. The presence of a SUM clause defines a SUM counter. If a SUM counter is to be referred to by a Procedure Division statement or Report Section entry, a data-name clause must be specified with the SUM clause entry. The data-name then represents the summation counter generated by the Report Writer to total the operands specified immediately following SUM. If reference is never made to a summation counter, the counter need not be named explicitly by a data-name entry.

Whether the elementary item entry that contains a SUM clause names the summation counter or not, the PICTURE clause must be specified for each SUM counter. Editing characters or editing clauses may be included in the description of a SUM counter. Editing of a SUM counter occurs only upon presentation of that SUM counter. At all other times, the SUM counter is treated as a numeric data item. The SUM counter must be large enough to accommodate the summed quantity without truncation of integral digits.

An operand of a SUM clause must be an elementary numeric data item that appears in the File, Working-Storage, or Linkage Section, or is the name of a SUM counter. A SUM counter that is an operand of SUM clause must be defined in the same CONTROL FOOTING report group that contains this SUM clause or in a CONTROL FOOTING report group that is at a lower level in the control hierarchy of this report.

A SUM counter is incremented by its operands in the following manner:

- An operand that is an elementary numeric data item appearing in the File, Working-Storage, or Linkage Section is added to the SUM counter upon the generation of a DETAIL report group that contains this operand as a SOURCE data item; even if the operand appears in more than one SOURCE clause of the DETAIL report group, it is added only once to the SUM counter. The operands must appear exactly as they are in the SOURCE clauses with regard to qualification, subscripting, and indexing. If the SUM clause contains the UPON option, the operand does not have to appear in a SOURCE clause of the DETAIL report group.

- An operand that is a SUM counter and is defined in a CONTROL FOOTING that is at any lower level in the control hierarchy of this report is summed before presentation of the CONTROL FOOTING in which it is defined. This counter updating is commonly called rolling counters forward.

- An operand that is a SUM counter and is defined in the same CONTROL FOOTING as this SUM clause, is summed before presentation of this CONTROL FOOTING. This counter updating is commonly called cross-footing. SUM counter operands are added to their respective SUM counters in the order in which they physically appear in the CONTROL FOOTING report group description, i.e., left to right within an elementary item entry and down the elementary item entries.

The UPON data-name option is required to obtain selective summation for a particular data item that is named as a SOURCE item in two or more DETAIL report groups. Identifier-2 and identifier-3 must be SOURCE data items in data-name. However, this compiler does not require that identifier-2, identifier-3, etc., be SOURCE data items in data-name. Data-name must be the name of a DETAIL report group.

The following is the chronology of summing events.

1. Cross-footing and counter rolling.
2. Execution of the USE BEFORE REPORTING procedure.
3. Presentation of the control footing if it is not a null group.

4. SUM counter resetting unless an explicit RESET clause appears in the entry that defines the SUM counter.

TALLY may be used in place of an identifier in the SUM clause. It is treated as an elementary numeric data item.

VALUE: The VALUE clause causes the report data item to assume the specified value each time its report group is presented only if the elementary item entry does not contain a GROUP INDICATE clause. If the GROUP INDICATE clause is present and a given object time condition exists, the item will not assume the specified value (see GROUP INDICATE rules).

PROCEDURE DIVISION CONSIDERATIONS

To produce a report, the INITIATE, GENERATE, and TERMINATE statements must be specified in the Procedure Division. In addition, a USE BEFORE REPORTING declarative section may be written in a declarative section of the Procedure Division. This option allows the programmer to manipulate or alter data immediately before it is printed.

GENERATE Statement

The GENERATE statement is used to produce a report.

Format
GENERATE identifier

Identifier is the name of either a DETAIL report group or an RD entry.

Detail Reporting

If identifier is the name of a DETAIL report group, the GENERATE statement does all the automatic operations within a Report Writer program and produces an actual output detail report group on the output medium. At least one DETAIL report group must be specified.

Summary Reporting

If identifier is the name of an RD entry, the GENERATE statement does all of the automatic operations of the Report Writer except producing any detail report group associated with the report. For summary reporting a DETAIL report group need not be specified.

In summary reporting, SUM counters are algebraically incremented in the same manner as for detail reporting.

If more than one DETAIL report group is specified in a report, SUM counters are algebraically incremented as though consecutive GENERATE statements were issued for all the DETAIL report groups of the report. This consecutive summing takes place in the order of the physical appearance of the DETAIL report group descriptions. Even if there is more than one DETAIL report group within a report, only one test for control break is made for each GENERATE report-name. This test is made by the Report Writer prior to the summary reporting. After initiating a report and before terminating the same report, both detail reporting and summary reporting may be performed.

Operation of the GENERATE Statement

A GENERATE statement, in both detail and summary reporting, implicitly produces the following automatic operations (if defined):

1. Steps and tests the LINE COUNTER and/or PAGE COUNTER to produce appropriate PAGE FOOTING and/or PAGE HEADING report groups, after a line is printed.

2. Recognizes any specified control breaks to produce appropriate CONTROL FOOTING and/or CONTROL HEADING report groups.

3. Accumulates into the SUM counters all specified identifier(s). Resets the SUM counters.

4. Executes any specified routines defined by a USE statement before generation of the associated report group(s).

During the execution of the first GENERATE statement, the following report groups associated with the report (if specified) are produced in the order:

1. REPORT HEADING report group

2. PAGE HEADING report group

3. All CONTROL HEADING report groups in the order FINAL, major to minor

4. The DETAIL report group if specified in the GENERATE statement

If a control break is recognized at the time of the execution of a GENERATE statement (other than the first that is executed for a report), all CONTROL FOOTING report groups specified for the report are produced from the minor report group, up to and including the report group specified for the identifier which caused the control break. Then, the CONTROL HEADING report group(s) specified for the report are produced, starting with the report group specified for the identifier that caused the control break, and continuing down to and ending with the minor report group. Then, the DETAIL report group specified in the GENERATE statement is produced.

Data is moved to the data item in the Report Group Description entry of the Report Section and is edited under control of the Report Writer according to the same rules for movement and editing as described for the MOVE statement (see "Procedure Division").

INITIATE Statement

The INITIATE statement begins the processing of a report.

Format
INITIATE report-name-1 [report-name-2] ...

Each report-name must be defined by a Report Description entry in the Report Section of the Data Division.

The INITIATE statement resets all data-name entries that contain SUM clauses associated with the report; the Report Writer controls for all the TYPE report groups that are associated with this report are set up in their respective order.

The PAGE-COUNTER register, if specified, is set to 1 (one) during the execution of the INITIATE statement. If a starting value other than 1 is desired, the programmer may reset this PAGE-COUNTER following the INITIATE statement.

The LINE-COUNTER register, if specified, is set to zero during the execution of the INITIATE statement.

The PRINT-SWITCH register is set to zero during the execution of the INITIATE statement.

The INITIATE statement does not open the file with which the report is associated; an OPEN statement for the file must be given by the user. The INITIATE statement performs Report Writer functions for individually described reports analogous to the input/output functions that the OPEN statement performs for individually described files.

A second INITIATE statement for a particular report-name may not be executed unless a TERMINATE statement has been executed for that report-name subsequent to the first INITIATE statement.

TERMINATE Statement

The TERMINATE statement completes the processing of a report.

Format
TERMINATE report-name-1 [report-name-2] ...

Each report-name given in a TERMINATE statement must be defined by an RD entry in the Data Division.

The TERMINATE statement produces all the CONTROL FOOTING report groups associated with this report as though a control break had just occurred at the highest level, and completes the Report Writer functions for the named reports. The TERMINATE statement also produces the last REPORT FOOTING report group associated with this report.

Appropriate PAGE HEADING and/or PAGE FOOTING report groups are prepared in their respective order for the report description.

A second TERMINATE statement for a particular report may not be executed unless a second INITIATE statement has been executed for the report-name.

The TERMINATE statement does not close the file with which the report is associated; a CLOSE statement for the file must be given by the user. The TERMINATE statement performs Report Writer functions for individually described report programs analogous to the input/output functions that the CLOSE statement performs for individually described files.

If, at object time, no GENERATE statement is executed for a report, the TERMINATE statement of the report will not produce any report groups and will not perform any SUM processing.

SOURCE clauses used in CONTROL FOOTING FINAL or REPORT FOOTING report groups refer to the values of the items during the execution of the TERMINATE statement.

USE Sentence

The USE sentence specifies Procedure Division statements that are executed just before a report group named in the Report Section of the Data Division is produced.

Format
USE BEFORE REPORTING data-name.

A USE sentence, when present, must immediately follow a section header in the declaratives portion of the Procedure Division and must be followed by a period followed by a space. The remainder of the section must consist of one or more procedural paragraphs that define the procedures to be used.

Data-name represents a report group named in the Report Section of the Data Division. A data-name must not appear in more than one USE sentence. Data-name must be qualified by the report-name if data-name is not unique.

No Report Writer statement (GENERATE, INITIATE, or TERMINATE) may be written in a procedural paragraph(s) following the USE sentence in the declaratives portion.

The USE sentence itself is never executed; rather it defines the conditions calling for the execution of the USE procedures.

The designated procedures are executed by the Report Writer just before the named report group is produced, regardless of page or control break associations with report groups. The report group may be any type except DETAIL.

Within a USE procedure, there must not be any reference to any nondeclarative procedures. Conversely, in the nondeclarative portion, there must be no reference to procedure names that appear in the Declaratives Section, except that PERFORM statements may refer to a USE procedure or to the procedures associated with the USE procedure.

When the user wishes to suppress the printing of the specified report groups, the statement

```
MOVE 1 TO PRINT-SWITCH
```

is used in the USE BEFORE REPORTING declarative section. When this statement is encountered, only the specified report group is not printed; the statement must be written for each report group whose printing is to be suppressed.

Use of PRINT-SWITCH to suppress the printing of a report group implies that:

1. Nothing is printed
2. The LINE-COUNTER is not altered
3. The function of the NEXT GROUP clause, if one appears in the report group description, is nullified

SPECIAL REGISTERS: PAGE-COUNTER AND LINE-COUNTER

The fixed data-names, PAGE-COUNTER and LINE-COUNTER, are numeric counters automatically generated by the Report Writer based on the presence of specific entries; they do not require data description clauses. The description of these two counters is included here in order to explain their resultant effect on the overall report format.

PAGE-COUNTER

A PAGE-COUNTER is a counter generated by the Report Writer to be used as a source data item in order to present the page number on a report line. A PAGE-COUNTER is generated for a report by the Report Writer if a PAGE-LIMIT clause is specified in the RD entry of the report. The numeric counter is a 3-byte COMPUTATIONAL-3 item that is presented according to the PICTURE clause associated with the elementary item whose SOURCE is PAGE-COUNTER.

If more than one PAGE-COUNTER is given as a SOURCE data item within a given report, the number of numeric characters indicated by the PICTURE clauses must be identical. If more than one PAGE-COUNTER exists in the program, the user must qualify PAGE-COUNTER by the report name.

PAGE-COUNTER may be referred to in Report Section entries and in Procedure Division statements. After an INITIATE statement, PAGE-COUNTER contains one; if a starting value for PAGE-COUNTER other than one is desired, the programmer may change the contents of the PAGE-COUNTER by a Procedure Division statement after an INITIATE statement has been executed. PAGE-COUNTER is automatically incremented by one each time a page break is recognized by the Report Writer, after the production of any PAGE FOOTING report group but before production of any PAGE HEADING report group.

LINE-COUNTER

A LINE-COUNTER is a counter used by the Report Writer to determine when a PAGE HEADING and/or a PAGE FOOTING report group is to be presented. One line counter is supplied for each report with a PAGE LIMIT(S) clause written in the Report Description entry (RD). The numeric counter is a 3-byte COMPUTATIONAL-3 item that is presented

according to the PICTURE clause associated with the elementary item whose SOURCE is LINE-COUNTER.

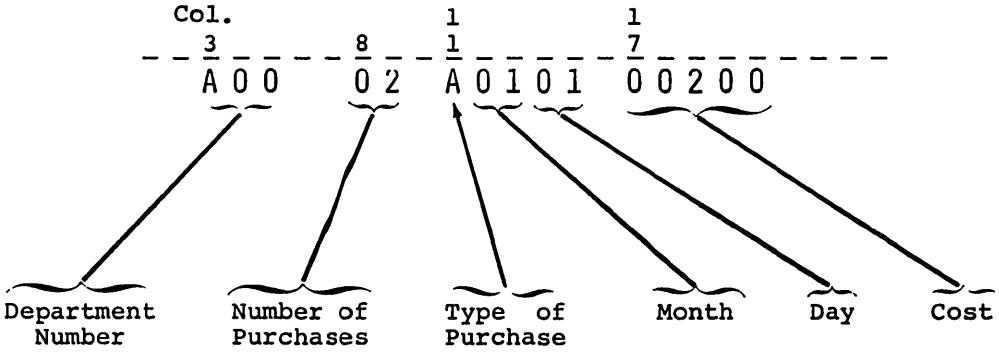
LINE-COUNTER may be referred to in Report Section entries and in Procedure Division statements. If more than one Report Description entry (RD) exists in the Report Section, the user must qualify LINE-COUNTER by the report-name. LINE-COUNTER is automatically tested and incremented by the Report Writer based on control specifications in the PAGE LIMIT(S) clause and values specified in the LINE NUMBER and NEXT GROUP clauses. After an INITIATE statement, LINE-COUNTER contains zero. Changing the value of LINE-COUNTER by Procedure Division statements may cause page format control to become unpredictable in the Report Writer.

The value of LINE-COUNTER during any Procedure Division test statement represents the number of the last line printed by the previously generated report group or represents the number of the last line skipped to by a previous NEXT GROUP specification.

In a USE BEFORE REPORTING, if no lines have been printed or skipped on the current page, LINE-COUNTER will contain zero. In all other cases, LINE-COUNTER represents the last line printed or skipped.

SAMPLE REPORT WRITER PROGRAM

The program in Figure 15 illustrates a Report Writer source program. The records used in the report (i.e., input data) are shown after the STOP RUN card in the program. Using the first record as an example, the data fields are arranged in the following format:



The decimal point in the cost field is assumed to be two places from the right.

```
000005  IDENTIFICATION DIVISION.
000010  PROGRAM-ID. ACME.
000015  REMARKS. THE REPORT WAS PRODUCED BY THE REPORT WRITER.
000020  ENVIRONMENT DIVISION.
000025  CONFIGURATION SECTION.
000030  SOURCE-COMPUTER.  IBM-360-H50.
000035  OBJECT-COMPUTER.  IBM-360-H50.
000040  INPUT-OUTPUT SECTION.
000045  FILE-CONTROL.
000050      SELECT INFILE ASSIGN TO UT-S-INFILE.
000055      SELECT REPORT-FILE ASSIGN TO UT-S-OUTPRINT.
000060  DATA DIVISION.
000065  FILE SECTION.
000070  FD  INFILE
000075      LABEL RECORDS ARE OMITTED
000080      DATA RECORD IS INPUT-RECORD.
000085  01  INPUT-RECORD.
000090      05  FILLER          PICTURE AA.
000095      05  DEPT            PICTURE XXX.
000100      05  FILLER          PICTURE AA.
000105      05  NO-PURCHASES    PICTURE 99.
000110      05  FILLER          PICTURE A.
000115      05  TYPE-PURCHASE   PICTURE A.
000120      05  MONTH           PICTURE 99.
000125      05  DAY-1           PICTURE 99.
000130      05  FILLER          PICTURE A.
000135      05  COST            PICTURE 999V99.
000140      05  FILLER          PICTURE X(59).
000145  FD  REPORT-FILE
000150      LABEL RECORDS ARE STANDARD
000151      RECORD CONTAINS 121 CHARACTERS
000155      REPORT IS EXPENSE-REPORT.
000160  WORKING-STORAGE SECTION.
000165  77  SAVED-MONTH         PICTURE 99 VALUE IS 0.
000175  77  CONTINUED           PICTURE X(11) VALUE IS SPACE.
```

Figure 15. Sample Program Using the Report Writer Feature (Part 1 of 4)

```
000180  01  MONTH-TABLE.
000185      05  RECORD-MONTH.
000190          10  FILLER  PICTURE A(9) VALUE IS "JANUARY  ".
000195          10  FILLER  PICTURE A(9) VALUE IS "FEBRUARY ".
000200          10  FILLER  PICTURE A(9) VALUE IS "MARCH    ".
000205          10  FILLER  PICTURE A(9) VALUE IS "APRIL    ".
000210          10  FILLER  PICTURE A(9) VALUE IS "MAY      ".
000215          10  FILLER  PICTURE A(9) VALUE IS "JUNE     ".
000220          10  FILLER  PICTURE A(9) VALUE IS "JULY     ".
000225          10  FILLER  PICTURE A(9) VALUE IS "AUGUST   ".
000230          10  FILLER  PICTURE A(9) VALUE IS "SEPTEMBER".
000235          10  FILLER  PICTURE A(9) VALUE IS "OCTOBER  ".
000240          10  FILLER  PICTURE A(9) VALUE IS "NOVEMBER ".
000245          10  FILLER  PICTURE A(9) VALUE IS "DECEMBER ".
000250      05  RECORD-AREA REDEFINES RECORD-MONTH.
000255          10  MONTHNAME   PICTURE A(9) OCCURS 12 TIMES.
000260  REPORT SECTION.
000265  RD  EXPENSE-REPORT
000270      CONTROLS ARE FINAL MONTH DAY-1
000275      PAGE LIMIT IS 59 LINES
000280      HEADING 1
000285      FIRST DETAIL 9
000290      LAST DETAIL 48
000295      FOOTING 52.
000300  01  TYPE IS REPORT HEADING.
000305      05  LINE NUMBER IS 1
000310          COLUMN NUMBER IS 27
000315          PICTURE IS A(26)
000320          VALUE IS "ACME MANUFACTURING COMPANY".
000325      05  LINE NUMBER IS 3
000330          COLUMN NUMBER IS 26
000335          PICTURE IS A(29)
000340          VALUE IS "QUARTERLY EXPENDITURES REPORT".
000345  01  PAGE-HEAD
000350      TYPE IS PAGE HEADING.
000355      05  LINE NUMBER IS 5.
000360          10  COLUMN IS 30
000365              PICTURE IS A(9)
000370              SOURCE IS MONTHNAME OF RECORD-AREA (MONTH).
000375          10  COLUMN IS 39
000380              PICTURE IS A(12)
000385              VALUE IS "EXPENDITURES".
000390          10  COLUMN IS 52
000395              PICTURE IS X(11)
000400              SOURCE IS CONTINUED.
000405      05  LINE IS 7.
000410          10  COLUMN IS 2
000415              PICTURE IS X(35)
000420              VALUE IS "MONTH      DAY   DEPT  NO-PURCHASES".
000425          10  COLUMN IS 40
000430              PICTURE IS X(33)
000435              VALUE IS "TYPE        COST  CUMULATIVE-COST".
```

Figure 15. Sample Program Using the Report Writer Feature (Part 2 of 4)

```
000440  01  DETAIL-LINE TYPE IS DETAIL LINE NUMBER IS PLUS 1.
000445      05  COLUMN IS 2 GROUP INDICATE PICTURE IS A(9)
000450          SOURCE IS MONTHNAME OF RECORD-AREA (MONTH).
000455      05  COLUMN IS 13 GROUP INDICATE PICTURE IS 99
000460          SOURCE IS DAY-1.
000465      05  COLUMN IS 19 PICTURE IS XXX SOURCE IS DEPT.
000470      05  COLUMN IS 31 PICTURE IS Z9 SOURCE IS NO-PURCHASES.
000475      05  COLUMN IS 42 PICTURE IS A SOURCE IS TYPE-PURCHASE.
000480      05  COLUMN IS 50 PICTURE IS ZZ9.99 SOURCE IS COST.
000485  01  TYPE IS CONTROL FOOTING DAY-1.
000490      05  LINE NUMBER IS PLUS 2.
000495          10  COLUMN 2 PICTURE X(22)
000500              VALUE "PURCHASES AND COST FOR".
000505          10  COLUMN 24  PICTURE Z9 SOURCE SAVED-MONTH.
000510          10  COLUMN 26  PICTURE X VALUE "-".
000515          10  COLUMN 27  PICTURE 99 SOURCE DAY-1.
000520          10  COLUMN 30  PICTURE ZZ9 SUM NO-PURCHASES.
000525          10  MIN
000530              COLUMN 49  PICTURE $$$9.99 SUM COST.
000535          10  COLUMN 65  PICTURE $$$$9.99 SUM COST
000540              RESET ON FINAL.
000545      05  LINE PLUS 1 COLUMN 2 PICTURE X(71)
000550          VALUE ALL "*".
000555  01  TYPE CONTROL FOOTING MONTH
000560      LINE PLUS 1 NEXT GROUP NEXT PAGE.
000565      05  COLUMN 16      PICTURE A(14) VALUE "TOTAL COST FOR".
000570      05  COLUMN 31      PICTURE A(9)
000575          SOURCE MONTHNAME OF RECORD-AREA (MONTH).
000580      05  COLUMN 43      PICTURE AAA VALUE "WAS".
000585      05  INT
000590          COLUMN 48      PICTURE $$$9.99 SUM MIN.
000595  01  TYPE CONTROL FOOTING FINAL LINE NEXT PAGE.
000600      05  COLUMN 16      PICTURE A(26)
000605          VALUE "TOTAL COST FOR QUARTER WAS".
000610      05  COLUMN 45      PICTURE $$$$9.99 SUM INT.
000615  01  TYPE PAGE FOOTING LINE 57.
000620      05  COLUMN 59      PICTURE X(12) VALUE "REPORT-PAGE-".
000625      05  COLUMN 71      PICTURE 99 SOURCE PAGE-COUNTER.
000630  01  TYPE REPORT FOOTING
000635      LINE PLUS 1 COLUMN 32 PICTURE A(13)
000640      VALUE "END OF REPORT".
000645  PROCEDURE DIVISION.
000650  DECLARATIVES.
000655  PAGE-HEAD-RTN SECTION.
000660      USE BEFORE REPORTING PAGE-HEAD.
000665  PAGE-HEAD-RTN-SWITCH.
000670      GO TO PAGE-HEAD-RTN-TEST.
000675  PAGE-HEAD-RTN-TEST.
000680      IF MONTH = SAVED-MONTH MOVE "(CONTINUED)" TO CONTINUED
000685          ELSE MOVE SPACES TO CONTINUED
000690               MOVE MONTH TO SAVED-MONTH.
000695      GO TO PAGE-HEAD-RTN-EXIT.
000697  PAGE-HEAD-RTN-ALTER.
000698      ALTER PAGE-HEAD-RTN-SWITCH
              TO PAGE-HEAD-RTN-SUPPRESS.
000700  PAGE-HEAD-RTN-SUPPRESS.
000705      MOVE 1 TO PRINT-SWITCH.
000710  PAGE-HEAD-RTN-EXIT.
000715      EXIT.
000720  END DECLARATIVES.
```

Figure 15. Sample Program Using the Report Writer Feature (Part 3 of 4)

```
000725  OPEN-FILES.  OPEN INPUT INFILE OUTPUT REPORT-FILE.
000735      INITIATE EXPENSE-REPORT.
000740  READATA.
000745      READ INFILE AT END GO TO COMPLETE.
000755      GENERATE DETAIL-LINE.
000760      GO TO READATA.
000765  COMPLETE.
000770      PERFORM PAGE-HEAD-RTN-ALTER.
000780      TERMINATE EXPENSE-REPORT.
000785      CLOSE INFILE REPORT-FILE.
000790      STOP RUN.
          .
          .
          .
  A00   02 A0101 00200
  A02   01 A0101 00100
  A02   02 C0101 01600
  A01   02 B0102 00200
  A04   10 A0102 01000
          .
          .
          .
  A01   06 C0329 04800
  A03   20 E0331 06000
```

Figure 15. Sample Program Using the Report Writer Feature (Part 4 of 4)

Key Relating Report to Report Writer Source Program

In the key, the numbers enclosed in circles (for example, (1)) relate the explanation below to the corresponding output line in Figure 16.

The 6-digit numbers (for example, 000615) show the source statement from the program illustrated in Figure 15.

(1) is the REPORT HEADING resulting from source lines 000300-000340.

(2) is the PAGE HEADING resulting from source lines 000345-000435.

(3) is the DETAIL line resulting from source lines 000440-000480 (note that since it is the first detail line after a control break, the fields defined with the GROUP INDICATE clause, lines 000445-000460, appear).

(4) is a DETAIL line resulting from the same source lines as (3). In this case, however, the fields described as GROUP INDICATE do not appear (since the control break did not immediately precede the detail line).

(5) is the CONTROL FOOTING (for DAY-1) resulting from source lines 000485-000550.

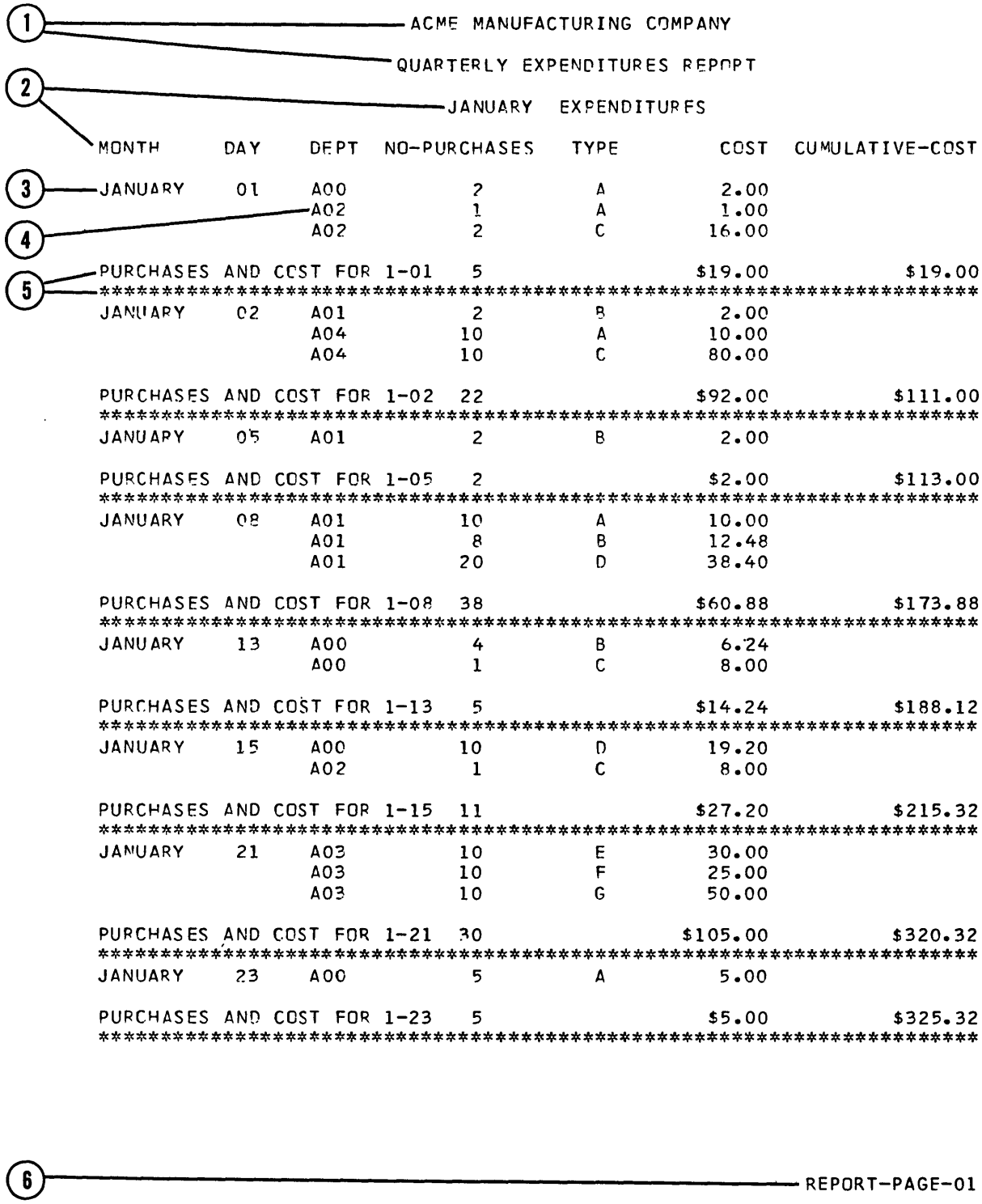
(6) is the PAGE FOOTING resulting from source lines 000615-000625.

(7) is the CONTROL FOOTING (for MONTH) resulting from source lines 000555-000575.

(8) is the CONTROL FOOTING 'for FINAL) resulting from source lines 000595-000610.

(9) is the REPORTING FOOTING resulting from source lines 000630-000640.

Lines 000650-000715 of the example illustrate a use of USE BEFORE REPORTING. The effect of the source is that each time a new page is started, a test is made to determine whether the new page is being started because a change in MONTH has been recognized (the definition for the control footing for MONTH specifies NEXT GROUP NEXT PAGE) or because the physical limits of the page were exhausted. If a change in MONTH has been recognized, spaces are moved to the PAGE HEADING; if the physical limits of the page are exhausted, "(CONTINUED)" is moved to the PAGE HEADING.



MONTH	DAY	DEPT	NO-PURCHASES	TYPE	COST	CUMULATIVE-COST
JANUARY	01	A00	2	A	2.00	
		A02	1	A	1.00	
		A02	2	C	16.00	
PURCHASES AND COST FOR 1-01			5		$19.00	$19.00
JANUARY	02	A01	2	B	2.00	
		A04	10	A	10.00	
		A04	10	C	80.00	
PURCHASES AND COST FOR 1-02			22		$92.00	$111.00
JANUARY	05	A01	2	B	2.00	
PURCHASES AND COST FOR 1-05			2		$2.00	$113.00
JANUARY	08	A01	10	A	10.00	
		A01	8	B	12.48	
		A01	20	D	38.40	
PURCHASES AND COST FOR 1-08			38		$60.88	$173.88
JANUARY	13	A00	4	B	6.24	
		A00	1	C	8.00	
PURCHASES AND COST FOR 1-13			5		$14.24	$188.12
JANUARY	15	A00	10	D	19.20	
		A02	1	C	8.00	
PURCHASES AND COST FOR 1-15			11		$27.20	$215.32
JANUARY	21	A03	10	E	30.00	
		A03	10	F	25.00	
		A03	10	G	50.00	
PURCHASES AND COST FOR 1-21			30		$105.00	$320.32
JANUARY	23	A00	5	A	5.00	
PURCHASES AND COST FOR 1-23			5		$5.00	$325.32

Figure 16. Report Produced by Report Writer Feature (Part 1 of 5)

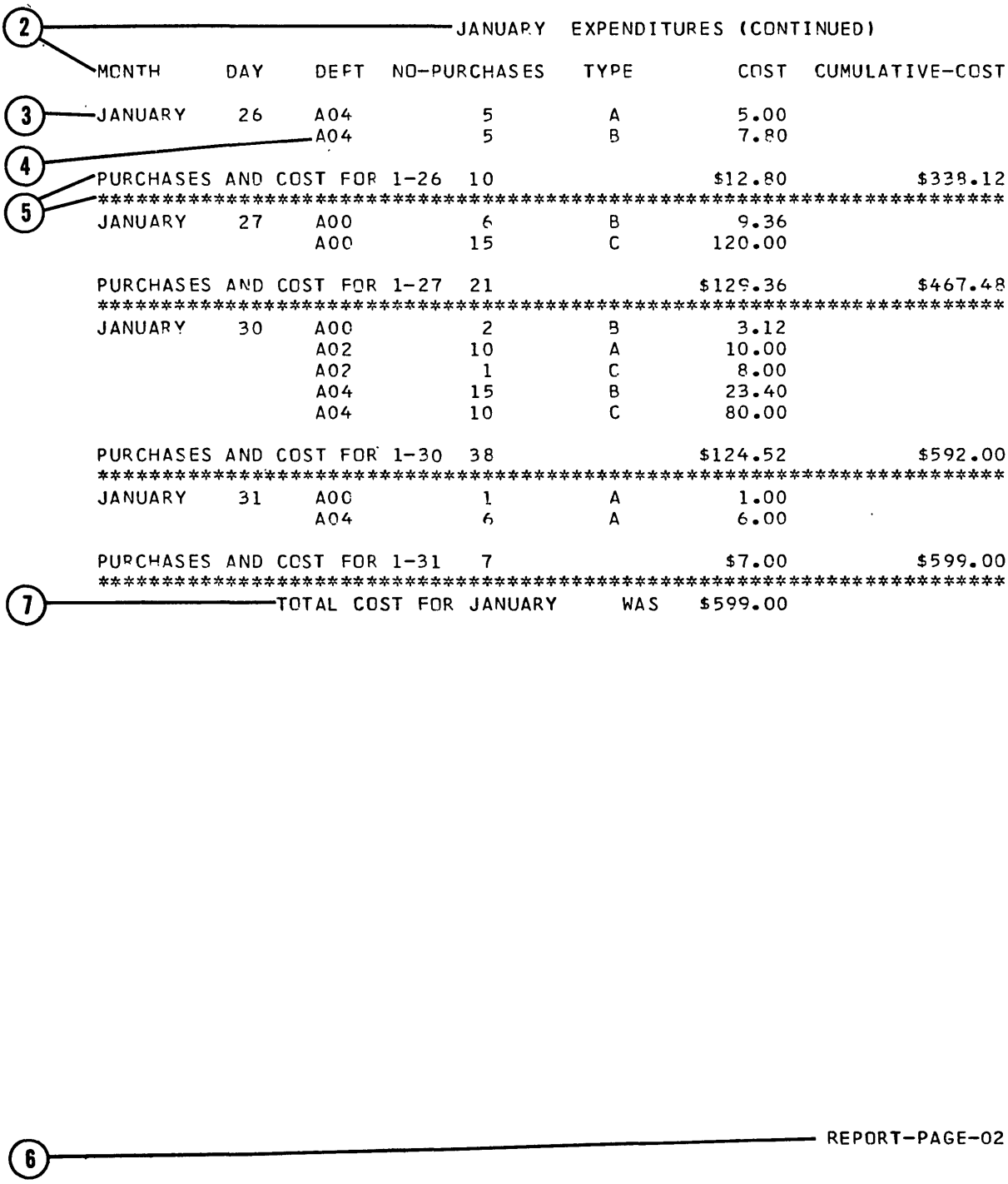


Figure 16. Report Produced by Report Writer Feature (Part 2 of 5)

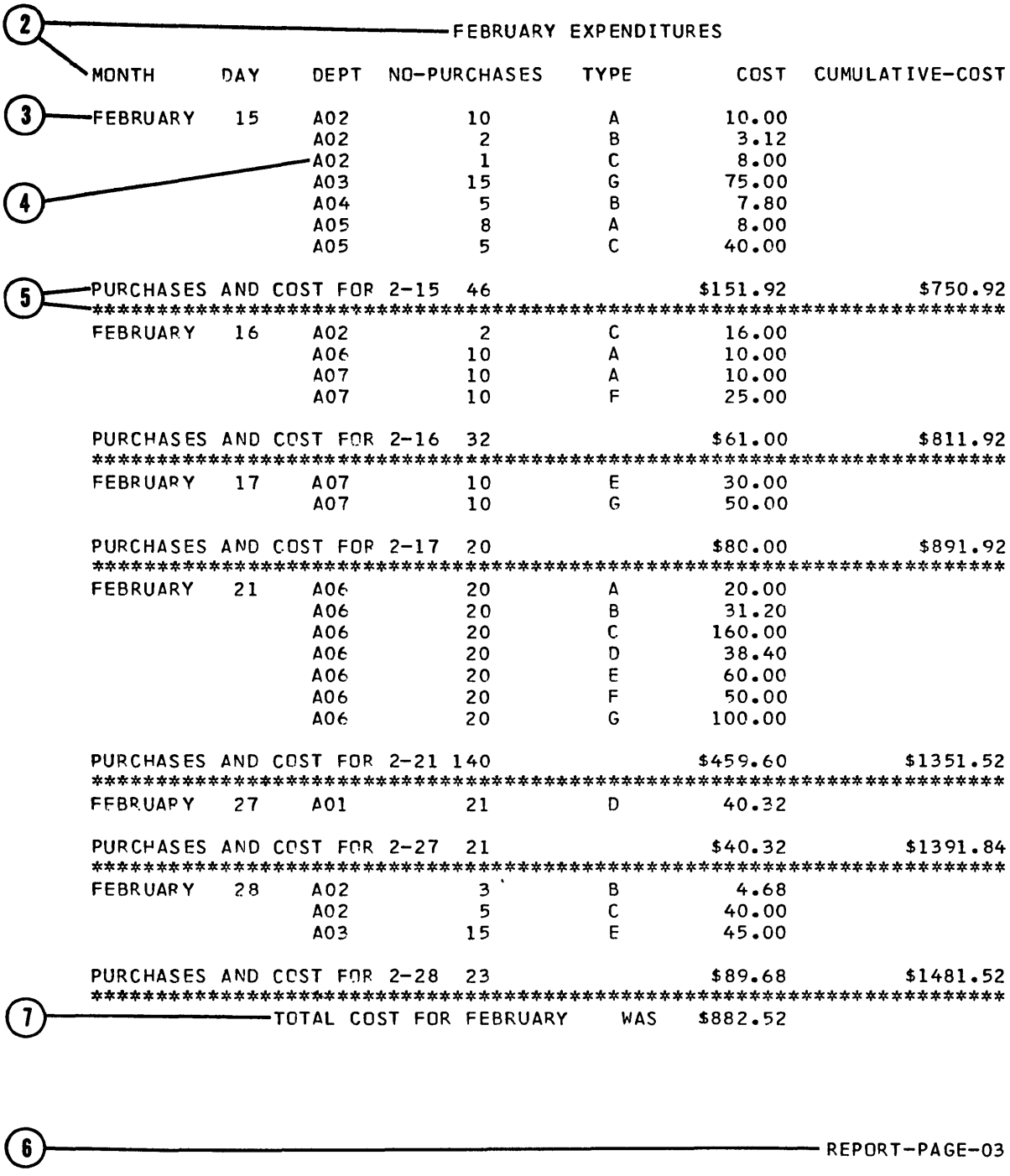


Figure 16. Report Produced by Report Writer Feature (Part 3 of 5)

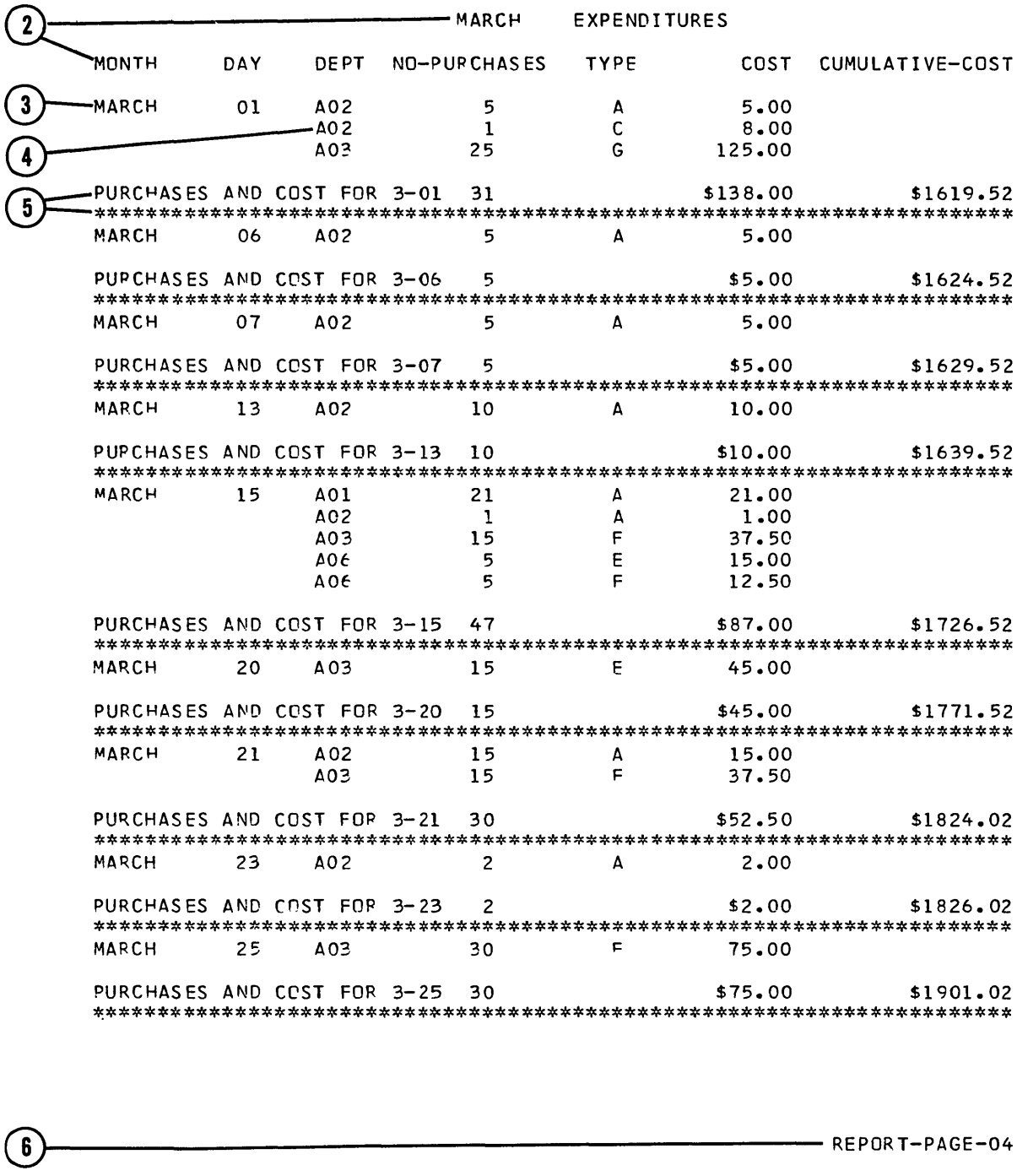


Figure 16. Report Produced by Report Writer Feature (Part 4 of 5)

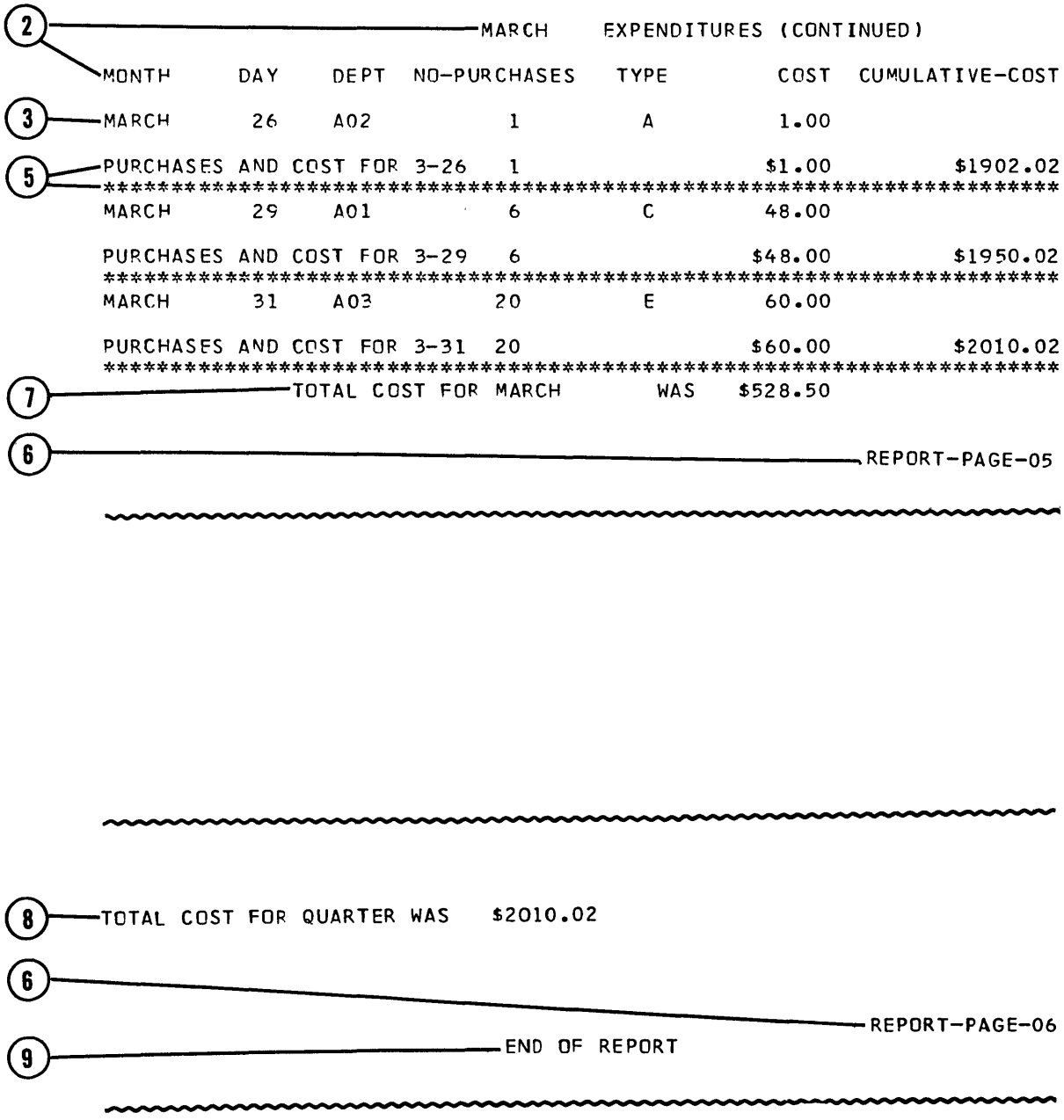


Figure 16. Report Produced by Report Writer Feature (Part 5 of 5)

TABLE HANDLING FEATURE

The Table Handling feature enables the programmer to process tables or lists of repeated data conveniently. A table may have up to three dimensions, i.e., three levels of subscripting or indexing can be handled. Such a case exists when a group item described with an OCCURS clause contains another group item with an OCCURS clause, which in turn contains an item with an OCCURS clause. To make reference to any element within such a table, each level must be subscripted or indexed.

