File Number S360-24
Form C24-5029-2

OS

IBM

Systems Reference Library

IBM System/360
Operating System
COBOL (E) Programmer's Guide

This reference publication describes how to compile, linkage edit, and execute a COBOL (E-Level Subset) program. It also describes the output of compilation and execution, how to make optimal use of the compiler and a load module, and compiler and load module restrictions.

The corequisite to this publication is IBM System/360 Operating System: COBOL Language, Form C28-6516.

Other publications related to this one are:
- IBM System/360 Principles of Operation, Form A22-6821.
- IBM System/360 Operating System: Control Program Services, Form C28-6541.
- IBM System/360 Operating System: Job Control Language, Form C28-6539.
- IBM System/360 Operating System: Utilities, Form C28-6586.
- IBM System/360 Operating System: Linkage Editor, Form C28-6538.
- IBM System/360 Operating System: Control Program Messages and Completion Codes, Form C28-6608.

For a list of other associated System/360 publications, see the IBM System/360 Bibliography, Form A22-6822.

PREFACE

The purpose of the Programmer's Guide is to enable programmers to compile, linkage edit, and execute COBOL (E-Level Subset) programs under control of IBM System/360 Operating System. The COBOL (E-Level Subset) language is described in the publication IBM System/360 Operating System: COBOL Language, Form C28-6516, which is a corequisite to this publication.

The Programmer's Guide is organized to fulfill its purpose at three levels:

1. Programmers who wish to use the cataloged procedures as provided by IBM need read only the Introduction and Job-Control Language sections to understand the job-control statements, and the Job Processing section to use cataloged procedures for compiling, linkage editing, and executing COBOL programs. The Programming Considerations and System Output sections are recommended for programmers who want to use the COBOL language more effectively.
2. Programmers who are also concerned with creating and retrieving data sets, optimizing the use of I/O devices, or temporarily modifying IBM-supplied cataloged procedures should read the entire Programmer's Guide.
3. Programmers concerned with making extensive use of the operating system facilities, such as writing their own cataloged procedures, should also read the entire Programmer's Guide in conjunction with the publications listed on the front cover of this publication.

In addition to providing reference information on compiling, linkage editing, and executing programs, this publication contains appendices that:

1. Give several examples of processing.
2. Contain detailed descriptions of the diagnostic messages produced during compilation and load module execution.

Third Edition, February 1967

This edition, Form C24-5029-2 is a major revision of Form C24-5029-1, which it obsoletes. Changes to this publication are indicated by a vertical line at the left of the text and portions of the figures affected. Significant changes and additions to the specifications contained in this publication will be reported in subsequent revisions or Technical Newsletters.

Changes are indicated by a vertical line to the left of affected text and to the left of affected parts of figures. A dot (•) next to a figure title or page number indicates that the entire figure or page should be reviewed.

Specifications contained herein are subject to change from time to time. Any such change will be reported in subsequent revisions or Technical Newsletters.

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form is provided at the back of this publication for readers' comments. If the form has been removed, comments may be addressed to IBM Corporation, Publications, Dept. D39, 1271 Avenue of the Americas New York, N.Y. 10020

CONTENTS

The IBM System/360 Operating System (referred to here as the operating system) consists of a control program and processing programs. The control program supervises execution of all processing programs, such as the COBOL-E compiler, and all problem programs, such as a COBOL problem program. Therefore, to execute a COBOL program, the programmer must first communicate with the operating system. The medium of communication between the programmer and the operating system is the job-control language.

Job-control language statements define units of work to the operating system. Two units of work are recognized: the job and the job step. The statements that define these units of work are the JOB and the EXEC (execute) statements. Another important statement is the DD (data definition) statement, which gives the operating system information about data used in jobs and job steps. The flow of control statements and any data placed in the flow of control statements is called the input stream.

Note: Throughout this publication certain arbitrary options are given in illustrative examples. Some of the options used are a function of system generation; therefore, these examples may not be valid for all systems.

JOB AND JOB STEP RELATIONSHIP

When a programmer is given a problem, he analyzes that problem and defines a precise problem-solving procedure; that is, he writes a program or a series of programs. Executing a main program (and its subprograms) is a job step to the operating system. A job consists of executing one or more job steps.

At its simplest, a job consists of one job step. For example, executing a payroll program is a job step.

In another sense, a job consists of several interdependent job steps, such as a compilation, linkage edit, and execution. Job steps can be related to each other as follows.

1. One job step may pass intermediate results recorded on an external storage volume to a later job step.
2. Whether or not a job step is executed may depend on results of preceding steps.

In the series of job steps (compilation, linkage edit, and execution), each step can be a separate job with one job step in each job. However, designating several related job steps as one job is more efficient: processing time is decreased because only one job is defined, and interdependence of job steps may be stated. (Interdependence of jobs cannot be stated.) Each step may be defined as a job step within one job that encompasses all processing.

```
JOB:  Compile, linkage edit, and execute
  JOB STEP 1:  Compile COBOL program
  JOB STEP 2:  Linkage edit compiled
               program
  JOB STEP 3:  Execute linkage edited
               program
```

Figure 1 illustrates these three job steps.

The important aspect of jobs and job steps is that they are defined by the programmer. He defines a job to the operating system by using a JOB statement; he defines a job step by the EXEC statement.

DATA SETS

In Figure 1, one collection of input data (source program) and one collection of output data (compiled program) are used in job step 1. In the operating system, a collection of data that can be named by the programmer is called a data set. A data set is defined to the operating system by a DD statement.

A data set resides on a volume(s), which is a unit of external storage that is accessible to an input/output device. (For example, a volume may be a reel of tape or a disk pack.)

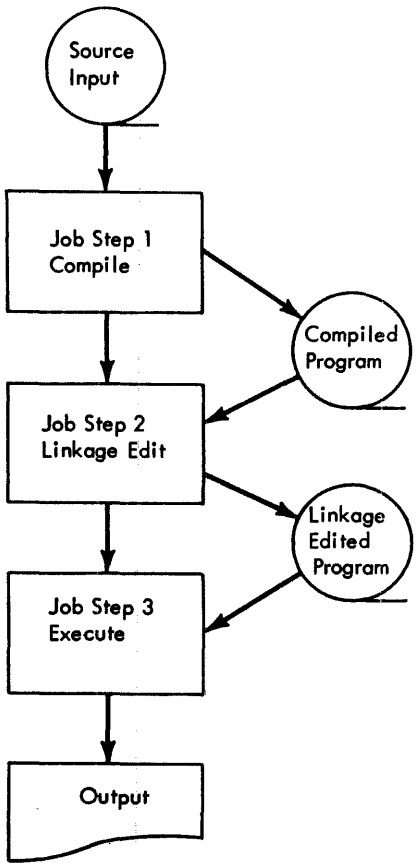


Figure 1. Job Example with Three Job Steps

Several I/O devices grouped together and given a single name when the system is generated constitute a device class. For example, a device class can consist of all the tape devices in the installation, another can consist of the printer, a direct-access device, and a tape device.

The name of a data set and information identifying the volume(s) on which the data set resides may be placed in an index to help the control program find the data set. This index, which is part of an index structure called the catalog, resides on a direct-access volume. Any data set whose name and volume identification are placed in this index is called a cataloged data set. When a data set is cataloged, the information needed to access the data set is its name, and disposition. Other information associated with the data set, such as device type, the position of the data set on the volume, and the format of records in the data set, is available to the control program.

Furthermore, a hierarchy of indexes may be devised to classify data sets and make names for data sets unique. For example, an installation may divide its cataloged data sets into four goups: SCIENCE, ENGRNG, ACCNTS, and INVNTRY. In turn, each of these groups may be subdivided. For example, the ACCNTS group may be subdivided into RECEIVE and PAYABLE; PAYABLE may contain volume identification for the data sets PAYROLL and OVERHEAD. To find the data set PAYROLL, the programmer specifies all indexes beginning with the largest group, ACCNTS; then the next largest group, PAYABLE; finally, the data set PAYROLL. The complete identification needed to find that data set PAYROLL is ACCNTS.PAYABLE.PAYROLL.

Data set names are of two classes: unqualified and qualified. An unqualified name is a data set name or an index name that is not preceded by an index name. A qualified name is a data set name or index name preceded by index names representing index levels; for example, in the preceding text, the qualified name of the data set PAYROLL is ACCNTS.PAYABLE.PAYROLL.

Data set identification may also be based upon the time of generation. In the operating system, a collection of successive, historically related data sets is a generation data group. Each of the data sets is a generation data set. A generation number is attached to the data group name to refer to a particular generation. The most recent generation is 0; the generation previous to 0 is -1; the generation previous to -1 is -2; etc. An index describing a generation data group must exist in the catalog.

For example, a data group named YTDPAY might be used for a payroll application. The generations for the generation data group YTDPAY are:

```
YTDPAY (0)
YTDPAY (-1)
YTDPAY (-2)
   .
   .
   .
```

When a new generation is being created, it is called generation (+n), where n is an integer greater than 0. For example, after a job step has created YTDPAY(+1), the operating system changes its name to YTDPAY(0). The data set that was YTDPAY(0) at the beginning of the job step becomes YTDPAY(-1), etc.

COBOL PROCESSING

In the operating system, a source program is called a source module; a compiled source module is an object module (object program). The object module cannot be executed until it is placed in a format suitable for loading and all references to subprograms are resolved. This is done by an IBM-supplied program, the linkage editor.

The executable output of the linkage editor is a load module. However, the input to the linkage editor may be either object modules or load modules. Linkage editor execution can be expanded further: several object modules and/or load modules may be combined to form one load module. The linkage editor inserts the requested subroutines into the load module. For example, if the compiled object module TEST calls subroutines ALPHA and BETA, the linkage editor combines the object module TEST and the previously linkage edited load modules ALPHA and BETA into one load module. This process is illustrated in Figure 2.

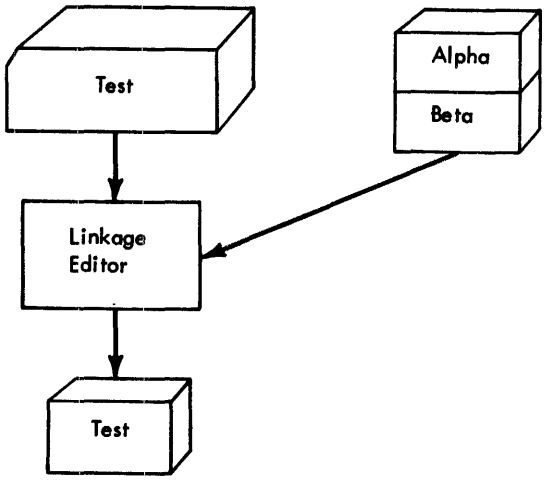


Figure 2. Linkage Editor Execution

A program written in COBOL may call subprograms written in the assembler language as long as the assembler subprogram uses the linkage conventions shown in Appendix B: Assembler Language Subprograms. The linkage editor resolves the references between assembler and COBOL modules.

After an object module is processed by the linkage editor, the resulting load module may be executed. Therefore, to compile, linkage edit, and execute a COBOL program, three or more job steps are necessary:

1. Compile the COBOL source module and any COBOL subprograms not compiled previously to produce one or more object modules. Note that each COBOL compilation requires a job step.

2. Linkage edit the resulting object module(s) and any modules needed to resolve external references to form a load module.

3. Execute the load module.

Figure 3 illustrates the problem program processing; COBOL subprograms and assembler subprograms (load modules) are used to resolve external references.

Each compilation, the linkage editor execution, and the load module execution may be defined as separate jobs, but combining the separate jobs into one job is more efficient.

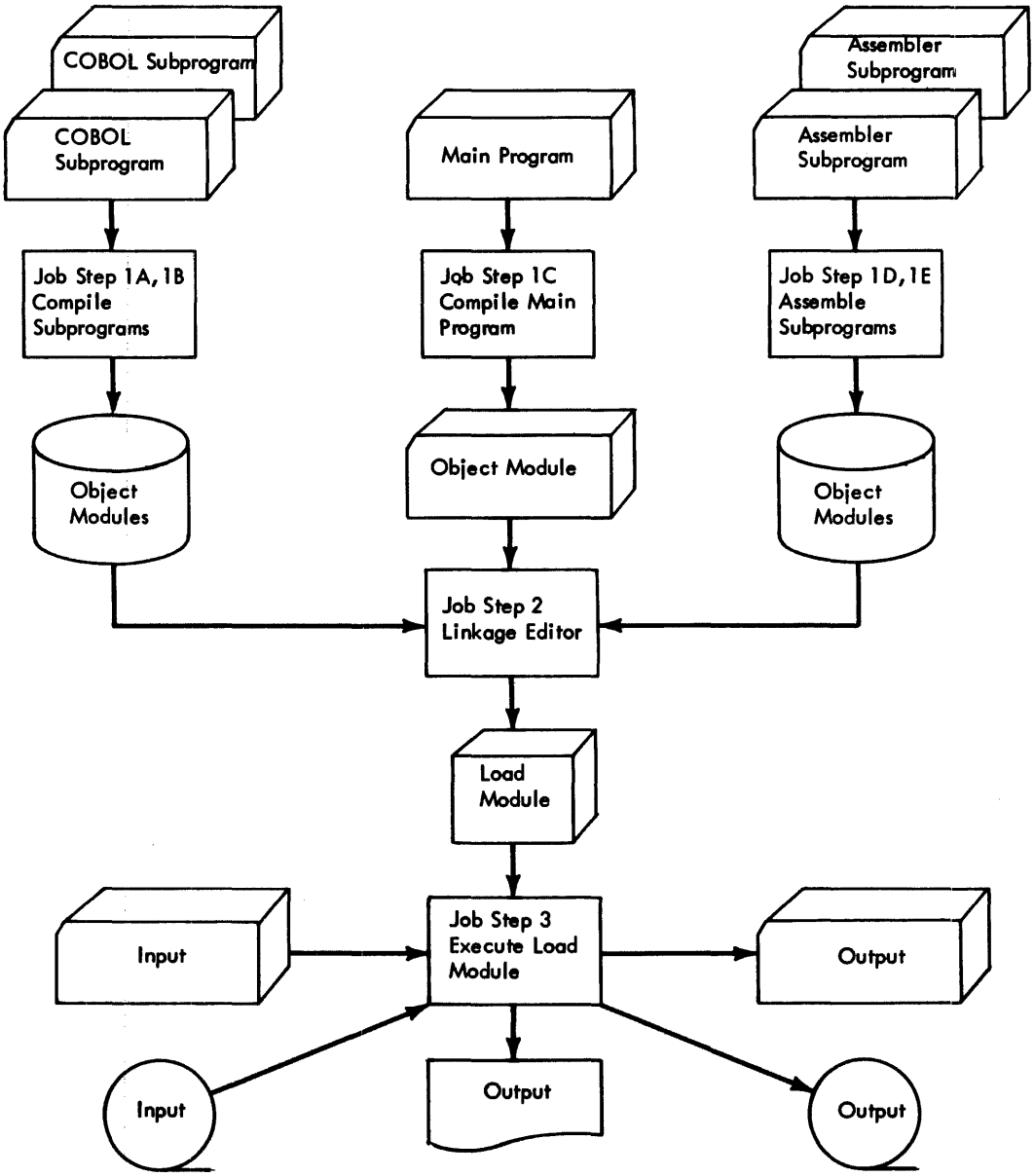


Figure 3. COBOL Processing Example

Data Set Considerations

A data set is defined as a collection of data. The COBOL compiler, linkage editor, and load modules process two types of data sets: sequential data sets and partitioned data sets.

A sequential data set is organized in the same way as a data set that resides on a tape volume, but a sequential data set may reside on any type of volume.

A partitioned data set (PDS) is composed of named, independent groups of sequential data, and resides on a direct access volume. A directory index resides in the PDS, and directs the operating system to any group of sequential data. Each group of sequential data is called a member. Partitioned data sets are used for storage of any type of sequentially organized data. In particular, they are used for storage of source and load modules (each module is a member). In fact, a load module can be executed only if it is a member of a partitioned data set. A PDS of load

modules is created by either the linkage editor or a utility program.

Load modules originally written in COBOL can access only sequential data sets.

Cataloged Procedures

An installation may have certain procedures to follow in its daily processing. To reduce the possibility of error in the daily reproduction of job-control statements for a job, a cataloged procedure may be written. A cataloged procedure is a set of EXEC and DD statements placed in a PDS accessed by the operating system. (The JOB statement cannot be cataloged.) A cataloged procedure consists of a procedure step or a series of procedure steps that is defined by EXEC statements. A procedure step in a cataloged procedure is equivalent to a job step in a job. For a job step, data sets must be defined by DD statements. Because DD statements can be included in cataloged procedures, a minimum of DD statement information must be supplied by the programmer.

An EXEC statement in the input stream may invoke a cataloged procedure. Therefore, the definition of job step is extended: executing a load module or invoking a cataloged procedure is a job step to the operating system.

To simplify the steps involved in compiling and linkage editing, three COBOL-E cataloged procedures are supplied by IBM. These three cataloged procedures and their uses are:

COBEC	compile
COBELG	linkage edit and execute
COBECLG	compile, linkage edit, and execute

Any cataloged procedure may be temporarily modified by EXEC and DD statements in the input stream; this temporary modification is called overriding.

The DD statement for overriding a DD name in a catalog procedure must have a DSNAME.

JOB-CONTROL LANGUAGE

The COBOL programmer uses the job-control statements shown in Table 1 to compile, linkage edit, and execute programs.

Table 1. Job-Control Statements

STATEMENT	FUNCTION
JOB	Indicates the beginning of a new job and describes that job.
EXEC	Indicates a job step and describes that job step; indicates the cataloged procedure or load module to be executed.
DD	Describes data sets, and controls device and volume assignment.
delimiter	Separates data sets in the input stream from control statements, it appears after each data set in the input stream i.e., after a COBOL source program.

CODING JOB-CONTROL STATEMENTS

Job-control statements are identified by the initial characters // or /* in card columns 1 and 2, and may contain four fields: name, operation, operand, and comment (Figure 4).

NAME FIELD

The name contains between one and eight alphameric characters, the first of which must be alphabetic. The name begins in card column 3, and is followed by one or more blanks to separate it from the operation field. The name is used:

1. To identify the control statement to the operating system.
2. To enable other control statements in the job to refer to information contained in the named statement.
3. To relate DD statements to I/O statements in the load module.

OPERATION FIELD

The operation field contains one of the following operation codes:

JOB
EXEC
DD

or, if the statement is a delimiter statement, the operation field is blank. The operation code is preceded and followed by one or more blanks.

OPERAND FIELD

The operand field contains the parameters that provide required and optional information to the operating system. The parameters are separated by commas. The

FORMAT	APPLICABLE CONTROL STATEMENTS
//Name Operation Operand [Comment]	JOB,EXEC,DD
// Operation Operand [Comment]	EXEC,DD
/* [Comment]	delimiter

Figure 4. Job-Control Statement Formats

operand field is ended by placing one or more blanks after the last parameter. There are two types of parameters, positional and keyword.

Positional Parameters: Positional parameters are placed first in the operand field and must appear in the specified order. If a positional parameter is omitted and other positional parameters follow, the omission must be indicated by a comma.

Keyword Parameters: A keyword parameter may be placed anywhere in the operand field following the positional parameters. A keyword parameter consists of a keyword, followed by an equal sign, followed by a single value or a list of subparameters. If there is a list of subparameters, the list must be enclosed in parentheses or apostrophes, and the subparameters in the list must be separated by commas. Keyword parameters are not order dependent; that is, they may appear in any order.

Subparameters: Subparameters are either positional or keyword. Positional and keyword subparameters are noted in the definition of control statements. Positional subparameters appear first in the parameter and must appear in the specified order. If a positional subparameter is omitted and other positional subparameters follow, the omission must be indicated by a comma.

COMMENTS

Comments must be separated from the last parameter (or the * in a delimiter statement) by one or more blanks and may appear in the remaining columns up to and including column 71.

CONTINUING CONTROL STATEMENTS

A control statement can be written in card columns 1 through 71. If a control statement exceeds 71 columns, it may be continued onto the next card. If a statement is continued, it must be interrupted after the comma that follows the last parameter or subparameter on the card, and a nonblank character must be placed in column 72. The continuation card must contain // in columns 1 and 2, columns 3 through 15 must be blank, and the continued portion of the statement must begin in any column after column 16. There is no limit to the number of continuation cards used for a single control statement.

Note: Excessive continuation cards should be avoided, whenever possible, to reduce processing time for the control program.

NOTATION FOR DEFINING CONTROL STATEMENTS

The notation used to define control statements in this publication is described in the following paragraphs.

1. The set of symbols listed below are used to define control statements, but are never written in an actual statement.

a.	hyphen	-
b.	or	|
c.	underscore	_
d.	braces	{ }
e.	brackets	[ ]
f.	ellipsis	...
g.	superscript	¹

   The special uses of these symbols are explained in paragraphs 4-10.

2. Uppercase letters and words, numbers, and the set of symbols listed below are written in an actual control statement exactly as shown in the statement definition. (Any exceptions to this rule are noted in the definition of a control statement.)

a.	apostrophe	'
b.	asterisk	*
c.	comma	,
d.	equal sign	=
e.	parentheses	( )
f.	period	.
g.	slash	/

3. Lowercase letters, words, and symbols appearing in a control statement definition represent variables for which specific information is substituted in the actual statement.

   Example: If name appears in a statement definition, a specific value (e.g., ALPHA) is substituted for the variable in the actual statement.

4. Hyphens join lowercase letters, words, and symbols to form a single variable.

   Example: If member-name appears in a statement definition, a specific value (e.g., BETA) is substituted for the variable in the actual statement.

5. Stacked items or items separated from each other by the "or" symbol represent alternatives. Only one such alternative should be selected.

   Example: The two representations

   $\left\{\begin{matrix}A\\B\\C\end{matrix}\right\}$ and A|B|C

   have the same meaning and indicate that either A or B or C should be selected.

6. An underscore indicates a default option. If an underscored alternative is selected, it need not be written in the actual statement.

   Example: The two representations

   $\left\{\begin{matrix}A\\\underline{B}\\C\end{matrix}\right\}$ and A|<u>B</u>|C

   have the same meaning and indicate that either A or B or C should be selected; however, if B is selected, it need not be written, because it is the default option.

7. Braces group related items, such as alternatives.

   Example: ALPHA={A|B|<u>C</u>},D)

   indicates that a choice should be made among the items enclosed within the braces. If A is selected, the result is ALPHA=(A,D). If C is selected, the result can be either ALPHA=(,D) or ALPHA=(C,D).

8. Brackets also group related items; however, everything within the brackets is optional and may be omitted.

   Example: ALPHA=([A|B|C],D)

   indicates that a choice can be made among the items enclosed within the brackets, or that the items within the brackets can be omitted. If B is selected, the result is ALPHA=(B,D). If no choice is made, the result is ALPHA=(,D).

9. An ellipsis indicates that the preceding item or group of items can be repeated more than once in succession.

   Example: ALPHA[,BETA]...

   indicates that ALPHA can appear alone or can be followed by ,BETA repeated optionally any number of times in succession.

10. A superscript refers to a prose description in a footnote.

    Example: $\left\{\begin{matrix}NEW^{1}\\OLD\\MOD\end{matrix}\right\}$

    indicates that additional information concerning the grouped items is contained in footnote number 1.

11. Blanks are used to improve the readability of control statement definitions. Unless otherwise noted, blanks have no meaning in a statement definition.

JOB STATEMENT

The JOB statement (Figure 5) is the first statement in the sequence of control statements that describe a job. The JOB statement contains the following information:

1. Job name.
2. Accounting information relative to the job.
3. Programmer's name.
4. Whether the job-control statements are printed for the programmer.
5. Conditions for terminating the execution of the job.

Examples of the JOB statement are shown in Figure 6.

NAME	OPERATION	OPERAND
		Positional Parameters
//jobname	JOB	[([account-number][,accounting-information])[1] [2] [3] [,programmer-name][4] [5] [6]
		Keyword Parameters
		{MSGLEVEL=0 / MSGLEVEL=1}
		[COND=((code,operator)[,(code,operator)]...[7])[8]]

[1] If the information specified ("account-number" and/or "accounting-information") contains blanks, parentheses, or equal signs, the information must be delimited by apostrophes instead of parentheses.
[2] If only "account-number" is specified, the delimiting parentheses may be omitted.
[3] The maximum number of characters allowed between the delimiting parentheses or apostrophes is 144.
[4] If "programmer-name" contains commas, parentheses, apostrophes, or blanks, it must be enclosed within apostrophes.
[5] When an apostrophe is contained within "programmer-name", the apostrophe must be shown as two consecutive apostrophes.
[6] The maximum number of characters allowed for "programmer-name" is 20.
[7] The maximum number of repetitions allowed is 7.
[8] If only one test is specified, the outer pair of parentheses may be omitted.

Figure 5. JOB Statement

```
                Example 1
//PROGRAM JOB (215,819,46W),'E.COBOL',  1
//            COND=(7,LT),MSGLEVEL=1


                Example 2
//PROG2 JOB 1087F-21,COND=(7,LT)
```

Figure 6. Sample JOB Statements

NAME FIELD

The "jobname" must always be specified; it identifies the job to the operating system.

OPERAND FIELD

Account Number and Accounting Information

The first positional parameter can contain the installation account number and any parameters passed to the installation accounting routines. These routines are written by the installation and inserted in the operating system when it is generated. The format of the accounting information is specified by the installation.

Programmer's Name

The "programmer-name" is the second positional parameter.

Control Statement Messages

The MSGLEVEL parameter indicates the type of control statement messages the programmer wishes to receive from the control program.

MSGLEVEL=0

indicates that only control statement errors and diagnostic messages are written for the programmer.

MSGLEVEL=1

indicates that all control statements, as well as control statement errors and

diagnostic messages, are written for the programmer.

Conditions for Terminating a Job

At the completion of a job step that uses the COBOL compiler or the linkage editor, a code is issued indicating the outcome of the job step. Instructions in a COBOL load module cannot generate the code. The generated code is tested against the conditions stated in control statements. The error codes generated by the COBOL compiler or linkage editor are:

0 - No errors or warnings detected.

4 - Level W (warning) diagnostic. Possible errors were detected.

8 - Level C (conditional) diagnostic. Errors were detected.

12 - Level E (error) diagnostic. Serious errors were detected.

For a description of these codes, refer to Source-Module Error Warning Diagnostics.

The COND parameter specifies conditions under which a job is terminated. Up to eight different tests, each consisting of a code and an operator, may be specified to the right of the equal sign. The code may be any number between 0 and 4095. The operator indicates the mathematical relationship between the code placed in the JOB statement and the codes issued by completed job steps. If the relationship is true, the job is terminated. The six operators and their meanings are:

Operator	Meaning
GT	greater than
GE	greater than or equal to
EQ	equal to
NE	not equal to
LT	less than
LE	less than or equal to

For example, if a code 8 is returned by the compiler and the JOB statement contains:

```
COND=(7,LT)
```

the job is terminated.

If more than one condition is indicated in the COND parameter and any of the conditions are satisfied, the job is terminated.

EXEC STATEMENT

The EXEC statement (Figure 7) indicates the beginning of a job step and describes that job step. The statement contains the following information.

1. Name of the cataloged procedure or load module to be executed.
2. Compiler and/or linkage editor options passed to the job step.
3. Accounting information relative to this job step.
4. Conditions for bypassing the execution of the subsequent job step.

NAME	OPERATION	OPERAND
		Positional Parameter
//[stepname][1]	EXEC	{PROC=cataloged-procedure-name cataloged-procedure-name PGM=program-name PGM=*.stepname.ddname PGM=*.stepname.procstep.ddname}
		Keyword Parameters
		[{PARM PARM.procstep[2]}=(option[,option]...)[3] [4] [5]]
		[{ACCT ACCT.procstep[2]}=(accounting-information)[3] [6] [7]]
		[{COND COND.procstep[2]}=((code,operator[,stepname[.procstep]]) [,(code,operator[,stepname[.procstep]])]...[8])[9]]

[1] "stepname" is required when information from this control statement is referenced in a later job step.
[2] If this format is selected, it may be repeated in the EXEC statement once for each step in the cataloged procedure.
[3] If the information specified contains blanks, parentheses, or equal signs, it must be delimited by apostrophes instead of parentheses.
[4] If only one option is specified, and it does not contain any blanks, parentheses, or equal signs, the delimiting parentheses may be omitted.
[5] The maximum number of characters allowed between the delimiting apostrophes or parentheses is 40.
[6] If "accounting-information" does not contain commas, blanks, parentheses, or equal signs, the delimiting parentheses may be omitted.
[7] The maximum number of characters allowed between the delimiting apostrophes or parentheses is 144.
[8] The maximum number of repetitions allowed is 7.
[9] If only one test is specified, the outer pair of parentheses may be omitted.

Figure 7. EXEC Statement

```
                    Example 1

// EXEC PGM=IEHPROGM,ACCT=(896,427),COND=(7,LT)

                    Example 2

//STEP4 EXEC  COBECLG,                                                    1
//              PARM.COB='DECK,LINECNT=64,MAPS,LIST',                     2
//              PARM.LKED=XREF,                                           3
//              COND.LKED=(7,GT,STEP4.COB),                               4
//              COND.GO=((7,GT,STEP4.LKED),(7,GT,STEP4.COB)),             5
//              ACCT=108LA
```

Figure 8. Sample EXEC Statements

Example 1 of Figure 8 shows the EXEC statement used to execute a program. Example 2 in Figure 8 shows an EXEC statement that invokes a cataloged procedure.

NAME FIELD

The "stepname" is the name of the job step.

OPERAND FIELD

Positional Parameter

The options in the positional parameter of an EXEC statement specify either the name of the cataloged procedure or program to be executed.

Each program (load module) to be executed must be a member of a PDS.

Specifying a Cataloged Procedure:

```
{PROC=cataloged-procedure-name}
{cataloged-procedure-name      }
```

indicate that a cataloged procedure is invoked. The "cataloged procedure name" is the unqualified name of the cataloged procedure. For example,

```
PROC=COBEC
```

indicates that the cataloged procedure COBEC is to be executed.

Specifying a Program in a Library:

```
PGM=program-name
```

indicates that a program is executed. The "program-name" is an unqualified member name of a load module in the sytem library (SYS1.LINKLIB) or private library. For example,

```
PGM=IEWL
```

indicates that the load module IEWL is executed. (A load module in a private PDS is executed by joining the private library with the system library through the use of a JOBLIB DD statement. See the following discussion concerning JOBLIB.)

Specifying a Program Described in a Previous Job Step:

```
PGM=*.stepname.ddname
```

indicates that a program is executed, but the program is taken from a data set specified in a DD statement of a previous job step. The * indicates the current job; "stepname" is the name of a previous step within the current job; and "ddname" is the name of a DD statement within that previous job step. (The "stepname" cannot refer to a job step in another job.) For example, in the statements,

```
//LXIX JOB ,JOHNSMITH,COND=(7,LT)
              .
              .
              .
//STEP4 EXEC PGM=IEWL
//SYSLMOD DD DSNAME=OBJECT(TEST1)
              .
              .
              .
//STEP5 EXEC PGM=*.STEP4.SYSLMOD
```

statement STEP5 indicates that the name of the program is taken from the DD statement SYSLMOD in job step STEP4. Consequently, the load module TEST1 in the PDS OBJECT is executed.

Specifying a Program Described in a Cataloged Procedure:

```
PGM=*.stepname.procstep.ddname
```

indicates that a program is executed, but the program is taken from the data set specified in a DD statement of a previously executed cataloged procedure. The * indicates the current job; "stepname" is the name of the job step that invoked the cataloged procedure; "procstep" is the name of a step within the procedure; "ddname" is the name of a DD statement within the procedure step. (The "stepname" cannot refer to a job step in another job.) For example, consider a cataloged procedure PROG1.

```
//COMPIL   EXEC PGM=IEPCBL00
//SYSUT1   DD   UNIT=TAPE
//SYSPUNCH DD   DSNAME=LINKINP
            .
            .
            .
//LKED     EXEC PGM=IEWL
//SYSLMOD  DD   DSNAME=RESULT(ANS)
            .
            .
            .
```

Furthermore, assume the following statements are placed in the input stream.

```
//GO        JOB  ,SMITH,COND=(7,LT)
//S1        EXEC PROC=PROG1
            .
            .
            .
//S2        EXEC PGM=*.S1.LKED.SYSLMOD
            .
            .
            .
```

The statement S2 in the input stream indicates that the name of the program is taken from the DD statement SYSLMOD in the procedure step LKED in the procedure PROG1 which was invoked by the EXEC statement S1. Consequently, the load module ANS in the PDS RESULT is executed.

Keyword Parameters

The keyword parameters may refer to a program, to an entire cataloged procedure, or to a step within a cataloged procedure.

If the parameter refers to a program, to the first step in a cataloged procedure (only with the PARM parameter), or to an entire cataloged procedure, the keyword is written followed by an equal sign and the list of subparameters. (In example 1, Figure 8, the parameter ACCT applies to the entire procedure.) When overriding parameters in a cataloged procedure step, the keyword is written, a period is placed after the keyword, and the stepname follows immediately. (In example 2, Figure 8, the cataloged procedure COBECLG is invoked. Two sets of PARM options apply to two different procedure steps; one applies to the procedure step COB and the other to the procedure step LKED.) More information about overriding cataloged procedures is given in the section, Job Processing.

Options for the Compiler and Linkage Editor: The PARM parameter is used to pass options to the compiler or linkage editor. (PARM has no meaning to COBOL load module.)

PARM

passes options to the compiler or linkage editor when either is invoked by the PGM parameter in the EXEC statement or to the first step in the cataloged procedure and cancels all other parameters specified in the cataloged procedure.

PARM.procstep

passes options to a compiler or linkage editor step within the named cataloged procedure step. Any PARM parameter in the procedure step is deleted, and the PARM parameter that is passed to the procedure step is inserted.

A maximum of 40 characters may be written between the parentheses or apostrophes that enclose the list of options.

The format for compiler options and linkage options most applicable to the COBOL programmer is shown in Figure 9.

Detailed information concerning compiler and linkage editor options is given in the section, Job Processing.

```
Compiler:

  {PARM          }=([LINECNT=nn][,BUFSIZE=nn][,DECK  ][,FLAGE ]
  {PARM.procstep }  [,NODECK]  [,FLAGW]

                    [,LIST  ][,DMAP  ][,PMAP  ][,MAPS  ][,DISPCK  ][,REGED  ])[1]
                    [,NOLIST][,NODMAP][,NOPMAP][,NOMAPS][,NODISPCK][,INVED ]

Linkage Editor

  {PARM          }=([XREF][,LET][,LIST])[1]
  {PARM.procstep }

[1] The subparameters (options) are keyword subparameters.
```

Figure 9. Compiler and Linkage Editor Options

Condition for Bypassing a Job Step: This COND parameter (unlike the one in the JOB statement) determines if the job step defined by the EXEC statement is bypassed.

COND

states conditions for bypassing the execution of a program or an entire cataloged procedure. If the EXEC statement invokes a cataloged procedure, the COND parameter replaces all COND parameters in each step of the procedure.

COND.procstep

states conditions for bypassing the execution of a specific cataloged procedure step "procstep". The specified COND parameter replaces all COND parameters in the procedure step.

The subparameters for the COND parameter are of the form:

(code,operator[,stepname])

The subparameters "code" and "operator" are the same as the code and operator described for the COND parameter in the JOB statement. The subparameter "stepname" identifies the previous job step that issued the code. For example, the COND parameter

COND=((5,LT,COBE),(5,LT,LKED))

Indicates that the step in which the COND parameter appears is bypassed if 5 is less than the code returned by either of the steps COBE or LKED.

If a step in a cataloged procedure issued the code, "stepname" must qualify the name of the procedure step; that is,

(code,operator[,stepname.procstep])

If "stepname" is not given, "code" is compared to all codes issued by previous job steps. Again, only compiler or linkage editor execution steps issue the code.

Accounting Information: The ACCT parameter specifies accounting information for a job step within a job.

ACCT

is used to pass accounting information to the installation accounting routines for this job step.

ACCT.procstep

is used to pass accounting information for a step within a cataloged procedure.

If both the JOB and EXEC statements contain accounting information, the installation accounting routines decide how the accounting information shall be used for the job step.

DATA DEFINITION (DD) STATEMENT

The DD statement (Figure 10) describes data sets. The DD statement can contain the following information:

1. Name of the data set to be processed.
2. Type and number of I/O devices for the data set.
3. Volume(s) on which the data set resides.
4. Amount and type of space allocated on a direct-access volume.
5. Label information for the data set.
6. Disposition of the data set before and after execution of the job step.
7. Allocation of data sets with regard to channel optimization.

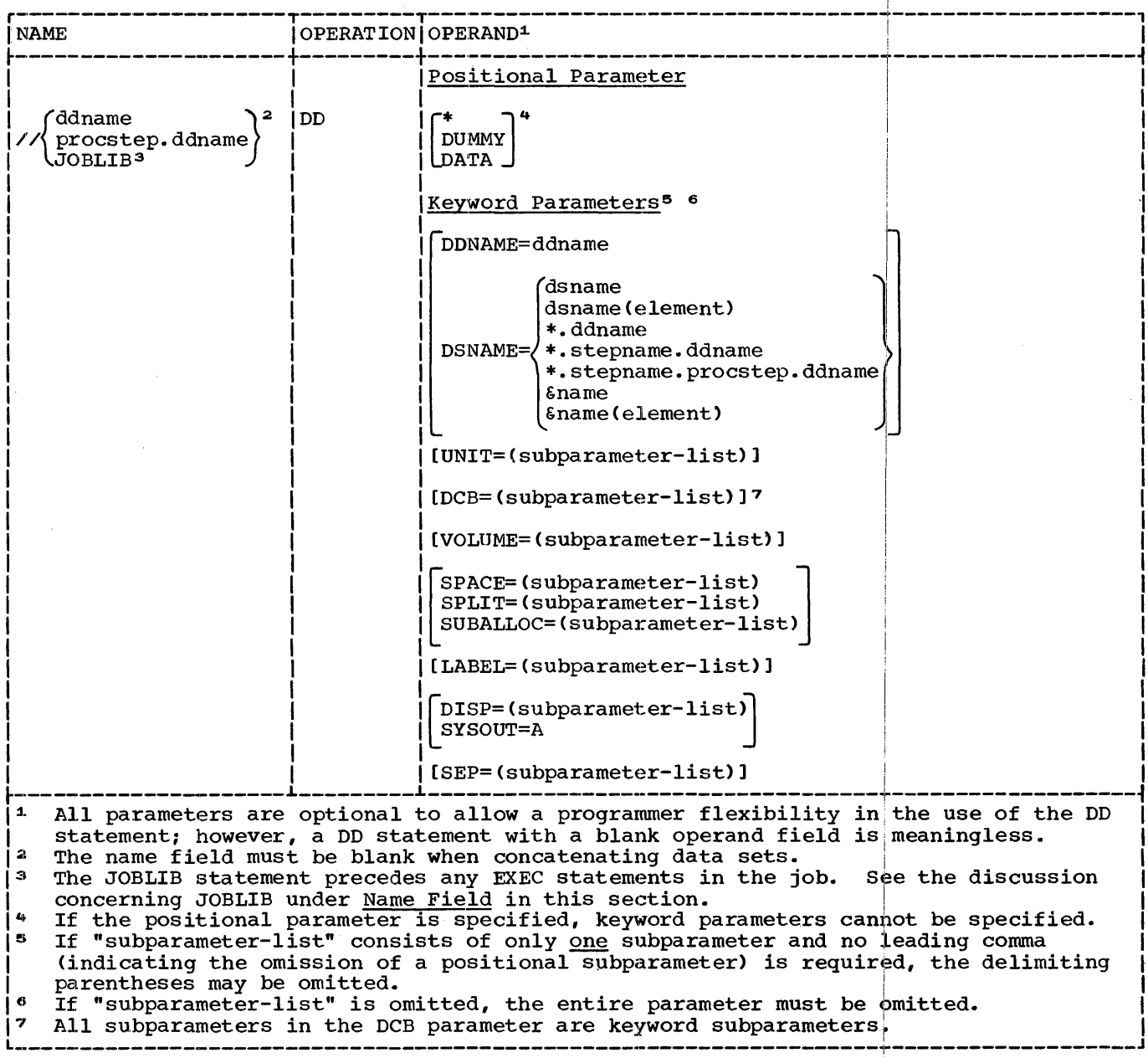


Figure 10. Data Definition Statement

NAME FIELD

ddname

is used:

1. To identify data sets defined by this DD statement to the compiler or linkage editor.
2. To relate files defined by a programmer in his source module to data set definitions in the DD statement. The ddname must be the same as the external name in the SELECT... ASSIGN clause in a COBOL program.
3. To identify this DD statement to other control statements in the input stream.

The "ddname" format is given in Job Processing.

procstep.ddname

is used to override DD statements in cataloged procedures. The step in the cataloged procedure is identified by "procstep". The "ddname" identifies either:

1. A DD statement in the cataloged procedure that is to be modified by the DD statement in the input stream, or

2. A DD statement that is to be added to the DD statement in the procedure step.

JOBLIB

is used to concatenate data sets with the operating system library; that is, the operating system library and the data sets specified in the JOBLIB DD statement are temporarily combined to form one library. The JOBLIB statement must immediately follow a JOB statement and the concatenation is in effect only for the duration of the job. However, if job steps other than the first job step are to use the data set specified in the JOBLIB DD statement, the DISP parameter must be specified with PASS as the second subparameter. (See the following text concerning the DISP parameter.) Only one JOBLIB statement may be specified for a job.

The "PGM=program name" parameter in the EXEC statement refers to a load module in the system library. However, if this parameter refers to a load module in a private library, a JOBLIB statement identifying the PDS in which the module resides must be specified for the job. The JOBLIB statement concatenates the private library with the system library.

The library indicated in the JOBLIB statement is searched for a module before the system library is searched.

A JOBLIB statement does not have to be entered for load modules created in this job, or for permanent members of the system library.

If the name field is omitted, the data set defined by the DD statement is concatenated with the data set defined in the preceding DD statement. In effect, these two data sets are combined into one data set. (Data sets may also be concatenated with the data set specified in the JOBLIB DD statement. Therefore, several data sets can be concatenated with the system library.)

OPERAND FIELD

For purposes of discussion, parameters for the DD statement are divided into six classes. Parameters are used to:

- Specify unit record data sets.
- Retrieve a previously created and cataloged data set.
- Retrieve a data set created in a previous job step in the current job and passed to the current job step.
- Retrieve a data set created but not cataloged in a previous job.
- Create data sets that reside on magnetic tape or direct access volumes.
- Optimize I/O operations.

The following text describes the DD statement parameters that apply to processing unit record data sets and retrieving data sets created in previous job steps or data sets created and cataloged in previous jobs (Figure 11). The method of retrieving uncataloged data sets created in previous jobs is also discussed in this section. Parameters shown in Figure 10 and not mentioned in this section are used to create data sets and optimize I/O operations in job steps.

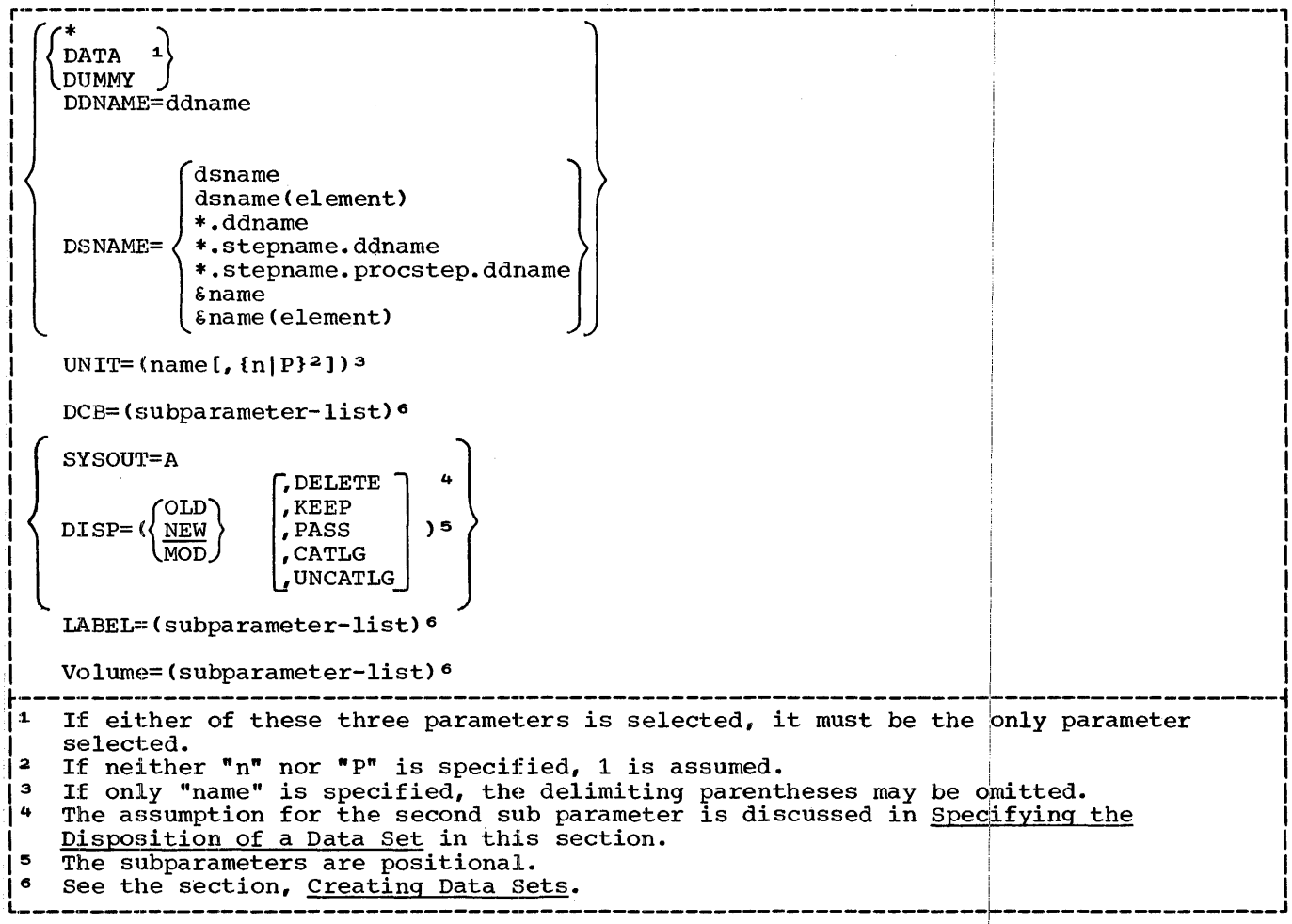


Figure 11. DD Statement Operands

```
          Example 1: Printer

//SYSPRINT DD SYSOUT=A

          Example 2: Card Punch

//SYSPUNCH DD UNIT=SYSCP

          Example 3: Card Reader

//SYSIN DD *
```

Figure 12. Unit Record Examples of DD Statements

Unit Record Parameters

The UNIT and SYSOUT parameters are used for unit record data sets; the * or DATA parameters designate that the data set for this job step follows in the input stream. Examples of DD statements for unit record data sets are shown in Figure 12.