SUBSCRIPTING

Subscripts are used only to refer to an individual element within a list or table of elements that have not been assigned individual data-names.

Format
data-name (subscript[, subscript][, subscript])

The subscript, or set of subscripts, that identifies the table element is enclosed in parentheses immediately following the space that terminates data-name, which is the name of the table element. When more than one subscript appears within a pair of parentheses, each subscript must be separated from the next by a comma followed by a space. However, this compiler allows the comma to be omitted. No space may appear between the left parenthesis and the leftmost subscript or between the rightmost subscript and the right parenthesis. To identify an element in the table named SALARY by the set of subscripts YEAR, MONTH, and WEEK, the programmer would write: SALARY (YEAR, MONTH, WEEK).

The subscript can be represented by a numeric literal that is a positive integer, by the special register TALLY, or by a data-name. Restrictions on the use of a data-name as a subscript are:

1. Data-name must be a numeric elementary item that represents a positive integer.
2. The name itself may be qualified, but not subscripted.

The subscript may contain a sign, but the lowest permissible subscript value is 1. Hence, the use of zero or a negative subscript is not permitted. The highest permissible subscript value in any particular case is the maximum number of occurrences of the item as specified in the OCCURS clause.

Qualification may be used in conjunction with subscripting, in which case OF or IN follows the data-name being subscripted.

Format

```
data-name {OF} data-name-1 [ {OF} data-name-2]...
          {IN}             {IN}

        (subscript[, subscript][, subscript])
```

Note: Data-name is the item being subscripted, not data-name-1. That is, in the statement SALARY OF EMPLOYEE-RECORD (YEAR, MONTH, WEEK), the data item SALARY is subscripted by YEAR, MONTH, and WEEK.

INDEXING

References can be made to individual elements within a table of elements by specifying indexing for that reference.

An index is assigned to a given level of a table by using an INDEXED BY clause in the definition of the table. A name given in the INDEXED BY clause is known as an index-name and is used to refer to the assigned index. An index-name must be initialized by a SET or PERFORM statement before it is used in a table reference. An index may be modified only by a SET, SEARCH, or PERFORM statement.

Format

```
data-name (index-name [ {±} integer]
    [, index-name [ {±} integer]][, index-name [ {±} integer]])
```

Direct indexing is specified by using an index-name in the form of a subscript. For example,

```
ELEMENT (PRIME-INDEX)
```

Relative indexing is specified when the terminal space of the data-name is followed by a parenthesized group of items: the index-name, followed by a space, followed by one of the operators + or -, followed by another space, followed by an unsigned integral numeric literal. For example,

```
ELEMENT (PRIME-INDEX + 5)
```

Qualification may be used in conjunction with indexing, in which case OF or IN follows the data-name being indexed.

```
┌──────────────────────────────────────────────────────────────────┐
│                            Format                                │
├──────────────────────────────────────────────────────────────────┤
│                                                                  │
│  data-name {OF} data-name-1 [ {OF} data-name-2]...               │
│            {IN}              {IN}                                │
│                                                                  │
│     (index-name [ {±} integer][, index-name [ {±} integer]]      │
│                                                                  │
│     [, index-name [ {±} integer]])                               │
│                                                                  │
└──────────────────────────────────────────────────────────────────┘
```

Note: Data items described by the USAGE IS INDEX clause permit storage of the values of index-names as data without conversion. Such data items are called index data items.

RESTRICTIONS ON INDEXING, SUBSCRIPTING, AND QUALIFICATION

Tables may have one, two, or three dimensions. Therefore, references to an element in a table may require up to three subscripts or indexes.

1. A data-name must not be subscripted or indexed when the data-name is itself being used as an index, subscript, or qualifier.
2. When qualification, subscripting, or indexing are required for a given data item, the indexes or subscripts are specified after all necessary qualification is given.
3. Subscripting and indexing must not be used together in a single reference.
4. Wherever subscripting is not permitted, indexing is not permitted.
5. The commas shown in the formats for indexes and subscripts are required.

EXAMPLE OF SUBSCRIPTING AND INDEXING

For a table with three levels of indexing, the following Data Division entries would result in a storage layout as shown in Figure 17.

```
01  PARTY-TABLE REDEFINES TABLE.
    05  PARTY-CODE OCCURS  3 TIMES INDEXED BY PARTY.
        10  AGE-CODE OCCURS 3 TIMES INDEXED BY AGE.
            15  M-F-INFO OCCURS 2 TIMES INDEXED BY M-F
                PICTURE 9(7)V9 USAGE DISPLAY.
```

PARTY-TABLE contains three levels of indexing. Reference to elementary items within PARTY-TABLE is made by use of a name that is subscripted or indexed. A typical Procedure Division statement might be:

```
MOVE M-F-INFO (PARTY, AGE, M-F) TO M-F-RECORD.
```

In order to use the Table Handling feature, the programmer must provide certain information in the Data Division and Procedure Division of the program.

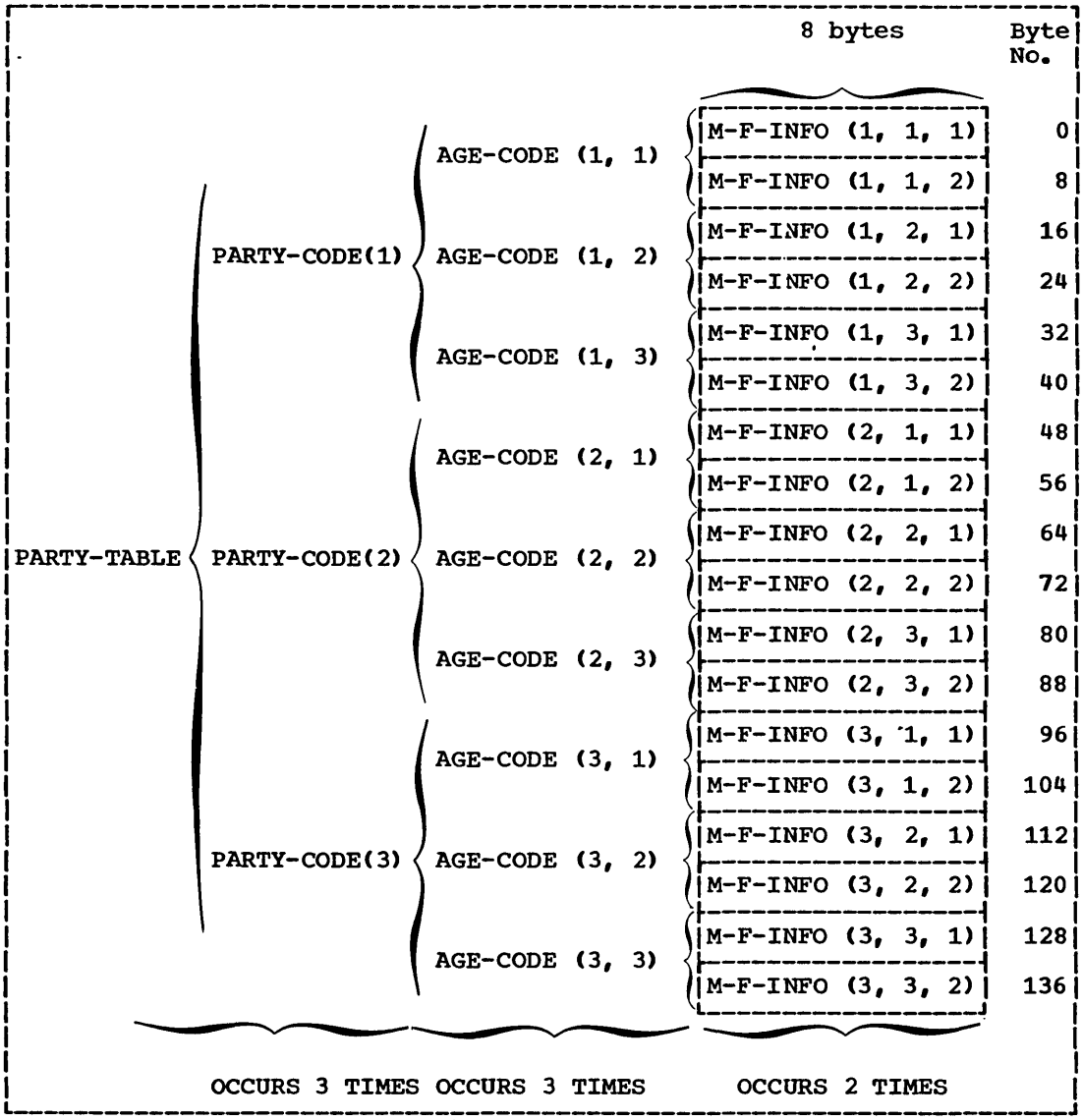


Figure 17. Storage Layout for PARTY-TABLE

Note: Programming techniques for Table Handling are given in detail in the Programmer's Guide.

DATA DIVISION CONSIDERATIONS FOR TABLE HANDLING

The OCCURS and USAGE clauses are included as part of the record description entries in a program utilizing the Table Handling feature.

OCCURS Clause

The OCCURS clause eliminates the need for separate entries for repeated data, since it indicates the number of times a series of records with identical format is repeated. It also supplies information required for the application of subscripts or indexes.

The OCCURS clause has three formats.

Format 1

```
OCCURS integer-2 TIMES
    [ {ASCENDING } KEY IS data-name-2 [data-name-3] ... ] ...
      {DESCENDING}
      [INDEXED BY index-name-1 [index-name-2] ... ]
```

Format 2

```
OCCURS integer-1 TO integer-2 TIMES [DEPENDING ON data-name-1]
    [ {ASCENDING } KEY IS data-name-2 [data-name-3] ... ] ...
      {DESCENDING}
      [INDEXED BY index-name-1 [index-name-2] ... ]
```

Format 3

```
OCCURS integer-2 TIMES [DEPENDING ON data-name-1]
    [ {ASCENDING } KEY IS data-name-2 [data-name-3] ... ] ...
      {DESCENDING}
      [INDEXED BY index-name-1 [index-name-2] ... ]
```

The other data description clauses associated with an entry whose description includes an OCCURS clause apply to each occurrence of the item described.

Since three subscripts or indexes are allowed, three nested levels of the OCCURS clause are allowed. That is, 3-dimensional tables can be specified. No table may be longer than 32767 bytes in length, except for fixed-length tables in the Working-Storage Section or Linkage Section, which may be as long as 131071 bytes.

The subject of an OCCURS clause is the data-name of the entry that contains this OCCURS clause. The subject of an OCCURS clause must be subscripted or indexed whenever it is referred to in any statement other than SEARCH.

When subscripted, the subject refers to one occurrence within the table. When not subscripted (permissible only in the SEARCH statement), the subject represents the entire table element.

The OCCURS clause may not be specified in a data description entry that:

1. Has a level-01 or level-77 number

2. Describes an item whose size is variable

(The size of an item is variable if the data description of any subordinate item within it contains an OCCURS DEPENDING ON clause -- that is, an OCCURS clause with the DEPENDING ON option.)

However, this compiler allows the size of any subordinate item to be variable -- that is, to contain an OCCURS DEPENDING ON clause.

Except for condition-name entries, a record description entry that contains an OCCURS clause must not also contain a VALUE clause.

Within a given record description, the VALUE clause must not be used in a data description entry that is subsequent to a data description entry that contains an OCCURS DEPENDING ON clause.

In the discussion which follows, the term "computational" refers to COMPUTATIONAL, COMPUTATIONAL-1, and COMPUTATIONAL-2 data items.

When a computational elementary item specifies both the OCCURS and SYNCHRONIZED clauses, any necessary slack bytes for each occurrence of the item are added by the compiler. When a group item specifies the OCCURS clause and also contains SYNCHRONIZED computational elementary items, any necessary slack bytes for each occurrence of the group are added by the compiler, as well as the necessary slack bytes for each occurrence of the computational elementary items. See "Slack Bytes" in "Data Division" for a complete discussion.

In Format 1, integer-2 represents the exact number of occurrences. In this case, integer-2 must be greater than zero.

DEPENDING ON OPTION: In Format 2 and Format 3, the DEPENDING ON option is used. This indicates that the subject of this entry has a variable number of occurrences. This does not mean that the length of the subject is variable, but rather that the number of times the subject may be repeated is variable, the number of times being controlled by the value of data-name-1 at object time.

Program Product Information (Version 4)

In Version 4, the OCCURS DEPENDING ON clause may not be specified for record description entries in the Communication Section of a COBOL TP program.

In Format 2, integer-1 represents the minimum number of occurrences, and integer-2 represents the maximum number of occurrences. Integer-1 may be zero or any positive integer. Integer-2 must be greater than zero, and also greater than integer-1. Integer-2 must be less than 32,768. The value of data-name-1 must not exceed integer-2.

In Format 3, integer-2 represents the maximum number of occurrences, and it must be greater than zero and less than 32,768. The value of data-name-1 must not exceed integer-2.

Data-name-1, the object of the DEPENDING ON option:

- Must be described as a positive integer
- Must not exceed integer-2 in value
- May be qualified, when necessary
- Must not be subscripted (that is, must not itself be the subject of, or an entry within, a table)
- Must, if it appears in the same record as the table it controls, appear before the variable portion of the record

If the value of data-name-1 is reduced, the contents of data items whose occurrence numbers exceed the new value of data-name-1 become unpredictable.

Unused character positions resulting from the DEPENDING ON option will not appear on external media.

The DEPENDING ON option is required only when the last occurrence of the subject cannot otherwise be determined.

Any Data Division entry which contains an OCCURS DEPENDING ON clause, or which has subordinate to it an entry which contains an OCCURS DEPENDING ON clause, cannot be the object of a REDEFINES clause.

KEY OPTION: The KEY option is used in conjunction with the INDEXED BY option in the execution of a SEARCH ALL statement. The KEY option is used to indicate that the repeated data is arranged in ASCENDING or in DESCENDING order, according to the values contained in data-name-2, data-name-3, etc.

Data-name-2 must be either the name of the entry containing an OCCURS clause, or it must be an entry subordinate to the entry containing the OCCURS clause. If data-name-2 is the subject of this table entry, it is the only key that may be specified for this table. If data-name-2 is not the subject of this table entry, all the keys identified by data-name-2, data-name-3, etc.;

- Must be subordinate to the subject of the table entry itself
- Must not be subordinate to any other entry that contains an OCCURS clause
- Must not themselves contain an OCCURS clause

When the KEY option is specified, the following rules apply:

- Keys must be listed in decreasing order of significance.
- The total number of keys for a given table element must not exceed 12.
- The sum of the lengths of all the keys associated with one table element must not exceed 256.
- A key may have the following usages: DISPLAY, COMPUTATIONAL-3, or COMPUTATIONAL.

When subordinate entries within the table are variable in length, the following rule also applies:

- Any key in a table element must be at a fixed displacement from the beginning of that element (that is, if a table element is of variable length, then the keys must precede the variable portion).

The following example shows a violation of the last preceding rule:

```
WORKING-STORAGE SECTION.
77  CURRENT-WEEK                          PICTURE 99.
01  TABLE-RECORD.
    05  EMPLOYEE-TABLE OCCURS 100 TIMES
        ASCENDING KEY IS WAGE-RATE EMPLOYEE-NO
        INDEXED BY A, B.
        10  WEEK-RECORD OCCURS 1 TO 52 TIMES
            DEPENDING ON CURRENT-WEEK
            ASCENDING KEY IS EMPLOYEE-NAME
            INDEXED BY C.
            15  WEEK-NO                   PIC 99.
            15  AUTHORIZED-ABSENCES       PIC  9.
            15  UNAUTHORIZED-ABSENCES     PIC  9.
            15  LATENESSES                PIC  9.
        10  EMPLOYEE-NAME                 PIC X(20).
        10  EMPLOYEE-NO                   PIC 9(6).
        10  WAGE-RATE                     PIC 9999V99.
            .
            .
            .
```

WAGE-RATE and EMPLOYEE-NO are invalid as keys, since they are placed after the variable portion of the table.

The following is a corrected example of the KEY option:

```
WORKING-STORAGE SECTION.
77  CURRENT-WEEK                          PICTURE 99.
01  TABLE-RECORD.
    05  EMPLOYEE-TABLE OCCURS 100 TIMES
        ASCENDING KEY IS WAGE-RATE EMPLOYEE-NO
        INDEXED BY A,  B.
        10  EMPLOYEE-NAME                 PIC X(20).
        10  EMPLOYEE-NO                   PIC 9(6).
        10  WAGE-RATE                     PIC 9999V99.
        10  WEEK-RECORD OCCURS 1 TO 52 TIMES
            DEPENDING ON CURRENT-WEEK
            ASCENDING KEY IS WEEK-NO INDEXED BY C.
            15  WEEK-NO                   PIC 99.
            15  AUTHORIZED-ABSENCES       PIC  9.
            15  UNAUTHORIZED-ABSENCES     PIC  9.
            15  LATENESSES                PIC  9.
```

The keys WAGE-RATE and EMPLOYEE-NO both appear at a fixed displacement from the beginning of the table element EMPLOYEE-TABLE.

INDEXED BY OPTION: The INDEXED BY option is required if the subject of this entry (the data-name described by the OCCURS clause, or an item within this data-name, if it is a group item) is to be referred to by indexing. The index-name(s) identified by this clause is not defined elsewhere in the program, since its allocation and format are dependent on the system, and, not being data, cannot be associated with any data hierarchy.

The number of index-names for a Data Division entry must not exceed twelve.

An index-name must be initialized through a SET or PERFORM statement before it is used.

Each index-name is a fullword in length and contains a binary value that represents an actual displacement from the beginning of the table

that corresponds to an occurrence number in the table. The value is calculated as the occurrence number minus one, multiplied by the length of the entry that is indexed by this index-name.

For example, if the programmer writes

```
A OCCURS 15 TIMES INDEXED BY Z PICTURE IS X(10).
```

on the fifth occurrence of A, the binary value contained in Z will be:

```
Z = (5 - 1) * 10 = 40
```

Note that, for a table entry of variable length, the value contained in the index-name entry will become invalid when the table entry length is changed, unless the user issues a new SET statement to correct the value contained in the index-name.

The following example of the setting of values in index-name is incorrect:

```
                .
                .
                .
DATA DIVISION.
                .
                .
                .
77  E PICTURE S9(5) COMP SYNC.
01  ...
    05  A OCCURS 10 INDEXED BY IND-1...
        10  B OCCURS 10 DEPENDING ON E INDEXED BY IND-2...
                .
                .
                .
PROCEDURE DIVISION.
                .
                .
                .
    MOVE 8 TO E
    SET IND-1 TO 3
    SEARCH A ...
                .
                .
                .
    MOVE 10 TO E
    SEARCH A ...
```

(Moving 10 to E changes the length of the table entry A, so that IND-1 now contains an invalid value.)

The following example of the setting of values in index-name is correct:

```
                .
                .
                .
    DATA DIVISION.
                .
                .
                .
    77  E PICTURE S9(5) COMP SYNC.
    77  D PICTURE S9(5) COMP SYNC.
    01  ...
        05  A OCCURS 10 INDEXED BY IND-1...
            10  B OCCURS 10 DEPENDING ON E INDEXED BY IND-2...
                .
                .
                .
    PROCEDURE DIVISION.
                .
                .
                .
        MOVE 8 TO E
        SET IND-1 TO 3
        SET D TO IND-1
        SEARCH A ...
                .
                .
                .
        MOVE 10 TO E
        SET IND-1 TO D
        SEARCH A ...
                .
                .
                .
```

(Here the user has saved the occurrence number in D, and then later reset IND-1 to obtain the corrected value.)

There are two types of indexing: direct indexing and relative indexing.

Direct Indexing: If a data-name is used in the procedure text with index-names, the data-name itself must be the subject of an INDEXED BY option, or be subordinate to a group(s) that is the subject of the INDEXED BY option.

In the following example

```
A (INDEX-1, INDEX-2, INDEX-3)
```

implies that A belongs to a structure with three levels of OCCURS clauses, each with an INDEXED BY option. However, if data-name (A, in this example) belongs to an OCCURS structure that does not use the INDEXED BY option, this compiler accepts the specification of index-names (in this example INDEX-1, INDEX-2, INDEX-3), and assumes the user has set them to values that correspond to the occurrence number he wishes to reference.

Relative Indexing: The index-name is followed by a space, followed by one of the operators + or -, followed by another space, followed by an unsigned numeric literal. The numeric literal is considered to be an occurrence number, and is converted to an index value before being added to, or subtracted from, the corresponding index-name.

Given the following example:

```
A (Z + 1, J + 3, K + 4)
```

where:

table element indexed by Z has an entry length of 100

table element indexed by J has an entry length of 10

table element indexed by K has an entry length of 2

the resulting address will be computed as follows:

```
(ADDRESS of A) + Z + 100 * 1 + J + 10 * 3 + K + 4 * 2
```

(100 * 1, 10 * 3, 4 * 2: conversion of integers to index values)

USAGE IS INDEX Clause

The USAGE IS INDEX clause is used to specify the format of a data item stored internally.

Format
[USAGE IS] INDEX

The USAGE IS INDEX clause allows the programmer to specify index data items.

An index data item is an elementary item (not necessarily connected with any table) that can be used to save index-name values for future reference. An index data item must be assigned an index-name value (i.e., (occurrence number - 1) * entry length) through the SET statement. Such a value corresponds to an occurrence number in a table.

The USAGE IS INDEX clause may be written at any level. If a group item is described with the USAGE IS INDEX clause, it is the elementary items within the group that are index data items; the group itself is not an index data item, and the group name cannot be used in SEARCH and SET statements or in relation conditions. The USAGE clause of an elementary item cannot contradict the USAGE clause of a group to which the item belongs.

An index data item can be referred to directly only in a SEARCH or SET statement or in a relation condition. An index data item can be part of a group which is referred to in a MOVE or an input/output statement. When such operations are executed, however, there is no conversion of the contents of the index data item.

An index data item cannot be a conditional variable.

The SYNCHRONIZED, JUSTIFIED, PICTURE, BLANK WHEN ZERO, or VALUE clauses cannot be used to describe group or elementary items described with the USAGE IS INDEX clause. However, this compiler allows the use of SYNCHRONIZED when USAGE IS INDEX to obtain efficient use of the item.

PROCEDURE DIVISION CONSIDERATIONS FOR TABLE HANDLING

The SEARCH and the SET statements may be used to facilitate table handling. In addition, there are special rules involving Table Handling elements when they are used in relation conditions.

Relation Conditions

Comparisons involving index-names and/or index data items conform to the following rules:

1. The comparison of two index-names is actually the comparison of the corresponding occurrence numbers.

2. In the comparison of an index-name with a data item (other than an index data item), or in the comparison of an index-name with a literal, the occurrence number that corresponds to the value of the index-name is compared with the data item or literal.

3. In the comparison of an index data item with an index-name or another index data item, the actual values are compared without conversion.

Any other comparison involving an index data item is illegal.

Table 25 gives permissible comparisons for index-names and index data items.

Table 25. Index-names and Index Data Items -- Permissible Comparisons

First Operand \ Second Operand	Index-name	Index Data Item	Data-name (numeric integer only)	Numeric literal (integer only)
Index-name	Compare occurrence number	Compare without conversion	Compare occurrence number with data-name	Compare occurrence number with literal
Index Data Item	Compare without conversion	Compare without conversion	Illegal	Illegal
Data-name (numeric integer only)	Compare occurrence number with data-name	Illegal	See Table 12 for permissible comparisons	
Numeric literal (integer only)	Compare occurrence number with literal	Illegal		

SEARCH Statement

The SEARCH statement is used to search a table for an element that satisfies a specified condition, and to adjust the value of the associated index-name to the occurrence number corresponding to that table element.

Format 1

```
SEARCH identifier-1 [VARYING  {index-name-1}  ]
                              {identifier-2}

    [AT END imperative-statement-1]

    WHEN condition-1          {imperative-statement-2}
                              {NEXT SENTENCE         }

    [WHEN condition-2         {imperative-statement-3}  ]...
                              {NEXT SENTENCE         }
```

Format 2

```
SEARCH ALL identifier-1 [AT END imperative-statement-1]

    WHEN condition-1          {imperative-statement-2}
                              {NEXT SENTENCE         }
```

The Data Division description of identifier-1 must contain an OCCURS clause with the INDEXED BY option. Identifier-1 must not be described as a floating-point item.

When written in the SEARCH statement, identifier-1 must refer to all occurrences within one level of a table; that is, it must not be subscripted or indexed.

Identifier-1 can be a data item subordinate to a data item that contains an OCCURS clause, thus providing for a two or three dimensional table. An index-name must be associated with each dimension of the table through the INDEXED BY phrase of the OCCURS clause. Execution of a SEARCH statement causes modification only of the setting of the index-name associated with identifier-1 (and, if present, of index-name-1 or identifier-2). Therefore, to search an entire two or three dimensional table, it is necessary to execute a SEARCH statement several times; prior to each execution, SET statements must be executed to adjust the associated index-names to their appropriate settings.

In the AT END and WHEN options, if any of the specified imperative statement(s) do not terminate with a GO TO statement, control passes to the next sentence after execution of the imperative statement.

Format 1 Considerations -- Identifier-2, when specified, must be described as an index data item, or it must be a fixed-point numeric elementary item described as an integer. When an occurrence number is

incremented, identifier-2 is simultaneously incremented by the same amount.

Condition-1, condition-2, etc., may be any condition, as follows:

relation condition

class condition

condition-name condition

sign condition

(condition)

[NOT] {AND / OR} condition

(See Conditions section of "Procedure Division.")

Upon the execution of a SEARCH statement, a serial search takes place, starting with the current index setting.

If, at the start of the SEARCH, the value of the index-name associated with identifier-1 is not greater than the highest possible occurrence number for identifier-1, the following actions take place:

1. The condition(s) in the WHEN option are evaluated in the order they are written.
2. If none of the conditions is satisfied, the index-name for identifier-1 is incremented to reference the next table element, and step 1 is repeated.
3. If, upon evaluation, one of the WHEN conditions is satisfied, the search terminates immediately, and the imperative-statement associated with that condition is executed. The index-name points to the table element that satisfied the condition.
4. If the end of the table is reached without the WHEN condition being satisfied, the search terminates as described in the next paragraph.

If at the start of the search, the value of the index-name associated with identifier-1 is greater than the highest permissible occurrence number for identifier-1, the search is terminated immediately, and if the AT END option is specified, imperative-statement-1 is executed. If this option is omitted, control passes to the next sentence.

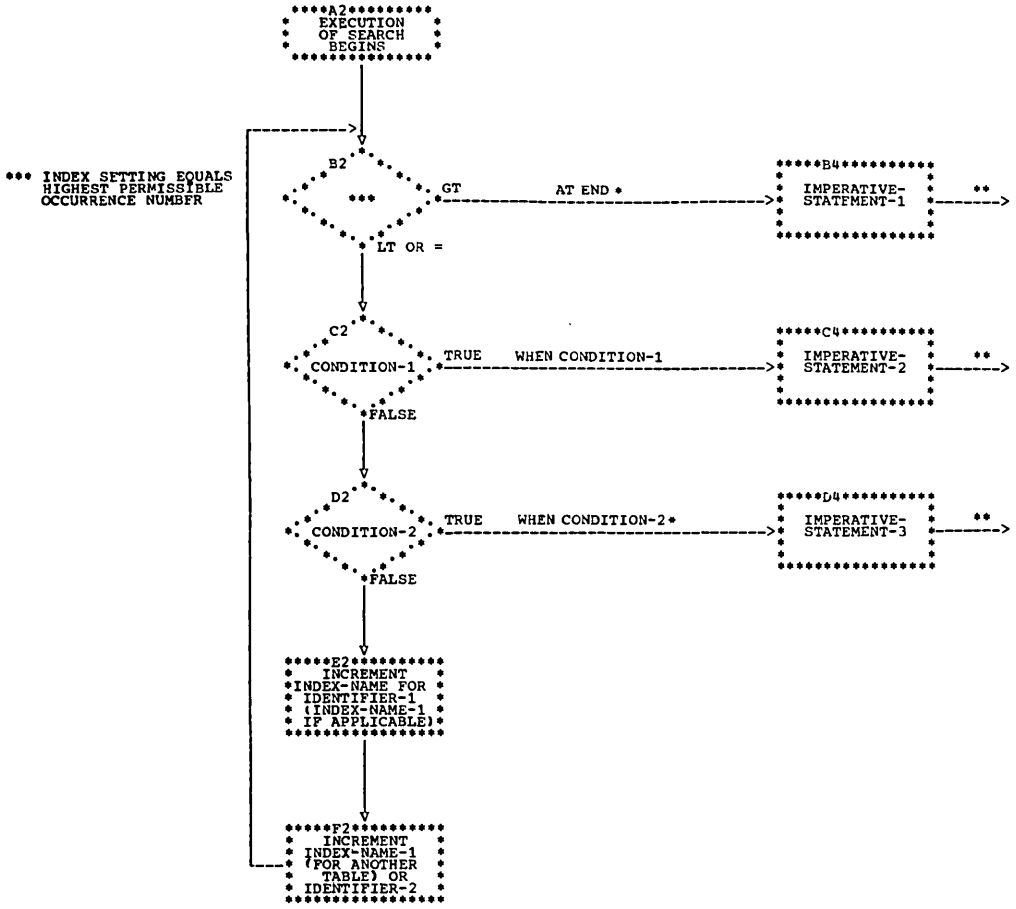
When the VARYING index-name-1 option is not specified, the index used for the search is the first (or only) index-name associated with identifier-1.

When the VARYING index-name-1 option is specified, one of the following applies:

- If index-name-1 is one of the indexes for identifier-1, index-name-1 is used for the search. Otherwise, the first (or only) index-name for identifier-1 is used.
- If index-name-1 is an index for another table entry, then when the index-name for identifier-1 is incremented to represent the next occurrence of the table, index-name-1 is simultaneously incremented to represent the next occurrence of the table it indexes.

A flowchart of the Format 1 SEARCH operation containing two WHEN options is shown in Chart 5.

Chart 5. Format 1 SEARCH Operation Containing Two WHEN Options



* THESE OPERATIONS ARE INCLUDED ONLY WHEN CALLED FOR IN THE STATEMENT.

** EACH OF THESE CONTROL TRANSFERS IS TO THE NEXT SENTENCE UNLESS THE IMPERATIVE-STATEMENT ENDS WITH A GO TO STATEMENT.

Format 2 Considerations -- The first index-name assigned to identifier-1 will be used for the search.

The description of identifier-1 must contain the KEY option in its OCCURS clause.

Condition-1 must consist of one of the following:

- A relation condition incorporating the EQUALS, EQUAL TO, or equal sign (=) relation. Either the subject or the object (but not both) of the relation-condition must consist solely of one of the data-names that appear in the KEY clause of identifier-1.
- A condition-name condition in which the VALUE clause describing the condition-name consists of a single literal only. The conditional variable associated with the condition-name must be one of the data-names that appear in the KEY clause of identifier-1.
- A compound condition formed from simple conditions of the types described above, with AND as the only connective.

Any data-name that appears in the KEY clause of identifier-1 may be tested in condition-1. However, all data-names in the KEY clause preceding the one to be tested must also be so tested in condition-1. No other tests may be made in condition-1.

For example, if the following table were defined in the Data Division:

```
77  VALUE-1 PICTURE 99.
                .
                .
                .
05  A OCCURS 10 TIMES ASCENDING KEY IS KEY1, KEY2, KEY3, KEY4
    INDEXED BY I.
    10  KEY1 PICTURE 9.
    10  KEY2 PICTURE 99.
    10  KEY3 PICTURE 9.
    10  KEY4 PICTURE 9.
        88  BLUE VALUE 1.
                .
                .
                .
```

in the Procedure Division, valid WHEN phrases could be:

```
WHEN KEY1 (I) = 3 AND KEY2 (I) = 10 AND KEY3 (I) = 5 ...

WHEN KEY1 (I) = 3 AND KEY2 (I) = VALUE-1
    AND KEY3 (I) = 5 AND BLUE (I) ...
```

During execution of a Format 2 SEARCH statement, a binary search takes place; the setting of index-name is varied during the search so that at no time is it less than the value that corresponds to the first element of the table, nor is it ever greater than the value that corresponds to the last element of the table. If condition-1 cannot be satisfied for any setting of the index within this permitted range, control is passed to imperative- statement-1 when the AT END option appears, or to the next sentence when this clause does not appear. In either case, the final setting of the index is not predictable. If the index indicates an occurrence that allows condition-1 to be satisfied, control passes to imperative-statement-2.

SET Statement

The SET statement establishes reference points for table handling operations by setting index-names to values associated with table elements. The SET statement must be used when initializing index-name values before execution of a SEARCH statement; it may also be used to transfer values between index-names and other elementary data items.

Format 1

```
SET { index-name-1 [index-name-2]... }     TO { index-name-3 }
    { identifier-1 [identifier-2]... }        { identifier-3 }
                                              { literal-1    }
```

Format 2

```
SET index-name-4 [index-name-5] ...  { UP BY   }  { identifier-4 }
                                     { DOWN BY }  { literal-2    }
```

All identifiers must name either index data items or fixed-point numeric elementary items described as integers; however, identifier-4 must not name an index data item. When a literal is used, it must be a positive integer. Index-names are related to a given table through the INDEXED BY option of the OCCURS clause; when index-names are specified in the INDEXED BY option, they are automatically defined.

All references to index-name-1, identifier-1, and index-name-4 apply equally to index-name-2, identifier-2, and index-name-5, respectively.

Format 1 Considerations -- When the SET statement is executed, one of the following actions occurs:

1. Index-name-1 is converted to a value that corresponds to the same table element to which either index-name-3, identifier-3, or literal-1 corresponds. If identifier-3 is an index data item, or if index-name-3 is related to the same table as index-name-1, no conversion takes place. To be valid, the resultant value of index-name must correspond to an occurrence number of an element in the associated table.

2. If identifier-1 is an index data item, it is set equal to either the contents of index-name-3 or identifier-3, where identifier-3 is also an index data item. Literal-1 cannot be used in this case.

3. If identifier-1 is not an index data item, it is set to an occurrence number that corresponds to the value of index-name-3. Neither identifier-3 nor literal-1 can be used in this case.

Format 2 Considerations -- When the SET statement is executed, the contents of index-name-4 (and index-name-5, etc., if present) are incremented (UP BY) or decremented (DOWN BY) by a value that corresponds to the number of occurrences represented by the value of literal-2 or identifier-4.

SAMPLE TABLE HANDLING PROGRAM

The program in Figure 18 illustrates the Table Handling feature, including the use of indexing, of the SET statement, and of the SEARCH statement (including the VARYING option and the SEARCH ALL format).

The census bureau uses the program to compare:

1. The number of births and deaths that occurred in any one of the 50 states in any one of the past 20 years with
2. The total number of births and deaths that occurred in the same state over the entire 20-year period

The input file, INCARDS, contains the specific information upon which the search of the table is to be conducted. INCARDS is formatted as follows:

STATE-NAME	a 4-character alphabetic abbreviation of the state name
M-F-CODE	1 = male; 2 = female
YEARCODE	a 4-digit field in the range 1950 through 1969

A typical run might determine the number of females born in New York in 1953 as compared with the total number of females born in New York in the past 20 years.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  TABLES.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.  IBM-360.
OBJECT-COMPUTER.  IBM-360.
SPECIAL-NAMES.  CONSOLE IS TYPEWRITER.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INFILE ASSIGN TO UT-2400-S-INTAPE.
    SELECT OUTFILE ASSIGN TO UR-S-PRTOUT.
    SELECT INCARDS ASSIGN TO UR-S-ICARDS.
DATA DIVISION.
FILE SECTION.
FD  INFILE LABEL RECORDS ARE OMITTED.
01  TABLE-1 PIC X(28200).
01  TABLE-2 PIC X(1800).
FD  OUTFILE LABEL RECORDS ARE OMITTED.
01  PRTLINE  PIC X(133).
FD  INCARDS LABEL RECORDS ARE OMITTED.
01  CARDS.
    05  STATE-NAME  PIC X(4).
    05  M-F-CODE  PIC 9.
    05  YEARCODE  PIC 9(4).
    05  FILLER    PIC X(71).
WORKING-STORAGE SECTION.
01  PRTAREA-20.
    05  FILLER    PIC X VALUE SPACES.
    05  YEARS-20  PIC 9(4).
    05  FILLER    PIC X(3) VALUE SPACES.
    05  BIRTHS-20 PIC 9(7).
    05  FILLER    PIC X(3) VALUE SPACES.
    05  DEATHS-20 PIC 9(7).
    05  FILLER    PIC X(108) VALUE SPACES.
```

Figure 18. Sample Table Handling Program (Part 1 of 2)

```
01  PRTAREA.
    05  FILLER     PIC X.
    05  YEAR       PIC 9(4).
    05  FILLER     PIC X(3) VALUE SPACES.
    05  BIRTHS     PIC 9(5).
    05  FILLER     PIC X(3) VALUE SPACES.
    05  DEATHS     PIC 9(5).
    05  FILLER     PIC X(112) VALUE SPACES.
01  CENSUS-STATISTICS-TABLE.
    05  STATE-TABLE OCCURS 50 TIMES INDEXED BY ST.
        10  STATE-ABBREV     PIC X(4).
        10  M-F OCCURS 2 TIMES INDEXED BY SE.
            15  STATISTICS OCCURS 20 TIMES ASCENDING KEY IS YEAR
                INDEXED BY YR.
                20  YEAR     PIC 9(4).
                20  BIRTHS   PIC 9(5).
                20  DEATHS   PIC 9(5).
01  STATISTICS-LAST-20-YRS.
    05  M-F-20 OCCURS 2 TIMES  INDEXED BY SE-20.
        10  STATE-20 OCCURS 50 TIMES INDEXED BY ST-20.
            15  YEARS-20     PIC 9(4).
            15  BIRTHS-20    PIC 9(7).
            15  DEATHS-20    PIC 9(7).
PROCEDURE DIVISION.
OPEN-FILES.
    OPEN INPUT INFILE INCARDS OUTPUT OUTFILE.
READ-TABLE.
    READ INFILE INTO CENSUS-STATISTICS-TABLE
        AT END GO TO READ-CARDS.
    READ INFILE INTO STATISTICS-LAST-20-YRS
        AT END GO TO READ-CARDS.
READ-CARDS.
    READ INCARDS
        AT END GO TO EOJ.
DETERMINE-ST.
    SET ST ST-20 TO 1.
    SEARCH STATE-TABLE VARYING ST-20 AT END GO TO ERROR-MSG-1
        WHEN STATE-NAME = STATE-ABBREV (ST) NEXT SENTENCE.
DETERMINE-SE.
    SET SE SE-20 TO M-F-CODE.
DETERMINE-YR.
    SEARCH ALL STATISTICS AT END GO TO ERROR-MSG-2
        WHEN YEAR OF STATISTICS (ST, SE, YR) = YEARCODE
            GO TO WRITE-RECORD.
ERROR-MSG-1.
    DISPLAY "INCORRECT STATE "  STATE-NAME UPQN TYPEWRITER.
    GO TO READ-CARDS.
ERROR-MSG-2.
    DISPLAY "INCORRECT YEAR "   YEARCODE UPON TYPEWRITER.
    GO TO READ-CARDS.
WRITE-RECORD.
    MOVE CORRESPONDING STATISTICS (ST, SE, YR) TO PRTAREA.
    WRITE PRTLINE FROM PRTAREA AFTER ADVANCING 3.
    MOVE CORRESPONDING STATE-20 (SE-20, ST-20) TO PRTAREA-20.
    WRITE PRTLINE FROM PRTAREA-20 AFTER ADVANCING 1.
    GO TO READ-CARDS.
EOJ.
    CLOSE INFILE INCARDS OUTFILE.
    STOP RUN.
```

Figure 18. Sample Program for the Table Handling Feature (Part 2 of 2)

SEGMENTATION FEATURE

The Segmentation Feature allows the problem programmer to communicate with the compiler to specify object program overlay requirements. The segmentation feature permits segmentation of procedures only. The Procedure Division and Environment Division are considered in determining segmentation requirements for an object program.

ORGANIZATION

Although it is not mandatory, the Procedure Division for a source program is usually written as several consecutive sections, each of which is composed of a series of closely related operations that are designed to perform collectively a particular function. However, when segmentation is used, the entire Procedure Division must be in sections. In addition, each section must be classified as belonging either to the fixed portion or to one of the independent segments of the object program. Segmentation in no way affects the need for qualification of procedure-names to ensure uniqueness.

FIXED PORTION

The fixed portion is defined as that part of the object program that is logically treated as if it were always in computer storage. This portion of the program is composed of two types of computer storage segments, permanent segments and overlayable fixed segments.

A permanent segment is a segment in the fixed portion that cannot be overlaid by any other part of the program.

An overlayable fixed segment is a segment in the fixed portion which, although logically treated as if it were always in storage, can be overlaid (if necessary) by another segment to optimize storage utilization. However, such a segment, if called for by the program, is always made available in the state it was in when it was last used.

Depending on the availability of storage, the number of permanent segments in the fixed portion can be varied through the use of a special facility called SEGMENT-LIMIT, which is discussed in "Structure of Program Segments."

INDEPENDENT SEGMENTS

An independent segment is defined as that part of the object program which can overlay, and be overlaid by, either an overlayable fixed segment or another independent segment. An independent segment is always considered to be in its initial state each time it is made available to the program.

SEGMENT CLASSIFICATION

Sections that are to be segmented are classified by means of a system of priority numbers. The following criteria should be used:

- Logical requirements: Sections that must be available for reference at all times, or which are referred to very frequently, are normally classified as belonging to one of the permanent segments; sections that are less frequently used are normally classified as belonging either to one of the overlayable fixed segments or to one of the independent segments, depending on logic requirements.
- Frequency of use: Generally, the more frequently a section is referred to, the lower its priority number should be; the less frequently it is referred to, the higher its priority number should be.
- Relationship to other sections: Sections that frequently communicate with one another should be given equal priority numbers. All sections with the same priority number constitute a single program segment.

SEGMENTATION CONTROL

The logical sequence of the program is the same as the physical sequence except for specific transfers of control. A reordering of the object module will be necessary if a given segment has its sections scattered throughout the source program. However, the compiler will provide transfers to maintain the logic flow of the source program. The compiler will also insert instructions necessary to load and/or initialize a segment when necessary. Control may be transferred within a source program to any paragraph in a section; that is, it is not mandatory to transfer control to the beginning of a section.

STRUCTURE OF PROGRAM SEGMENTS

PRIORITY NUMBERS

Section classification is accomplished by means of a system of priority numbers. The priority number is included in the section header.

Format
section-name SECTION [priority-number].

All sections that have the same priority-number constitute a program segment with that priority.

The priority-number must be an integer ranging in value from 0 through 99.

Segments with priority-numbers 0 through 49 belong to the fixed portion of the object program.

Segments with priority-numbers 50 through 99 are independent segments.

Sections in the declaratives portion of the Procedure Division must not contain priority-numbers in their section headers. They are treated as fixed segments with a priority-number of zero.

If the priority-number is omitted from the section header, the priority is assumed to be zero.

When a procedure-name in an independent segment is referred to by a PERFORM statement contained in a segment with a different priority number, the segment referred to is made available in its initial state for each execution of the PERFORM statement.

SEGMENT LIMIT

Ideally, all program segments having priority-numbers ranging from 0 through 49 are treated as permanent segments. However, when insufficient storage is available to contain all permanent segments plus the largest overlayable segment, it becomes necessary to decrease the number of permanent segments. The SEGMENT-LIMIT feature provides the user with a means by which he can reduce the number of permanent segments in his program, while these permanent segments still retain the logical properties of fixed portion segments (priority numbers 0 through 49).

Format
[SEGMENT-LIMIT IS priority-number]

The SEGMENT-LIMIT clause is coded in the OBJECT-COMPUTER paragraph.

Priority-number must be an integer that ranges in value from 1 through 49.

When the SEGMENT-LIMIT clause is specified, only those segments having priority-numbers from 0 up to, but not including, the priority number designated as the segment limit are considered as permanent segments of the object program.

Those segments having priority numbers from the segment limit through 49 are considered as overlayable fixed segments.

When the SEGMENT-LIMIT clause is omitted, all segments having priority-numbers from 0 through 49 are considered to be permanent segments of the object program.

RESTRICTIONS ON PROGRAM FLOW

When segmentation is used, the following restrictions are placed on the ALTER and PERFORM statements, and called programs:

ALTER Statement

1. A GO TO statement in a section whose priority number is 50 or higher must not be referred to by an ALTER statement in a section with a different priority number.

2. A GO TO statement in a section whose priority number is lower than 50 may be referred to by an ALTER statement in any section, even if the GO TO statement to which the ALTER refers is in a segment of the program that has not yet been called for execution.

PERFORM Statement

1. A PERFORM statement that appears in a section whose priority number is lower than the segment limit can have within its range only the following:

   a. Sections with priority numbers lower than 50.

   b. Sections wholly contained in a single segment whose priority number is higher than 49.

However, this compiler allows the PERFORM to have within its range sections with any priority numbers.

2. A PERFORM statement that appears in a section whose priority number is equal to or higher than the segment limit can have within its range only the following:

   a. Sections with the same priority number as the section containing the PERFORM statement.

   b. Sections with priority numbers that are lower than the segment limit.

However, this compiler allows the PERFORM to have within its range sections with any priority numbers.

When a procedure-name in a permanent segment is referred to by a PERFORM statement in an independent segment, the independent segment is reinitialized upon exit from the performed procedures.

Called Programs

A called program may not be segmented.

SOURCE PROGRAM LIBRARY FACILITY

Prewritten source program entries can be included in a source program at compile time. Thus, an installation can use standard file descriptions, record descriptions, or procedures without recoding them. These entries and procedures are contained in user-created libraries; they are included in a source program by means of a COPY statement.

COPY Statement

The COPY statement permits the user to include prewritten Data Division entries, Environment Division clauses, and Procedure Division procedures in his source program.

Format

```
COPY library-name [SUPPRESS]

     [REPLACING word-1  BY  { word-2      }
                            { literal-1   }
                            { identifier-1}

     [word-3 BY  { word-4      }     ]...].
                 { literal-2   }
                 { identifier-2}
```

No other statement or clause may appear in the same entry as the COPY statement, with the exception of the report description entry and the FILE-CONTROL paragraph.

When the library text is copied from the library, compilation is the same as though the text were actually part of the source program.

The COPY statement processing is terminated by the end of the library text.

The text contained in the library must not contain any COPY statements.

General Format
Option 1 (within the Configuration Section): SOURCE-COMPUTER. COPY statement. OBJECT-COMPUTER. COPY statement. SPECIAL-NAMES. COPY statement.
Option 2 (within the Input-Output Section): FILE-CONTROL. COPY statement. I-O-CONTROL. COPY statement.
Option 3 (within the FILE-CONTROL Paragraph): SELECT file-name COPY statement.
Option 4 (within the File Section): FD file-name COPY statement. SD sort-file-name COPY statement.
Option 5 (within the Report Section): RD report-name COPY statement. RD report-name [WITH CODE mnemonic-name] COPY statement.
Option 6 (within a File or Sort description entry, or within the Working-Storage Section or the Linkage Section): 01 data-name COPY statement.
Option 7 (with a Report Group): 01 [data-name] COPY statement.
Option 8 (within the Working-Storage Section or the Linkage Section): 77 data-name COPY statement.
Option 9 (within the Working-Storage Section or the Linkage Section): 01 data-name-1 REDEFINES data-name-2 COPY statement. 77 data-name-1 REDEFINES data-name-2 COPY statement.
Option 10 (within the Procedure Division): section-name SECTION [priority-number]. COPY statement. paragraph-name. COPY statement.
Program Product Information (Version 4) Option 11 (within the Communication Section): CD cd-name COPY statement.

Library-name is the name of a member of a partitioned data set contained in the user's library; it identifies the library subroutine to the control program. Library-name must follow the rules of formation for a program-name. The first eight characters are used as the identifying name.

The words preceding COPY conform to margin restrictions for COBOL programs. On a given source program card containing the completion of a COPY statement, there must be no information beyond the statement terminating period. The material introduced into the source program by the COPY statement will follow the COPY statement on the listing beginning on the next line. However, the SUPPRESS option may be used to indicate that the library entry is not to be listed.

If the REPLACING option is used, each word specified in the format is replaced by the stipulated word, identifier, or literal which is associated with it in the format.

Word-1, word-2, etc., may be a data-name, procedure-name, condition-name, mnemonic-name, or file-name.

Use of the REPLACING option does not alter the material as it appears in the library.

When options 1, 2, 3, 4, 5, or 10 are written, the words COPY library-name are replaced by the information identified by library-name. This information comprises the sentences or clauses needed to complete the paragraph, sentence, or entry containing the COPY statement.

When options 6, 7, 8, or 9 are written, the entire entry is replaced by the information identified by library-name, except that data-name (if specified) replaces the corresponding data-name from the library.

Program Product Information (Version 4)

When Option 11 is written, the words COPY library-name are replaced by the information identified by library-name. This information comprises the clauses needed to complete the CD entry containing the COPY statement.

For example, if the library entry PAYLIB consists of the following Data Division record:

```
01  A.
    05  B PIC S99.
    05  C PIC S9(5)V99.
    05  D PIC S9999 OCCURS 0 TO 52 TIMES
          DEPENDING ON B OF A.
```

the programmer can use the COPY statement in the Data Division of his program as follows:

```
 01  PAYROLL COPY PAYLIB.
```

In this program, the library entry is then copied; the resulting entry is treated as if it had been written as follows:

```
01  PAYROLL.
    05  B PIC S99.
    05  C PIC S9(5)V99.
    05  D PIC S9999 OCCURS 0 TO 52 TIMES
          DEPENDING ON B OF A.
```

Note that the data-name A has not been changed in the DEPENDING ON option.

To change some (or all) of the names within the library entry to names he wishes to reference in his program, the programmer can use the REPLACING option:

```
01  PAYROLL COPY PAYLIB REPLACING A BY PAYROLL
        B BY PAY-CODE C BY GROSSPAY.
```

In this program the library entry is then copied; the resulting entry is treated as if it had been written as follows:

```
01  PAYROLL.
    05  PAY-CODE PIC S99.
    05  GROSSPAY PIC S9(5)V99.
    05  D PIC S9999 OCCURS 0 TO 52 TIMES
          DEPENDING ON PAY-CODE OF PAYROLL.
```

The entry as it appears in the library remains unchanged.

Program Product Information (Version 3 and Version 4)

A sequence number may appear in columns 1 through 6 of a COPY card.

EXTENDED SOURCE PROGRAM LIBRARY FACILITY

A complete program may be included as an entry in the user's library, and may be used as the basis of compilation. Input to the compiler is a BASIS card, followed by any number of INSERT and/or DELETE cards, followed by any number of debugging packets, if desired. These packets can be requested and modified through INSERT and DELETE cards (see "Debugging Language").

Program Product Information (Version 3 and Version 4)

On BASIS, INSERT, and DELETE cards, a sequence number may appear in columns 1 through 6.

BASIS Card

Format
BASIS library-name

The word BASIS followed by library-name may appear anywhere within columns 1 through 72 on the card. There must be no other text on the card.

Library-name must follow the rules of formation for program-name; it is the name by which the library entry is known to the control program. The first eight characters are used as the identifying name.

If the INSERT or DELETE cards follow the BASIS card, the library entry is modified prior to being processed by the compiler. Use of INSERT or DELETE cards does not alter the material in the library.

INSERT Card

Format
INSERT sequence-number-field

DELETE Card

Format
DELETE sequence-number-field

The word INSERT or DELETE, followed by a space, followed by sequence-number-field may appear anywhere within columns 1 through 72 on the card. There must be no other text on the card.

Each number in the sequence-number-field must refer to a sequence number of the basic library entry. The sequence number is the 6-digit number the programmer assigns in columns 1 through 6 of the COBOL coding form.

The numbers specified in the sequence-number-field must be in ascending numerical order from the first INSERT/DELETE card to the last INSERT/DELETE card in the program.

The sequence-number-field of an INSERT card must be a single number (e.g., 000310). At least one new source program card must follow the INSERT card for insertion after the card specified by the sequence-number-field.

The entries comprising sequence-number-field of a DELETE card must be numbers or ranges of numbers. Each entry must be separated from the preceding entry by a comma followed by a space. Ranges of numbers are indicated by separating the two bounding numbers of the range by a hyphen. For example:

```
000001-000005, 000010
```

Source program cards may follow a DELETE card, for insertion before the card following the last one deleted.

The following statements are provided for program debugging. They may appear anywhere in a COBOL program or in a compile-time debugging packet.

For the TRACE and EXHIBIT statements, the output is written on the system logical output device (SYSOUT). A maximum logical record size of 120 characters is assumed. This assumed size is overridden if a logical record size is specified on the associated SYSOUT DD statement.

READY/RESET TRACE Statement

Format

```
{READY}
{RESET}    TRACE
```

After a READY TRACE statement is executed, each time execution of a paragraph or section begins, its compiler-generated card number is displayed. The execution of a RESET TRACE statement terminates the functions of a previous READY TRACE statement.

EXHIBIT Statement

Format

```
                {NAMED          }  {identifier-1         }
EXHIBIT         {CHANGED NAMED  }  {                     }
                {CHANGED        }  {nonnumeric-literal-1 }

     [identifier-2         ]
     [                     ] ...
     [nonnumeric-literal-2 ]
```

The execution of an EXHIBIT statement causes a formatted display of the identifiers (or nonnumeric literals) listed in the statement.

Identifiers listed in the statement cannot be any special register except TALLY.

Program Product Information (Version 3 and Version 4)

Identifiers listed in the statement cannot be specified using relative indexing. (That is, where INX is an index-name for TABLE-A, the following statement is invalid: EXHIBIT NAMED TABLE-A (INX + 2).)

Nonnumeric-literals listed in the statement are followed by a blank when displayed.

The display of the operands is continued as described for the DISPLAY statement. A maximum logical record size of 120 characters is assumed.

EXHIBIT NAMED: Each time an EXHIBIT NAMED statement is executed, there is a formatted display of each identifier listed and its value. Since both the identifying name and the value of the identifer are displayed, a fixed columnar format is unnecessary. If the list of operands includes nonnumeric-literals, they are displayed as remarks each time the statement is executed.

The format of the output for each identifier listed in the EXHIBIT NAMED statement is:

```
original identifying name, including qualifiers if written
     (no more than 120 characters in length)
space
equal sign
space
value of identifier (no more than 256 bytes in length)
space
```

EXHIBIT CHANGED NAMED: Each time an EXHIBIT CHANGED NAMED statement is executed, there is a display of each identifier listed and its value only if the value has changed since the previous time the statement was executed. The initial time such a statement is executed, all values are considered changed and are displayed. If the list of operands includes nonnumeric-literals, they are displayed as remarks each time the statement is executed.

Since both the identifying name and the value of each identifier is displayed, a fixed columnar format is unnecessary. If some of the identifiers have not changed in value, no space is reserved for them. If none of the identifiers have changed in value, no blank line(s) will be printed.

The format of the output for each identifier listed in the EXHIBIT CHANGED NAMED statement is:

```
original identifying name, including qualifiers if written
     (no more than 120 characters in length)
space
equal sign
space
value of identifier (no more than 256 bytes in length)
space
```

EXHIBIT CHANGED: Each time an EXHIBIT CHANGED statement is executed, there is a display of the current value of each identifier listed only if the value has changed since the previous time the statement was executed. The initial time the statement is executed, all values are considered changed and are displayed. If the list of operands includes nonnumeric-literals, they are printed as remarks each time the statement is executed.

The format of the output for a specific EXHIBIT CHANGED statement presents each operand in a fixed columnar position. Since the operands are displayed in the order they are listed in the statement, the programmer can easily distinguish each operand.

The following considerations apply:

- If there are two or more identifiers as operands, and some, but not all, are changed from the previous execution of the statement, only the changed values are displayed. The positions reserved for a given operand are blank when the value of the operand has not changed.
- If none of the operands have changed in value from the previous execution of the statement, a blank line(s) will be printed.
- Variable length identifiers are not permitted as operands.
- The storage reserved for any operand cannot exceed 256 bytes.

Note: The combined total length of all operands for all EXHIBIT CHANGED NAMED plus all EXHIBIT CHANGED statements in one program cannot exceed 32,767 bytes.

If two distinct EXHIBIT CHANGED NAMED or two EXHIBIT CHANGED statements appear in one program, each specifying the same identifiers, the changes in value of those identifiers are associated with each of the two separate statements. Depending on the path of program flow, the values of the identifier saved for comparison may differ for each of the two statements.

ON (Count-conditional) Statement

The ON statement allows the programmer to specify when the statements it contains are to be executed.

Format 1

```
ON integer-1 [AND EVERY integer-2] [UNTIL integer-3]

{ imperative-statement } { ELSE      } { statement ... }
{ NEXT SENTENCE       } { OTHERWISE } { NEXT SENTENCE }
```

Format 2 (Version 3 and Version 4)

```
ON { integer-1    } [AND EVERY { integer-2    } ]
   { identifier-1 }            { identifier-2 }

   [UNTIL { integer-3    } ] { imperative-statement }
          { identifier-3 }   { NEXT SENTENCE        }

{ ELSE      } { statement...  }
{ OTHERWISE } { NEXT SENTENCE }
```

All integers contained in the ON statement must be positive and no greater than 16,777,215.

The phrase ELSE/OTHERWISE NEXT SENTENCE may be omitted if it immediately precedes the period for the sentence.

Program Product Information (Version 3 and Version 4)

Format 2: All identifiers must be fixed-point numeric items described as integers. Their values must be positive and no greater than 16,777,215.

At object time each identifier must be initialized to a positive value before the first execution of the ON statement. Between executions of the ON statement, the values contained in the identifiers may be modified. The programmer's manipulation of these values in no way affects the compiler-generated counter associated with the ON statement.

In the discussion that follows, each reference to integer-1 applies equally to identifier-1. Similarly, each reference to integer-2 applies to identifier-2, and each reference to integer-3 applies to identifier-3.

In Format 1 and Format 2 the ON statement is evaluated and executed as follows:

- Each ON statement has a compiler-generated counter associated with it. The counter is initialized to zero in the object program. Each time the path of program flow reaches the ON statement, the counter is incremented by one, and the count-condition (integer-1 AND EVERY integer-2 UNTIL integer-3) is tested.
- If the count-condition is satisfied, the imperative-statement (or NEXT SENTENCE) preceding ELSE/OTHERWISE is executed. (Note that an imperative-statement may consist of a series of imperative statements.)
- If the count-condition is not satisfied, the statement(s) (or NEXT SENTENCE) following ELSE/OTHERWISE is executed. If the ELSE/OTHERWISE option does not appear, the next sentence is executed.

The count-condition is evaluated as follows:

- If only integer-1 has been specified, then the count-condition is satisfied only once: when the path of program flow has reached the ON statement integer-1 times -- that is, when the value in the counter equals integer-1.
- When only integer-1 and integer-3 are specified, then the value of integer-2 is assumed to be one, and the count-condition is satisfied when the value in the counter is any value within the range integer-1 through integer-3.
- If only integer-1 and integer-2 are specified, then the count-condition is satisfied each time the value in the counter is equal to integer-1 + (integer-2 * K), where K is any positive integer or zero. No upper limit for the execution of the ON statement is assumed.
- When all three integers are specified, then the count-condition is satisfied as in the last preceding case, except that an upper limit beyond which the count-condition cannot be satisfied is specified. The upper limit is integer-3.

COMPILE-TIME DEBUGGING PACKET

Debugging statements for a given paragraph or section in a program may be grouped together into a debugging packet. These statements will be compiled with the source language program and will be executed at object time. Each packet refers to a specified paragraph-name or section-name in the Procedure Division. Compile-time debugging packets are grouped together and are placed immediately following the source program. No reference to procedure-names in debug packets may be made in the body of the program.

DEBUG Card

Each compile time debug packet is headed by the control card DEBUG.

Format
DEBUG location

The word DEBUG followed by location may appear anywhere within columns 1 through 72 on the card. There must be no other text on the card.

The location is the section-name or paragraph-name (qualified, if necessary) indicating the point in the program at which the packet is to be executed. Effectively, the statements in the packet are executed as though they were physically placed in the source program following the section-name or paragraph-name, but preceding the text associated with the procedure. The same location must not be used in more than one DEBUG control card. Location cannot be a paragraph-name within any debug packet.

A debug packet may consist of any procedural statements conforming to the requirements of American National Standard COBOL. The following considerations apply:

- A PERFORM or ALTER statement in a debug packet may refer to a procedure-name in any debug packet or in the main body of the Procedure Division.
- A GO TO statement in a debug packet may not refer to a procedure-name in another debug packet, but it may refer to a procedure-name in the main body of the Procedure Division.

Program Product Information (Version 3 and Version 4)

On the DEBUG card, the sequence number may appear in columns 1 through 6 followed by at least one space; in this case, the word DEBUG may not begin before column 8.

FORMAT CONTROL OF THE SOURCE PROGRAM LISTING

There are four statements that allow the programmer to control the spacing of the source program listings produced by the COBOL compiler. These statements are: EJECT, SKIP1, SKIP2, and SKIP3. They may be written anywhere in the source program.

EJECT Statement

The EJECT statement instructs the compiler to print the next source statement at the top of the next page.

Format
1 Area B EJECT

The word EJECT may be written anywhere within Area B and must be the only statement on the card. There must be no punctuation.

SKIP1, SKIP2, and SKIP3 Statements

These statements instruct the compiler to skip 1, 2, or 3 lines before printing the next source statement.

Format
1 Area B { SKIP1 } { SKIP2 } { SKIP3 }

SKIP1 tells the compiler to skip 1 line (double spacing).

SKIP2 tells the compiler to skip 2 lines (triple spacing).

SKIP3 tells the compiler to skip 3 lines (quadruple spacing).

SKIP1, SKIP2, or SKIP3 may be written anywhere within Area B and must be the only statement on the card. There must be no punctuation.

STERLING CURRENCY FEATURE AND INTERNATIONAL CONSIDERATIONS

COBOL provides facilities for handling sterling currency items by means of an extension of the PICTURE clause. Additional options and formats, necessitated by the nondecimal nature of sterling and by the conventions by which sterling amounts are represented in punched cards, are also available.

COBOL provides a means to express sterling currency in pounds, shillings, and pence, in that order. There are 20 shillings in a pound, and 12 pence in a shilling. Although sterling amounts are sometimes expressed in shillings and pence only (in which case the number of shillings may exceed 99), within machine systems, shillings will always be expressed as a 2-digit field. Pence, when in the form of integers, likewise will be expressed as a 2-digit field. However, provision must be made for pence to be expressed as decimal fractions as well, as in the form 17s.10.237d.

The IBM method for representing sterling amounts in punched cards uses two columns for shillings and one for pence. Tenpence (10d.) is represented by an '11' punch and elevenpence (11d.) by a '12' punch. The British Standards Institution (B.S.I.) representation uses single columns for both shillings and pence. The B.S.I. representation for shillings consists of a '12' punch for ten shillings and the alphabetic punches A through I for 11 through 19 shillings, respectively.

Note: The B.S.I. representation for shillings precludes the use of more than 19 shillings in a sterling expression; therefore, 22/10 (that is, 22 shillings 10 pence) must be expanded by the user to 1/2/10. Similarly, the guinea -- 21 shillings -- or any multiple thereof, must be expanded to pounds and shillings.

The indicated representations may be used separately or in combination, resulting in four possible conventions.

1. IBM shillings and IBM pence
2. IBM shillings and B.S.I. pence
3. B.S.I. shillings and IBM pence
4. B.S.I. shillings and B.S.I. pence

Any of these conventions may be associated with any number of digits (or none) in the pound field and any number of decimal places (or none) in the pence field. In addition, sign representation may be present as an overpunch in one of several allowable positions in the amount, or may be separately entered from another field.

Two formats are provided in the PICTURE clause for the representation of sterling amounts: sterling report format (used for editing) and sterling nonreport format (used for arithmetic).

In the formats that follow, n stands for a positive integer other than zero. This integer enclosed in parentheses and following the symbols 9, B, etc., indicates the number of consecutive occurrences of the preceding symbol. For example, 9(6) and 999999 are equivalent. The PICTURE characters used to describe sterling items are:

6 7 8 9 C D * , / B Z V . £ : s d CR DB + -

(The character £ is the sterling equivalent of the character $.)

Note: The lower-case letters "s" and "d" are represented by an 11-0-2 punch and a 12-0-4 punch, respectively.

STERLING NONREPORT

The format of the PICTURE clause for a sterling nonreport data item is:

Format
{PICTURE / PIC} IS 9[(n)]D[8]8D {6[6] / 7[7]}
[[V]9[(n)]] [USAGE IS] DISPLAY-ST

Note: For a sterling nonreport picture to be valid, it must contain a pound field, a shilling field, and a pence field.

The representation for pounds is 9[(n)]D where:

1. The character 9 indicates that a character position will always contain a numeric character, and may extend to n positions.
2. The character D indicates the position of an assumed pound separator.

The representation for shillings is [8]8D where:

1. The characters [8]8 indicate the position of the shilling field and the convention by which shillings are represented in punched cards. 88 indicates IBM shilling representation occupying a 2-column field, the character 8 indicates B.S.I. single-column shilling representation.
2. The character D indicates the position of an assumed shilling separator.

The representation for pence is:

{6[7] / 7[7]} [[V]9[(n)]]

1. The character 6 indicates IBM single-column pence representation wherein 10d. is represented by an '11' punch and 11d. by a '12' punch. The characters 66 indicate 2-column representation of pence, usually from some external medium other than punched cards.
2. The character 7 indicates B.S.I. single-column pence representation wherein 10d. is represented by a '12' punch and 11d. by an '11' punch. The characters 77 indicate 2-column representation of pence. Consequently, 66 and 77 serve the same purpose and are interchangeable.

3. The character V indicates the position of an assumed decimal point in the pence field. Its properties and use are identical with that of V in dollar amounts. Decimal positions in the pence field may extend to n positions.

4. The character 9 indicates that a character position will always contain a numeric character, and may extend to n positions.

Example: Assume that a sterling currency data item used in arithmetic expressions is to be represented in IBM shillings and IBM pence, and that this data item will never exceed 99/19s/11d. Its picture should be:

```
PICTURE 9(2)D88D6 DISPLAY-ST.
```

The VALUE clause must not be specified for sterling nonreport items.

Sterling Sign Representation

Signs for sterling amounts may be entered as overpunches in one of several allowable positions of the amount. A sign is indicated by an embedded S in the nonreport PICTURE immediately to the left of the position containing the overpunch. Allowable overpunch positions are the high-order and low-order positions of the pound field, the high-order shilling digit in 2-column shilling representation, the low-order pence digit in 2-column pence representation, or the least significant decimal position of pence.

The following are examples of sterling currency nonreport data items showing sign representation in each of the allowable positions:

```
PICTURE S99D88D6V9(3) DISPLAY-ST

PICTURE 9S9D88D6V9(3) DISPLAY-ST

PICTURE 9(2)DS88D6V9(3) DISPLAY-ST

PICTURE 9(2)D88D6S6V9(3) DISPLAY-ST

PICTURE 9(2)D88D6V99S9 DISPLAY-ST
```

STERLING REPORT

The sterling currency report data item is composed of four portions: pounds, shillings, pence, and pence decimal fractions.

```
┌──────────────────────────────────────────────────────────────────┐
│                              Format                              │
├──────────────────────────────────────────────────────────────────┤
│                                                                  │
│ {PICTURE}                                                        │
│ {PIC    }  IS                                                    │
│                                                                  │
│     [pound-report-string][pound-separator-string]   delimiter    │
│                                                                  │
│     shilling-report-string   [shilling-separator-string]   delimiter
│                                                                  │
│     pence-report-string   [pence-separator-string][sign-string]  │
│                                                                  │
│    [USAGE IS] DISPLAY-ST                                         │
│                                                                  │
└──────────────────────────────────────────────────────────────────┘
```

Pound-Report-String - This string is optional. It is subject to the same rules as other numeric edited items, with the following exceptions:

- The allowable characters are: £ (pound symbol) 9 Z * + - 0 (zero) B , (comma).
- The total number of digits in the pound-report-string plus the fractional-pence field cannot exceed 15. (That is, if there are 11 digits in the pound-report-string, there cannot be more than four digits in the fractional-pence-field.)
- The character £ is the sterling equivalent of $.
- Termination is controlled by the pound-separator-string.

Pound-Separator-String - This string is optional. It may include one character, or any combination of the following characters:

B : / . (period or decimal point)

Editing of the separator characters is dependent upon the use of C or D as delimiters.

The Delimiter Characters - The delimiter characters C and D are required. They primarily serve to indicate the end of the pounds and shillings portions of the picture. In addition, they serve to indicate the type of editing to be applied to separator characters to the right of the low-order digit (of the pounds and shillings integer portions of the item).

The delimiter character D indicates that separator character(s) to the right of the low-order digit position (of the field delimited) are always to appear; that is, no editing is performed on the separator character(s).

The delimiter character C indicates that if the low-order digit position (of the field delimited) is represented by other than the edit character 9, editing continues through the separator character(s).

The delimiter characters C and D are used for editing purposes only. They do not take up space in the printed result.

The following examples show the editing performed when a value of zero is moved to a sterling report item.

```
**/CZ9s/D99d
```

would result in

```
***b0s/00d
```

whereas, if the picture were

```
**/DZ9s/D99d
```

the result would be

```
**/b0s/00d
```

The delimiter C is equivalent to D when the low-order digit position is represented by a 9. That is, the following two pictures are equivalent:

```
ZZ9/DZ9/D99
ZZ9/CZ9/C99
```

The delimiters used for the pounds and shillings portion of the picture need not be the same.

Note: Although the pound-report-string and the pound-separator-string are optional, a delimiter character (either C or D) must be present; thus, when programming for shillings and pence only, the PICTURE clause must begin PICTURE IS C (or D).

Shilling-Report-String - This is a required two-character field. It is made up of the following characters:

```
9   8   Z   *
```

The valid combinations of these characters are:

```
99   Z9   ZZ   Z8   *9   **   *8
```

The 8 is an edit character and is treated as a 9. However, if the digits to the left of the edit character 8 are zeros, the 8 is treated as the character that precedes it (either Z or *).

Shilling-Separator-String - This string is optional. It may include one character, or any combination of the following characters:

```
B   :   /   s   .  (period or decimal point)
```

Editing of the shilling-separator characters is dependent upon the use of C or D as delimiters.

Pence-Report-String - This field is made up of two parts: a required whole-pence field, and an optional fractional-pence field.

The required whole-pence field is a two-character field, made up of the following symbols:

9 8 Z *

Valid combinations of these characters are:

99 Z9 ZZ Z8 *9 ** *8

The function of the editing character 8 is the same as in the shilling-report-string.

The optional fractional-pence field is indicated by a decimal point followed by one or more 9's. It is used to specify fractional pence in decimal form.

The total number of digits in the fractional-pence field plus the pound-report-string cannot exceed 15.

Pence Separator-String - This string is optional and may consist of one or both of the following characters:

d . (period or decimal point)

If both characters are used, they must be used in the order shown above.

Sign-Field - This field is optional and may consist of:

- optionally, one or more blanks (B), followed by
- one of the following one- or two-character combinations:

  \+ - CR DB

Sterling Report editing applications are shown in Table 26.

Table 26. Sterling Currency Editing Applications

Picture	Numeric Value (in pence)	Sterling Equivalent £ s d	Printed Result
£££ /D99s/D99d	3068	12 15 08	£12/15s/08d
£££ /D99s/D99d	0668	2 15 08	£2/15s/08d
£££ /D99s/D99d	0188	0 15 08	/15s/08d
£££ /C99s/D99d	0188	0 15 08	15s/08d
ZZZ/DZZs/DZZd	0000	0 00 00	/ s/ d
ZZZ/CZZs/DZZd	0000	0 00 00	s/ d
£BD99sBD99.9d	080.5	0 06 08.5	06s 08.5d
***/C**D/C**.99d	1040.12	4 06 08.12	**4/*6s/*8.12d
***:C**s:C**.99d	080.12	0 06 08.12	*****6s:*8.12d
***/D**s/D**.99d	00001.23	0 00 01.23	***/**s/*1.23d
£££ /D*9s/D**.99d	00961.23	4 00 01.23	£4/*0s/*1.23d
£**/D*9s/D**.99d	00961.23	4 00 01.23	£*4/*0s/*1.23d
£**/D*9s/D**.99d	00001.23	0 00 01.23	£**/*0s/*1.23d
£££ /D99s/D99dCR	-3068	12 15 08	£12/15s/0dCR

A sterling report PICTURE may have a BLANK WHEN ZERO clause associated with it specifying that the item described is filled with spaces whenever the value of the item is zero.

If the VALUE clause is specified for a sterling report item, the literal must be alphanumeric. The VALUE clause is treated exactly as it is specified, with no editing performed.

The maximum length of a sterling report item is 127 characters.

If the VALUE clause is specified for a sterling report item, the literal must be alphanumeric.

PROCEDURE DIVISION CONSIDERATIONS

The MOVE, DISPLAY, ACCEPT, EXAMINE, and TRANSFORM statements, arithmetic statements, and relation tests may be written containing identifiers that represent sterling items.

Sterling items are not considered to be integers and should not be used where an integer is required.

INTERNATIONAL CONSIDERATIONS

1. The functions of the period and the comma may be exchanged in the PICTURE character-string and in numeric literals by writing the clause DECIMAL-POINT IS COMMA in the SPECIAL-NAMES paragraph of the Environment Division.

2. The PICTURE of report items may terminate with the currency symbol in cases where the graphic $ is supplanted by a particular national currency symbol, through use of the CURRENCY SIGN IS literal clause in the SPECIAL-NAMES paragraph of the Environment Division.

Program Product Information (Version 4)

TELEPROCESSING (TP)

The Teleprocessing (TP) Feature of the Version 4 Compiler permits the COBOL Programmer to create device-independent message processing programs for teleprocessing applications. A teleprocessing network consists of a central computer, remote (or local) station(s), and the communication lines connecting such station(s) to the central computer.

In TP applications, data flow into the system is random and proceeds at relatively slow speeds. Data in the system exists as messages from remote stations, or as messages generated by internal programs. Once delivered to the computer, the messages can be processed at computer speeds. Thus, TP applications require a Message Control Program (MCP) that acts as an interface between the COBOL program and the remote stations.

The MCP acts as the logical interface between the entire network of communications devices and the COBOL program, in much the same manner as the system acts as an interface between the COBOL object program and conventional input/output devices. The MCP also must perform device-dependent tasks such as character translation, and insertion of control characters, so that the COBOL program itself is device-independent. The MCP and the COBOL TP program operate asynchronously; that is, there is no fixed time relationship between the receipt of a message by the MCP and its subsequent processing by the COBOL TP program.

To store the messages until they are to be processed, the MCP uses destination queues, which may be thought of as sequential data sets. The queues act as buffers between the COBOL TP program and the remote stations. To the COBOL TP program, the MCP queue from which it accepts messages is logically an input queue; the queue into which it places messages is logically the output queue. In this publication, these terms are used with this meaning.

More detailed information about requirements for an MCP are given in the publication IBM OS Full American National Standard COBOL Compiler and Library, Version 4, Programmer's Guide, Order No. SC28-6456.

COMMUNICATION SECTION

The Communication Section of a COBOL program must be specified if the program is to utilize the TP features of COBOL. The Communication Section, through the definition of Communication Description (CD) entries, establishes the interface between the COBOL object program and the MCP.

The Communication Section is identified by, and must begin with the section header COMMUNICATION SECTION. The header is followed by Communication Description (CD) entries. Specification of the CD entry causes an implicitly defined data area to be created; that is, the generated data area has a fixed format. Level-01 record description entries may optionally follow the CD entry; these record description entries implicitly redefine the fixed data areas of the CD.