Specifying Data in the Input Stream

*

indicates that a data set immediately follows this DD statement in the input stream. This parameter is used to specify a source deck or data in the input stream. If the EXEC statement for the job step invokes a cataloged procedure, a data set may be placed in the input stream for each

procedure step. If the EXEC statement specifies execution of a program, only one data set may be placed in the input stream. The DD * statement must be the last DD statement for the procedure step or program. The end of the data set must be indicated by a delimiter statement. The data cannot contain // in the first two characters of the record.

DATA

also indicates data in the input stream. The restrictions and use of the DATA parameter are the same as the *, except that // may appear in the first and second positions in the record.

UNIT Parameter:

UNIT=(name[,{n|P}])

specifies an input/output device, a type of device, or class of devices for a data set. When the system is generated, the "name" is assigned by the operating system or the installation. The programmer can use only the assigned names in his DD statements. For example,

UNIT=190, UNIT=2311, UNIT=TAPE

where 190 is a device address, 2311 is a device type, and TAPE is a device class.

[n|P]

specifies the number of devices allocated to the data set. If a number "n" is specified, the operating system assigns that number of devices to the data set. Parallel, "P", is used with cataloged data sets. The control program assigns as many devices as there are volumes indicated in the index and label fields of the cataloged data set.

SYSOUT Parameter: A SYSOUT parameter may be specified for printer data sets.

SYSOUT=A

indicates the device class A for the data set. The data set defined by the DD statement that contains the SYSOUT parameter is written on a device chosen by the operator. No parameter other than the DCB parameter has any meaning when the SYSOUT parameter is used.

Retrieving Previously Created Data Sets

If a data set on a magnetic tape or a direct-access volume is created and cataloged in a previous job or job step, all information for the data set such as device, volume, space, etc., is stored in the catalog and labels. This information need not be repeated in other DD statements. To retrieve the data set, the name (DSNAME) and disposition (DISP) of the data set must be specified.

If the data set was created in a previous job step in the current job, the information in the previous DD statement is available to the control program, and is accessible by referring to the previous DD statement. To retrieve the data set, a pointer to a data set created in a previous job step is specified by the DSNAME parameter. The disposition (DISP) of the data set is also specified.

If the data set was created in a previous job but not cataloged, information concerning the data set, such as space, record format, etc., is stored in the labels. The volume and device information is not stored. To retrieve the data set, the name (DSNAME), disposition (DISP), label (LABEL), volume (VOLUME), and device (UNIT) must be specified. The VOLUME and LABEL parameters are discussed in the section, Creating Data Sets.

Examples of the use of DD statements to retrieve previously created data sets are shown in Figure 13.

```
          Example 1: Retrieving a Cataloged Data Set

 //CBL01   DD DSNAME=EXP(WKLY),DISP=(OLD,PASS)

          Example 2: Retrieving a Data Set Created in a Previous Step

 //CBL05   DD DSNAME=*.STEP4.CBL01,DISP=(MOD,KEEP)

          Example 3: Retrieving an Uncataloged Data Set Created in a Previous Job

 //CBL09   DD DSNAME=DATA.SIM,DISP=OLD,UNIT=180,VOLUME=SER=Z1
```

Figure 13. Retrieving Previously Created Data Sets

IDENTIFYING A CREATED DATA SET: The DSNAME parameter indicates the name of a data set or refers to a data set defined in the current or a previous job step.

Specifying a Cataloged Data Set by Name:

DSNAME=dsname

the fully qualified name of the data set is indicated by "dsname". If the data set was previously created and cataloged, the control program uses the catalog to find the data set and instructs the operator to mount the required volumes.

Specifying a Generation Data Group or PDS:

DSNAME=dsname(element)

indicates either a generation data set contained in a generation data group or a member of a partitioned data set. The name of the generation data group or partitioned data set is indicated by "dsname"; if "element" is either 0 or a signed integer, a generation data set is indicated. For example,

DSNAME=ACCNT(-2)

indicates the thirdmost recent member of the generation data group ACCNT. If "element" is a name, a member of a partitioned data set is indicated.

Referring to a Data Set in the Current Job Step:

DSNAME=*.ddname

indicates a data set that is defined previously in a DD statement in this job step. The * indicates the current job. The name of the data set is copied from the DSNAME parameter in the DD statement named "ddname".

Referring to a Data Set in a Previous Job Step:

DSNAME=*.stepname.ddname

indicates a data set that is defined in a DD statement in a previous job step in this job. The * indicates the current job, and "stepname" is the name of a previous job step. The name of the data set is copied from the DSNAME parameter in the DD statement named "ddname". For example, in the control statements:

```
//SAMPLE JOB
//JOBLIB DD DSNAME=CALC,DISP=(OLD,PASS)
//S1 EXEC PGM=INVNTRY
//COBL01 DD DSNAME=OUT(+1)
//COBL02 DD DSNAME=CURNT,DISP=OLD
//S2 EXEC PGM=UPDATE
//COBL05 DD DSNAME=*.S1.COBL01
//COBL07 DD *
          .
          .
          .
```

The DD statement COBL05 in job step S2 indicates the data set (OUT) is defined in the DD statement COBL01 in job step S1.

Referring to a Data Set in a Cataloged Procedure:

DSNAME=*.stepname.procstep.ddname

indicates a data set that is defined in a cataloged procedure invoked by a previous job step in this job. The * indicates the current job; "stepname" is the name of a previous job step; "procstep" is the name of a step in the cataloged procedure; and "ddname" is the name of the DD statement defining the data set.

Assigning Names to Temporary Data Sets:

DSNAME=&name
  assigns a name to a temporary data set.

The control program assigns the data set a unique name which exists only until the end of the current job. The data set may be accessed in following job steps by &name. This option is useful in passing an object module from a compiler job step to a linkage editor job step.

DSNAME=&name(element)
assigns a name to a member of a temporary PDS. The name is assigned in the same manner as the DSNAME=&name. This option is useful in storing object modules that will be linkage edited in a later job step in the current job.

SPECIFYING THE DISPOSITION OF A DATA SET:
The DISP parameter is specified for both previously created data sets and data sets being created in this job step.

```
DISP=({NEW  }[,DELETE ])
      {OLD  } [,KEEP   ]
      {MOD  } [,PASS   ]
              [,CATLG  ]
              [,UNCATLG]
```

is used for all data sets residing on magnetic tape or direct access volumes.

The first subparameter indicates when the data set is (was) created.

NEW
indicates that the data set is created in this step. NEW is discussed in more detail in the section, Creating Data Sets.

OLD
indicates that the data set was created by a previous job or job step.

MOD
indicates that the data set was created in a previous job or job step, but records are to be added to the data set. Before the first I/O operation for the data set occurs, the data set is positioned following the last record. If a data set specified as MOD does not exist, the specification is assumed to be NEW.

The second subparameter indicates the disposition of the data set.

DELETE
causes the space occupied by the data set to be released and made available at the end of the current job step. If the data set was cataloged and the catalog was used to retrieve it, it is removed from the catalog.

KEEP
ensures that the data set is kept intact until a DELETE option is specified in a subsequent job or job step. KEEP is used to retain uncataloged data sets for processing in future jobs. Keep does not imply PASS.

PASS
indicates that the data set is referred to in a later job step. When a subsequent reference to the data set is made, its PASS status lapses unless another PASS is issued. The final disposition of the data set should be stated in the last job step that uses the data set. When a data set is in PASS status, the volume(s) on which it is mounted is retained. If dismounting is necessary, the control program issues a message to mount the volume(s) when needed. PASS is used to pass data sets among job steps in the same job.

CATLG
causes the creation of a catalog entry that points to the data set. The data set can then be referred to in subsequent jobs or job steps by name (CATLG implies KEEP).

UNCATLG
causes the data set to be removed from the catalog at the end of the job step.

If the second subparameter is not specified, no action is taken to alter the status of the data set. If the data set was created in this job (NEW), it is deleted at the end of the current job step. If the data set existed before this job (MOD or OLD), it exists after the end of the job.

DELIMITER STATEMENT

The delimiter statement (Figure 14) is used to separate data from subsequent control statements in the input stream, and is placed after each data set in the input stream.

The delimiter statement contains a slash in column 1, an asterisk in column 2, and a blank in column 3. The remainder of the card may contain comments.

NAME	OPERATION OPERAND
/*	

Figure 14. Delimiter Statement

Three steps are required to execute a COBOL program: compiling, linkage editing, and executing.

For each of the three steps involved in processing, ddnames and device names are specified by the operating system. These ddnames, options for the compiler and linkage editor, and specifying additional libraries for the linkage editor are discussed in this section.

The output of a single COBOL compilation is an object module made up of one control section. The name of the control section is derived from the PROGRAM-ID statement in the COBOL source program. A control section is a unit of coding (instructions and data) that is, in itself, an entity. All elements of a control section are loaded and executed in a constant relationship to each other. A control section is, therefore, the smallest separately relocatable unit of a program.

USING CATALOGED PROCEDURES

Because writing job-control statements can become time-consuming work for the programmer, IBM supplies three cataloged procedures to aid in the compiling, linkage editing, and executing of COBOL-E programs. Each procedure requires that a

```
//procstep.SYSIN DD
```

statement be provided in the input stream, indicating the location of a source module or object module to the control program. The job-control statements needed to invoke the procedures, and deck structures used with the procedures, are described in the following text.

COMPILE

COBEC is the cataloged procedure for compilation. It is invoked by specifying the name COBEC as the first parameter in an EXEC statement.

(The cataloged procedure, COBEC, consists of the control statements shown in Figure 27 in Cataloged Procedures.)

With the procedure COBEC, a DD statement COB.SYSIN indicating the location of the source module must be supplied in the input stream. Figure 15 shows control statements that can be used to invoke the procedure.

```
//jobname JOB
// EXEC COBEC
//COB.SYSIN DD *
    COBOL Source Module
/*
```

Figure 15. Invoking the Cataloged Procedure COBEC

A sample deck structure to compile a source module is shown in Figure 16.

```
//JOBC JOB 00,COBOLPROG,MSGLEVEL=1
//EXECC EXEC PROC=COBEC
//COB.SYSIN DD *
    COBOL Source Module
/*
```

Figure 16. Compiling a Source Module

The SYSIN data set containing the source module is defined as data in the input stream for the compiler. Note that a delimiter statement follows the last COBOL statement.

LINKAGE EDIT AND EXECUTE

COBELG is the cataloged procedure to linkage edit COBOL object modules and execute the resulting load module. It is invoked by specifying the name COBELG as the first parameter in an EXEC statement.

(The cataloged procedure to linkage edit and execute consists of the control statements shown in Figure 28 in Cataloged Procedures.)

With the procedure COBELG, a DD statement LKED.SYSIN, which indicates the location of the object module, must be supplied.

Figure 17 shows control statements that can be used to invoke the COBELG cataloged procedure.

```
//jobname JOB
// EXEC COBELG
//LKED.SYSIN DD *
    COBOL Object Module
/*
```

Figure 17. Invoking the Cataloged Procedure COBELG

A sample deck structure to linkage edit and execute, as one load module, several object modules entered in the input stream is shown in Figure 18.

```
//JOBBLG JOB 00,ECOBOL,MSGLEVEL=1
//EXECLG EXEC PROC=COBELG
//LKED.SYSIN DD *
    First COBOL Object Module
                .
                .
                .
    Last COBOL Object Module
/*
```

Figure 18. Linkage Edit and Execute

The object module decks were created by the DECK compiler option. The linkage editor recognizes the end of one module and the beginning of another, and resolves references between them.

Figure 19 shows a sample deck structure to linkage edit object modules that are within a cataloged sequential data set, OBJMODS, and subsequently execute the program.

```
//JOBBLG JOB 00,ECOBOL,MSGLEVEL=1
//EXECLG EXEC COBELG
//LKED.SYSIN DD DSNAME=OBJMODS,DISP=OLD
//GO.SYSIN DD *
            Data
/*
```

Figure 19. Linkage Edit and Execute (Object Modules in a Cataloged Data Set)

COMPILE, LINKAGE EDIT, AND EXECUTE

The third cataloged procedure, COBECLG, passes a source module through three procedure steps: compile, linkage edit, and execute. The cataloged procedure is invoked by specifying the name COBECLG as the first parameter in an EXEC statement.

(Figure 29 in Cataloged Procedures shows the statements that make up the cataloged procedure, COBECLG.)

The SYSIN data set (source module) must be defined to the compiler. Figure 20 shows statements that can be used to invoke the procedure, COBECLG.

```
//jobname JOB
// EXEC PROC=COBECLG
//COB.SYSIN DD *
    COBOL Source Module
/*
```

Figure 20. Invoking the Cataloged Procedure, COBECLG

Figure 21 shows a sample deck structure to compile, linkage edit, and execute a source module.

```
//JOBCLG JOB 00,ECOBOL,MSGLEVEL=1
//EXECC EXEC COBECLG
//COB.SYSIN DD *
    COBOL Source Module
/*
```

Figure 21. Compile, Linkage Edit, and Execute

COMPILER PROCESSING

The names for DD statements (ddnames) relate I/O statements in the compiler with data sets used by the compiler. These ddnames must be used for the compiler. When the system is generated, names for I/O device classes are also established and must be used by the programmer.

Compiler Name

The program name for the compiler is IEPCBL00. If the compiler is to be executed without using the supplied cataloged procedures in a job step, the EXEC statement parameter

PGM=IEPCBL00

must be used.

Compiler ddnames

The compiler can use up to eight data sets. To establish communication between the compiler and the programmer, each data set is assigned a specific ddname. Each data set has a specific function and device requirement. Table 2 lists the ddnames, functions, and device requirements for the data sets.

Table 2. Compiler ddnames

ddname	FUNCTION	DEVICE REQUIREMENTS
SYSIN	reading the source program	• card reader • intermediate storage
SYSPRINT	writing the storage map, listings, and messages	• printer • intermediate storage
SYSPUNCH	punching the object module deck, or creating an object module data set as input to the linkage editor	• card punch • direct-access • magnetic tape
SYSUT1	work data set needed by the compiler during compilation	• direct-access • magnetic tape
SYSUT2	work data set needed by the compiler during compilation	• direct-access • magnetic tape
SYSUT3	work data set needed by the compiler during compilation	• direct-access • magnetic tape
SYSUT4	optional work data set needed when using debug packet(s)	• direct-access • magnetic tape
SYSLIB	optional user source program library	• direct-access

To compile a COBOL source module, five of these data sets are necessary: SYSIN, SYSPRINT, SYSUT1, SYSUT2, and SYSUT3, along with the direct-access volume(s) that contains the operating system. With these five data sets, only a listing is generated by the compiler. If an object module is to be punched or written on a direct-access or magnetic tape volume, a SYSPUNCH DD statement must be supplied. If the debug packet(s) is to be used, a SYSUT4 DD statement must be supplied. If the compiler is to COPY or INCLUDE a source-language module from the user's source program library, a SYSLIB DD statement must be supplied.

For the DD statement SYSIN or SYSPRINT, an intermediate storage device may be specified instead of the card reader or

printer. The intermediate storage device usually is magnetic tape, but can be a direct-access device.

If an intermediate device is specified for SYSIN, the compiler assumes that the source module deck was placed on intermediate storage by a previous job or job step. If an intermediate device is specified for SYSPRINT, the maps, listing, and error/warning messages are written on that device; a new job or job step can print the contents of the data set.

Compiler Device Classes

Names for input/output device classes used for compilation are also specified by the operating system when the system is generated. The class names, functions, and types of devices are shown in Table 3.

Table 3. Device Class Names

CLASS NAME	CLASS FUNCTIONS	DEVICE TYPE
SYSSQ	writing, reading	• direct-access • magnetic tape
SYSDA	writing, reading	• direct-access
SYSCP	punching cards	• card punch
A	SYSOUT output	• printer • magnetic tape

The data sets used by the compiler must be assigned to the device classes listed in Table 4.

Table 4. Correspondence Between Compiler ddnames and Device Classes

ddname	POSSIBLE DEVICE CLASSES
SYSIN	SYSSQ, or the input stream device (specified by DD * or DD DATA)
SYSPRINT	A,SYSSQ,SYSDA
SYSPUNCH	SYSCP,SYSSQ,SYSDA
SYSUT1	SYSSQ,SYSDA
SYSUT2	SYSSQ,SYSDA
SYSUT3	SYSSQ,SYSDA
SYSUT4	SYSSQ,SYSDA
SYSLIB	SYSDA

```
{PARM                }
{PARM.procstep       } =([LINECNT=nn][,BUFSIZE=nn]
                        [,DECK  ] [,FLAGE ] [,LIST  ]
                        [,NODECK] [,FLAGW ] [,NOLIST]
                        [,DMAP  ] [,PMAP  ] [,MAPS  ]
                        [,NODMAP] [,NOPMAP] [,NOMAPS]
                        [,DISPCK] [,REGED ]
                        [,NODISPCK] [,INVED] )
```

Figure 22. Compiler Options

Compiler Options

Options (Figure 22) may be passed to the compiler through the PARM parameter in the EXEC statement. The following information may be specified:

1. The number of lines to be printed per page on the compiler output listing.
2. The size of each of the six work buffers used during a compilation.
3. Whether an object module is created.
4. The type of diagnostic messages to be generated by the compiler.
5. Whether a list of the source statements is printed.
6. Whether a list of data-name addresses is generated.

7. Whether a list of object code is generated.

8. Whether a list of both data-name addresses and object code is generated.

9. Whether the compiler will generate object code to test length of fields to be DISPLAYed.

10. The type of editing performed in the PICTURE clause and numeric literals.

There is no specified order for compiler options in the PARM parameter.

LINECNT=nn: The LINECNT option indicates the number of lines to be printed on each page of the compilation output listing. The programmer specifies a number nn, where nn is a 2-digit integer in the range of 10 to 99. If the option is not specified, the number of lines per page will be that specified when the system was generated.

BUFSIZE=nn: The BUFSIZE option indicates the size of each of the six work buffers used during a COBOL compilation. The BUFSIZE parameter should not be used on a 32K system. The following formula can be used to determine the maximum value to be used for this parameter.

$$S = \frac{C - 30000 - [(13 + L)(N)]}{6}$$

where: S is the size of each work buffer
C is the total main storage
L is the length of the average data name
N is the number of data names.

If the work buffers are for disk, the maximum value of S is 3625. If the work buffers are for tape, the maximum value of S is 32670. If the option is not specified, the buffer size will be that specified when the system was generated.

DECK or NODECK: The DECK option specifies that the compiled source module (i.e., the object module) is written on the data set specified by the SYSPUNCH DD statement. NODECK specifies that no object module is written. A description of the deck is given in the section, System Output. If neither option is specified, an object module is produced.

FLAGE or FLAGW: The FLAGE option specifies that the compiler will suppress warning diagnostic messages. The FLAGW option specifies that the compiler will generate diagnostic messages for actual errors in the source module, plus warning diagnostic messages for possible errors. Diagnostic messages are written on the data set specified by the SYSPRINT DD statement. If neither option is specified, the class of diagnostic message produced is that specified when the system was generated.

LIST or NOLIST: The LIST option specifies that the source listing is written on the data set specified by the SYSPRINT DD statement. The NOLIST option indicates that no source listing is written. A description of the source listing is given in the section, System Output. If neither option is specified, a source listing is produced.

DMAP or NODMAP: The DMAP option specifies that the compiler will generate a listing of the DATA DIVISION data-names and their addresses relative to the load point of the object module. The listing is written on the data set specified by the SYSPRINT DD statement. The NODMAP option specifies that a data-name listing will not be generated. If neither DMAP nor NODMAP is specified, the option taken will be that specified when the system was generated.

PMAP or NOPMAP: The PMAP option specifies that the compiler will generate a listing of object code for each statement in the PROCEDURE DIVISION. The listing is written on the data set specified by the SYSPRINT DD statement. The NOPMAP option specifies that a listing of object code will not be generated. If neither PMAP nor NOPMAP is specified, the option taken will be that specified when the system was generated.

MAPS or NOMAPS: The MAPS option is equivalent to specifying both DMAP and PMAP. The NOMAPS option is equivalent to specifying both NODMAP and NOPMAP.

DISPCK or NODISPCK: The DISPCK option specifies that the compiler will generate object code that will test, at execution time, to determine if a field to be DISPLAYed exceeds the record length of the device on which it is to be written. The NODISPCK option specifies that no such code will be generated. If neither DISPCK nor NODISPCK is specified, the option taken will be that specified when the system was generated.

REGED or INVED: The REGED option specifies that the character"." represents a decimal point and the character"," represents an insertion character. The INVED option specifies that the above rolls of these characters ".", "," be reversed

LINKAGE EDITOR PROCESSING

The linkage editor processes COBOL object modules, COBOL subroutines, resolves any references to subprograms, and constructs a load module. To communicate with the linkage editor, the programmer supplies an EXEC statement and DD statements that define all required data sets; he may also supply linkage editor control statements.

LINKAGE EDITOR NAME

The program name for the linkage editor is IEWL. If the linkage editor is executed without using cataloged procedures in a job step, the EXEC statement parameter

PGM=IEWL

must be used.

LINKAGE EDITOR INPUT AND OUTPUT

There are two types of input to the linkage editor: primary and secondary.

Primary input is a sequential data set that contains object modules and linkage editor control statements. Any external references among object modules in the primary input are resolved by the linkage editor as the primary input is processed. Furthermore, the primary input contains references to the secondary input. These references are linkage editor control statements and/or COBOL external references in the object modules.

Secondary input resolves references and is separated into two types: automatic call library and additional input specified by the programmer. The automatic call library must always be the COBOL library (SYS1.COBLIB), which is the PDS that contains the COBOL object time subroutines. Through the use of DD statements, the automatic call library can be concatenated with other partitioned data sets. Three types of additional input may be specified by the programmer:

1. An object module used as the main program in the load module being constructed. This object module, which can be accompanied by linkage editor control statements, is either a member of a PDS or is a sequential data set. The first record in the primary input data set must be a linkage editor INCLUDE control statement that tells the linkage editor to process the main program.

2. An object module used to resolve external references made in another module. The object module, which can be accompanied by linkage editor control statements, is a sequential data set. An INCLUDE statement that defines the data set must be given.

3. A module used to resolve external references made in another module. The load module which can be accompanied by linkage editor control statements, is a member of a PDS. The module can be included from the call library.

In addition, the secondary input can contain external references and linkage editor control statements. If a load module is not in the automatic call library, the linkage editor LIBRARY statement can be used to direct the linkage editor to reference additional libraries during the automatic library call process.

The output load module of the linkage editor is always placed in a PDS as a named member. The name can be provided in the SYSLMOD DD statement for the linkage editor execution. For the execution of the load module, this name can be used. Error messages and optional diagnostic messages are written on an intermediate storage device or a printer. Also, a work data set on a direct-access device is required by the linkage editor to do its processing. Figure 23 shows the I/O flow in linkage editor processing.

LINKAGE EDITOR DDNAMES AND DEVICE CLASSES

The programmer communicates data set information to the linkage editor through DD statements identified by specific ddnames (similar to the ddnames used by the compiler). The ddnames, functions, and requirements for data sets are shown in Table 5.

Any data sets specified by SYSLIB or SYSLMOD must be partitioned data sets. (Additional inputs are partitioned data sets or sequential data sets.) The ddname for the DD statement that defines any additional libraries or sequential data sets is written in INCLUDE and LIBRARY statements and is not fixed by the linkage editor.

The device classes used by the compiler (see Table 3) are also used with the linkage editor. The data sets used by the linkage editor may be assigned to the device classes listed in Table 6.

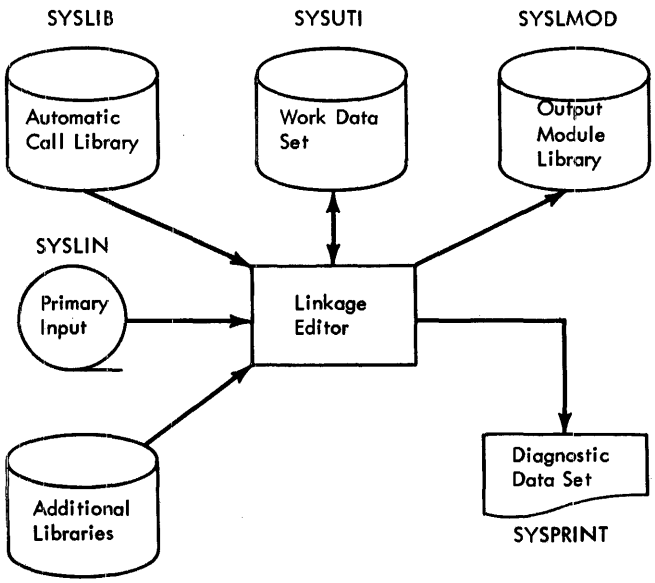


Figure 23. Linkage Editor Input and Output

Table 5. Linkage Editor ddnames

ddname	FUNCTION	DEVICE REQUIREMENTS
SYSLIN	primary input data, normally the output of the compiler	• direct access • magnetic tape • card reader
SYSLIB	automatic call library (SYS1.COBLIB)	• direct access
SYSUT1	work data set	• direct access
SYSPRINT	diagnostic messages	• printer • intermediate storage device
SYSLMOD	output data set for the load module	• direct access
user-specified	additional libraries and object modules	• direct access • magnetic tape

Table 6. Correspondence Between Linkage Editor ddnames and Device Classes

ddname	POSSIBLE DEVICE CLASSES
SYSLIN	SYSSQ, SYSDA, or the input stream device (specified by DD * or DD DATA)
SYSLIB	SYSDA
SYSUT1	SYSDA
SYSLMOD	SYSDA
SYSPRINT	A,SYSSQ
user-specified	SYSDA,SYSSQ

ADDITIONAL INPUT

The INCLUDE and LIBRARY statements are used to specify additional secondary input to the linkage editor. Modules neither specified by INCLUDE or LIBRARY statements nor contained in the primary input are retrieved from the automatic call library.

INCLUDE Statement

Operation	Operand
INCLUDE	ddname[(member-name [,member-name]...)] [,ddname[(member-name [,member-name]...)]]...

The INCLUDE statement is used to include either members of additional libraries (PDS) or sequential data sets. The "ddname" specified a DD statement that defines either a PDS containing object modules and control statements or just load modules, or defines a sequential data set containing object modules. The "member name" is the name of a member of a PDS and is not used when a sequential data set is specified.

The linkage editor processes the object module or load module when the INCLUDE statement is encountered.

LIBRARY Statement

Operation	Operand
LIBRARY	ddname(member-name [,member-name]...) [,ddname (member-name [,member-name]...)]...

The LIBRARY statement is used to include members of additional libraries during the automatic library call process. The "ddname" must be the name of a DD statement that specifies a PDS that contains either object modules and linkage editor control statements, or just load modules. The "member name" is an external reference that is unresolved after primary input processing is complete.

The LIBRARY statement differs from the INCLUDE statement in that external references specified in the LIBRARY statement are not resolved until all other processing, except references reserved for the automatic call library, is completed by the linkage editor. (INCLUDE statements resolve external references when the INCLUDE statement is encountered.)

Example: Two subprograms, SUB1 and SUB2, and a main program, MAIN, are compiled by separate job steps. In addition to the COBOL library, a private library, MYLIB, is used to resolve external references to the symbols X, Y, and Z. Each of the object modules is placed in a sequential data set by the compiler, and passed to the linkage editor job step.

Figure 24 shows the control statements for this job. (Note: Cataloged procedures are not used in this job.) In this job, an additional library, MYLIB, is specified by the LIBRARY statement and the ADDLIB DD statement. SUB1 and SUB2 are included in the load module because SYSLIN input is the &GOFILE data set containing the MAIN, SUB1, and SUB2 object modules. The MOD parameter of DISP in STEP2 and STEP3 cause the SUB1 and SUB2 object modules to be added to the sequential data set containing the MAIN object module. The linkage editor input stream, SYSLIN, is two concatenated data sets. The first data set is the sequential data set &GOFILE, which contains the MAIN, SUB1, and SUB2 programs. The second data set is the input stream containing the LIBRARY statement. After linkage editor execution, the load module is placed in the PDS PROGLIB and given the name CALC, as specified on the SYSLMOD DD statement for STEP4.

```
//JOBX      JOB
//STEP1     EXEC    PGM=IEPCBL00
                      .
                      .
                      .
//SYSPUNCH  DD      DSNAME=&GOFILE,DISP=(MOD),UNIT=SYSSQ
//SYSIN     DD      *
            Source module for MAIN
/*
//STEP2     EXEC    PGM=IEPCBL00
                      .
                      .
                      .
//SYSPUNCH  DD      DSNAME=&GOFILE,DISP=(MOD),UNIT=SYSSQ
//SYSIN     DD      *
            Source module for SUB1
/*
//STEP3     EXEC    PGM=IEPCBL00
                      .
                      .
                      .
//SYSPUNCH  DD      DSNAME=&GOFILE,DISP=(MOD),UNIT=SYSSQ
//SYSIN     DD      *
            Source module for SUB2
/*
//STEP4     EXEC    PGM=IEWL
                      .
                      .
                      .
//SYSLIB    DD      DSNAME=SYS1.COBLIB,DISP=OLD
//SYSLMOD   DD      DSNAME=PROGLIB(CALC),UNIT=SYSDA
//ADDLIB    DD      DSNAME=MYLIB,DISP=OLD
//SYSLIN    DD      DSNAME=&GOFILE,DISP=OLD
//          DD      *
            LIBRARY ADDLIB(X,Y,Z)
/*
```

Figure 24. Linkage Editor Example

LINKAGE EDITOR PRIORITY

If control sections with the same name appear in the input to linkage editor, the linkage editor inserts only one of the control sections. The following priority for control sections is established by the linkage editor:

1. Control sections appearing in SYSLIN or control sections appearing in modules identified by INCLUDE statements in SYSLIN.
2. Control sections in modules identified by the LIBRARY statement.
3. Control sections in modules appearing in SYSLIB.

If control sections with the same name appear in a single data set, only the module encountered first is inserted in the output load module.

OTHER LINKAGE EDITOR CONTROL STATEMENTS

In addition to the LIBRARY and INCLUDE statements, other control statements are available for use with the linkage editor. These statements enable the user to: specify additional names for load modules (ALIAS), replace control sections within a load module (REPLACE), and change control section names and subprogram entry point names (CHANGE). Also, two statements (OVERLAY and INSERT) enable the programmer to overlay load modules. For a detailed description of these control statements, see the publication, IBM System/360 Operating System: Linkage Editor.

ENTRY Statement

The ENTRY statement specifies the first instruction to be executed.

Operation	Operand
ENTRY	external name

External name is defined as a control section name or an entry name in a linkage editor input module. It must be the name of an instruction, not of data. In an overlay program, the external name must be defined as the name of an instruction in the root segment.

OPTIONS FOR LINKAGE EDITOR PROCESSING

The linkage editor options are specified in an EXEC statement. The options that are most applicable to the COBOL programmer are:

$$\left\{\begin{matrix}\text{PARM} \\ \text{PARM.procstep}\end{matrix}\right\}=([\text{XREF}][,\text{LET}][,\text{LIST}])$$

XREF: The XREF option informs linkage editor to produce a map of the load module; this map indicates the relative location and length of main programs and subprograms. Also, a cross-reference list indicating all external references in each main program and subprogram is generated. Descriptions of the map and cross-reference listing are given in System Output.

LET: The LET option informs linkage editor to mark the load module ready for execution even though error conditions were found.

LIST: The LIST option indicates that linkage editor control statements are listed in card-image format on the diagnostic output data set.

Other options can also be specified for the linkage editor. For a detailed description of all linkage editor options, see the publication, IBM System/360 Operating System: Linkage Editor.

LOAD MODULE EXECUTION

When the system is generated, device names are assigned by the operating system and the installation. The programmer chooses devices by specifying either the installation or operating system names.

Program Name

When "PGM=program name" is used to indicate the execution of a load module, the module must be in either the system library (SYS1.LINKLIB) or a private library. When the module is in a private library, a JOBLIB DD statement must be supplied to indicate the name of the private library. For example, assume that the load modules FICA, FITX, and SITX are in the PDS PAYROLL. These load modules are executed as follows:

```
//JOBPAY JOB 00,ECOBOL
//JOBLIB DD DSNAME=PAYROLL,DISP=(OLD,PASS)
//STEP1 EXEC PGM-FICA
          .
          .
          .
//STEP2 EXEC PGM=FITX
          .
          .
          .
//STEP3 EXEC PGM=SITX
          .
          .
          .
```

The JOBLIB DD statement concatenates the private library PAYROLL with the system library.

Execution ddnames

In the source module, data sets are identified by the external names specified in the environment division SELECT and ASSGN clause. These names must correspond to the ddname for the associated DD statement at execution time.

Execution Error Messages

When an error condition recognized by compiler-generated code arises during execution, an error message is written on the console typewriter. These messages, with accompanying format and descriptions, are shown in Appendix F.

SYSABEND Data Set

During execution of a load module, there are various conditions that may arise to cause the abnormal termination of the execution. At this point, the programmer could utilize an object-program

main-storage dump for debugging purposes. This is the function of a SYSABEND data set.

When a SYSABEND data set is specified:

```
//SYSABEND DD SYSOUT=A
```

in the execution job step, the system provides an object-program main-storage dump on the SYSOUT device whenever the job step is abnormally terminated.

Execution Device Classes

For load module execution, the programmer can use the same names assigned to device classes used by the compiler (shown in Table 3). However, additional names for specific devices and device classes can be assigned by the installation. The programmer can choose which device to use for his data sets, and specify the name of the device or class of devices in the UNIT parameter of the DD statement.

DCB Parameter

The DCB parameter may be specified for data sets when a load module is executed. For information concerning the DCB parameter, see Creating Data Sets.

Scratching Disk Data Sets

The cataloged procedures supplied by IBM inform the operating system to scratch, at the normal end of a job, the utility data sets and the temporary data sets that are passed from one job step to another. This means that both the disk area and data set names are reusable for the next job to be processed.

If a job terminates abnormally, a dump of main storage is provided by the operating system, after which the data sets are scratched. However, there may be instances where a job is abnormally terminated without the system's providing a dump of main storage, or an instance when the programmer or operator manually interrupts the complete generation of a dump by the system. In these cases, the data sets are not scratched automatically by the operating system; they must be scratched by a system utility program, IEHPROGM. For a description of this utility program, see the publication IBM System/360 Operating System: Utilities. An example of scratching a data set is shown in Appendix A of this publication.

In some cases, the dsname is that which is provided in the DD statement. However, some utility data sets do not have external dsnames assigned (such as SYSUT1, SYSUT2, etc.) in the cataloged procedures. In these cases, the operating system assigns an internal temporary dsname in the format

```
AAAAAAAA.AAAAAAAA.AAAAAAAA.AAAAAAAA.
nnnnnnnn
```

where n denotes a digit from 0 through 9.

To obtain the internally assigned dsnames, the system utility program IEHLIST must be executed. This utility program provides a listing of the Volume Table of Contents (VTOC) on the disk pack. All internal dsnames will appear on the VTOC listing and will be in the preceding format. These internal dsnames can then be specified to the scratch utility program IEHPROGM.

The following example shows the control statements required to execute the IEHLIST utility program.

```
//LIST JOB 123,DOE,MSGLEVEL=1
// EXEC PGM=IEHLIST
//SYSPRINT DD SYSOUT=A
//DD1 DD UNIT=2311,DISP=OLD
//DD2 DD UNIT=2311,DISP=OLD,                    X
//             VOLUME=SER=222222
//SYSIN DD *
             LISTVTOC VOL=2311=222222
/*
```

The //SYSPRINT statement specifies the device on which the listing will be created. The //DD1 statement specifies the system residence volume. The //DD2 statement specifies a mountable volume. The LISTVTOC statement specifies the specific device from which the VTOC is to be listed. If the VOL operand is omitted, the system residence volume is assumed.

The following example shows the control statements required to execute the IEHPROGM utility program, which scratches the data set.

```
//SCR JOB ,SCRATCH,MSGLEVEL=1
//STP EXEC PGM=IEHPROGM
//SYSPRINT DD SYSOUT=A
//DD1 DD UNIT=2311,DISP=OLD
//DD2 DD UNIT=2311,DISP=OLD,                X
//             VOLUME=SER=222222
//SYSIN DD *
           SCRATCH DSNAME=+.+.+.+.≠,      1
           VOL=2311=222222
               .
               .
               .
           SCRATCH DSNAME=LOADSET,        1
           VOL=2311=222222
           SCRATCH DSNAME=GODATA.RUN,     1
           VOL=2311=222222
/*
```

NOTE

The entry for the DSNAME of the SCRATCH statement must be one continuous line of code. Therefore, for each + appearing in the statement just illustrated, substitute AAAAAAAA (four are required); for ≠, substitute 00000001.

The last SCRATCH statement (GODATA.RUN) assumes that the COBECLG catalog procedure was used, and the job name given in the job card was RUN.

Data sets may be created in either of two ways:

1. By writing a COBOL source program and executing it with the proper DD statements.

2. By using a data set utility program. (The publication, IBM System/360 Operating System: Utilities, discusses data set utility programs.)

This section discusses the use of the DD statement.

To create data sets, the DSNAME, UNIT, VOLUME, SPACE, LABEL, DISP, SYSOUT, and DCB parameters are of special significance (see Figure 25). These parameters specify:

DSNAME - name of the data set

UNIT - class of devices used for the data set

VOLUME - volume on which the data set resides

LABEL - Label specification

DISP - the disposition of the data set before and after the job step

SYSOUT - ultimate device for unit record data sets

DCB - tape density, record format, record length, etc.

Examples of DD statements used to create data sets are shown in Figure 26.

DATA SET NAME

DUMMY

is specified in the DD statement to inhibit write operations specified for the data set. The write statement is recognized, but no data is transmitted. (When the programmer specifies DUMMY in a DD statement used to override a cataloged procedure, all parameters in the cataloged DD statement are overridden.) The programmer should not specify DUMMY for a data set that is to be read: an end of data set condition results, and the execution of the load module is terminated. Because dummy is a positional parameter, no keyword parameters may be specified with it.

The DSNAME parameter specifies the name of the data set. Only four forms of the DSNAME parameter are used to create data sets.

```
DSNAME=dsname
DSNAME=dsname(element)
```

dsname

specifies the fully qualified name of a data set. This is the name under which it can be cataloged or tabulated.

dsname(element)

specifies a particular generation of a generation data group, a member of a partitioned data set, or an area of an indexed sequential data set. To indicate a generation of a generation data group, the element is a zero or a signed integer. To indicate a member of a partitioned data set, the element is a name. To indicate an area of an indexed sequential data set, the element is PRIME, OVFLOW, or INDEX. The significance of the elements for indexed sequential data sets is described under Allocating Space for Indexed Sequential Data Sets.

```
DSNAME=&name
DSNAME=&name(element)
```

specify data sets that are temporarily created for the execution of a single job or job step.

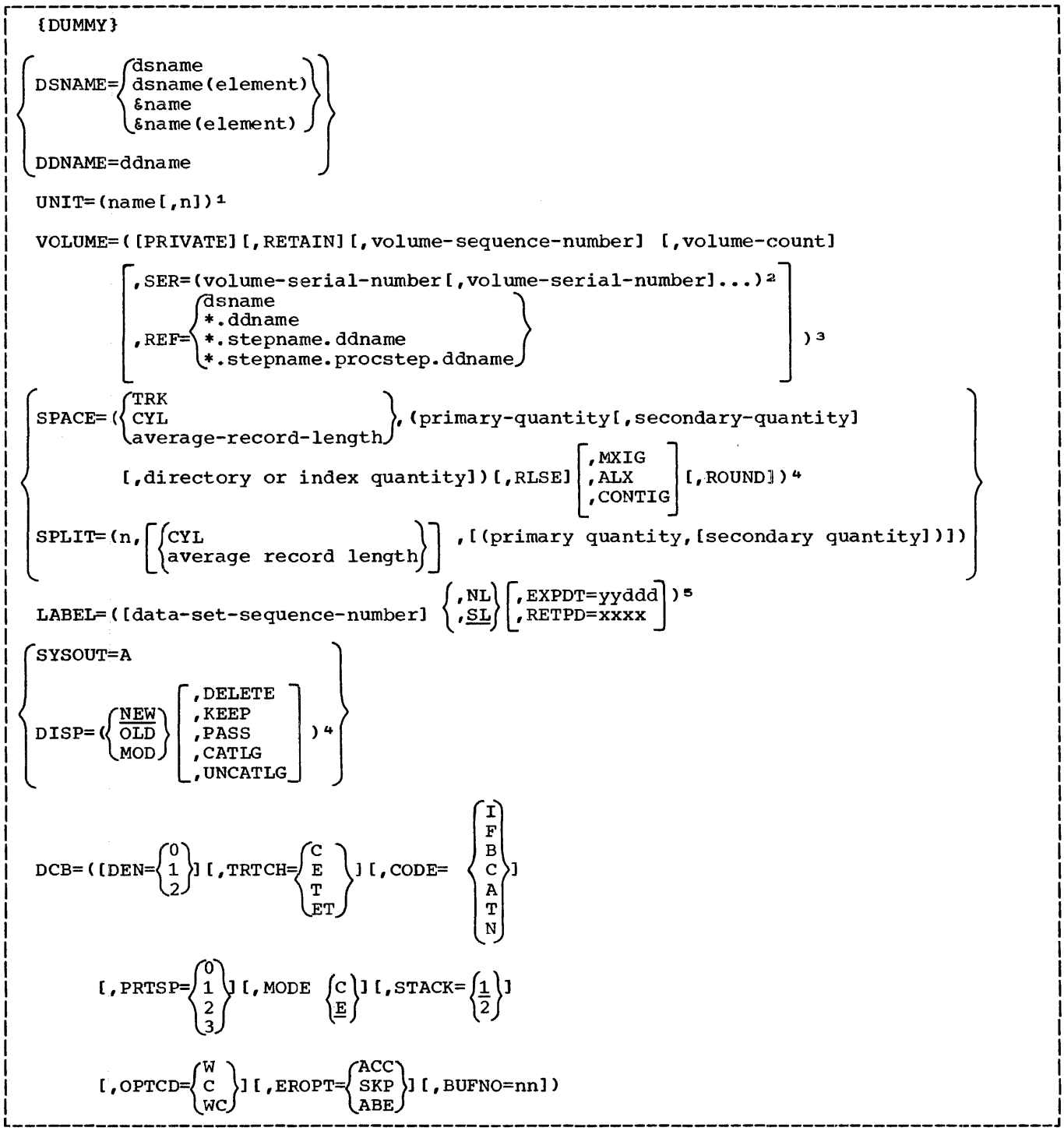


Figure 25. DD Parameters for Creating Data Sets (Part 1 of 2)

[1] If only "name" is specified, the delimiting parentheses may be omitted.

[2] If only one "volume-serial-number" is specified, the delimiting parentheses may be omitted.

[3] SER and REF are keyword subparameters; the remaining subparameters are positional subparameters.

[4] All subparameters are positional subparameters.

[5] EXPDT and RETPD are keyword subparameters; the remaining subparameters are positional subparameters.

[6] All subparameters are keyword subparameters.

Figure 25. DD Parameters for Creating Data Sets (Part 2 of 2)

```
   Example 1:  Creating a Cataloged Data Set

//CALC DD DSNAME=PROCESS,DISP=(NEW,CATLG),LABEL=(,SL,EXPDT=66031),    1
//             UNIT=DACLASS,VOLUME=(PRIVATE,RETAIN,SER=AA69),         2
//             SPACE=(300(100,100),,CONTIG,ROUND)

   Example 2:  Creating a Data Set for a Job

//SYSUT1 DD DSNAME=&TEMP,UNIT=(TAPECLS,3),DISP=(NEW,PASS),           1
//             VOLUME=(,RETAIN,1,9,SER=(777,888,999)),               2
//             DCB=(DEN=2)

   Example 3:  Specifying a SYSOUT Date Set

//SYSPRINT DD SYSOUT=A

   Example 4:  Creating a Data Set that Is Kept, but Not Cataloged

//TEMPFILE DD DSNAME=FILE,DISP=(,KEEP),                              1
//             DCB=(DEN=2)

   Example 5:  Creating a Data Set on a 7-Track Tape

//TEMPFILE DD DSNAME=FILE,DISP=(OLD,KEEP),                           1
//             VOLUME=(PRIVATE,,,SER=222,333),                       2
//             DCB=(DEN=1,TRTCH=ET),UNIT=(2400,2)
```

Figure 26. Examples of DD Statements

DDNAME=ddname

indicates a DUMMY data set that will assume the characteristics specified in a following DD statement "ddname". The DD statement identified by "ddname" then loses its identity; that is, it cannot be referred to by an *....ddname parameter. The statement in which the DDNAME parameter appears may be referenced by subsequent *....ddname parameters. If a subsequent statement identified by "ddname" does not appear, the data set defined by the DD statement containing the DDNAME parameter is assumed to be an unused statement. The DDNAME parameter can be used five times in any given job step or procedure step, and no two uses can refer to the same "ddname". The DDNAME parameter is used mainly for cataloged procedures.

SPECIFYING I/O DEVICES

The name of an input/output device or class of devices and the number of devices are specified in the UNIT parameter.

```
UNIT=(name[,n])
```

name

is the name assigned to the input/output device classes when the system is generated, or an absolute device address.

[,n]

specifies the number-of devices allocated to the data set. If this parameter is omitted, 1 is assumed.

SPECIFYING VOLUMES

The programmer indicates the volumes used for the data set in the VOLUME parameter.

```
VOLUME=([PRIVATE][,RETAIN]
        [,volume-sequence-number]
        [,volume-count]
   [,SER=(volume-serial-number
       [,volume-serial-number]...)
    ,REF={dsname
          *.ddname
          *.stepname.ddname
          *.stepname.procstep.ddname}]  )
```

identifies the volume(s) assigned to the data set.

PRIVATE

is used only for direct-access volumes. This option indicates that the assigned volume is to contain only the data set defined by this DD statement. PRIVATE is overridden when the DD statement for a data set requests the use of the private volume with the SER or REF subparameter. Volumes other than direct-access volumes are always considered PRIVATE.

RETAIN

indicates that this volume is to remain mounted after the job step is completed. Volumes are retained so that data may be transmitted to or from the data set, or so that other data sets may reside in the volume. If the data set requires more than one volume, only the last volume is retained; the other volumes are previously dismounted. Another job step indicates when to dismount the volume by omitting RETAIN. If each job step issues a RETAIN for the volume, the retained status lapses when execution of the job is completed.

volume-sequence-number

is a one-to-four digit number that specifies the sequence number of the first volume of the data set that is read or written. The volume sequence number is meaningful only if the data set is cataloged and earlier volumes omitted.

volume-count

specifies the number of volumes required by the data set. Unless the SER or REF subparameter is used this subparameter is required for every multi-volume output data set.

SER

specifies one or more serial numbers for the volumes required by the data sets. A volume serial number consists of one to six alphameric characters. If it contains less than six characters, the serial number is left adjusted and padded with blanks. If SER is not specified, and DISP is not specified as NEW, the data set is assumed to be cataloged and serial numbers are retrieved from the catalog. A volume serial number is not required for output data sets.

REF

indicates that the data set is to occupy the same volume(s) as the data set identified by "dsname", "*.ddname", "*.stepname.ddname", or *.stepname.procstep.ddname. Table 7 shows the data set references.

Table 7. Data Set References

OPTION	REFERS TO
REF=dsname	a data set named "dsname"
REF=*.ddname	a data set indicated by DD statement "ddname" in the current job step
REF=*.stepname.ddname	a data set indicated by DD statement "ddname" in the job step "stepname"
REF=*.stepname.procstep.ddname	a data set indicated by DD statement "ddname" in the procedure step "procstep" invoked in the job step "stepname"

When the data set resides on a tape volume and REF is specified, the data set is placed on the same volume, immediately behind the data set referred to by this subparameter. When this subparameter is used, the UNIT parameter may be omitted.

If SER or REF is not specified, the control program will allocate any nonprivate volume that is available.

SPECIFYING SPACE ON DIRECT-ACCESS VOLUMES

SPACE PARAMETER

```
SPACE=({TRK                          },
       {CYL                          }
       {average-record-length        }

        (primary-quantity

        [,secondary-quantity]

        [,directory or index quantity])

        [,RLSE] [,MXIG ] [,ROUND])
                [,ALX  ]
                [,CONTIG]
```

specifies space on a direct-access volume. Although SPACE has no meaning for tape volumes, if a data set is assigned to a device class that contains both direct-access devices and tape devices, SPACE should be specified.

Note: For indexed sequential data sets, only the CYL subparameter is permitted. Neither the TRK subparameter nor the average record length can be specified. When an indexed sequential data set is defined by more than one DD statement, all DD statements must contain a SPACE parameter. For the details on how to compute the space requirements of an Indexed Sequential Data Set, refer to the publication; IBM System/360 Operating System: Control Program Services.

The SPACE specifies:

1. Units of measurement in which space is allocated.
2. Amount of space allocated.
3. Whether unused space can be released.
4. In what format space is allocated.
5. Whether space is to begin on a cylinder boundary.

```
TRK
CYL
average-record-length
```

specifies the units of measurement in which storage is assigned. The units may be tracks (TRK), cylinders (CYL), or records (average record length expressed in decimal numbers).

```
(primary-quantity[,secondary-quantity]
        [,directory-quantity])
```

specifies the amount of space allocated for the data set.

The "primary quantity" indicates the number of records, tracks, or cylinders to be allocated when the job step begins. For indexed sequential data sets, this subparameter specifies the number of cylinders for the prime, overflow, or index area. For details of these parameters, refer to Allocating Space for Indexed Sequential Data Sets.

The "secondary quantity" indicates how much space is to be allocated each time previously allocated space is exhausted. This subparameter must not be specified when defining an indexed sequential data set.

The "directory quantity" is used only when writing a PDS, and it specifies the number of 256-byte records to reserve for the directory of the PDS. The "index quantity" specifies the number of cylinders to be allocated for an index area embedded within the prime area, when a new indexed sequential data set is being defined. For details of these parameters, refer to Allocating Space for Indexed Sequential Data Sets.

For example, in the DD statement:

```
//TEMPFILE DD  SPACE=(120,(400,100))
```

space is reserved for 400 records, the average record length is 120 characters. Each time space is exhausted, space for 100 additional records is allocated, for a maximum of fifteen times.

In the statement:

```
 //FICAFILE DD SPACE=(CYL,(20,2,5))
```

20 cylinders are allocated to the data set. When previously allocated space is exhausted, two additional cylinders are allocated. In addition, space is reserved for five records in the directory of a PDS. Each record can contain seven members.

RLSE

indicates that all unused external storage assigned to this data set is released when processing of the data set is completed.

```
[MXIG  ]
[ALX   ]
[CONTIG]
```

specify the format of the space allocated to the data set. MXIG requests the largest single block of storage that is greater than or equal to the space requested in the "primary quantity". ALX requests up to five contiguous blocks of storage, each block greater than the "primary quantity". CONTIG requests that the space indicated in the "primary quantity" be contiguous.

If the subparameter is not specified, or if any option cannot be fulfilled, the operating system attempts to assign contiguous space. If there is not enough contiguous space, up to five noncontiguous areas are allocated.

For indexed sequential data sets, RLSE, MXIG, ALX, or ROUND must not be specified; only CONTIG or blank (none of these subparameters) is permitted.

ROUND

indicates that allocation of space for the specified number of records is to begin and end on a cylinder boundary.

Note: The SPACE parameter in the DD statement must be used if a data set might be written on a direct-access device. For the compiler, the programmer should allow 150 characters per source statement in the "primary quantity" for each data set except SYSPRINT. For SYSPRINT, he should allow approximately 220 characters per source statement.

SPLIT PARAMETER

```
SPLIT=(n, [{CYL                    }])
          [{average record length}]
           [primary quantity,
           [secondary]])]
```

The split (SPLIT) parameter is specified when other data sets in the job step require space on the same volume, and the user wishes to minimize access arm movement by sharing cylinders with the other data sets. The device is then said to be operating in a split cylinder mode. In this mode, two or more data sets are stored so that portions of each occupy tracks within every allocated cylinder.

A group of data sets that share cylinders on the same device is defined by a sequence of DD statements. The first statement in the sequence must specify all parameters except "secondary quantity," which is optional. Each of the statements that follow must specify only n, the amount of space required.

n — Indicates the number of tracks per cylinder to be used for this data set if CYL is specified. If the average record length is specified, n is the percentage of the tracks per cylinder to be used for this data set.

CYL
average record length — Indicates the units in which the space requirements are expressed in the next subparameter. The units may be cylinders (CYL) or physical records (in which case the average record length in bytes is specified as a decimal number not exceeding 65,535). If the average record length is given, and the data set is defined to have a key, the key length must be given in the DCB parameter of this DD statement.

primary quantity — Defines the number of cylinders or space for records to be allocated to the entire group of data sets.

secondary quantity — Defines the number of cylinders or space for records to be allocated each time the space

allocated to any of the data sets in the group has been exhausted and more data is to be written. This quantity will not be split.

LABEL INFORMATION

If the programmer wishes to catalog a data set so that he can refer to it without repeating information that was supplied when the data set was created, he must specify certain information in the LABEL parameter. If the parameter is omitted and the data set is cataloged or passed, the label information is retrieved from data set labels stored with the data set.

```
LABEL=([data set sequence number]{,NL|,SL}
[,EXPDT=yyddd|,RETPD=xxxx])
```

data-set-sequence-number

is a 4-digit number that identifies the relative location of the data set with respect to the first data set on a tape volume. (For example, if there are three data sets on a magnetic tape volume, the third data set is identified by data set sequence number 3.) If the data set sequence number is not specified, the operating system assumes 1. (This option should not be confused with the volume sequence number, which represents a particular volume for a data set.)

{NL|SL}

specifies whether standard labels exist for a data set. SL indicates standard labels. NL indicates no labels.

[EXPDT=yyddd|RETPD=xxxx]

specifies how long the data set shall exist. The expiration date, EXPDT=yyddd, indicates the year (yy) and the day (ddd) the data set can be deleted. The period of retention, RETPD=xxxx, indicates the period of time, in days, that the data set is to be retained. If neither is specified, the retention period is assumed to be zero.

DISPOSITION OF A DATA SET

The disposition of a data set is specified by the DISP parameter; see Data Definition (DD) Statement. The same options are used for both creating data sets and using previously created data sets. When a data set is created, the subparameters used are NEW, KEEP, PASS, and CATLG.

WRITING A UNIT RECORD DATA SET ON THE PRINTER

A printed output data set may be written using the following parameter.

```
SYSOUT=A
```

DCB PARAMETER

For load module execution, the COBOL programmer may specify the details of a data set by using COBOL source statements and DD statement subparameters of the DCB parameter. The illustrations given in the following are examples of DCB subparameters for processing these file organizations:

- Sequential
- Indexed Sequential
- Direct or Relative

Sequentially organized data sets may reside on magnetic tape or direct-access volumes. Direct relative or indexed files must reside on direct-access volumes. Note that some DCB subparameter values (see Tables 10, 11, and 12) may be supplied by DD statements; other values are supplied either by certain COBOL source statements or by the COBOL compiler.

DCB FOR PROCESSING SEQUENTIAL DATA SET

```
DCB= ([DEN={0|1|2}]
  [,TRTCH={C|E|T|ET|U|UC}]
  [,PRTSP={0|1|2|3}]
  [,MODE={C|E}][,STACK={1|2}]
  [,OPTCD={W|C|WC}][ERROPT={ACC|SKP|ABE}]
  [,DSORG=PS][,MACRF=({GL|PL|GL,PL})
  [,DDNAME=symbol][,RECFM={F|U|V}
  [,LRECL=absexp][,BLKSIZE=absexp]
  [,BFTEK=S][,BUFNO=absexp]
  [,BFALN= F D ][,BUFL=absexp]
  [,BUFCB=relexp][,EODAD=relexp]
  [,SYNAD=relexp])
```

A description of the DCB subparameters follows.

DEN={0|1|2}

can be used with magnetic tape, and specifies a value for the tape recording density in bits per inch as listed in Table 8.

Table 8. DEN Values

DEN Value	TAPE RECORDING DENSITY (BITS/INCH)	
	Model 2400	
	7 Track	9 Track
0	200	-
1	556	-
2	800	800

TRTCH={C|E|T|ET|U|UC|}

is used as with 7-track tape to specify the tape recording technique, as follows:

C - specifies that the data conversion feature is to be used; if data conversion is not available, only format-F and -U records are supported by the control program.

E - specifies that even parity is to be used; if omitted, odd parity is assumed.

T - specifies that BCDIC to EBCDIC translation is required.

ET- specifies that even parity is to be used and BCDIC to EBCDIC translation is required.

U - unblock (permit) data checks on a printer with the Universal Character Set feature.

UC- unblock data checks on a printer and use chained scheduling.

PRTSP={0|1|2|3}

specifies the line spacing on a printer as 0, 1, 2, or 3.

MODE={C|E}

can be used with a card reader, a card punch, or a card read punch and specifies the mode of operation as follows:

C - the card image (column binary) mode

E - the EBCDIC code

If this information is not supplied by any source, E is assumed.

STACK={1|2}

can be used with a card reader, a card punch, or a card read punch and specifies which stacker bin is to receive the card. Either 1 or 2 is specified. If this information is not supplied by any source, 1 is assumed.

OPTCD={W|C|WC}

specifies an optional service to be performed by the control program, as follows.

W - perform a write validity check (on direct-access devices only).

C - process using the chained scheduling method.

WC- perform a validity check and use chained scheduling.

If this information is not supplied by any source, none of the services are provided.

EROPT={ACC|SKP|ABE}

specifies the option to be executed if an error occurs and either there is no synchronous exceptional error (SYNAD) exit routine or there is a SYNAD routine and the programmer wishes to return from it to his processing program. One of the following is specified:

ACC - accept error block

SKP - skip error block

ABE - terminate the task

Table 9 indicates the choices that are permitted for each type of data set processing.

Table 9. Error Options for QSAM

OPERAND	PROCESS DATA SET FOR	
	INPUT,RDBACK	OUTPUT
ACC	X	X[1]
SKP	X	
ABE	X	X

[1]Valid for printer only.

DSORG=PS

specifies the organization of the data set

as PS (a physical sequential organization).

MACRF=({GL|PL|GL,PL})

specifies the types of macro instructions that will be used in processing the data sets, where:

G indicates the GET macro instruction,
P indicates the PUT macro instruction, and
L indicates locate-mode operation

DDNAME=symbol

specifies the name of the DD statement that will be used to describe the data set to be processed.

RECFM={F|U|V}

specifies the characteristics of the records in the data set, where:

F - fixed-length records
U - undefined records
V - variable-length records

LRECL=absexp

specifies the length, in bytes, of a format-F logical record or the maximum length of a format-V logical record. This operand is omitted for format-U records, but must be supplied for format-F and -V records. The maximum value is 32,760.

BLKSIZE=absexp

specifies the maximum length, in bytes, of a block. For format-F records, the length must be an integral multiple of the LRECL value. For format-V records, the length must include the 4-byte block-length field that is recorded at the beginning of each block. The maximum value is 32,760.

When writing records on magnetic tape, the block size should be at least 18 bytes. Shorter blocks will be treated as noise records by the control-program error-recovery routines.

BFTEK=S

specifies the type of buffering to be supplied by the control program is S (simple buffering).

BUFNO=absexp

specifies the number of buffers to be assigned to the data control block. The maximum number is 255.

BFALN={F|D}

specifies the boundary alignment, in bytes, of each buffer, as follows:

F - the buffer starts on a full-word boundary (one that is not necessarily a double-word boundary).

D - the buffer starts on a double-word boundary.

BUFL=absexp

specifies the length in bytes of each buffer to be obtained for a buffer pool. The maximum value is 32.760. If this information is not supplied by any source, the control program calculates the length by using the value supplied for the BLKSIZE operand.

BUFCB=relexp

specifies the address of a buffer pool control block (i.e., the 8-byte field preceding the buffers in a buffer pool).

EODAD=relexp

specifies the address of the user's end-of-data set exit routine for input data sets. This routine is entered when the user requests a record and there are no more records to be retrieved. If no routine has been provided, the task is abnormally terminated.

SYNAD=relexp

specifies the address of the user's synchronous error exit routine. The routine is entered if input/output errors result from an attempt to process data records. If no routine is specified and an error occurs, the option specified by the EROPT parameter is executed.

Table 10 shows the values supplied for DCB subparameters by the COBOL compiler, by statements in the COBOL source program, and those subparameters that may be supplied by a DD statement for a sequential data set.

Table 10. DCB Subparameter Values For Sequential Data Set

DCB Parameter	Value Supplied Unconditionally by COBOL Compiler	Value Supplied by COBOL Source Statement	Value Supplied by DD statement
DEN			DEN={0|1|2}
TRTCH			TRTCH={C|E|T|ET|U|UC}
PRTSP			PRTSP={0|1|2|3}
MODE			MODE={C|E}
STACK			STACK={1|2}
OPTCD			OPTCD={W|C|WC}
EROPT			EROPT={ACC|SKP|ABE}
DSORG	PS		
MACRF	GL PL GL,PL	OPEN INPUT OPEN OUTPUT OPEN I-O	
DDNAME		External-name in ASSIGN clause	
RECFM		[1]RECORDING MODE BLOCK CONTAINS, ADVANCING clauses	
LRECL		RECORD CONTAINS clause[2]	
BLKSIZE		BLOCK CONTAINS clause[3]	
BFTEK	S		
BUFNO[4]		RESERVE clause	BUFNO=nn
BFLAN	D		
BUFL	0[5]		
BUFCB		SAME AREA clause	
EODAD		AT END clause	
SYNAD		USE statement option 5	

Notes:
[1] If RECORDING MODE is not specified in the source program, the compiler assumes a V format.
[2] The record length is calculated by the compiler.
[3] If this clause is omitted, the data set is considered to be unblocked.
[4] This parameter may be specified optionally from the DD statement or the COBOL RESERVE clause. If the RESERVE clause is specified, the DD statement BUFNO parameter is considered noise and does not override the number inserted by the compiler.
[5] When BUFL=0, the system makes the buffer size equal to the block size.

ALLOCATING SPACE FOR INDEXED SEQUENTIAL DATA SETS

Indexed sequential data sets consist of one, two, or three areas:

- Prime area. This area contains data records and the accompanying track indexes. It exists in all indexed sequential data sets.
- Overflow area. This area contains data records that overflow from tracks of the prime area when records are added to the data set. This area may or may not exist in an indexed sequential data set.
- Index area. This area contains the master and cylinder indexes for an indexed sequential data set. It exists for any data set that has a prime area on more than one cylinder.

The areas allocated and their locations depend on the parameters specified in the DD statement or statements that define the data set. For a description of the parameters and subparameters that can be used in DD statements defining a new indexed sequential data set or specifying an existing one, refer to the publication, IBM System/360 Operating System: Job Control Language.

DCB FOR CREATING INDEXED SEQUENTIAL DATA SETS

```
DCB=( [,OPTCD={WLI}] ,DSORG=IS
  [,MACRF=(PL)] [,DDNAME=symbol]
  [,RECFM={F|FB}] [,LRECL=absexp]
  [,BLKSIZE=absexp] [,RKP=absexp]
  [,KEYLEN=absexp]
  [,BUFNO=absexp]
  [,SYNAD=relexp])
```

OPTCD

OPTCD={WLI}

specifies an optional service to be performed by the program as follows:

W - a write validity check (on direct-access devices only)

L - delete option: user marks records for deletion; records so marked may actually be deleted when new records are added to the data set.

I - use independent overflow area.

DSORG=IS

specifies the organization of the data set as IS (an indexed sequential organization). This subparameter is required to be supplied by the programmer in the DD statement.

MACRF=(PL)

specifies the macro instruction that will be used in processing the data sets as follows:

PL - indicates that locate mode PUT macro instructions are to be used.

DDNAME=symbol

specifies the name of the DD statement that will be used to describe the data set to be processed.

RCFM={F|FB}

specifies the characteristics of the record in the data sets as follows:

F - fixed-length records

FB - fixed-length, blocked records

LRECL=absexp

specifies the length of a logical record in bytes.

BLKSIZE=absexp

specifies the maximum length of a block in bytes. For fixed-length records, the block must be an integral multiple of the LRECL value.

RKP=absexp

specifies the relative position of the first byte of the record key within each logical record. The value specified cannot exceed the logical record length minus the record key length.

KEYLEN=absexp

specifies the length of the record key, in bytes, associated with a logical record. The maximum length of the record key is 255 bytes.

BUFNO=absexp

specifies the number of buffers to be assigned to the data control block. The maximum number that can be specified is 255; however, the number must not exceed the limit on input/output requests established during system generation. This information can be supplied by the DD statement or the user's problem program.

SYNAD=relexp

specifies the address of the user's synchronous error exit routine. The routine is entered if input/output errors result from an attempt to process data records. If no routine is specified and an error occurs, the option specified by the EROPT parameter is executed.

ACCESSING INDEXED SEQUENTIAL DATA SETS

When accessing and/or updating indexed sequential data sets, the DCB subparameters specified for creating indexed sequential data sets are applicable with the following differences, and additions.

DIFFERENCES

[,MACRF={(GL)|(GL,PU)|(R)|(RU,WUA)}]

G - indicates GET macro instruction
L - indicates locate mode

P - indicates PUT macro instruction
U - indicates sequential updating

R - indicates READ macro instruction
U - indicates read for update

W - indicates WRITE macro instruction
UA - indicates add new records, update existing records.

ADDITIONS

[,NCP=1]

specifies the number of channel programs to be established for this data control block. The value 1 is supplied by the compiler.

[,MSWA=relexp]

specifies the address of a main storage work area reserved for the control program.

If specified when fixed-length records are being added to the data set, the control program uses the work area to speed up record insertion.

[,SMSW=absexp]

specifies the number of bytes reserved for the main storage work area. For unblocked records, the work area must be large enough to contain the count, key, and data fields of all the blocks on one track. For blocked records, the work area must be large enough to contain one logical record plus the count and data fields of all the blocks on one track. The maximum number of blocks on one track is 32,767.

[,EODAD=relexp]

specifies the address of the user's end-of-data set exit routine for input data sets. This routine is entered when the user requests a record and there are no more records to be retrieved. If no routine has been provided, the task is abnormally terminated.

Table 11 shows the values supplied for DCB subparameters by the COBOL compiler, by statements in the COBOL source program, and those subparameters that may be supplied by a DD statement for an indexed sequential data set.

Table 11. DCB Subparameter Values For Indexed Sequential Data Set

DCB Parameter	Value Supplied Unconditionally By COBOL Compiler	Value Supplied by COBOL Source Statement	Value Supplied By DD Statement
OPTCD	WLI		
DSORG	IS		DSORG=IS
MACRF			
Sequential	GL	OPEN INPUT	
	GL,PU	OPEN I-O	
	PL	OPEN OUTPUT	
Random	R	OPEN INPUT	
	RU,WUA	OPEN I-O	
DDNAME		External-name in ASSIGN Clause	
RECFM		RECORDING MODE Clause	
LRECL		RECORD CONTAINS Clause	
BLKSIZE		BLOCK CONTAINS Clause	
RKP		RECORD KEY Clause	
KEYLEN		RECORD KEY Clause	
NCP	1		
MSWA		TRACK AREA Clause	
BUFNO		RESERVE Clause	BUFNO=nnn
SMSW		TRACK AREA Clause	
EODAD		AT END Clause	
SYNAD		USE Statement Option 5	

DCB FOR CREATING DIRECT OR RELATIVE ORGANIZATION DATA SET

```
DCB=([,OPTCD=W] [,DSORG=PS]
  [,MACRF=(WL)] [,DDNAME=symbol]
  [,RECFM={F|V|U}] [,LRECL=absexp]
  [,BLKSIZE=absexp] [,DEVD=DA,KEYLEN=value]
  [,NCP=1] [,EODAD=relexp]
  [,SYNAD=relexp])
```

OPTCD=W

specifies an optional service to be performed by the program as follows:

W - a write validity check (on direct-access devices only)

DSORG=PS

specifies the organization of the data set as PS (a physical sequential organization)

MACRF=(WL)

specifies the macro instruction that will be used in processing the data sets as follows:

W - indicates use of WRITE macro instruction

L - indicates LOAD mode for direct data set

DDNAME=symbol

specifies the name of the DD statement that will be used to describe the data set to be processed.

RECFM={F|V|U}

specifies the characteristics of the record in the data set as follows:

F - fixed-length records

V - variable-length records
U - undefined records

LRECL=absexp

specifies the length of a format-F logical record in bytes or the maximum length of a format-V or U logical record

BLKSIZE=absexp

specifies the maximum length of the block in bytes for format-F records. The length must be an integral multiple of the LRECL value. For format-V records, the length must include the four-byte block length field that is recorded at the beginning of each block.

DEVD=DA,KEYLEN=value

specifies the device or devices on which the data set resides
DA - specifies a direct-access device
KEYLEN - specifies the length of the key, in bytes, associated with a physical block.

NCP=1

specifies the maximum number of READ or WRITE macro instructions that are issued before a CHECK macro instruction.

SYNAD=relexp

specifies the address of the user's synchronous error exit routine. The routine is entered if input/output errors result from an attempt to process data records. If no routine is specified and an error occurs, the option specified by the EROPT parameter is executed.

ACCESSING DIRECT OR RELATIVE ORGANIZATION DATA SETS

When accessing and/or updating direct data sets, the DCB subparameters specified for creating direct data sets are applicable, with the following differences, and additions.

DIFFERENCES

[,OPTCD={WE|WR}]

W - indicates a write validity check be performed
E - indicates an extended search be performed
R - indicates that relative block addresses be used

[,DSORG=DA]
DA - indicates direct or relative organization

[,MACRF=$\left\{\begin{matrix}\text{R}\\\text{WL}\\\text{(RKC,[WAKC])}\\\text{(RIC,[WAIC])}\end{matrix}\right\}$

R - indicates use of READ macro instruction
K - indicates that search argument is a key
I - indicates that search argument is a block identification
W - indicates use of WRITE macro instruction
A - indicates that blocks are to be added to the data set
C - indicates use of check macro.

ADDITIONS

[,KEYLEN=absexp]

specifies the length of the key for each physical record in bytes

[,LIMCT=absexp]

specifies the maximum number of blocks or tracks searched when the extended search option is chosen

[,EODAD=relexp]

specifies the address of the user's end-of-data set exit routine for input data sets. This routine is entered when the user requests a record and there are no more records to be retrieved. If no routine has been provided, the task is abnormally terminated.

Table 12 shows the values supplied for DCB subparameters by the COBOL compiler, by statements in the COBOL source program, and those subparameters that may be supplied by a DD statement for a direct-access data set.

Table 12. DCB Subparameter Values For Direct or Relative Organization Data Sets

DCB Parameter	Value Supplied Unconditionally By COBOL Compiler	Value Supplied by COBOL Source Statement	Value Supplied By DD Statement
OPTCD			
Direct organization	WE		
Relative organization	WR		
DSORG			
Sequential access	PS		
Random-access	DA		
MACRF			
Sequential-access	R	OPEN INPUT	
	WL	OPEN OUTPUT	
Random-access Direct organization	RKC	OPEN INPUT	
	RKC,WAKC	OPEN I-O	
Relative organization	RIC	OPEN INPUT	
	RIC,WAIC	OPEN I-O	
DDNAME		External-name in ASSIGN clause	
DEVD	DA,KEYLEN=nnn (nnn=0 - 255)	SYMBOLIC KEY Clause	
RECFM		RECORDING MODE Clause	
LRECL		RECORD CONTAINS Clause	
BLKSIZE		BLOCK CONTAINS Clause	
NCP	1		
KEYLEN		SYMBOLIC KEY Clause	
LIMCT		APPLY Clause Option 1	
EODAD		AT END Clause	
SYNAD		USE Statement Option 5	

The following DD statements are examples for processing indexed sequential, direct, relative, sets.

Example of DD statements for Indexed Sequential organization:

```
//GO.SYSUT5 DD DSNAME=ISAM(PRIME),        X
//             UNIT=2311,                 X
//             VOLUME=SER=111111,         X
//             DCB=(,DSORG=IS),           X
//             SPACE=(CYL,(3)),           X
//             DISP=(NEW,KEEP)
//          DD DSNAME=ISAM(OVFLOW),       X
//             UNIT=2311,                 X
//             VOLUME=SER=111111,         X
//             DCB=(,DSORG=IS),           X
//             SPACE=(CYL,(1)),           X
//             DISP=(NEW,KEEP)
//          DD DSNAME=ISAM(INDEX),        X
//             UNIT=2311,                 X
//             VOLUME=SER=111111,         X
//             DCB=(,DSORG=IS),           X
//             SPACE=(CYL,(1)),           X
//             DISP=(NEW,KEEP)
```

This example specifies:

- that an indexed sequential data set (named ISAM), is to be processed on a 2311 disk pack;
- that the volume serial number of the volumes required by the data set is 111111;
- that the data set is to be kept after execution of the run;
- that the prime area consists of three cylinders, the overflow area, and the index area of one cylinder each, and
- that the COBOL external name for the data set is SYSUT5.

Example of DD statement for Direct or Relative organizations:

```
//GO.SYSUT6 DD DSNAME=&RANDOM,UNIT=SYSDA,X
//             SPACE=(TRK,(10,5))
```

This example specifies:

- that a temporary data set (named RANDOM) is to be processed on a direct access device;
- that the data set be allocated a space of ten tracks, with a secondary allocation of 5 tracks, if needed;
- that the COBOL external name for this data set is SYSUT6.

Example of DD statement for sequential organization:

```
//GO.SYSUT7 DD DSNAME=SEQUENTIAL,         X
//             UNIT=2311,                 X
//             DISP=(NEW,DELETE),         X
//             DCB(,OPTCD=W),             X
//             SPACE=(TRK,(20,5))
```

This example specifies:

- that a data set (named SEQUENTIAL) is to be processed on a 2311 disk pack;
- that the data set is to be deleted after execution;
- that the data set be allocated 20 tracks with a secondary allocation of 5 tracks, if needed; and
- that the COBOL external name for the data set is SYSUT7

Note: For sequential, direct, and relative organizations, essentially the same DD statements can be used.

This section contains figures showing the job-control statements used in the COBOL (E-Level Subset) cataloged procedures and a brief description of each procedure. This section also describes statements used to override statements and parameters in any cataloged procedure. (The use of cataloged procedures is discussed under Job Processing.)

COMPILE

The cataloged procedure for compilation (COBEC) is shown in Figure 27.

```
//COB EXEC PGM=IEPCBL00
//SYSPRINT DD SYSOUT=A
//SYSPUNCH DD UNIT=SYSCP
//SYSUT1 DD UNIT=SYSDA,                          X
//             SPLIT=(2,CYL,(40,10))
//SYSUT2 DD UNIT=SYSDA,SPLIT=4
//SYSUT3 DD UNIT=SYSDA,SPLIT=4
```

Figure 27. Compile Cataloged Procedure (COBEC)

The EXEC statement that invokes the COBOL-E compiler is named COB; the EXEC statement indicates that the operating system is to execute the program IEPCBL00 (the name for the COBOL-E compiler). Compiler options are not explicitly supplied with the procedure: default options are assumed. The programmer can override these default options by using an EXEC statement that includes the options he desires. To execute a compilation using the cataloged compile procedure, the programmer must add SYSIN and, if necessary, SYSLIB.

LINKAGE EDIT AND EXECUTE

The cataloged procedure to linkage edit COBOL object modules and execute the resulting load modules (COBELG) is shown in Figure 28.

```
//LKED EXEC PGM=IEWL,                             1
//             PARM=(XREF,LIST,LET)
//SYSLIB DD DSNAME=SYS1.COBLIB,                   1
//             DISP=(OLD,KEEP)
//SYSLMOD DD DSNAME=&GODATA(RUN),                 1
//             DISP=(NEW,PASS),                   2
//             UNIT=SYSDA,                        3
//             SPACE=(1024,(50,20,1))
//SYSUT1 DD UNIT=SYSDA,                           1
//             SPACE=(1024,(50,20))
//SYSPRINT DD SYSOUT=A
//SYSLIN DD DDNAME=SYSIN
//GO EXEC PGM=*.LKED.SYSLMOD,                     1
//             COND=(5,LT,LKED)
//SYSABEND DD SYSOUT=A
//SYSOUT DD SYSOUT=A,                             1
//             DCB=(,BLKSIZE=120,                 2
//             LRECL=120)
```

Figure 28. Linkage Edit and Execute Cataloged Procedure

The EXEC statement that invokes the linkage editor is named LKED and specifies that the operating system is to execute the program IEWL, the name for the linkage editor program. This statement also specifies the XREF, LIST, and LET options for the linkage editor. The programmer can override these options by using the EXEC statement in the input stream.

The EXEC statement named GO executes the load module produced by the linkage editor procedure step. The PGM parameter specifies that the operating system is to execute the data set defined by the DD statement SYSLMOD in the procedure step LKED. To execute a run using the cataloged linkage edit and execute procedure, the programmer must add SYSIN.

COMPILE, LINKAGE EDIT, AND EXECUTE

The cataloged procedure to compile, linkage edit, and execute a COBOL source module (COBECLG) is shown in Figure 29.

```
//COB EXEC PGM=IEPCBL00
//SYSPRINT DD SYSOUT=A
//SYSUT1 DD UNIT=SYSDA,SPLIT=(2,CYL,(40,10))
//SYSUT2 DD UNIT=SYSDA,SPLIT=4
//SYSUT3 DD UNIT=SYSDA,SPLIT=4
//SYSPUNCH DD DSNAME=&LOADSET,DISP=(MOD,PASS),                    1
//             UNIT=SYSDA,SPACE=(TRK,(50,10))
//LKED EXEC PGM=IEWL,PARM=(XREF,LIST,LET),COND=(9,LT,COB)
//SYSLIN DD DSNAME=&LOADSET,DISP=(OLD,DELETE)
//          DD DDNAME=SYSIN
//SYSLMOD DD DSNAME=&GODATA(RUN),DISP=(NEW,PASS),                 1
//             UNIT=SYSDA,SPACE=(1024,(50,20,1))
//SYSLIB DD DSNAME=SYS1.COBLIB,DISP=(OLD,KEEP)
//SYSUT1 DD UNIT=(SYSDA,SEP=(SYSLIN,SYSLMOD)),                    1
//             SPACE=(1024,(50,20))
//SYSPRINT DD SYSOUT=A
//GO EXEC PGM=*.LKED.SYSLMOD,COND=((9,LT,COB),(5,LT,LKED))
//SYSABEND DD SYSOUT=A
//SYSOUT DD SYSOUT=A,DCB=(,BLKSIZE=120,LRECL=120)
```

Figure 29. Compile, Linkage Edit, and Execute Cataloged Procedure

The cataloged procedure COBECLG consists of the statements in the COBEC and COBELG procedures, with one exception. The DD Statement SYSLIN in the linkage editor procedure step LKED identifies the compiler output as the primary input. The programmer does not have to define the linkage editor input as he did with the procedure COBELG, but he must define the data set SYSIN for the compiler so that the source module can be read.

USER CATALOGED PROCEDURES

The programmer can write his own cataloged procedures and tailor them to the facilities in his installation. He can also permanently modify IBM-supplied cataloged procedures. For information about modifying cataloged procedures that are members of a symbolic library, see the publication, IBM System/360 Operating System: Utilities. An example of modifying cataloged procedures is shown in Appendix A of this publication.

OVERRIDING CATALOGED PROCEDURES

Cataloged procedures are composed of EXEC and DD statements. A feature of the operating system is its ability to read control statements and modify a cataloged procedure for the duration of the current job. Overriding is only temporary; that is, the parameters added or modified are in effect only for the duration of the job.

The following text discusses the techniques used to modify cataloged procedures.

OVERRIDING PARAMETERS IN THE EXEC STATEMENT

Two forms of keyword parameters ("keyword" and "keyword.procstep") are discussed under Job-Control Language. The form "keyword.procstep" is used to add or override parameters in an EXEC statement in a cataloged procedure.

Note: When the PARM parameter is overridden, all options stated in the EXEC statement in the procedure step are deleted.

The COBOL programmer can, for example, add (or override) compiler or linkage editor options for an execution of a cataloged procedure, or he can state different conditions for bypassing a job step.

Example 1: Assume the cataloged procedure COBEC is used to compile a program, and the programmer wants to specify the NODECK option. The following statement can be used to invoke the procedure, and to supply the compiler options.

```
//STEP1 EXEC COBEC,                                              X
//             PARM.COB=NODECK
```

The PARM option applies to the procedure step COB.

Example 2: Assume the cataloged procedure COBELG is used to linkage edit and execute

a program. Furthermore, the MAP option overrides XREF, LET, and LIST in the linkage editor step and the COND parameter is changed for the execution of the load module. The following EXEC statement adds and overrides parameters in the procedure.

```
//PERFORM EXEC COBELG,PARM.LKED=MAP,      X
//             COND.GO=(3,LT,PERFORM.LKED)
```

The PARM parameter applies to the linkage editor procedure step LKED, and the COND parameter applies to the execution procedure step GO.

Example 3: Assume a source module is compiled, linkage edited, and executed using the cataloged procedure COBECLG. Furthermore, the compiler option BUFSIZE and the linkage editor option MAP are specified. The following EXEC statement adds and overrides parameters in the procedure.

```
//STEP1 EXEC COBECLG,                      1
//             PARM.COB='BUFSIZE=600',     2
//             PARM.LKED=MAP
```

OVERRIDING AND ADDING DD STATEMENTS

A DD statement with the name "stepname.ddname" is used to override parameters in DD statements in cataloged procedures or to add DD statements to cataloged procedures. The "stepname" identifies the step in the cataloged procedure. If "ddname" is the name of a DD statement:

1. present in the step, the parameters in the new DD statement override parameters in the DD statement in the procedure step.

2. not present in the step, the new DD statement is added to the step.

In any case, the modification is only effective for the current execution of the cataloged procedure.

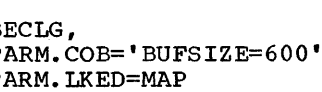
When overriding, the original DD statement in the cataloged procedure is copied, and the parameters specified in it are replaced by the corresponding parameters in the new DD statement. Therefore, only parameters that must be changed are specified in the new DD statement. Therefore, only parameters that must be changed are specified in the new DD statement. Except for the DCB parameter, only an entire parameter may be overridden.

If more than one DD statement is modified, the overriding DD statements must be in the same order as the DD statements appear in the cataloged procedure. Any DD statements added to the procedure must follow overriding DD statements.

When the procedures COBEC and COBECLG are used, a DD statement must be added to define the SYSIN data set to the compile step in the procedures (see Figures 15 and 20). When the procedure COBELG is used, a DD statement must be added to define the SYSIN data set (see Figure 17).

Example 1: Assume the data sets identified by ddnames CALC1 and CALC2 are named, cataloged, and assigned specific volumes. The following DD statements are used to add this information and indicate the location of the source module.

```
//JOB1 JOB MSGLEVEL=1
//STEP1 EXEC COBECLG
//COB.SYSIN DD *
```




```
/*
//GO.CALC1 DD DSNAME=FTAX,                 X
//            DISP=(NEW,CATLG),            X
//            VOLUME=(PRIVATE,SER=987K)
//GO.CALC2 DD DSNAME=STAX,                 X
//            DISP=(NEW,CATLG),            X
//            VOLUME=(PRIVATE,SER=1020)
```

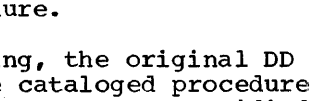
Example 2: Assume the compile, linkage edit, and execute cataloged procedure (COBECLG) is used with:

1. A COBOL main program MAIN in the input stream.

2. A linkage editor control statement that specifies an additional library, MYLIB. MYLIB is used to resolve external references for the symbols A, B, and C.

The following example shows the deck structure.

```
//JOBCLG JOB 00,COBOLPROG,MSGLEVEL=1
//STEP1 EXEC COBECLG
//COB.SYSIN DD *
```




```
/*
//LKED.ADDLIB DD DSNAME=MYLIB
//LKED.SYSIN DD *
  LIBRARY ADDLIB (A,B,C)
/*
//GO.ddname DD statements
```

The DD statement COB.SYSIN indicates to the compiler that the source module is in the input stream. The DD statement LKED.ADDLIB defines the additional library MYLIB to the linkage editor. The DD statement LKED.SYSIN defines a data set that is

concatenated with the primary input to the linkage editor. The linkage editor control statements and the object modules appear as one data set to the linkage editor. The DD statements GO.ddname define data sets for input and output for the load module.

This section is intended to help the programmer reduce the amount of storage required for a program, which should result in a reduction of execution time, and/or linkage editing time for that program. This section discusses:

- General COBOL programming suggestions for effective coding.
- Descriptions of data forms, numeric data format usage and other related factors affecting the use of main storage.
- Specific examples (of data definitions, relationals, arithmetics and complex instructions) to illustrate the effect they have on main storage.
- Specific examples of good and bad coding techniques along with some important considerations when using certain types of data.
- Effective techniques for handling files along with I/O considerations and debugging techniques.

Application of the techniques and suggestions discussed should result in a more efficient program.

CONSERVING STORAGE

The data division is important in that the definition of data can affect the number of program steps generated in the procedure division.

The definition of data used in computationals is also important. The saving of one byte in the data division can cause a significant increase in the number of instructions generated in the procedure division. Conversely, a meaningful addition of one byte in the data division can result in a savings of 20 or more bytes of generated instructions for the procedure division. By judicious choice of such items as decimal-point alignment, sign declaration, and usage, the object code produced for the procedure division is more efficient. The compiler resolves all of the allowable mixed data usages encountered. If the programmer is unconcerned about the program's efficiency, the required additional instructions are generated and additional storage is used.

A programmer, coding according to the suggestions set forth here, can effect a substantial savings in storage. Attention to decimal alignment (one of the suggestions) saves storage as follows.

To execute a statement, data must be aligned. Neglecting decimal alignment when defining data, forces the compiler to align decimal points, which costs 18 or more bytes for each alignment procedure executed, thus using storage unnecessarily.

To give the programmer an idea of the effect data has on storage when data is defined without regard to optimization of data declarations, consider the following percentages and the ensuing example.

In a typical source statement deck, the frequency of the most common verbs written in the procedure division of a COBOL program, averaged over a number of programs, is:

MOVES - 50%
GO TO - 20%
IF - 15%
Miscellaneous (arithmetics, I/O PERFORMS, etc.) - 15%

Assume that the number of move statements, out of a total of 250 procedural statements, is 125 and that all the sending fields and related receiving fields are defined without decimal alignment (worst case).

An example of one pair of fields is:

```
77 A PICTURE   99V9 COMPUTATIONAL-3.
   (sending field)
77 B PICTURE 999V99 COMPUTATIONAL-3.
   (receiving field)
```

Because the receiving field is one decimal position larger than the sending field, decimal alignment must be performed.

The cost in bytes of decimal alignment for these moves is: 125 moves times 18, or 2,250 bytes of storage. Each time these moves are executed 2,250 bytes of storage are used.

A programmer aware of the cost of nonalignment can conserve great amounts of storage by simply aligning decimals. Using one additional byte to align decimals in the data sending or receiving fields is small in cost, considering the savings possible in the procedure division.

The programming suggestions given in the ensuing text should result in a savings in storage and/or faster compilations.

BASIC PRINCIPLES OF EFFECTIVE COBOL CODING

The techniques described in this section will help the programmer write efficient programs. If followed, the suggestions will reduce the number of bytes used by his program. The basic principles for writing efficient COBOL programs are:

- Match decimal places in related fields (decimal-point alignment).
- Match integer places in related fields (unequal-length fields).
- Do not mix usage of data formats (mixed-data formats).
- Include an S (sign) in all numeric pictures (sign control).
- Keep arithmetic expressions out of conditionals (conditional statements).

GENERAL PROGRAMMING SUGGESTIONS

The following is a list of general coding suggestions to aid the programmer in writing COBOL programs. Simple examples are given here to illustrate the use of the suggestions listed. The vast number of ways data can be defined and used makes it prohibitive to illustrate the cost (in bytes) of handling each situation. The values in number of bytes in the examples given are representative. They vary widely according to the way data is defined and used.

Specific costs in number of bytes for several different methods of representing data are given under Examples Showing Effect of Data Declarations.

DECIMAL-POINT ALIGNMENT

The number of decimal positions should be the same whenever possible. If they are not, additional moves for padding, sign movement, and blanking-out result. The impact on storage is illustrated under Conserving Storage.

Statements involving fields with an unequal number of digits require intermediate operations for decimal-point alignment.

Define data efficiently, or move it to a work area to align data used in multiple operation.

To get efficient code, the programmer should align decimal points wherever possible. As a general rule, two or four additional instructions (12 to 18 bytes) are required in basic arithmetic statements and IF statements when decimal-point alignment is necessary to process two COMPUTATIONAL-3 fields.

Example:

```
77 A PICTURE S999V99  COMPUTATIONAL-3.
77 B PICTURE S99V9    COMPUTATIONAL-3.
```

By adding one more decimal place to FIELD B, (PICTURE S999V99), the need for alignment instructions is eliminated, and no more bytes are required for field B. (Remember, hardware requires an odd number of digits for internal decimal fields. Use an odd number of nines when defining data in COMPUTATIONAL-3 format. This practice results in more efficient object code without using additional storage for the item defined.)

Example: ADD 1 TO A.

The literal is compiled in internal decimal form, but decimal-point alignment instructions are necessary (4 instructions, 18 bytes). If instead, the literal is written 1.00, only one byte is added in the literal area. The 18 bytes required for alignment of decimal points are eliminated.

UNEQUAL-LENGTH FIELDS

Use the same number of integer digits in a field. An intermediate operation may be required when handling fields of unequal length. For example, zeros may have to be inserted in numeric fields and blanks in alphabetic or alphameric fields in order to pad out to the proper length. To avoid these operations, be sure that the number of integer digits in fields used together are equal. Any increase in data field size is more than compensated for by the savings in generated object code.

For example, if data is defined as:

```
SENDFLD PICTURE S999
RECEIVEFLD PICTURE S99999.
```

and SENDFLD is moved to RECEIVEFLD, the cost of zeroing high-order positions (numeric fields are justified right) is 10

bytes. To eliminate these 10 bytes define SENDFLD as:

```
SENDFLD PICTURE S99999.
```

MIXED-DATA FORMATS

Do not mix data formats. When fields are used together in move, arithmetic, or relational statements, they should be in the same format whenever possible. Conversions require additional storage and execution time. Any operations involving data items of different formats require conversion of one of the items to a matching data format before the operation can be executed. For example, when comparing a DISPLAY field to a COMPUTATIONAL-3 field, the code generated by the COBOL processor moves the DISPLAY field to an internal work area, converting it to a COMPUTATIONAL-3 field. It then executes the compare. This usage, although valid in COBOL, has the effect of reducing the efficiency of the program, by increasing its size. For maximum efficiency, avoid mixed data formats or use a onetime conversion; that is move the data to a work area, thus converting it to the matching data format. By referencing the work area in procedural statements, the data is converted only once instead of for each operation.

The following example illustrates the conversions that take place when the components of a COMPUTE are defined:

```
A COMPUTATIONAL-1.
B PICTURE S99V9 COMPUTATIONAL-3.
C PICTURE S9999V9 COMPUTATIONAL-3.
```

and the following computation is specified,

```
COMPUTE C = A * B.
```

the internal decimal data (COMPUTATIONAL-3) is converted to floating-point format and then the COMPUTE is executed.

The result (which is in floating point) is converted to internal decimal. The required conversion routines are time consuming and use storage unnecessarily.

The following examples show what must logically be done, before the indicated operations can be performed, when working with mixed-data fields.

DISPLAY to COMPUTATIONAL-3

To Execute a MOVE: No Additional code is required (if proper alignment exists) because one instruction can both move and convert the data.

To Execute a COMPARE: Before a COMPARE is executed, DISPLAY data must be converted to COMPUTATIONAL-3 format.

To Perform Arithmetics: Before arithmetics are performed, DISPLAY data is converted to COMPUTATIONAL-3 data format.

DISPLAY to COMPUTATIONAL

To Execute a MOVE: Before the MOVE is executed, DISPLAY data is converted to COMPUTATIONAL-3 data format, and then the COMPUTATIONAL-3 data to COMPUTATIONAL data format.

To Execute a COMPARE: Before a compare is executed, DISPLAY data is converted to COMPUTATIONAL-3 data format, and the COMPUTATIONAL data to COMPUTATIONAL-3 format.

To Perform Arithmetics: Before arithmetics are performed, DISPLAY data is converted to COMPUTATIONAL-3 format, and then the COMPUTATIONAL-3 data to COMPUTATIONAL format.

COMPUTATIONAL-3 to COMPUTATIONAL

To Execute a MOVE: Before a MOVE is executed, COMPUTATIONAL-3 data is moved to a work field, and then converted to COMPUTATIONAL data format.

To Execute a COMPARE: Before a COMPARE is executed, COMPUTATIONAL data is converted to COMPUTATIONAL-3 data format.

To Perform Arithmetics: Before arithmetics are performed, COMPUTATIONAL-3 data is converted to COMPUTATIONAL data format.

COMPUTATIONAL to COMPUTATIONAL-3

To Execute a MOVE: Before a MOVE is executed, COMPUTATIONAL data is converted to COMPUTATIONAL-3 data format.

To Execute a COMPARE: Before a COMPARE is

executed COMPUTATIONAL data is converted to COMPUTATIONAL-3 data format.

To Perform Arithmetics: Before arithmetics are performed, COMPUTATIONAL data is converted to COMPUTATIONAL-3 data format.

COMPUTATIONAL to DISPLAY

To Execute a MOVE: Before a MOVE is executed, COMPUTATIONAL data is converted to COMPUTATIONAL-3 data format, and then the COMPUTATIONAL-3 data to DISPLAY data format.

To Execute a COMPARE: Before a COMPARE is executed, COMPUTATIONAL data is converted to COMPUTATIONAL-3 data format, and DISPLAY data to COMPUTATIONAL-3 data format.

To Perform Arithmetics: Before arithmetics are performed, COMPUTATIONAL data is converted to COMPUTATIONAL-3 data format, and DISPLAY data to COMPUTATIONAL-3 data format. The result is generated in a COMPUTATIONAL-3 work area, which is then moved to the DISPLAY result field.

COMPUTATIONAL-3 to DISPLAY

To Execute a MOVE: Before a compare is executed, DISPLAY data is converted to COMPUTATIONAL-3 data format.