```
                        General Format

    COMMUNICATION SECTION.

    {communication description entry

    [record description entry]...}... .
```

When it is specified, the Communication Section should contain at least one CD entry. A single CD entry is sufficient if messages are only of one type, that is, only FOR INPUT or only FOR OUTPUT. If the COBOL TP program is to both receive and send messages, then at least two CD entries are required -- one FOR INPUT and one FOR OUTPUT. However, multiple input and/or output CD entries may be specified.

The CD entry is valid only in the Communication Section.

CD Entry

The CD entry represents the highest level of organization in the Communication Section. The Communication Section header is followed by CD entries, each consisting of a level indicator, a data-name, and a series of optional independent clauses.

```
                          Format 1

    CD cd-name FOR INPUT

       [[[SYMBOLIC QUEUE          IS data-name-1]
         [SYMBOLIC SUB-QUEUE-1    IS data-name-2]
         [SYMBOLIC SUB-QUEUE-2    IS data-name-3]
         [SYMBOLIC SUB-QUEUE-3    IS data-name-4]
         [MESSAGE DATE            IS data-name-5]
         [MESSAGE TIME            IS data-name-6]
         [SYMBOLIC SOURCE         IS data-name-7]
         [TEXT LENGTH             IS data-name-8]
         [END KEY                 IS data-name-9]
         [STATUS KEY              IS data-name-10]
         [QUEUE DEPTH             IS data-name-11]]

        [data-name-1 data-name-2 ... data-name-11]].
```

```
                          Format 2

    CD cd-name  FOR OUTPUT
         [DESTINATION COUNT       IS data-name-1]
         [TEXT LENGTH             IS data-name-2]
         [STATUS KEY              IS data-name-3]
         [ERROR KEY               IS data-name-4]
         [SYMBOLIC DESTINATION    IS data-name-5].
```

Format 3

```
CD  cd-name  COPY  library-name  [SUPPRESS]
      [REPLACING { word-1       } BY { word-2       }
                 { identifier-1 }    { identifier-2 }
        [ { word-3       } BY { word-4       } ] ... ].
          { identifier-3 }    { identifier-4 }
```

The CD entry serves as a storage area through which the COBOL program and the MCP interface. The COBOL programmer moves information about the message into the CD before initiating any request. The MCP, after acting upon the request, returns through the same CD information pertaining to the request.

The CD entry is defined in such a way that any number of message queues may be accessed through the same CD entry. Conversely, different portions of one message may be accessed through multiple CD entries in the same program or in different COBOL subprograms residing in the same region or partition. Thus, any one COBOL TP program need specify only one input CD entry and/or one output CD entry. Rules controlling the accessing of MCP queues are specified in the detailed descrptions of both input (Format 1) and output (Format 2) CD entries.

The level indicator CD identifies the beginning of a Communication Description entry, and must appear in Area A. It must be followed in Area B by cd-name. Cd-name follows the rules for formation of a data-name. Cd-name may be followed by a series of optional independent clauses (as shown in Format 1 and Format 2).

The optional clauses may be followed by an optional level-01 record description entry. This record description entry implicitly redefines that of the fixed data area described by the CD entry. The total length of the record description entry must be the same as or less than the fixed data descriptions of the CD entry; if it is not, an error message is produced. However, the MCP always references this data area according to the implicit data descriptions of the CD entry; that is, for an input CD the contents of positions 1 through 12 are always used as the symbolic queue, the contents of positions 13 through 24 are always used as symbolic sub-queue-1, and so forth.

The optional clauses of the CD entry may be written in any order. Since the data areas of both the input CD and the output CD have implicit definitions, the optional clauses are necessary only to assign user names for those areas that the COBOL program will reference. However, if all the options of either format are omitted, then a level-01 record description entry must follow the CD entry.

Except for a level-88 entry, the level-01 record description entry must not contain any VALUE clauses.

FORMAT 1: This format is required if the CD entry is FOR INPUT. At least one input CD entry must be specified if input messages are to be received from a queue. Any number of queues may be accessed through the same input CD entry. This is accomplished simply by moving a different symbolic queue name into the input CD.

Conversely, different portions of one message may be accessed through different CD entries. Thus, CD entries in the same or different COBOL subprograms in the same run unit may be used to access different portions of one message. The same CD entry may be used to access a message from another queue before the first message is completed. The following restrictions apply:

- Only one region (or partition) can have access to any particular queue at one time.
- The data in a queue must be accessed sequentially. That is, a second message in any queue cannot be accessed until the entire first message in that queue is accessed. However, a second message from another queue may be accessed before the entire message in the first queue is accessed.

The specification of an input CD entry results in a record whose implicit description is equivalent to the following:

```
Equivalent COBOL Record Description     Description of Use
01  data-name-0.
     02  data-name-1  PICTURE X(12).    Symbolic Queue
     02  data-name-2  PICTURE X(12)     Sub-queue-1
     02  data-name-3  PICTURE X(12)     Sub-queue-2
     02  data-name-4  PICTURE X(12)     Sub-queue-3
     02  data-name-5  PICTURE 9(6).     Message Date
     02  data-name-6  PICTURE 9(8).     Message Time
     02  data-name-7  PICTURE X(12).    Symbolic Source
     02  data-name-8  PICTURE 9(4).     Text Length
     02  data-name-9  PICTURE X.        End Key
     02  data-name-10 PICTURE XX.       Status Key
     02  data-name-11 PICTURE 9(6).     Queue Depth
```

For each input CD entry, a record area of 87 contiguous Standard Data Format characters is always generated, implicitly defined as previously specified. The data names corresponding to the various fields of the CD record area may be explicitly defined, through the use of the optional clauses as follows:

Format 1 -- Option 1: The data names corresponding to the various fields of the CD record area may be explicitly defined, through the use of the optional clauses as follows:

SYMBOLIC QUEUE and SUB-QUEUE Clauses: These clauses define data-name-1, data-name-2, data-name-3, and data-name-4 as the names of alphanumeric data items each of 12 characters in length, and occupying character positions within the record as follows:

```
data-name-1 occupies character positions  1 through 12
data-name-2 occupies character positions 13 through 24
data-name-3 occupies character positions 25 through 36
data-name-4 occupies character positions 37 through 48
```

The contents of the SYMBOLIC QUEUE can be specified as a queue structure. SUB-QUEUE-1, SUB-QUEUE-2, and SUB-QUEUE-3 specify the levels of such a structure. When a given level of such a structure is specified, all higher levels must also be specified. However, no given queue structure need specify all four levels.

For example, if only a three-level queue structure is needed for a given program, then the following COBOL statements adequately specify the levels of the structure:

```
SYMBOLIC QUEUE IS QNAME
     SYMBOLIC SUB-QUEUE-1 IS SUBQ1
          SYMBOLIC SUB-QUEUE-2 IS SUBQ2...
```

Since SYMBOLIC SUB-QUEUE-2 is specified, both SYMBOLIC SUB-QUEUE-1 and SYMBOLIC QUEUE must also be specified. (It would be invalid to specify SUB-QUEUE-2 without also specifying SUB-QUEUE-1.)

When symbolic sub-queues are used in the COBOL program, the associated queue structures must be predefined to the MCP. Queue structures are described in the publication OS Full American National Standard COBOL Compiler and Library, Version 4, Programmer's Guide, Order No. SC28-6456.

A RECEIVE statement causes the serial return of the next message (or portion of a message) from the queue specified in data-name-1, and, if SUB-QUEUE clauses are specified, from one of the sub-queues specified in data-name-2, data-name-3, or data-name-4.

Before the RECEIVE statement is executed, the data-name of the queue, and, if specified, of the sub-queue(s) must contain the symbolic name(s) of the wanted queue. All such symbolic names must be previously defined to the MCP. The compiler initializes the sub-queue names to SPACES; if a sub-queue has been accessed, then it is the responsibility of the user to reinitialize each sub-queue name that is not to be used to SPACES.

When the RECEIVE statement is executed, the MCP uses the symbolic name of the wanted queue to access the next message. After execution of the RECEIVE statement the contents of data-name-1 remain unchanged; the contents of data-name-2 through data-name-4 (if applicable) are updated to contain the name of the sub-queue from which the message was received.

MESSAGE DATE Clause: This clause defines data-name-5 as the name of an unsigned 6-digit integer data item, occupying character positions 49 through 54 of the record.

Data-name-5 has the format YYMMDD (year, month, day). Its contents represent the date on which the MCP received this message.

The contents of data-name-5 are updated by the MCP as part of the execution of each RECEIVE statement.

MESSAGE TIME Clause: This clause defines data-name-6 as the name of an unsigned 8-digit integer data item, occupying character positions 55 through 62 of the record.

Data-name-6 has the format HHMMSSTT (hours, minutes, seconds, hundredths of a second). Its contents represent the time of day the message was received into the system by the MCP.

The contents of data-name-6 are updated by the MCP as part of the execution of each RECEIVE statement.

SYMBOLIC SOURCE Clause: This clause defines data-name-7 as the name of an elementary alphanumeric data item of 12 characters, occupying character positions 63 through 74 of the record.

During execution of a RECEIVE statement, the MCP provides in data-name^7 the symbolic name of the terminal that is the source of this message. (The symbolic names the MCP uses are one through eight characters in length; the remaining characters are set to SPACES.) However, if the symbolic name of the source terminal is not known to the MCP, the contents of data-name-7 are set to SPACES.

TEXT LENGTH Clause: This clause defines data-name-8 as the name of an unsigned 4-digit integer data item, occupying character positions 75 through 78 of the record.

The MCP indicates through the contents of data-name-8 the number of main storage bytes of the user's work area filled as a result of the execution of the RECEIVE statement.

END KEY Clause: This clause defines data-name-9 as the name of a 1-character elementary alphanumeric data item, occupying character position 79 of the record.

The MCP sets the contents of data-name-9, as part of the execution of each RECEIVE statement, according to the following rules:

- When RECEIVE MESSAGE is specified, then the contents of data-name-9 are:

  3 if end-of-group has been detected

  2 if end-of-message has been detected

  0 if less than a message has been moved into the user-specified area

- When RECEIVE SEGMENT is specified, then the contents of data-name-9 are:

  3 if end-of-group has been detected

  2 if end-of-message has been detected

  1 if end-of-segment has been detected

  0 if less than a message segment has been moved into the user-specified area.

- When more than one of the above conditions is satisfied simultaneously, the rule first satisfied in the order listed determines the contents of data-name-9. An End of Group is a logical End Of File condition caused by a user request in the MCP. In general, depending on the size of the work unit and the work area provided, End Keys may be associated with a text length of zero. This is always the case for End Of Group, and may be for End Of Message when receiving a segmented message with the RECEIVE SEGMENT option.

  Note: The MCP never removes the End Of Transmission line control character. This character is translated to EBCDIC as X'37'. COBOL assumes that the message is being translated and the user wants the X'37' removed. Therefore, the last data character of a message must never be X'37'.

STATUS KEY Clause: This clause defines data-name-10 as the name of a 2-character elementary alphanumeric data item, occupying character positions 80 and 81 of the record.

The contents of data-name-10 indicate the status condition of the previously executed RECEIVE or IF MESSAGE statement. The program should, therefore, check the STATUS KEY immediately after each RECEIVE operation to verify the status. The values data-name-10 can contain, and their meanings, are defined in Figure 19.

Figure 19 indicates the possible values that the STATUS KEY field (for both input and output CD entries) may contain at the completion of execution for each statement. An X on a line in a statement column indicates that the associated code on that line is possible for that statement.

STATUS KEY Code	Meaning	RECEIVE	SEND	IF MESSAGE
00	No error detected. Request completed.	X	X	X
20	Destination unknown. Data-name-5 gives unknown destination. Request canceled.		X	
20	1) Queue name unknown (No DD card). 2) Invalid queue structure. Request canceled.	X		X
21	Insufficient storage available for control blocks and/or buffers. Request canceled.	X	X	X
22	Queue name unknown. (No DD card.) Request canceled.		X	
29	An input/output error has occurred. Request canceled.	X	X	X
50	Character count greater than sending field. Request ignored.		X	
60	Partial segment with either zero character count or no sending area specified. Request ignored.		X	

Figure 19. STATUS KEY Field -- Possible Values

QUEUE DEPTH Clause: This clause defines data-name-11 as the name of an unsigned 6-digit integer data item, occupying character positions 82 through 87 of the record.

The contents of data-name-11 indicate the number of messages that exist in an input queue. The MCP updates the contents of data-name-11 only as part of the execution of an IF MESSAGE statement.

Format 1 -- Option 2: The second option of Format 1 allows the programmer to specify data-name-1 through data-name-11 without the descriptive clauses. If any data-names are to be omitted, the word FILLER must be substituted for each omitted name, except that FILLER need not be specified for any data-name that comes after the last name to be referenced.

For example, if the programmer wishes to refer to the SYMBOLIC QUEUE as QUEUE-IN and to the MESSAGE DATE as DATE-IN, he can write the input CD entry as follows:

```
CD INPUT-AREA FOR INPUT
   QUEUE-IN FILLER FILLER FILLER DATE-IN.
```

In this case data-name-6 through data-name-11 can be omitted, nor need FILLER be written in their place.

The same input CD entry can be written as follows (in this case, an optional level-01 record description entry redefining the data areas is also included.):

```
CD   INPUT-AREA FOR INPUT
     SYMBOLIC QUEUE IS QUEUE-IN
     MESSAGE DATE IS DATE-IN.
01   INAREA-RECORD.
     05  FILLER         PICTURE X(78).
     05  ENDKEY-CODE    PICTURE X(1).
         88  PARTIAL-SEGMENT   VALUE "0".
         88  END-SEGMENT       VALUE "1".
         88  END-MESSAGE       VALUE "2".
         88  END-TRANSMISSION  VALUE "3".
     05  FILLER         PICTURE X(8).
```

By naming the SYMBOLIC QUEUE and MESSAGE DATE fields of the CD the programmer can refer to these data areas within his program without further defining them. By redefining the END KEY data area, the programmer can use condition names to refer to the values contained in that area.

FORMAT 2: This format is required if the CD entry is FOR OUTPUT. At least one output CD entry must be specified if messages are to be placed into an output queue. A number of output CD entries in the same program or in different subprograms in the same run unit may be used to send different portions of the same message, so that parts of one message may be transferred to the MCP using different CD entries.

Until the transfer of a first message from the COBOL program to the MCP has been completed, the transfer of a second message may not begin. Changing the destination before indicating End Of Message causes unpredictable results.

The specification of an output CD entry always results in a record whose implicit description is equivalent to the following:

Equivalent COBOL Record Description			Description of Use
01	data-name-0.		
02	data-name-1	PICTURE 9(4).	Destination Count
02	data-name-2	PICTURE 9(4).	Text Length
02	data-name-3	PICTURE XX.	Status Key
02	data-name-4	PICTURE X.	Error Key
02	data-name-5	PICTURE X(12).	Symbolic Destination

For each output CD entry, a record area of 23 contiguous Standard Data Format character positions is always generated. It is implicitly defined as previously illustrated. Through the use of the optional clauses, user data-names may be explicitly associated with the output CD subfields as follows:

DESTINATION COUNT Clause: The DESTINATION COUNT clause defines data-name-1 as the name of an unsigned 4-digit integer data item, occupying character positions 1 through 4 of the record. The CODASYL specification for teleprocessing defines the DESTINATION COUNT clause as shown in Format 2. However, since COBOL allows only one destination, the DESTINATION COUNT clause, if specified, is treated as comments.

TEXT LENGTH Clause: This clause defines data-name-2 as the name of an unsigned 4-digit integer data item, occupying character positions 5 through 8 of the record.

As part of the execution of a SEND statement, the MCP interprets the contents of data-name-2 as the user's indication of the number of leftmost bytes of main storage of the identifier named in the SEND statement to be transferred (see SEND statement).

STATUS KEY Clause: This clause defines data-name-3 as the name of a 2-character elementary alphanumeric data item, occupying character positions 9 and 10 of the record.

The contents of data-name-3 indicate the status condition of the previously executed SEND statement. The values data-name-3 can contain, and their meanings, are defined in Figure 19.

ERROR KEY Clause: This clause defines data-name-4 as the name of a 1-character elementary alphanumeric data item, occupying character position 11 of the record.

If, during the execution of a SEND statement, the MCP determines that the specified destination is unknown, the MCP updates the contents of data-name-4. Data-name-4 will contain:

1 if the symbolic destination contained in data-name-4 is unknown to the MCP.

0 if the symbolic destination is known to the MCP.

Note: The ERROR KEY field is set to '1' only when the STATUS KEY is set to '20'. Therefore, the programmer should not examine the ERROR KEY unless the STATUS KEY field is set to '20'.

SYMBOLIC DESTINATION Clause: This clause defines data-name-5 as the name of a 12-character elementary alphanumeric data item, occupying character positions 12 through 23 of the record.

Data-name-5 contains a symbolic destination. The first 1 through 8 characters of data-name-5 must be previously defined to the MCP.

The following example illustrates an output CD entry, with an optional level-01 record description entry redefining the data areas:

```
CD  OUTPUT-AREA FOR OUTPUT
    TEXT LENGTH IS MSG-LGTH
    SYMBOLIC DESTINATION IS Q-OUT.
01  OUTAREA-RECORD.
    05  FILLER          PICTURE X(10).
    05  ERRKEY-CODE     PICTURE X.
        88  KNOWN         VALUE "0".
        88  UNKNOWN       VALUE "1".
    05  FILLER          PICTURE X(12).
```

By naming the TEXT LENGTH and SYMBOLIC DESTINATION fields of the CD entry, the programmer can refer to those data areas within his program without further defining them. By redefining the ERROR KEY data area, the programmer can use condition-names to refer to the values contained in that area.

Note: When a message is being sent to a remote station, TCAM adds the proper End Of Transmission line control character.

FORMAT 3: The CD entry may be pre-written and included in the user-created library. The entry may then be included in a COBOL source program by means of a COPY statement. (See "COPY Statement" in the chapter on the Source Program Library Facility.)

PROCEDURE DIVISION

In the Procedure Division, there is an additional condition which may be used by a COBOL TP program: the message condition.

There are two additional input/output statements used by a COBOL TP program to communicate with the MCP: the RECEIVE statement and the SEND statement.

Each of these language elements is described in the sections that follow.

Message Condition

The message condition determines whether or not one or more complete messages exist in a designated queue of messages. The condition can then be specified in an IF statement.

Format
[NOT] MESSAGE FOR cd-name

The cd-name must specify an input CD entry.

At the time of the test, the CD entry must contain the name of the SYMBOLIC QUEUE to be tested.

A MESSAGE condition exists only if one or more complete messages are present in the named queue. A NOT MESSAGE condition exists if there are no complete messages in the named queue.

Execution of the message condition causes the QUEUE DEPTH field of the named input CD to be updated with the number of complete messages present in the input queue or queue structure. Executing a message condition to a queue structure returns a count of the number of complete messages in the entire structure. Thus the COBOL TP program can check a queue or queue structure for a predetermined message count before invoking a specific TP processing program.

When using compound IF statements, care must be taken to ensure that the message condition is actually tested, so that the QUEUE DEPTH field will actually be updated. For example, suppose the programmer writes:

```
IF A = B AND MESSAGE FOR QUEUE-IN ...
```

then when A is not equal to B, the message condition is not tested, and the QUEUE DEPTH field for QUEUE-IN is not updated. To ensure that the message condition is tested, the programmer must always write it as the first condition tested within a multiple condition.

When the message condition is executed, the STATUS KEY field of the named input CD is set as follows:

'00' for a valid request

'20' invalid queue name or queue structure

'21' insufficient storage for system control blocks

'29' input/output error

(See Figure 19 for a complete explanation.)

When a STATUS KEY other than '00' is returned, the QUEUE DEPTH field is unchanged.

RECEIVE Statement

The RECEIVE statement makes available to the COBOL program a message, message segment, or a portion of a message or message segment, and pertinent information about that message data from a queue maintained by the MCP.

Format

```
RECEIVE cd-name { MESSAGE } INTO identifier-1
                { SEGMENT }

    [NO DATA imperative-statement]
```

The cd-name must specify an input CD entry.

Before a RECEIVE statement is executed, this input CD entry must contain, in its SYMBOLIC QUEUE field, a name of up to 12 characters. The first 1 through 8 characters of this name must be unique, and must match the DDname of the DD statement that specifies the queue.

Upon execution of the RECEIVE statement, data is transferred to the receiving character positions of identifier-1, aligned to the left without any SPACE fill and without any data format conversion. The following data items in the input CD are appropriately updated when the RECEIVE statement is executed: MESSAGE DATE field, MESSAGE TIME field, SYMBOLIC SOURCE field, TEXT LENGTH field, END KEY field, STATUS KEY field (see Figure 19), and if the message was retrieved through a queue structure, SYMBOLIC SUB-QUEUE-1 through SYMBOLIC SUB-QUEUE-3.

A complete message need not be received before another MCP queue is accessed. Thus, messages from different MCP queues may be processed at the same time by a COBOL program. (Note, however, that a message is not made available to the COBOL program until it is completely received by the MCP and placed in a queue.)

A single execution of a RECEIVE statement never returns more than a single message (when the MESSAGE phrase is used) or a single segment (when the SEGMENT phrase is used), regardless of the size of the receiving area.

When the MESSAGE phrase is used the end-of-segment condition, if present, is ignored, and the end-of-segment indicator is treated as a data character. (This occurs only when the user, through the MCP, segments the message, and the COBOL program uses MESSAGE mode to RECEIVE the message.) The following rules apply to the data transfer:

- If a message is the same size as identifier-1, the message is stored in identifier-1.

- If a message size is smaller than identifier-1, the message is aligned to the leftmost character position of identifier-1 with no SPACE fill.

- If a message size is larger than identifier-1, the message fills identifier-1 left to right, starting with the leftmost character of the message. The remainder of the message can be transferred to identifier-1 with subsequent RECEIVE statements referencing the same queue. Either the MESSAGE or the SEGMENT option may be specified for the subsequent RECEIVE statements.

When the SEGMENT phrase is used, the end-of-segment condition, if present (or the end-of-message condition, if present), determines the end of data transfer. In this case, the end-of-segment indicator is not treated as a data character, and is not transferred with the data. The following rules apply to the data transfer:

- If a segment is the same size as identifier-1, the segment is stored in identifier-1.

- If the segment size is smaller than identifier-1, the segment is aligned to the leftmost character position of identifier-1 with no SPACE fill.

- If a segment size is larger than identifier-1, the segment fills identifier-1 left to right starting with the leftmost character position of the segment. The remainder of the segment can be transferred to identifier-1 with subsequent RECEIVE statements referencing the same queue. Either the MESSAGE or the SEGMENT option may be specified for the subsequent RECEIVE statements.

Once the execution of a RECEIVE statement has returned a portion of a message, only subsequent execution of RECEIVE statements in that run unit can cause the remaining portions of the message to be returned.

After the execution of a STOP RUN statement, or of a GOBACK statement in a main program, the disposition of the remaining portions of any message only partially obtained is not defined.

When the NO DATA option is specified and the queue is empty (that is, there are no complete messages in the input queue), then control passes to the imperative-statement specified in the NO DATA option.

When the NO DATA option is not specified and the queue is empty, execution of the COBOL object program is suspended (that is, placed in wait status) until data is made available in identifier-1.

SEND Statement

The SEND statement causes a message, a message segment, or a portion of a message or message segment to be released to the Message Control Program.

Format 1
SEND cd-name FROM identifier-1

Format 2

```
SEND cd-name [FROM identifier-1] { WITH identifier-2 }
                                 { WITH ESI        }
                                 { WITH EMI        }
                                 { WITH EGI        }
```

Messages may be transferred to the MCP in segments, as complete messages, or in parts of segments or messages. However, data is never transmitted to the named destination until a complete message has been transferred to the MCP.

Until the transfer of a first message from the COBOL program to the MCP has been completed, the transfer of a second message may not begin. Changing the destination before indicating End Of Message causes unpredictable results.

The cd-name must specify an output CD entry.

Before a SEND statement is executed, this output CD entry must contain:

- in the TEXT LENGTH field, the number of leftmost bytes of contiguous data to be transferred to the output queue from identifier-1.
- in the SYMBOLIC DESTINATION field, the symbolic identification of the remote station(s) that are to receive the message. (The first 1 through 8 characters of this field must be previously defined to the MCP.)

Upon execution of the SEND statement, data is transferred from identifier-1 to the MCP queue corresponding to the terminal identifier contained in the SYMBOLIC DESTINATION field.

As part of the execution of the SEND statement, the MCP interprets the contents of the TEXT LENGTH field to be the user's indication of the number of leftmost character positions of identifier-1 from which data is to be transferred.

If the contents of the TEXT LENGTH field are zero, no characters of data are transferred from identifier-1. (A zero TEXT LENGTH field is valid only with the Format 2 SEND statement.)

If the contents of the TEXT LENGTH field are outside the range of zero through the size of identifier-1 inclusive, an error is indicated in the STATUS KEY field, no data is transferred, and the name in the SYMBOLIC DESTINATION field is not validated. The contents of the STATUS KEY field are updated by the MCP. (See Figure 19, STATUS KEY Field -- Possible Values.)

If the user causes special control characters to be embedded as data characters within the message, these control characters are enqueued with the message, and it is the user's responsibility to ensure that these characters function as intended.

The disposition of a portion of a message not terminated by a subsequent and associated EMI or EGI is undefined. (However, such a message portion will not be transmitted to the destination.)

Format 2 Considerations: This format of the SEND statement allows the programmer to specify whether or not an end indicator is associated with the message.

If the FROM identifier-1 option is omitted, then an end indicator is associated with the data enqueued by a previous SEND statement.

The hierarchy of end indicators, and their meanings, is as follows:

EGI End of Group Indicator -- the CODASYL specification defines the EGI as indicating that the group of messages to be transmitted is complete. However, for this implementation, the EGI is regarded as equivalent to the EMI. Therefore, if EGI is specified without a preceding EMI, the EGI is regarded as an EMI; if the EGI is specified after a preceding EMI, the EGI is treated as comments (that is, is ignored).

EMI End of Message Indicator -- the message to be transmitted is complete.

ESI End of Segment Indicator -- the segment to be transmitted is complete.

An EGI need not be preceded by an EMI or ESI. An EMI need not be preceded by an ESI.

Identifier-2 must reference a 1-character integer without an operational sign. The contents of identifier-2 indicate that the contents of identifier-1 have an end indicator associated with them according to the following codes:

If identifier-2 contains:	Then identifier-1 has associated with it:	Which means:
0	no indicator	no indicator
1	ESI	End of Segment Indicator
2	EMI	End of Message Indicator
3	EGI	End of Group Indicator

Any character other than 1, 2, or 3 is interpreted as 0.

If the contents of identifier-2 are other than 1, 2, or 3, and identifier-1 is not specified, then an error is indicated in the STATUS KEY field of the associated CD entry, and no data is transferred.

Program Product Information (Version 4)

STRING MANIPULATION

String manipulation statements allow the COBOL programmer greater flexibility in data manipulation. With the STRING statement he can concatenate two or more subfields into a single field. With the UNSTRING statement he can separate contiguous data in a single field into multiple logical subfields. The subfields need not be contiguous.

STRING Statement

The STRING statement provides juxtaposition of the partial or complete contents of two or more data items into a single data item.

Format

```
STRING {identifier-1} [identifier-2] ... DELIMITED BY {identifier-3}
       {literal-1   }  [literal-2   ]                  {literal-3   }
                                                      {SIZE        }

 [{identifier-4} [identifier-5] ... DELIMITED BY {identifier-6}
  {literal-4   }  [literal-5   ]                 {literal-6   }] ...
                                                 {SIZE        }

  INTO identifier-7 [WITH POINTER identifier-8]

  [ON OVERFLOW imperative-statement]
```

All literals must be described as nonnumeric literals. Each literal may be any figurative constant without the optional word ALL.

All identifiers, except identifier-8, must be described implicitly or explicitly as USAGE IS DISPLAY. Identifier-3 and identifier-6 must each reference a fixed length data item.

Identifier-7 must represent an elementary data item without editing symbols. If a SEPARATE SIGN clause is specified, it is ignored during execution of the STRING statement.

Identifier-8 must represent an elementary numeric integer data item of sufficient size to contain a value equal to the size plus 1 of the area referenced by identifier-7.

All references to identifier-1, identifier-2, identifier-3, literal-1, literal-2, and literal-3 apply equally to identifier-4, identifier-5, identifier-6, literal-4, literal-5, and literal-6, respectively, and all repetitions thereof.

identifier-1, literal-1, identifier-2, and literal-2 represent the sending items. identifier-7 represents the receiving item.

literal-3 and identifier-3 indicate the character(s) delimiting the move. If the SIZE phrase is used, the complete data item defined by identifier-1, literal-1, identifier-2, literal-2 is moved.

When a figurative constant is specified as literal-1, literal-2, it refers to an implicit one-character data item whose USAGE IS DISPLAY.

When the STRING statement is executed, the transfer of data is governed by the following rules:

- Those characters from the sending item(s) are transferred to the receving item in accordance with the rules for alphanumeric to alphanumeric moves, except that no SPACE filling is provided. (See the MOVE statement in "Procedure Division".)

- If the DELIMITED phrase is specified without the SIZE option, the contents of each sending item are transferred to the receiving data item in the sequence specified in the STRING statement, beginning with the leftmost character of the first sending item, and continuing from left to right through each successive sending item until either:

  1. the delimiting character(s) (literal-3/identifier-3, or literal-6/identifier-6) for this sending item are reached, or

  2. the rightmost character of this sending item has been transferred.

  The delimiting character(s) are not transferred into the receiving data item. When the receiving field is filled, or when all of the DELIMITED data in all of the sending fields has been transferred the operation is ended.

- If the DELIMITED phrase is specified with the SIZE option, the entire contents of each sending item are transferred, in the sequence specified in the STRING statement, to the receiving data item. The operation is ended either when all data has been transferred or when the receiving field is filled.

The POINTER option may be used explicitly by the programmer to designate where data is to be placed in the receiving area. If the POINTER option is specified, identifier-8 is explicitly available to the user, and he is responsible for setting its initial value. The initial value must not be less than one and must not exceed the number of character positions of the receiving item. (Note that the POINTER item must be defined as of sufficient size to contain a value equal to the size of the receiving item plus one. This precludes the possibility of arithmetic overflow when the system updates the pointer. The following rule applies:

- Conceptually, when the STRING statement is executed, the following actions occur. Characters are transferred into the receiving item one at a time, beginning at the character position indicated by the POINTER value. After each character is positioned, the value of the POINTER item (identifier-8) is increased by one. The value associated with the POINTER item is changed only in this manner. At the termination of any STRING operation, the value in the POINTER item always points to one character beyond the last character moved into the receiving item.

Note: The POINTER value may therefore be used in a subsequent STRING statement to place additional characters immediately to the right of those already placed in the receiving item.

If the POINTER option is not specified, the STRING statement acts as if the user had specified a pointer with an initial value of one. When the statement is executed, the implicit pointer is incremented as described above. The implicit pointer is not available to the programmer.

At the end of execution of a STRING statement, only that portion of the receiving item that was referenced during the execution of the STRING statement is changed. All other portions of the receiving item contain data that was present before this execution of the STRING statement.

If at any time during or after initialization of the STRING statement, but before execution of the STRING statement is completed, the value associated with the POINTER item is less than one, or exceeds the number of character positions in the receiving item, no (further) data is transferred, and, if specified, the imperative-statement in the ON OVERFLOW option is executed.

If the ON OVERFLOW option is not specified and the conditions described above are encountered, control passes to the next statement as written.

Example: The following example illustrates some of the considerations that apply to the STRING statement.

In the Data Division, the programmer has defined the following fields:

```
77  RPT-LINE  PICTURE X(120).
77  LINE-POS  PICTURE 99.
77  LINE-NO   PICTURE 9(5) VALUE 1.
77  DEC-POINT PICTURE X Value ".".
```

In the File Section he has defined the following input record:

```
01  RCD-01.
    05  CUST-INFO.
        10 CUST-NAME     PICTURE X(15)
        10 CUST-ADDR     PICTURE X(35)
    05  BILL-INFO.
        10 INV-NO        PICTURE X(6).
        10 INV-AMT       PICTURE $$,$$$.99.
        10 AMT-PAID      PICTURE $$,$$$.99.
        10 DATE-PAID     PICTURE X(8).
        10 BAL-DUE       PICTURE $$,$$$.99.
        10 DATE-DUE      PICTURE X(8).
```

The programmer wishes to construct an output line consisting of portions of the information from RCD-01. The line is to consist of a line number, customer name and address, invoice number, next billing date, and balance due, truncated to the dollar figure shown.

The record as read in contains the following information:

```
J.B.bSMITHbbbbb
444bSPRINGbST.,bCHICAGO,bILL.bbbbbb
A14725
$4,736.85
$2,400.00
09/22/71
$2,336.85
10/22/71
```

In the Procedure Division, the programmer initializes RPT-LINE to SPACES, and sets LINE-POS (which is to be used as the POINTER field) to 4. Then he issues this STRING statement:

```
STRING LINE-NO SPACE CUST-INFO DELIMITED BY SIZE INV-NO SPACE
     DATE-DUE SPACE DELIMITED BY SIZE BAL-DUE DELIMITED BY
     DEC-POINT INTO RPT-LINE WITH POINTER LINE-POS.
```

When the statement is executed, the following actions take place:

1. The field LINE-NO is moved into positions 4 through 8 of RPT-LINE.
2. A space is moved into position 9.
3. The group item CUST-INFO is moved into positions 10 through 59.
4. INV-NO is moved into positions 60 through 65.
5. A space is moved into position 66.
6. DATE-DUE is moved into positions 67-74.
7. A space is moved into position 75.
8. The portion of BAL-DUE that precedes the decimal point is moved into positions 76 through 81.

At the end of execution of the STRING statement, RPT-LINE appears as follows:

```
Column
4       10          25                          60
|       |           |                           |
|       |           |                           |
|       |           |                           |
V       V           V                           V

00001 J.B. SMITH 444 SPRING ST., CHICAGO, ILL. A14725 10/22/71  $2,336
```

UNSTRING Statement

The UNSTRING statement causes contiguous data in a sending field to be separated and placed into multiple receiving fields.

Format

```
UNSTRING identifier-1

[DELIMITED BY [ALL] { identifier-2 } [OR [ALL] { identifier-3 } ]...]
                    { literal-1    }           { 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]
```

Each literal must be described as a nonnumeric literal. In addition, each literal may be any figurative constant except the figurative constant ALL literal. (That is, the form ALL ALL literal may not be specified.)

Identifier-1, identifier-2, identifier-3, identifier-5, and identifier-8 must each be described, implicitly or explicitly, as an alphanumeric data item.

Identifier-4 and identifier-7 must each be described, implicitly or explicitly, as USAGE DISPLAY. Each may be described as alphabetic, alphanumeric, or numeric (without the symbol P in the PICTURE character string).

Identifier-6, identifier-9, identifier-10, and identifier-11 must be described as elementary numeric integer data items.

No identifier may name a level-88 entry.

The DELIMITER IN option and the COUNT IN option may be specified only if the DELIMITED BY option is specified.

All references to identifier-2, literal-1, identifier-4, identifier-5, and identifier-6 apply equally to identifier-3, literal-2, identifier-7, identifier-8, and identifier-9, respectively, and all repetitions thereof.

Identifier-1 represents the sending area.

Identifier-4 represents the data receiving area. Identifier-5 represents the receiving area for delimiters.

Literal-1 or identifier-2 specifies a delimiter. No more than 15 delimiters may be specified.

Identifier-6 represents the count of the number of characters within the sending area isolated by the delimiters for the move

into the current receiving area. This value does not include the count of the delimiter character(s).

Identifier-10 contains a value that indicates a relative character position within the sending area.

Identifier-11 is a counter that records the number of receiving areas acted upon during the execution of the UNSTRING statement.

When the ALL option is specified, two or more contiguous occurrences of literal-1 or of identifier-2 are treated as if they were only one occurrence. However, identifier-5 (the receiving area for delimiters) contains as many complete occurrences of the delimiter as are present or as it can hold, whichever is smaller.

When ALL is specified, and two or more delimiters are found, as much of the first occurrence of the delimiter as will fit is moved into identifier-5. Each additional occurrence of the delimiter is moved into identifier-5 only if the complete occurrence will fit.

When ALL is not specified, and the examination encounters two contiguous occurrences of literal-1 or identifier-2, the current receiving area for data is either space-filled or zero-filled, according to the description of the receiving area.

Literal-1 or identifier-2 may contain any characters in the EBCDIC character set.

Each literal-1 or identifier-2 represents one delimiter. When a delimiter contains two or more characters, all of the characters must be present in contiguous positions in the sending field, and in the sequence specified, to be recognized as that delimiter. When a figurative constant is used as a delimiter, it stands for a single character nonnumeric literal.

When two or more delimiters are specified in the DELIMITED BY option, an OR condition exists. Each non-overlapping occurrence of any one of them is considered a delimiter, and is applied to the sending field in the sequence specified in the UNSTRING statement. For example, if DELIMITED BY AB OR BC is specified, then an occurrence of either AB or BC in the sending field is considered a delimiter; an occurrence of ABC is considered an occurrence of AB.

When the UNSTRING statement is initiated, the current receiving area is identifier-4. Data is transferred from identifier-1 to identifier-4 according to the following rules:

- If the POINTER option is specified the string of characters in the sending area is examined beginning with the relative character position indicated by the contents of the POINTER item. If the POINTER option is not specified, the character string is examined beginning with the leftmost character position.
- If the DELIMITED BY option is specified, the examination proceeds left to right until a delimiter specified by either literal-1 or the value in identifier-2 is encountered. If the end of the sending item is encountered before the delimiting condition is met, the examination terminates with the last character examined.
- If the DELIMITED BY option is not specified, the number of characters examined is equal to the size of the current

receiving area. The size of the receiving area depends on its data category:

1. If it is alphanumeric or alphabetic, its size is equal to the size of the current receiving area.

2. If it is numeric, then its size is equal to the integer portion of the current receiving field.

3. If it is described with the SEPARATE SIGN clause, the characters examined are one fewer than the size of the current receiving area.

4. If it is described as a variable-length data item, the number of characters examined is determined by the current size of the receiving area.

- The characters thus examined (excluding the delimiting character(s), if any) are treated as an elementary alphanumeric data item, and are moved into the current receiving area according to the rules for the MOVE statement. (See the MOVE statement in the Procedure Division chapter.) Note that if two delimiters are adjacent, that is, with no data characters between them, the null receiving field is filled with zeroes or spaces, depending on its description.

- If the DELIMITER IN option is specified, the delimiting character(s) are treated as an elementary alphanumeric data item and are moved into identifier-5 according to the rules for an elementary move. If the delimiting condition is the end of the sending area, then identifier-5 (the DELIMITER) is space-filled or zero-filled according to its PICTURE character string.

- If the COUNT IN option is specified, a value equal to the number of characters thus examined (excluding the delimiter character(s), if any) is moved into identifier-6 according to the rules for an elementary move.

- If the DELIMITED BY option is specified the string of characters is further examined beginning with the first character to the right of the delimiter. If the DELIMITED BY option is not specified the string of characters is further examined beginning with the character to the right of the last character transferred.

- After data is transferred to identifier-4, the current receiving area becomes identifier-7. The procedure described is then repeated either until all the characters in the sending area have been transferred, or until there are no more unfilled receiving areas.

The initialization of the data items associated with the POINTER phrase or the TALLYING phrase is the responsibility of the user.

The contents of the data item referenced by identifier-10 (the POINTER item) behave as if incremented by one for each character examined in the sending area. When the execution of an UNSTRING statement with a POINTER option is completed, the contents of identifier-10 contain a value equal to the initial value plus the number of characters examined in the sending area.

When the execution of an UNSTRING statement with the TALLYING option is completed, the contents of identifier-11 contain a value equal to the initial value plus the number of data receiving areas acted upon (including null fields).

Either of the following situations causes an overflow condition:

- An UNSTRING statement is initiated, and the value in the POINTER item (identifier-10) is less than one or greater than the size of the sending area.
- If, during the execution of an UNSTRING statement, all receiving areas have been acted upon, and the sending area still contains characters that have not been examined.

When an overflow condition exists, the UNSTRING operation is terminated. If an ON OVERFLOW option has been specified, the imperative-statement included in the ON OVERFLOW option is executed. If the ON OVERFLOW option is not specified, control passes to the next statement as written.

Example: The following example illustrates some of the considerations that apply to the UNSTRING statement.

In the Data Division, the programmer has defined the following input record to be acted upon by the UNSTRING statement:

```
01  INV-RCD.
    05  CONTROL-CHARS          PIC XX.
    05  ITEM-IDENT             PIC X(20).
    05  FILLER                 PIC X.
    05  INV-CODE               PIC X(10).
    05  FILLER                 PIC X.
    05  NO-UNITS               PIC 9(6).
    05  FILLER                 PIC X.
    05  PRICE-PER-M            PIC 99999.
    05  FILLER                 PIC X.
    05  RTL-AMT                PIC 9(6).99.
```

The next two records are defined as receiving fields for the UNSTRING statement. DISPLAY-REC is to be used for printed output. WORK-REC is to be used for further internal processing.

```
01  DISPLAY-REC.
    05  INV-NO                 PIC X(6).
    05  FILLER                 PIC X VALUE SPACE.
    05  ITEM-NAME              PIC X(20).
    05  FILLER                 PIC X VALUE SPACE.
    05  DISPLAY-DOLS           PIC 9(6).
```

```
01  WORK-REC.
    05  M-UNITS                      PIC 9(6).
    05  FIELD-A                      PIC 9(6).
    05  WK-PRICE REDEFINES FIELD-A   PIC 9999V99.
    05  INV-CLASS                    PIC X(3).
```

He has also defined the following fields for use as control fields in the UNSTRING statement:

```
77  DBY-1              PIC X.
77  CTR-1              PIC 99.
77  CTR-2              PIC 99.
77  CTR-3              PIC 99.
77  CTR-4              PIC 99.
77  DLTR-1             PIC X.
77  DLTR-2             PIC X.
77  CHAR-CT            PIC 99.
77  FLDS-FILLED        PIC 99.
```

In the Procedure Division, the programmer writes the following UNSTRING statement to move subfields of INV-RCD to the subfields of DISPLAY-REC and WORK-REC:

```
UNSTRING INV-RCD DELIMITED BY ALL SPACES OR "/" OR DBY-1
    INTO ITEM-NAME COUNT IN CTR-1
         INV-NO DELIMITER IN DLTR-1 COUNT IN CTR-2
         INV-CLASS
         M-UNITS COUNT IN CTR-3
         FIELD-A
         DISPLAY-DOLS DELIMITER IN DLTR-2 COUNT IN CTR-4
         WITH POINTER CHAR-CT
         TALLYING IN FLDS-FILLED
         ON OVERFLOW GO TO UNSTRING-COMPLETE.
```

Before the UNSTRING statement is issued, the programmer places the value 3 in CHAR-CT (the POINTER item), since he does not wish to work with the two control characters at the beginning of INV-RCD. In DBY-1 he places a period (.) for use as a delimiter, and in FLDS-FILLED (the TALLYING item) he places the value 0 (zero). The following data is then read into INV-RCD:

```
Column:
   1        10        20        30        40        50
   |        |         |         |         |         |
   |        |         |         |         |         |
   V        V         V         V         V         V

   ZYFOUR-PENNY-NAILS    707890/BBA 475120 00122 000379.50
```

When the UNSTRING statement is executed, the following actions take place:

1. Positions 3 through 18 (FOUR-PENNY-NAILS) of INV-RCD are placed in ITEM-NAME, left-justified within the area, and the unused character positions are padded with spaces. The value 16 is placed in CTR-1.

2. Since ALL SPACES is specifed as a delimiter, the 5 contiguous SPACE characters are considered to be one occurrence of the delimiter.

3. Positions 24 through 29 (707890) are placed in INV-NO. The delimiter character / is placed in DLTR-1, and the value 6 is placed in CTR-2.

4. Positions 31 through 33 are placed in INV-CLASS. The delimiter is a SPACE, but since no field has been defined as a receiving area for delimiters, the SPACE is merely bypassed.

5. Positions 35 through 40 (475120) are examined, and are placed in M-UNITS. The delimiter is a SPACE, but since no receiving field has been defined as a receiving area for delimiters, the SPACE is bypassed. The value 6 is placed in CTR-3.

6. Positions 42 through 46 (00122) are placed in FIELD-A, and right-justified within the area. The high-order digit position is filled with a 0 (zero). The delimiter is a SPACE, but since no field has been defined as a receiving area for delimiters, the SPACE is bypassed.

7. Positions 48 through 53 (000379) are placed in DISPLAY-DOLS. The period (.) delimiter character is placed in DLTR-2, and the value 6 is placed in CTR-4.

8. Since all receiving fields have been acted upon and two characters of data in INV-RCD have not been examined, the ON OVERFLOW exit is taken, and execution of the UNSTRING statement is completed.

At the end of execution of the UNSTRING statement, DISPLAY-REC contains the following data:

```
707980 FOUR-PENNY-NAILS    000379
```

WORK-REC contains the following data

```
475120000122BBA
```

CHAR-CT (the POINTER field) contains the value 55, and FLDS-FILLED (the TALLYING field) contains the value 6.

- APPENDIXES

  A: Intermediate Results

  B: Sample Programs

  C: IBM American National Standard COBOL Formats and Reserved Words

  D: File-Processing Techniques and Applicable Statements and Clauses

  E: ASCII Considerations (Version 3 and Version 4)

  F: SYMBOLIC DEBUGGING (Version 4)

  G: 3505/3525 Processing (Version 4)

- IBM American National Standard COBOL Glossary

This appendix discusses the conceptual compiler algorithms for determining the number of integer and decimal places reserved for intermediate results. The following abbreviations are used:

i -- number of integer places carried for an intermediate result.

d -- number of decimal places carried for an intermediate result.

dmax -- either

- the maximum number of decimal places defined for any operand (except for floating-point operands, exponents, or divisors), or
- the number of decimal places needed for the final result field

whichever is larger, in a particular statement.

op1 -- first operand in a generated arithmetic statement.

op2 -- second operand in a generated arithmetic statement.

d1,d2 -- number of decimal places defined for op1 or op2, respectively.

ir -- intermediate result field obtained from the execution of a generated arithmetic statement or operation. Ir1, ir2, etc., represent successive intermediate results. These intermediate results are generated either in registers or in storage locations. Successive intermediate results may have the same location.

In the case of an arithmetic statement containing only a single pair of operands, no intermediate results are generated, except when the TRUNC option is specified for COMPUTATIONAL items. Intermediate results are possible in the following cases:

1. In an ADD or SUBTRACT statement containing multiple operands immediately following the verb.
2. In a COMPUTE statement specifying a series of arithmetic operations.
3. In arithmetic expressions contained in IF or PERFORM statements.

In such cases, the compiler treats the statement as a succession of operations. For example, the following statement:

```
COMPUTE Y = A + B * C - D / E + F ** G
```

is replaced by

```
**F            BY G          yielding ir1
MULTIPLY B     BY C          yielding ir2
DIVIDE E       INTO D        yielding ir3
ADD A          TO ir2        yielding ir4
SUBTRACT ir3   FROM ir4      yielding ir5
ADD ir5        TO ir1        yielding Y
```

COMPILER CALCULATION OF INTERMEDIATE RESULTS

The number of integer places an ir is calculated as follows:

- The compiler first determines the maximum value that the ir can contain by performing the statement in which the ir occurs.

  1. If an operand in this statement is a data-name, the value used for the data-name is equal to the numerical value of the PICTURE for data-name (e.g., PICTURE 9V99 has the value 9.99).

  2. If an operand is a literal, the literal's actual value is used.

  3. If an operand is an intermediate result, the value determined for the intermediate result in a previous arithmetic operation is used.

  4. If the operation is division:

     a. If op2 is a data-name, the value used for op2 is the minumum nonzero value of the digit in the PICTURE for the data-name (e.g., PICTURE 9V99 has the value 0.01).

     b. If op2 is an intermediate result, the intermediate result is treated as though it had a PICTURE, and the minimum nonzero value of the digits in this PICTURE is used.

- When the maximum value of the ir is determined by the above procedures, i is set equal to the number of integers in the maximum value.

- The number of decimal places contained in an ir is calculated as:

Operation	Decimal Places
+ or -	d1 or d2, whichever is greater
*	d1 + d2
/	d1 - d2 or dmax, whichever is greater
**	dmax if op2 is nonintegral or a data-name; d1 * op2 if op2 is an integral literal

Note: It is the user's responsibility to ensure that he defines the operands of any arithmetic statement with enough decimal places to give the desired accuracy in the final result.

Table 27 indicates the action of the compiler when handling intermediate results.

Table 27. Compiler Action on Intermediate Results

Value of i + d	Value of d	Value of i + dmax	Action Taken
<30	Any value	Any value	i integer and d decimal places are carried for ir
=30			
>30	<dmax	Any value	30 - d integer and d decimal places are carried for ir
	=dmax		
	>dmax	<30	i integer and 30 - i decimal places are carried for ir
		=30	
		>30	30 - dmax integer and dmax decimal places are carried for ir

The three programs in this appendix illustrate several methods of accessing mass storage files. The programs are:

1. CREATION OF A DIRECT FILE

2. CREATION OF AN INDEXED FILE

3. RANDOM RETRIEVAL AND UPDATING OF AN INDEXED FILE

CREATION OF A DIRECT FILE

This program creates a file with direct organization through the use of an ACTUAL KEY. The ACTUAL KEY consists of a relative track address and a unique record identifier. In the program, a field in the input record (CD-ITEM-CODE) is converted to a track address (TRACK-ID) through the use of a simple remainder randomizing technique. This technique consists of dividing the value in the field of the input record (CD-ITEM-CODE) by 19, and using the resulting remainder (TRACK-ID) as the relative track address.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CREATEDF.
REMARKS. ILLUSTRATE CREATION OF A DIRECT  FILE.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-360-H50.
OBJECT-COMPUTER. IBM-360-H50.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT DA-FILE ASSIGN TO DA-2311-D-MASTER
        ACCESS IS RANDOM
        ACTUAL KEY IS FILEKEY.
    SELECT CARD-FILE ASSIGN TO UR-1442R-S-INFILE
        RESERVE 3 ALTERNATE AREAS.
DATA DIVISION.
FILE SECTION.
FD  DA-FILE
    DATA RECORD IS DISK
    LABEL RECORDS ARE STANDARD.
01  DISK.
    05  DISK-ITEM-CODE     PICTURE X(3).
    05  DISK-ITEM-NAME     PICTURE X(29).
    05  DISK-STOCK-ON-HAND PICTURE S9(6)     USAGE COMP SYNC.
    05  DISK-UNIT-PRICE    PICTURE S999V99   USAGE COMP SYNC.
    05  DISK-STOCK-VALUE   PICTURE S9(9)V99  USAGE COMP SYNC.
    05  DISK-ORDER-POINT   PICTURE S9(3)     USAGE COMP SYNC.
FD  CARD-FILE
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS CARDS.
01  CARDS.
    05  CD-ITEM-CODE       PICTURE X(3).
    05  CD-ITEM-NAME       PICTURE X(29).
    05  CD-STOCK-ON-HAND   PICTURE S9(6).
    05  CD-UNIT-PRICE      PICTURE S999V99.
    05  CD-STOCK-VALUE     PICTURE S9(9)V99.
    05  CD-ORDER-POINT     PICTURE S9(3).
    05  FILLER             PICTURE X(23).
WORKING-STORAGE SECTION.
77  SAVE                   PICTURE S9(5) USAGE COMP SYNC RIGHT.
77  QUOTIENT               PICTURE S9(4) USAGE COMP SYNC RIGHT.
77  PRODUCT                PICTURE S9(4) USAGE COMP SYNC RIGHT.
01  FILEKEY.
    05  TRACK-ID           PICTURE S9(5) USAGE COMP SYNC RIGHT.
    05  RECORD-ID          PICTURE X(29).
PROCEDURE DIVISION.
BEGIN.
    OPEN INPUT CARD-FILE.
    OPEN OUTPUT DA-FILE.
```

```
PARA-1.
    READ CARD-FILE AT END GO TO END-JOB.
    MOVE CD-ITEM-CODE TO SAVE.
    DIVIDE 19 INTO SAVE GIVING QUOTIENT
        REMAINDER TRACK-ID.
    MOVE CD-ITEM-NAME TO RECORD-ID.
    MOVE CD-ITEM-CODE TO DISK-ITEM-CODE.
    MOVE CD-ITEM-NAME TO DISK-ITEM-NAME.
    MOVE CD-STOCK-ON-HAND TO DISK-STOCK-ON-HAND.
    MOVE CD-UNIT-PRICE TO DISK-UNIT-PRICE.
    MOVE CD-STOCK-VALUE TO DISK-STOCK-VALUE.
    MOVE CD-ORDER-POINT TO DISK-ORDER-POINT.
WR.
    WRITE DISK INVALID KEY GO TO ERROR-ROUTINE.
    GO TO PARA-1.
END-JOB.
    CLOSE CARD-FILE DA-FILE.
    DISPLAY "END OF JOB".
    STOP RUN.
ERROR-ROUTINE.
    DISPLAY "UNABLE TO WRITE RECORD".
    DISPLAY TRACK-ID.
    GO TO PARA-1.
```

CREATION OF AN INDEXED FILE

This program creates an indexed file. These records are presented in ascending sequence by RECORD KEY. The operating system builds the index, prime, and overflow areas.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CREATEIS.
REMARKS. ILLUSTRATE CREATION OF INDEXED SEQUENTIAL FILE.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-360-H50.
OBJECT-COMPUTER. IBM-360-H50.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT IS-FILE ASSIGN TO DA-2311-I-MASTER
        RESERVE NO ALTERNATE AREAS
        ACCESS IS SEQUENTIAL
        RECORD KEY IS REC-ID.
    SELECT CARD-FILE ASSIGN TO UR-1442R-S-INFILE
        RESERVE 10 ALTERNATE AREAS.
DATA DIVISION.
FILE SECTION.
FD  IS-FILE
    BLOCK CONTAINS 5 RECORDS
    RECORDING MODE IS F
    LABEL RECORDS ARE STANDARD
    DATA RECORD IS DISK.
01  DISK.
    05  DELETE-CODE      PICTURE X.
    05  REC-ID           PICTURE 9(10).
    05  DISK-FLD1        PICTURE X(10).
    05  DISK-NAME        PICTURE X(20).
    05  DISK-BAL         PICTURE 99999V99.
    05  FILLER           PICTURE X(52).
FD  CARD-FILE
    RECORDING MODE IS F
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS CARDS.
01  CARDS.
    05  KEY-ID           PICTURE 9(10).
    05  CD-NAME          PICTURE X(20).
    05  CD-BAL           PICTURE 99999V99.
    05  FILLER           PICTURE X(43).
PROCEDURE DIVISION.
BEGIN.
    OPEN INPUT CARD-FILE.
    OPEN OUTPUT IS-FILE.
PARA-1.
    READ CARD-FILE AT END GO TO END-JOB.
    MOVE KEY-ID TO REC-ID.
    MOVE LOW-VALUE TO DELETE-CODE.
    MOVE CD-NAME TO DISK-NAME.
    MOVE CD-BAL TO DISK-BAL.
    WRITE DISK INVALID KEY GO TO ERR.
    GO TO PARA-1.
ERR.
    DISPLAY "DUPLICATE OR SEQ-ERR" UPON CONSOLE.
    DISPLAY KEY-ID UPON CONSOLE.
    GO TO PARA-1.
END-JOB.
    CLOSE CARD-FILE IS-FILE.
    DISPLAY "END OF JOB" UPON CONSOLE.
    STOP RUN.
```

RANDOM RETRIEVAL AND UPDATING OF AN INDEXED FILE

This program randomly updates an existing indexed file. The READ IS-FILE statement causes a search of indexes for an equal compare between the NOMINAL KEY obtained from the input record and the RECORD KEY of the I-O file. If an equal compare occurs, the record is updated, and the details of this update are printed. If a matching record is not found, the invalid key branch is taken.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. RANDOMIS.
REMARKS. ILLUSTRATE RANDOM RETRIEVAL FROM IS-FILE.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-360-H50.
OBJECT-COMPUTER. IBM-360-H50.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT IS-FILE ASSIGN TO DA-2311-I-MASTER
        ACCESS IS RANDOM
        NOMINAL KEY IS KEY-ID
        RECORD KEY IS REC-ID.
    SELECT CARD-FILE ASSIGN TO UR-1442R-S-INFILE
        RESERVE 10 ALTERNATE AREAS.
    SELECT PRINT-FILE ASSIGN TO UT-2400-S-PROUT
        RESERVE NO ALTERNATE AREAS.
I-O-CONTROL.
    RERUN ON UT-2400-S-CKPT EVERY 10000 RECORDS OF IS-FILE.
DATA DIVISION.
FILE SECTION.
FD  IS-FILE
    BLOCK CONTAINS 5 RECORDS
    RECORD CONTAINS 100 CHARACTERS
    LABEL RECORDS ARE STANDARD
    RECORDING MODE IS F
    DATA RECORD IS DISK.
01  DISK.
    05  DELETE-CODE      PICTURE X.
    05  REC-ID           PICTURE 9(10).
    05  DISK-FLD1        PICTURE X(10).
    05  DISK-NAME        PICTURE X(20).
    05  DISK-BAL         PICTURE 99999V99.
    05  FILLER           PICTURE X(52).
FD  CARD-FILE
    RECORDING MODE IS F
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS CARDS.
01  CARDS.
    05  KEY-IDA          PICTURE 9(10).
    05  CD-NAME          PICTURE X(20).
    05  CD-AMT           PICTURE 99999V99.
    05  FILLER           PICTURE X(43).
FD  PRINT-FILE
    RECORDING MODE IS F
    LABEL RECORDS ARE STANDARD
    DATA RECORD IS PRINTER.
```

```
01  PRINTER.
    05  FORMSC          PICTURE X.
    05  PRINT-ID        PICTURE X(10).
    05  FILLER          PICTURE X(10).
    05  PRINT-NAME      PICTURE X(20).
    05  FILLER          PICTURE X(10).
    05  PRINT-BAL       PICTURE $ZZZ,999.99-.
    05  FILLER          PICTURE X(10).
    05  PRINT-AMT       PICTURE $ZZZ,ZZZ.99-.
    05  FILLER          PICTURE X(10).
    05  PRINT-NEW-BAL   PICTURE $ZZZ,ZZZ.99-.
WORKING-STORAGE SECTION.
77  KEY-ID              PICTURE 9(10).
PROCEDURE DIVISION.
BEGIN.
    OPEN INPUT CARD-FILE.
    OPEN OUTPUT PRINT-FILE.
    OPEN I-O IS-FILE.
PARA-1.
    MOVE SPACES TO PRINTER.
    READ CARD-FILE AT END GO TO END-JOB.
    MOVE KEY-IDA TO KEY-ID.
    READ IS-FILE INVALID KEY GO TO NO-RECORD.
    MOVE REC-ID TO PRINT-ID.
    MOVE DISK-NAME TO PRINT-NAME.
    MOVE DISK-BAL TO PRINT-BAL.
    MOVE CD-AMT TO PRINT-AMT.
    ADD CD-AMT TO DISK-BAL.
    MOVE DISK-BAL TO PRINT-NEW-BAL.
    REWRITE DISK.
    WRITE PRINTER AFTER POSITIONING 2 LINES.
    GO TO PARA-1.
NO-RECORD.
    DISPLAY 'NO RECORD FOUND' UPON CONSOLE.
    DISPLAY KEY-ID UPON CONSOLE.
    GO TO PARA-1.
END-JOB.
    CLOSE CARD-FILE PRINT-FILE IS-FILE.
    DISPLAY 'END OF JOB' UPON CONSOLE.
    STOP RUN.
```

APPENDIX C: AMERICAN NATIONAL STANDARD COBOL FORMAT SUMMARY AND RESERVED WORDS

The Formats and Reserved Words in this appendix have been printed in a specially reduced size with pages numbered in sequence to make up a pocket-sized reference booklet for use when coding IBM Full American National Standard COBOL programs. Although most readers may prefer to retain this reference material within the manual, the booklet can be prepared as follows:

- cut along trim lines.
- place sheets so that page numbers at lower right-hand corner are in ascending order in odd-number progression (i.e., 1, 3, 5, etc.); lower left-hand page numbers will then be in descending order in even-number progression (i.e., 20, 18, 16, etc.).
- fold trimmed sheets after collation.
- staple along fold if desired.
- punch for six-hole binder.

FOLD

TRIM HERE

WHEN
(xac) WHEN-COMPILED
WITH
WORDS
WORKING-STORAGE
WRITE
(xac) WRITE-ONLY
(spn) WRITE-VERIFY

ZERO
ZEROES
ZEROS

IBM © Reference Data

Operating System

IBM Full American National Standard COBOL

Appendix C: IBM Full American National Standard COBOL Format Summary and Reserved Words

The general format of a COBOL program is illustrated in these format summaries. Included within the general format is the specific format for each valid COBOL statement. All clauses are shown as though they were required by the COBOL source program, although within a given context many are optional. Several formats are included under special headings, which are different from, or additions to, the general format. Under these special headings are included formats peculiar to the following COBOL features: Sort, Report Writer, Table Handling, Segmentation, Source Program Library Facility, Debugging Language, Format Control of the Source Program Listing, Sterling Currency, Teleprocessing, and String Manipulation. Each of these features is explained within a special chapter of this publication — *IBM OS Full American National Standard COBOL*, Order Nos. GC28-6396-3, and -4.

Note: OS/VS COBOL formats are included and identified as OS/VS COBOL only.

TRIM HERE

20

IBM

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue
White Plains, New York 10604
[U.S.A. only]

IBM World Trade Corporation
821 United Nations Plaza
New York, New York 10017
[International]

Printed in U.S.A. Extracted from GC28-6396-4
Not orderable separately.

TRIM HERE

FOLD

FOLD

TRIM | HERE

IDENTIFICATION DIVISION — BASIC FORMATS

```
{IDENTIFICATION DIVISION.}
{ID 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 — BASIC FORMATS

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. computer-name.
OBJECT-COMPUTER. computer-name [MEMORY SIZE integer {WORDS     }]
                                                    {CHARACTERS}
                                                    {MODULES   }
    [SEGMENT-LIMIT IS priority-number].
SPECIAL-NAMES. [function-name IS mnemonic-name] ...
    [CURRENCY SIGN IS literal]
    [DECIMAL-POINT IS COMMA].
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    {SELECT [OPTIONAL] file name
    ASSIGN TO [integer-1] system-name-1 [system-name-2] ...
        [FOR MULTIPLE {REEL}]
                      {UNIT}
    RESERVE {NO        } ALTERNATE [AREA ]
            {integer-1}            [AREAS]
    {FILE-LIMIT IS  } {data-name-1} THRU {data-name-2}
    {FILE-LIMITS ARE} {literal-1  }      {literal-2  }
        [ {data-name-3} THRU {data-name-4} ] ...
          {literal-3  }      {literal-4  }
    ACCESS MODE IS {SEQUENTIAL}
                   {RANDOM    }
    PROCESSING MODE IS SEQUENTIAL
    ACTUAL KEY IS data-name
    NOMINAL KEY IS data-name
    RECORD KEY IS data-name
    TRACK-AREA IS {data-name} CHARACTERS
                  {integer  }
    TRACK-LIMIT IS integer [TRACK ] ) ...
                           [TRACKS]
I-O-CONTROL.
    RERUN ON system-name EVERY [integer RECORDS                ]
                               [END OF] {REEL} OF file-name
                                        {UNIT}
    SAME [RECORD] AREA FOR file-name-1 {file-name-2} ...
         [SORT  ]
    MULTIPLE FILE TAPE CONTAINS file-name-1 [POSITION integer-1]
        [file-name-2 [POSITION integer-2]] ...
    APPLY WRITE-ONLY ON file-name-1 [file-name-2] ...
    APPLY CORE-INDEX ON file-name-1 [file-name-2] ...
    APPLY RECORD-OVERFLOW ON file-name-1 [file-name-2] ...
    APPLY REORG-CRITERIA TO data-name ON file-name ...
```

NOTE: Format 2 of the RERUN Clause (for Sort Files) is included with Formats for the SORT feature.

2

(ca) PRINTING
PROCEDURE
(ca) PROCEDURES
PROCEED
(ca) PROCESS
PROCESSING
(xa) PROGRAM
PROGRAM-ID
(xa) QUEUE
QUOTE
QUOTES
RANDOM
RD
READ
(xac) READY
(xa) RECEIVE
RECORD
(xac) RECORD-OVERFLOW
(xa) RECORDING
RECORDS
REDEFINES
REEL
(ca) REFERENCES
(ca) RELATIVE
RELEASE
(xac) RELOAD
REMAINDER
REMARKS
(ca) REMOVAL
RENAMES
(xac) REORG-CRITERIA
REPLACING
REPORT
REPORTING
REPORTS
(xac) REREAD
RERUN
RESERVE
RESET
RETURN
(xac) RETURN-CODE
REVERSED
REWIND
(xa) REWRITE
RF
RH
RIGHT
ROUNDED
RUN
(ca) SA
SAME
SD
SEARCH
SECTION
SECURITY
SEEK
(xa) SEGMENT
SEGMENT-LIMIT
SELECT
(xa) SEND
SENTENCE
(xa) SEPARATE
SEQUENTIAL
(xac) SERVICE
SET
SIGN
SIZE
(xac) SKIP1
(xac) SKIP2
(xac) SKIP3
SORT
(xac) SORT-CORE-SIZE
(xac) SORT-FILE-SIZE
(ca) SORT-MERGE
(xac) SORT-MESSAGE
(xac) SORT-MODE-SIZE
(spn) SORT-OPTION
(xac) SORT-RETURN
SOURCE
SOURCE-COMPUTER
SPACE
SPACES
SPECIAL-NAMES
STANDARD
(xa) START
STATUS
STOP
(xa) STRING
(xa) SUB-QUEUE-1
(xa) SUB-QUEUE-2
(xa) SUB-QUEUE-3
SUBTRACT
SUM
(ca) SUPERVISOR
(xa) SUPPRESS
(ca) SUSPEND
(xa) SYMBOLIC
SYNC
SYNCHRONIZED
(sp) SYSIN
(spn) SYSIPT
(spn) SYSLST
(sp) SYSOUT
(spn) SYSPCH
(sp) SYSPUNCH
(sp) S01
(sp) S02
(ca) TABLE
TALLY
TALLYING
TAPE
(ca) TERMINAL
TERMINATE
(xa) TEXT
THAN
(xac) THEN
THROUGH
THRU
(xa) TIME
(xac) TIME-OF-DAY
TIMES
TO
(ca) TOP
(xac) TOTALED
(xac) TOTALING
(xac) TRACE
(xac) TRACK
(xac) TRACK-AREA
(xac) TRACK-LIMIT
(xac) TRACKS
(xa) TRAILING
(xac) TRANSFORM
TYPE
(ca) UNEQUAL
UNIT
(xa) UNSTRING
UNTIL
UP
UPON
(spn) UPSI-0
(spn) UPSI-1
(spn) UPSI-2
(spn) UPSI-3
(spn) UPSI-4
(spn) UPSI-5
(spn) UPSI-6
(spn) UPSI-7
USAGE
USE
USING
VALUE
VALUES
VARYING

TRIM | HERE

FOLD

FOLD

TRIM HERE

TRIM HERE

DECIMAL-POINT
DECLARATIVES
| (xa) DELETE
(xa) DELIMITED
(xa) DELIMITER
DEPENDING
(xa) DEPTH
DESCENDING
(xa) DESTINATION
DETAIL
(ca) DISABLE
(xac) DISP
DISPLAY
(xac) DISPLAY-ST
(ca) DISPLAY-n
DIVIDE
DIVISION
DOWN
| (ca) DUPLICATES
| (xa) DYNAMIC

(xa) ECI
(xac) EJECT
ELSE
(xa) EMI
(ca) ENABLE
END
END-OF-PAGE
(xa) ENDING
ENTER
(xac) ENTRY
ENVIRONMENT
(xa) EOP
EQUAL
(ca) EQUALS
ERROR
(xa) ESI
EVERY
EXAMINE
(ca) EXCEEDS
| (xa) EXCEPTION
(xac) EXHIBIT
EXIT
| (xa) EXTEND
(spn) EXTENDED-SEARCH

FD
FILE
FILE-CONTROL
FILE-LIMIT
FILE-LIMITS
FILLER
FINAL
FIRST
FOOTING
FOR
FROM

GENERATE
GIVING
GO
(xac) GOBACK
GREATER
GROUP

HEADING
HIGH-VALUE
HIGH-VALUES
(ca) HOLD

I-O
I-O-CONTROL
(xac) ID
IDENTIFICATION
IF
IN
INDEX
(ca) INDEX-n
INDEXED
INDICATE
(ca) INITIAL
| (ca) INITIALIZE
INITIATE
INPUT

18

INPUT-OUTPUT
(xac) INSERT
(ca) INSPECT
INSTALLATION
INTO
INVALID
IS

JUST
JUSTIFIED

KEY

LABEL
(xac) LABEL-RETURN
LAST
LEADING
(xac) LEAVE
LEFT
(xa) LENGTH
LESS
(ca) LIBRARY
LIMIT
LIMITS
(ca) LINAGE
(ca) LINAGE-COUNTER
LINE
LINE-COUNTER
LINES
(xa) LINKAGE
LOCK
LOW-VALUE
LOW-VALUES

(spn) MASTER-INDEX
MEMORY
(xa) MERGE
(xa) MESSAGE
MODE
MODULES
(xac) MORE-LABELS
MOVE
MULTIPLE
MULTIPLY

(xac) NAMED
NEGATIVE
NEXT
NO
(xac) NOMINAL
NOT
NOTE
(spn) NSTD-REELS
NUMBER
NUMERIC
(ca) NUMERIC-EDITED

OBJECT-COMPUTER
(ca) OBJECT-PROGRAM
OCCURS
OF
— OFF
OMITTED
ON
OPEN
OPTIONAL
OR
| (xa) ORGANIZATION
(xac) OTHERWISE
OUTPUT
— (xa) OVERFLOW

PAGE
PAGE-COUNTER
| (xac) PASSWORD
PERFORM
PF
PH
PIC
PICTURE
PLUS
(xa) POINTER
POSITION
(xac) POSITIONING
POSITIVE
— (xac) PRINT-SWITCH

3

DATA DIVISION — BASIC FORMATS

DATA DIVISION.
FILE SECTION.
FD *file-name*

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

RECORD CONTAINS [*integer-1* TO] *integer-2* CHARACTERS

RECORDING MODE IS *mode*

LABEL {RECORD IS / RECORDS ARE} {OMITTED / STANDARD / *data-name-1* [*data-name-2*] . . . [TOTALING AREA IS *data-name-3* TOTALED AREA IS *data-name-4*]}

VALUE OF *data-name-1* IS {*literal-1* / *data-name-2*} [*data-name-3* IS {*literal-2* / *data-name-4*}] . . .

DATA {RECORD IS / RECORDS ARE} *data-name-1* [*data-name-2*] . . . .

NOTE: Format for the REPORT Clause is included with Formats for the REPORT WRITER feature.

01-49 {*data-name-1* / FILLER}
REDEFINES *data-name-2*
BLANK WHEN ZERO
{JUSTIFIED / JUST} RIGHT
{PICTURE / PIC} IS *character string*
[SIGN IS] {LEADING / TRAILING} [SEPARATE CHARACTER] (Version 3 & 4)
{SYNCHRONIZED / SYNC} [LEFT / RIGHT]
[USAGE IS] {INDEX / DISPLAY / {COMPUTATIONAL / COMP} / {COMPUTATIONAL-1 / COMP-1} / {COMPUTATIONAL-2 / COMP-2} / {COMPUTATIONAL-3 / COMP-3} / {COMPUTATIONAL-4 / COMP-4} (Version 3 & 4) / DISPLAY-ST}

88 *condition-name* {VALUE IS / VALUES ARE} *literal-1* [THRU *literal-2*] [*literal-3* [THRU *literal-4*]] . . .

66 *data-name-1* RENAMES *data-name-2* [THRU *data-name-3*].

NOTE: Formats for the OCCURS Clause are included with Formats for the TABLE HANDLING feature.

WORKING-STORAGE SECTION.
77 *data-name-1*
01-49 {*data-name-1* / FILLER}
REDEFINES *data-name-2*
BLANK WHEN ZERO
{JUSTIFIED / JUST} RIGHT
{PICTURE / PIC} IS *character string*
[SIGN IS] {LEADING / TRAILING} [SEPARATE CHARACTER] (Version 3 & 4)
{SYNCHRONIZED / SYNC} [LEFT / RIGHT]

TRIM HERE

FOLD

FOLD

TRIM HERE

[USAGE IS] { INDEX / DISPLAY / {COMPUTATIONAL / COMP} / {COMPUTATIONAL-1 / COMP-1} / {COMPUTATIONAL-2 / COMP-2} / {COMPUTATIONAL-3 / COMP-3} / {COMPUTATIONAL-4 / COMP-4} / DISPLAY-ST } (Version 3 & 4)

VALUE IS *literal*

88 *condition-name* {VALUE IS / VALUES ARE} *literal-1* [THRU *literal-2*]
[*literal-3* [THRU *literal-4*]] ...

66 *data-name-1* RENAMES *data-name-2* [THRU *data-name-3*].

NOTE: Formats for the OCCURS Clause are included with Formats for the TABLE HANDLING feature.

LINKAGE SECTION.

77 *data-name-1*

01-49 {*data-name-2* / FILLER}

REDEFINES *data-name-2*

BLANK WHEN ZERO

{JUSTIFIED / JUST} RIGHT

{PICTURE / PIC} IS *character string*

[SIGN IS] {LEADING / TRAILING} [SEPARATE CHARACTER] (Version 3 & 4)

{SYNCHRONIZED / SYNC} [LEFT / RIGHT]

[USAGE IS] { INDEX / DISPLAY / {COMPUTATIONAL / COMP} / {COMPUTATIONAL-1 / COMP-1} / {COMPUTATIONAL-2 / COMP-2} / {COMPUTATIONAL-3 / COMP-3} / {COMPUTATIONAL-4 / COMP-4} / DISPLAY-ST } (Version 3 & 4)

88 *condition-name* {VALUE IS / VALUES ARE} *literal-1* [THRU *literal-2*]
[*literal-3* [THRU *literal-4*]] ...

66 *data-name-1* RENAMES *data-name-2* [THRU *data-name-3*].

NOTE: Formats for the OCCURS Clause are included with Formats for the TABLE HANDLING feature.

PROCEDURE DIVISION — BASIC FORMATS

{PROCEDURE DIVISION. / PROCEDURE DIVISION USING *identifier-1* [*identifier-2*] ... .}

ACCEPT Statement

FORMAT 1

ACCEPT *identifier* [FROM {SYSIN / CONSOLE / *mnemonic-name*}]

FORMAT 2 (Version 4)

ACCEPT *identifier* FROM {DATE / DAY / TIME}

IBM AMERICAN NATIONAL STANDARD COBOL RESERVED WORDS

No word in the following list should appear as a programmer-defined name. The keys that appear before some of the words, and their meanings, are:

(xa) before a word means that the word is an IBM extension to American National Standard COBOL.

(xac) before a word means that the word is an IBM extension to both American National Standard COBOL and CODASYL COBOL.

(ca) before a word means that the word is a CODASYL COBOL reserved word not incorporated in American National Standard COBOL or in IBM American National Standard COBOL.

(sp) before a word means that the word is an IBM function-name established in support of the SPECIAL-NAMES function.

(spn) before a word means that the word is used by an IBM American National Standard COBOL compiler, but not this compiler.

(asm) before a word means that the word is defined by American National Standard COBOL, but is not used by this compiler.

ACCEPT
ACCESS
ACTUAL
ADD
(asm) ADDRESS
ADVANCING
AFTER
ALL
ALPHABETIC
(ca) ALPHANUMERIC
(ca) ALPHANUMERIC-EDITED
ALTER
ALTERNATE
AND
(xa) APPLY
ARE
AREA
AREAS
ASCENDING
ASSIGN
AT
AUTHOR

(xac) BASIS
BEFORE
BEGINNING
BLANK
BLOCK
(ca) BOTTOM
BY

(xa) CALL
(xa) CANCEL
(xac) CBL
(xa) CD
CF
CH
(xac) CHANGED
(xa) CHARACTER
CHARACTERS
(asm) CLOCK-UNITS
CLOSE
(asm) COBOL
CODE
COLUMN
(spn) COM-REG
COMMA
(xa) COMMUNICATION
COMP
(xa) COMP-1
(xa) COMP-2
(xa) COMP-3
(xa) COMP-4
COMPUTATIONAL
(xa) COMPUTATIONAL-1
(xa) COMPUTATIONAL-2
(xa) COMPUTATIONAL-3
(xa) COMPUTATIONAL-4
COMPUTE
CONFIGURATION
(sp) CONSOLE
CONTAINS
CONTROL
CONTROLS
COPY
(xac) CORE-INDEX
CORR
CORRESPONDING
(xa) COUNT
(sp) CSP
CURRENCY
(xac) CURRENT-DATE
(spn) CYL-INDEX
(spn) CYL-OVERFLOW
(sp) C01
(sp) C02
(sp) C03
(sp) C04
(sp) C05
(sp) C06
(sp) C07
(sp) C08
(sp) C09
(sp) C10
(sp) C11
(sp) C12

DATA
(xa) DATE
DATE-COMPILED
DATE-WRITTEN
(xa) DAY
(ca) DAY-OF-WEEK
DE
(xac) DEBUG
(ca) DEBUG-CONTENTS
(ca) DEBUG-ITEM
(ca) DEBUG-LINE
(ca) DEBUG-NAME
(ca) DEBUG-SUB-1
(ca) DEBUG-SUB-2
(ca) DEBUG-SUB-3
(ca) DEBUGGING

TRIM HERE

FOLD

FOLD

TRIM HERE

TRIM HERE

REWRITE Statement

REWRITE *record-name* [FROM *identifier*]
[INVALID KEY *imperative-statement*]

START Statement

START *file-name* [KEY IS { EQUAL TO | = | GREATER THAN | > | NOT LESS THAN | NOT < }]
[INVALID KEY *imperative-statement*]

USE Sentence

USE AFTER STANDARD { EXCEPTION | ERROR } PROCEDURE
ON { *file-name-1* [*file-name-2*] ... | INPUT | OUTPUT | I-O | EXTEND }.

WRITE Statement

WRITE *record-name* [FROM *identifier*]
[INVALID KEY *imperative-statement*]

MERGE FACILITY FORMATS (OS/VS COBOL Only)

Environment Division — Input-Output Section

FILE-CONTROL Entry

FILE-CONTROL.
(SELECT *file-name*
ASSIGN TO *system-name-1* [*system-name-2*] ...) ...