To Execute a COMPARE: Before a compare is executed, DISPLAY data is converted to COMPUTATIONAL-3 data format.

To Perform Arithmetics: Before arithmetics are performed, DISPLAY data is converted to COMPUTATIONAL-3 data format. The result is generated in a COMPUTATIONAL-3 work area, which is then converted and moved to the DISPLAY result field.

DISPLAY to DISPLAY

To perform Arithmetics: Before arithmetics are performed, all DISPLAY data is converted to COMPUTATIONAL-3 data format. The result is generated in a COMPUTATIONAL-3 work area, which is then converted and moved to the DISPLAY result field.

Conversion of COMPUTATIONAL-1 or -2 Data

For efficient object code, use of floating-point (COMPUTATIONAL-1 or -2) numbers mixed with other usages should be held to a minimum. The conversion from internal to external floating point and vice-versa is done by subroutines. Fields used in conjunction with a floating-point number are converted to floating point, causing the object program to perform conversions. For example, assume a COMPUTE is specified as:

```
COMPUTE A = B * C + D + E.
```

Assume B is COMPUTATIONAL-1 or -2 data and all other fields are defined as COMPUTATIONAL-3 data. Fields C, D, and E are converted to COMPUTATIONAL-1 or -2 data format, the calculation performed, and the result converted back from COMPUTATIONAL-1 or -2 data format to COMPUTATIONAL-3 data. If field B is defined as COMPUTATIONAL-3, no conversion is necessary. Use of floating-point numbers is more efficient when used in programs with computational data that is practically all COMPUTATIONAL-1 or -2 type. If it is necessary to use floating-point data, be careful not to mix data formats.

SIGN CONTROL

For numeric fields specified as unsigned (no S in the picture clause of decimal items), the COBOL processor attempts to ensure that a special positive sign (F) is present so that the values are treated as absolute.

The processor moves in a hexadecimal F whenever the possibility of the sign changing exists. Examples are: Subtracting unsigned fields, moving a signed field to an unsigned field, or an arithmetic operation on signed fields where an unsigned result field is specified.

The sign is not checked on input data or on group level moves. The programmer must know what type of data is being used, under those circumstances.

The use of unsigned numeric fields increases the possibility of error (an unintentional negative sign could cause invalid results) and requires additional generated code to control the sign. The use of unsigned fields should be limited to fields that are to be treated as absolute values.

Note: The hexadecimal F, while treated as a plus, does not cause the digit to be printed or punched as a signed digit.

The programmer should include a sign in numeric pictures unless absolute values are desired. The following example illustrates the additional instructions generated by the compiler each time an unsigned field is modified.

If data is defined as:

```
A PICTURE 999.
B PICTURE S999.
C PICTURE S999.
```

and the following moves are made,

```
MOVE B TO A.
MOVE B TO C.
```

moving B to A causes four more bytes of storage to be used than moving B to C, because an absolute value is specified for receiving field A.

CONDITIONAL STATEMENTS

Keep arithmetic expressions out of conditional statements. Computing arithmetic values separately and then comparing them may produce more accurate results than including arithmetic statements in conditional statements. The final result of an expression included in a conditional statement is limited to an accuracy of six decimal places. The following example shows how separating computations from conditionals can improve accuracy.

If data is defined as:

```
77 A PICTURE S9V9999 COMPUTATIONAL-3.
77 B PICTURE S9V9999 COMPUTATIONAL-3.
77 C PICTURE S999V99999999 COMPUTATIONAL-3.
```

and the following conditional statement is written,

```
IF A * B = C GO TO EQUALX.
```

the final result will be 99V999999. Although the receiving field for the final result (C) specifies eight decimal positions, the final result actually obtained in this example contains six decimal places. For increased accuracy, define the final result field as desired, perform the computation, and then make the desired comparison as follows.

```
77 X PICTURE IS S999V9(7) COMPUTATIONAL-3.
COMPUTE X = A * B.
IF X = C GO TO EQUALX.
```

OTHER CONSIDERATIONS WHEN USING DISPLAY AND COMPUTATIONAL FIELDS

DISPLAY (Non-Numeric and External Decimal) Fields

Zeros and blanks are not inserted automatically by the logical instruction set. A move requires coding to insert zeros or blanks. On compares, the smaller item must be moved to a work area where zeros or blanks are inserted before the compare.

COMPUTATIONAL-3 (Internal Decimal) Fields

The decimal feature provides for the automatic insertion of high-order zeros on adds, subtracts, and compares.

When a blank field (40) is moved into a field defined as COMPUTATIONAL-3, the sign position is not changed; thus the invalid sign bits of the blank field are retained. An arithmetic operation with such a field results in a program check. Before moving a blank field into a COMPUTATIONAL-3 field to be operated on, the sign position must be converted to a valid COBOL sign (F0).

COMPUTATIONAL Field

Operating System furnishes a large repertoire of halfword and fullword instructions. Binary instructions require one of the operands to be in a register where a halfword is automatically expanded to a fullword. Therefore, handling mixed halfword and fullword fields requires no additional operations.

COMPUTATIONAL 1 and 2 Fields

A full set of short- and long-precision instructions are provided which enables operations involving mixed precision fields to be handled without conversion.

DATA FORMS

To conserve storage, the programmer must know COBOL data forms, and how they affect storage. Equally important is the way he organizes his data. The following

information illustrates the various types of COBOL data forms, and their respective costs in alignment. Characteristics and requirements are described for the possible usages of numeric data, along with symbolic illustrations of what forms they take within the machine. Also included is a brief discussion of how to organize data efficiently.

ELEMENTARY ITEMS

The number of bytes occupied by data in main storage depends on its format (or mode). Table 13 illustrates the number of bytes required for each class of elementary item.

If files and working storage are organized so that all halfwords, fullwords, and doublewords are grouped together, essentially no additional storage is used. However, if these items are not grouped together properly, the amount of storage required for alignment is:

Halfword - 1 byte
Fullword - 1 to 3 bytes
Doubleword - 1 to 7 bytes

Table 13. Number of Bytes Required for Each Class of Elementary Item

TYPE OF ITEM	CALCULATION OF REQUIRED BYTES FROM PICTURE
DISPLAY	
Alphabetic	Bytes = Number of A's in picture
Alphanumeric	Bytes = Number of X's in picture
External Decimal	Bytes = Number of 9's in picture
{External floating point}	Bytes = Number of characters in picture
Report	Bytes = Number of characters in picture except P, V
COMPUTATIONAL-3	
Internal Decimal	Bytes = (Number of 9's +1 divided by 2, rounded up)
COMPUTATIONAL	Size / Alignment
{Binary} Bytes =	2 if 1≤N≤4 — Halfword Machine Address 4 if 5≤N≤9 — Fullword Machine Address 8 if 10≤N≤18 — Fullword Machine Address Where N=Number of 9's in picture
COMPUTATIONAL-1 or COMPUTATIONAL-2 {Internal floating point} Bytes =	4 if short-precision (computational-1) — Fullword machine address 8 if long precision (computational-2) — Doubleword machine address

GROUP ITEM

Group moves of 256 or less bytes cost less than a series of single alphanumeric moves of the elementary item within the group item. Any move of a group or elementary item greater than 256 bytes in size results in a subroutine being executed.

When computational usage is specified in COBOL slack bytes are inserted to give proper halfword or fullword or double-word boundary alignment. This is necessary so that the elementary item can be handled properly in binary arithmetic. However, using group items that include slack bytes could cause problems.

It is possible for two group items, defined exactly the same, to have a different number of slack bytes because they begin in different places, relative to word boundaries. Since group items use slack bytes as normal data, a move of the smaller of these to the larger can cause a loss of data.

For example, assume two groups are defined as follows:

```
        01 RECORD-1.
           02 GOLD PICTURE XX DISPLAY.
           02 MINERALS COMPUTATIONAL.
Case 1        03 OPAL PICTURE 99.
              03 QUARTZ PICTURE 99999.

        01 RECORD-1.
           02 MINERALS COMPUTATIONAL.
Case 2        03 OPAL PICTURE 99.
              03 QUARTZ PICTURE 99999.
```

Case 1 group (02 MINERALS) consists of a total of six bytes (it does not contain slack bytes).

Case 2 group (02 MINERALS) consists of a total of eight bytes, including two slack bytes.

In case 2, 03 QUARTZ will be preceded by two slack bytes. Thus, if case 2 group (02 MINERALS) is moved to case 1 group, the last two bytes of data will be lost.

If case 1 group (02 MINERALS) is moved to case 2 group, no data will be lost but the elementary 03 QUARTZ will be improperly aligned.

NUMERIC DATA FORMAT USAGE

Figure 30 lists the common characteristics and special characteristics of numeric data.

Type of Data	Bytes Required	Typical Usage	Converted in Arithmetics	Boundary Alignment Required	Special Characteristics
DISPLAY (External Decimal)	1 per digit	Input from cards Output to cards, listings	Yes	No	May be used for numeric fields up to 18 digits long. Fields over 15 digits require extra instructions if used in computations.
COMPUTATIONAL-3 (Internal Decimal)	1 byte per 2 digits after the 1st byte for low-order digit	Input to a report item Arithmetic fields Work areas	Not normally	No	Requires less space than display. Convenient form for decimal alignment. The natural form contains an odd number of digits.
COMPUTATIONAL (Binary)	*2 bytes if 1≤N≤4 *4 bytes if 5≤N≤9 *8 bytes if 10≤N≤18	Subscripting Arithmetic	Yes/No--for mixed usages No--for unmixed usage	Yes	Rounding and on-size error tests are cumbersome. Always must be signed. Fields of over 8 digits require more handling.
COMPUTATIONAL-1 COMPUTATIONAL-2 (Floating) Point)	4 bytes 8 bytes	Fractional exponentiation, or very large or small values	No	Yes	Tends to produce less accuracy. Computational-2 is more accurate than Computational-1. Requires floating-point feature.

*Where N= number of digits in PICTURE.

Figure 30. Characteristics of Numeric Data

MACHINE REPRESENTATION OF DATA ITEMS

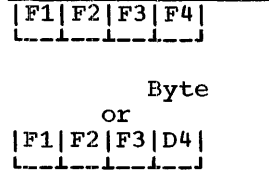
The following examples are machine representations of the various data items in COBOL.

DISPLAY (External Decimal)

If value is -1234, and:

Picture and Usage are:
PICTURE 9999.
or
PICTURE S9999.

Machine Representation is:



The sign position of an unsigned receiving field is changed to a hexadecimal F.

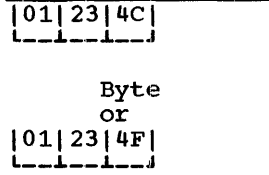
Hexadecimal F is arithmetically treated as plus in low-order byte. The character D represents a negative sign. This form of data is referred to as external decimal.

COMPUTATIONAL-3 (Internal Decimal)

If value is +1234, and:

Picture and Usage are:
PICTURE S9999 COMPUTATIONAL-3.
or
PICTURE 9999 COMPUTATIONAL-3.

Machine Representation is:



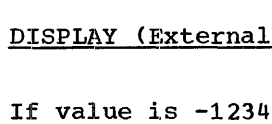
Hexadecimal F is arithmetically treated as plus. The character C represents a positive sign. This form of data is referred to as internal decimal.

COMPUTATIONAL (Binary)

If value is 1234, and:

Picture and Usage are:
PICTURE S9999 COMPUTATIONAL.

Machine Representation is:



A 1 in sign position means number is negative. A 0 in sign position means number is positive.

This form of data is referred to as binary.

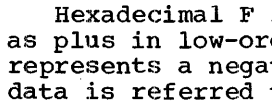
COMPUTATIONAL-1 or COMPUTATIONAL-2 (Internal Floating Point)

If value is +1234, and:

Picture and Usage are:

COMPUTATIONAL-1.

Machine Representation is:



S is the sign position of the number. A 0 in the sign position indicates that the sign is plus. A 1 in the sign position indicates that the sign is minus.

This form of data is referred to as floating point. The example is one of short precision. In long precision, the fraction length is 56 bits. For a detailed explanation of floating-point representation, refer to IBM System/360 Principles of Operation.

EXAMPLES SHOWING EFFECT OF DATA DECLARATIONS

The specific series of instructions that are generated vary widely with the description of the data fields involved. Some examples of the range to be expected by slight differences in the data descriptions follow. The examples of possible expansions used are illustrative and should not be used for estimates of storage.

MOVE

Assume that data items A,B,C, and D are defined for the purpose of being moved as COMPUTATIONAL-3 fields or DISPLAY fields.

```
A PICTURE S99V99.
B PICTURE S99V99.
C PICTURE S99V9.
D PICTURE S99.
```

COMPUTATIONAL-3 Fields

If items A, B, C and D are defined as COMPUTATIONAL-3 fields, then the cost in bytes to:

Move A to B is: (when both integer and decimal places are equal) 6 bytes for a simple move.

Move C to B is: (The sign position must be moved, and the original sign changed.)
6 bytes for a simple move, and 18 bytes for decimal alignment. Total = 24 bytes.

Move C to D is: (The sign requires a separate move.)
6 bytes for a simple move, and 18 bytes for decimal alignment. Total = 24 bytes.

DISPLAY Fields

If data items A, B, C, and D are defined as DISPLAY fields, then the cost in bytes to:

Move A to B is: (When both integer and decimal places are equal) 6 bytes for a simple move.

Move C to D is:
6 bytes for a simple move, and 6 bytes for decimal alignment. Total = 12 bytes.

Move DISPLAY to COMPUTATIONAL-3

The cost in bytes of moving DISPLAY data to a COMPUTATIONAL-3 field is: 6 bytes for conversion, and up to 24 bytes for decimal alignment.

Move COMPUTATIONAL-3 to Report

The cost in bytes of moving COMPUTATIONAL-3 data to a REPORT field is:
24 bytes for a simple move,
12 bytes for floating insertion character,
24 bytes for non-floating digit position,
18 bytes for decimal alignment,
24 bytes for trailing characters,
12 bytes for unmatched digit positions.

RELATIONALS

IF COMPUTATIONAL-3 = COMPUTATIONAL-3

The cost in bytes to execute an IF statement when all data is defined as COMPUTATIONAL-3 is:
6 bytes for the compare and branch instruction (no decimal alignment);
42 bytes for the compare and branch with decimal alignment.

IF DISPLAY = COMPUTATIONAL-3

The cost in bytes to execute an IF statement when data is defined as DISPLAY and COMPUTATIONAL-3 is:
6 bytes for conversion,
18 bytes for the compare and branch instruction, and
18 bytes for decimal alignment.

COMPUTATIONAL = COMPUTATIONAL

The cost in bytes to execute an IF statement when all data is defined as COMPUTATIONAL is:
18 bytes for the compare and branch instruction, when the number of decimal digits is 1 to 9.

The number of bytes required to execute the IF statement is unpredictable when the number of decimal digits is from 10 to 18.

IF A * B = C * D, ETC.

For optimum use of storage when writing any IF statement, first make all computations, and then compare results.

ARITHMETICS

ADD COMPUTATIONAL-3 TO COMPUTATIONAL-3

The cost in bytes to execute an ADD statement when all data is defined as COMPUTATIONAL-3 is:
6 bytes to execute the add, up to 56 bytes for alignment of decimals, and 4 bytes for blanking the sign.

GENERAL TECHNIQUES FOR CODING

The following examples illustrate how COBOL data fields can be manipulated. Some of the techniques illustrated are basic, and can be used in most programs, while others are designed to give the programmer an insight into techniques applicable to more sophisticated programs.

INTERMEDIATE RESULTS IN COMPLEX EXPRESSIONS

The compiler can process complicated statements, but not always with the same efficiency of storage utilization as the source programmer. Because truncation may occur during computations, unexpected intermediate results may be obtained. The rules for truncation are in the publication, IBM System/360 Operating System: COBOL Language.

A method of avoiding unexpected intermediate results is to make critical computations by assigning maximum (or minimum) values to all fields and analyzing the results by testing critical computations for results expected.

Because of concealed intermediate results, the final result is not always obvious.

Alternate Method of Solution (Unexpected Intermediate Results)

The necessity of computing worst case (or best case) results can be eliminated by keeping statements simple. This can be accomplished by splitting the expression, and controlling intermediate results to be sure unexpected final results are not obtained. Consider the following example:

```
COMPUTE B = (A + 3) / C + 27.600.
```

First define adequate intermediate result fields, e.g.:

```
02 INTERMEDIATE-RESULT-A
  PICTURE S9(6)V999.
02 INTERMEDIATE-RESULT-B
  PICTURE S9(6)V999.
```

Then, split up the expression as follows.

```
ADD A,3 GIVING INTERMEDIATE-RESULT-A.
```

Then write:

```
DIVIDE C INTO INTERMEDIATE-RESULT-A
  GIVING INTERMEDIATE-RESULT-B.
```

Then, compute the final result by writing:

```
ADD INTERMEDIATE-RESULT-B, 27.600 GIVING B.
```

ARITHMETIC SUGGESTIONS

ARITHMETIC FIELDS

Initialize arithmetic fields before using them in computations. If the user attempts to use a field without it being initialized, the contents of the field is upredictable: therefore, invalid results might be obtained, or the job might terminate abnormally.

EXPONENTIATION

Avoid exponentiation to a fractional power. For example: V ** (P / N).

This requires the use of the floating-point feature. Use of floating point can be avoided by dividing the statements into separate computations. The first example given requires the use of the floating-point feature. The second example restates the problem, illustrating how the use of floating point can be circumvented.

Assume data is defined:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77 FLD PICTURE S99V9, COMPUTATIONAL-3.
77 EXPO PICTURE S99, COMPUTATIONAL-3.
77 P PICTURE S99.
77 N PICTURE S99.
77 VALUE1 PICTURE S99.
```

Assume values used in the example were appropriately moved into their respective symbolic names as follows: VALUE1 = 5, P = 10, and N = 5.

Example 1

```
COMPUTE FLD = VALUE 1 ** (P / N).
```

Because (P/N) = 10/5 = 2.00 (with decimal places), the floating-point feature is required to solve this statement even though the exponent is an integer. The use of this type of statement involves the floating-point feature because it is not known whether decimal digits are present when the exponent is developed.

Example 2

The statement in example 1 can be solved by writing:

```
COMPUTE EXPO = (P / N).
```

The result is truncated to two significant digits (S99).

Then write:

```
COMPUTE FLD = VALUE1 ** EXPO.
```

Thus, the statement written in example 1 can be solved by dividing it into two separate computations, avoiding the need for floating-point instructions.

Another occurrence that can affect final results is intermediate result truncation. For example:

Assume that VALUE1 = 10, and N = 2.

If COMPUTE FLD = (VALUE1 ** N) - 2 is written, by substitution the result is:

```
  FLD = (VALUE1 ** N) - 2
S99V9 = (S99 ** S99) - 2
S99V9 = (10 ** 2) - 2
S99V9 = 100.0 - 2 By the rule for
    truncation:
     ┌----------┐
  ┌--┴--┐    ┌--┴--┐
S|99V9| = 1|00.0| - 2.
```

The most significant digit is truncated. The final result is then:

```
FLD = 00.0 - 2
FLD = 02.0, could be an unexpected result.
```

The situation can be corrected by expanding the target field (FLD) as follows:

```
77 FLD PICTURE S999V9.
```

Then, when the statement is written (assuming VALUE1 = 10, and N = 2):

```
COMPUTE FLD = (VALUE1 ** N) - 2.
```

The result is:

```
   FLD = (VALUE1 ** N) - 2
S999V9 = (S99 ** S99) - 2
S999V9 = (10 ** 2) - 2.
```

By the rule for truncation:

```
     ┌----------┐
  ┌--┴--┐    ┌--┴--┐
S|999V9| = |100.0| - 2.
```

The result is,

```
          +
FLD = 098.0, which is the expected result.
```

SUBSCRIPTING

Use a constant subscript instead of a variable (data-name) subscript whenever possible. Constant subscripts are resolved during compile time, whereas variable (data-name) subscripts are resolved at object time.

Example

Instead of NAME (S1, S2) use: NAME (1,23) where S1=1, and S2=23.

The address of NAME (in the latter case) is resolved at compile time, based on the given constant subscripts.

When variable subscripting is used, the address of the field is computed each time a subscripted field is referenced.

For efficient coding, frequently referenced subscripted fields should be moved to a work area, manipulated, and if necessary, returned.

Example

```
      ⎧ADD D TO TAB-FIELD (A,B,C).
      ⎪IF TAB-FIELD (A,B,C) = LIMIT-FLD
Bad   ⎪   GO TO ERR.
Code  ⎨MOVE TAB-FIELD (A,B,C) to F.
      ⎪COMPUTE TAB-FIELD (A,B,C) = TAB-FIELD
      ⎩  (A,B,C) + F / G.
```

This coding could be improved by writing:

Good Code
```
MOVE TAB-FIELD (A,B,C) TO WORK-FLD.
ADD D TO WORK-FLD.
IF WORK-FLD = LIMIT-FLD
GO TO ERR.
MOVE WORK-FLD TO F, COMPUTE TAB-FIELD
(A,B,C) = WORK-FLD + F / G.
```

Binary Subscripting

Use binary mode items for subscripting. Data-name subscripts not in binary are converted to binary at object time.

COMPARISONS

Numeric comparisons are usually done in COMPUTATIONAL-3 format; therefore, COMPUTATIONAL-3 is usually the most efficient data format.

Because compiler inserted slack bytes can contain meaningless data, group compares should not be attempted when slack bytes are within the group unless the programmer knows the contents of the slack bytes.

REDUNDANT CODING

To avoid redundant coding of usage designators, use computational designators at the group level (this does not affect the object program).

Example

Instead of:
```
02 FULLER.
  03 A COMPUTATIONAL-3 PICTURE 99V9.
  03 B COMPUTATIONAL-3 PICTURE 99V9.
  03 C COMPUTATIONAL-3 PICTURE 99V9.
```

Write:
```
02 FULLER COMPUTATIONAL-3.
  03 A PICTURE 99V9.
  03 B PICTURE 99V9.
  03 C PICTURE 99V9.
```

EDITING

A high-order nonfloating digit position involves more instructions than a floating digit position.

Example

nonfloating vs floating
999.99 vs $$$9.99

The blank-when-zero is implied in certain pictures. For example:
ZZZ.ZZ

If blank-when-zero is not required for low-order characters, much more efficient coding is generated by pictures such as:
ZZZ.99

OPENING FILES

Open requires a work area that cannot be recovered in a COBOL program. Less storage is used if single-file opens are given (reusing the positions) instead of a multiple open, which requires approximately 500 bytes of additional storage for each file-name.

To conserve storage, use:
```
OPEN INPUT FILEA OPEN INPUT FILEB.
```

rather than:
```
OPEN INPUT FILEA, FILEB.
```

ACCEPT

The ACCEPT verb does not provide for recognition of the last card being read from a card reader. When COBOL detects a /* card a system ABEND occurs (completion code 337). Because of this system action, an end-of-file detection requires special treatment. Thus the programmer must provide his own end card (some card other than /*) which can be tested to detect an end of file.

PARAGRAPH NAMES

Paragraph names use storage when the PERFORM verb is used in the program. Use of paragraph names for comments requires more storage than the use of NOTE or a

blank card. Use NOTE and/or a blank card for identifying in-line procedures where paragraph names are not required.

Example

Avoid.

```
                MOVE A TO B.
                PERFORM JOES-ROUTINE.
JOES-ROUTINE.   COMPUTE A = D + E * F.
```

Recommended:

```
                MOVE A TO B.
                PERFORM ROUTINE.
                NOTE JOE'S ROUTINE.
ROUTINE.        COMPUTE A = D + E * F.
```

TRAILING CHARACTERS

Pictures with a trailing period or comma require that punctuation follow, or the trailing picture character is treated as punctuation.

Example

```
77 A PICTURE IS 999., USAGE IS DISPLAY
```

REDEFINITION

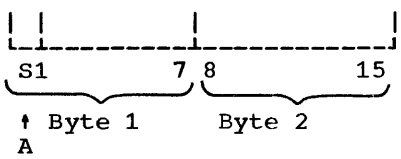
The results of moving a field to itself through the use of redefinition are unpredicatable.

To manipulate unusual data forms, use REDEFINES. For example, a technique for isolating one binary byte follows.

```
  02 A PICTURE S99 COMPUTATIONAL.
  02 FILLER REDEFINES A.
     03 FILLER PICTURE X.
     03 B PICTURE X.
```

Explanation:

COMPUTATIONAL sets up a binary halfword:



02 FILLER REDEFINES A., states that A is to be redefined as follows.

- Ignore first byte (03 FILLER PICTURE X).
- Name second byte B. (03 B PICTURE X).

Now byte B can be moved to a work area, and operated on logically at the assembler level, or compared logically at the COBOL level. It can be stored on a file, and later moved back to its point in a similarly defined field.

Use of data in this manner can present problems regarding signs and numeric values. These problems require a knowledge of both System/360, and COBOL.

Another illustration of using REDEFINES to manipulate data concerns the test IF NUMERIC. A field is considered numeric (under normal language usage) if all the positions of the field are numeric with the exception of the sign position.

If a field is to be considered numeric only when it is unsigned, the sign position must be tested. A technique for relocating the sign (or "shifting") so that it can be tested as an unsigned numeric value follows.

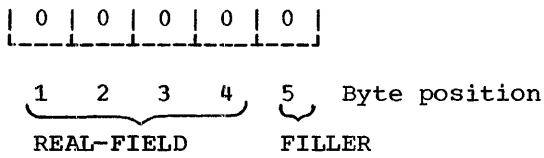
Assume a field is defined:

```
02 IF-NUM-FIELD PICTURE X(5) VALUE '00000'.
02 CHANGE-FIELD REDEFINES IF-NUM-FIELD.
   03 REAL-FIELD, PICTURE S9(4).
   03 FILLER, PICTURE X.
```

IF-NUM-FIELD defines a 5-byte alphameric field.
REAL-FIELD redefined this field to be 4 bytes numeric.

The fields appear in storage as follows:

IF-NUM-FIELD

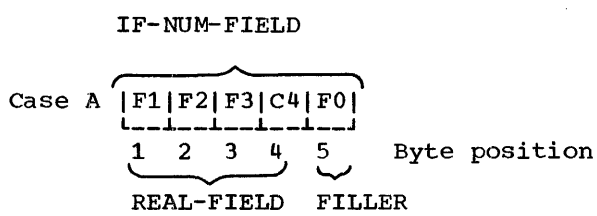


To make an IF NUMERIC, test true for only unsigned fields.

1. Move the 4-byte value to be tested into REAL-FIELD. The value and its sign occupy bytes 1-4.

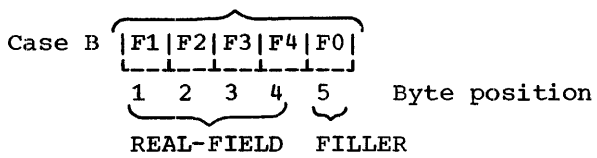
   For example:

   If +1234 is moved to REAL-FIELD, the resultant field appears in storage as follows:



Note that the low-order byte (rightmost byte) of IF-NUM-FIELD retains its initial value of 0.

If 1234 is moved to REAL-FIELD, the resultant field appears in storage as follows:



2. Test IF-NUM-FIELD FOR NUMERIC. All four bytes of REAL-FIELD will be tested as an unsigned numeric value because the sign position was "shifted left one position," and is no longer in the units position of IF-NUM-FIELD. If the value is unsigned, a hexadecimal F appears in the sign position or fourth byte of the 4-byte field, and it appears as un unsigned numeric.

Thus in the preceding example, when the fourth byte is tested in case A, the numeric test fails, but when tested in case B the numeric test is satisfied.

ALIGNMENT AND SLACK BYTES. - (A CONSIDERATION WHEN USING BINARY OR FLOATING POINT DATA.)

Unless binary or floating-point data is used the user need no be concerned with slack bytes. The number of bytes of main storage necessary for the data division must include bytes added to produce valid boundary alignment for binary and floating-point data fields.

Slack bytes required to align data are generated by the compiler.

Example:

```
01 RECORD.
  02 FLD-1 PICTURE IS X(2).
  02 FLD-2 PICTURE IS S99999 COMPUTATIONAL.
```

Because FLD-2 is binary and five digits in length, the compiler sets aside one fullword which must be aligned on a fullword boundary. In this example, two slack bytes are required. The compiler inserts them automatically.

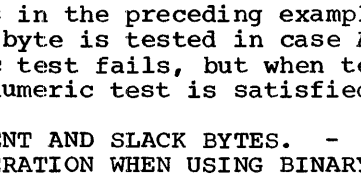
A warning diagnostic is given when slack bytes are inserted by the compiler.

Because COBOL aligns computational fields on output files and expects them to contain slack bytes (where required) on input files, a problem could exist when reading or writing a file.

A file to be read that contains computational fields without slack bytes must be coded in the same manner. That is, it must be coded with the knowledge that it does not contain slack bytes. If the file contains computational data without slack bytes, the data will not be properly aligned when read from the file; thus it cannot be processed by the compiler.

The following is a technique for manipulating computational data not containing slack bytes so that it may be processed by the compiler.

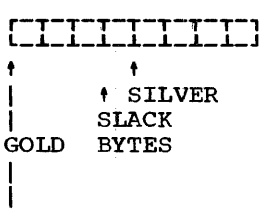
Assume a group record called RECORD-C exists on a file and consists of 2-bytes of alphameric data called GOLD, and 4-bytes of binary data called SILVER. The record on the file would look like:



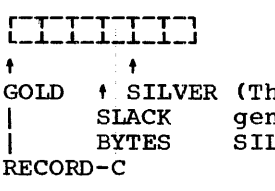
If an FD were defined:

```
01 RECORD-C.
  02 GOLD PICTURE XX.
  02 SILVER PICTURE S99999 COMPUTATIONAL.
```

The compiler assumes the following structure:



When the record on the file is read, it is placed in the area defined, left justified. The area thus contains the following:



Thus the first 2-bytes of the 02 SILVER are lost because of misalignment. Hence, when the 02 SILVER is accessed, only the last 2-bytes are available.

To circumvent this problem, define RECORD-C as follows:

```
01 RECORD-C.
  02 GOLD PICTURE XX.
  02 SILVER PICTURE XXXX.
```

and a GROUP item such as:

```
01 LEAD.
  02 DIAMOND PICTURE S99999 COMPUTATIONAL.
```

Now, access the record (RECORD-C). This places it in the buffer, properly aligned. Then move the 4-byte elementary 02 SILVER (which is defined as alphameric but is actually binary data) to the record 01 LEAD. Because the 01 LEAD is a group item, the data moved retains its original form (no data conversion takes place) and the elementaries 02 SILVER and 02 DIAMOND are properly aligned. Thus, by accessing DIAMOND, the binary data can be operated on as desired.

Assuming the same record (RECORD-C) out on the file, an alternate method of obtaining proper alignment when reading the record follows.

Define a record in an FD as follows:

```
01 RECORD-C.
  02 GOLD PICTURE XX.
  02 SILVER PICTURE XXXX.
```

The area defined would look like:

Then define a record in the WORKING-STORAGE section as:

```
01 BRASS.
  02 LEAD PICTURE XXXX.
  02 DIAMOND REDEFINES LEAD PICTURE,
     S99999 COMPUTATIONAL.
```

As before, when the record is accessed, it is placed in the buffer properly aligned. Its structure in the buffer would be:

Now move the 4-byte elementary 02 SILVER to the elementary 02 LEAD. Because the 02 SILVER and 02 LEAD elementaries are both defined as display, the data retains its original form and the elementaries are properly aligned. By accessing the REDEFINES (DIAMOND), the binary data can be operated on as desired. The same problem could exist when reading or writing floating-point data.

For a complete discussion of slack bytes, refer to the publication, IBM System/360 Operating System: COBOL Language.

GENERAL INFORMATION--FILE HANDLING

BUFFERS

In IBM System/360 Operating System COBOL, a buffer is as a designated area in main storage for I/O transactions. When a file is read, a block is read into a buffer where the records are addressed directly as they are accessed. Use of the READ or WRITE directs a pointer to the appropriate record, or record area, of interest in the buffer.

RECORD BLOCKING

The size of the buffer area is computed by multiplying the number of records specified in the BLOCK CONTAINS clause by the maximum record size (slack bytes and control fields included). When fixed-length records are written each physical record contains the number of records specified in the BLOCK CONTAINS clause. The last physical record may be short. No padding records are generated for short records. As many variable-length records as can fit into the buffer area are written, providing that there is sufficient room for a maximum-length record. For example, where the number of records is 6 and the maximum record size is 500, a 3,000-position buffer is provided. Records are located in the buffer until such time as less than 500 positions remain.

For Example:

1	2	3	4	5	6	7	
500	250	375	500	375	375	250	3000

Because the records occupy 2,625 positions of the buffer, and it is not known if the next record is greater than 375 positions, these seven records are written out as a 2,625-character block. Record eight is generated as the first record in an empty buffer. This means that the actual blocking is variable, depending on record size. Again, no padding records are provided.

This technique provides for good utilization of storage buffers in most cases. Efficiency is lost if a small blocking factor is specified and there is a large variablility in record size. For example, if a 'BLOCK CONTAINS 2000 CHARACTERS' clause is written with a maximum record size of 1,000 characters, the following situation could exist.

1	2	3	4	
250	250	250	300	2000

The four records total 1,050 characters, but since a 1,000-character maximum size must be anticipated, the 4-record 1,050-character block has to be written. Note that in any event, the records per block at least equal the number of records specified in the BLOCK CONTAINS clause.

APPLY WRITE ONLY

This clause permits the maximum use of a variable block.

When this clause is specified, the compiler checks each record, before it is written, to determine if the record can fit into the area remaining in the block. If it fits, the record is written into the block. If the record is too large to fit, the block is written out and a new one is started. Thus, use of the APPLY WRITE ONLY results in the maximum record size specified being ignored.

PROCESSING BUFFERS

Files can be processed using multiple buffers. Logical records are referenced in the proper block by adjusting registers (using them as pointers).

This technique eliminates the need for moving a record from the buffer area to a separate record work area, as well as the record work area itself. The record can be operated on directly in the buffer area.

When processing records in a buffer, the next read results in the previous record not being available. Because the previous record is no longer available, the technique of moving a high value to the control field of the last record (to force the processing of records remaining on the other file) cannot be used.

Here are several alternate approaches:

1. A GO TO statement, prior to the compare, can be altered during the AT END procedure to GO TO the low compare procedure, thus bypassing the compare.

2. A dummy record having a high value in its control field can be provided as the last logical record. This automatically causes the associated files to compare low. However, this can result in the AT END condition never occurring.

3. The control field can be moved to a separate work area following the read, and compared in the work area. The control field is then available in the work area following an AT END condition. The AT END procedure can move a high value into the control field.

VARIABLE RECORD ALIGNMENT CONTAINING OCCURS DEPENDING CLAUSE

Records are processed in the file's buffer area. The first record starts on a doubleword boundary. If there is no OCCURS DEPENDING clause, a diagnostic is given indicating the padding to be added to the record to assure proper alignment of succeeding records.

To align blocked V-type records containing an OCCURS DEPENDING clause in the buffer:

1. Determine the largest alignment factor within the record.

Alignment factor is	For
2	COMPUTATIONAL (1-4 digits)
4	COMPUTATIONAL-1 or COMPUTATIONAL (5-18 digits)
8	COMPUTATIONAL-2
0	OTHER

2. For alignment factors of four or less, pad both the fixed and the variable portions of the record to an even multiple of the alignment factor.

3. For an alignment factor of eight, move the record, as a group, to a 01 in the working storage section.

I/O PROGRAMMING CONSIDERATIONS

The following text discusses:

- Use after standard error considerations.
- Dummy record codes for direct organization files.
- The use of rewrite with random indexed sequential files.
- Considerations when updating or adding to a BISAM file.
- The DD requirements, and DCB parameters supplied (by the compiler) when using ACCEPT and DISPLAY verbs.
- The allocation of utility work space for the COBOL-E compiler.
- Labeling requirements for compiling and executing.
- The use of additional storage by the COBOL-E compiler.

USE AFTER STANDARD ERROR CONSIDERATIONS

When an uncorrectable I/O error occurs, the USE AFTER STANDARD ERROR declarative is entered, and general registers 14, 15, 0 and 1 are stored, respectively, in four words located at address: DCB - 20 bytes. A description of the contents of these registers can be found, under the appropriate access method, in the publication, IBM System/360 Operating System: Control Program Services. If a subprogram is called using file-name as a parameter, the address of the DCB is passed, and any action on the error can be taken at the user's discretion.

DUMMY RECORD CODES FOR DIRECT ORGANIZATION FILES

When reading direct files sequentially, dummy records are presented along with valid user records. They can be identified by the first byte of the symbolic key, which is hexadecimal FF.

THE USE OF REWRITE WITH RANDOM INDEXED SEQUENTIAL FILES

When using rewrite with random sequential files, each record read must be rewritten, otherwise the data set will be destroyed.

CONSIDERATIONS WHEN UPDATING OR ADDING TO A BISAM FILE

If a BISAM file is updated using a REWRITE statement, a REWRITE statement must be executed after every READ statement and prior to any other input/output statement on that file. A suggested sequence for reading a record in BISAM, inserting a different record into that file, and rewriting the updated original record is:

```
READ       record x
REWRITE    record x
WRITE      record y
READ       record x
REWRITE    record x
```

If the TRACK-AREA clause is not specified for BISAM, and a record is added to the file, the contents of the SYMBOLIC KEY is unpredictable after the WRITE is executed.

DD REQUIREMENTS AND DCB PARAMETERS FOR ACCEPT AND DISPLAY VERBS

Tables 14 and 15 indicate the DD requirements, and DCB parameters supplied when using the ACCEPT and DISPLAY verbs.

The parameters in Table 14 (and any other appropriate DCB parameters) can be specified by the user on his DD cards.

Table 14. Relationship of ACCEPT and DISPLAY Verbs and DD Card

VERB FORMAT	DD CARD REQUIRED
DISPLAY...UPON CONSOLE	No
DISPLAY...UPON SYSPUNCH	Yes
DISPLAY...(Default option is SYSOUT)	Yes
ACCEPT...FROM CONSOLE	No
ACCEPT... (Default option is SYSIN)	Yes

The DISPLAY and ACCEPT data set DCB parameters given in Table 15 are filled in by the COBOL-E compiler.

Those parameters not supplied by the compiler must be supplied by the user.

Table 15. Compiler Supplied Data Set DCB parameters for ACCEPT and DISPLAY verbs.

DCB PARM.	DISPLAY SYSPUNCH	DISPLAY SYSOUT	ACCEPT SYSIN
DSORG	PS	PS	PS
MACRF	PL	PL	GL
DDNAME	SYSPUNCH	SYSOUT	SYSIN
RECFM	F	U	U
LRECL	80	*	---
BLKSIZE	80	*	---
BFTEK	S	S	S

The parameters given in Table 15 are defined as follows:

PS - specifies a physical sequential organization
PL - specifies put-locate-mode operation
GL - specifies get-locate-mode operation
F - specifies fixed-length records
U - specifies undefined records
80 - for LRECL, specifies length in bytes of a format-F logical record
80 - for BLKSIZE, specifies the length of a block in bytes
S - specifies simple buffering
* - must be supplied in DD cards by user. The user must add an additional character for the purpose of forms control.

ALLOCATION OF UTILITY WORK SPACE

Table 16 is a guide to allocation of utility work space, and enables the programmer to specify a reasonable number of tracks for utility files without using storage unnecessarily.

Table 16. Track Allocation for Utility Work Space

The number of tracks required for:			
Device	150 Source Cards are	500 Source Cards are	1000 Source Cards are
2311	17	57	114

The programmer can estimate the total number of tracks required for utility work space by extrapolating from the figures given in Table 16. The number of tracks needed for each utility can then be specified in the SPACE parameter of the appropriate DD statement as follows:

SYSUT1 = 20% of the total number of tracks
SYSUT2 = 40% of the total number of tracks
SYSUT3 = 40% of the total number of tracks,

The SYSPUNCH and SYSOUT data sets on a direct-access device can be allocated space on a 10-track base with secondary allocation of 10-track increments, if needed.

The parameter can be specified on the DD card for the data set as follows:

```
SPACE=(TRK,(10,10))
```

LABELING FOR UTILITY WORK FILES

Labels for utility data sets are not required for compilation. The system OPEN routines process utility work files automatically. However, if labels are present, the programmer should be certain that they are specified in the appropriate utility DD statements. If they are not, compilation may be unsuccessful.

The compiler interprets the NO REWIND parameter of the OPEN clause as comments. When the file is opened at execution time, the operating system accesses the file as specified by the data set sequence number in the label parameter or the MOD parameter of the DD card.

At object time, labels are the complete responsibility of the user in that the same label status must be indicated in the DD cards as specified in the COBOL source program.

User labels are not supported by the control program for the initial release of COBOL-E.

USE OF ADDITIONAL STORAGE BY THE COBOL-E COMPILER

Additional storage is used by the compiler in one or both of two ways:

- to increase table space, and/or
- to increase buffer sizes

If the initial space allocated for tables was completely used, and additional storage is needed and available, the table space is dynamically increased by the compiler. The user cannot control the allocation of additional storage space for tables.

However, the programmer can specify buffer sizes for a specific compilation by using the BUFSIZE parameter in the EXEC statement. This temporarily overrides the system generated buffer sizes for the one compilation.

DEBUGGING TECHNIQUES

The DEBUG feature in the COBOL (E-Level Subset) Language allows the programmer to use three new verbs (as well as any other verb) for the purposes of debugging COBOL source programs. These verbs are TRACE, EXHIBIT, and ON. They can appear anywhere in the COBOL program or in a compile-time debugging packet. Their formats and a description of their use is contained in the publication, IBM System/360 Operating System: COBOL Language. However, this section is included in this publication to give the programmer an idea of when to use the debugging language, and how to construct a debugging packet, and what job-control statements are needed to use the debugging packet(s).

Appendix F contains a complete list of debugging packet error messages. These messages reflect errors in the debugging packet(s) only. They are not associated with compiling.

The TRACE and EXHIBIT clauses cause compiler generated "DISPLAY's". Therefore, a DD card for SYSOUT is required for specifying the logical output device.

TRACE

When a job fails to execute correctly and the diagnostic messages fail to indicate how to correct the error, a READY TRACE statement can be inserted at a point known to be prior to the trouble area. The TRACE displays each paragraph name as control passes into that paragraph. To reduce the volume of such a trace, it is possible to turn on the trace with a READY TRACE statement and turn if off with a RESET TRACE if the area can be localized. The TRACE function can be used any number of times within the program.

It would reduce the volume if RESET were issued upon entering a loop (containing a paragraph-name) and READY were issued upon leaving the loop.

It is sometimes difficult to determine what the specific path of program logic is. This is especially true with a series of PERFORMS or nested conditions. A TRACE statement can be very beneficial as an aid to this problem. Also, if values are inconsistent, a TRACE statement will again aid in determining whether or not a program is actually going through a certain point.

EXHIBIT

To find out what specifically caused the error within the paragraph, additional data can be obtained from the fields within the specific paragraph by use of the EXHIBIT statement. The EXHIBIT statement displays the field and the source name for

identification purposes. Its use can be restricted to display the field only if it has changed since the last time the program passed through that point. This permits the programmer to check on the value of the subscript name or other fields that are pertinent to a given field, and check out logic errors. An example of the various forms of this statement follows.

```
DATA DIVISION.
  77 NO-CHANGE-NAME PICTURE XX VALUE 'AB'.
  77 SUB-SCRIPT-NAME PICTURE S999
      COMPUTATIONAL VALUE 30.

PROCEDURE DIVISION.

TEST-LOOP.
  EXHIBIT NAMED NO-CHANGE-NAME.
  EXHIBIT CHANGED NAMED SUB-SCRIPT-NAME.
  EXHIBIT CHANGED SUB-SCRIPT-NAME.
  EXHIBIT CHANGED NO-CHANGE-NAME.
  ADD 10 TO SUB-SCRIPT-NAME.  IF
  SUB-SCRIPT-NAME = 100 NEXT SENTENCE ELSE
  GO TO TEST-LOOP.
```

The print out for this example is:

```
  NO-CHANGE-NAME = AB
  SUB-SCRIPT-NAME = 30
  30
  AB
  NO-CHANGE-NAME = AB
  SUB-SCRIPT-NAME = 40
  40
  NO-CHANGE-NAME = AB
  SUB-SCRIPT-NAME = 50
  50
  .
  .
  .
```

ON

It is possible, where large volumes of data are involved, to sample specific portions of a program by use of the ON statement. The ON statement allows the programmer to perform a series of operations at certain times when a program passes a particular point. For example, a series of operations could be performed the 110th time through a loop and every 5th time thereafter until the 275th time. This allows the programmer to determine whether or not a given loop gets out of the expected range for a particular program. There can be any number of these statements, and there is a five-digit compiler counter generated for each one. The counter starts as zero, and is increased by one each time the path of program execution falls through that specific point. For example, if the programmer knows that the error occurs on the 500th record processed, the ON statement may be used to count records. Then a READY TRACE can be set as the counter approaches the point at which the error occurred. This eliminates tracing each statement up to that point.

Note that this type of example could also have been done by a counter or a PERFORM, but this method is easier.

THE DEBUG PACKET

The debug packet is a tool used for debugging COBOL object modules. It is positioned in the job input stream before the COBOL source module. The packet is combined (merged) with the COBOL source module before compilation begins. Where the packet is positioned within the COBOL source module is determined by the procedure division name specified in the *DEBUG card of the packet.

Job Control Setup for Using Debug Packets

Debug packets for a given compilation are processed, as separate job steps, immediately preceding the job step that executes the COBOL compiler program. Figure 31 contains the job-control statements, and data sets needed to use debug packets.