I-O-CONTROL Entry

I-O-CONTROL.
SAME { SORT | SORT-MERGE | RECORD } AREA FOR *file-name-1* [*file-name-2*] ... .

Data Division — Merge File Description Entry

SD *merge-file-name*
[RECORD CONTAINS [*integer-1* TO] *integer-2* CHARACTERS]
[DATA { RECORD IS | RECORDS ARE } *data-name-1* [*data-name-2*] ...].

Procedure Division — Merge Statement

MERGE *file-name-1*
ON { ASCENDING | DESCENDING } KEY *data-name-1* [*data-name-2*] ...
[ON { ASCENDING | DESCENDING } KEY *data-name-3* [*data-name-4*] ...] ...
USING *file-name-2 file-name-3* [*file-name-4*] ...
{ GIVING *file-name-5* | OUTPUT PROCEDURE IS *section-name-1* [THRU *section-name-2*] }

16

ADD Statement

FORMAT 1

ADD { *identifier-1* | *literal-1* } [*identifier-2* | *literal-2*] ... TO *identifier-m* [ROUNDED]
[*identifier-n* [ROUNDED]] ... [ON SIZE ERROR *imperative-statement*]

FORMAT 2

ADD { *identifier-1* | *literal-1* } { *identifier-2* | *literal-2* } [*identifier-3* | *literal-3*] ... GIVING
identifier-m [ROUNDED] [ON SIZE ERROR *imperative-statement*]

FORMAT 3

ADD { CORRESPONDING | CORR } *identifier-1* TO *identifier-2* [ROUNDED]
[ON SIZE ERROR *imperative-statement*]

ALTER Statement

ALTER *procedure-name-1* TO [PROCEED TO] *procedure-name-2*
[*procedure-name-3* TO [PROCEED TO] *procedure-name-4*] ...

CALL Statement

FORMAT 1

CALL *literal-1* [USING *identifier-1* [*identifier-2*] ...]

FORMAT 2 (Version 4)

CALL *identifier-1* [USING *identifier-2* [*identifier-3*] ...]

CANCEL Statement (Version 4)

CANCEL { *literal-1* | *identifier-1* } [*literal-2* | *identifier-2*] ...

CLOSE Statement

FORMAT 1

CLOSE *file-name-1* [REEL | UNIT] [WITH { NO REWIND | LOCK }]
[*file-name-2* [REEL | UNIT] [WITH { NO REWIND | LOCK }]] ...

FORMAT 2

CLOSE *file-name-1* [WITH { NO REWIND | LOCK | DISP }]
[*file-name-2* [WITH { NO REWIND | LOCK | DISP }]] ...

FORMAT 3

CLOSE *file-name-1* { REEL | UNIT } [WITH { NO REWIND | LOCK | POSITIONING }]
[*file-name-2* { REEL | UNIT } [WITH { NO REWIND | LOCK | POSITIONING }]] ...

COMPUTE Statement

COMPUTE *identifier-1* [ROUNDED] = { *identifier-2* | *literal-1* | *arithmetic-expression* }
[ON SIZE ERROR *imperative-statement*]

DECLARATIVE Section

PROCEDURE DIVISION.
DECLARATIVES.
{*section-name* SECTION. USE *sentence*.
{*paragraph-name*. {*sentence*} ...} ...} ...
END DECLARATIVES.

DISPLAY Statement

DISPLAY { *literal-1* | *identifier-1* } [*literal-2* | *identifier-2*] ... [UPON { CONSOLE | SYSPUNCH | SYSOUT | *mnemonic-name* }]

5

TRIM HERE

FOLD

FOLD

TRIM HERE

DIVIDE Statement

FORMAT 1

```
DIVIDE {identifier-1 | literal-1} INTO identifier-2 [ROUNDED]
       [ON SIZE ERROR imperative-statement]
```

FORMAT 2

```
DIVIDE {identifier-1 | literal-1} {INTO | BY} {identifier-2 | literal-2} GIVING identifier-3
  [ROUNDED] [REMAINDER identifier-4] [ON SIZE ERROR imperative-statement]
```

ENTER Statement

```
ENTER language-name [routine-name].
```

ENTRY Statement

```
ENTRY literal-1 [USING identifier-1 [identifier-2] . . .]
```

EXAMINE Statement

FORMAT 1

```
EXAMINE identifier TALLYING {UNTIL FIRST | ALL | LEADING} literal-1
        [REPLACING BY literal-2]
```

FORMAT 2

```
EXAMINE identifier REPLACING {ALL | LEADING | FIRST | UNTIL FIRST} literal-1 BY literal-2
```

EXIT Statement

```
paragraph-name. EXIT [PROGRAM].
```

GOBACK Statement

```
GOBACK.
```

GO TO Statement

FORMAT 1

```
GO TO procedure-name-1
```

FORMAT 2

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

FORMAT 3

```
GO TO.
```

IF Statement

```
IF condition THEN {statement-1 | NEXT SENTENCE} {ELSE | OTHERWISE} {statement-2 | NEXT SENTENCE}
```

MOVE Statement

FORMAT 1

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

FORMAT 2

```
MOVE {CORRESPONDING | CORR} identifier-1 TO identifier-2
```

MULTIPLY Statement

FORMAT 1

```
MULTIPLY {identifier-1 | literal-1} BY identifier-2 [ROUNDED]
         [ON SIZE ERROR imperative-statement]
```

FORMAT 2

```
MULTIPLY {identifier-1 | literal-1} BY {identifier-2 | literal-2} GIVING identifier-3
  [ROUNDED] [ON SIZE ERROR imperative-statement]
```

VSAM FORMATS (OS/VS COBOL Only)

Environment Division — File-Control Entry

FORMAT 1 — Sequential VSAM Files

```
FILE-CONTROL.
  {SELECT [OPTIONAL] file-name
  ASSIGN TO system-name-1 [system-name-2] . . .
  [RESERVE integer [AREA | AREAS]]
  [ORGANIZATION IS SEQUENTIAL]
  [ACCESS MODE IS SEQUENTIAL]
  [PASSWORD IS data-name-1]
  [FILE STATUS IS data-name-2].} . . .
```

FORMAT 2 — Indexed VSAM Files

```
FILE-CONTROL.
  {SELECT file-name
  ASSIGN TO system-name-1 [system-name-2] . . .
  [RESERVE integer [AREA | AREAS]]
  ORGANIZATION IS INDEXED
  [ACCESS MODE IS {SEQUENTIAL | RANDOM | DYNAMIC}]
  RECORD KEY IS data-name-3
  [PASSWORD IS data-name-1]
  [FILE STATUS IS data-name-2].} . . .
```

Environment Division — I-O-Control Entry

```
I-O-CONTROL.
  [RERUN ON system-name EVERY integer RECORDS
    OF file-name-1] . . .
  [SAME [RECORD] AREA
    FOR file-name-2 [file-name-3] . . .] . . .
```

Data Division

LABEL RECORDS Clause

```
LABEL {RECORD IS | RECORDS ARE} {STANDARD | OMITTED}
```

NOTE: Other Data Division clauses have the same syntax for VSAM files that they have for other files.

Procedure Division

CLOSE Statement

```
CLOSE file-name-1 [WITH LOCK]
      [file-name-2 [WITH LOCK]] . . .
```

DELETE Statement

```
DELETE file-name RECORD
  [INVALID KEY imperative-statement]
```

OPEN Statement

```
OPEN {INPUT file-name-1 [file-name-2] . . .
     | OUTPUT file-name-1 [file-name-3] . . .
     | I-O file-name-1 [file-name-2] . . .
     | EXTEND file-name-1 [file-name-2] . . .} . . .
```

READ Statement

FORMAT 1

```
READ file-name [NEXT] RECORD [INTO identifier]
  [AT END imperative-statement]
```

FORMAT 2

```
READ file-name RECORD [INTO identifier]
  [INVALID KEY imperative-statement]
```

15

TRIM HERE

TRIM HERE

FOLD

FOLD

TRIM HERE

TRIM HERE

STRING MANIPULATION — BASIC FORMATS

STRING Statement

```
STRING {identifier-1 | literal-1} [identifier-2 | literal-2] ... DELIMITED BY {identifier-3 | literal-3 | SIZE}
    [{identifier-4 | literal-4} [identifier-5 | literal-5] ... DELIMITED BY {identifier-6 | literal-6 | SIZE}] ...
    INTO identifier-7 [WITH POINTER identifier-8]
    [ON OVERFLOW imperative-statement]
```

UNSTRING Statement

```
UNSTRING identifier-1
    [DELIMITED BY [ALL] {identifier-2 | literal-2} [OR [ALL] {identifier-3 | literal-3}] ...]
    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]
```

NOTE Statement

```
NOTE character string
```

OPEN Statement

FORMAT 1

```
OPEN [INPUT {file-name [REVERSED | WITH NO REWIND]} ...]
     [OUTPUT {file-name [WITH NO REWIND]} ...]
     [I-O {file-name} ...]
```

FORMAT 2

```
OPEN [INPUT {file-name [REVERSED | WITH NO REWIND] [LEAVE | REREAD | DISP]} ...]
     [OUTPUT {file-name [WITH NO REWIND] [LEAVE | REREAD | DISP]} ...]
     [I-O {file-name} ...]
```

PERFORM Statement

FORMAT 1

```
PERFORM procedure-name-1 [THRU procedure-name-2]
```

FORMAT 2

```
PERFORM procedure-name-1 [THRU procedure-name-2] {identifier-1 | integer-1} TIMES
```

FORMAT 3

```
PERFORM procedure-name-1 [THRU procedure-name-2] UNTIL condition-1
```

FORMAT 4

```
PERFORM procedure-name-1 [THRU procedure-name-2]
    VARYING {index-name-1 | identifier-1} FROM {index-name-2 | literal-2 | identifier-2} BY {literal-3 | identifier-3} UNTIL condition-1
    [AFTER {index-name-4 | identifier-4} FROM {index-name-5 | literal-5 | identifier-5} BY {literal-6 | identifier-6} UNTIL condition-2
    [AFTER {index-name-7 | identifier-7} FROM {index-name-8 | literal-8 | identifier-8} BY {literal-9 | identifier-9} UNTIL condition-3]]
```

READ Statement

```
READ file-name RECORD [INTO identifier]
        {AT END | INVALID KEY} imperative-statement
```

REWRITE Statement

```
REWRITE record-name [FROM identifier] [INVALID KEY imperative-statement]
```

SEEK Statement

```
SEEK file-name RECORD
```

START Statement

FORMAT 1

```
START file-name [INVALID KEY imperative-statement]
```

FORMAT 2 (Version 3 & 4)

```
START file-name
    USING KEY data-name {EQUAL TO | =} identifier
    [INVALID KEY imperative-statement]
```

STOP Statement

```
STOP {RUN | literal}
```

14

7

TRIM HERE

FOLD

FOLD

TRIM HERE

SUBTRACT Statement

FORMAT 1

```
SUBTRACT {identifier-1} [identifier-2] ... FROM identifier-m [ROUNDED]
         {literal-1   } [literal-2   ]

      [identifier-n [ROUNDED]] ... [ON SIZE ERROR imperative-statement]
```

FORMAT 2

```
SUBTRACT {identifier-1} [identifier-2] ... FROM {identifier-m} GIVING identifier-n
         {literal-1   } [literal-2   ]          {literal-m   }

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

FORMAT 3

```
SUBTRACT {CORRESPONDING} identifier-1 FROM identifier-2 [ROUNDED]
         {CORR         }

      [ON SIZE ERROR imperative-statement]
```

TRANSFORM Statement

```
TRANSFORM identifier-3 CHARACTERS FROM {figurative-constant-1}
                                       {nonnumeric-literal-1 }
                                       {identifier-1         }

      TO {figurative-constant-2}
         {nonnumeric-literal-2 }
         {identifier-2         }
```

USE Sentence

FORMAT 1

Option 1:

```
USE {BEFORE} STANDARD [BEGINNING] [REEL]
    {AFTER }                      [FILE]
                                  [UNIT]

      LABEL PROCEDURE ON {{file-name} ...}
                         {OUTPUT         }.
                         {INPUT          }
                         {I-O            }
```

Option 2:

```
USE {BEFORE} STANDARD [ENDING] [REEL]
    {AFTER }                   [FILE]
                               [UNIT]

      LABEL PROCEDURE ON {{file-name} ...}
                         {OUTPUT         }.
                         {INPUT          }
                         {I-O            }
```

FORMAT 2

```
USE AFTER STANDARD ERROR PROCEDURE

      ON {{file-name-1} [file-name-2] ...}
         { INPUT                         }
         { OUTPUT                        }
         { I-O                           }
         [GIVING data-name-1 [data-name-2]].
```

NOTE: Format 3 of the USE Sentence is included in Formats for the REPORT WRITER feature.

WRITE Statement

FORMAT 1

```
WRITE record-name [FROM identifier-1] [ {BEFORE} ADVANCING
                                        {AFTER }

      {identifier-2 LINES}
      {integer LINES     } ] [AT {END-OF-PAGE} imperative-statement]
      {mnemonic-name     }       {EOP        }
```

FORMAT 2

```
WRITE record-name [FROM identifier-1] AFTER POSITIONING {identifier-2} LINES
                                                        {integer     }

      [AT {END-OF-PAGE} imperative-statement]
          {EOP        }
```

FORMAT 3

```
WRITE record-name [FROM identifier-1] INVALID KEY imperative-statement
```

8

FORMAT CONTROL — BASIC FORMATS

EJECT Statement

```
1         Area B
          EJECT
```

SKIP1, SKIP2, SKIP3 Statements

```
1         Area B
          {SKIP1}
          {SKIP2}
          {SKIP3}
```

STERLING CURRENCY — BASIC FORMATS

Data Division Sterling Formats

Nonreport PICTURE Clause

```
{PICTURE} IS 9 [(n)] D [8] 8D {6[6]} [[V] 9 [(n)]] [USAGE IS] DISPLAY-ST
{PIC    }                     {7[7]}
```

Report PICTURE Clause

```
{PICTURE} IS
{PIC    }
```

[pound-report-string] [pound-separator-string] delimiter shilling-report-string [shilling-separator-string] delimiter pence-report-string [pence-separator-string] [sign-string] [USAGE IS] DISPLAY-ST

PROGRAM PRODUCT INFORMATION — VERSION 4

TELEPROCESSING — BASIC FORMATS

Data Division Teleprocessing Formats

CD Entry

FORMAT 1

```
CD cd-name FOR INPUT
  [[[SYMBOLIC QUEUE           IS data-name-1]
    [SYMBOLIC SUB-QUEUE-1     IS data-name-2]
    [SYMBOLIC SUB-QUEUE-2     IS data-name-3]
    [SYMBOLIC SUB-QUEUE-3     IS data-name-4]
    [MESSAGE DATE             IS data-name-5]
    [MESSAGE TIME             IS data-name-6]
    [SYMBOLIC SOURCE          IS data-name-7]
    [TEXT LENGTH              IS data-name-8]
    [END KEY                  IS data-name-9]
    [STATUS KEY               IS data-name-10]
    [QUEUE DEPTH              IS data-name-11]]
    [data-name-1 data-name-2 . . . data-name-11]].
```

FORMAT 2

```
CD cd-name FOR OUTPUT
    [DESTINATION COUNT        IS data-name-1]
    [TEXT LENGTH              IS data-name-2]
    [STATUS KEY               IS data-name-3]
    [ERROR KEY                IS data-name-4]
    [SYMBOLIC DESTINATION     IS data-name-5].
```

Procedure Division Teleprocessing Formats

Message Condition

```
[NOT] MESSAGE FOR cd-name
```

RECEIVE Statement

```
RECEIVE cd-name {MESSAGE} INTO identifier-1
                {SEGMENT}
   [NO DATA imperative-statement]
```

SEND Statement

FORMAT 1

```
SEND cd-name FROM identifier-1
```

FORMAT 2

```
                                       {WITH identifier-2}
SEND cd-name [FROM identifier-1]       {WITH ESI         }
                                       {WITH EMI         }
                                       {WITH EGI         }
```

13

TRIM HERE

TRIM HERE

FOLD

FOLD

TRIM HERE

SEGMENTATION — BASIC FORMATS

Environment Division Segmentation Formats

OBJECT-COMPUTER PARAGRAPH

SEGMENT-LIMIT Clause

SEGMENT-LIMIT IS *priority-number*

Procedure Division Segmentation Formats

Priority Numbers

section-name SECTION [*priority-number*].

SOURCE PROGRAM LIBRARY FACILITY

COPY Statement

```
COPY library-name [SUPPRESS]
    [REPLACING word-1 BY { word-2 | literal-1 | identifier-1 } [word-3 BY { word-4 | literal-2 | identifier-2 }] ...].
```

Extended Source Program Library Facility

BASIS Card

```
BASIS        library-name
```

INSERT Card

```
INSERT       sequence-number-field
```

DELETE Card

```
DELETE       sequence-number-field
```

DEBUGGING LANGUAGE — BASIC FORMATS

Procedure Division Debugging Formats

EXHIBIT Statement

```
EXHIBIT { NAMED | CHANGED NAMED | CHANGED } { identifier-1 | nonnumeric-literal-1 } [ identifier-2 | nonnumeric-literal-2 ] ...
```

ON (Count-Conditional) Statement

FORMAT 1

```
ON integer-1 [AND EVERY integer-2] [UNTIL integer-3]
        { imperative-statement | NEXT SENTENCE } { ELSE | OTHERWISE } { statement ... | NEXT SENTENCE }
```

FORMAT 2 (Version 3 & 4)

```
ON { integer-1 | identifier-1 } [AND EVERY { integer-2 | identifier-2 }] [UNTIL { integer-3 | identifier-3 }]
    { imperative-statement | NEXT SENTENCE } { ELSE | OTHERWISE } { statement ... | NEXT SENTENCE }
```

TRACE Statement

```
{ READY | RESET } TRACE
```

Compile-Time Debugging Packet

DEBUG Card

```
DEBUG        location
```


SORT — BASIC FORMATS

Environment Division Sort Formats

FILE-CONTROL PARAGRAPH — SELECT SENTENCE

SELECT Sentence (for GIVING option only)

```
SELECT file-name
    ASSIGN TO [integer-1] system-name-1 [system-name-2] ...
        OR system-name-3 [FOR MULTIPLE { REEL | UNIT }]
        [RESERVE { integer-2 | NO } ALTERNATE [ AREA | AREAS ]].
```

SELECT Sentence (for Sort Work Files)

```
SELECT sort-file-name
    ASSIGN TO [integer] system-name-1 [system-name-2] ...
```

I-O-CONTROL PARAGRAPH

RERUN Clause

```
RERUN ON system-name
```

SAME RECORD/SORT AREA Clause

```
SAME { RECORD | SORT } AREA FOR file-name-1 (file-name-2) ...
```

Data Division Sort Formats

SORT-FILE DESCRIPTION

```
SD sort-file-name
    RECORDING MODE IS mode
    DATA { RECORD IS | RECORDS ARE } data-name-1 [data-name-2] ...
    RECORD CONTAINS [integer-1 TO] integer-2 CHARACTERS
    [LABEL { RECORD IS | RECORDS ARE } { STANDARD | OMITTED }]. (Version 4)
```

Procedure Division Sort Formats

RELEASE Statement

```
RELEASE sort-record-name [FROM identifier]
```

RETURN Statement

```
RETURN sort-file-name RECORD [INTO identifier]
    AT END imperative-statement
```

SORT Statement

```
SORT file-name-1 ON { DESCENDING | ASCENDING } KEY {data-name-1} ...
    [ON { DESCENDING | ASCENDING } KEY {data-name-2} ...] ...
    { INPUT PROCEDURE IS section-name-1 [THRU section-name-2] | USING file-name-2 }
    { OUTPUT PROCEDURE IS section-name-3 [THRU section-name-4] | GIVING file-name-3 }
```

REPORT WRITER — BASIC FORMATS

Data Division Report Writer Formats

NOTE: Formats that appear as Basic Formats within the general description of the Data Division are illustrated there.

FILE SECTION — REPORT Clause

```
{ REPORT IS | REPORTS ARE } report-name-1 [report-name-2] ...
```


TRIM HERE

FOLD

FOLD

TRIM HERE

REPORT SECTION

REPORT SECTION.
RD *report-name*
[WITH] CODE *mnemonic-name*
{CONTROL IS / CONTROLS ARE} {FINAL / *identifier-1* [*identifier-2*] ... / FINAL *identifier-1* [*identifier-2*] ...}
PAGE [LIMIT IS / LIMITS ARE] *integer-1* {LINE / LINES}
[HEADING *integer-2*]
[FIRST DETAIL *integer-3*]
[LAST DETAIL *integer-4*]
[FOOTING *integer-5*].

REPORT GROUP DESCRIPTION ENTRY

FORMAT 1

01 [*data-name-1*]
LINE NUMBER IS {*integer-1* / PLUS *integer-2* / NEXT PAGE}
NEXT GROUP IS {*integer-1* / PLUS *integer-2* / NEXT PAGE}
TYPE IS {REPORT HEADING / RH / PAGE HEADING / PH / CONTROL HEADING / CH} {*identifier-n* / FINAL}
{DETAIL / DE}
{CONTROL FOOTING / CF} {*identifier-n* / FINAL}
{PAGE FOOTING / PF / REPORT FOOTING / RF}
USAGE Clause.

FORMAT 2

nn [*data-name-1*]
LINE Clause — See Format 1
USAGE Clause.

FORMAT 3

nn [*data-name-1*]
COLUMN NUMBER IS *integer-1*
GROUP INDICATE
JUSTIFIED Clause
LINE Clause — See Format 1
PICTURE Clause
RESET ON {*identifier-1* / FINAL}
BLANK WHEN ZERO Clause
SOURCE IS {TALLY / *identifier-2*}
SUM {TALLY / *identifier-3*} [TALLY / *identifier-4*] ... [UPON *data-name*]
VALUE IS *literal-1*
USAGE Clause.

FORMAT 4

01 *data-name-1*
BLANK WHEN ZERO Clause
COLUMN Clause — See Format 3
GROUP Clause — See Format 3
JUSTIFIED Clause
LINE Clause — See Format 1
NEXT GROUP Clause — See Format 1
PICTURE Clause
RESET Clause — See Format 3
{SOURCE Clause / SUM Clause / VALUE Clause} See Format 3
TYPE Clause — See Format 1
USAGE Clause.

Procedure Division Report Writer Formats

GENERATE Statement

GENERATE *identifier*

INITIATE Statement

INITIATE *report-name-1* [*report-name-2*] ...

TERMINATE Statement

TERMINATE *report-name-1* [*report-name-2*] ...

USE Sentence

USE BEFORE REPORTING *data-name*.

TABLE HANDLING — BASIC FORMATS

Data Division Table Handling Formats

OCCURS Clause

FORMAT 1

OCCURS *integer-2* TIMES
[{ASCENDING / DESCENDING} KEY IS *data-name-2* [*data-name-3* ...] ...
[INDEXED BY *index-name-1* [*index-name-2*] ...]

FORMAT 2

OCCURS *integer-1* TO *integer-2* TIMES [DEPENDING ON *data-name-1*]
[{ASCENDING / DESCENDING} KEY IS *data-name-2* [*data-name-3*] ...] ...
[INDEXED BY *index-name-1* [*index-name-2*] ...]

FORMAT 3

OCCURS *integer-2* TIMES [DEPENDING ON *data-name-1*]
[{ASCENDING / DESCENDING} KEY IS *data-name-2* [*data-name-3*] ...] ...
[INDEXED BY *index-name-1* [*index-name-2*] ...]

USAGE Clause

[USAGE IS] INDEX

Procedure Division Table Handling Formats

SEARCH Statement

FORMAT 1

SEARCH *identifier-1* [VARYING {*index-name-1* / *identifier-2*}]
[AT END *imperative-statement-1*]
WHEN *condition-1* {*imperative-statement-2* / NEXT SENTENCE}
[WHEN *condition-2* {*imperative-statement-3* / NEXT SENTENCE}] ...

FORMAT 2

SEARCH ALL *identifier-1* [AT END *imperative-statement-1*]
WHEN *condition-1* {*imperative-statement-2* / NEXT SENTENCE}

SET Statement

FORMAT 1

SET {*index-name-1* [*index-name-2*] ... / *identifier-1* [*identifier-2*] ...} TO {*index-name-3* / *identifier-3* / *literal-1*}

FORMAT 2

SET *index-name-4* [*index-name-5*] ... {UP BY / DOWN BY} {*identifier-4* / *literal-2*}

TRIM HERE

TRIM HERE

FOLD

APPENDIX D: SUMMARY OF FILE-PROCESSING TECHNIQUES AND APPLICABLE STATEMENTS AND CLAUSES

This appendix summarizes the statements and clauses that may be specified for each file-processing technique. In addition, each file-name must be specified in a SELECT clause in the Environment Division and must be defined by an FD entry in the File Section of the Data Division.

STANDARD SEQUENTIAL FILES – Required and Optional Entries

Device Type	Required Entries					Optional Entries						
	System-name	LABEL RECORDS	OPEN	CLOSE	Access Verbs	APPLY[3]	RESERVE	ACCESS	Other ENVIRONMENT DIVISION Clauses	BLOCK CONTAINS[4]	RECORDING MODE	USE
Reader	UR [-xxxx] -S-name	OMITTED	INPUT	[LOCK]	READ [INTO] AT END		{integer / NO}	SEQUENTIAL	SAME [RECORD] AREA RERUN	[n TO] m	{F / U / V}	ERROR
Punch	UR [-xxxx] -S-name	OMITTED	OUTPUT	[LOCK]	WRITE [FROM] [{BEFORE / AFTER} ADVANCING] [AFTER POSITIONING]	WRITE-ONLY (V-mode only)	{integer / NO}	SEQUENTIAL	SAME [RECORD] AREA RERUN	[n TO] m	{F / U / V}	ERROR
Printer	UR [-xxxx] -S-name	OMITTED	OUTPUT	[LOCK]	WRITE [FROM] [{BEFORE / AFTER} ADVANCING] [AFTER POSITIONING]	WRITE-ONLY (V-mode only)	{integer / NO}	SEQUENTIAL	SAME [RECORD] AREA RERUN	[n TO] m	{F / U / V}	ERROR REPORTING
					[END-OF-PAGE]		NO					
Tape	UT [-xxxx] -S-name	{STANDARD / OMITTED / data-name} [TOTALING-TOTALED]	INPUT [REVERSED / NO REWIND] [LEAVE / REREAD / DISP]	[REEL] [LOCK / NO REWIND / POSITIONING / DISP]	READ [INTO] AT END	WRITE-ONLY (V-mode only)	{integer / NO}	SEQUENTIAL	SAME [RECORD] AREA RERUN MULTIPLE FILE TAPE	[n TO] m	{F / U / V / S}	LABEL ERROR
			OUTPUT [NO REWIND] [LEAVE / REREAD / DISP]	[REEL] [LOCK / NO REWIND / POSITIONING / DISP]	WRITE[1] [FROM] [{BEFORE / AFTER} ADVANCING] [AFTER POSITIONING]							LABEL ERROR REPORTING
Mass Storage	{UT / DA} [-xxxx] -S-name	{STANDARD / OMITTED / data-name} [TOTALING-TOTALED]	INPUT	[UNIT] [LOCK]	READ [INTO] AT END	RECORD-OVERFLOW (not for S-mode)	{integer / NO}	SEQUENTIAL	SAME [RECORD] AREA RERUN	[n TO] m	{F / U / V / S}	AFTER LABEL ERROR
			OUTPUT	[UNIT] [LOCK]	WRITE[1] [FROM] INVALID KEY WRITE[1] [FROM] [{BEFORE / AFTER} ADVANCING] [AFTER POSITIONING]	WRITE-ONLY (V-mode only)						AFTER LABEL ERROR REPORTING
			I-O	[LOCK]	READ [INTO] AT END WRITE[2] [FROM] INVALID KEY REWRITE[2] [FROM] [INVALID KEY]							AFTER LABEL ERROR

[1]Create [2]Update [3]These APPLY clauses have meaning only for OUTPUT files; however, the compiler accepts them when the same file is opened for INPUT or I-O. [4]Not for U mode

DIRECT FILES (mass storage devices only) – Required and Optional Entries

Required Entries							Optional Entries			
ACCESS	KEY	System-name	LABEL RECORDS	OPEN	CLOSE	Access Verbs	RECORDING MODE	APPLY[5]	Other ENVIRONMENT DIVISION Clauses	USE
[SEQUENTIAL]	[ACTUAL]	DA [-xxxx]-D-name	{STANDARD / data-name}	INPUT	[UNIT] [LOCK]	READ [INTO] AT END	{F / U / V / S}		SAME [RECORD] AREA RERUN	AFTER LABEL ERROR
[SEQUENTIAL]	ACTUAL	DA [-xxxx]-D-name	{STANDARD / data-name}	OUTPUT	[UNIT] [LOCK]	WRITE[1] [FROM] INVALID KEY	F	RECORD-OVERFLOW	SAME [RECORD] AREA TRACK-LIMIT RERUN	AFTER LABEL ERROR
							{U / V / S}			
RANDOM	ACTUAL	DA [-xxxx]-D-name	{STANDARD / data-name}	INPUT	[LOCK]	SEEK READ [INTO] INVALID KEY	{F / U / V / S}		SAME [RECORD] AREA RERUN ON RECORDS	AFTER LABEL ERROR
				OUTPUT	[LOCK]	SEEK WRITE[1] [FROM] INVALID KEY	F	RECORD-OVERFLOW	SAME [RECORD] AREA TRACK-LIMIT RERUN ON RECORDS	
							{U / V / S}			
				I-O	[LOCK]	SEEK READ [INTO] INVALID KEY WRITE[2] [FROM] INVALID KEY	{F / U / V / S}		SAME [RECORD] AREA RERUN ON RECORDS	
RANDOM	ACTUAL	DA [-xxxx]-W-name	{STANDARD / data-name}	INPUT	[LOCK]	SEEK READ [INTO] INVALID KEY	{F / U / V / S}		SAME [RECORD] AREA RERUN ON RECORDS	AFTER LABEL ERROR
				OUTPUT	[LOCK]	SEEK WRITE[1] [FROM] INVALID KEY	F	RECORD-OVERFLOW	SAME [RECORD] AREA TRACK-LIMIT RERUN ON RECORDS	
							{U / V / S}			
RANDOM	ACTUAL	DA [-xxxx]-W-name	{STANDARD / data-name}	I-O	[LOCK]	SEEK READ [INTO] INVALID KEY WRITE[3] [FROM] INVALID KEY REWRITE[4] [FROM] [INVALID KEY]	{F / U / V / S}		SAME [RECORD] AREA RERUN ON RECORDS	AFTER LABEL ERROR

[1]Create [2]Update and add [3]Add [4]Update [5]These APPLY clauses have meaning only for OUTPUT files; however, the compiler accepts them when the same file is opened for INPUT or I-O.

INDEXED FILES (mass storage devices only) – Required and Optional Entries

Required Entries							Optional Entries					
ACCESS	KEY	System-name	LABEL RECORDS	OPEN	CLOSE	Access Verbs	APPLY[4]	RESERVE	Other ENVIRONMENT DIVISION Clauses	RECORDING MODE	BLOCK CONTAINS	USE
[SEQUENTIAL]	RECORD	DA [-xxxx]-I-name	STANDARD	INPUT	[LOCK]	READ [INTO] AT END		{ integer / NO }	SAME [RECORD] AREA RERUN	F	m	ERROR
	RECORD NOMINAL					START [INVALID KEY]						
	RECORD [NOMINAL]					START USING KEY [INVALID KEY]						
	RECORD			OUTPUT	[LOCK]	WRITE[1] [FROM] INVALID KEY						
	RECORD			I-O	[LOCK]	READ [INTO] AT END REWRITE[2] [FROM] [INVALID KEY]						
	RECORD NOMINAL					START [INVALID KEY]						
	RECORD [NOMINAL]					START USING KEY [INVALID KEY]						
RANDOM	RECORD NOMINAL	DA [-xxxx]-I-name	STANDARD	INPUT	[LOCK]	READ [INTO] INVALID KEY	REORG-CRITERIA CORE-INDEX	NO	SAME [RECORD] AREA RERUN ON RECORDS	F	m	ERROR
				I-O	[LOCK]	READ [INTO] INVALID KEY WRITE[3] [FROM] INVALID KEY REWRITE[2] [FROM] [INVALID KEY]			SAME [RECORD] AREA TRACK-AREA RERUN ON RECORDS			

[1]Create [2]Update [3]Add [4]These APPLY clauses have meaning only for OUTPUT files; however, the compiler accepts them when the same file is opened for INPUT or I-O.

RELATIVE FILES (mass storage devices only) – Required and Optional Entries

Required Entries							Optional Entries		
ACCESS	KEY	System-name	LABEL RECORDS	OPEN	CLOSE	Access Verbs	Other ENVIRONMENT DIVISION Clauses	RECORDING MODE	USE
[SEQUENTIAL]	[NOMINAL]	DA [-xxxx]-R-name	{STANDARD / data-name}	INPUT	[UNIT] [LOCK]	READ [INTO] AT END	SAME [RECORD] AREA RERUN	F	ERROR AFTER LABELS
				OUTPUT	[UNIT] [LOCK]	WRITE[1] [FROM] INVALID KEY	APPLY RECORD-OVERFLOW		
RANDOM	NOMINAL	DA [-xxxx]-R-name	{STANDARD / data-name}	INPUT	[LOCK]	READ [INTO] INVALID KEY	SAME [RECORD] AREA RERUN ON RECORDS	F	ERROR AFTER LABELS
				I-O	[LOCK]	READ [INTO] INVALID KEY REWRITE[2] [FROM] [INVALID KEY]	APPLY RECORD-OVERFLOW		

[1] Create [2] Update

APPENDIX E: ASCII CONSIDERATIONS

The compiler supports the American National Standard Code for Information Interchange (ASCII). Thus the programmer can create and process tape files recorded in accordance with the following standards:

- American National Standard Code for Information Interchange, X3.4-1968
- American National Standard Magnetic Tape Labels for Information Interchange, X3.27-1969
- American National Standard Recorded Magnetic Tape for Information Interchange (800 CPI, NRZI), X3.22-1967

ASCII encoded tape files, when read into the system, are automatically translated in the buffers into EBCDIC. Internal manipulation of data is performed exactly as if they were EBCDIC encoded files. For an output file, the system translated the EBCDIC characters into ASCII in the buffers before writing the file out on tape. Therefore there are special considerations concerning ASCII encoded files when they are processed in COBOL. The following paragraphs discuss these considerations.

I --ENVIRONMENT DIVISION

Environment Division clauses affected by the specification of ASCII files are the ASSIGN clause and the RERUN clause.

ASSIGN Clause

When ASCII files are to be processed, the system-name in the ASSIGN clause has the following format:

```
UT[-device]-C[-offset]-name
```

device, if specified, must specify a magnetic tape device. If this field is omitted, the magnetic tape device must be specified through control cards at execution time.

C in the organization field specifies that an ASCII encoded sequential file is to be processed, or that an ASCII collated Sort is to be performed.

offset may be specified only for an ASCII file, and then only if a buffer offset in the range 01 through 99 exists. It is a 2-digit field, and may be specified as follows:

```
01 through 99  for an input file
04             for an output file (D-mode records only)
```

name is a 1- to 8-character field specifying the external-name by which the file is known to the system. It is the name that appears in the name field of the DD card for the file.

RERUN Clause

The system-name in a RERUN clause must not specify an ASCII encoded file.

ASCII files containing checkpoint records cannot be processed.

II. DATA DIVISION

In the Data Division there are special considerations for ASCII files, both in the File Section and in Data Description Entries.

FILE SECTION

In the File Section the BLOCK CONTAINS, the LABEL RECORDS clause and RECORDING MODE clause are affected. There are also special considerations regarding the compiler default options for recording mode.

BLOCK CONTAINS Clause

For an ASCII file that contains a buffer offset field, the following considerations apply:

- If the BLOCK CONTAINS clause with the RECORDS option is specified, or if the BLOCK CONTAINS clause is omitted, the compiler compensates for the buffer offset field.
- If the BLOCK CONTAINS clause with the CHARACTERS option is specified, the programmer must include the buffer offset area as part of the physical record.

Note: If the BLOCK CONTAINS 0 CHARACTERS option is used and the block size is determined at object time from the DD card or from the data set label for the file, then the programmer must calculate the offset field as part of the block size.

LABEL RECORDS Clause

All three options of the clause (OMITTED/STANDARD/data-name) are allowed. However, if the programmer specifies the data-name option, he must make sure that data-name refers only to user standard labels. Nonstandard labels are not allowed for ASCII files.

RECORDING MODE Clause

For ASCII files, mode may be specified as F, U, or V. S mode may not be specified.

Compiler Calculation of Recording Mode

When the RECORDING MODE clause is not used to specify the mode of the records in an ASCII file, the COBOL compiler determines the mode by scanning each record description entry. The default option may be:

F if all the records are defined as being the same size.

D if the records are defined as variable in size, or if the RECORD CONTAINS clause specifies variable size records. Internally D mode is the equivalent of V mode for EBCDIC encoded files.

DATA DESCRIPTION ENTRIES

For ASCII files the Data Description Entries affected are the PICTURE clause, the SIGN clause, and the USAGE clause.

PICTURE Clause

For ASCII files all five categories of data are valid.

If a data item is numeric, however, and the item is signed, then the SIGN clause with the SEPARATE CHARACTER option must also be specified.

SIGN Clause

If a data item in an ASCII file is numeric and has a sign, then the SIGN clause with the SEPARATE CHARACTER option must be specified.

USAGE Clause

For data items in ASCII files, only the DISPLAY option of the USAGE clause is valid.

III -- PROCEDURE DIVISION

For ASCII files, there are special considerations in regard to Label Declaratives and relation conditions.

LABEL PROCEDURE Declarative

Since the user may not specify nonstandard labels for an ASCII file, the BEFORE option of the LABEL PROCEDURE declarative is not allowed.

Relation Conditions

If the ASCII strings to be compared contain mixed alphabetic/numeric characters and/or special characters, then the TRANSFORM verb can be used before the comparison is made to ensure a valid comparison.

The following example illustrates a method of making the comparison (Figure 20 shows the necessary COBOL statements).

Suppose that the COBOL programmer specifies that one alphanumeric data item (ASCII-1) from ASCII-FILE is to be compared with another such data item (ASCII-2), and that the results of the comparison determine the path of program execution. Each data item may contain any valid COBOL character.

When ASCII-RECORD is read into the buffer, the system changes each ASCII character into its EBCDIC equivalent. Therefore, before a valid ASCII comparison can be made, the relative position of each character in the ASCII collating sequence must be reestablished.

In the Working-Storage Section, the VALUE of IDENT-EBCDIC is the ascending EBCDIC collating sequence (as shown in Figure 20; similarly, the VALUE of IDENT-ASCII is the ascending ASCII collating sequence. The contents of ASCII-1 are moved to DN-1, and the contents of ASCII-2 are moved to DN-2, and DN-1 and DN-2 are then used in the two TRANSFORM statements. (This avoids the necessity of a second pair of TRANSFORM statements to restore the original contents of ASCII-1 and ASCII-2.)

When the two TRANSFORM statements are executed each EBCDIC character is exchanged for another EBCDIC character that occupies the original ASCII character's position in the ASCII collating sequence. Thus, when the comparison is made, it is valid for the ASCII collating sequence.

(Note that if ASCII-1 and ASCII-2 are restricted to mixed alphabetic and numeric characters, then the VALUE clauses in IDENT-EBCDIC and IDENT-ASCII need only contain alphabetic and numeric characters from the collating sequences. Note too that in the VALUE clause when quotation marks (") are used as delimiters, then the quotation mark itself cannot be one of the literals contained within the delimiter; similarly, if the apostrophe (') is used as the delimiter, then the apostrophe cannot be contained within the delimiter.)