```
//PACKCBL   JOB   1234,BROWN,MSGLEVEL=1
//DEBUG     EXEC  PGM=IEPDBG00
//SYSPRINT  DD    SYSOUT=A
//SYSUT1    DD    UNIT=SYSDA,SPACE=(TRK,(10,10))
//SYSUT4    DD    DSNAME=&COMPSET,DISP=(NEW,PASS),                    1
//                UNIT=SYSDA,SPACE=(CYL,(10,10))
//SYSIN     DD    *

            DEBUG Packet(s)
            Source Modules

            {COBOL Source Module}

/*
//COMPILG   EXEC  PROC=COBECLG
//COB.SYSIN DD    DSNAME=&COMPSET,DISP=(OLD,DELETE)
            DD cards for the execute step of the procedure if
            needed.
```

Figure 31. Deck Setup For Debug Packet

The functions of the job-control statements and the data sets needed to employ the debug packet are as follows.

The //DEBUG statement specifies that the debug packet processing program be executed.

The //SYSPRINT DD statement specifies the data set for DEBUG packet diagnostics.

The //SYSUT1 DD statement specifies a work data set on which the DEBUG packets are stored for future merging into the procedure division of the COBOL source module.

The //SYSUT4 DD statement specifies a work data set on which the COBOL source module is written, with the DEBUG packets properly inserted in the procedure division. This data set then becomes SYSIN, the input data set to the compiler.

The //COMPILG step specifies that the cataloged procedure to compile, linkage edit, and execute be executed.

The //COB.SYSIN DD statement specifies that the input data set (SYSIN) to the compiler by SYSUT4 data set of the DEBUG step.

Any diagnostic messages generated during the DEBUG step appear on the listing preceding the source listing produced by the compiler. Refer to Appendix E for a list of the debug diagnostic messages.

COBOL SOURCE PROGRAM LIBRARY

Incorporated in the COBOL language are clauses for utilizing the source program library facility.

Prewritten source program entries in a user-created library can be included in a COBOL program at compile time. Thus, standard file descriptions, record descriptions, or procedures can be used without having to restate them. They are included in a source statement program by means of a COPY or INCLUDE clause.

To catalog or update a source program in a user-created library, a utility program must be used. Following are examples of:

- Cataloging some source statements to a user-created library, and what happens when they are retrieved. Included in this example is the job-control statement for automatically sequence-numbering the source statements cataloged.
- Updating an existing member of the user-created library.

EXAMPLE OF CATALOGING SOURCE PROGRAM STATEMENTS TO A LIBRARY

The job-control statements to catalog source statements to the source statement library are:

```
//CATALOG JOB
//        EXEC  PGM=IEBUPDAT,                X
//              PARM=(NEW)
//SYSUT2  DD    DSNAME=COBOLLIB,             X
//              UNIT=2311,                   X
//              DISP=(NEW,KEEP),             X
//              VOLUME=SER=111111,           X
//              SPACE=(TRK,(15,10,10)),      X
//              DCB=(,RECFM=F,               X
//              BLKSIZE=80)
//SYSPRINT      DD    SYSOUT=A
//SYSIN         DD    *
./      ADD     CFILEA,01,1,1
                BLOCK CONTAINS 13 RECORDS
                RECORD CONTAINS
                120 CHARACTERS
./      NUMBR   00000000,00000000,
                00000010,00000010
./      ENDUP
/*
```

In this example, IEBUPDAT is the name of the IBM-supplied utility program that accomplishes the cataloging. These statements are copied in an FD entry. The library entry does not include either FD or the file-name, but instead begins with the first clause following the file-name.

The NUMBR statement in this procedure results in the source statements being automatically sequence numbered. The first source statement will be numbered 00000010, and each succeeding statement a number incremented by 00000010.

The same procedure can be used to catalog entire source programs, if desired.

Note: At compile time, the data set containing the cataloged source statements, must be assigned to SYSLIB. (In the example given, this data set DSNAME is COBOLLIB.)

COPY (DATA DIVISION)

The COBOL COPY clause permits the user to include prewritten data-division entries or environment-division clauses in this source program at compile time. An example illustrating what actually gets copied when the cataloged entry 'CFILEA' is retrieved from the user-created source program library follows.

Assume the following source COBOL statement is written:

```
  FD FILEA COPY 'CFILEA'.
```

COPY 'CFILEA' is replaced by the actual entries i.e., BLOCK CONTAINS 13 RECORDS, etc. within the compiler for compilation purposes.

The output listing would show the following:

```
FD FILEA COPY 'CFILEA'
*           BLOCK CONTAINS 13 RECORDS
*           RECORD CONTAINS 120 CHARACTERS.
```

Internally (to the compiler) the output would look like:

```
FD FILEA BLOCK CONTAINS 13 RECORDS
RECORD CONTAINS 120 CHARACTERS.
```

The source statement referencing the user-created library is followed by the

actual library entries, except for data entries which have a duplicate level number and data-name. Explicitly, CFILEA identifies the entries actually recorded in the library. This is the library name. It is the header record required for identification of the entries, and is not itself retrieved (not copied internally by the compiler).

All entries associated with the library name are copied.

In the case of data entries which have a duplicate level number and data-name, the following results are obtained when issuing a COBOL COPY statement.

Assume the job-control and COBOL statements written to catalog a file are:

```
//CATALOG   JOB
//         EXEC PGM=IEBUPDAT,PARM=(NEW)
//SYSUT2    DD   DSNAME=COBOLLIB,          X
//               UNIT=2311,                X
//               DISP=(NEW,KEEP),          X
//               VOLUME=SER=111111,        X
//               SPACE=(TRK,(15,10,10)), X
//               DCB=(,RECFM=F,BLKSIZE=80)
//SYSPRINT  DD   SYSOUT=A
//SYSIN     DD   *
./      ADD XFILEY,01,1,1
            01 PAYFILE USAGE IS DISPLAY.
               02 CALC PICTURE 99.
               02 GRADE PICTURE 9 OCCURS 1
                     DEPENDING ON CALC OF
                     PAYFILE.
./      ENDUP
/*
```

and, the source COBOL statement written is:

```
  01 GROSS COPY 'XFILEY'.
```

On the output listing, the statements would look like:

```
  01 GROSS COPY 'XFILEY'.
  01 PAYFILE USAGE IS DISPLAY.
 *      02 CALC PICTURE 99.
 *      02 GRADE PICTURE 9 OCCURS 1
 *         DEPENDING ON CALC OF PAYFILE.
```

Internally (within the compiler), the statements would look like:

```
01 GROSS USAGE IS DISPLAY.
  02 CALC PICTURE 99.
  02 GRADE PICTURE 9 OCCURS 1
       DEPENDING ON CALC OF PAYFILE.
```

INCLUDE (PROCEDURE DIVISION)

The procedure for copying from the user-created source program library from within the procedure division is the same as that described for the data division. The results are identical.

Assume a procedure named PROCESS is in the user-created source program library, and was cataloged as follows.

```
//CATALOG   JOB
//         EXEC PGM=IEBUPDAT,PARM=(NEW)
//SYSUT2    DD   DSNAME=COBOLLIB,          X
//               UNIT=2311,                X
//               DISP=(NEW,KEEP),          X
//               VOLUME-SER=111111,        X
//               SPACE=(TRK,(15,10,10)), X
//               DCB=(,RECFM=F,            X
//               BLKSIZE=80)
//SYSPRINT  DD   SYSOUT=A
//SYSIN     DD   *
./      ADD PROCESS,01,1,1
            COMPUTE QTY-ON-HAND =
            TOTAL-USED-NUMBER-ON-HAND.
./      ENDUP
/*
```

To retrieve catalog entry PROCESS, write:

```
  Paragraph-name.  INCLUDE 'PROCESS'.
```

It is the user's responsibility to supply the name for paragraph-name.

UPDATING AN EXISTING MEMBER OF A USER-CREATED LIBRARY

Assume a member called CFILEA is cataloged to a user-created library. The following is an example of a procedure for updating this member.

```
//UPDATE    JOB
//         EXEC PGM=IEBUPDAT,PARM=(MOD)
//SYSUT1    DD   DSNAME=COBOLLIB,          X
//               UNIT=2311,                X
//               DISP=(NEW,KEEP),          X
//               VOLUME=SER=111111,        X
//               SPACE=(TRK,(15,10,10)),   X
//               DCB=(,RECFM=F,            X
//               BLKSIZE=80)
//SYSUT2    DD   DSNAME=COBOLLIB,          X
//               UNIT=2311,                X
//               DISP=(NEW,KEEP),          X
//               VOLUME=SER=222222,        X
//               SPACE=(TRK,(15,10,10)),   X
//               DCB=(,RECFM=F,BLKSIZE=80)
//SYSPRINT DD   SYSOUT=A
//SYSIN     DD   *
./      CHNGE CFILEA,01,1,1
           BLOCK CONTAINS
           20 RECORDS ......00000010
./      REPRO XFILEY,01,1,1
./      ENDUP
/*
```

To update a member of an existing library, another library is built. Thus (as illustrated in the procedure) CFILEA is

altered. Note that XFILEY is also copied into the new library in its entirety.

The programmer should be sure to supply the appropriate sequence number (in columns 73-80) for the member of the library that is changed. In the example, 00000010 is the sequence number supplied for the statement: BLOCK CONTAINS 20 RECORDS. It is assumed to be positioned in columns 73-80.

The compiler, linkage editor, and load modules produce aids that can be used to document and debug programs. This section describes the listings, maps, card decks, and error messages produced by these components of the operating system.

COMPILER OUTPUT

The compiler can generate a listing of source statements, a storage map, and an object module card deck. Source module diagnostic messages are also produced during compilation.

SOURCE LISTING (LIST)

A description of a source module listing follows. The listing is obtained on SYSPRINT when LIST is specified in the PARM parameter of the EXEC statement. The header line printed across the top of the source listing is the first line on the first page of the listing.

PRINT POSITIONS	PRINTED INFORMATION
1-13	LEVEL: NMMMYY (where N = distribution number, MMMY = month and year of distribution)
Positioned centrally in print line	Official name and design point of compiler (COBOL-E)
106 - 117	DATE: YY.DDD (where YY = year and DDD = day)

Figure 32 is a skeleton example of a COBOL source listing. The associated job control cards are given in Figure 42 of Appendix A.

```
                                        COBOL E                    DATE 66.021
  1  000000 IDENTIFICATION DIVISION.
  2  000010 PROGRAM-ID.   'TCECAP08'.
  3  000020 REMARKS.
                .
                .
                .
                .        In this example,
              Lines 4 through 34 were comments
                .
                .
                .
                .
 35  000360 ENVIRONMENT DIVISION.
 36  000370 CONFIGURATION SECTION.
 37  000380 SOURCE-COMPUTER. IBM-360 E50.
 38  000390 OBJECT-COMPUTER. IBM-360 E50.
 39  000400 INPUT-OUTPUT SECTION.
 40  000410 FILE-CONTROL.
 41  000420     SELECT EQUAL-HETROGEN-FILE ASSIGN TO
 42  000430     'DATASET1'
 43  000440     UTILITY.
 44  000450     SELECT EQUAL-HETROGEN-FILE-2 ASSIGN TO
 45  000460     'DATASET1'
 46  000470     UTILITY.
 47  000480 DATA DIVISION.
 48  000490 FILE SECTION.
 49  000500 FD  EQUAL-HETROGEN-FILE
S50  000210     BLOCK CONTAINS 800 CHARACTERS
 51  000520     RECORD CONTAINS 80 CHARACTERS
 52  000530     RECORDING MODE IS F
 53  000540     LABEL RECORDS ARE STANDARD
 54  000550     DATA RECORDS ARE CHCK-RECORD-11 CHCK-RECORD-21.
 55  000560 01  CHCK-RECORD-11.
 56  000570   02 NAME1 PICTURE IS A(12).
 57  000580   02 ADDRESS1 PICTURE IS X(10).
 58  000590   02 FILLER PICTURE IS X(5).
                .
                .
                .
                .
                .
                .
                .
           User written source statements
                .
                .
                .
                .
 97  000980 01  CHCK-RECORD-21.
 98  000990    02 FILLER    PICTURE IS AA.
 99  001000    02 DEPT      PICTURE IS X(3).
100  001010    02 FILLER    PICTURE IS 9(11).
101  001020    02 MAN-NO    PICTURE IS 9(6).
102  001030    02 DATE-HIER PICTURE IS X(10).
103  001040    02 FILLER    PICTURE IS X(45).
104  001050    02 STATE     PICTURE IS 99.
105  001060       88 PENNA  VALUE   IS 10.
106  001070       88 NEW-YORK VALUE IS 15.
107  001080    02 ID-CODE   PICTURE IS 9.
108  001090       88 RECORD-2 VALUE IS 2.
109  001100 WORKING-STORAGE SECTION.
110  001110 77  WRITE-COUNTER
```

Figure 32. Example of COBOL Source Listing (Part 1 of 2)

```
111  001120          PICTURE IS 99
112  001130          VALUE IS 00
113  001140          USAGE IS DISPLAY.
114  001150 77  READ-COUNTER
115  001160          PICTURE IS 99
116  001170          VALUE IS 00
117  001180          USAGE IS DISPLAY.
118  001190 77  GOOD-COMP-COUNTER
                .
                .
                .
                .
                .
                .
                .
           User written source statements
                .
                .
                .
141  001420    02 COMP-DEPT
142  001430          PICTURE IS X(3)
143  001440          VALUE IS 'D43'
144  001450          USAGE IS DISPLAY.
145  001460    02 COMP-MAN-NO
146  001470          PICTURE IS 9(6)
147  001480          VALUE IS 960640
148  001490          USAGE IS DISPLAY.
149  001500 PROCEDURE DIVISION.
150  001510 DISPLAY-HEADER.
151  001520      DISPLAY '           GROUP A LEVEL P TEST CASE 8 '.
152  001530      DISPLAY ' '.
153  001540      OPEN OUTPUT EQUAL-HETROGEN-FILE.
154  001550 START1.
155  001560      MOVE COMP-NAME TO NAME1.
156  001570      MOVE BORN-DATE TO BIRTH-DATE1.
157  001580      MOVE 1 to IDENT1.
158  001590 WRITE-REC1.
159  001600      WRITE CHCK-RECORD-11.
160  001610      ADD 1 TO WRITE-COUNTER.
161  001620 START2.
162  001630      MOVE COMP-DEPT TO DEPT1.
163  001640      MOVE COMP-MAN-NO TO MAN-NO1.
164  001650      MOVE 2 TO ID-CODE1.
                .
                .
                .
                .
                .
                .
                .
           User written source statements
                .
                .
239  002410      DISPLAY 'WRITE-COUNTER = ' WRITE-COUNTER.
240  002420      DISPLAY 'SHOULD BE = 20 '.
241  002430      DISPLAY ' READ-COUNTER = '  READ-COUNTER.
242  002440      DISPLAY ' SHOULD BE = 21 '.
243  002450      DISPLAY ' ERROR-COMP-COUNTER = ' ERROR-COMP-COUNTER.
244  002460      DISPLAY ' SHOULD BE = 00 '.
245  002470      DISPLAY ' GOOD-COMP-COUNTER =  ' GOOD-COMP-COUNTER.
246  002480      DISPLAY ' SHOULD BE = 20 '.
247  002490      STOP RUN.
```

Figure 32. Example of COBOL Source Listing (Part 2 of 2)

The components of a source listing are:

1. A compiler generated line number which is shown in the leftmost columns followed by the source card image. The compiler generated line number is used in diagnostic and PMAP references.

2. All COBOL words, punctuation, and other groups of characters on each line are referenced as elements on the line in diagnostics and PMAP listings.

3. Sequence numbers out of order. If columns 1-6 of the source statement are not blank, they are sequence checked. The character "S" is placed to the left of a compiler-generated line number when a source sequence number is not in logical ascending order.
   Example: Assume that a statement numbered 50 (refer to Figure 32) was out of sequence. The compiler would list the source statement as:

   ```
   S50      BLOCK CONTAINS 800 CHARACTERS.
   ```

4. Library cards. Cards coming from the source statement library as a result of a COPY or INCLUDE statement are noted with the character "*", which is printed to the right of the compiler-generated line number.

STORAGE MAP

The storage map consists of a data map (DMAP), and a procedure map (PMAP).

Data Map (DMAP)

A data map for a source listing is obtained when DMAP is specified in the PARM parameter of the EXEC statement. The data map is output by SYSPRINT.

Figure 33 is an example of a data map. It is a portion of the data map generated for Figure 32.

This listing shows each non-procedure name defined in the program and its relative address. File-names, record-names, and condition-names are identified in the name column. The relative location of each entry is shown (column headed LOCATION). Linkage and file entries are relative to the 01 or 77. Working storage is relative to the load point for the program. The relative addresses are expressed as hexadecimal numbers.

TYPE	LOCATION	DATA NAME
FILE		EQUAL-HETROGEN-FILE
REC	000000	CHCK-RECORD-11
	000000	NAME1
	00000C	ADDRESS1
	00001B	WIFE1
	000021	BIRTH-DATE1
	000021	DAY1
	000023	MONTH1
	000025	YEAR1
	00004F	IDENT1
COND		REC11
REC	000000	CHCK-RECORD-21
	000002	DEPT1
	000010	MAN-NO1
	000016	DATE-HIER1
	00004D	STATE1
COND		PENNA1
COND		NEW-YORK1
	00004F	ID-CODE1
COND		RECORD-21
FILE		EQUAL-HETROGEN-FILE-2

Figure 33. Example of Data Map Generated for a COBOL Program

Procedure Map (PMAP)

A procedure map is obtained when PMAP is specified in the PARM parameter of the EXEC card. The details of PMAP are given for their debugging value to a programmer. Figure 34 is an example of a procedure map. It is a portion of the procedure map generated for Figure 32.

LINE/POS - Contains the statement line number, and position of the COBOL verb on the line. These numbers are decimal numbers. The actual instruction(s) used to accomplish the COBOL statement is identified by the compiler-generated internal line number(s). If more than one instruction was generated, the compiler-generated line number for that COBOL statement would be repeated for each instruction listed.

The line counter cannot exceed 4095. At this point it resets to zero.

ADDR - Contains the relative address of each instruction in the procedure division in hexadecimal. The addresses are relative to the program's load point.

INSTRUCTION - Contains the actual instruction generated for the COBOL statement.

Note line number 00167-01 and 00167-03. Line number 00167-01 refers to a verb which is the first item on the line, whereas 00167-03 refers to a verb which is the third item on the line.

LINE/POS	ADDR	INSTRUCTION
00165 01	000564	58 F0 3 234 0 5EF
00166 01	00056A	41 10 4 050
00166 01	00056E	58 F0 1 030
00166 01	000572	05 EF
00166 01	000574	18 51
00167 01	000576	58 F0 3 234 0 5EF
00167 03	00057C	F2 F1 3 168 4 000
00167 03	000582	FA F0 3 168 4 142
00167 03	000588	F3 1F 4 000 3 168
00167 03	00058E	96 F0 4 001
00168 01	000592	58 F0 3 234 0 5EF

Figure 34. Example of a Procedure Map Generated for a COBOL Program

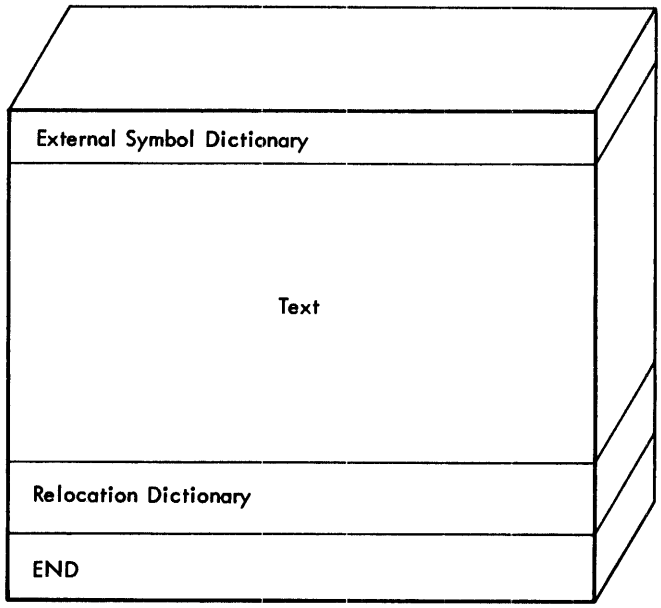


Figure 35. Example of an Object Module

OBJECT MODULE CARD DECK

An object module card deck is produced unless NODECK is specified in the PARM parameter of the EXEC statement.

An object module, the output of a COBOL (E) execution, consists of control dictionaries and text (instructions and data). The control dictionaries contain the information necessary to resolve cross-references between control sections and modules. Figure 35 illustrates the contents of an object module.

The COBOL-E compiler also produces an END statement that marks the end of the object module. The deck is made up of four types of cards: TXT, RLD, ESD and END. A functional description of these cards is given in the following paragraphs.

Object Module Cards

Every card in the object module deck contains a 12-2-9 punch in column 1 and an identifier in columns 2 through 4. The identifier consists of the characters ESD, RLD, TXT or END. The first four characters of the name of the program are placed in columns 73 through 76 with the sequence number of the card in columns 77-80.

EXTERNAL SYMBOL DICTIONARY: The external symbol dictionary contains entries for all external symbols defined or referred to within a module. (An external symbol is one that is defined in one module so that it can be referred to in another.) Each entry identifies a symbol, or a symbol reference, and gives its location, if any, within the module.

Three types of ESD (external symbol dictionary) cards are generated as follows:

ESD, type 0 - contains the name of the compiled control section and indicates its compiled origin.

ESD, type 1 - contains the name of a secondary entry point within a control section. These ESD's result from COBOL ENTRY statements.

ESD, type 2 - contains the names of subprograms referred to by CALL statements, and names of COBOL object time subroutines to be linkage edited with the compiled control section.

The type number: 0, 1 or 2 is placed in card column 25.

RELOCATION DICTIONARY: The relocation dictionary lists all relocatable address constants that must be modified when the linkage editor produces an output load module. The RLD is used to adjust the value of address constants. The RLD contains at least one entry for every relocatable address constant in a module. An RLD entry identifies an address constant by indicating its location within a control section and the external symbol (in the ESD) whose value must be used to compute the value of the address constant.

An RLD (relocation dictionary) card is generated for external references indicated in ESD, type 2 cards. When the linkage editor has resolved external references, the address constant at the address indicated in the RLD card contains the relative address assigned to the subprogram indicated in the ESD, type 2 card. RLD cards are also generated for branching and subroutine linkage.

TXT Card: The TXT card contains the literals used by the programmer in his source module, and any literals generated by the compiler, coded information for DISPLAY statements, and machine instructions generated by the compiler from the source module.

END Card: One END card is generated for each compiled source module. This card indicates the end of the object module to the linkage editor. It also contains the entry point of the object module.

OBJECT MODULE DECK STRUCTURE: Figure 36 illustrates the COBOL object module deck structure.

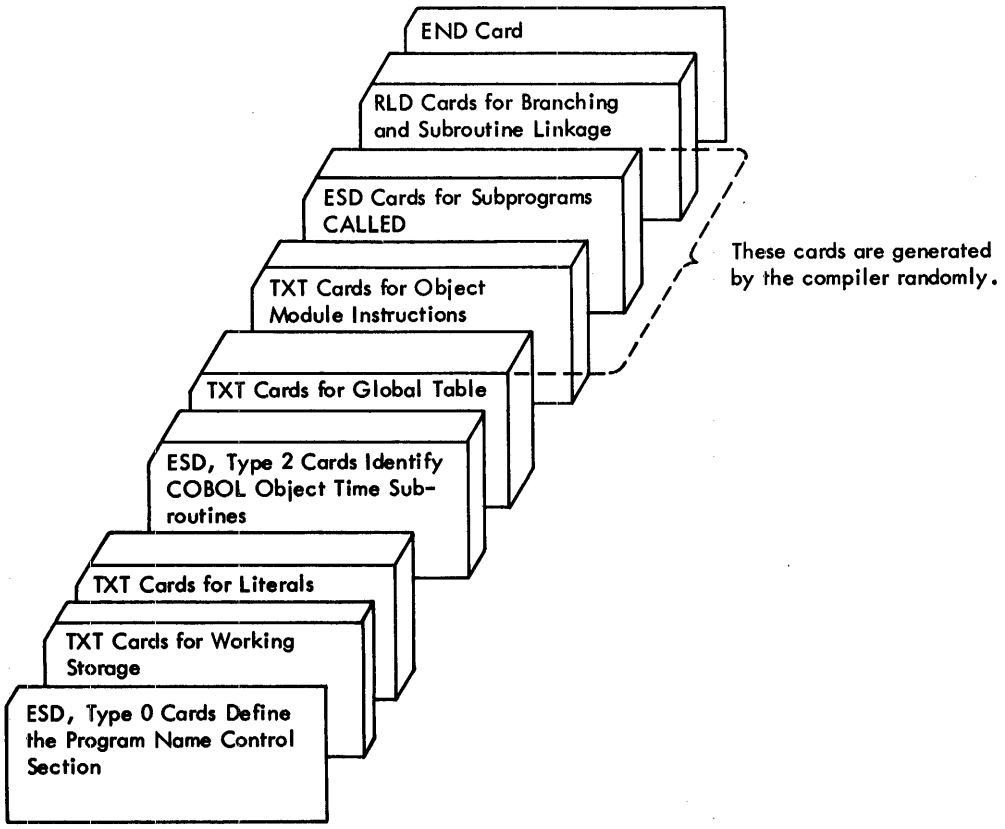


Figure 36. COBOL Object Module Deck Structure

SOURCE MODULE DIAGNOSTICS

Two types of diagnostic messages are written by the compiler: error and error-warning.

When the FLAGE option is specified in the PARM parameter of the EXEC card, the compiler will not generate error warning diagnostic messages.

When the FLAGW option is specified in the EXEC card, the compiler generates messages for actual errors, plus warning diagnostic messages.

Source Module Error-Warning Messages

All error-warning messages produced are written in a group following the source module listing and storage map. Figure 37 shows the format of each message as it is written on the data set specified by the SYSPRINT DD statement.

These diagnostics were generated by the compiler for the program shown in Figure 32. For a complete list, and descriptions of the error messages refer to Appendix F.

DIAGNOSTICS

LINE/POS	ER CODE	CLAUSE	MESSAGE
129-1	IEP051W	ALIGNMENT	FOR PROPER ALIGNMENT, A 5 BYTE LONG FILLER ENTRY IS INSERTED PRECEDING ASTR.
132-1	IEP05IW	ALIGNMENT	FOR PROPER ALIGNMENT, A 5 BYTE LONG FILLER ENTRY IS INSERTED PRECEDING DATA-INFO.

Figure 37. Example of Source Module Diagnostics

LINE/POS - Contains the internal line numbers of the source statements, and the element position of the COBOL verb on the line where the error was detected. When the compiler cannot locate the item in error on the line, it only identifies the line at fault. When the compiler generates the line number 0-0, it is referring to an entire section (the section may be missing).

ER CODE - Contains a message number and the severity level of the error:

MESSAGE NUMBER - The format of the message number, and the associated message is described in Appendix F.

Severity Code.
W = WARNING
C = CONDITIONAL
E = ERROR

W WARNING - Your attention is called to a condition that can cause a problem, but should permit a successful run.

C CONDITIONAL - The error statement is dropped or corrective action is taken. The compilation is continued as it may have debugging value, but the programs should not execute as intended.

E ERROR - This condition seriously affects execution of the job. Execution should not be attempted.

CLAUSE - This column identifies either the particular COBOL clause being processed at the time the diagnostic was discovered or the basic area that was involved, such as ALIGNMENT, FD, I-O CONTROL, or similar items.

MESSAGE - The actual message is given here. For specific details of these messages, refer to Appendix F.

Working with Diagnostics

1. Handle the diagnostics in the order in which they appear on the source listing. It is possible to get compound diagnostics. Frequently, an earlier diagnostic indicates the reason for a later diagnostic. For example, a messing quote for an alphabetic or alphameric literal could involve the inclusion of some clauses not intended in that particular literal. This could cause some apparently valid clause to be diagnosed as invalid because it is not complete, or is in conflict with something that preceded it.

2. Check for missing or extra punctuation, or other errors of this type.

3. Frequently, a seemingly meaningless message is clarified when the valid syntax or reference format is referenced. Diagnostics are coded directly from the reference format and are designed for use in conjunction with the particular type of reference. (See Appendix D.)

How Diagnostic Messages Are Determined

The compiler scans the statement element by element to determine whether the words are combined in a meaningful manner. Based upon the elements that have already been scanned, there are only certain words or elements that can be correctly encountered.

If the anticipated elements are not encountered, a diagnostic message is produced. Some errors may not be uncovered until information from various sections of the program are combined and the inconsistency indicated. Diagnostics uncovered in this manner can produce a slightly different format than those uncovered when the actual source text is still available. The message that is made unique through that particular error may not have, for example, the actual source statement that produced the error. The position and sequence reference, however, indicates the place at which the error was uncovered.

Errors appearing to be identical are diagnosed in a slightly different manner, depending on where they were encountered by the compiler and how they fit within the context of valid syntax. For example, a period missing from the end of the working-storage section clause, is diagnosed specifically as a period required. There is no other information that can occur at that point. However, if at the end of a record description entry, an element is encountered that is not valid at that point, such as the digits 02, they are diagnosed as invalid. Any clauses associated with the clause at that entry, that conflict with the entries in the previous entry (the one that had the missing period), are diagnosed. Thus, a missing period produces a different type of diagnostic in one case than in another.

If a given compilation produces more than 25 diagnostic messages, they are presented in a batched sequence. The first 25 messages are sorted in order, followed by the second series, which is also sorted in order.

If an error occurs after the 4095 source statement the line sequence number of the source statement in error can usually be determined by adding 4095 to the sequence number given in the diagnostic message. A message frequently suggests the division of a COBOL source program in which the error occurred.

Examples of How Diagnostics Are Generated

Each message has a general or skeleton form. Unique words for each message are inserted to identify the specific error that was encountered. The following two examples illustrate this form.

Example 1:

COBOL format is

```
MOVE data-name TO data-name...
     literal
```

```
Error 1        MOVE FIELDA TOO FIELDB
023
               ERROR #178

                                Information
               INSERT1 TO       passed to
                                diagnostic
               INSERT2 TOO      out of phase.
```

```
Skeleton Message #178 E SYNTAX REQUIRES
WORD "Insert1".  FOUND "Insert2".
```

```
Message appears as:  23-3 IEP178 E SYNTAX
REQUIRES WORD "TO".  FOUND "TOO".
```

Example 2:

```
Error 2
023            NOVE FIELDA TO FIELDB
               ERROR #549
               INSERT1 NOVE
```

```
Skeleton message #549 E WORD 'Insert 1' WAS
EITHER INVALID OR SKIPPED DUE TO ANOTHER
DIAGNOSTIC.
```

```
Message appears as:  23-1 IEP549E "NOVE"
UNHANDLED.  WORD NOVE WAS EITHER INVALID OR
SKIPPED DUE TO ANOTHER DIAGNOSTIC.
```

LINKAGE EDITOR OUTPUT

The linkage editor produces a map of a load module (module map), or cross-reference list and a module map when the MAP or XREF options, respectively, are specified in the PARM parameter of the EXEC statement. The linkage editor also produces diagnostic messages. For a complete list of linkage editor diagnostics, refer to the

publication IBM System/360 Operating System: Linkage Editor.

MODULE MAP

The module map is written on the data set specified in the DD statement called SYSPRINT. The module map is a listing of the control sections processed by the linkage editor.

Each control section is listed giving:

- Its name, origin, and length. The name is the program ID. The origin and length of the control section are listed in hexadecimal numbers.
- Any entry points within the control section and their locations.

Also listed are:

- Any functions called from the data set specified by the SYSLIB DD statement. These functions are subprograms included in the main program by automatic library calls, and are identified by asterisks.
- A segment number for each control section in an overlay structure.
- The total length and entry point for the load module.

Figure 38 is an example of a load module map. It is the load module map for the program shown in Figure 32. The map contains a main program (TCECAP08, the PROGRAM ID.) and four subprograms, each of which is a control section: IEP02300, IEP02800, IEP03000, and IEP00400. The asterisks following the names of these control sections indicate that the linkage editor obtained them from the automatic call library for the purpose of resolving references. The origin of the main program (TCECAP08) is the relative address 00 and its length is BB0. The entry point for the main program is at 490. The origin of subprogram IEP02300 is BB0 and its length is 21E (all numbers are hexadecimal numbers).

The entry points within a subprogram are listed in a string under the heading ENTRY. The entry points within subprogram IEP02300 are IEP02301 at location BB0, IEP02302 at location CAC, RETURN at location CAC, and IEP02304 at location CDE.

CONTROL NAME	SECTION ORIGIN	LENGTH	ENTRY NAME	LOCATION	NAME	LOCATION	NAME	LOCATION	NAME	LOCATION
TCECAP08	00	BB0								
IEP02300*	BB0	21E	IEP02301	BB0	IEP02302	CAC	RETURN	CAC	IEP02304	CDE
IEP02800*	DD0	19C	IEP02801	DD0	IEP02802	E46				
IEP03000*	F70	3C	IEP03001	F70						
IEP00400*	FD0	2A4	IEP00401	FB0						
ENTRY ADDRESS		490								
TOTAL LENGTH		1254								

Figure 38. Example of a Module Map

CROSS-REFERENCE TABLE

The cross-reference table is written along with the module map, when the option XREF is specified. It lists the location from which an external reference is made, the symbol externally referenced, the control section in which the symbol appears, and the segment number of the control section in which the symbol appears. The cross-reference table appears after the module map for all control sections, unless the linkage editor is building an overlay structure. Figure 39 is an example of a cross-reference table. It is the cross-reference table for Figure 32.

Location 478, in the cross-reference table, is the address where the CALL or reference is made to subprogram entry point

IEP02301 given in the REFERS TO SYMBOL column.

REFERS TO SYMBOL column lists all references to the entry points of each subprogram, and from one control section to another, for the entire load module (which is the entire program).

The control sections that contain the entry points referenced are listed in the column labeled IN CONTROL SECTION. Thus, entry point IEP02301 resides in control section IEP02300.

The ENTRY ADDRESS of the load module is 490, and its entire length, including all subprograms, is 1,254 hexadecimal bytes.

CROSS REFERENCE TABLE

CONTROL SECTION NAME	ORIGIN	LENGTH	ENTRY NAME	LOCATION	NAME	LOCATION	NAME	LOCATION	NAME	LOCATION
TCECAP08	00	BB0								
IEP02300*	BB0	21E	IEP02301	BB0	IEP02302	CAC	RETURN	CAC	IEP02304	CDE
IEP02800*	DD0	19C	IEP02801	DD0	IEP02802	E46				
IEP03000*	F70	3C	IEP03001	F70						
IEP00400*	FB0	2A4	IEP00401	FB0						

LOCATION	REFERS TO SYMBOL	IN CONTROL SECTION
478	IEP02301	IEP02300
47C	IEP02302	IEP02300
480	IEP02304	IEP02300
484	IEP02801	IEP02800
488	IEP02802	IEP02800
480	IEP03001	IEP03000
CF4	IEP00401	IEP00400
ENTRY ADDRESS	490	
TOTAL LENGTH	1254	

Figure 39. Example of a Cross-Reference Table

LOAD MODULE OUTPUT

The programmer defines the output data sets for load module execution through the appropriate source module statements and corresponding DD statements. The environment and data division statements define the data set. The WRITE and DISPLAY verbs in the procedure division generate the creation of the data set. Two types of messages can be generated from a load module: object time messages and operator messages.

OBJECT TIME MESSAGES

When an error condition that is recognized by compiler generated code occurs during execution, an error message is written on the CONSOLE typewriter. These messages and their descriptions are contained in Appendix F.

OPERATOR MESSAGES

A message is transmitted to the operator when a STOP 'literal' or an ACCEPT ... FROM CONSOLE source statement is executed. The messages are written on the console typewriter. Refer to Appendix F for a description of these messages.

OBJECT PROGRAM DUMPS

An object program can dump before normal termination of a procedure. A dump could be caused by any of the errors listed here. Several of these errors can occur at the COBOL language level while others can occur at the job-control level.

Typical Source Program Errors Initiating Dumps at Execution Time

A dump can occur at the COBOL language level for the following reasons.

1. A GO TO statement with no procedure name following it may not have been properly initialized with an ALTER statement. The execution of this statement would cause an invalid branch.
2. Performing arithmetics or moves on numeric fields that have not been properly initialized could cause an interrupt and a dump. For example, neglecting to initialize an OCCURS DEPENDING ON name, or referencing data fields prior to the first read.
3. Invalid data in a numeric field resulting from redefinition.
4. Input/output errors that are nonrecoverable.
5. Destroying a machine instruction in the program could move data fields into the procedure division. This could happen, for example, by using a subscript whose value exceeds its defined maximum.
6. Attempting to execute an invalid operation code through a systems error or invalid program.
7. Generating an invalid address to an area that has address protection.
8. Subprogram linkage declarations that are not defined exactly as they are stated in the calling program.
9. Data or instructions can be modified by entering a subprogram and manipulating data incorrectly. A COBOL subprogram could acquire invalid information from the main program; that is, a CALL using a procedure-name and an ENTRY using a data-name.
10. There is no conversion, alignment, or error checking of incoming data associated with the clause ACCEPT FROM CONSOLE. Any assumptions made by the programmer concerning these functions could result in the initiation of a dump.
11. Data records must be 80 characters in length for files in the input stream when the input device is a disk or tape unit.
12. An input file contains invalid data such as a blank numeric field or data incorrectly specified by its data description.

    The compiler does not generate a test to check the sign position for a valid configuration before the item is used as an operand. The programmer can test for valid data by means of the numeric class test and, by use of the TRANSFORM statement, convert it to valid data under certain circumstances.

    For example, if the units position of a numeric data item described as USAGE IS DISPLAY contained a blank, the blank could be transformed to a zero, thus forcing a valid sign.

Abnormal Termination Dumps

The control program prints an abnormal termination dump if a task is abnormally terminated, and a DD statement with a data definition name of SYSABEND in the name field was specified.

The abnormal termination dump is written in the SYSABEND data set. The details for specifying the abnormal termination dump are given in the section Job Processing. This data set can be on a printer, so that the dump is printed as it is produced, or on any other type of device, so that the dump can be printed later.

Figure 40 gives the format of an abnormal termination dump. Only the items pertaining to the module load address, and the program entry point are discussed here. For a complete description of the abnormal termination dump, refer to the publication, IBM System/360 Operating System: Control Program Messages and Completion Codes.

How to Use a Dump

Information regarding the error and the reason for an interrupt (and therefore a dump) can be obtained from the completion code, which appears at the beginning of the abnormal termination dump. The completion code indicates the reason for the SYSABEND dump, such as a permanent I/O error, incomplete job control, etc. A description of all the completion codes is given in the publication, IBM System/360 Operating System: Control Program Messages and Completion Codes.

The INTERRUPT at hhhhhh entry, located approximately halfway down in the dump, gives the instruction address that follows the address at which the interrupt occurred. Thus the immediately preceding instruction is that which initiated the dump. The instruction address can be compared to the procedure map. A procedure map is obtained by specifying PMAP in the PARM parameter of the EXEC statement. The load address of the module must be subtracted from the instruction address to obtain the relative instruction address as shown in the procedure map.

The load address of the module (load module) can be obtained from the abnormal termination dump, ACTIVE RBS (request blocks) specification. The last six digits, hhhhhh, of the Addd hhhhhh specification under ACTIVE RBS, are the hexadecimal address of the first RB. The load address of the module (entire program, or object program) is 20 hexadecimal bytes beyond this point. The last six hexadecimal digits of the USE/EP hhhhhhhh specification under ACTIVE RBS are the entry point of the program. The address of the first generated instruction of the procedure division is located 52 bytes beyond the entry point, and is the first address given in the procedure map (PMAP). The first 52 bytes contain a COBOL initialization routine.

The contents of PMAP provide a relative address for each statement. By using the error address and PMAP, the programmer can locate a specific statement appearing within a line of the source program, if the interrupt was within the COBOL program. Examination of the statement and the fields associated with it, may produce information as to the specific nature of the error.

STORAGE LAYOUT OF OBJECT PROGRAM

Each COBOL program written is positioned in main storage in a prescribed manner. The relative position in storage of all the components of a program is given in Figure 41.

```
   * * * A B D U M P   R E Q U E S T E D * * *

JOB cccccccc          STEP cccccccc          DATE ddddd                                                    PAGE dddd

COMPLETION CODE       SYSTEM = hhh (or USER = dddd)

PSW UPON ENTRY TO ABEND hhhhhhhh hhhhhhhh

FL.PT. 0-6    hh.hhhhhh hhhhhhhh         hh.hhhhhh hhhhhhhh         hh.hhhhhh hhhhhhhh         hh.hhhhhh hhhhhhhh

TCB  hhhhhh              RB hhhhhh  PIE hhhhhh  DEB hhhhhh         TIOT hhhhhh      CMP hhhhhh       TRN hhhhhhhh
MSS  hhhhhhhh     PK/FLGS hhhhhhhh        FLGS/LDP hhhhhhhh         LLS  hhhhhh      JLB hhhhhh       JSE hhhhhhhh
ID/FSA  hhhhhhhh        TCB hhhhhh               TME hhhhhh

PIE      PICA hhhhhhhh  PSW hhhhhhhh hhhhhhhh  14 hhhhhhhh  15 hhhhhhhh  00 hhhhhhhh  01 hhhhhhhh  02 hhhhhhhh

ACTIVE RBS

Addd hhhhhh  NM cccccccc  SZ/STAB hhhhhhhh  USE/EP hhhhhhhh  PSW hhhhhhhh hhhhhhhh  Q hhhhhh  WT/LNK hhhhhhhh  UB hhhhhh
REGS 0-7     hhhhhhhh     hhhhhhhh     hhhhhhhh     hhhhhhhh         hhhhhhhh     hhhhhhhh     hhhhhhhh     hhhhhhhh
REGS 8-15    hhhhhhhh     hhhhhhhh     hhhhhhhh     hhhhhhhh         hhhhhhhh     hhhhhhhh     hhhhhhhh     hhhhhhhh

LOAD LIST

Lddd hhhhhh  NM cccccccc  SZ/STAB hhhhhhhh  USE/EP hhhhhhhh                                          UB hhhhhh

SAVE AREA TRACE

cccccccc WAS ENTERED VIA LINK(CALL) ddddd AT EP cccccc...

SA hhhhhhhh  WD1 hhhhhhhh  HSA hhhhhhhh  LSA hhhhhhhh  RET hhhhhhhh   EP hhhhhhhh
00 hhhhhhhh   01 hhhhhhhh   02 hhhhhhhh   03 hhhhhhhh   04 hhhhhhhh   05 hhhhhhhh   06 hhhhhhhh
07 hhhhhhhh   08 hhhhhhhh   09 hhhhhhhh   10 hhhhhhhh   11 hhhhhhhh   12 hhhhhhhh

INCORRECT BACK CHAIN

INTERRUPT AT hhhhhh

PROCEEDING BACK VIA REG 13


      REGS AT ENTRY TO ABEND
     REG 0-7       hhhhhhhh   hhhhhhhh   hhhhhhhh   hhhhhhhh         hhhhhhhh   hhhhhhhh   hhhhhhhh   hhhhhhhh
     REG 8-15      hhhhhhhh   hhhhhhhh   hhhhhhhh   hhhhhhhh         hhhhhhhh   hhhhhhhh   hhhhhhhh   hhhhhhhh

      hhhhhh       hhhhhhhh   hhhhhhhh   hhhhhhhh   hhhhhhhh         hhhhhhhh   hhhhhhhh   hhhhhhhh   hhhhhhhh
      hhhhhh       hhhhhhhh   hhhhhhhh   hhhhhhhh   hhhhhhhh         hhhhhhhh   hhhhhhhh   hhhhhhhh   hhhhhhhh
      LINES hhhhhh-hhhhhh SAME AS ABOVE
      hhhhhh       hhhhhhhh   hhhhhhhh   hhhhhhhh   hhhhhhhh         hhhhhhhh   hhhhhhhh   hhhhhhhh   hhhhhhhh

          END OF DUMP
```

Figure 40. Format of Abnormal Termination Dump

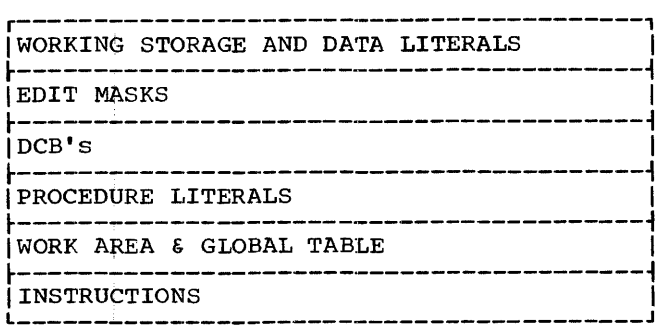


Figure 41. Object Storage Layout

The following examples show the job-control statements used to compile, linkage edit and execute a source module, scratch an existing data set, and catalog the programmer's own procedures. No attempt is made to describe all the parameters used in the job-control language. The assumption is made that the user has read, and is familiar with, the sections Job-Control Language and Job Processing.

The comments stress the major points of importance for the particular example given.

DEFAULT OPTIONS

For the examples given, it is assumed that LIST, PMAP, and DMAP are default options in the compiler. Because these parameters are established at system generation time for the examples given, they are absent from the parameter lists used in the examples.

EXAMPLE 1. COMPILE, LINKAGE EDIT, AND EXECUTE

The example given in Figure 42 processes records by writing them on a disk pack, and then reading them back. Illustrated are excerpts of the actual listing produced. The example indicates the job-control statements required, and the system information provided regarding the data sets used.

```
//EXAMPLE1  JOB ,JOHNDOE,MSGLEVEL=1.
//
//STP1   EXEC  PGM=IEPCBL00
//SYSUT1   DD  DSNAME=UT1,DISP=(NEW,DELETE),SPACE=(TRK,(50,10)),    X
//             UNIT=2311,VOLUME=SER=111111
//SYSUT2   DD  DSNAME=UT2,DISP=(NEW,DELETE),SPACE=(TRK,(50,10)),    X
//             UNIT=2311,VOLUME=SER=111111
//SYSUT3   DD  DSNAME=UT3,DISP=(NEW,DELETE),SPACE=(TRK,(50,10)),    X
//             UNIT=2311,VOLUME=SER=111111
//SYSPRINT  DD   SYSOUT=A
//SYSPUNCH DD  DSNAME=PCH,DISP=(NEW,PASS),SPACE=(TRK,(50,10)),      X
//             UNIT=2311,VOLUME=SER=111111
//STP1.SYSIN DD    *
IEF236I ALLOCATION FOR EXECUTE  STP1     EXAMPLE1
IEF237I SYSUT1   ON 190
IEF237I SYSUT2   ON 190                                    *
IEF237I SYSUT3   ON 190
IEF237I SYSPUNCH ON 190
IEF237I SYSIN    ON 00C

               Compilation listing
                   .
                   .
                   .
                   .


 *Control program messages indicating allocation of data sets.