```
          .
          .
          .
DATA DIVISION.
FILE SECTION.
FD  ASCII-FILE ...
01  ASCII-RECORD.
    05  ASCII-1 ...
    05  ASCII-2 ...
          .
          .
          .
WORKING-STORAGE SECTION.
77  IDENT-ASCII  PICTURE X(51)  VALUE
      " $'()*+,-./0123456789;<=>ABCDEFGHIJKLMNOPQRSTUVWXYZ".
77  IDENT-EBCDIC  PICTURE X(51)  VALUE
      " .<(+$*);-/,>'=ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789".
77  DN-1 ...
77  DN-2 ...
          .
          .
          .
PROCEDURE DIVISION.
          .
          .
          .
TEST-ASCII.
    MOVE ASCII-1 TO DN-1.
    MOVE ASCII-2 TO DN-2.
    TRANSFORM DN-1 FROM IDENT-EBCDIC TO IDENT-ASCII.
    TRANSFORM DN-2 FROM IDENT-EBCDIC TO IDENT-ASCII.
    IF DN-1 IS NOT LESS THAN DN-2
       PERFORM PROCESS-1
       ELSE PERFORM PROCESS-2.
          .
          .
          .
PROCESS-1.
          .
          .
          .
PROCESS-2.
          .
          .
          .
```

Figure 20. Using the TRANSFORM Statement with ASCII Comparisons

EBCDIC Collating Sequence			ASCII Collating Sequence		
1.		(space)	1.		(space)
2.	.	(period)	2.	"	(quotation mark)
3.	<	(less than)	3.	$	(currency symbol)
4.	(	(left parenthesis)	4.	[redacted]	[redacted]
5.	+	(plus symbol)			
6.	$	(currency symbol)	5.	(	(left parenthesis)
7.	*	(asterisk)	6.	)	(right parenthesis)
8.	)	(right parenthesis)	7.	*	(asterisk)
9.	;	(semicolon)	8.	+	(plus symbol)
10.	-	(hyphen, minus symbol)	9.	,	(comma)
11.	/	(stroke, virgule, slash)	10.	-	(hyphen, minus symbol)
			11.	.	(period, decimal point)
12.	,	(comma)	12.	/	(stroke, virgule, slash)
13.	>	(greater than)	13-22.		0 through 9
14.	[redacted]	[redacted]	23.	;	(semicolon)
			24.	<	(less than)
15.	=	(equal sign)	25.	=	(equal sign)
16.	"	(quotation mark)	26.	>	(greater than)
17-42.		A through Z	27-52.		A through Z
43-52.		0 through 9			

Figure 21. EBCDIC and ASCII Collating Sequences for COBOL Characters -- in Ascending Order

IV -- SORT FEATURE

For ASCII collated sorts, there are special considerations in the Environment Division and in the Data Division.

Environment Division

For ASCII-collated sorts, there are special considerations for the ASSIGN clause [redacted]

ASSIGN Clause

The ASSIGN clause in the SELECT sentence for a sort-file specifies the use of the ASCII collating sequence in the sorting operation through the system-name. System-name must take the following form:

```
class[-device]-C-name
```

C must be encoded in the organization field to specify that the file is to be sorted on the ASCII collating sequence.

The class, device, and name fields have the same meanings they have for sort-file system-names for EBCDIC files, and can be used to describe the sort work files. However, except for the organization field, the compiler treats the system-name as comments.

Note: For an ASCII collated sort, the buffer offset field of system-name is not permitted.

RERUN Clause

Checkpoint records for ASCII collated sorts can be taken. However, the system-name specified in the RERUN clause must not specify an ASCII encoded file.

DATA DIVISION

For ASCII-collated sorts, there are special considerations for the SIGN clause and for the USAGE clause.

SIGN Clause

If an S is specified in the PICTURE of a numeric item to be used as a sort key in an ASCII-collated sort, the SEPARATE option of the SIGN clause must be specified.

USAGE Clause

If an ASCII-collated sort is requested, the sort keys must, explicitly or implicitly, be DISPLAY items.

APPENDIX F: SYMBOLIC DEBUGGING FEATURES (VERSION 4)

A programmer using the Full American National Standard COBOL Compiler, Version 4, under the IBM Operating System, has several methods available to him for testing and debugging his programs. Use of the symbolic debugging features is the easiest and most efficient method for testing and debugging and is described in detail in this appendix.

If symbolic debugging is in effect, a symbolic formatted dump of the object program's data area is produced when the program abnormally terminates. This option also enables the programmer to request dynamic dumps of specified data-names at strategic points during program execution. If two or more COBOL programs are link-edited together and one of them terminates abnormally, a formatted dump is produced for all programs in the current calling sequence compiled with the symbolic debug option, up to and including the main program.

Note: The terminating program need not have been compiled with the symbolic debugging option.

The abnormal termination dump consists of the following parts:

1. An abnormal termination message, including the number of the statement and of the verb being executed at the time of an abnormal termination.

2. Selected areas in the Task Global Table.

3. A formatted dump of the Data Division including:

   (a) For an SD -- the card number, the sort-file-name, the type, and the sort record.

   (b) For an FD -- the card number, the file-name, the type, the DDname, the DECB and/or DCB status, the contents of the DECB and/or DCB in hexadecimal, and the fields of the record.

   (c) For an RD -- the card number, the report-name, the type, the report line, and the contents of PAGE-COUNTER and LINE-COUNTER if present.

   (d) For a CD -- the CD itself in its implicit format, as well as the area containing the message data currently being buffered.

   (e) For an index name -- the name, the type, and the contents in decimal.

Object-Time Control Cards

The operation of the symbolic debugging option is determined by two types of control cards placed in the input stream:

Program-control card -- required if abnormal termination and/or dynamic dumps are requested.

Line-control card -- required only if dynamic dumps are requested.

Program-Control Cards: A program-control card must be present at execution time for any program requesting symbolic debugging. A program-control card must contain the following information:

The 1-8 character program-name of the COBOL program compiled using symbolic debugging.

The DDname assigned to the file produced at compile time on SYSUT5.

Additional optional parameters can also be specified:

An entry used to provide a trace of a program-name when several programs are link edited together. Each time the specified program is entered, its program name is displayed.

Two formats of the Data Division area in the abnormal termination dump are allowed:

1. Level-01 items are provided in hexadecimal. Items subordinate to level-01 items are printed in EBCDIC if possible. Level-77 items are printed both in hexadecimal and EBCDIC.

2. Level-77 items and items subordinate to level-01 items are provided in EBCDIC. If these items contain unprintable characters, hexadecimal notation is provided. This is the default option.

Line-Control Cards: A line-control card must contain the following information:

The card number associated with the point in the Procedure Division at which the dynamic dump is to be taken. The number specified is either the compiler-generated card number, or, if NUM is in effect, the user's number in source card columns 1 through 6.

Additional optional parameters can also be specified:

The position of the verb in the specified line number at which the dynamic dump is to be taken. When the verb position is not specified, the first verb in the line is assumed. Any verb position not exceeding 15 may be specified.

An equivalent to the COBOL statement "ON n AND EVERY m UNTIL k ...". This option limits the requested dynamic dumps to specified times. For example "ON n" results in one dump, produced the nth time the line number is reached during execution. "ON n AND EVERY m" results in a dump the first time at the nth execution of the specified line number, and thereafter at every mth execution until end-of-job.

Two formats of the Data Division areas displayed in the dynamic dump are allowed:

1. Level-01 items are provided in hexadecimal. Items subordinate to level-01 items are provided in EBCDIC, if possible. Level-77 items are provided both in hexadecimal and EBCDIC.

2. Level-77 items and items subordinate to level-01 items are provided in EBCDIC. If these items contain unprintable characters, hexadecimal notation is provided. Note that if a group item is specified, neither the group nor the elementary items in the group are provided in hexadecimal. This is the default option.

Selected areas of the Data Division to be dumped. A single data-name or a range of consecutive data-names can be specified. (If the programmer wishes to see a subscripted item, he specifies the name of the item without the subscript; this results in a dump of every occurrence of the subscripted item.)

A dump of everything that would be dumped in the event of an abnormal termination can also be specified. This allows the programmer to receive a formatted dump at normal end-of-job. To do this, the programmer must specify the generated statement number of the STOP RUN, GOBACK, or EXIT PROGRAM statement.

SAMPLE PROGRAM -- TESTRUN

Figure 22 is an illustration of a program that utilizes the symbolic debugging features. In the following description of the program and its output, letters identifying the text correspond to letters in the program listing.

(A) Because the SYMDMP option is requested in the PARM parameter of the EXEC card, the logical unit SYSUT5 must be assigned at compile time.

(B) The PARM parameter specifications on the EXEC indicates that an alphabetically ordered cross-reference dictionary, a flow trace of 10 procedures, and the symbolic debug option are being requested.

(C) An alphabetically ordered cross-reference dictionary of data-names and procedure-names is produced by the compiler as a result of the SXREF specification in the PARM parameter of the EXEC card.

(D) The file assigned at compile time to SYSUT5 to store SYMDMP information is assigned to SYSUT9 at execution time.

(E) The SYMDMP control cards placed in the input stream at execution time are printed along with any diagnostics.

(1) The first card is the program-control card where:

(a) TESTRUN is the PROGRAM-ID.

(b) DD1 is the DDname on the file SYSUT5.

(2) The second card is a line-control card which requests a (HEX) formatted dynamic dump of KOUNT, NAME-FIELD, NO-OF-DEPENDENTS, and RECORD-NO prior to the first and every fourth execution of generated card number 70.

(3) The third card is also a line-control card which requests a (HEX) formatted dynamic dump of WORK-RECORD and B prior to the execution of generated card number 81.

(F) The type code combinations used to identify data-names in abnormal termination and dynamic dumps are defined. Individual codes are illustrated in Table 28.

Ⓖ The dynamic dumps requested by the first line-control card.

Ⓗ The dynamic dumps requested by the second line-control card.

Ⓘ Program interrupt information is provided by the system when a program terminates abnormally.

Ⓙ The statement number information indicates the number of the verb and of the statement being executed at the time of the abnormal termination. The name of the program containing the statement is also provided.

Ⓚ A flow trace of the last 10 procedures executed is provided because FLOW=10 was specified in the PARM parameter on the EXEC card.

Ⓛ Selected areas of the Task Global Table are provided as part of the abnormal termination dump.

Ⓜ For each file-name, the generated card number, the file type, the file status, the file organization, the DCB status, and the fields of the DCB and DECB, if applicable, are provided.

Ⓝ The fields of records associated with each FD are provided in the format requested on the program-control card.

Ⓟ The contents of the fields of the Working-Storage Section are provided in the format requested on the program-control card.

Ⓠ The value associated with each of the possible subscripts is provided for data items described with an OCCURS clause.

Ⓡ Asterisks appearing within the EBCDIC representation of the value of a given field indicate that the type and the actual content of the field conflict.

Note: When using the SYMDMP option, level numbers appear "normalized" in the symbolic dump produced. For example, a group of data items described as:

```
01  RECORDA.
    05  FIELD-A.
        10  FIELD-A1 PIC X.
        10  FIELD-A2 PIC X.
```

will appear as follows in SYMDMP output:

```
01 RECORDA...
02 FIELD-A...
03 FIELD-A1...
03 FIELD-A2...
```

Debugging TESTRUN

1. Reference to the statement number information Ⓙ provided by the SYMDMP option shows that the abend occurred during the execution of the first verb on card 81.

2. Generated card number 81 contains the statement COMPUTE B = B + 1.

3. Through verification of the contents of B at the time of the abnormal termination Ⓡ, it can be seen that the usage of B (numeric packed) conflicts with the value contained in the data area reserved for B (numeric display).

4. The abnormal termination occurred while trying to perform an addition on a display item.

More complex errors may require the use of dynamic dumps to isolate the problem area. Line-control cards are included in TESTRUN merely to illustrate how they are used and the output they produce.

• Table 28. Individual Type Codes Used in SYMDMP Output

Code	Meaning
A	Alphabetic
B	Binary
D	Display
E	Edited
*	Subscripted Item
F	Floating Point
N	Numeric
P	Packed Decimal
S	Signed
OL	Overpunch Sign Leading
OT	Overpunch Sign Trailing
SL	Separate Sign Leading
ST	Separate Sign Trailing

```
IEF298I  DEBUG    SYSOUT=U.
//DEBUG  JOB   7074722674,'D. DAVIDSON',MSGLEVEL=1,MSGCLASS=G
//JOBLIB DD DSN=DUMMY0S,UNIT=2314,VOL=SER=DC156,DISP=SHR
// DD DSN=PRODVER4,DISP=SHR
(B)-->// EXEC UCOB4CLG,PARM.COB='DMAP,PMAP,SXREF,FLOW=10,SYMDMP,QUOTE,NORES'
XXCOB EXEC PGM=IKFCBL00,REGION=80K,PARM=(LOAD)                          00000010
//COB.SYSPRINT  DD  SYSOUT=G,OUTLIM=1000                                    SMF
X/SYSPRINT DD SYSOUT=U,OUTLIM=1000                                      00000SMF
XXSYSUDUMP DD SYSOUT=U,OUTLIM=1000                                      00000SMF
XXSYSUT1 DD  SPACE=(CYL,(10,2)),UNIT=2314                               00000040
XXSYSUT2 DD  SPACE=(CYL,(10,2)),UNIT=(2314,SEP=SYSUT1)                  00000050
XXSYSUT3 DD  SPACE=(CYL,(10,2)),UNIT=(2314,SEP=(SYSUT1,SYSUT2))         00000060
XXSYSUT4 DD  SPACE=(CYL,(10,2)),UNIT=(2314,SEP=(SYSUT1,SYSUT2,SYSUT3)) 00000070
(A)-->//COB.SYSUT5 DD    DSNAME=&&UT5,UNIT=SYSDA,SPACE=(TRK,(100,10)),
//  DISP=(NEW,PASS)
X/SYSUT5  DD  SPACE=(CYL,(10,2)),UNIT=2314,DSN=&SYMDBG,DISP=(NEW,PASS)  00000080
XXSYSLIN DD DSN=&LOADSET,DISP=(MOD,PASS),UNIT=2314,SPACE=(CYL,(10,2))   00000090
//COB.SYSIN DD *
```

•Figure 22. Symbolic Debugging Option: TESTRUN (Part 1 of 11)

```
IEC130I SYSLIB   DD STATEMENT MISSING
IEF373I STEP /COB     / START 72144.0024
IEF374I STEP /COB     / STOP  72144.0029 CPU    0MIN 04.09SEC MAIN  78K LCS   0K
STEP COB      ENDED. COMP CODE 0004 CORE REQUSTED= 0080K. CORE USED= 0078K.          MU=      2.02
XXLKED  EXEC PGM=IEWL,PARM=(XREF,LIST,LET),COND=(5,LT,COB),              00000100
XX            REGION=96K                                                 00000110
XXSYSLIN DD DSN=&LOADSET,DISP=(OLD,DELETE)                               00000120
XX    DD DDNAME=SYSIN                                                    00000130
XXSYSLMOD DD DSN=&GODATA(RUN),DISP=(NEW,PASS),                           00000140
XX      UNIT=2314,SPACE=(1024,(50,20,1))                                 00000150
//LKED.SYSLIB  DD  DSN=NEWSYMJB,UNIT=2314,VOL=SER=DC157,DISP=SHR
X/SYSLIB DD DSN=SYS1.DYNAMLIB,DISP=SHR                                   00000160
//             DD  DSNAME=SYS1.DYNAMLIB,DISP=SHR
X/  DD DSN=SYS1.TELCMLIB,DISP=SHR                                        00000170
XXSYSUT1 DD UNIT=(2314,SEP=(SYSLIN,SYSLMOD)),SPACE=(1024,(50,20))        00000180
//LKED.SYSPRINT  DD  SYSOUT=G,OUTLIM=1000                                     SMF
X/SYSPRINT DD SYSOUT=U,OUTLIM=1000                                       00000SMF
XXSYSUDUMP DD SYSOUT=U,OUTLIM=1000                                       00000SMF


IEF373I STEP /LKED    / START 72144.0029
IEF374I STEP /LKED    / STOP  72144.0030 CPU    0MIN 00.67SEC MAIN  96K LCS   0K
STEP LKED     ENDED. COMP CODE 0000 CORE REQUSTED= 0096K. CORE USED= 0096K.          MU=      .60
XXGO  EXEC PGM=*.LKED.SYSLMOD,COND=((5,LT,COB),(5,LT,LKED))              00000210
//GO.SYSUDUMP  DD  SYSOUT=G,OUTLIM=1000                                       SMF
X/SYSUDUMP DD SYSOUT=U,OUTLIM=1000                                       00000SMF
XXSYSDBOUT  DD  SYSOUT=U,OUTLIM=1000                                     00000SMF
(D)→//GO.DD1 DD DSN=&&UT5,UNIT=SYSDA,DISP=(OLD,DELETE)
X/DD1  DD  DSN=&SYMDBG,DISP=(OLD,DELETE)                                 00000240
//GO.SAMPLE DD UNIT=2400,LABEL=(,NL),DISP=(NEW,DELETE),VOL=SER=TESTER
//GO.SYSOUT  DD  SYSOUT=G,OUTLIM=1000                                         SMF
//GO.SYSDBOUT  DD  SYSOUT=G,OUTLIM=1000                                       SMF
//GO.STEPLIB  DD  DSN=NEWSYMJB,UNIT=2314,VOL=SER=DC157,DISP=SHR
//            DD  DSNAME=SYS1.DYNAMLIB,DISP=SHR
//GO.SYSDBG    DD  *
//
```

•Figure 22. Symbolic Debugging Option: TESTRUN (Part 2 of 11)

```
IEC130I SYSDTERM DD STATEMENT MISSING
A 0001 NYC 0
B 0002 NYC 1
C 0003 NYC 2
D 0004 NYC 3
E 0005 NYC 4
F 0006 NYC 0
G 0007 NYC 1
H 0008 NYC 2
I 0009 NYC 3
IEF460I WTP MESSAGE LIMIT EXCEEDED
COMPLETION CODE - SYSTEM=0C7  USER=0000
IEF242I ALLOC. FOR DEBUG    GO                AT ABEND
IEF237I 136   ALLOCATED TO JOBLIB
IEF237I 355   ALLOCATED TO
IEF237I 240   ALLOCATED TO PGM=*.DD
IEF237I 242   ALLOCATED TO SYSUDUMP
IEF237I 242   ALLOCATED TO SYSDBOUT
IEF237I 241   ALLOCATED TO DD1
IEF237I 282   ALLOCATED TO SAMPLE
IEF237I 242   ALLOCATED TO SYSOUT
IEF237I 242   ALLOCATED TO SYSDBOUT
IEF237I 137   ALLOCATED TO STEPLIB
IEF237I 355   ALLOCATED TO
IEF237I 241   ALLOCATED TO SYSDBG
IEF285I   DUMMY0S                                 PASSED
IEF285I   VOL SER NOS= DC156 .
IEF285I   PRODVER4                                PASSED
IEF285I   VOL SER NOS= DC160 .
IEF285I   SYS72144.T002347.RV000.DEBUG.GODATA     PASSED
IEF285I   VOL SER NOS= 231400.
IEF285I   SYS72144.T002347.SV000.DEBUG.R0000011   SYSOUT
IEF285I   VOL SER NOS= 231402.
IEF285I   SYS72144.T002347.SV000.DEBUG.R0000012   SYSOUT
IEF285I   VOL SER NOS= 231402.
IEF285I   SYS72144.T002347.RV000.DEBUG.UT5        DELETED
IEF285I   VOL SER NOS= 231401.
IEF285I   SYS72144.T002347.RV000.DEBUG.R0000013   DELETED
IEF285I   VOL SER NOS= TESTER.
IEF285I   SYS72144.T002347.SV000.DEBUG.R0000014   DELETED
IEF285I   VOL SER NOS= 231402.
IEF285I   SYS72144.T002347.SV000.DEBUG.R0000015   DELETED
IEF285I   VOL SER NOS= 231402.
IEF285I   NEWSYMJB                                KEPT
IEF285I   VOL SER NOS= DC157 .
IEF285I   SYS1.DYNAMLIB                           KEPT
IEF285I   VOL SER NOS= DC160 .
IEF285I   SYS72144.T002347.RV000.DEBUG.S0000016   SYSIN
IEF285I   VOL SER NOS= 231401.
IEF285I   SYS72144.T002347.RV000.DEBUG.S0000016   DELETED
IEF285I   VOL SER NOS= 231401.
IEF373I STEP /GO      / START 72144.0030
IEF374I STEP /GO      / STOP  72144.0033 CPU   0MIN 03.20SEC MAIN  52K LCS   0K
STEP GO      ENDED. COMP CODE 00C7 CORE REQUSTED= 0052K. CORE USED= 0052K.                    MU=      1.16
IEF285I   DUMMY0S                                 KEPT
IEF285I   VOL SER NOS= DC156 .
IEF285I   PRODVER4                                KEPT
IEF285I   VOL SER NOS= DC160 .
IEF285I   SYS72144.T002347.RV000.DEBUG.GODATA     DELETED
IEF285I   VOL SER NOS= 231400.
IEF375I  JOB /DEBUG   / START 72144.0024
IEF376I  JOB /DEBUG   / STOP  72144.0033 CPU   0MIN 07.96SEC
JOB DEBUG    ENDED. CODE= 00C7 JOB READ IN AT 00.40    ON 72144 JOB STRTED AT 00.41    ON 72144 JOB ENDED AT 00.56    ON 72144
 *** THIS JOB WAS RUN ON MODEL 65NG
```

Figure 22. Symbolic Debugging Option: TESTRUN (Part 3 of 11)

```
PP 5734-CB2 V4 LVL76                IBM OS AMERICAN NATIONAL STANDARD COBOL                DATE JAN  6,1972

   1

IKF0011I-W SYSLIB NOT USABLE. COMPILATION CONTINUING.
$LEDMP   7    X
$LEDMP   H    X


   2

00001   100010 IDENTIFICATION DIVISION.
00002   100020 PROGRAM-ID.  TESTRUN.
00003   100030   AUTHOR. PROGRAMMER NAME.
00004   100040   INSTALLATION. NEW YORK PROGRAMMING CENTER.
00005   100050   DATE-WRITTEN. JULY 12, 1968.
00006   100060 DATE-COMPILED. JAN  6,1972
00007   100070   REMARKS. THIS PROGRAM HAS BEEN WRITTEN AS A SAMPLE PROGRAM FOR
00008   100080     COBOL USERS. IT CREATES AN OUTPUT FILE AND READS IT BACK AS
00009   100090     INPUT.
00010
00011   100100 ENVIRONMENT DIVISION.
00012   100110 CONFIGURATION SECTION.
00013   100120   SOURCE-COMPUTER. IBM-360-H50.
00014   100130   OBJECT-COMPUTER. IBM-360-H50.
00015   100140 INPUT-OUTPUT SECTION.
00016   100150 FILE-CONTROL.
00017   100160     SELECT FILE-1 ASSIGN TO UT-2400-S-SAMPLE.
00018   100170     SELECT FILE-2 ASSIGN TO UT-2400-S-SAMPLE.
00019
00020   100180 DATA DIVISION.
00021   100190 FILE SECTION.
00022   100200 FD  FILE-1
00023   100210     LABEL RECORDS ARE OMITTED
00024   100220     BLOCK CONTAINS 100 CHARACTERS
00025   100225     RECORD CONTAINS 20 CHARACTERS
00026   100230     RECORDING MODE IS F
00027   100240     DATA RECORD IS RECORD-1.
00028   100250   01 RECORD-1.
00029   100260     02 FIELD-A PICTURE IS X(20).
00030   100270 FD  FILE-2
00031   100280     LABEL RECORDS ARE OMITTED
00032   100290     BLOCK CONTAINS 5 RECORDS
00033   100300     RECORD CONTAINS 20 CHARACTERS
00034   100310     RECORDING MODE IS F
00035   100320     DATA RECORD IS RECORD-2.
00036   100330   01 RECORD-2.
00037   100340     02 FIELD-A PICTURE IS X(20).
00038
00039   100350 WORKING-STORAGE SECTION.
00040   100360 77 KOUNT PICTURE S99 COMP SYNC.
00041   100370 77 NUMBER  PICTURE S99 COMP SYNC,
00042   100375   01 FILLER.
00043   100380     02 ALPHABET PICTURE X(26) VALUE "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
00044   100395     02 ALPHA REDEFINES ALPHABET PICTURE X OCCURS 26 TIMES.
00045   100405     02 DEPENDENTS PICTURE X(26)  VALUE "01234012340123401234012340
00046   100410-    "0".
00047   100420     02 DEPEND REDEFINES DEPENDENTS PICTURE X OCCURS 26 TIMES.
00048   100440   01 WORK-RECORD.
00049   100450     02 NAME-FIELD  PICTURE X.
00050   100460     02 FILLER PICTURE X VALUE IS SPACE.
00051   100470     02 RECORD-NO PICTURE 9999.
00052   100480     02 FILLER PICTURE X VALUE IS SPACE.
00053   100490     02 LOCATION PICTURE AAA VALUE IS "NYC".
00054   100500     02 FILLER PICTURE X VALUE IS SPACE.
00055   100510     02 NO-OF-DEPENDENTS PICTURE XX.
00056   100520     02 FILLER PICTURE X(7) VALUE IS SPACES.
00057   100521     01 RECORDA.
```

Figure 22. Symbolic Debugging Option: TESTRUN (Part 4 of 11)

```
00058   100522         02 .A PICTURE S9(4) VALUE 1234.
00059   100523         02  B REDEFINES A PICTURE S9(7) COMPUTATIONAL-3.
00060   100530 PROCEDURE DIVISION.
00061   100540 BEGIN. READY TRACE.
00062   100550      NOTE THAT THE FOLLOWING OPENS THE OUTPUT FILE TO BE CREATED
00063   100560      AND INITIALIZES COUNTERS.
00064   100570 STEP-1. OPEN OUTPUT FILE-1. MOVE ZERO TO KOUNT NUMBER.
00065   100580      NOTE THAT THE FOLLOWING CREATES INTERNALLY THE RECORDS TO BE
00066   100590      CONTAINED IN THE FILE, WRITES THEM ON TAPE, AND DISPLAYS
00067   100600      THEM ON THE CONSOLE.
00068   100610 STEP-2. ADD 1 TO KOUNT, ADD 1 TO NUMBER, MOVE ALPHA (KOUNT) TO
00069   100620      NAME-FIELD.
00070   100630      MOVE DEPEND (KOUNT) TO NO-OF-DEPENDENTS.
00071   100640      MOVE NUMBER TO RECORD-NO.
00072   100650 STEP-3. DISPLAY WORK-RECORD UPON CONSOLE. WRITE RECORD-1 FROM
00073   100660      WORK-RECORD.
00074   100670 STEP-4. PERFORM STEP-2 THRU STEP-3 UNTIL KOUNT IS EQUAL TO 26.
00075   100680      NOTE THAT THE FOLLOWING CLOSES OUTPUT AND REOPENS IT AS
00076   100690      INPUT.
00077   100700 STEP-5. CLOSE FILE-1. OPEN INPUT FILE-2.
00078   100710      NOTE THAT THE FOLLOWING READS BACK THE FILE AND SINGLES OUT
00079   100720      EMPLOYEES WITH NO DEPENDENTS.
00080   100730 STEP-6. READ FILE-2 RECORD INTO WORK-RECORD AT END GO TO STEP-8.
00081   100731         COMPUTE B = B + 1.
00082   100740 STEP-7. IF NO-OF-DEPENDENTS IS EQUAL TO "0" MOVE "Z" TO
00083   100750      NO-OF-DEPENDENTS. EXHIBIT NAMED WORK-RECORD. GO TO
00084   100760      STEP-6.
00085   100770 STEP-8. CLOSE FILE-2.
00086   100780      STOP RUN.
```

18

(C)——→CROSS-REFERENCE DICTIONARY

DATA NAMES	DEFN	REFERENCE			
A	000058				
ALPHA	000044	000068			
ALPHABET	000043				
B	000059	000081			
DEPEND	000047	000070			
DEPENDENTS	000045				
FIELD-A	000029				
FIELD-A	000037				
FILE-1	000017	000064	000072	000077	
FILE-2	000018	000077	000080	000085	
KOUNT	000040	000064	000068	000070	000074
LOCATION	000053				
NAME-FIELD	000049	000068			
NO-OF-DEPENDENTS	000055	000070	000082		
NUMBER	000041	000064	000068	000071	
RECORD-NO	000051	000071			
RECORD-1	000028	000072			
RECORD-2	000036	000080			
RECORDA	000057				
WORK-RECORD	000048	000072	000080	000083	

Figure 22. Symbolic Debugging Option: TESTRUN (Part 5 of 11)

```
 19

PROCEDURE NAMES                  DEFN     REFERENCE

BEGIN                            000061
STEP-1                           000064
STEP-2                           000068   000074
STEP-3                           000072   000074
STEP-4                           000074
STEP-5                           000077
STEP-6                           000080   000083
STEP-7                           000082
STEP-8                           000085   000080


  20

 CARD   ERROR MESSAGE
 58     IKF2190I-W     PICTURE CLAUSE IS SIGNED, VALUE CLAUSE UNSIGNED. ASSUMED POSITIVE.

PHASE      FILE1     FILE2     FILE3     FILE4     FILE5
1       00000000 00000000 0000034C 00000000 00000000
2       C0000000 00000000 00000000 00000000 00000000
3       00000000 000002D6 00000000 00000000 00000000
4       00000000 00000000 00000000 0000040A 00000000
5       00000000 00000000 000002C1 00000000 000C0000
6       C0000000 00000000 00000000 000003FF 00000000
7       00000000 00000000 00000000 00000000 00000400
8       00000000 00000000 00000351 00000074 00000000
9       000005DD 00000000 00000000 00000000 00000000
A       00000000 00000000 00000000 00000000 00000000
B       00000000 0000083C 00000035 00000000 00000000
C       00000A34 00000000 00000246 00000001 00000000
D       00000000 00000000 C0000000 00000000 000C0000
E       00000000 00000000 00000000 00000000 00000000
F       00000000 00000783 00000000 000000E2 00000000
G       000001EB 00000000 00000114 00000000 00000000
H       00000000 00000000 0000000C 00000000 00000600
```

• Figure 22. Symbolic Debugging Option: TESTRUN (Part 6 of 11)

```
                                                    (E)→SYMDMP CONTROL CARDS

 (1)→TESTRUN,EE1
 (2)→70,ON 1,4,(HEX),KOUNT,NAME-FIELD,NO-OF-DEPENDENTS,RECORD-NO
 (3)→81,(HEX),NCBR-RECORD,E
          TESTRUN   UNIDENTIFIED ELEMENTS ON CONTROL CARDS
*ERROR* CARD/VERB
IKF1601I   70       IDENTIFIER NOT FOUND

                                        001 ERRORS FOUND IN CONTROL CARDS
```

(F)→TYPE CODES USED IN SYMDMP OUTPUT

CODE	MEANING
A	= ALPHABETIC
AN	= ALPHANUMERIC
ANE	= ALPHANUMERIC EDITED
D	= DISPLAY (STERLING NONREPORT)
DE	= DISPLAY EDITED (STERLING REPORT)
F	= FLOATING POINT (COMP-1/COMP-2)
FD	= FLOATING POINT DISPLAY (EXTERNAL FLOATING POINT)
NB	= NUMERIC BINARY UNSIGNED (COMP)
NB-S	= NUMERIC BINARY SIGNED
ND	= NUMERIC DISPLAY UNSIGNED (EXTERNAL DECIMAL)
ND-OL	= NUMERIC DISPLAY OVERPUNCH SIGN LEADING
ND-OT	= NUMERIC DISPLAY OVERPUNCH SIGN TRAILING
ND-SL	= NUMERIC DISPLAY SEPARATE SIGN LEADING
ND-ST	= NUMERIC DISPLAY SEPARATE SIGN TRAILING
NE	= NUMERIC EDITED
NP	= NUMERIC PACKED DECIMAL UNSIGNED (COMP-3)
NP-S	= NUMERIC PACKED DECIMAL SIGNED
*	= SUBSCRIPTED

```
TESTRUN  AT CARD 000070
 LOC     CARD    LV NAME                          TYPE    VALUE
(G)→0D0778  000040  77 KOUNT                        NB-S    +01
                                              (HEX)         0001
0D07B8  000049  02 NAME-FIELD                     AN      A
0DC7BA  0C0051  02 RECORD-NO                      NC      ****
                                              (HEX)         4750C0FE


TESTRUN  AT CARD 000070
 LOC     CARD    LV NAME                          TYPE    VALUE
0D0778  000040  77 KOUNT                         NB-S    +05
                                              (HEX)         0005
0D07B8  000049  02 NAME-FIELD                     AN      E
0DC7BA  000051  02 RECORD-NO                      NC      0004


TESTRUN  AT CARD 000070
 LOC     CARD    LV NAME                          TYPE    VALUE
0DC778  0C0040  77 KOUNT                         NB-S    +09
                                              (HEX)         0009
0D07B8  000049  02 NAME-FIELD                     AN      I
```

Figure 22. Symbolic Debugging Option: TESTRUN (Part 7 of 11)

```
0DC78A  000051  02 RECORD-NO                          NC      0004

TESTRUN  AT CARD 000070
 LOC     CARD   LV NAME                               TYPE    VALUE
0DC778  000040  77 KOUNT                             NB-S    +13
                                                (HEX)       000D
0DC788  000049  02 NAME-FIELD                        AN      M
0D07EA  000051  02 RECORD-NO                        ND      0012

TESTRUN  AT CARD 000070
 LOC     CARD   LV NAME                               TYPE    VALUE
0D0778  000040  77 KOUNT                             NB-S    +17
                                                (HEX)       0011
0D0788  000049  02 NAME-FIELD                        AN      Q
0DC78A  000051  02 RECORD-NO                        NC      0016

TESTRUN  AT CARD 000070
 LOC     CARD   LV NAME                               TYPE    VALUE
0DC778  000040  77 KOUNT                             NB-S    +21
                                                (HEX)       0015
0D07B8  000049  02 NAME-FIELD                        AN      U
0D07EA  000051  02 RECORD-NO                        NC      0020

TESTRUN  AT CARD 000070
 LOC     CARD   LV NAME                               TYPE    VALUE
0D0778  000040  77 KOUNT                             NB-S    +25
                                                (HEX)       0019
0D0788  000049  02 NAME-FIELD                        AN      Y
0D07BA  000051  02 RECORD-NO                        ND      0024

TESTRUN  AT CARD 000081
 LOC     CARD   LV NAME                               TYPE    VALUE
        000048  01 WORK-RECORD
0DC7B8                                          (HEX)       C140F0F0 F0F140D5 E4C340F0   40404040 40404040
0D07E8  000049  02 NAME-FIELD                        AN      A
0D07B9  000050  02 FILLER                            AN
0D07EA  000051  02 RECORD-NO                        ND      0001
0DC7BE  000052  02 FILLER                            AN
0D07EF  000053  02 LOCATION                          A       NYC
0DC7C2  000054  02 FILLER                            AN
0E07C3  000055  02 NO-OF-DEPENDENTS                  AN      0
0D07C5  000056  02 FILLER                            AN

0D07E0  000059  02 B                                 NP-S    +1+2+3+
                                                (HEX)       F1F2F3C4
```

•Figure 22. Symbolic Debugging Option: TESTRUN (Part 8 of 11)

```
                                   COBOL ABEND DIAGNOSTIC AIDS

(I)→PROGRAM         TESTRUN

     EXCEPTION CODE = 0C7     LAST PSW BEFORE ABEND = FFD5000DD00D0A06

(J)→LAST CARD NUMBER/VERB NUMBER EXECUTED -- CARD NUMBER 000081/VERB NUMBER 01.