**Control program messages indicating disposition of data sets.
```

Figure 42. Example of Job Control Statements for Compile, Linkage Edit and Execute (Part 1 of 3)

```
                    .
                    .
                    .
               End compilation listing

IEF285I  UT1                                                  DELETED
IEF285I  VOL SER NOS= 111111.
IEF285I  UT2                                                  DELETED
IEF285I  VOL SER NOS= 111111.
IEF285I  UT3                                                  DELETED
IEF285I  VOL SER NOS= 111111.                                          **
IEF285I  SYSOUT                                               SYSOUT
IEF285I  VOL SER NOS=    FGG.
IEF285I  SYSOUT                                               SYSOUT
IEF285I  VOL SER NOS=    FGG.
IEF285I  PCH                                                  PASSED
IEF285I  VOL SER NOS= 111111.
//STP2   EXEC  PGM=IEWL,PARM='XREF,LIST,LET'
//SYSLIB   DD  DSNAME=SYS1.COBLIB,DISP=(OLD,KEEP),UNIT=2311,          X
//            VOLUME=SER=111111
//SYSLMOD  DD  DSNAME=OBJECT(TEST1),DISP=(NEW,PASS),UNIT=2311,        X
//            VOLUME=SER=111111,SPACE=(TRK,(10,10,10))
//SYSUT1   DD  UNIT=2311,SPACE=(TRK,(10,10)),                         X
//            VOLUME=SER=111111,DISP=(NEW,DELETE),DSNAME=LINKUTL
//SYSLIN   DD  DSNAME=*.STP1.SYSPUNCH,UNIT=2311,                      X
//            DISP=(OLD,DELETE),VOLUME=SER=111111
//SYSPRINT DD  SYSOUT=A
IEF236I ALLOCATION FOR EXECUTE  STP2     EXAMPLE1
IEF237I SYSLIB   ON 190                                               *
IEF237I SYSLMOD  ON 190                                               *
IEF237I SYSUT1   ON 190                                               *
IEF237I SYSLIN   ON 190                                               *

               Module Map and Cross-Reference Table
                              .
                              .
                              .
                              .
                              .
                              .
                              .
                              .
               End of Module Map and Cross-Reference Table

IEF285I  SYS1.COBLIB                                          KEPT
IEF285I  VOL SER NOS= 111111.
IEF285I  OBJECT                                               PASSED
IEF285I  VOL SER NOS= 111111.                                          **
IEF285I  LINKUTL                                              DELETED
IEF285I  VOL SER NOS= 111111.
IEF285I  PCH                                                  DELETED
```

Figure 42. Example of Job Control Statement for Compile, Linkage Edit and Execute (Part 2 of 3)

```
IEF285I  VOL SER NOS= 111111.
IEF285I  SYSOUT                                              SYSOUT
IEF285I  VOL SER NOS=    FGG.                                          **
IEF285I  SYSOUT                                              SYSOUT
IEF285I  VOL SER NOS=    FGG.
//STP3   EXEC  PGM=*.STP2.SYSLMOD
//SYSOUT   DD  SYSOUT=A,DCB=,BLKSIZE=120,LRECL=120)
//SYSABEND DD  SYSOUT=A
//STP3.DATASET1  DD  DSNAME=DISKTEST,UNIT=2311,VOLUME=SER=111111,   C
//             SPACE=(TRK,(10,10)),DISP=(NEW,KEEP)
IEF236I ALLOCATION FOR EXECUTE  STP3     EXAMPLE1
IEF237I SYSLMOD  ON 190                                          *
IEF237I DATASET1 ON 190

         GROUP A LEVEL P TEST CASE 8

         END TEST
WRITE-COUNTER = 20
SHOULD BE = 20                          Output generated by
READ-COUNTER = 21                       load module execution
SHOULD BE = 21
ERROR-COMP-COUNTER = 00
SHOULD BE = 00
GOOD-COMP-COUNTER = 20
SHOULD BE = 20
GOOD-COMP-COUNTER = 20

         IEF285I   OBJECT                                    PASSED
         IEF285I   VOL SER NOS=111111.
         IEF285I   SYSOUT                                    SYSOUT
         IEF285I   VOL SER NOS= FGG.
         IEF285I   SYSOUT                                    SYSOUT
         IEF285I   VOL SER NOS= FGG.                                 **
         IEF285I   DISKTEST                                  KEPT
         IEF285I   VOL SER NOS= 111111.
         IEF285I   OBJECT                                    DELETED
         IEF285I   VOL SER NOS= 111111.
         IEF285I   DISKTEST                                  DELETED
         IEF285I   VOL SER NOS= 111111.
```

Figure 42. Example of Job Control Statements for Compile, Linkage Edit and Execute (Part 3 of 3)

STEP 1. COMPILATION

The //STP1 EXEC statement designates that the program to be executed is the COBOL-E compiler.

The //SYSUT1 DD statement defines the first utility data set. It is on a 2311 disk pack. The DISP parameter (NEW,DELETE) specifies that this is a new data set, and it is to be deleted when the step is terminated.

SYSUT2 and SYSUT3 are the second and third utility files needed for the compilation. Their parameters are identical to those used for SYSUT1.

//SYSPRINT DD statement specifies that a source listing of the compilation be written on SYSOUT which could be a tape or printer.

The //SYSPUNCH DD statement defines a data set that will receive the object module in card image format for subsequent linkage editor processing. Notice the DISP parameter PASS. This parameter specifies that this data set will be referred to in a subsequent job step. If several COBOL programs (e.g. a main program and some subprograms) are compiled in separate job steps and then linkage edited together as part of the same job, the DISP parameter NEW should be replaced by MOD for all but the first compilation.

The //STP1.SYSIN DD * statement defines the input to the compiler to be the source statements that immediately follow.

The information printed on the lines identified by IEF236I indicates the physical unit assignments (at the addresses indicated in the listing) for the logical data sets during the compilation.

The IEF285I lines are a history of the defined data sets for the compilation. For example:

- The SYSUT1 DD statement and DISP parameter specified that this data set (SYSUT1) be deleted. The listing indicates that it was deleted - (UT1 ... DELETED).
- The SYSPUNCH DD statement DISP parameter specified that this data set (SYSPUNCH) be passed. The listing indicates that it was passed -- (PCH ..... PASSED).

STEP 2. LINKAGE EDITOR PROCESSING

The //STP2 EXEC statement designates that the program will be linkage edited with the following options exercised:

1. XREF - This option specifies that a module map and cross-reference table be printed in the listing.
2. LIST - This option specifies that all linkage editor control statements be printed in the listing.
3. LET - This option specifies that the load module be executed even though errors are detected.

The //SYSLIB DD statement defines the COBOL subroutine library for the linkage editor automatic call library as SYS1.COBLIB.

The //SYSLMOD DD statement specifies the load module data set which is the result of linkage editor processing. Note that the DISP parameter specifies that the data set be passed.

The //SYSUT1 DD statement defines a work data set.

The //SYSLIN DD statement defines the input data set to the linkage editor. It is the SYSPUNCH data set of STP1, and is identified by the DSNAME parameter *.STP1.SYSPUNCH. This data set is the object deck produced as specified in STP1.

The //SYSPRINT DD statement specifies that linkage editor diagnostics be output on the printer.

The information printed on the lines identified by IEF236I indicates the physical unit assignments for the logical data sets during the linkage editing.

The information printed on the lines identified by IEF285I is the history of the defined data sets for the linkage edit.

STEP 3. LOAD MODULE EXECUTION

The //STP3 EXEC statement specifies that the load module to be executed is the data set called *.STP2.SYSLMOD.

The //SYSOUT DD statement is required because the DISPLAY verb option is used in the program. Each DISPLAY ... verb results in the printing of a record on the printer. The block size and logical record size for this data set is 120 characters.

The //SYSABEND DD statement specifies an abnormal termination dump for a job. Refer to How To Use A Dump for a brief description of an abnormal termination dump. For complete details of the abnormal termination dump, refer to the publication, IBM System/360 Operating System: Control Program Messages and Completion Codes.

The //STP3.DATASET1 DD statement defines the data set that is processed by the problem program. Notice that DATASET1 in the STP3 DD statement is the external-name used in the ASSIGN clause of this program. (Refer to the environment division in Figure 32.)

The information printed on the lines identified by IEF237I indicates the physical unit assignments for the logical data sets used for execution of the load module.

The WRITE-COUNTER, READ-COUNTER, etc. are display data specified in the WORKING-STORAGE SECTION, and requested in the procedure division of the COBOL source program. (Refer to the source listing in Figure 32.)

The information printed on the lines identified by IEF285I is a history of the data sets defined for execution of the load module.

Figure 43 shows the I/O flow for this example.

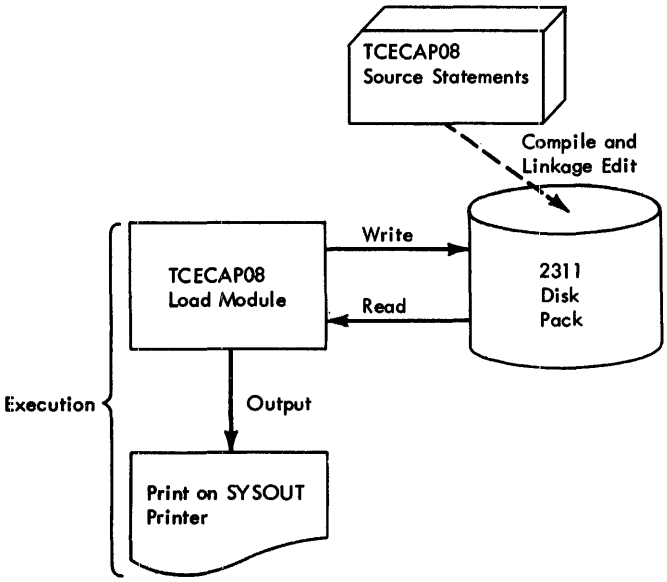


Figure 43. I/O Flow Diagram For Example 1

```
//SCR     JOB   ,SCRATCH,MSGLEVEL=1
//STP     EXEC  PGM=IEHPROGM
//SYSPRINT DD   SYSOUT=A
//DD1       DD  DISP=OLD,                    X
//                VOLUME=SER=111111,         X
//                UNIT=2311
//SYSIN     DD  *

          {Specified data sets to be
                scratched                   }

IEF236I  ALLOCATION FOR SCR      STP
IEF237I  DD1      ON 190
IEF237I  SYSIN    ON 000
```

Figure 44. Example of Job-Control Statements for Scratching Data Sets

EXAMPLE 2. SCRATCHING A DATA SET

In the event of an abnormal job termination, defined data sets might be retained by the system. When the same program or any other program is executed again, using the identically defined data sets, the system recognizes these data sets as duplicates and terminates the job. It checks each data set against those it already retains. It does not accept already recorded data sets; as a result, the job is terminated.

The scratch procedure ensures that a job is not terminated because of an already existing data set.

Figure 44 is an example of a program that scratches existing data sets.

JOB STATEMENTS AND DATA SETS FOR SCRATCHING DATA SETS

In example 2, //STP EXEC card parameter IEHPROGM is the name of the IBM-supplied utility program that accomplishes the scratch. That is, it erases the data sets specified after the //SYSIN DD * statement. Thus, the data sets defined in the program to be executed are accepted by the system, and the program can be executed.

//SYSPRINT DD, //DD1 DD, and //SYSIN DD * are work data sets required by the utility program.

For details on how to specify data sets to be scratched, refer to Scratching Data Sets in the Job Processing section.

EXAMPLE 3. CATALOGING A PROCEDURE

Figure 45 illustrates how to catalog a procedure. The procedure being cataloged is a compile, linkage edit, and execute procedure.

Once a procedure is cataloged, it is available to the user by merely specifying the name of the procedure in the // EXEC statement.

```
//CATLG3 JOB   ,CATLGPROC,MSGLEVEL=1
//STEPA  EXEC  PGM=IEBUPDAT,PARM=(NEW)
//SYSUT2   DD  DSNAME=SYS1.PROCLIB,DISP=(OLD)
//SYSPRINT DD  SYSOUT=A
//SYSIN    DD  DATA
./       ADD   CBLPROC3,00,01
./       NUMBR 00000000,00000000,00000000,                X
./             00000010
//STP1   EXEC  PGM=IEPCBL00
//SYSLIB   DD  DSNAME=COBOLLIB,UNIT=2311,                  X
//            DISP=(OLD,KEEP),VOLUME=SER=111111
//SYSUT1   DD  DSNAME=UT1,DISP=(NEW,DELETE),               X
//            SPACE=(TRK,(50,10)),UNIT=2311,               X
//            VOLUME=SER=222222
//SYSUT2   DD  DSNAME=UT2,DISP=(NEW,DELETE),               X
//            SPACE=(TRK,(50,10)),UNIT=2311,               X
//            VOLUME=SER=222222
//SYSUT3   DD  DSNAME=UT3,DISP=(NEW,DELETE),               X
//            SPACE=(TRK,(50,10)),UNIT=2311,               X
//            VOLUME=SER=222222
//SYSPRINT DD  SYSOUT=A
//SYSPUNCH DD  DSNAME=PCH,DISP=(NEW,PASS),                 X
//            SPACE=(TRK,(50,10)),UNIT=2311,               X
//            VOLUME=SER=222222
//STP2   EXEC  PGM=IEWL,PARM='XREF,LIST,LET'
//SYSLIB   DD  DSNAME=SYS1.COBLIB,DISP=(OLD,KEEP),         X
//            UNIT=2311,VOLUME=SER=111111
//SYSLMOD  DD  DSNAME=&GODATA(TEST),DISP=(NEW,PASS),       X
//            UNIT=2311,VOLUME=SER=111111,                 X
//            SPACE=(TRK,10,10,10))
//SYSUT1   DD  UNIT=2311,SPACE=(TRK,(10,10)),              X
//            VOLUME=SER=222222,DISP=(NEW,DELETE)
//SYSLIN   DD  DSNAME=*.STP1.SYSPUNCH,UNIT=2311,           X
//            DISP=(OLD,DELETE),VOLUME=SER=222222
//STP3   EXEC  PGM=*.STP2.SYSLMOD
//SYSOUT   DD  SYSOUT=A,DCB=(,BLKSIZE=120,LRECL=120)
//SYSABEND DD  SYSOUT=A
./ENDUP
/*
```

Figure 45. Example of Job-Control Statement for Cataloging a Procedure

JOB-CONTROL STATEMENTS AND DATA SETS FOR CATALOGING A PROCEDURE

In example 3, the //STEPA EXEC PGM parameter IEBUPDAT is the name of the IBM-supplied utility program that accomplishes the cataloging. The parameter NEW indicates that the input to the utility program consists of the SYSIN data set.

//SYSUT2 defines the work file for the utility program. The parameter PROCLIB defines the PDS to be updated.

The //SYSIN DD DATA statement indicates to the system that the job-control statements that follow are to be treated as data and are not to be interpreted.

The ./ ADD CBLPROC3 statement is a utility statement and indicates that the following procedure (data set) is to be added to the library. The ./ENDUP statement (at the end of the listing) is a utility statement signifying the end of the source statements to be cataloged. (Neither ./ utility statements are entered in the library.) The name CBLPROC3 specified in the ./ ADD statement identifies the procedure to be cataloged, and is the procedure name to be used in the // EXEC statement when the procedure is desired.

The ./ NUMBR statement specifies to the utility program that sequence numbers be assigned to the records within the new catalog procedure.

The //SYSLIB DD statement (in STP1 of the procedure) indicates to the compiler that a program to be compiled might contain COPY or INCLUDE statements. The remainder

of the parameters on the //SYSLIB DD statement describes the source statement library. If this DD statement is omitted, the compiler terminates the job upon encountering a COPY or INCLUDE statement in the source statement program.

The //SYSLIB DD statement (in STP2 of the linkage edit procedure) identifies the COBOL subroutine library for the linkage editor. The parameter SYS1.COBLIB is the name of the COBOL subroutine library.

The //SYSLMOD DD statement defines the output data set to the linkage editor.

The //SYSUT1 DD statement defines the work data set for the linkage editor.

The //SYSLIN DD statement defines the primary input data set to the linkage editor. It is identified by the parameter STP1.SYSPUNCH.

The //STP3 EXEC parameter .STP2.SYSLMOD identifies the load module to be executed by the system.

For descriptions of the //SYSABEND DD statement and any other statements or parameters not covered in Figure 45, refer to Example 1 (Figure 42) in this appendix, the section, Creating Data Sets, or the publications IBM System/360 Operating System: Control Program Messages and Completion Codes, and IBM System/360 Operating System: Job Control Language.

APPENDIX B. ASSEMBLER LANGUAGE SUBPROGRAMS

This appendix provides information needed to prepare and use subprograms written in assembler language with a main program written in COBOL.

CALLED AND CALLING PROGRAMS

Any program referred to by another program is a called subprogram. If this called subprogram refers to another subprogram, it is both a called and calling subprogram. In Figure 46, program A calls subprogram B; subprogram B calls subprogram C; therefore:

1. A is considered a calling program by B.
2. B is considered a called subprogram by A.
3. B is considered a calling subprogram by C.
4. C is considered a called subprogram by B.

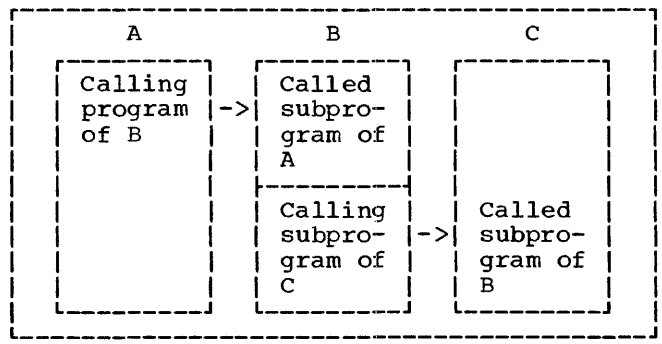


Figure 46. Called and Calling Programs

There are three basic ways to use assembler-written subprograms with a main program written in COBOL:

1. A COBOL main program or subprogram calling an assembler-written subprogram.
2. An assembler-written subprogram calling a COBOL subprogram.
3. An assembler-written subprogram calling another assembler-written subprogram.

From these combinations, more complicated structures can be formed.

The operating system has established certain conventions to give control to and return control from assembler-written subprograms. These conventions, called linkage conventions, are described in the following text.

LINKAGE CONVENTIONS

The save and return routines for assembler subprograms need not be written exactly the same as those generated by the COBOL compiler. However, there are basic conventions for COBOL programs to which the assembler programmer must adhere. These conventions include:

1. Using the proper registers to establish linkage.
2. Reserving, in the calling program, an area that is used by the called subprogram to refer to the argument list.
3. Reserving, in the calling program, a save area in which the registers may be saved.

REGISTER USE

The operating system has assigned functions to certain registers used in linkages. The function of each linkage register is shown in Table 17.

Table 17. Linkage Registers

REGISTER NUMBER	REGISTER NAME	FUNCTION
1	Argument Register	Address of the argument list passed to the called subprogram.
13	Save Register	Address of the area reserved by the calling program in which the contents of certain registers are stored by the called program.
14	Return Register	Address of the location in the calling program to which control is returned after execution of the called program.
15	Entry Point Register	Address of the entry point in the called subprogram.

ARGUMENT LIST

Every assembler-written subprogram that calls another subprogram must reserve an area of storage (argument list) in which the argument list used by the called subprogram is located. Each entry in the parameter list occupies four bytes and is on a full-word boundary.

In the first byte of each entry in the parameter list, bits 1 through 7 contain zeros. However, bit 0 may contain a 1 to indicate the last entry in the parameter area.

The last three bytes of each entry contain the 24-bit address of the argument.

SAVE AREA

An assembler subprogram that calls another subprogram must reserve an area of storage (save area) in which certain registers (i.e., those used in the called subprogram and those used in the linkage to the called subprogram) are saved.

The maximum amount of storage reserved by the calling subprogram is 18 words. Figure 47 shows the layout of the save area and the contents of each word.

A called COBOL subprogram does not save floating-point registers. The programmer is responsible for saving and restoring the contents of these registers in the calling program.

AREA	
(word 1)	This word is a part of the standard linkage convention established under the operating system. The word must be reserved for proper addressing of the succeeding entries. However, an assembler subprogram may use the word for any desired purpose.
AREA+4 (word 2)	The address of the previous save area; that is, the save area of the subprogram that called this one.
AREA+8 (word 3)	The address of the next save area; that is, the save area of the subprogram to which this subprogram refers.
AREA+12 (word 4)	The contents of register 14; that is, the return address.
AREA+16 (word 5)	The contents of register 15; that is, the entry address.
AREA+20 (word 6)	The contents of register 0.
AREA+24 (word 7)	The contents of register 1.
. .	. . .
AREA+68 (word 18)	The contents of register 12.

Figure 47. Save Area Layout and Word Contents

```
deckname  START  0
          ENTRY  name1
          EXTRN  name2
          USING  *,15
*  Save Routine
name1     STM    14,r1,12(13)        The contents of registers 14, 15, and 0 through
*                                    r1 are stored in the save area of the calling
*                                    program (previous save area). r1 is any number
*                                    from 0 through 12.
          LR     r2,13               Loads register 13, which points to the save area
*                                    of the calling program, into any general
*                                    register, r2, except 0 and 13.
          LA     13,AREA             Loads the address of this program's save area
*                                    into register 13.
          ST     13,8(0,r2)          Store the address of this program's save area
*                                    into word 3 of the save area of the calling
*                                    program.
          ST     r2,4(0,13)          Stores the address of the previous save area
*                                    (i.e., the same area of the calling program) into
*                                    word 2 of this program's save area.
          BC     15,prob1
AREA      DS     18F                 Reserves 18 words for the save area.  This is
*                                    last statement of save routine.
prob1     User-written program statements
*  Calling Sequence
          LA     1,ARGLST            First statement in calling sequence.
          L      15,ADCON
          BALR   14,15
*         Remainder of user-written program statements
*  Return Routine
          L      13,AREA+4           First statement in return routine.  Loads the
*                                    address of the previous save area back into
*                                    register 13.
          LM     2,R1,28(13)         The contents of registers 2 through r1, are
*                                    restored from the previous save area.
          L      14,12(13)           Loads the return address, which is in word 4 of
*                                    the calling program's save area, into register
*                                    14.
          MVI    12(13),X'FF'        Sets flag FF in the save area of the calling
*                                    program to indicate that control has returned to
*                                    the calling program.
          BCR    15,14               Last statement in return routine.
ADCON     DC     A(name2)            Contains the address of subprogram name2.
*  Parameter List
ARGLST    DC     AL4(arg1)           First statement in parameter area setup.
          DC     AL4(arg2)
          DC     X'80'               First byte of last argument sets bit 0 to 1.
          DC     AL3(argn)           Last statement in parameter area setup.
```

Figure 48. Sample Linkage Routines Used with a Calling Subprogram

Example

The linkage conventions used by an assembler subprogram that calls another subprogram are shown in Figure 48. The linkage should include:

1. The calling sequence.
2. The save and return routines.
3. The out-of-line parameter list. (An in-line parameter list may be used; see In-line Parameter List.)
4. A save area on a fullword boundary.

LOWEST LEVEL SUBPROGRAM

If an assembler subprogram does not call any other subprogram (i.e., if it is at the lowest level), the programmer should omit the save routine, calling sequence, and

parameter list shown in Figure 48. If the assembler subprogram uses any registers, it must save them. Figure 49 shows the appropriate linkage conventions used by an assembler subprogram at the lowest level.

```
deckname        START         0
                ENTRY         name
                USING         *,15
name            STM           14,r1,12(13)
                  .
                  .
                  .
User-written program statements
                  .
                  .
                  .
                LM            2,r1,28(13)
                MVI           12(13),X'FF'
                BCR           15,14
```

Note: If registers 13 and/or 14 are used in the called subprogram, their contents should be saved and restored by the called subprogram.

Figure 49. Sample Linkage Routines Used with a Lowest Level Subprogram

IN-LINE PARAMETER LIST

The assembler programmer may establish an in-line parameter list instead of out-of-line list. In this case, he may substitute the calling sequence and parameter list shown in Figure 50 for that shown in Figure 48.

```
ADCON           DC            A(prob1)
                  .
                  .
                  .
                LA            14,RETURN
                L             15,ADCON
                CNOP          2,4
                BALR          1,15
                DC            AL4(arg1)
                DC            AL4(arg2)
                  .
                  .
                  .
                DC            X'80'
                DC            AL3(argn)
RETURN          BC            0,X'isn'
```

Figure 50. Sample In-Line Parameter List

DATA FORMAT OF ARGUMENTS

Any assembler-written subprogram must be coded with a detailed knowledge of the data formats of the arguments being passed. Most coding errors will probably occur because of the data-format discrepancies of the arguments.

If one programmer writes both the main program and the subprogram, the data formats of the arguments should not present a problem when passed as parameters. However, when the programs are written by different programmers, the data-format specifications for the arguments must be clearly defined for the user.

ACCESSING INFORMATION NOT DIRECTLY AVAILABLE AT THE COBOL LANGUAGE LEVEL

Figures 51 and 52 are listings of a COBOL language source program and an assembler language subprogram, respectively. These programs illustrate how a COBOL programmer can access information in subprograms not directly available through the COBOL language. They allow the programmer to:

- Obtain a value from the PARM parameter of the EXEC card
- Obtain the date from the control program
- Set a condition code to be used by the next job step.

The documentation within the assembler subprogram explains what is accomplished within each segment of the subprogram.

```
001001    IDENTIFICATION DIVISION.
001002     PROGRAM-ID. 'EXAMPLEA'.
002001    ENVIRONMENT DIVISION.
002002     CONFIGURATION SECTION.
002003      SOURCE-COMPUTER. IBM-360 H50.
002004      OBJECT-COMPUTER. IBM-360 H50.
002005     INPUT-OUTPUT SECTION.
002006      FILE-CONTROL.
002007       SELECT INFILE ASSIGN 'CARDREAD' UTILITY.
002008       SELECT OUTFILE ASSIGN 'TAPEOUT' UTILITY.
003001    DATA DIVISION.
003002     FILE SECTION.
003003      FD INFILE RECORDING F LABEL RECORD OMITTED DATA RECORD INAREA.
003004       01 INAREA.
003005        02 PART-NUMBER PICTURE IS X(10).
003006        02 QUANTITY PICTURE IS 9(6).
003007        02 COST PICTURE IS 9(4)V99.
003008        02 FILLER PICTURE X(58).
003009      FD OUTFILE RECORDING F LABEL RECORD STANDARD DATA RECORD
003010         OUTAREA BLOCK CONTAINS 10 RECORDS.
003011       01 OUTAREA.
003012        02 PART-NUMBER PICTURE IS X(10).
003013        02 SERIAL-DAY PICTURE IS 9(3).
003014        02 QUANTITY PICTURE IS S9(6) USAGE IS COMPUTATIONAL-3.
003015        02 COST PICTURE IS 9(4)V99 USAGE IS COMPUTATIONAL-3.
003016        02 EXTENSION PICTURE IS 9(6)V99 USAGE IS COMPUTATIONAL-3.
003017         02 MFG-DAY PICTURE IS X(3).
003018        02 FILLER PICTURE IS X.
003019     WORKING-STORAGE SECTION.
003020      77 MFG-DAY-LENGTH PICTURE S9 VALUE +3 COMPUTATIONAL.
003021      77 DATA-ERROR-COUNT PICTURE S99 COMPUTATIONAL VALUE 0.
003022      77 COND-CODE-FOR-NEXT-JOB-STEP PICTURE 99 COMPUTATIONAL.
003023      77 GO-COND-CODE PICTURE 99 VALUE 4 COMPUTATIONAL.
003024      77 STOP-COND-CODE PICTURE 99 VALUE 64 COMPUTATIONAL.
003025      01 EXEC-PARM-VALUE.
004001       02 PARM-LENGTH PICTURE S9 COMPUTATIONAL.
004002       02 MFG-DAY-FROM-EXEC-PARM PICTURE IS 9(3).
004003      01 SERIAL-DAY-FROM-TIME-MACRO PICTURE 9(3).
005001    PROCEDURE DIVISION.
005002     GO-TO-SUBROUTINE.
005003        ENTER LINKAGE.
005004        CALL 'GETPARM' USING EXEC-PARM-VALUE,
005005        SERIAL-DAY-FROM-TIME-MACRO.
005006        ENTER COBOL.
005007     START-MAIN-PROGRAM.
005008        IF PARM-LENGTH EQUAL TO MFG-DAY-LENGTH THEN NEXT SENTENCE,
005009         OTHERWISE GO TO ABORT-JOB.
005010        IF MFG-DAY-FROM-EXEC-PARM IS NOT NUMERIC GO TO ABORT-JOB.
005011        OPEN INPUT INFILE. OPEN OUTPUT OUTFILE.
005012     READFILE.
005013        READ INFILE AT END GO TO END-OF-JOB.
005014        IF COST OF INAREA NOT NUMERIC GO TO ERRORENTRY.
005015        IF QUANTITY OF INAREA NOT NUMERIC GO TO ERRORENTRY.
005016        MOVE QUANTITY OF INAREA TO QUANTITY OF OUTAREA.
005017        MOVE COST OF INAREA TO COST OF OUTAREA.
005018        MULTIPLY QUANTITY OF OUTAREA BY COST OF OUTAREA GIVING
005019         EXTENSION ON SIZE ERROR GO TO ERRORENTRY.
005020        MOVE PART-NUMBER OF INAREA TO PART-NUMBER OF OUTAREA.
005021        MOVE MFG-DAY-FROM-EXEC-PARM TO MFG-DAY.
005022        MOVE SERIAL-DAY-FROM-TIME-MACRO TO SERIAL-DAY.
005023        WRITE OUTAREA. GO TO READFILE.
```

Figure 51. COBOL Source Program (Part 1 of 2)

```
005024   ERRORENTRY.
005025        ADD 1 TO DATA-ERROR-COUNT.
006001        DISPLAY 'ERROR PN' PART-NUMBER OF INAREA.
006002        GO TO READFILE.
006003   ABORT-JOB.
006004      DISPLAY 'IMPROPER PARM VALUE IN EXEC CARD, JOB TERMINATED'.
006005   SET-STOP-CODE.
006006      MOVE STOP-COND-CODE TO COND-CODE-FOR-NEXT-JOB-STEP.
006007      GO TO EXIT-JOB.
006008   END-OF-JOB.
006009      DISPLAY 'END OF JOB'.
006010      CLOSE INFILE, OUTFILE.
006011      IF DATA-ERROR-COUNT GREATER THAN 5, GO TO SET-STOP-CODE.
006012      MOVE GO-COND-CODE TO COND-CODE-FOR-NEXT-JOB-STEP.
006013   EXIT-JOB.
006014      ENTER LINKAGE.
006015      CALL 'SETCODE' USING COND-CODE-FOR-NEXT-JOB-STEP.
006016      ENTER COBOL.
```

Figure 51. COBOL Source Program (Part 2 of 2)

```
GETPARM      START
             ENTRY SETCODE
*            SAVE REGISTERS AND POST SAVE AREAS
*
             STM   14,5,12(13)          SAVE REGISTERS
             BALR  5,0                  ESTABLISH BASE REGISTERS
             USING *,5
             LA    14,SAVEAREA          LOAD NEW SAVE AREA ADDRESS
             ST    14,8(13)             NEW SAVE AREA ADDR TO OLD SAVE AREA
             ST    13,4(14)             OLD SAVE AREA ADDR TO NEW SAVE AREA
             L     2,4(0,13)            ADDR OF SAVE AREA USED BY COBOL
             LR    13,14                NEW SAVE AREA ADDRESS T- REGISTER 13
*
*            MOVE EXEC CARD PARM OPERAND TO COBOL PROGRAM
*
             L     4,24(0,2)            ADD OF INITIATOR PARAMETER LIST
             L     2,0(0,4)             ADDR OF EXEC CARD PARM OPERAND DATA
             L     4,0(0,1)             ADDR OF COBOL NAME 'EXEC-PARM-VALUE'
             MVC   0(5,4),0(2)          MOVE PARM DATA TO COBOL PROGRAM
*            OBTAIN DATE AND PLACE SAME IN COBOL PROGRAM
*
             LR    4,1                  SAVE ADDR OF COBOL PARAMETER LIST
             TIME  DEC                  GET DATE
             ST    1,DATEAREA           SAVE DATE
             L     2,4(0,4)             ADDR OF COBOL DATE RECEIVING FIELD
             UNPK  0(3,2),DATEAREA+2(2) MOVE DATE TO COBOL PROGRAM
*
*            RESTORE REGISTERS, RETURN TO COBOL PROGRAM
*
             L     13,4(13)             RESTORE REGISTER 13
             LM    14,5,12(13)          RESTORE REGISTERS
             BR    14                   RETURN TO COBOL PROGRAM
*
*            SET CONDITION CODE AT TERMINATION OF COBOL PROGRAM
*
SETCODE      L     2,0(0,1)             COND-CODE VALUE ADDR
             LH    15,0(0,2)            SET COND-CODE
             L     13,4(0,13)           RESTORE REGISTER 13
             L     14,12(13)            RETURN ADDR IN TERMINATOR
             LM    0,12,20(13)          RESTORE REGISTERS
             BR    14                   RETURN TO TERMINATOR
*
*            CONSTANTS USED BY ASSEMBLER ROUTINE
*
DATEAREA     DS    F                    WORK-AREA FOR DATE
SAVEAREA     DS    18F
             END   GETPARM
```

Figure 52. Assembler Subprogram

The COBOL program itself is produced as one control section. However, there may be subprograms and other external references, such as entry points to subprograms, to be resolved. The subprograms that a user may wish to combine with the main program can be obtained from SYSIN, the automatic call library (COBLIB), or one of his own libraries.

The following discussion illustrates the procedures available for processing COBOL subprograms. The first technique employs the linkage editor without using the overlay facility. The second technique employs the linkage editor using an overlay technique. This technique allows the programmer to specify, at linkage edit time, the overlays required for a program. During execution of a program overlays are performed automatically for the programmer by the control program. The third technique, which is used during execution, requires that the programmer generate the needed macro instructions to effect the overlays.

Note: The largest load module that can be processed by Fetch is 524,248 bytes. If a load module exceeds this limit, it should be divided.

CONSIDERATIONS FOR OVERLAY

Assume a COBOL main program exists, called COBMAIN, that contains calls at one or more points in its logic to COBOL subprograms: CSUB1, CSUB2, CSUB3, CSUB4, and CSUB5. Also assume that the load module sizes for the main program and the subprograms are as given in Figure 53.

PROGRAM	MODULE SIZE (IN BYTES)
COBMAIN	20,000
CSUB1	4,000
CSUB2	5,000
CSUB3	6,000
CSUB4	3,000
CSUB5	4,000

Figure 53. Assumed Program Module Sizes

Through the linkage mechanism, ENTER LINKAGE, CALL SUB1..., all subprograms plus COBMAIN must be linkage edited together to form one module 42,000 bytes in size. Therefore, COBMAIN would require 42,000 bytes of storage in order to be executed.

Normally, all subprograms referenced by the COBOL source program, including the main program, will fit into main storage. Therefore, the linkage editor nonoverlay technique of processing can be used to execute the entire program.

Figure 54 illustrates the storage layout for nonoverlay processing.

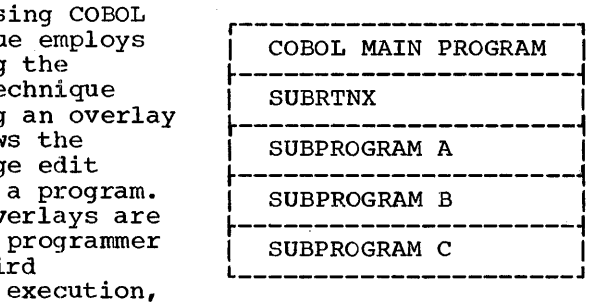


Figure 54. Storage Layout for Nonoverlay Processing

LINKAGE EDIT WITHOUT OVERLAY

Figure 55 shows a deck setup for a nonoverlay structure. In this case, all the subprograms (including MAIN PROGRAM) fit into main storage.

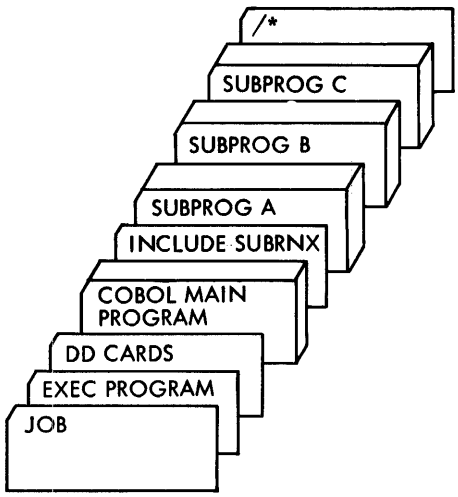


Figure 55. Example Deck for Linkage Editor Nonoverlay Structure

OVERLAY PROCESSING

If the subprograms needed do not fit into main storage, it is still possible to use them. The technique that enables using subprograms that do not fit into main storage (along with the main program) is called overlay.

Figure 56 illustrates storage layout for overlay processing.

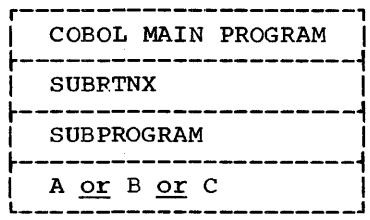


Figure 56. Storage Layout for Overlay Processing

There are two techniques of overlay available to the COBOL programmer. They are:

- Preplanned overlay using the linkage editor
- Dynamic overlay using macro instructions during execution

PREPLANNED LINKAGE EDITING WITH OVERLAY

The preplanned linkage editor facility permits the reuse of storage locations already occupied. By judiciously segmenting a program, and using the preplanned linkage editor overlay facility, the programmer can accomplish the execution of a program too large to fit into storage at one time.

In using the preplanned overlay technique, the programmer specifies, to the linkage editor, which subprograms are to overlay each other. The subprograms specified are processed, as part of the program, by the linkage editor so they can be automatically placed in main storage for execution when requested by the program. The resulting output of the linkage editor is called an overlay structure.

It is possible, at linkage edit time, to set up an overlay structure by using the COBOL source language statement ENTER LINKAGE and the linkage editor OVERLAY statement. These statements enable a user to call a subprogram that is not actually in storage. The details for setting up the linkage editor control statements for accomplishing this procedure can be found in the publication, IBM System/360 Operating System: Linkage Editor.

In a linkage editor run, the programmer specifies the overlay points in a program by using OVERLAY statements. The linkage editor treats the entire input as one program, resolving all symbols and inserting tables into the program.

These tables are used by the control program to bring the overlay subprograms into storage automatically, when called.

Figure 57 shows the deck setup for an overlay structure using preplanned linkage editor overlay. The OVERLAY statements specify to the linkage editor that the overlay structure to be established is one in which SUBPROGA, SUBPROGB and SUBPROGC overlay each other when called during execution.

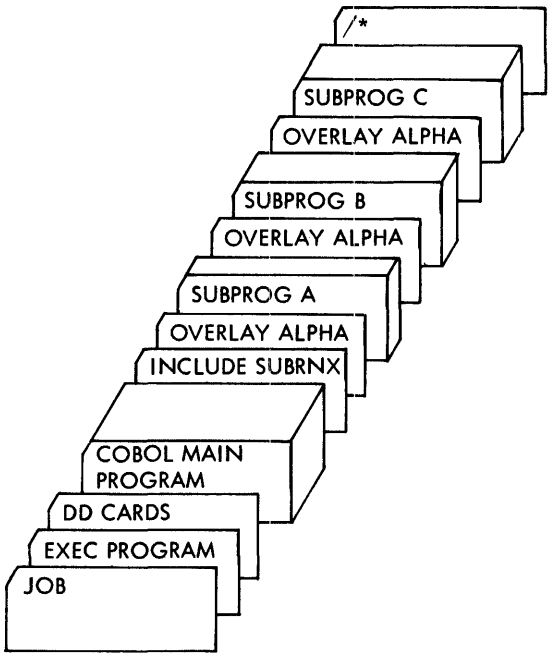


Figure 57. Example Deck for Linkage Editor Overlay Structure

DYNAMIC OVERLAY FEATURE

In preparation for the dynamic overlay technique, each part of the program that is brought into storage independently should be processed separately by the linkage

editor. (Hence, each part must be processed as a separate load module). To execute the entire program, the programmer must:

1. Specify the main program (in Figure 53 COBMAIN) in the EXEC statement, and

2. Bring the separately processed load modules into storage, when they are required, by using the appropriate supervisor linkage macro instructions. This is accomplished during execution.

This technique can be used to overlay subprograms during execution. To accomplish dynamic overlay of subprograms, the programmer must write an assembler language subprogram that employs the LINK macro to call each COBOL subprogram. For a detailed description of the LINK macro instruction, refer to the publication, IBM System/360 Operating System: Control Program Services.

In using this technique, the main program (in Figure 53, COBMAIN) communicates with the assembler language subprogram by using the COBOL language CALL statement. The COBOL CALL statement can be used to pass the name of the COBOL subprogram (to be linked), and the specified parameter list, to the assembler language subprogram. This procedure is effected with each CALL used in the main program. Hence, each CALL results in linking with a subprogram through the assembler language subprogram.

When the COBOL subprogram is finished executing, it returns to the assembler language subprogram, which in turn returns to the main program (in Figure 53, COBMAIN). The process is repeated for each CALL to the assembler language subprogram.

This technique requires that a programmer have detailed knowledge of the linkage conventions, assembler language, and the LINK macro with its features and restrictions.

Beyond this, the programmer must ensure that the COBOL subprogram modules exist in a private library (PDS) and are defined by a //JOBLIB DD statement in the job-control language for execution of the main program. Refer to Job-Control Language for a description of the //JOBLIB DD statement.

APPENDIX D. COBOL SYNTAX FORMATS

The following is a list of COBOL statements to be used with initial release version of the COBOL (E) compiler.

```
IDENTIFICATION DIVISION.

PROGRAM-ID.  'program-name'.

[AUTHOR.  sentence...]
[INSTALLATION. sentence...]
[DATE-WRITTEN. sentence...]
[DATE-COMPILED. sentence...]
[SECURITY. sentence...]
[REMARKS. sentence...]

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

[SOURCE-COMPUTER. IBM-360 [model-number].]
[OBJECT-COMPUTER. IBM-360 [model-number].]
INPUT-OUTPUT SECTION. [COPY library-name.]
FILE-CONTROL. [COPY library-name.]
    [ SELECT file-name [COPY library-name]
      ASSIGN TO external-name      { DIRECT-ACCESS }
                                   { UTILITY       } device-number
                                   { UNIT--RECORD  } UNIT[S]      ....

      [ RESERVE { NO      } ALTERNATE AREA[S] ]
                { integer }

      [ACCESS IS { SEQUENTIAL }]
                 { RANDOM     }

      [ORGANIZATION IS { INDEXED  }]
                       { DIRECT   }
                       { RELATIVE }

      [SYMBOLIC KEY IS data-name]

      [ACTUAL KEY IS data-name]

      [RECORD KEY IS data-name]

      [TRACK-AREA IS data-name CHARACTERS]

      [FILE-LIMIT IS integer TRACKS] ]

I-O-CONTROL.

     [SAME AREA FOR file-name-1 file-name-2... .]
     [APPLY overflow-name to FORM-OVERFLOW ON file-name.]
     [APPLY RESTRICTED SEARCH OF integer TRACKS ON file-name... .]
     [APPLY WRITE-ONLY ON file-name... .]

DATA DIVISION.

FILE SECTION.

FD file-name    [COPY library-name.]
     [BLOCK CONTAINS integer { CHARACTERS } ]
                             { RECORDS    }

     [RECORDING MODE IS { U } ]
                        { F }
                        { V }
```

```
     [RECORD CONTAINS [integer-1 TO]  integer-2 CHARACTERS]
                 {RECORD IS }  {STANDARD}
     LABEL       {RECORDS are} {OMITTED }

                 {RECORD IS }
     DATA        {RECORDS ARE} record-name... .
```

```
Record Description Entry.
WORKING-STORAGE SECTION.
Record Description entries
LINKAGE SECTION.
Record Description entries

                 {data-name}
level-number     {FILLER   }        [REDEFINES data-name-2]  [COPY library-name.]

[                {alpha-form  }  ]
[PICTURE IS      {an-form     }  ]
[                {numeric-form}  ]
[                {report-form }  ]
[                {fp-form     }  ]

[OCCURS integer TIMES[DEPENDING ON data-name]]
[JUSTIFIED RIGHT]
BLANK WHEN ZERO]
[VALUE IS literal]

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

PROCEDURE DIVISION.
[Section-name SECTION.]
Paragraph-name.

     ACCEPT data-name [FROM CONSOLE]

                {numeric-literal      }          {TO   }
     ADD        {floating-point-literal}... {   }{GIVING} data-name-n
                {data-name-1          }          {     }

          [ROUNDED] [ON SIZE ERROR imperative-statement...]

     ALTER {procedure-name-1 TO PROCEED TO procedure-name-2}...

     CLOSE     {file-name [REEL  ][WITH NO REWIND]}    ...
               {          [UNIT  ][WITH LOCK     ]}

                                    {data-name-2          }
COMPUTE data-name-1   [ROUNDED] =   {numeric-literal      }
                                    {floating-point-literal}
                                    {arithmetic-expression}

     [ON SIZE ERROR imperative-statement...]

                {data-name}...  [          {CONSOLE }]
DISPLAY         {literal  }     [UPON      {SYSPUNCH}]

                {data-name-1          }       {data-name-2 [GIVING data-name-3]          }
DIVIDE          {numeric-literal-1    }INTO   {numeric-literal-2 GIVING data-name-3      }
                {floating-point-literal-1}    {floating-point-literal-2 GIVING data-name-3}

                    [ROUNDED]  [ON SIZE ERROR imperative statement...]
```

```
ENTER LINKAGE.
CALL entry-name [USING argument...].
ENTRY entry-name [USING data-name...].
RETURN.
ENTER COBOL.
```

```
EXAMINE  data-name    TALLYING { ALL       } 'character-1' [REPLACING BY 'character-2']
                               { LEADING   }
                               { UNTIL FIRST }
```

```
EXAMINE data-name REPLACING { ALL       } 'character-1'BY 'character-2'
                            { LEADING   }
                            { UNTIL FIRST }
                            { FIRST     }
```

```
EXHIBIT { NAMED         } { data-name         }...
        { CHANGED NAMED } { non-numeric-literal }
        { CHANGED       }
```

```
paragraph-name. EXIT.
```

```
GO TO procedure-name-1 [procedure-name-2... DEPENDING ON data-name]
```

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

```
CONDITIONS:

{ data-name-1                 } IS [NOT] { >             } { data-name-2                 }
{ arithmetic-expression-1     }          { <             } { arithmetic-expression-2     }
{ figurative-constant-1       }          { =             } { figurative-constant-2       }
                                         { GREATER THAN  } { literal-2                   }
                                         { LESS THAN     }
                                         { EQUAL TO      }

{ data-name       } IS [NOT] { POSITIVE }
{ arithmetic-expression }    { ZERO     }
                             { NEGATIVE }

[NOT] condition-name

[NOT] overflow-name

data-name IS [NOT] { NUMERIC    }
                   { ALPHABETIC }

{ section-name SECTION. } INCLUDE library-name.
{ paragraph-name.       }

MOVE { data-name-1 } TO data-name-2 ...
     { literal     }

MULTIPLY { data-name-1                 }
         { numeric-literal-1           }
         { floating-point-literal-1    }

      BY { data-name-2 [GIVING data-name-3]                  }
         { numeric-literal-2 GIVING data-name-3              }
         { floating-point-literal-2 GIVING data-name-3       }

            [ROUNDED] [ON SIZE ERROR imperative statement...]
```

```
NOTE comment...
```

```
ON integer-1 [AND EVERY integer-2]  [UNTIL integer-3]  { imperative-statement... }
                                                       { NEXT SENTENCE          }

                  [ { ELSE      } { statement...   } ]
                  [ { OTHERWISE } { NEXT SENTENCE } ]
```

```
     {INPUT {file-name [with NO REWIND] [REVERSED]} ...
       [OUTPUT {file-name [with NO REWIND]}....]
         [I-O {file-name}...]
OPEN {OUTPUT {file-name [with NO REWIND]}...
       [INPUT {file-name [with NO REWIND] [REVERSED]}...]
         I-O {file-name}...]
     {I-O {file-name}... [OUTPUT {file-name [with NO REWIND ]}...]
       [INPUT {file-name [with NO REWIND] [REVERSED]}...]