                                                                  FLOW TRACE
(K)→TESTRUN  000068 000072 000068 00C072 000068 00C072 00006B 000072 000077 000080


                                   DATA DIVISION DUMP OF TESTRUN

(L)→TASK GLOBAL TABLE        LCC       VALUE

     SAVE AREA               0D0938    009A9200 000DC768 000DA2F8 700D0EB6   0000E23A 700D0ECE 000DA400 000D08B8
                             0D0958    00026CE4 00C000D0 700D0E0E 000D0778   000DA414 000DA400 000D0FFE 000D06F0
                             0D0978    700D0ECE 000D0B70
     SWITCH                  0D0980    7D00004B
     TALLY                   0D0984    00000000
     SORT-SAVE               0D0988    000C0000
     ENTRY-SAVE              0D098C    000D0BDC
     SORT-CORE-SIZE          0D0990    00000000
     RETURN-CODE             0D0994    05EF
     SORT-RETURN             0D0996    5891
     WORKING CELLS           0D0998    000D2456 000D2E1A FFFFFF2E 000DC7F8   00026B5C 00000000 00108000 E940F0F0
                             0D09B8    F2F640D5 E8C340F0 02004020 40404040   000D06F0 000D06F0 000D0B70 60089202
                             0D09D8    6C081B99 50910000 4140C0EA 47F0C1F8   D2001000 7000D200 D0584000 D700D058
                             0D09F8    7000DC00 18F00700 FA306058 C04F0700   5800D1D8 07FB4000 00000FA0 000DC658
                             0D0A18    20001000 30C00000 000A7788 00000000   00000001 000A7BC0 47F0F00E 000D0E96
                             0D0A38    000D080C 010090EC 400D0E84 000D108A   00000030 000D0E9E 000D080C 00026CE4
                             0D0A58    00C000D0 700D0E0E 000D0778 000DA414   000D06F0 000D0FFE 000D06F0 000D0BDC
                             0D0A78    000D0B70 000D0E9E 000D108A 00000030   8F0DC020 000D080C 00011A78 000D0F96
                             0D0A98    4270B001 9240B002 9200B003 41405000   440090F6 5060B004 06704470 917A50C0
                             0D0AB8    E00841B0 B00C41C7 C0014177 80019102
     SORT-FILE-SIZE          0D0AC8    00000000
     SORT-MODE-SIZE          0D0ACC    00000000
     SGT-VN TBL              0D0AD0    86D29142
     TGT-VN TBL              0D0AD4    50E0D008
     VCON ADDR               0D0AD8    5050D000
     VN TBL LENGTH           0D0ADC    4177
     LABEL RETURN            0D0ADE    00
     CURRENT PRIORITY        0D0ADF    00
     DEBUG SAVE14            0D0AE0    700D0ECE
     COBOL INDICATOR         0D0AE4    ANSC
     A(INIT1)                0D0AE8    000D06F0
     DEBUG TABLE PTR         0D0AEC    00000478
     SUBCOM ADDR             0D0AF0    000D0630
     SORT DDNAME             0D0AF4
     UNUSED                  0D0AFC    585CD000 00000000 1F744100 000C1815   58505004
     DEBUG SAVE11            0D0B10    000F0BDC
     UNUSED                  0D0B14    00011815
     PRBADR CELL             0D0B18    000D0BD8
     GENCE TABLE             0D0B1C    D0081FD2
     UNUSED                  0D0B20    06
     TRANSIENT AREA LENGTH   0D0B21    7E9500
     UNUSED                  0D0B24    50049600 B0024140
     OVERFLOW CELLS          (NONE)
     BL CELLS                0D0B2C    000DA414 000DA400 000D0778
     DECBAIR CELLS           (NONE)
     TEMP STORAGE            0D0B38    00000000 0000026C
     ELL CELLS               0D0B40    000C0000 00C00000
```

Figure 22. Symbolic Debugging Option: TESTRUN (Part 9 of 11)

```
                                    DATA DIVISION DUMP OF TESTRUN

VLC CELLS                (NONE)
SEL CELLS                (NONE)
INDEX CELLS              (NONE)
CTHFB (SEE MEMORY MAP)   0D0B48    000D0799 000D0793 000D0990 000D0D90   800D0888 18141E11 4101100C 00000001
                         0D0F68    0A0C08CA 20C60A0A


                                    DATA DIVISION DUMP OF TESTRUN

     LOC      CARD    LV NAME                          TYPE     VALUE

(M)------>000017  FD FILE-1                            QSAM     FILE: CLOSED   ORGANIZATION: PHYSICAL SEQUENTIAL

     0DC80C                                  DCB     C0000000 00000000 00000000 00000006   00810000 000DA391 00C04000 00000001
     0DC82C                                          46000001 900D07DC E2C1D4D7 D3C54040   02000048 00000001 060D2456 00000064
     0DD84C                                          C0000000 00000000 00000000 00000001   00000014 00000001 00000000 00000000

          000028  01 RECORD-1
(N)->0DA414 000029  02 FIELD-A                           AN       A 0002 NYC 1

(M)------>0C0018  FD FILE-2                            QSAM     FILE: OPEN     ORGANIZATION: PHYSICAL SEQUENTIAL

     0D08E8                                  DCB     00000000 00000000 00000000 00000002   0081C100 020DA390 00004000 00000001
     0D08D8                                          460D0EC8 900D0888 007C4800 00026CE4   120FEE00 000FEC40 060D2456 00090064
     0DC8F8                                          2801282A 000DB030 000DA464 000DA400   00000014 00000001 00000000 000FF898

(N)
          000036  01 RECORD-2
(N)->CDA400 000037  02 FIELD-A                           AN       A 0001 NYC 0
(P)->0D0778 000040  77 KOUNT                             NB-S     +26
     0DC77A 000041  77 NUMBER                            NB-S     +26

          000042  01 FILLER
     0D0780 000043  02 ALPHABET                          AN       ABCDEFGHIJKLMNOPQRSTUVWXYZ
          000044  02 ALPHA                               *AN

                         (SUB1)
(Q)->0D0780                1                                     A
     0D0781                2                                     B
     0D0782                3                                     C
     0D0783                4                                     D
     0DC784                5                                     E
     0D0785                6                                     F
     0DC786                7                                     G
     0D0787                8                                     H
     0DC789                9                                     I
     0D0789               10                                     J
     0DC78A               11                                     K
     0D078B               12                                     L
     0DC78C               13                                     M
     0D078D               14                                     N
     0DC78E               15                                     O
     0D078F               16                                     P
     0DC790               17                                     Q
     0D0791               18                                     R
     0DC792               19                                     S
     0D0793               20                                     T
     0DC794               21                                     U
     0D0795               22                                     V
     0DC796               23                                     W
     0D0797               24                                     X
```

• Figure 22. Symbolic Debugging Option: TESTRUN (Part 10 of 11)

```
                                        DATA DIVISION DUMP OF TESTRUN

 LOC      CARD    LV NAME                          TYPE    VALUE

0DC798                    25                              Y
0DC799                    26                              Z
0DC79A  000045  02 DEPENDENTS                       AN      012340123401234012340123 40
        000047  02 DEPEND                          *AN

                          (Q)-->(SUB1)
0DC79A                     1                              0
0DC79B                     2                              1
0DC79C                     3                              2
0DC79D                     4                              3
0DC79E                     5                              4
0DC79F                     6                              0
0DC7A0                     7                              1
0DC7A1                     8                              2
0DC7A2                     9                              3
0DC7A3                    10                              4
0DC7A4                    11                              0
0DC7A5                    12                              1
0DC7A6                    13                              2
0DC7A7                    14                              3
0DC7A8                    15                              4
0DC7A9                    16                              0
0DC7AA                    17                              1
0DC7AB                    18                              2
0DC7AC                    19                              3
0DC7AD                    20                              4
0DC7AE                    21                              0
0DC7AF                    22                              1
0DC7B0                    23                              2
0DC7B1                    24                              3
0DC7B2                    25                              4
0DC7B3                    26                              0

        000048  01 WORK-RECORD <-(P)
0DC7B8  000049  02 NAME-FIELD                       AN      A
0DC7F9  000050  02 FILLER                          AN
0DC7BA  000051  02 RECORD-NO                       NT      0001
0DC7EE  000052  02 FILLER                          AN
0DC7BF  000053  02 LOCATION                        A       NYC
0DC7C2  000054  02 FILLER                          AN
0DC7C3  000055  02 NO-OF-DEPENDENTS                AN      0
0DC7C5  000056  02 FILLER                          AN

        000057  01 RECORDA
0DC7C0  000058  02 A                               ND-OT   +1234
0DC7D0  000059  02 B                               NP-S    +1+2+3+<-(R)
                                            (HEX)          F1F2F3C4

                                        DATA DIVISION DUMP OF TESTRUN

 LOC      CARD    LV NAME                          TYPE    VALUE

                                        END OF COBOL DIAGNOSTIC AIDS
```

•Figure 22. Symbolic Debugging Option: TESTRUN (Part 11 of 11)

APPENDIX G: 3505/3525 CARD PROCESSING

The IBM 3505 card reader and the 3525 card punch are 80-column devices that offer more flexible processing capabilities than former card devices. The 3505 card reader can be used for sequential reading; it can also be used for Optical Mark Read (OMR) processing. Both the 3505 and the 3525 support Read Column Eliminate (RCE) processing. The 3525 card punch, when equipped with appropriate special features, can be used separately as a card reader, as a card punch, as an interpreting card punch, and as a printer (either 2-line or multiline printing is available); in addition, the read, punch, and print functions (any two or all three) can be combined, so that those functions specified are all performed during one pass of a card through the device.

Note: The interpreting card punch is considered one function. It cannot be combined with the other functions, but is specified through the DD statement for the data set.

The processing functions are all specified through new parameters of the DD statement. For OMR and RCE processing, format descriptor card(s) must also be included as the first card(s) of the data set. (For OMR processing, the format descriptor specifies those columns that are optically marked; for RCE processing, the format descriptor specifes those columns that are to be ignored.) Detailed information on these considerations is given in the publication IBM System/360 Planning Guide for IBM 3505 Card Reader and IBM 3525 Card Punch On System/370, Order No. GC21-5027.

The following paragraphs describe the special COBOL programming considerations when these devices are used.

3505 OMR PROCESSING

If the user wishes to inspect the substitution character (hexadecimal "3F") placed in column 80 by the system for a defective optically marked card, he must specify a record description of 80 characters. (Note that the "3F" is placed in both card column 80 and the defective (unreadable) card column.

3505/3525 RCE PROCESSING

When RCE processing is specified for input, the user must not refer to the ignored columns (as specified by the format descriptor), or results are unpredictable.

When RCE processing is specified for output, any data in the COBOL record that corresponds to the ignored columns (as specified by the format descriptor) is not punched or printed.

3525 COMBINED FUNCTION PROCESSING

COBOL handles each of the separate functions to be combined as a separate logical file. Each such logical file has its own file structure and procedural processing requirements. However, because such combined function files refer to one physical unit, the user must observe certain restrictions during processing. The following sections explain the programming requirements for combined function processing in OS American National Standard COBOL.

The COBOL language does not define the files as being combined function files; instead, the combined functions are specified through new parameters for the files' DD statements. (In this way, the user can, if he so desires, process the same COBOL files as completely separate read, punch, and print files.) The necessary parameters are given in the publication:

IBM System/360 Planning Guide for IBM 3505 Card Reader and IBM 3525 Card Punch on System/370, Order No. GC21-5027

I -- ENVIRONMENT DIVISION CONSIDERATIONS

For each function, there must be a separate SELECT sentence written in the Environment Division. Each read function file and each punch function file must specify RESERVE NO ALTERNATE AREA(S).

SPECIAL-NAMES Paragraph

If stacker selection of punched output, or line control of printed output is desired, mnemonic-names for the purpose can be specified in the SPECIAL-NAMES Paragraph. The mnemonic-names may be equated with the following function-names:

Function-name	Meaning
S01	Stacker 1
S02	Stacker 2
C01	Line 1
C02	Line 3
C03	Line 5
...	...
C12	Line 23

Note: When stacker selection is specified, RESERVE NO ALTERNATE AREAS must also be specified.

II -- DATA DIVISION CONSIDERATIONS

For each logical file defined in the Environment Division for the combined function structure, there must be a corresponding FD entry and 01 record description entry in the File Section of the Data Division.

III -- PROCEDURE DIVISION CONSIDERATIONS

Input/output operations must proceed in a specified order in the Procedure Division. In the 3525 device, the card passes first through the reading station, next through the punching station, and last through the printing station. Therefore, the following combined functions may be specified, but only in the order shown:

Functions to be Combined	Order of Operations	Associated COBOL Statement
read/punch/print	read	READ ... AT END
	punch	WRITE ... ADVANCING/POSITIONING
	[print]	WRITE ... AFTER ADVANCING/POSITIONING
read/punch	read	READ ... AT END
	punch	WRITE ... ADVANCING/POSITIONING
read/print	read	READ ... AT END
	[print]	WRITE ... AFTER ADVANCING/POSITIONING
punch/print	punch	WRITE ... ADVANCING/POSITIONING
	[print]	WRITE ... AFTER ADVANCING/POSITIONING

All required operations on one card must be completed before the next card is obtained, or there is an abnormal termination of the job.

The following Procedure Division considerations in the COBOL source program apply:

OPEN Statement

For any specified function, an OPEN statement must be issued before the input/output operation for that function is attempted. The following additional considerations apply:

- For the read function, the file must be opened INPUT.
- For the punch function and print function the file must be opened OUTPUT.

WRITE Statement -- Punch Function Files

If the user wishes to punch additional data into some of the cards and not into others, he must issue a dummy WRITE statement for the null cards, first filling the output area with SPACES.

If stacker selection for the punch function file is desired, the user can specify S01 (for stacker one) and S02 (for stacker two) as function-names in the SPECIAL-NAMES Paragraph. He can then issue WRITE ADVANCING statements using the associated mnemonic-names.

Alternatively, if he specifies WRITE AFTER POSITIONING, he must use the identifier-2 option. The values placed in identifier-2 before the statement is issued must be V (for stacker 1) or W (for stacker 2). Stacker selection may be specified only for the punch function file.

WRITE Statement -- Print Function Files

After the punch function operations (if specified) are completed, the user can issue WRITE statement(s) for the print function file.

If the user wishes to print additional data on some of the data cards and not on others, he may omit the WRITE statement for the null cards.

Any attempt to write beyond the limits of the card results in abnormal termination of the job; thus, the END-OF-PAGE may not be specified.

Depending on the capabilities of the specific model in use, the print file may be either a 2-line print file or a multi-line print file. Up to 64 characters may be printed on each line.

For a 2-line print file, the lines are printed on line 1 (top edge of card) and line 3 (between rows 11 and 12). Line control may not be specified. Automatic spacing is provided.

For a multi-line print file up to 25 lines of characters may be printed. Line control may be specified. If line control is not specified, automatic spacing is provided.

Line control is specified by issuing WRITE AFTER ADVANCING statements, or WRITE AFTER POSITIONING statements for the print function file. If line control is used for one such statement, it must be used for all other WRITE statements issued to the file. The maximum number of printable characters, including any SPACE characters, is 64. The first character of the record defined may be reserved by the programmer for the line control character; therefore, the record may be defined as containing up to 65 characters. Such WRITE statements must not specify space suppression.

Identifier and integer have the same meanings they have for other WRITE AFTER ADVANCING or WRITE AFTER POSITIONING statements. However, such WRITE statements must not increase the line position on the card beyond the card limits, or abnormal termination results.

The mnemonic-name of the WRITE AFTER ADVANCING statement may also be specified. In the SPECIAL-NAMES Paragraph, the following function-names may be associated with the mnemonic-names:

Function Name	Meaning
C02	Line 3
C03	Line 5
C04	Line 7
...	...
C12	Line 23

CLOSE Statement

When processing is completed, a CLOSE statement must be issued for each of the combined function files. After a CLOSE statement has been issued for any one of the functions, an attempt to perform processing for any of the functions results in abnormal termination.

ACCESS: The manner in which files are referenced by the computer. Access can be sequential (records are referred to one after another in the order in which they appear on the file), or it can be random (the individual records can be referred to in a nonsequential manner).

Actual Decimal Point: The physical representation, using either of the decimal point characters (. or ,), of the decimal point position in a data item. When specified, it will appear in a printed report, and it requires an actual space in storage.

ACTUAL KEY: A key which can be directly used by the system to locate a logical record on a mass storage device. An ACTUAL KEY must be a data item of 5 to 259 bytes in length.

Alphabetic Character: A character which is one of the 26 characters of the alphabet, or a space. In COBOL, the term does not include any other characters.

Alphanumeric Character: Any character in the computer's character set.

Alphanumeric Edited Character: A character within an alphanumeric character string which contains at least one B or 0.

Arithmetic Expression: A statement containing any combination of data-names, numeric literals, and figurative constants, joined together by one or more arithmetic operators in such a way that the statement as a whole can be reduced to a single numeric value.

Arithmetic Operator: A symbol (single character or 2-character set) or COBOL verb which directs the system to perform an arithmetic operation. The following list shows arithmetic operators:

Meaning	Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/
Exponentiation	**

Assumed Decimal Point: A decimal point position which does not involve the existence of an actual character in a data item. It does not occupy an actual space in storage, but is used by the compiler to align a value properly for calculation.

BLOCK: In COBOL, a group of characters or records which is treated as an entity when moved into or out of the computer. The term is synonymous with the term Physical Record.

Buffer: A portion of main storage into which data is read or from which it is written.

Byte: A sequence of eight adjacent binary bits. When properly aligned, two bytes form a halfword, four bytes a fullword, and eight bytes a doubleword.

Channel: A device that directs the flow of information between the computer main storage and the input/output devices.

Character: One of a set of indivisible symbols that can be arranged in sequences to express information. These symbols include the letters A through Z, the decimal digits 0 through 9, punctuation symbols, and any other symbols which will be accepted by the data-processing system.

Character Set: All the valid COBOL characters. The complete set of 51 characters is listed in "Language Considerations."

Character String: A connected sequence of characters. All COBOL characters are valid.

Checkpoint: A reference point in a program at which information about the contents of core storage can be recorded so that, if necessary, the program can be restarted at an intermediate point.

Class Condition: A statement that the content of an item is wholly alphabetic or wholly numeric. It may be true or false.

Clause: A set of consecutive COBOL words whose purpose is to specify an attribute of an entry. There are three types of clauses: data, environment, and file.

COBOL Character: Any of the 51 valid characters (see CHARACTER) in the COBOL character set. The complete set is listed in "Language Considerations."

Collating Sequence: The arrangement of all valid characters in the order of their relative precedence. The collating sequence of a computer is part of the computer design -- each acceptable character has a predetermined place in the sequence. A collating sequence is used primarily in comparison operations.

COLUMN Clause: A COBOL clause used to identify a specific position within a report line.

Comment: An annotation in the Identification Division or Procedure Division of a COBOL source program. A comment is ignored by the compiler. As an IBM extension, comments may be included at any point in a COBOL source program.

Communication Description: In COBOL teleprocessing, an implicitly defined fixed-format storage area that serves as the interface between the COBOL object program and the Message Control Program (MCP). It is specified in the Communication Section.

Communication Description Entry: An entry in the Communication Section of the Data Division that describes the interface between the MCP and the COBOL TP object program. The entry is composed of the level indicator CD, followed by a cd-name, and then optionally followed by a set of independent clauses.

Communication Section: The section in the Data Division that describes the interface area between the MCP and the COBOL TP program. It is composed of one or more CD description entries that define the fields in the interface area.

Communications Device: A mechanism (hardware or hardware-software) capable of sending data to a queue and/or receiving data from a queue. This mechanism may be a computer or a peripheral device. One or more programs containing Communication Description entries and residing within the same computer define one or more of these mechanisms.

Compile Time: The time during which a COBOL source program is translated by the COBOL compiler into a machine language object program.

Compiler: A program which translates a program written in a higher level language into a machine language object program.

Compiler Directing Statement: A COBOL statement which causes the compiler to take a specific action at compile time, rather than the object program to take a particular action at execution time.

Compound Condition: A statement that tests two or more relational expressions. It may be true or false.

Condition:

- One of a set of specified values a data item can assume.
- A simple conditional expression: relation condition, class condition, condition-name condition, sign condition, NOT condition.

Conditional Statement: A statement which specifies that the truth value of a condition is to be determined, and that the subsequent action of the object program is dependent on this truth value.

Conditional Variable: A data item that can assume more than one value; the value(s) it assumes has a condition-name assigned to it.

Condition-Name: The name assigned to a specific value, set of values, or range of values, that a data item may assume.

Condition-name Condition: A statement that the value of a conditional variable is one of a set (or range) of values of a data item identified by a condition-name. The statement may be true or false.

CONFIGURATION SECTION: A section of the Environment Division of the COBOL program. It describes the overall specifications of computers.

Connective: A word or a punctuation character that does one of the following:

- Associates a data-name or paragraph-name with its qualifier
- Links two or more operands in a series
- Forms a conditional expression

CONSOLE: A COBOL mnemonic-name associated with the console typewriter.

Contiguous Items: Consecutive elementary or group items in the Data Division that have a definite relationship with each other.

Control Break: A recognition of a change in the contents of a control data item that governs a hierarchy.

Control Bytes: Bytes associated with a physical record that serve to identify the record and indicate its length, blocking factor, etc.

Control Data Item: A data item that is tested each time a report line is to be printed. If the value of the data item has changed, a control break occurs and special actions are performed before the line is printed.

CONTROL FOOTING: A report group that occurs at the end of the control group of which it is a member.

Control Group: An integral set of related data that is specifically associated with a control data item.

CONTROL HEADING: A report group that occurs at the beginning of the control group of which it is a member.

Control Hierarchy: A designated order of specific control data items. The highest level is the final control; the lowest level is the minor control.

Core Storage: Storage within the central processing unit of the computer, so called because this storage exists in the form of magnetic cores.

Data Description Entry: An entry in the Data Division that is used to describe the characteristics of a data item. It consists of a level number, followed by an optional data-name, followed by data clauses that fully describe the format the data will take. An elementary data description entry (or item) cannot logically be subdivided further. A group data description entry (or item) is made up of a number of related group and/or elementary items.

DATA DIVISION: One of the four main component parts of a COBOL program. The Data Division describes the files to be used in the program and the records contained within the files. It also describes any internal Working-Storage records that will be needed (see "Data Division" for full details).

Data Item: A unit of recorded information that can be identified by a symbolic name or by a combination of names and subscripts. Elementary data items cannot logically be subdivided. Group data items are made up of logically related group and/or elementary items, and can be a logical group within a record or can itself be a complete record.

Data-name: A name assigned by the programmer to a data item in a COBOL program. It must contain at least one alphabetic character.

DECLARATIVES: A set of one or more compiler-directing sections written at the beginning of the Procedure Division of a COBOL program. The first section is preceded by the header DECLARATIVES. The last section is followed by the header END DECLARATIVES. There are three options:

1. Input/output label handling
2. Input/output error-checking procedures
3. Report Writing procedures

Each has its standard format (see "Procedure Division").

Delimiter: A character or sequence of contiguous characters that identify the end of a string of characters and that separate the string of characters from the following string of characters. A delimiter is not part of the string of characters that it delimits.

Destination: In teleprocessing, the symbolic identification of the receiver of a transmission (i.e., a message) from a queue.

Destination Queue: In teleprocessing, an MCP storage queue for one or more messages from one or more remote stations or to one or more remote stations. Destination queues serve as buffers between a COBOL TP program and the remote stations.

Device-number: The reference number assigned to any external device.

Digit: Any of the numerals from 0 through 9. In COBOL, the term is not used in reference to any other symbol.

DIVISION: One of the four major portions of a COBOL program:

- IDENTIFICATION DIVISION, which names the program.
- ENVIRONMENT DIVISION, which indicates the machine equipment and equipment features to be used in the program.
- DATA DIVISION, which defines the nature and characteristics of data to be processed.

- PROCEDURE DIVISION, which consists of statements directing the processing of data in a specified manner at execution time.

Division Header: The COBOL words that indicate the beginning of a particular division of a COBOL program. The four division headers are:

- IDENTIFICATION DIVISION.
- ENVIRONMENT DIVISION.
- DATA DIVISION.
- PROCEDURE DIVISION.

Division-name: The name of one of the four divisions of a COBOL program.

EBCDIC Character: Any one of the symbols included in the eight-bit EBCDIC (Extended Binary-Coded-Decimal Interchange Code) set. All 51 COBOL characters are included.

Editing Character: A single character or a fixed two-character combination used to create proper formats for output reports (see "Language Considerations" for a complete list of editing characters).

Elementary Item: A data item that cannot logically by subdivided.

Entry: Any consecutive set of descriptive clauses terminated by a period, written in the Identification, Environment, or Procedure Divisions of a COBOL program.

Entry-name: A programmer-specified name that establishes an entry point into a COBOL subprogram.

ENVIRONMENT DIVISION: One of the four main component parts of a COBOL program. The Environment Division describes the computers upon which the source program is compiled and those on which the object program is executed, and provides a linkage between the logical concept of files and their records, and the physical aspects of the devices on which files are stored (see "Environment Division" for full details).

Execution Time: The time at which an object program actually performs the instructions coded in the Procedure Division, using the actual data provided.

Exponent: A number, indicating how many times another number (the base) is to be repeated as a factor. Positive exponents denote multiplication, negative exponents denote division, fractional exponents denote a root of a quantity. In COBOL, exponentiation is indicated with the symbol ** followed by the exponent.

F-mode Records: Records of a fixed length. Blocks may contain more than one record.

Figurative Constant: A reserved word that represents a numeric value, a character, or a string of repeated values or characters. The word can be written in a COBOL program to represent the values or characters without being defined in the Data Division (see "Language Considerations" for a complete list).

FILE-CONTROL: The name and header of an Environment Division paragraph in which the data files for a given source program are named and assigned to specific input/output devices.

File Description: An entry in the File Section of the Data Division that provides information about the identification and physical structure of a file.

File-name: A name assigned to a set of input data or output data. A file-name must include at least one alphabetic character.

FILE SECTION: A section of the Data Division that contains descriptions of all externally stored data (or files) used in a program. Such information is given in one or more file description entries.

Floating-Point Literal: A numeric literal whose value is expressed in floating-point notation -- that is, as a decimal number followed by an exponent which indicates the actual placement of the decimal point.

Function-name: A name, specified by IBM, that identifies system logical units, printer and card punch control characters, and report codes. When a function-name is associated with a mnemonic name in the Environment Division, the mnemonic-name may then be substituted in any format in which such substitution is valid.

Group Item: A data item made up of a series of logically related elementary items. It can be part of a record or a complete record.

Header Label: A record that identifies the beginning of a physical file or a volume.

High-Order: The leftmost position in a string of characters.

IDENTIFICATION DIVISION: One of the four main component parts of a COBOL program. The Identification Division identifies the source program and the object program and, in addition, may include such documentation as the author's name, the installation where written, date written, etc., (see "Identification Division" for full details).

Identifier: A data-name, unique in itself, or made unique by the syntactically correct combination of qualifiers, subscripts, and/or indexes.

Imperative-Statement: A statement consisting of an imperative verb and its operands, which specifies that an action be taken, unconditionally. An imperative-statement may consist of a series of imperative-statements.

Index: A computer storage position or register, the contents of which identify a particular element in a table.

Index-Data Item: A data item in which the contents of an index can be stored without conversion to subscript form.

Index-name: A name, given by the programmer, for an index of a specific table. An index-name must contain at least one alphabetic character. It is one word (4 bytes) in length.

Indexed-Data-name: A data-name identifier which is subscripted with one or more index-names.

INPUT-OUTPUT SECTION: In the Environment Division, the section that names the files and external media needed by an object program. It also provides information required for the transmission and handling of data during the execution of an object program.

INPUT PROCEDURE: A set of statements that is executed each time a record is released to the sort file. Input procedures are optional; whether they are used or not depends upon the logic of the program.

Input Queue: In teleprocessing, an MCP destination queue from which the COBOL TP program accepts messages from the remote stations.

Integer: A numeric data item or literal that does not include any character positions to the right of the decimal point, actual or assumed. Where the term "integer" appears in formats, "integer" must not be a numeric data item.

INVALID KEY Condition: A condition that may arise at execution time in which the value of a specific key associated with a mass storage file does not result in a correct reference to the file (see the READ, REWRITE, START, and WRITE statements for the specific error conditions involved).

I-O-CONTROL: The name, and the header, for an Environment Division paragraph in which object program requirements for specific input/output techniques are specified. These techniques include rerun checkpoints, sharing of same areas by several data files, and multiple file storage on a single tape device.

KEY: One or more data items, the contents of which identify the type or the location of a record, or the ordering of data.

Key Word: A reserved word whose employment is essential to the meaning and structure of a COBOL statement. In this manual, key words are indicated in the formats of statements by underscoring. Key words are included in the reserved word list.

Level Indicator: Two alphabetic characters that identify a specific type of file, or the highest position in a hierarchy. The level indicators are: FD, RD, SD.

Level Number: A numeric character or 2-character set that identifies the properties of a data description entry. Level numbers 01 through 49 define group items, the highest level being identified as 01, and the subordinate data items within the hierarchy being identified with level numbers 02 through 49. Level numbers 66, 77, and 88 identify special properties of a data description entry in the Data Division.

Library-name: The name of a member of a data set containing COBOL entries, used with the COPY and BASIS statements.

LINKAGE SECTION: A section of the Data Division that describes data made available from another program.

Literal: A character string whose value is implicit in the characters themselves. The numeric literal 7 expresses the value 7, and the nonnumeric literal "CHARACTERS" expresses the value CHARACTERS.

Logical Operator: A COBOL word that defines the logical connections between relational operators. The three logical operators and their meanings are:

OR (logical inclusive -- either or both)

AND (logical connective -- both)

NOT (logical negation)

(See "Procedure Division" for a more detailed explanation.)

Logical Record: The most inclusive data item, identified by a level-01 entry. It consists of one or more related data items.

Low-Order: The rightmost position in a string of characters.

Main Program: The highest level COBOL program involved in a step. (Programs written in other languages that follow COBOL linkage conventions are considered COBOL programs in this sense.)

Mantissa: The decimal part of a logarithm. Therefore, the part of a floating-point number that is expressed as a decimal fraction.

Mass Storage: A storage medium -- disk, drum, or data cell -- in which data can be collected and maintained in a sequential, direct, indexed or relative organization.

Mass Storage File: A collection of records assigned to a mass storage device.

Mass Storage File Segment: A part of a mass storage file whose beginning and end are defined by the FILE-LIMIT clause in the Environment Division.

Message: In teleprocessing, a string of characters associated with an end-of-message indicator or end-of-group indicator. A message may consist of one or more related message segments. See "Message Indicators".

Message Control Program (MCP): A TCAM communications control program that supports the processing of messages.

Message Indicators: In COBOL TP programs, three message indicators are allowed. Each signals that some specific condition exists:

EGI indicates logical end-of-group of a group of messages
EMI indicates end-of-message
ESI indicates end-of-segment

The hierarchy of message indicators is in the order of the preceding list. Within this hierarchy an EGI is conceptually equivalent to an EGI EMI, and ESI; an EMI is conceptually equivalent to an EMI and an ESI. Thus a segment may be terminated by an EGI, EMI, or ESI, and a message may be terminated by an EGI or EMI.

Message Segment: In teleprocessing, a string of characters that forms a logical subdivision of a message, and is normally associated with an end-of-segment indicator. A message segment is the equivalent of a TCAM record. See "Message Indicators".

Mnemonic-name: A programmer-supplied word associated with a specific function-name in the Environment Division. It then may be written in place of the function-name in any format where such a substitution is valid.

MODE: The manner in which records of a file are accessed or processed.

Name: A word composed of not more than 30 characters, which defines a COBOL operand (see "Language Considerations" for a more complete discussion).

Noncontiguous Item: A data item in the Working-Storage Section of the Data Division which bears no relationship to other data items.

Nonnumeric Literal: A character string bounded by quotation marks, which means literally itself. For example, "CHARACTER" is the literal for, and means, CHARACTER. The string of characters may include any characters in the computer's set, with the exception of the quotation mark. Characters that are not COBOL characters may be included.

Nonswitched Line: In teleprocessing, a line that is a continuous link between a remote station and the computer. It may connect the central computer with either a single station or more than one station.

Numeric Character: A character that belongs to one of the set of digits 0 through 9.

Numeric-Edited Character: A numeric character which is in such a form that it may be used in a printed output. It may consist of external decimal digits 0 through 9, the decimal point, commas, the dollar sign, etc., as the programmer wishes (see "Data Division" for a fuller explanation).

Numeric Item: An item whose description restricts its contents to a value represented by characters chosen from the digits 0 through 9; if signed, the item may also contain a + or -, or other representation of an operational sign.

Numeric Literal: A numeric character or string of characters whose value is implicit in the characters themselves. Thus, 777 is the literal as well as the value of the number 777.

OBJECT-COMPUTER: The name of an Environment Division paragraph in which the computer upon which the object program will be run is described.

Object Program: The set of machine language instructions that is the output from the compilation of a COBOL source program. The actual processing of data is done by the object program.

Object Time: The time during which an object program is executed.

Operand: The "object" of a verb or an operator. That is, the data or equipment governed or directed by a verb or operator.

Operational Sign: An algebraic sign associated with a numeric data item, which indicates whether the item is positive or negative.

Optional Word: A reserved word included in a specific format only to improve the readability of a COBOL statement. If the programmer wishes, optional words may be omitted.

OUTPUT PROCEDURE: A set of programmer-defined statements that is executed each time a sorted record is returned from the sort file. Output procedures are optional; whether they are used or not depends upon the logic of the program.

Output Queue: In teleprocessing, an MCP destination queue into which a COBOL TP program places messages for one or more remote stations.

Overflow Condition: In string manipulation, a condition that occurs when the sending area(s) contain untransferred characters after the receiving area(s) have been filled.

Overlay: The technique of repeatedly using the same areas of internal storage during different stages in processing a problem.

PAGE: A physical separation of continuous data in a report. The separation is based on internal requirements and/or the physical characteristics of the reporting medium.

PAGE FOOTING: A report group at the end of a report page which is printed before a page control break is executed.

PAGE HEADING: A report group printed at the beginning of a report page, after a page control break is executed.

Paragraph: A set of one or more COBOL sentences, making up a logical processing entity, and preceded by a paragraph-name or a paragraph header.

Paragraph Header: A word followed by a period that identifies and precedes all paragraphs in the Identification Division and Environment Division.

Paragraph-name: A programmer-defined word that identifies and precedes a paragraph.

Parameter: A variable that is given a specific value for a specific purpose or process. In COBOL, parameters are most often used to pass data values between calling and called programs.

Physical Record: A physical unit of data, synonymous with a block. It can be composed of a portion of one logical record, of one complete logical record, or of a group of logical records.

Print Group: An integral set of related data within a report.

Priority-number: A number, ranging in value from 0 to 99, which classifies source program sections in the Procedure Division (see "Segmentation" for more information).

Procedure: One or more logically connected paragraphs or sections within the Procedure Division, which direct the computer to perform some action or series of related actions.

PROCEDURE DIVISION: One of the four main component parts of a COBOL program. The Procedure Division contains instructions for solving a problem. The Procedure Division may contain imperative-statements, conditional statements, paragraphs, procedures, and sections (see "Procedure Division" for full details).

Procedure-name: A word that precedes and identifies a procedure, used by the programmer to transfer control from one point of the program to another.

Process: Any operation or combination of operations on data.

Program-name: A word in the Identification Division that identifies a COBOL source program.

Punctuation Character: A comma, semicolon, period, quotation mark, left or right parenthesis, or a space.

Qualifier: A group data-name that is used to reference a non-unique data-name at a lower level in the same hierarchy, or a section-name that is used to reference a non-unique paragraph. In this way, the data-name or the paragraph-name can be made unique.

Queue: In teleprocessing, a logical collection of messages awaiting transmission or processing.

Queue Blocks: In teleprocessing, blocks containing status and control information pertaining to the message being processed and to each active queue. Created when a queue is first accessed by a COBOL IP run unit, all queue blocks in one region/partition are chained to each other.

Queue Name: In teleprocessing, a symbolic name that indicates to the MCP the logical path by which a message, or portion of a completed message, may be accessible in a queue. (The first eight characters must match the DDname of the DD statement that specifies the queue.)

Random Access: An access mode in which specific logical records are obtained from or placed into a mass storage file in a nonsequential manner.

RECORD: A set of one or more related data items grouped for handling either internally or by the input/output systems (see "Logical Record").

Record Description: The total set of data description entries associated with a particular logical record.

Record-name: A data-name that identifies a logical record.

REEL: A module of external storage associated with a tape device.

Relation Character: A character that expresses a relationship between two operands. The following are COBOL relation characters:

Character	Meaning
>	Greater than
<	Less than
=	Equal to

Relation Condition: A statement that the value of an arithmetic expression or data item has a specific relationship to another arithmetic expression or data item. The statement may be true or false.

Relational Operator: A reserved word, or a group of reserved words, or a group of reserved words and relation characters. A relational operator plus programmer-defined operands make up a relational expression. A complete listing is given in "Procedure Division."

Remote Station: In teleprocessing, a control unit and one or more input/output devices connected to the central computer through common carrier facilities. A remote station may be a terminal device, or it may be another computer.

REPORT: A presentation of a set of processed data described in a Report File.

Report Description Entry: An entry in the Report Section of the Data Division that names and describes the format of a report to be produced.

Report File: A collection of records, produced by the Report Writer, that can be used to print a report in the desired format.

REPORT FOOTING: A report group that occurs, and is printed, only at the end of a report.

Report Group: A set of related data that makes up a logical entity in a report.

REPORT HEADING: A report group that occurs, and is printed, only at the beginning of a report.

Report Line: One row of printed characters in a report.

Report-name: A data-name that identifies a report.

REPORT SECTION: A section of the Data Division that contains one or more Report Description entries.

Reserved Word: A word used in a COBOL source program for syntactical purposes. It must not appear in a program as a user-defined operand.

Routine: A set of statements in a program that causes the computer to perform an operation or series of related operations.

Run Unit: A set of one or more object programs which function, at object time, as a unit to provide problem solutions. This compiler considers a run unit to be the highest level calling program plus all called subprograms.

S-Mode Records: Records which span physical blocks. Records may be fixed or variable in length; blocks may contain one or more segments. Each segment contains a segment-descriptor field and a control field

indicating whether it is the first and/or last or an intermediate segment of the record. Each block contains a block-descriptor field.

SECTION: A logically related sequence of one or more paragraphs. A section must always be named.

Section Header: A combination of words that precedes and identifies each section in the Environment, Data, and Procedure Divisions.

Section-name: A word specified by the programmer that precedes and identifies a section in the Procedure Division.

Sentence: A sequence of one or more statements, the last ending with a period followed by a space.

Separator: An optional word or character that improves readability.

Sequential Access: An access mode in which logical records are obtained from or placed into a file in such a way that each successive access to the file refers to the next subsequent logical record in the file. The order of the records is established by the programmer when creating the file.

Sequential Processing: The processing of logical records in the order in which records are accessed.

Sign Condition: A statement that the algebraic value of a data item is less than, equal to, or greater than zero. It may be true or false.

Simple Condition: An expression that can have two values, and causes the object program to select between alternate paths of control, depending on the value found. The expression can be true or false.

Slack Bytes: Bytes inserted between data items or records to ensure correct alignment of some numeric items. Slack bytes contain no meaningful data. In some cases, they are inserted by the compiler; in others, it is the responsibility of the programmer to insert them. The SYNCHRONIZED clause instructs the compiler to insert slack bytes when they are needed for proper alignment. Slack bytes between records are inserted by the programmer.

Sort File: A collection of records that is sorted by a SORT statement. The sort file is created and used only while the sort function is operative.

Sort-File-Description Entry: An entry in the File Section of the Data Division that names and describes a collection of records that is used in a SORT statement.

Sort-file-name: A data-name that identifies a Sort File.

Sort-key: The field within a record on which a file is sorted.

Sort-work-file: A collection of records involved in the sorting operation as this collection exists on intermediate device(s).

Source: In teleprocessing, the symbolic identification of the originator of a transmission to a queue.

SOURCE-COMPUTER: The name of an Environment Division paragraph. In it, the computer upon which the source program will be compiled is described.

Source Program: A problem-solving program written in COBOL.

Special Character: A character that is neither numeric nor alphabetic. Special characters in COBOL include the space ( ), the period (.), as well as the following: + - * / = $, ; ") (

SPECIAL-NAMES: The name of an Environment Division paragraph, and the paragraph itself, in which names supplied by IBM are related to mnemonic-names specified by the programmer. In addition, this paragraph can be used to exchange the functions of the comma and the period, or to specify a substitution character for the currency sign, in the PICTURE string.

Special Register: Compiler-generated storage areas primarily used to store information produced with the use of specific COBOL features. The special registers are: TALLY, LINE-COUNTER, PAGE-COUNTER, CURRENT-DATE, TIME-OF-DAY, LABEL-RETURN, RETURN-CODE, SORT-RETURN, SORT-FILE-SIZE, SORT-CORE-SIZE, and SORT-MODE-SIZE.

Standard Data Format: The concept of actual physical or logical record size in storage. The length in the Standard Data Format is expressed in the number of bytes a record occupies and not necessarily the number of characters, since some characters take up one full byte of storage and others take up less.

Statement: A syntactically valid combination of words and symbols written in the Procedure Division. A statement combines COBOL reserved words and programmer-defined operands.

Subject of entry: A data-name or reserved word that appears immediately after a level indicator or level number in a Data Division entry. It serves to reference the entry.

Subprogram: A COBOL program that is invoked by another COBOL program. (Programs written in other languages that follow COBOL linkage conventions are COBOL programs in this sense.)

Subscript: An integer or a variable whose value references a particular element in a table.

Switched Line: In teleprocessing, a communication line for which no single continuous path between the central computer and the remote station exists. Several alternative paths are available for transmission; the common carrier switching equipment selects the path. The remote station is continuously connected to the switching center by an access line associated with a specific telephone number.

SYSIN: The system logical input device.

SYSOUT: The system logical output device.

SYSPUNCH: The system logical punch device.

System-name: A name that identifies any particular external device used with the computer, and characteristics of files contained within it.

Table: A collection and arrangement of data in a fixed form for ready reference. Such a collection follows some logical order, expressing particular values (functions) corresponding to other values (arguments) by which they are referenced.

Table Element: A data item that belongs to the set of repeated items comprising a table.

Test Condition: A statement that, taken as a whole, may be either true or false, depending on the circumstances existing at the time the expression is evaluated.

Trailer Label: A record that identifies the ending of a physical file or of a volume.

U-mode Records: Records of unspecified length. They may be fixed or variable in length; there is only one record per block.

Unary Operator: An arithmetic operator (+ or -) that can precede a single variable, a literal, or a left parenthesis in an arithmetic expression. The plus sign multiplies the value by +1; the minus sign multiplies the value by -1.

UNIT: A module of external storage. Its dimensions are determined by IBM.

V-mode Records: Records of variable length. Blocks may contain more than one record. Each record contains a record length field, and each block contains a block length field.

Variable: A data item whose value may be changed during execution of the object program.

Verb: A COBOL reserved word that expresses an action to be taken by a COBOL compiler or an object program.

Volume: A module of external storage. For tape devices it is a reel; for mass storage devices it is a unit.

Volume Switch Procedures: Standard procedures executed automatically when the end of a unit or reel has been reached before end-of-file has been reached.

WORD:

1. In COBOL: A string of not more than 30 characters, chosen from the following: the letters A through Z, the digits 0 through 9, and the hyphen (-). The hyphen may not appear as either the first or last character.

2. In System/360: A fullword is four bytes of storage; a doubleword is eight bytes of storage; a halfword is two bytes of storage.

Word Boundary: Any particular storage position at which data must be aligned for certain processing operations in System/360. The halfword boundary must be divisible by 2, the fullword boundary must be divisible by 4, the doubleword boundary must be divisible by 8.

WORKING-STORAGE SECTION: A section-name (and the section itself) in the Data Division. The section describes records and noncontiguous data items that are not part of external files, but are developed and processed internally. It also defines data items whose values are assigned in the source program.

(Where more than one page reference is given, the major reference is first.)

IBM OS Full American
National Standard COBOL
GC28-6396-5

Reader's Comment Form

Your comments about this publication will help us to improve it for you. Comment in the space below, giving specific page and paragraph references whenever possible. All comments become the property of IBM.

Please do not use this form to ask technical questions about IBM systems and programs or to request copies of publications. Rather, direct such questions or requests to your local IBM representative.

If you would like a reply, please provide your name, job title, and business address (including ZIP code).

Fold on two lines, staple, and mail. No postage necessary if mailed in the U.S.A. (Elsewhere, any IBM representative will be happy to forward your comments.) Thank you for your cooperation.

Fold and Staple

First Class Permit
Number 439
Palo Alto, California

Business Reply Mail

No postage necessary if mailed in the U.S.A.

Postage will be paid by:

IBM Corporation
System Development Division
LDF Publishing—Department J04
1501 California Avenue
Palo Alto, California 94304

Fold and Staple

IBM OS Full ANS COBOL Printed in U.S.A. GC28-6396-5

IBM®

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)