                                                    {integer  TIMES}
PERFORM procedure-name-1 [THRU procedure-name-2]    {data-name     }

          UNTIL condition

PERFORM procedure-name-1 [THRU procedure-name-2]

     VARYING data-name-1 FROM {numeric-literal-1}
                              {data-name-2      }

     BY {numeric-literal-2}    UNTIL test-condition-1
        {data-name-3      }

     [[AFTER data-name-4       FROM {numeric-literal-3}
                                    {data-name-5      }

     BY {numeric-literal-4}    UNTIL test-condition-2]
        {data-name-6      }

     [AFTER data-name-7        {FROM numeric-literal-5}
                               {     data-name-8      }

     BY {numeric-literal-6}    UNTIL test-condition-3]
        {data-name-9      }]

READ file-name RECORD [INTO data-name] AT END imperative-statement...

READ file-name RECORD [INTO data-name]
     INVALID KEY imperative statement...

STOP {RUN    }
     {literal}

SUBTRACT  {data-name-1          }...
          {numeric-literal-1    }
          {floating-point-literal-1}

       FROM {data-name-m [GIVING data-name-n]                  }
            {numeric-literal-m GIVING data-name-n              }
            {floating-point-literal-m GIVING data-name-n       }

                  [ROUNDED] [ON SIZE EROR Imperative statement...]

{READY}
{RESET}TRACE

TRANSFORM data-name-3 CHARACTERS
       FROM {figurative-constant-1} TO {figurative-constant-2}
            {non-numeric-literal-1}    {non-numeric-literal-2}
            {data-name-1          }    {data-name-2          }

WRITE record-name [FROM data-name-1] [AFTER ADVANCING   {data-name-2}  LINES]
                                                        {integer     }

WRITE record-name [FROM data-name-1]
     INVALID KEY imperative statement...

REWRITE record-name [FROM data-name]
     INVALID KEY imperative statement...
```

Permissible values for data-name-2:

Value	Interpretation
b (blank)	single spacing
0	double spacing
-	triple spacing
+	suppress spacing
1 through 9	skip to channel 1 through 9, respectively
A, B, C,	skip to channels 10, 11, 12, respectively
V, W	pocket select 1 or 2, respectively on the IBM 1442, or 2520 and P2 or RP3 on the IBM 2540.

Permissible Integers:

0 - skip to next-page
1 - skip 1 line
2 - skip 2 lines
3 - skip 3 lines

Permissible Comparisons.

	Second Operand									
First Operand	GR	AL	AN	ED	ID	BI	EF	IF	RP	FC
Group Item (GR)	NN	NN	NN	NN	NN	NN	NN	NN	NN	NN
Alphabetic Item (AL)	NN	NN	NN							NN[1]
Alphanumeric (non-report) Item (AN)	NN	NN	NN	NN[5]					NN	NN
External Decimal Item (ED)	NN		NN[5]	NU	NU	NU	NU	NU		NN[3]
Internal Decimal Item (ID)	NN			NU	NU	NU	NU	NU		NU[2]
Binary Item (BI)	NN			NU	NU	NU	NU	NU		NU[2]
External Floating-point Item (EF)	NN			NU	NU	NU	NU	NU		NU[2]
Internal Floating-point Item (IF)	NN			NU	NU	NU	NU	NU		NU[2]
Report Item (RP)	NN		NN						NN	NN[4]
Figurative Constant (FC)	NN	NN[1]	NN	NN[3]	NU[2]	NU[2]	NU[2]	NU[2]	NN[4]	

Abbreviations for Types of Comparison:
NN--Comparison as described for non-numeric items.
NU--Comparison as described for numeric items.
[1] Permitted with the figurative constants SPACE and ALL 'character' where character must be alphabetic.
[2] Permitted only if figurative constant is ZERO.
[3] Permitted only if figurative constant is ZERO or ALL 'character' where character must be numeric.
[4] Not permitted with figurative constant QUOTE.
[5] External decimal field must consist of integers.

Permissible Moves.

Source Field	Receiving Field								
	GR	AL	AN	ED	ID	BI	EF	IF	RP
Group (GR)	Y	Y	Y	N	N	N	N	N	N
Alphabetic (AL)	Y	Y	Y	N	N	N	N	N	N
Alphanumeric (AN)	Y	Y	Y	N	N	N	N	N	N
External Decimal (ED)	Y	N	Y[1]	Y	Y	Y	Y	Y	Y
Internal Decimal (ID)	Y	N	Y[1]	Y	Y	Y	Y	Y	Y
Binary (BI)	Y	N	Y[1]	Y	Y	Y	Y	Y	Y
External Floating-Point (EF)	Y	N	N	Y	Y	Y	Y	Y	Y
Internal Floating-Point (IF)	Y	N	N	Y	Y	Y	Y	Y	Y
Report (RP)	Y	N	Y	N	N	N	N	N	N
ZEROS	Y	N	Y	Y	Y	Y	Y	Y	Y
SPACES	Y	Y	Y	N	N	N	N	N	N
All 'character', HIGH-VALUES, LOW-VALUES, QUOTES	Y	N	Y	N	N	N	N	N	N

[1] For integers only

APPENDIX E. SUBROUTINES USED BY COBOL

A table of subroutines used by COBOL to accomplish the statements or actions specified follows. With the use of this table and the linkage editor cross-reference list, the programmer can determine the effect of his source statements. This table should guide the programmer in his efforts to conserve storage and isolate a trouble to a specific reason (debugging).

TABLE OF COBOL SUBROUTINES

SUBROUTINE NAME	ACTION
IHD00000 Converts an external floating-point number to an internal floating-point number.	Required for manipulation of external floating-point data in: MOVE - When send field is external floating point in MOVE statement. COMPUTATIONAL - When one field is external, and one field is internal floating point in computational statement.
IHD00100 Floating-point exponential subroutine.	Required for exponentiation to non-integer power.
IHD00200 Packed divides subroutine. It divides 16-byte 30-character dividend by a 1-byte 30-character divisor producing a 16-byte 30-character quotient. No registers are used.	Required for division of complex computes, COMPUTATIONAL of over 9 digits and COMPUTATIONAL-3 of over 16 digits.
IHD00300 Packed multiply subroutine. It multiplies two 30-character packed fields and produces a 60-character packed product.	Required for complex computes, COMPUTATIONAL fields of over 9, or COMPUTATIONAL-3 of over 16 digits.
IHD00400 Error message subroutine. It outputs object time messages.	Required with floating-point and non-integer exponentiation.
IHD00500 Packed exponentiation subroutine.	Required for exponentiation to an integer power. (Used with IEP00700 [floating-point exponentiation] subroutine.)
IHD00600 Floating-point logarithm subroutine.	Required whenever floating conversion is needed. Used with IEP00700 (floating-point exponentiation) subroutine.
IHD00700 Floating-point exponentiation subroutine.	Required to set up floating-point conversion routines for non-floating point exponentiation.

IHD00800 Converts packed decimal to floating point. Conversion is accomplished by calling two other subroutines IHD01600, which converts the number from packed decimal to binary and IHD01500, which converts the binary number to floating point and then returns.	May be required when floating-point and/or non-integer exponentiation is used. ARITHMETIC - Required when packed and floating-point operation are in the same statement. MOVE - Required if the sending field is packed and the receiving field is floating point in a move statement. COMPUTATIONAL - Required if one field is packed, and one field is floating point in a computational statement.
IHD00900 Converts floating-point numbers to zoned decimal numbers. Conversion is accomplished by calling two other subroutines; IHD01100, which converts the number from floating point to binary, and IHD01800, which converts the binary number to zoned decimal and returns.	ARITHMETIC - Required when there is a floating-point operand and the receiving field is zoned in an arithmetic statement. MOVE - Required if the sending field is floating point, and the receiving field is zoned in a move statement.
IHD01000 Converts a binary number to a packed decimal number. Used with IHD01300 (floating point to packed decimal) subroutine.	Required for: ARITHMETIC - Required when multiplying a binary field by a packed field or vice versa. - Required if multiplication is done in binary. MOVE - (Special Class) - If sending field is internal floating point, and receiving field is binary. The binary number must fall within the limits specified. (9 decimal digits <binary number <18 decimal digits.) - If sending field is binary and receiving field is binary. - If sending field is less than 9 and Receiving field is less than or equal to 9, or both are greater than 9 decimal digits. - If sending field is binary and receiving field is packed, and sending field is greater than 9 decimal digits. COMPUTATIONAL - If one field is binary and the other is zoned. - If one field is binary and the other is packed. - If both fields are binary and A is less than 10, B is less than 10, and the scales of both fields are equal. - If the scale of the sending field is greater than the scale of the receiving field, and the real or implied integer positions of the receiving field plus the scale of the sending field is less than 10. - If the scale of the sending field is less than the scale of the receiving field, and the real or implied decimal positions plus the scale of the receiving field is less than 10.

IHD01100 Converts an external floating-point number to a binary number. Used with IHD00900 (floating point to zoned decimal) subroutine, IEP01300 (floating point to packed decimal) subroutine, IHD01400 (floating point to binary) subroutine and IHD01900 (miscellaneous fields to external floating point) subroutine.	MOVE - Required when send field is external or internal floating point, and receiving field is external floating point.
IHD01200 Converts a zoned decimal number of a floating point number. Conversion is accomplished by calling the same subroutines used by IHD00900.	MOVE - Required when send field is zoned and receiving field is floating point. COMPUTATIONAL - Required when one field is zoned and the other field internal floating point.
IHD01300 Converts a floating point number to packed decimal format. Conversion is accomplished by calling IHD01100, which converts a floating-point number to binary, and IHD01000, which converts the binary number to packed decimal and then returns.	MOVE - Required when send field is external or internal floating point and receiving field is packed.
IHD01400 Converts an internal floating-point number to a binary format. Conversion is accomplished by calling subroutine IHD01100 which does the actual converting of the floating-point number to a binary number format.	MOVE - Required when sending field is external or internal floating point and receiving field is binary.
IHD01500 Converts a binary number into double precision floating-point. May be required when floating-point and/or non-integer exponentiation are used. Used with IHD00800 (packed to floating point) subroutine, IHD00000 (external floating point) subroutine, IHD01200 (zoned decimal to floating point) subroutine, IHD01900 (miscellaneous field type to external floating point) subroutine.	MOVE - Required when sending field is binary and receiving field is floating point. ARITHMETIC - Required when one operand is binary and one operand is floating point. COMPUTATIONAL - Required when one field is binary and one is internal floating point.

IHD01600 Converts either a packed decimal or a zoned decimal number to a binary number when receiving field is greater than 9 digits.	MOVE - Required: If the sending field is external decimal, and receiving field is packed, receiving field must be 9 decimal digits. COMPUTATIONAL - If one field is binary or zoned and one field is packed. - If both fields are binary and the following conditions are not met: • the length of the fields are unequal • A and B are both less than 10, and the scales of the fields are equal - If the scale of the sending field is greater than the scale of the receiving field and the real or implied integer positions of the receiving field plus the scale of the sending field is less than 10. - If the scale of the sending field is less than the scale of the receiving field and the real or implied decimal positions plus the scale of the receiving field is less than 10.
IHD01700 Compares two alphabetic fields of different lengths, no restriction on maximum length, when either or both fields are greater than 255 bytes.	COMPUTATIONAL - Required when either or both fields are 255 bytes.
IHD01800 Converts a binary number to a zoned decimal number. Used with IHD00900 (floating-point zoned decimal) subroutine.	ARITHMETICS - Required when operations are performed in binary and the receiving field is zoned. MOVE - Required when sending field is binary and receiving field is zoned. MISCELLANY - Required if user displays binary item.
IHD01900 Converts a field of any of the following formats to external floating point: external decimal, internal decimal, binary, internal floating point, figurative constant of zero. Conversion is accomplished in some cases by calling IHD01100, which converts internal floating point to binary, and IHD01500 which converts binary to external floating point.	MOVE - Required when receiving field is external floating point. MISCELLANY - Required if user displays internal floating point.
IHD02000	Used to move group items longer than 256 bytes.
IHD02100	Performs the class test on alphameric as specified in the publication IBM System/360 Operating System: COBOL Language.
IHD02200 Converts a packed decimal number to a zoned decimal number.	ARITHMETIC - Required when the operations are performed in packed, and the receiving field is zoned. MISCELLANY - Required if user displays packed format.

IHD02300	This subroutine consists of three parts: 1. The first part builds a table of the beginning and end addresses of the PERFORM or nested PERFORMS and the return address. It checks the validity of addresses. 2. The second part checks to see if the PERFORM is complete by comparing return addresses. 3. The third part deletes or eliminates the table entries by resetting pointers and counters. Required for object program compatibility with Version I COBOL.
IHD02400	Used to move fields when either, or both fields are variable groups. Requirements: R1 points to 'sending' field R2 points to 'receiving' field WORKA is length of 'sending' field WORKA+2 is length of 'receiving' field WORKA+4 is '01' if 'receiving' field is right-justified.
IHD02500	Used to compare two fields either or both of which are group variable. Used with fields defined with occurs depending on Requirements: R1 points to FIELD1. R2 points to FIELD2. WORKA is the same length as FIELD1. WORKA+2 is the same length as FIELD2.
IHD02600	Checks length of field to be displayed to be sure it fits into defined field, and moves display data to an output buffer. Used if a display data fit check is specified at object time. Requirements: WORKW - must be address of byte after buffer. WORKA+4 - must be number of bytes to move minus 1. R1 - points to next available buffer byte. R2 - points to data to be moved.
IHD02700	Writes out display data on SYSPUNCH. Used when display on SYSPUNCH is specified.
IHD02800	Writes out display data on SYSOUT. Required when EXHIBIT, TRACE, or standard DISPLAY statements are used (i.e., not UPON CONSOLE or UPON SYSPUNCH).
IHD02900	Reads a record from SYSIN and moves data to the field specified by data-name. Required when ACCEPT is specified (not ACCEPT FROM CONSOLE).
IHD03001	Required when QSAM or QISAM files are used.
IHD03002	Completes the creation of a relative file. Required when access sequential and organization relative clauses are used.
IHD03004	Structures buffers and directly organization files accessed sequentially. Required when access sequential and organization direct clauses are used.

IHD03008	Completes the creation of a direct file. Required when access sequential and organization direct clauses are used.
IHD03101	Synchronous error routine for QISAM. Required whenever QISAM files are used.
IHD03102	Check routine for BISAM. Required whenever BISAM files are used.
IHD03104	Synchronous error routine for QSAM and BSAM. Required whenever QSAM or BSAM files are used.
IHD03108	Synchronous error routine for BDAM. Required whenever BDAM is used.
IHD03300	If one field is divided by another and the divisor is zero, this subroutine links to the on size error routine.
IHD03402	Create direct organization files. Required when a write is given for a file with access sequential and organization direct.

APPENDIX F. SYSTEM/360 DIAGNOSTICS

This appendix contains a detailed description of system diagnostics. They consist of:

- System diagnostic messages
- Compiler diagnostic messages
- Load module execution diagnostic messages
  - Object time messages
  - Operator messages
- Debug packet error messages

These messages are produced during compilation and load module execution.

Certain conditions that are present when a module is being processed can cause linkage editor diagnostics. For a complete description of these messages, refer to the publication, IBM System/360 Operating system: Linkage Editor.

SYSTEM DIAGNOSTIC MESSAGES

System diagnostic messages consist of messages and completion codes, which are directed to the programmer by the IBM System/360 Operating System control program. The messages indicate coding errors found in job-control statements, system macro instructions, and errors detected during processing by the job scheduler. The completion codes indicate conditions causing the control program to abnormally terminate execution of a task. Where possible, appropriate user responses are suggested. For a complete list of system diagnostic messages, refer to the publication, IBM System/360 Operating System: Control Program Messages and Completion Codes.

COMPILER DIAGNOSTIC MESSAGES

Explanations and the action taken on compiler diagnostic messages are placed, in each case, after the particular error message. Where no action is indicated, the statement causing the message may be dropped. Although the messages are arranged in ascending numeric order, they are not necessarily numbered consecutively.

The sequence number counter associated with diagnostic messages cannot exceed 4095. At this point the sequence counter resets to zero. The nature of the message usually indicates the COBOL division in which the error was detected.

Accompanying each error message is a severity code: W = WARNING, C = CONDITIONAL, E = ERROR. For a description of these codes, refer to the discussion under Source Module Diagnostics. Where uniquely applicable, a System Action and/or a User Response accompanies the message.

UNEXPECTED DIAGNOSTICS

It is possible for the user to write COBOL source statements that can result in diagnostics being generated that do not appear in the list given. These diagnostic messages cover features of the compiler not supported at this time.

IEP001I C LITERAL EXCEEDS 120 CHARACTERS

System Action: The element count begins following the next quote on the line, if there is one, or following the element beginning after the 120th character.

User Response: Change the length of the literal so it does not exceed the allowed maximum, or insert the missing quote, or define the literal with two statements; execute the compilation again.

IEP002I W LITERAL CONTINUATION QUOTE INVALID IN MARGIN A

Explanation: The literal continuation quote should appear in Margin B.

System Action: The continuation is allowed.

IEP003I C LITERAL IMPROPERLY CONTINUED OR CONTINUATION QUOTE IS MISSING

Explanation: This may be the result of a missing quote sign on the preceding line.

System Action: The non-numeric literal is truncated at the end of the preceding line. The syntax scan resumes with the first element of the next line.

User Response: Check for missing quote, column 7 continuation hyphen, or improper formation of the non-numeric literal.

IEP004I C SYNTAX REQUIRES A BLANK AFTER A PERIOD OR THIS PERIOD IS INVALID DECIMAL POINT

System Action: The inverted print edit word with the invalid decimal point is dropped, and processing continues with the next word.

User Response: Check syntax of statement in error, and try again.

IEP005I C XXX EXCEEDS 30 CHARACTERS

Explanation: Any element that is not a non-numeric literal is truncated after 30 characters.

System Action: Normal processing continues with a literal made up of the first 30 characters.

User Response: Alter the length of the literal to conform with the specifications for this class of literal.

IEP006I C XXX REQUIRES QUALIFICATION

Explanation: This indicates that the name is defined in more than one location, and requires qualification in order to be unique.

System Action: The first name defined is used, and the compilation continues. If it is the name desired, the run will compile as desired. For further system action, see message IEP013I. It explains the handling for the procedure division statement.

User Response: Correct the procedural statements in error, or change the duplicate data names so they are unique. Execute the job again.

IEP007I C XXX HAS UNDEFINED QUALIFICATION

Explanation: One or more of the names in the qualification hierarchy are not defined as a group containing the data-name. This may have resulted from the dropping of a data-name because of an error at its point of declaration, or because of a misspelling.

System Action: The first name defined is used. If it is the name desired, the run will compile as desired.

User Response: Check for misspelling of the data-name, or the data-name's qualifier in the hierarchy order.

IEL008I C XXX REQUIRES MORE QUALIFICATION

Explanation: The number of qualifiers or the names are not sufficient to make the subject name unique. Another name could have the same qualification.

System Action: The first name defined is used, and the compilation continues. If it is the name desired, the run will compile as desired. For further system action, see message IEP013I. It explains the handling for the procedure division statement.

IEP009I E SUBSCRIPTED 88 MUST HAVE A RIGHT PARENTHESIS. WILL BE TREATED AS A DATA NAME

IEP010I W SYNTAX REQUIRES A BLANK AFTER A RIGHT PAREN, SEMICOLON AND OR COMMA

Explanation: Normal processing continues.

IEP011I E XXX IS UNDEFINED

IEP012I C XXX HAS MORE SUBSCRIPTS THAN DECLARED IN THE DATA DIVISION

Explanation: The Procedure Division reference to the data-name has too may

subscripts. The number of subscripts must match the number of OCCURS clauses in the definition hierarchy in the Data Division.

System Action: Normal processing continues with the next word.

IEP013I C RECORD-NAME 'XXX' IS ASSOCIATED WITH INVALID FD ENTRY

Explanation: The FD associated with the SELECT clause is invalid.

System Action: The error attribute for the record is output, and normal processing continues with the next word.

User Response: Check FD entries for proper device labels, requires clauses, missing period terminator, etc.

IEP023I C COPY AND INCLUDE MUST NOT BE USED WITHIN LIBRARY ENTRIES

System Action: Words following the library name are diagnosed according to the clause being processed, up to the next required clause.

IEP024I C PERIOD MISSING FOLLOWING XXX. THE NEXT CARD MAY BE SKIPPED.

System Action: For the Data Division COPY statement - Any other entry following the name is diagnosed as the missing period and the return is made to the phase. The phase diagnoses all entries up to the next period according to the current clause string. Normal processing continues. For the Procedure Division INCLUDE statement - Interrogation of the library name continues to determine its validity and whether or not it is on the library. If the library name is valid, and it is found, normal processing continues.

User Response: A period should be inserted following library book name.

IEP025I C XXX IS AN INVALID LIBRARY NAME OR NOT FOUND ON LIBRARY

Explanation: The library name may have been misspelled, not previously cataloged or not properly terminated with a quote.

System Action: Any word other than period immediately following the library name is diagnosed according to the current clause string up to the next period. This includes the current card and the next card, if read.

User Response: Check for the possible causes given in the explanation.

IEP026I C FLOATING-POINT NUMBER XXX IS BELOW OR ABOVE VALID RANGE

System Action: The value zero is assumed.

IEP027I W NUMBER OF DECIMALS IN LITERAL XXX AND DATA ENTRY DISAGREE

System Action: Truncation or padding is performed according to the rules governing the MOVE verb.

IEP028I C LITERAL XXX IS INVALID AND IS DROPPED

Explanation: The value clause conflicts with the description of the entry.

System Action: The value clause is dropped.

IEP029I W LITERAL XXX AND PICTURE SIZE DISAGREE

Explanation: This diagnostic indicates a literal that is larger than its picture.

System Action: The literal is truncated to picture size from left to right, unless right justification is specified. The scan is continued as if no error occurred.

IEP030I W LITERAL XXX WAS SIGNED, ENTRIES PICTURE WAS UNSIGNED

Explanation: The literal encountered in this entry

contains a sign, it does not appear as part of the entry because the picture is unsigned.

IEP031I W NUMBER OF INTEGERS IN LITERAL XXX AND DATA ENTRY DISAGREE

System Action: Same as for message IEP027I.

IEP032I C LIBRARY NAME IS AN INVALID EXTERNAL NAME OR NOT ON THE LIBRARY.

Explanation: The library name may have been misspelled, not cataloged, or not properly terminated with a quote.

System Action: The invalid or not found library name is dropped and the next card is read.

IEP041I C THIS CLAUSE IGNORED AT THE 01 LEVEL IN XXX ENTRY

Explanation: The occurs clause not valid as an 01 or 88 entry.

System Action: The clause is dropped.

User Response: Alter the clause's level number to one that is valid or remove the occurs from the statement in error.

IEP042I C THIS CLAUSE IGNORED IN XXX ENTRY AS IT PROVIDES MORE THAN 3 LEVELS OF SUBSCRIPTING

IEP043I C DEPENDING ON OPTION IN XXX ENTRY IS IGNORED DUE TO PRIOR USE

IEP44I C DEPENDING ON OPTION IN XXX ENTRY IS IGNORED BECAUSE IT IS SUBORDINATE TO A PREVIOUS CLAUSE

IEP045I C THE LEVEL OF XXX ENTRY INVALIDATES THE DEPENDING OPTION AT THE PRECEDING XXX ENTRY. THE DEPENDING OPTION IS DROPPED

Explanation: The level number just encountered indicates that there was an occurs depending that did not include the last entry within the 01.

System Action: The depending option is dropped.

IEP046I C XXX ENTRY CONTAINS AN ILLEGAL LEVEL NUMBER OR REDEFINES CLAUSE WHICH IS IGNORED

Explanation: A redefines clause must redefine an entry at the same level number.

System Action: The level number or the redefines clause is ignored.

User Response: Alter the level number or relocate the redefines clause to conform with the specification.

IEP047I E INTERNAL QUALIFIER TABLE OVERFLOWED WHEN HANDLING XXX. RESTARTED QUALIFIERS WITH XXX.

Explanation: The sum of all the characters in the data-name and all its qualifiers + 4 times (the number of qualifiers+1) must not exceed 300.

IEP048I W ENTRY PRECEDING XXX IS OF VARIABLE LENGTH

IEP049I W XXX IS LARGER THAN ENTRY REDEFINED

Explanation: The current entry is larger than the area redefined.

System Action: The area is assumed to be expanded.

User Response: The redefined area may be expanded.

IEP050I W XXX ENTRY PRECEDING XXX IS LARGER THAN ENTRY REDEFINED

Explanation: The same as for message IEP049I, only for a group entry

System Action: Same as for message IEP049I.

IEP051I C THIS CLAUSE INVALID IN XXX ENTRY AS REDEFINED AREA IS SUBSCRIPTED.

Explanation: It is invalid to redefine an area containing an occurs clause.

System Action: The redefinition clause is dropped.

IEP052I C THIS CLAUSE IGNORED IN XXX ENTRY DUE TO REDEFINES OR OCCURS CLAUSE IN PRECEDING XXX LEVEL

Explanation: A value clause cannot appear in an entry subordinate to a redefines clause.

System Action: The value clause is dropped.

IEP053I W FOR PROPER ALIGNMENT, A XXX BYTE LONG FILLER ENTRY IS INSERTED PRECEDING XXX

Explanation: Binary or floating point data improperly aligned for computations.

System Action: Binary and floating-point data are aligned on an appropriate boundary by the compiler. The alignment is performed by inserting an assumed filler entry preceding the item requiring alignment.

User Response: The number of slack bytes required can be reduced by the use of a different data format such as: internal decimal, grouping aligned items to the beginning of a record, or otherwise positioning them so that they will have the proper alignment within the record. A discussion of slack bytes can be found in the publication, IBM System/360 Operating System: COBOL Language.

IEP054I W FOR PROPER ALIGNMENT, A XXX BYTE LONG XXX FILLER ENTRY IS INSERTED PRECEDING XXX

Explanation: Binary or floating-point data is improperly aligned for computations.

System Action: Groups are aligned according to the alignment requirements of the first elementary within that group. The level number indicated in the diagnostic message shows exactly where the implied filler entry was inserted. For further explanation, see message IEP053I.

IEP055I E XXX ENTRY PRECEDING XXX EXCEEDS MAXIMUM SIZE OF 4092 BYTES

Explanation: The group defined at the indicated level preceding the point where this message was generated exceeded the maximum size permitted in the file or linkage section.

System Action: The compilation is continued, but execution is not attempted.

User Response: Reduce the record size to the allowable maximum size.

IEP056I W XXX ENTRY PRECEDING XXX EXCEEDS MAXIMUM LENGTH OF 32,768 BYTES

Explanation: See message IEP055I. It applies to Working-Storage section.

System Action: See message IEP055I. It applies to the Working-Storage section.

IEP057I E PROGRAM EXCEEDS 240 BASE LOCATORS MAXIMUM AT XXX

Explanation: A base locator is assigned for each file for each 01 or 77 in the linkage section, and for every 4,096 bytes in the working-storage section.

System Action: The base locator counter wraps around and the results are unpredictable.

User Response: Reduce the number of base locators.

IEP058I E ERRONEOUS OR MISSING DATA DIVISION

Explanation: No data division entries were present.

System Action: All data division entries were present.

System Action: All data division entries were dropped because of errors.

IEP059I E SYMBOLIC KEY NOT ALLOWED WITH SEQUENTIAL ACCESS METHOD ON FILE 'XXX'

IEP060I W XXX LEVEL PRECEDING XXX IS OF VARIABLE LENGTH

Explanation: The entry, defined at the level indicated, that preceded this clause, contained an occurs depending clause.

System Action: The redefined clause is dropped because it is illegal to redefine a variable-length entry.

IEP061I C XXX ENTRY EXCEEDS MAXIMUM LENGTH FOR ITS DATA TYPE

Explanation: The maximum permitted length of an entry depends on the type of data defined for that entry. Numeric data cannot exceed 18 digit positions, report entries cannot exceed 127 character positions.

System Action: The maximum size is used.

IEP062I W XXX REQUIRED ALIGNMENT AND STARTS XXX BYTES PAST THE START OF THE ENTRY IT REDEFINED

Explanation: The entry containing the redefines clause requires alignment that differs from the alignment of the clause redefined. If alignment is required, insert a filler the size of the number of bytes indicated in the message before the item being redefined.

IEP063I W TO ALIGN BLOCKED RECORDS ADD XXX BYTES TO THE 01 CONTAINING DATA NAME XXX

Explanation: The first record in a buffer is aligned on a double word boundary. All 01's are assumed to start on a double word boundary. If binary or floating-point numbers are used in the record and if the records are blocked in a buffer, the succeeding records may not be properly aligned. Alignment can be obtained by padding each record by the indicated number of bytes and processing in the buffer, or by moving each record, as a group, to an 01 in the working storage section before processing the computational field. The pointer to this diagnostic indicates the last element within a record. The padding must go into the preceding 01 record, not the 01 that may immediately follow the indicated data name.

IEP064I W IF THE PRECEDING RECORD IS BLOCKED, IT MAY BE ALIGNED BY MOVING TO AN 01 IN THE WORKING-STORAGE SECTION

Explanation: When records are variable and blocked, only the first record can be aligned.

IEP076I W INTEGER OPTION IS NOT PERMITTED

System Action: The clause is dropped.

IEP077I E USER LABELS NOT SUPPORTED IN THIS VERSION

IEP078I C INTERNAL FILE-NAME AND DESCRIPTION TABLE OVERFLOWED. XXX NOT PROCESSED

Explanation: There is a fixed number of files that can be handled by a given COBOL compilation (25). If additional files must be handled, they can be processed in a subprogram and accessed via the linkage facility.

System Action: Any files encountered after the maximum permitted are dropped. The maximum permitted is 25.

IEP079I C RESTRICTED SEARCH INTEGER TOO LARGE ON XXX. CLAUSE DROPPED

IEP080I C MORE THAN THREE FORMS OVERFLOW CLAUSES. OVERFLOW-NAME XXX

ENTRY IS DROPPED

IEP081I W XXX APPEARED PREVIOUSLY IN A 'SAME' CLAUSE. REMAINDER OF 'SAME' CLAUSE DROPPED

Explanation: A given filename can appear in only one same-area clause. Any duplication encountered is dropped.

System Action: The entire same-area clause is dropped.

User Response: Eliminate the duplicate statement.

IEP082I W INTERNAL 'SAME' TABLE OVERFLOW. ENTRIES AFTER XXX DROPPED

Explanation: A fixed number of filenames and combinations of filenames are allowed in an internal same-area table. If reducing the number of filenames or the number of same-area clauses does not relieve the situation, it may require an entry to a subprogram to permit a large number of files to be referenced in this manner.

IEP083I W RECORD LENGTH SPECIFIED DISAGREES WITH CALCULATED MAX. RECORD LENGTH OF XXX ON XXX. CALCULATED RECORD LENGTH ASSUMED.

Explanation: The actual length of each record is calculated during compilation time by totaling all its components. If the length disagrees with the specified maximum, this warning diagnostic is given to indicate that the specified record size is ignored.

IEP084I W BLOCK SIZE FOR XXX TOO BIG. 32K ASSUMED

Explanation: The integer specifying block size of the referenced files is too large.

System Action: The maximum size allowed is used.

IEP087I C THE XXX FILE MUST BE DESCRIBED IN A SELECT CLAUSE. CURRENT ENTRY IGNORED

Explanation: The subject file was referenced in the environment division or in an FD clause. There is no select clause to define this file. The filename referenced may be an invalid entry encountered at the point that a filename was expected.

IEP088I C LABEL RECORD DATA-NAME MUST BE DEFINED IN LINKAGE SECTION

System Action: Label records are assumed standard.

IEP090I C THE DESCRIPTION OF XXX FILE CONFLICTS ON THE FOLLOWING POINTS --- XXX

Explanation: The description of the file referenced contains factors that conflict with each other. The factors can be in the description of the file in the environment division, in the FD of the file section, or in other areas such as the record description for that file.

System Action: The points in conflict are defined by the trailing clauses of the diagnostic.

IEP093I E XXX NOT HANDLED WITH PRESENT RELEASE

IEP094I E XXX FILE WAS NOT DEFINED BY AN FD ENTRY

Explanation: No data control block is built for this file, therefore, it cannot be used.

IEP096I W ONLY ONE CHECKPOINT FILE MAY BE SPECIFIED

IEP098I C XXX FILE ASSUMED TO BE UTILITY

IEP099I C XXX FILE UNIT MISSING AND ASSUMED TO BE 1403 PRINTER

IEP100I E DIRECT-ACCESS ASSIGNED TO XXX NOT SUPPORTED IN THIS VERSION

IEP101I C XXX FILE IS ASSIGNED TO UNIT

RECORD AND MUST BE RECORDING MODE IS F

Explanation: Unit record must be fixed length.

System Action: The largest described length is assumed.

IEP102I C A MAXIMUM OF 1 ALTERNATE AREA IS ALLOWED FOR XXX FILE

System Action: One alternate area is reserved.

IEP106I W ONLY ONE AREA SUPPORTED FOR INDEXED OR DIRECT ORGANIZATION. ONE AREA ASSIGNED FOR XXX

IEP107I C RECORD KEY REQUIRED FOR INDEX ORGANIZATION FILE XXX

IEP108I E LENGTH OF SYMBOLIC/RECORD KEY GREATER THAN 255

IEP110I E INCORRECT DATA ITEM TYPE SPECIFIED FOR KEY

IEP112I C SYMBOLIC AND RECORD KEY LENGTH FOR XXX DISAGREE

IEP115I C LENGTH OF ACTUAL KEY IS GREATER/LESS THAN 5

IEP116I E FILE LIMIT VALID FOR DIRECT ORGANIZATION, SEQUENTIAL ACCESS OUTPUT FILES ONLY

IEP117I E SYMBOLIC KEY MUST BE SPECIFIED FOR XXX

IEP118I W ACTUAL KEY MUST BE SPECIFIED FOR OUTPUT FILES

IEP119I C ONLY ONE AREA SUPPORTED FOR OTHER THAN STANDARD SEQUENTIAL, AND INDEXED ORGANIZATION - SEQUENTIAL ACCESS FILES

IEP176I C WORD RECORD OR RECORDS IS REQUIRED. FOUND XXX.

Explanation: Syntax skips until the next clause, level number, or period at the end of the file description is encountered.

IEP177I W PERIOD REQUIRED AFTER WORD SECTION

IEP178I C SYNTAX REQUIRES XXX. FOUND XXX.

System Action: This clause is ignored.

IEP179I W XXX IS AN INVALID FILE-NAME FORMAT

Explanation: A filename must follow the format rules for data-names.

System Action: Invalid names are truncated to 30 characters and assumed to be valid.

IEP180I E XXX EXCEEDS 30 CHARACTERS AND IS DROPPED

System Action: The picture is too long, and is dropped.

IEP181I W THE OPTION WORD IS MISSPELLED OR OMITTED. FOUND XXX.

System Action: The usage assumed is display.

IEP183I C XXX IS AN INVALID OR EXCESSIVE INTEGER

Explanation: The integer indicated in this clause is determined to be invalid.

System Action: The integer is not used.

IEP184I W XXX IS AN INVALID LEVEL NUMBER

IEP185I W LABEL RECORDS IS OMITTED. LABELS ASSUMED STANDARD.

IEP186I W SYNTAX REQUIRES DATA RECORD CLAUSE

System Action: Syntax scanning proceeds.

IEP187I C MODE MUST BE V, F, OR U. FOUND XXX.

User Response: If V, F, or U was specified, check the element number on this line for a misspelled optional word.

IEP190I W XXX IS AN INVALID DATA-NAME FORMAT

System Action: The invalid data-name(s) are truncated to 30 characters and used.

IEP191I W SD OR SA ENTRY REQUIRES F LEVEL COMPILER

System Action: Syntax skips to next margin-A entry.

IEP192I W XXX IS AN INVALID RECORD-NAME FORMAT

System Action: Invalid record names are truncated to 30 characters and assumed valid.

IEP194I C XXX IS INVALID AT THIS POINT. CHECK FOR SYNTAX ERROR ON CURRENT/PREVIOUS STATEMENT

Explanation: While processing a given clause or sentence, an unexpected element was encountered. The clause may be valid but misplaced. This diagnostic is also given for clauses that are not valid source input to this level compiler.

User Response: Check for prior diagnostics, extra or missing period, invalid continuation of non-numeric literals or a misspelled word.

IEP195I E SYNTAX REQUIRES AN FD ENTRY. FOUND XXX.

IEP196I W SYNTAX REQUIRES AN 01 LEVEL ENTRY. FOUND XXX.

IEP197I W NOT VALID FOR THIS LEVEL COMPILER.

IEP201I C XXX IS AN INVALID DATA-NAME FORMAT BUT ASSUMED VALID.

System Action: Invalid data-names are truncated to 30 characters and assumed valid.

IEP202I C XXX IS INVALID AT THIS POINT. CHECK FOR SYNTAX ERROR OR CURRENT/PREVIOUS STATEMENT.

Explanation: The explanation and user response is the same as that for message IEP194I.

IEP203I C THIS USAGE XXX CONFLICTS WITH THE GROUP USAGE AND IS IGNORED

IEP204I C XXX IS AN INVALID OR EXCESSIVE INTEGER

System Action: The invalid integer is dropped.

IEP205I W XXX IS AN INVALID DATA-NAME FORMAT, BUT ASSUMED VALID.

IEP206I W WORD ZERO IS REQUIRED. FOUND XXX.

System Action: The clause is ignored.

IEP207I W WORD RIGHT IS REQUIRED. FOUND XXX.

System Action: The clause is ignored.

IEP210I C THIS ENTRY CONFLICTS WITH THE FOLLOWING DESCRIPTIONS ---XXX.

Explanation: Various clauses specified for a data entry are compared with previous specifications for the entry. If there is any factor that conflicts with the subject clause, it is listed as a trailer to this entry. Factors included that are not themselves clauses would be elementary or group item usage, specified at a group level in previous clauses. This message can appear if a period is missing at the end of a data entry or (for example) when the picture clause for the second entry is encountered, and automatically conflicts with the picture clause for the previous entry.

IEP211I C XXX EXCEEDS 30 CHARACTERS AND

IS TRUNCATED.

IEP212I C ONLY LEVELS 77 OR 01 ARE PERMITTED AT THIS POINT. FOUND XXX.

System Action: Syntax skips until a section name or level number is found.

IEP213I W THE FOLLOWING DESCRIPTIONS INVALID AT GROUP LEVEL ---XXX.

Explanation: The data entry described is determined to be a group, although the entries specified as trailers to this diagnostic are invalid at the group level. This diagnostic can be produced by an invalid level number that was changed to an 01, or a misunderstanding as to how a group is defined and what clauses are valid at the group level. A missing period can also produce this diagnostic.

IEP214I C XXX DATA ENTRY REQUIRES A PICTURE, COMPUTATIONAL-1 OR COMPUTATIONAL-2.

Explanation: This diagnostic can be produced by an error in the following level number which caused its level to be changed to an 01, thereby making this entry an elementary.

System Action: Any statement in the procedure division containing a reference to this entry is diagnosed and dropped.

User Response: Check for missing periods or other diagnostic messages.

IEP215I W SYNTAX REQUIRES AN ENTRY IN MARGIN A. FOUND XXX IN MARGIN B

System Action: Following certain entries in a source program, a specific clause must be encountered in margin A. If it is found in margin B, it is diagnosed but handled by the compiler.

User Response: Appropriately locate the clause in margin A.

IEP216I W SYNTAX REQUIRES AN ENTRY IN MARGIN B. FOUND XXX IN MARGIN A CHECK FOR MISSING PERIOD.

Explanation: All entries in margin A must be preceded by a period.

System Action: The compiler was in the middle of processing a clause or sentence and encountered the indicated word in margin A, thus a diagnostic is issued and the word is processed as if valid.

IEP217I W LEVEL 77 ENTRIES MUST PRECEDE OTHER LEVELS AND ARE ASSUMED TO BE 01 LEVEL.

IEP218I W SYNTAX PERMITS ONLY LEVELS 77, 88, OR 01 AFTER A 77 LEVEL. CHANGED XXX TO 01.

IEP221I C SYNTAX FOR ALL REQUIRES XXX BE A SINGLE CHARACTER IN QUOTES

System Action: The value clause is dropped.

IEP222I C PICTURE XXX WAS FOUND INVALID WHILE PROCESSING XXX. THE PICTURE IS DROPPED

Explanation: Any element that follows the word picture in a data description, other than the word that is dropped, is assumed to be a picture, and is passed to a later phase for analysis. The analysis proceeds from left to right on a character-by-character basis. The character identified in the message is the one processed at the time the picture is determined to be invalid. The specific character itself may be invalid or may have indicated that a previous character or condition is invalid. For example, an E encountered in an external floating-point picture may indicate that a preceding decimal was omitted in the mantissa.

System Action: The picture is dropped, and the entry

identified as an error.

IEP227I E FILE SECTION OUT OF SEQUENCE

IEP228I E SYNTAX PERMITS ONLY ONE XXX IN SOURCE PROGRAM

System Action: Syntax scan proceeds.

IEP229I E WORKING STORAGE SECTION OUT OF SEQUENCE

IEP233I C REPORT SECTION REQUIRES F LEVEL COMPILER

IEP234I W WORD SECTION MISSING

IEP235I W PERIOD MUST FOLLOW WORD SECTION

IEP237I E XXX IS MISPLACED

Explanation: The statement is probably out of place in the source deck; i.e., FD is WORKING-STORAGE

System Action: The statement is processed as it is, however, execution may not be as desired.

User Response: Properly locate the misplaced statement.

IEP238I W XXX IS AN INVALID SECTION NAME, A MISSING FD OR AN INVALID/MISPLACED LEVEL INDICATOR

System Action: Syntax skips until a valid section-name or level number is found.

IEP239I W SYNTAX REQUIRES 'DIVISION'

IEP241I C LEVEL PRECEDING 88 MUST BE AN ELEMENTARY.

Explanation: Any level number preceding an 88 entry must be an elementary.

System Action: If the level number preceding an 88 is not an elementary, it is assumed to be one and is processed.

IEP242I W THE 88 ENTRY DOES NOT HAVE A VALUE, THEREFORE, IT IS DROPPED.

IEP301I W SYNTAX REQUIRES XXX IN MARGIN A. FOUND XXX. RESTART WITH XXX.

Explanation: Syntax requires the specific entry indicated to be in margin A. If the entry is found in margin B, compilation resumes.

IEP302I C SYNTAX REQUIRES XXX. FOUND XXX. RESTART WITH XXX. IF WORDS REQUIRED AND FOUND ARE THE SAME, THE ENTRY IS IN THE WRONG MARGIN.

System Action: Syntax skipped to the restart clause.

IEP303I W XXX IS AN INVALID CONDITION-NAME FORMAT.

Explanation: The name shown is an invalid condition name.

System Action: The name is truncated to 30 characters and assumed valid.

IEP304I E XXX IS AN INVALID EXTERNAL-NAME FORMAT. RESTART WITH XXX.

Explanation: An external name was expected at this point in the scan of the subject clause. An external name must be enclosed in quotes. It must start with an alphabetic character, cannot contain more than eight characters, and the only valid characters are letters and numerals. A dash is not permitted.

IEP305I C SYNTAX REQUIRES SAME, RERUN, APPLY, OR XXX DIVISION. FOUND XXX. RESTART WITH XXX.

User Response: Check for invalid sequence of source program cards or extra periods.

IEP306I W SYNTAX REQUIRES ENVIRONMENT OR XXX DIVISION IN MARGIN A. FOUND XXX. RESTART WITH XXX.

User Response: Same as for message IEP305I.

IEP307I E SYNTAX REQUIRES I-O-CONTROL INPUT-OUTPUT, OR XXX DIVISION IN MARGIN A. FOUND XXX. RESTART WITH XXX.

User Response: Same as for message IEP305I.

IEP308I W XXX IS AN INVALID DATA-NAME FORMAT. RESTART WITH XXX.

Explanation: A data-name was expected at this point in the scan of the subject clause.

System Action: Invalid format is truncated to 30 characters and processed as if valid.

IEP309I C ENVIRONMENT PARAGRAPHS OUT OF ORDER.

System Action: Statements are handled anyway.

IEP310I W XXX IS AN INVALID 360 MODEL-NUMBER. RESTART WITH XXX.

System Action: Syntax scan skips to the restart clause.

IEP311I E SYNTAX REQUIRES FILE-CONTROL, XXX OR DATA DIVISION IN MARGIN A. FOUND XXX. RESTART WITH XXX.

User Response: Same as for message IEP305I.

IEP312I C XXX IS AN INVALID OR EXCESSIVE INTEGER. RESTART WITH XXX.

Explanation: The syntax at this point of scan of the specified clause requires an integer.

System Action: The element found was invalid and is dropped.

IEP313I W XXX IS AN INVALID FILE-NAME FORMAT. RESTART WITH XXX.

Explanation: The syntax scan of the subject clause requires a filename at this point.

System Action: The element found was invalid. It was truncated to 30 characters and used as if valid.

IEP314I E XXX IS AN INVALID LIBRARY-NAME FORMAT. RESTART WITH XXX.

Explanation: A library name is required at this point.

System Action: The format is invalid. It is dropped.

IEP315I W - MORE THAN THREE OVERLFOW OPTION CLAUSES ARE USED.

Explanation: An internal table permits a maximum of three form overflow names to be assigned in any compilation.

System Action: All form overflow names in excess of the maximum allowed (three) are dropped.

IEP316I C SYNTAX REQUIRES INDEXED OR XXX. FOUND XXX. RESTART WITH XXX.

Explanation: This message applies to a direct access storage device only.

IEP317I C SYNTAX REQUIRES SEQUENTIAL OR XXX. FOUND XXX. RESTART WITH XXX.

Explanation: This message applies to a direct access storage device only.

IEP318I E SYNTAX REQUIRES XXX OR DATA DIVISION IN MARGIN A, OR SELECT IN MARGIN B. FOUND XXX. RESTART WITH XXX.

Explanation: The syntax for the specific clause requires specific entries at this point.

User Response: Check for misspelled words, or excessive periods.

IEP319I C SYNTAX REQUIRES UTILITY, DIRECT-ACCESS OR XXX. FOUND XXX. RESTART WITH XXX.

Explanation: Same as for

message IEP318I.

IEP320I W XXX IS AN INVALID I-O-DEVICE-NUMBER. RESTART WITH XXX.

Explanation: Same as for message IEP318I.

IEP321I E NO PROCESSING OF THIS MULTIPLE SPECIFIED DIVISION OR SECTION. RESTART WITH XXX.

Explanation: A section or division was encountered more than once.

System Action: The additional section or division is dropped, rather than disturb the internal sequence of the compilation.

IEP322I W FILE-NAME OR DATA-NAME EXCEEDS 30 CHARACTERS. TREATED AS 30-CHARACTER NAME.

IEP323I W SYNTAX REQUIRES XXX OR CLAUSE-NAME. FOUND XXX. RESTART WITH XXX.

System Action: Syntax scan skips to the restart clause.

IEP324I E SYNTAX REQUIRES REEL OR XXX. FOUND XXX. RESTART WITH XXX.

System Action: Syntax scan skips to restart clause.

IEP401I C SYNTAX REQUIRES A DATA-NAME. FOUND XXX.

Explanation: The syntax of the indicated clause requires a data-name. The element found was not defined as a valid data-name. The element may be indicated here, or, an indication given that it was an invalid name such as, filename, condition name, figcon, or overflow name.

System Action: The compilation continues at the next verb or paragraph label.

User Response: Check for misspelled data-name in diagnostics, which would nullify the definition of a valid data-name, or the use of a COBOL word as a data-name.

IEP402I C SYNTAX REQUIRES NEXT ITEM BE XXX.

Explanation: The syntax for this clause requires a specific word that was not found. The item encountered was probably a data-name. The next item indicates that the syntax requires a specific word or words. None were found.

System Action: The element found is displayed unless it was a name, in which case the word invalid name or data name is indicated. Compilation continues at the next verb or paragraph level.

User Response: The reference format for the clause specified should be consulted if the meaning of the message is not immediately clear. Also check for: missing periods, preceding diagnostic messages, invalid non-numeric literals, COBOL words used as data-names.

IEP403I C SYNTAX REQUIRES A DATA-NAME OR NUMERIC-LITERAL. FOUND XXX.

Explanation: See message IEP402I.

IEP404I C SYNTAX REQUIRES EITHER WORD TO, OR GIVING. FOUND XXX.

Explanation: See message IEP402I.

IEP405I C SYNTAX REQUIRES A SINGLE CHARACTER IN QUOTES OR A FIGCON. FOUND XXX.

Explanation: See message IEP402I.

IEP406I C SYNTAX REQUIRES A FILE-NAME. FOUND XXX.

Explanation: See message IEP402I.

IEP407I C SYNTAX REQUIRES DATA-NAME OR INTEGER. FOUND XXX.

Explanation: See message IEP402I.

IEP408I C SYNTAX REQUIRES WORK INPUT, OUTPUT, OR I-O. FOUND XXX.

Explanation: See message IEP402I.

IEP409I C SYNTAX REQUIRES A PROCEDURE-NAME. FOUND XXX.

Explanation: See message IEP402I.

IEP410I C SYNTAX REQUIRES A DATA-NAME OR LITERAL. FOUND XXX.

Explanation: See message IEP402I.

IEP411I C SYNTAX REQUIRES WORD CALL, ENTRY, OR RETURN. FOUND XXX.

Explanation: See message IEP402I.

IEP412I E SYNTAX REQUIRES AND EXTERNAL-NAME. FOUND XXX.

Explanation: See message IEP402I.

IEP413I C SYNTAX REQUIRES =. FOUND XXX.

Explanation: See message IEP402I.

IEP414I C SYNTAX REQUIRES EXPRESSION TO BEGIN WITH EITHER A DATA-NAME, NUMERIC-LITERAL, +, -, OR (. FOUND XXX. TWO OPERATORS MAY NOT APPEAR ADJACENT TO ONE ANOTHER.

Explanation: See message IEP402I.

IEP415I C SYNTAX REQUIRES CALL PARAMETERS TO BE EITHER DATA-NAME, PROCEDURE-NAME OR FILE-NAME. FOUND XXX.

Explanation: See message IEP402I.

IEP416I C SYNTAX REQUIRES DATA-NAME, LITERAL, FIGCON, +, -, (OR NOT. FOUND XXX.

Explanation: See message IEP402I.

IEP417I C SYNTAX REQUIRES ARITHMETIC OPERATOR OR RELATIONAL. FOUND XXX.

Explanation: See message IEP402I.

IEP418I C SYNTAX REQUIRES A DATA-NAME, NUMERIC-LITERAL, OR (AFTER AN OPERATOR. FOUND XXX.

Explanation: See message IEP402I.

IEP419I C SYNTAX REQUIRES A DATA-NAME, LITERAL, FIGCON, (, + OR - AFTER A RELATIONAL. FOUND XXX.

Explanation: See message IEP402I.

IEP420I C SYNTAX REQUIRES A VERB, PERIOD, ELSE OR OTHERWISE. FOUND XXX.

Explanation: The end of a valid clause was encountered. The element that followed the valid termination of this clause is not valid.

System Action: Compilation continues at the next verb or paragraph label.

User Response: If the preceding clause had some options, check the reference format to determine if the options were specified incorrectly. A COBOL word used as a data-name, or an extra period, can also produce this diagnostic.

IEP421I C ENTRY PARAMETER MUST BE A DATA-NAME. FOUND XXX.

Explanation: The only parameters that can be passed to a COBOL subprogram are data-names. The data-names must be defined in the linkage section of the subprogram.

System Action: Compilation continues at the next verb or paragraph label.

IEP422I C SYNTAX REQUIRES A RELATIONAL.

FOUND XXX.

Explanation: Syntax requires that the next element be a relational.

System Action: Compilation continues at the next verb or paragraph label.

User Response: Check for invalid punching or a preceding error.

IEP423I C SYNTAX REQUIRES WORD INPUT OR OUTPUT. FOUND XXX.

Explanation: See message IEP402I.

IEP424I C SYNTAX REQUIRES WORDS - TO PROCEED TO -. FOUND XXX.

Explanation: See message IEP402I.

IEP425I C SYNTAX REQUIRES WORD CONSOLE OR SYSPUNCH. FOUND XXX.

Explanation: See message IEP402I.

IEP427I C SYNTAX REQUIRES A DATA-NAME, FIGCON OR NON-NUMERIC LITERAL. FOUND XXX.

Explanation: See message IEP402I.

IEP428I C SYNTAX REQUIRES A PROCEDURE-NAME AFTER -GO TO - NOT PRECEDED BY A PARAGRAPH-NAME. FOUND XXX.

Explanation: See message IEP402I.

IEP429I C SYNTAX REQUIRES ALL, LEADING, UNTIL, OR FIRST. FOUND XXX.

Explanation: See message IEP402I.

IEP430I C SYNTAX REQUIRES WORD TALLYING OR REPLACING. FOUND XXX.

Explanation: See message IEP402I.

IEP431I C SYNTAX REQUIRES WORD - DEPENDING ON -. FOUND XXX.

Explanation: See message IEP402I.

IEP432I C DATA TYPE MUST BE ED, ID OR BI.

Explanation: Valid syntax for the subject verb permits only specific data types. The data type as determined by the definition in the data division is invalid for its use here.

System Action: The statement is dropped from the point of error.

IEP433I C SYNTAX REQUIRES WORD TRACE. FOUND XXX.

Explanation: See message IEP402I.

IEP434I C SYNTAX REQUIRES THAT A PERIOD OR SECTION FOLLOWS PARAGRAPH-NAME. FOUND XXX.

Explanation: See message IEP402I.

IEP435I E DATANAME AND ANY QUALIFIER MUST APPEAR WITHIN THE FIRST SEVEN OPERANDS OF STATEMENT FOR CHANGED OPTION.

Explanation: See message IEP402I.

IEP436I C SYNTAX REQUIRES A DATA-NAME, FIGCON OR LITERAL. FOUND XXX.

Explanation: See message IEP402I.

IEP437I C SYNTAX REQUIRES A FIGCON. FOUND XXX.

Explanation: See message IEP402I.

IEP438I C SYNTAX REQUIRES DATA-ITEM TO BE NO LONGER THAN FOUR.

Explanation: See message IEP402I.

IEP439I C WRONG SUBSCRIPT SPECIFICATION.

Explanation: Data names and

condition names can be subscripted to a depth of three. A subscript is required for each occurs clause specified at the specified data name or in groups containing that data name.

System Action: The compilation continues at the next verb or paragraph label.

User Response: Check for fewer or more subscripts than occurs clauses in the hierarchy. Subscripts must be enclosed in parentheses, and separated from each other by a comma or a blank.

IEP440I C INCORRECT SPECIFICATION IN DECLARATIVE-SECTION. FOUND XXX.

Explanation: See message IEP402I.

IEP441I C SYNTAX REQUIRES AN INTEGER NOT LONGER THAN 5. FOUND XXX.

Explanation: The integer exceeds the size permitted by language specifications.

System Action: The compilation continues at the next verb or paragraph label.

IEP442I C THE DECLARATION OF THIS DATA-NAME CAUSED IT TO BE FLAGGED AS AN ERROR.

Explanation: The data-name encountered was flagged by the data division as containing an error in its declaration.

System Action: Compilation continues at the next verb or paragraph label.

User Response: Correct the declaration as indicated by the data division diagnostics and recompile.

IEP443I E SYNTAX REQUIRES A VERB. FOUND XXX.

Explanation: A point was reached where a verb was required, and was missing. For example 'IF = B.' requires a verb between B and period.

System Action: The statement is skipped from the point of the error.

IEP444I E SYNTAX REQUIRES A RECORD NAME. FOUND XXX.

Explanation: See message IEP402I.

IEP500I W AN OPERAND'S LENGTH EXCEEDS AND TRUNCATED TO 256 BYTES

Explanation: The maximum number of bytes that can be displayed is 256.

System Action: The operand is truncated to 256 bytes and displayed.

IEP501I W IF THIS VARIABLE-LENGTH ENTRY EXCEEDS 256, RESULTS WILL BE UNPREDICTABLE.

Explanation: A maximum of 256 bytes can be displayed.

System Action: The entry is truncated to 256 bytes and displayed.

IEP502I W LITERAL EXCEEDS AND IS TRUNCATED TO 72 BYTES.

System Action: In a stop-literal statement only the first 72 bytes of a longer field are typed on the console.

IEP503I W DATA EXCEEDS AND IS TRUNCATED TO 72 BYTES.

Explanation: A maximum of one line (72 bytes) can be retrieved using the ACCEPT FROM CONSOLE statement.

IEP504I W DATA EXCEEDS AND IS TRUNCATED TO 256 BYTES.

Explanation: A maximum of 256 bytes can be accepted from SYSIN.

IEP505I C FILENAMES OR STERLING-DATATYPE NOT ALLOWED IN COMPARE.

Explanation: See message

IEP506I.

IEP506I C USAGE OF DATA-TYPES CONFLICT. THE TEST DROPPED.

Explanation: Only certain data types can be compared to each other. The types specified are invalid. Reference can be made to the compared table to determine the valid combinations. Logical compares of fields that are classified as invalid compares can often be made through a redefinition, and a description of one or both of the fields as alphanumeric.

IEP507I W EXIT MUST BE ONLY STATEMENT IN PARAGRAPH.

System Action: Compilation continues normally.

IEP508I E THE STATEMENT CONTAINS AN UNDEFINED DATANAME.

Explanation: See message IEP402I.

IEP509I C AN ALPHABETIC DATA-NAME CAN BE TESTED ONLY FOR ALPHABETIC OR NOT ALPHABETIC, AND NUMERIC DATA-NAME ONLY FOR NUMERIC OR NOT NUMERIC, THE TEST IS DROPPED.

IEP510I C COMPARISON OF TWO LITERALS OR FIGCONS IS INVALID.

Explanation: See message IEP506I.

IEP511I C DATA-TYPE IN ARITHMETIC STATEMENT IS NOT NUMERIC OR RECEIVING FIELD IS NOT NUMERIC OR REPORT.

Explanation: See message IEP506I.

IEP512I C DATA-NAME IN CLASS-TEST MUST BE AN, ED, OR ID.

Explanation: See message IEP506I.

IEP513I C DATA-NAME IN SIGN-TEST MUST BE NUMERIC.

Explanation: See message IEP506I.

IEP514I W DATA EXCEEDS AND IS TRUNCATED TO 72 BYTES.

System Action: If the data is longer than 72 bytes, only the first 72 bytes are printed for DISPLAY ON CONSOLE statement.

IEP515I W DATA EXCEEDS AND IS TRUNCATED TO 120 BYTES.

System Action: If the data is longer than 120 bytes, only the first 120 bytes are printed for a DISPLAY statement.

IEP516I C OPEN 'NO REWIND' OR 'REVERSED' CANNOT BE SPECIFIED FOR A UNIT RECORD, DIRECT-ACCESS OR DISK/DATA CELL UTILITY FILE.

System Action: The options are ignored.

IEP517I C 'NO REWIND' OR 'LOCK' CANNOT BE SPECIFIED FOR A UNIT RECORD, DIRECT-ACCESS OR DISK/DATA CELL UTILITY FILE.

System Action: The options are ignored.

IEP518I E MORE THAN FORTY PARAMETERS ARE NOT ALLOWED WITH THE STATEMENT.

IEP519I C SYNTAX ALLOWS ZERO AS ONLY VALID FIGCON IN A COMPARISON WITH BI, ID, EF, AND IF.

Explanation: See message IEP506I.

IEP520I C SYNTAX ALLOWS SPACE OR ALL AS ONLY VALID FIGCONS IN COMPARISON WITH AN ALPHABETIC FIELD.

Explanation: See message IEP506I.

IEP521I C DATATYPE MUST BE ED, EF, AL, AN, OR GF. FOUND XXX.

Explanation: The data types indicated are the only valid ones that can be used in the

clause indicated.

System Action: Compilation continues at the next verb or paragraph label.

IEP522I C SYNTAX REQUIRES WORD RUN OR LITERAL. FOUND XXX.

System Action: The syntax scan skips the rest of the statement.

IEP523I C RECEIVING FIELDS IN PRECEDING STATEMENT IS A LITERAL.

Explanation: A procedure division literal cannot be changed as the result of arithmetic or a move. The statement, SUBTRACT data name FROM literal, would specify invalid action of this type.

System Action: Compilation continues at the next verb or paragraph label.

IEP524I C SYNTAX REQUIRES AT LEAST TWO OPERANDS BEFORE GIVING OPTION.

Explanation: For example, ADD A GIVING B.

System Action: The statement is skipped.

IEP525I C THE EXPRESSION HAS MORE RIGHT PARENS THAN LEFT PARENS TO THIS POINT. FOUND XXX.

Explanation: The number of right parentheses and left parentheses in a statement must agree. At no point in time can there be more right parentheses than left parentheses.

System Action: The statement is skipped from the point of the error.

User Response: Check for extra periods or missing periods, an error in a non-numeric literal, or mispunched operators or subscripted fields that are invalidly packed together without an intervening blank.

IEP526I C THE EXPRESSION HAS UNEQUAL NUMBER OF RIGHT AND LEFT PARENS.

Explanation: See message IEP525I.

IEP527I C DATA-TYPE MUST BE ED, ID, OR BI, FOUND XXX.

System Action: The statement is skipped from the point of error.

IEP528I C VARYING OPTION EXCEEDS THREE LEVELS.

Explanation: A maximum of three levels is permitted with the varying option of the PERFORM verb.

System Action: The statement is dropped from the point of error.

IEP529I C DATA-TYPE MUST BE ED, ID, BI, EF, OR IF.

Explanation: The data types shown are the only valid ones. The data-name found is not one of these types.

System Action: The statement is skipped from the point of error.

IEP530I C NUMBER OF ELSES EXCEEDS NUMBER OF IFS.

Explanation: Number of else must balance out with the appropriate number of else or otherwise.

System Action: Statement is skipped from the point of error.

User Response: Recount and make corrections.

IEP531I E INTERNAL OCCURS-DEPENDING-ON TABLE OVERFLOWED AVAILABLE CORE

IEP532I E STATEMENT HAS TOO MANY OPERANDS

Explanation: The statement referenced is too large or complex for the internal tables needed for compilation.

System Action: The statement is skipped from the occurrence of this condition.

User Response: The statement should be divided into more than one statement.

IEP533I E PARENTHESIZING REQUIRES SAVING TOO MANY OPERANDS.

Explanation: See message IEP532I.

IEP534I E PARENTHESIZING REQUIRES SAVING TOO MANY INTERNALLY GENERATED LABELS.

Explanation: See message IEP532I.

IEP535I E PARENTHESIZING REQUIRES SAVING TOO MUCH OF STATEMENT

Explanation: See message IEP532I.

IEP536I E ARITHMETIC EXPRESSION REQUIRES MORE THAN 9 INTERMEDIATE RESULT FIELDS.

Explanation: See message IEP532I.

IEP537I C NOT HANDLED IN THIS VERSION

IEP549I E WORD XXX WAS EITHER INVALID OR SKIPPED DUE TO ANOTHER DIAGNOSTIC

Explanation: The majority of these messages will probably be caused by words skipped because of another diagnostic that occurred earlier in the statement. This diagnostic also occurs because of misspelled words.

User Response: In the case of words skipped, correct the previous error, or correct misspellings.

IEP550I C A FIGURATIVE CONSTANT IS NOT ALLOWED AS A CALL OR ENTRY PARAMETER.

System Action: The statement is skipped from the point of error.

IEP551I C SYNTAX REQUIRES WORD TO. FOUND XXX.

System Action: Syntax scan skips the rest of the statement.

IEP552I C RECEIVING FIELD MUST BE A DATA-NAME. FOUND XXX.

System Action: The statement is skipped from the point of error.

IEP553I E FIGURATIVE CONSTANT IS NOT ALLOWED AS A RECEIVING FIELD.

System Action: The statement is skipped from the point of the error.

IEP554I C THE XXX DATA-TYPE IS NOT A LEGAL RECEIVING FIELD.

System Action: The statement is skipped from the point of the error.

User Response: Check the table of permissible moves in the COBOL specification.

IEP555I C OVERFLOW NAME IS NOT A VALID SENDING FIELD.

System Action: The statement is skipped from the point of the error.

IEP556I E END DECLARATIVES IS MISSING FROM PROGRAM.

Explanation: The entire procedure division is treated as a declarative section.

IEP557I W FLOATING-POINT CONVERSION MAY RESULT IN TRUNCATION.

Explanation: Conversion of floating-point numbers can result in truncation of low-order digits.

IEP558I E I-O OPTION FOR FILE CONFLICTS WITH NO REWIND.

System Action: The statement is skipped from the point of the error.

IEP559I E OUTPUT OPTION FOR FILE CONFLICTS WITH REVERSED.

Explanation: The output option conflicts with an opening of a file, reversed.

System Action: The statement is skipped from the point of the error.

IEP560I C SYNTAX REQUIRES WORD NAMED, CHANGED, OR CHANGED NAMED. FOUND XXX.

System Action: The statement is skipped from the point of error.

IEP561I C DATA TYPE MUST BE ED, ID, BI, EF, IF, RP, AL, AN, OR GF. FOUND XXX.

Explanation: A filename, condition name, figcon, or variable-length group is not valid at this point.

System Action: The statement is skipped from the point of the error.

IEP562I C DATA ENTRY MUST NOT EXCEED 120 CHARACTERS.

Explanation: The data entry specified exceeds the maximum permitted for this type of output.

System Action: The statement is skipped from the point of the error.

IEP563I C DATA ENTRY MUST BE DISPLAY.

System Action: The statement is skipped from the point of the error.

IEP564I C SYNTAX REQUIRES ONE OF THE ALLOWABLE CHARACTERS. FOUND XXX.

System Action: The statement is skipped from the point of the error.

IEP565I C IF STATEMENT MUST BE TERMINATED BY A PERIOD.

Explanation: This diagnostic is obtained when the IF statement is the last statement of a paragraph and a label is detected instead of a period.

System Action: The statement is skipped from the point of error.

IEP566I C DATA TYPE MUST BE AL, AN, RP, OR GROUP.

System Action: The statement is skipped from the point of error.

IEP567I C DATA TYPE MUST BE AL, AN, FIGCON OR FIXED-LENGTH GROUP.

System Action: The statement is skipped from the point of the error.

IEP568I C DATA ITEM MUST NOT EXCEED 256 CHARACTERS.

System Action: The statement is skipped from the point of the error.

IEP569I C DATA ENTRIES MUST BE OF EQUAL LENGTH.

System Action: The statement is skipped from the point of the error.

IEP570I C THE LENGTH OF THE SECOND OPERAND MUST BE EQUAL TO THE FIRST OR A SINGLE CHARACTER

System Action: The statement is skipped from the point of the error.

IEP571I E A RECORD NAME MUST BE ASSOCIATED WITH THIS FILE. FOUND XXX.

System Action: The statement is skipped from the point of the error.

IEP572I C ONLY ONE DATA-NAME MAY BE ASSOCIATED WITH THE CHANGED OPTION.

System Action: The statement is skipped from the point of the error.

IEP573I C DATA TYPE MUST BE ED, ID, BI, EF, IF, SN, SR, RP, AL, AN, FC, OR GROUP.

System Action: The statement is skipped from the point of error.

IEP601I W NO SIGNIFICANT POSITION MATCHES BETWEEN SENDING AND RECEIVING FIELDS IN MOVE. RECEIVING FIELD IS SET TO ZERO.

Explanation: There are no digit positions in common between the sending and receiving fields. This can be illustrated by moving a field with picture 99 to a receiving field with picture V99.

System Action: The receiving field is set to zero.

IEP602I W DESTINATION FIELD DOES NOT ACCEPT THE WHOLE SENDING FIELD IN MOVE.

Explanation: The sending field is larger than the receiving field in either its integer or decimal positions or both.

System Action: The sending field is truncated.

IEP603I C AFTER ADVANCING OPTION NOT ALLOWED WITH REWRITE.

System Action: The statement is skipped from the point of the error.

IEP604I E SOURCE PROGRAM EXCEEDS INTERNAL LIMITS.

Explanation: The program is too large.

User Response: The user should do one of the following, and try again:

- Divide the program into two or more parts
- Simplify compound conditional statements.

IEP605I E PROCEDURE NAME MULTIPLY DEFINED.

Explanation: Procedure name indicated was multiply defined and was not qualified properly by the appropriate section name when used.

IEP606I E PROCEDURE-NAME XXX NOT DEFINED

Explanation: The name indicated was incorporated into a GO TO or a PERFORM statement, and was never defined. Procedure names must begin in columns 8 through 11 at the point where they are defined.

IEP607I E INVALID LITERAL XXX.

User Response: Check for multiple decimal points, non-numeric characters not enclosed in quotes.

IEP608I E XXX IS NOT ALLOWED TO HANDLE MORE THAN 25 FILES IN ONE STATEMENT.

System Action: The rest of the statement is skipped. Only 25 files are handled.

IEP609I E PROCEDURE-NAME XXX HAS ILLEGAL CONTENT AND IS DROPPED.

IEP610I E 'CONDITION NAME' WAS EITHER NOT ALLOWED IN THIS STATEMENT OR SKIPPED DUE TO ANOTHER DIAGNOSTIC

IEP611I E TOO MANY PARAGRAPH NAMES HAVE BEEN USED IN CALL STATEMENTS.

IEP612I E OPEN STATEMENT CONTAINS MORE THAN 9 FILENAMES. OPEN WILL SPLIT.

System Action: Handles multiple OPEN statements each containing 9 filenames.

IEP613I W USING STATEMENT HAS BEEN INCORRECTLY SPECIFIED.

IEP614I E THIS CONDITIONAL HAS A MISSING RELATIONAL OPERATOR.

System Action: The statement is skipped from the point of the error.

IEP615I E READ 'AT END' REQUIRED FOR FILES WITH ACCESS SEQUENTIAL

System Action: The entire statement is skipped.

IEP616I E 'INVALID KEY' REQUIRED FOR FILES WITH ACCESS RANDOM

System Action: The entire statement is dropped.

IEP617I E WRITE 'FROM' OPTION REQUIRED WITH APPLY WRITE-ONLY

System Action: The entire statement is dropped.

IEP618I E REWRITE INVALID ON DIRECT OR RELATIVE SEQUENTIAL FILES

System Action: The entire statement is dropped.

IEP619I E WRITE INVALID FOR RELATIVE RANDOM FILE

System Action: The entire statement is dropped.

IEP620I E WRITE 'INVALID KEY' REQUIRED FOR INDEXED SEQUENTIAL FILE

System Action: The entire statement is dropped.

IEP621I E OPEN 'I-O' INVALID FOR DIRECT OR RELATIVE SEQUENTIAL FILES

Explanation: On OPEN and CLOSE no code is generated for the file in error.

System Action: Syntax scan skips to the next file in the statement.

IEP622I C OPEN 'OUTPUT' INVALID FOR FILES WITH ACCESS RANDOM, I-O ASSUMED.

Explanation: See message IEP621I.

IEP623I E OPEN 'REVERSED' VALID ONLY ON STANDARD SEQUENTIAL FILES

Explanation: See message IEP621I.

IEP625I E OPEN 'REVERSED' INVALID FOR FILES WITH FORMAT V RECORDS

IEP626I E CLOSE 'UNIT' OR 'REEL' VALID ONLY FOR STANDARD SEQUENTIAL FILES

Explanation: See message IEP621I.

IEP627I E 'INVALID KEY' INVALID FOR STANDARD, DIRECT OR RELATIVE SEQUENTIAL FILES.

System Action: The clause is skipped.

IEP628I E 'ACTUAL KEY' REQUIRED FOR DIRECT SEQUENTIAL OUTPUT FILES

IEP700I E IDENTIFICATION DIVISION NOT FOUND

IEP701I E DATA DIVISION NOT FOUND. COMPILATION CANCELED.

IEP702I E PROCEDURE DIVISION NOT FOUND. COMPILATION CANCELED.

IEP703I E SOURCE PROGRAM EXCEEDS INTERNAL LIMITS. COMPILATION CANCELED.

IEP704I E DATA-NAME TABLE OVERFLOW. COMPILATION CANCELED.

Explanation: The data-name attribute table has a maximum size of 64K bytes.

User Response: Reduce the length of data-names, and recompile.

IEP705I NO DIAGNOSTICS IN THIS COMPILATION.

IEP709I W INCORRECT EXECUTE PARAMETER - XXX.

LOAD MODULE EXECUTION DIAGNOSTIC MESSAGES

Load module execution diagnostic messages are of two types: object time messages, and operator messages.

OBJECT TIME MESSAGES

Most object time messages are self explanatory. Where necessary, examples are included to explain the message.

IEP999I MINUS BASE MADE POSITIVE & FLOATING POINT EXPONENTIATION CONTINUED.

IEP998I ZERO BASE TO POSITIVE EXPONENT - FLOATING-POINT ANSWER MADE ZERO.

IEP997I ZERO BASE TO MINUS EXPONENT - FLOATING-POINT ANSWER IS MAX F.P. NUMBER.

IEP996I RESULT TOO BIG - FLOATING-POINT EXPONENTIATION ANSWER IS MAX F.P. NUMBER.

IEP993I ZERO BASE TO MINUS EXPONENT - PACKED EXPONENTIATION RESULT MADE ALL NINES.

OPERATOR MESSAGES

In addition to system diagnostic and object time messages the COBOL load module may issue operator messages.

The following message is generated by STOP 'literal'.

IEP000D text provided by object program.

Explanation: This message is issued at the programmer's discretion to indicate possible alternative action to be taken by the operator.

Operator Response: Follow the instructions given both by the message and on the job request form supplied by the programmer. If the job is to be resumed, issue a REPLY command with a text field that contains any 1-character message.

The following message is generated by an ACCEPT ... FROM CONSOLE.

IEP990D 'AWAITING REPLY'

Explanation: This message is issued by the object program when operator intervention is required.

Operator Response: Issue a REPLY command. (The contents of the text field should be supplied by the programmer on the job request form.)

DEBUG PACKET ERROR MESSAGES

The following is a complete list of precompile error messages. They apply to errors in the debugging packets only.

IEP850I TABLE OF DEBUG REQUESTS OVERFLOWED. RUN TERMINATED.

IEP851I THE FOLLOWING CARD DUPLICATES A PREVIOUS *DEBUG CARD. THIS PACKET WILL BE IGNORED.

IEP852I THE FOLLOWING PROCEDURE DIVISION NAMES WERE NOT FOUND. INCOMPLETE DEBEG EDIT IS NOT TERMINATED.

IEP853I THE FOLLOWING *DEBUG CARD DOES NOT CONTAIN A VALID LOCATION FIELD. THIS PACKET WILL BE IGNORED.

IEP854I IDENTIFICATION DIVISION NOT FOUND. RUN TERMINATED.

IEP855I DEBUG EDIT RUN COMPLETE. INPUT FOR COBOL COMPILATION ON SYSUT4.

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]

READER'S COMMENT FORM

IBM System/360
Operating System
COBOL (E) Programmer's Guide C24-5029-2

- Your comments, accompanied by answers to the following questions, help us produce better publications for your use. If your answer to a question is "No" or requires qualification, please explain in the space provided below. All comments will be handled on a non-confidential basis. Copies of this and other IBM publications can be obtained through IBM Branch Offices.

	Yes	No
• Does this publication meet your needs?	☐	☐
• Did you find the material:		
Easy to read and understand?	☐	☐
Organized for convenient use?	☐	☐
Complete?	☐	☐
Well illustrated?	☐	☐
Written for your technical level?	☐	☐

- What is your occupation? _____
- How do you use this publication?

As an introduction to the subject?	☐	As an instructor in a class?	☐
For advanced knowledge of the subject?	☐	As a student in a class?	☐
For information about operating procedures?	☐	As a reference manual?	☐

Other _____

- Please give specific page and line references with your comments when appropriate.

COMMENTS:

- Thank you for your cooperation. No postage necessary if mailed in the U. S. A.

Staple

C24-5029-2

Fold

Fold

FIRST CLASS
PERMIT NO. 33504
NEW YORK, N.Y.

BUSINESS REPLY MAIL

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .

IBM CORPORATION

1271 AVENUE OF THE AMERICAS
NEW YORK, N.Y. 10020

ATTENTION: PUBLICATIONS, DEPT. D39

Fold

Fold

IBM

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]

Additional Comments:

IBM **Technical Newsletter**

File Number S360-24

Re: Form No. C24-5029-2

This Newsletter No. N28-0229

Date November 15, 1967

Previous Newsletter Nos. None

IBM System/360 Operating System

COBOL (E) Programmer's Guide

This Technical Newsletter amends the publication IBM System/360 Operating System: COBOL (E) Programmer's Guide, Form C24-5029-2.

In the referenced publication, replace the pages listed below with the corresponding pages attached to this newsletter.

Pages	Subject of Amendment
1- 4	Front matter
5- 6	Clarification of the term "job" rather than "job step" with reference to data set and group generation
27- 28	Specification of Write Verify in COBOL work files
29- 30	Amends buffer size relating to the system utility device
39- 40	Amends table showing format for UNIT
43- 44, 47- 50	Amends the DCB subparameter "OPTCD"
51- 52	Amends example of DD statements for Indexed Sequential organization
103-104	Amends example of statement in return routine
145-146	Adds error messages

A vertical line to the left of the column shows where text has been changed; changes to illustrations are shown by a bullet (•) to the left of the caption.

The specifications contained in this publication correspond to Release #14 of the IBM System/360 Operating System.

Please file this page at the back of the publication. It will provide a reference to changes, a method of determing that all amendments have been received, and a check that the publication contains the proper pages.

IBM Corporation, Programming Systems Publications, 1271 Avenue of the Americas, New York, N.Y. 10020

PRINTED IN U.S.A.

File Number S360-24
Form C24-5029-2

OS

IBM Systems Reference Library

IBM System/360
Operating System
COBOL (E) Programmer's Guide

This reference publication describes how to compile, linkage edit, and execute a COBOL (E-Level Subset) program. It also describes the output of compilation and execution, how to make optimal use of the compiler and a load module, and compiler and load module restrictions.

The corequisite to this publication is IBM System/360 Operating System: COBOL Language, Form C28-6516.

Other publications related to this one are:
- IBM System/360 Principles of Operation, Form A22-6821.
- IBM System/360 Operating System: Control Program Services, Form C28-6541.
- IBM System/360 Operating System: Job Control Language, Form C28-6539.
- IBM System/360 Operating System: Utilities, Form C28-6586.
- IBM System/360 Operating System: Linkage Editor, Form C28-6538.
- IBM System/360 Operating System: Control Program Messages and Completion Codes, Form C28-6608.

For a list of other associated System/360 publications, see the IBM System/360 Bibliography, Form A22-6822.

Form C24-5029-2
Page Revised 11/15/67 by TNL N28-0229

PREFACE

The purpose of the Programmer's Guide is to enable programmers to compile, linkage edit, and execute COBOL (E-Level Subset) programs under control of IBM System/360 Operating System. The COBOL (E-Level Subset) language is described in the publication IBM System/360 Operating System: COBOL Language, Form C28-6516, which is a corequisite to this publication.

The Programmer's Guide is organized to fulfill its purpose at three levels:

1. Programmers who wish to use the cataloged procedures as provided by IBM need read only the Introduction and Job-Control Language sections to understand the job-control statements, and the Job Processing section to use cataloged procedures for compiling, linkage editing, and executing COBOL programs. The Programming Considerations and System Output sections are recommended for programmers who want to use the COBOL language more effectively.
2. Programmers who are also concerned with creating and retrieving data sets, optimizing the use of I/O devices, or temporarily modifying IBM-supplied cataloged procedures should read the entire Programmer's Guide.
3. Programmers concerned with making extensive use of the operating system facilities, such as writing their own cataloged procedures, should also read the entire Programmer's Guide in conjunction with the publications listed on the front cover of this publication.

In addition to providing reference information on compiling, linkage editing, and executing programs, this publication contains appendices that:

1. Give several examples of processing.
2. Contain detailed descriptions of the diagnostic messages produced during compilation and load module execution.

Third Edition

This edition, Form C24-5029-2, is a major revision of, and makes obsolete, Form C24-5029-1. Changes to this publication are indicated by a vertical line to the left of the text that is affected. Changes to illustrations are indicated by a bullet (•) at the left of the caption.

Significant changes and additions to the specifications contained in this publication will be reported in subsequent revisions or Technical Newsletters.

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

Comments may be addressed to IBM Corporation, Programming Publications, 1271 Avenue of the Americas, New York, N.Y. 10020

Form C24-5029-2
Page Revised 11/15/67 by TNL N28-0229

CONTENTS

The IBM System/360 Operating System (referred to here as the operating system) consists of a control program and processing programs. The control program supervises execution of all processing programs, such as the COBOL-E compiler, and all problem programs, such as a COBOL problem program. Therefore, to execute a COBOL program, the programmer must first communicate with the operating system. The medium of communication between the programmer and the operating system is the job-control language.

Job-control language statements define units of work to the operating system. Two units of work are recognized: the job and the job step. The statements that define these units of work are the JOB and the EXEC (execute) statements. Another important statement is the DD (data definition) statement, which gives the operating system information about data used in jobs and job steps. The flow of control statements and any data placed in the flow of control statements is called the input stream.

Note: Throughout this publication certain arbitrary options are given in illustrative examples. Some of the options used are a function of system generation; therefore, these examples may not be valid for all systems.

JOB AND JOB STEP RELATIONSHIP

When a programmer is given a problem, he analyzes that problem and defines a precise problem-solving procedure; that is, he writes a program or a series of programs. Executing a main program (and its subprograms) is a job step to the operating system. A job consists of executing one or more job steps.

At its simplest, a job consists of one job step. For example, executing a payroll program is a job step.

In another sense, a job consists of several interdependent job steps, such as a compilation, linkage edit, and execution. Job steps can be related to each other as follows.

1. One job step may pass intermediate results recorded on an external storage volume to a later job step.

2. Whether or not a job step is executed may depend on results of preceding steps.

In the series of job steps (compilation, linkage edit, and execution), each step can be a separate job with one job step in each job. However, designating several related job steps as one job is more efficient: processing time is decreased because only one job is defined, and interdependence of job steps may be stated. (Interdependence of jobs cannot be stated.) Each step may be defined as a job step within one job that encompasses all processing.

```
JOB:      Compile, linkage edit, and execute
  JOB STEP 1:  Compile COBOL program
  JOB STEP 2:  Linkage edit compiled
               program
  JOB STEP 3:  Execute linkage edited
               program
```

Figure 1 illustrates these three job steps.

The important aspect of jobs and job steps is that they are defined by the programmer. He defines a job to the operating system by using a JOB statement; he defines a job step by the EXEC statement.

DATA SETS

In Figure 1, one collection of input data (source program) and one collection of output data (compiled program) are used in job step 1. In the operating system, a collection of data that can be named by the programmer is called a data set. A data set is defined to the operating system by a DD statement.

A data set resides on a volume(s), which is a unit of external storage that is accessible to an input/output device. (For example, a volume may be a reel of tape or a disk pack.)

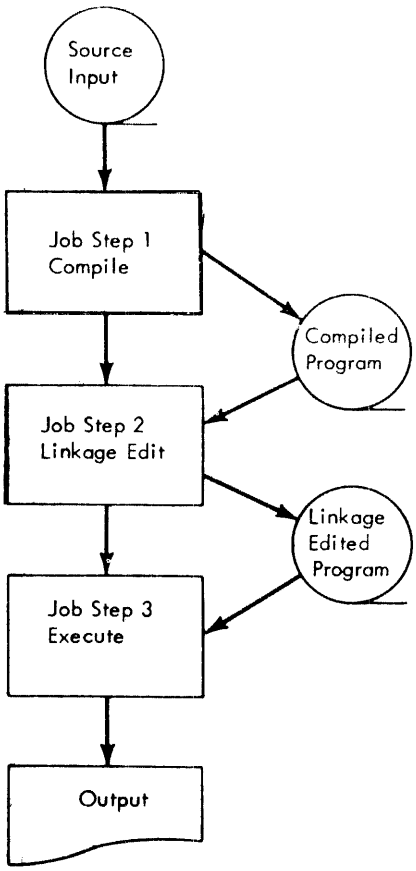


Figure 1. Job Example with Three Job Steps

Several I/O devices grouped together and given a single name when the system is generated constitute a device class. For example, a device class can consist of all the tape devices in the installation, another can consist of the printer, a direct-access device, and a tape device.

The name of a data set and information identifying the volume(s) on which the data set resides may be placed in an index to help the control program find the data set. This index, which is part of an index structure called the catalog, resides on a direct-access volume. Any data set whose name and volume identification are placed in this index is called a cataloged data set. When a data set is cataloged, the information needed to access the data set is its name and disposition. Other information associated with the data set, such as device type, position of the data set on the volume, and format of records in the data set, is available to the control program.

Furthermore, a hierarchy of indexes may be devised to classify data sets and make names for data sets unique. For example, an installation may divide its cataloged data sets into four groups: SCIENCE, ENGRNG, ACCNTS, and INVNTRY. In turn, each of these groups may be subdivided. For example, the ACCNTS group may be subdivided into RECEIVE and PAYABLE; PAYABLE may contain volume identification for the data sets PAYROLL and OVERHEAD. To find the data set PAYROLL, the programmer specifies all indexes beginning with the largest group, ACCNTS; then the next largest group, PAYABLE; finally, the data set PAYROLL. The complete identification needed to find that data set PAYROLL is ACCNTS.PAYABLE.PAYROLL.

Data set names are of two classes: unqualified and qualified. An unqualified name is a data set name or an index name that is not preceded by an index name. A qualified name is a data set name or index name preceded by index names representing index levels; for example, in the preceding text, the qualified name of the data set PAYROLL is ACCNTS.PAYABLE.PAYROLL.

Data set identification may also be based upon the time of generation. In the operating system, a collection of successive, historically related data sets is a generation data group. Each of the data sets is a generation data set. A generation number is attached to the data group name to refer to a particular generation. The most recent generation is 0; the generation previous to 0 is -1; the generation previous to -1 is -2; etc. An index describing a generation data group must exist in the catalog.

For example, a data group named YTDPAY might be used for a payroll application. The generations for the generation data group YTDPAY are;

```
YTDPAY (0)
YTDPAY (-1)
YTDPAY (-2)
   .
   .
   .
```

When a new generation is being created, it is called generation (+n), where n is an integer with value greater than 0. For example, after the data set YTDPAY(+1) has been created, at the end of the job the operating system changes the data set name to YTDPAY(0). The data set that was YTDPAY(0) at the beginning of the job becomes YTDPAY(-1) at the end of the job, and so on.

Compiler Name

The program name for the compiler is IEPCBL00. If the compiler is to be executed without using the supplied cataloged procedures in a job step, the EXEC statement parameter

```
PGM=IEPCBL00
```

must be used.

Compiler ddnames

The compiler can use up to eight data sets. To establish communication between the compiler and the programmer, each data set is assigned a specific ddname. Each data set has a specific function and device requirement. Table 2 lists the ddnames, functions, and device requirements for the data sets.

To compile a COBOL source module, five of these data sets are necessary: SYSIN, SYSPRINT, SYSUT1, SYSUT2, and SYSUT3, along with the direct-access volume(s) that contains the operating system. With these five data sets, the compiler generates a listing only. If an object module is to be punched or written on a direct-access or magnetic tape volume, a SYSPUNCH DD statement must be supplied. If the debug packet(s) is to be used, a SYSUT4 DD statement must be supplied. If the compiler is to COPY or INCLUDE a source-language module from the user's source program library, a SYSLIB DD statement must be supplied.

Table 2. Compiler ddnames

ddname	FUNCTION	DEVICE REQUIREMENTS
SYSIN	reading the source program	• card reader • intermediate storage
SYSPRINT	writing the storage map, listings, and messages	• printer • intermediate storage
SYSPUNCH	punching the object module deck, or creating an object module data set as input to the linkage editor	• card punch • direct-access • magnetic tape
SYSUT1	work data set needed by the compiler during compilation	• direct-access • magnetic tape
SYSUT2	work data set needed by the compiler during compilation	• direct-access • magnetic tape
SYSUT3	work data set needed by the compiler during compilation	• direct-access • magnetic tape
SYSUT4	optional work data set needed when using debug packet(s)	• direct-access • magnetic tape
SYSLIB	optional user source program library	• direct-access

For the DD statement SYSIN or SYSPRINT, an intermediate storage device may be specified instead of the card reader or

printer. The intermediate storage device usually is magnetic tape, but can be a direct-access device.

If an intermediate device is specified for SYSIN, the compiler assumes that the source module deck was placed on intermediate storage by a previous job or job step. If an intermediate device is specified for SYSPRINT, the maps, listing, and error/warning messages are written on that device; a new job or job step can print the contents of the data set.

Compiler Device Classes

Names for input/output device classes used for compilation are also specified by the operating system when the system is generated. The class names, functions, and types of devices are shown in Table 3.

Table 3. Device Class Names

CLASS NAME	CLASS FUNCTIONS	DEVICE TYPE
SYSSQ	writing, reading	• direct-access • magnetic tape
SYSDA	writing, reading	• direct-access
SYSCP	punching cards	• card punch
A	SYSOUT output	• printer • magnetic tape

```
{PARM          }=([LINECNT=nn][,BUFSIZE=nn]
{PARM.procstep }
                 [,DECK  ] [,FLAGE] [,LIST ]
                 [,NODECK] [,FLAGW] [,NOLIST]
                 [,DMAP  ] [,PMAP ] [,MAPS ]
                 [,NODMAP] [,NOPMAP] [,NOMAPS]
                 [,DISPCK] [,REGED]
                 [,NODISPCK] [,INVED] )
```

Figure 22. Compiler Options

The data sets used by the compiler must be assigned to the device classes listed in Table 4.

Table 4. Correspondence Between Compiler ddnames and Device Classes

ddname	POSSIBLE DEVICE CLASSES
SYSIN	SYSSQ, or the input stream device (specified by DD * or DD DATA)
SYSPRINT	A,SYSSQ,SYSDA
SYSPUNCH	SYSCP,SYSSQ,SYSDA
SYSUT1	SYSSQ,SYSDA
SYSUT2	SYSSQ,SYSDA
SYSUT3	SYSSQ,SYSDA
SYSUT4	SYSSQ,SYSDA
SYSLIB	SYSDA

Compiler Options

Options may be passed to the compiler through the PARM parameter in the EXEC (Figure 22). The following information may be specified:

1. The number of lines to be printed per page on the compiler output listing.
2. The size of each of the six work buffers used during a compilation. For workfiles in COBOL, a write validity check is not specified. This can be overridden by specifying OPTCD=W on the SYSUT DD cards.
3. Whether an object module is created.
4. The type of diagnostic messages to be generated by the compiler.
5. Whether a list of the source statements is printed.
6. Whether a list of data-name addresses is generated.

7. Whether a list of object code is generated.

8. Whether a list of both data-name addresses and object code is generated.

9. Whether the compiler will generate object code to test length of fields to be DISPLAYed.

10. The type of editing performed in the PICTURE clause and numeric literals.

There is no specified order for compiler options in the PARM parameter.

LINECNT=nn: The LINECNT option indicates the number of lines to be printed on each page of the compilation output listing. The programmer specifies a number nn, where nn is a 2-digit integer in the range of 10 to 99. If the option is not specified, the number of lines per page will be that specified when the system was generated.

BUFSIZE=nn: The BUFSIZE option indicates the size of each of the six work buffers used during a COBOL compilation. The BUFSIZE parameter should not be used on a 32K system. The following formula can be used to determine the maximum value to be used for this parameter.

$$S = \frac{C - 30000 - [(13 + L)(N)]}{6}$$

where: S is the size of each work buffer
C is the total main storage
L is the length of the average data name
N is the number of data names.

The maximum value of S can never exceed the block size of a particular utility file as specified for the device. For example, if the work buffers are for tape, the maximum value of S is 32670. If the work buffers are for disk, the maximum value of S depends upon the type of direct-access device assigned to the system utility device:

Device	Maximum Value of S
2301	20483
2302	4984
2303	4892
2311	3625
2314	7294
2321	2000

Note: If the assignments to the system utility devices are mixed (i.e., SYSUT1 assigned to 2311, SYSUT2 assigned to 2301, SYSUT3 assigned to 2314, etc.), the maximum value of S cannot exceed the smallest value corresponding to the assigned devices. For example, if SYSUT1 is assigned to a 2311 and SYSUT2 is assigned to a 2314, S cannot exceed 3625, the smaller of 3625 and 7294.

DECK or NODECK: The DECK option specifies that the compiled source module (i.e., the object module) is written on the data set specified by the SYSPUNCH DD statement. NODECK specifies that no object module is written. A description of the deck is given in the section, System Output. If neither option is specified, an object module is produced.

FLAGE or FLAGW: The FLAGE option specifies that the compiler will suppress warning diagnostic messages. The FLAGW option specifies that the compiler will generate diagnostic messages for actual errors in the source module, plus warning diagnostic messages for possible errors. Diagnostic messages are written on the data set specified by the SYSPRINT DD statement. If neither option is specified, the class of diagnostic message produced is that specified when the system was generated.

LIST or NOLIST: The LIST option specifies that the source listing is written on the data set specified by the SYSPRINT DD statement. The NOLIST option indicates that no source listing is written. A description of the source listing is given in the section, System Output. If neither option is specified, a source listing is produced.

DMAP or NODMAP: The DMAP option specifies that the compiler will generate a listing of the DATA DIVISION data-names and their addresses relative to the load point of the object module. The listing is written on the data set specified by the SYSPRINT DD statement. The NODMAP option specifies that a data-name listing will not be generated. If neither DMAP nor NODMAP is specified, the option taken will be that specified when the system was generated.

PMAP or NOPMAP: The PMAP option specifies that the compiler will generate a listing of object code for each statement in the PROCEDURE DIVISION. The listing is written on the data set specified by the SYSPRINT DD statement. The NOPMAP option specifies that a listing of object code will not be generated. If neither PMAP nor NOPMAP is specified, the option taken will be that specified when the system was generated.

MAPS or NOMAPS: The MAPS option is equivalent to specifying both DMAP and PMAP. The NOMAPS option is equivalent to specifying both NODMAP and NOPMAP.

DISPCK or NODISPCK: The DISPCK option specifies that the compiler will generate object code that will test, at execution

time, to determine if a field to be DISPLAYed exceeds the record length of the device on which it is to be written. The NODISPCK option specifies that no such code will be generated. If neither DISPCK nor NODISPCK is specified, the option taken will be that specified when the system was generated.

REGED or INVED: The REGED option specifies that the character "." represents a decimal point and the character "," represents an insertion character. The INVED option specifies that the above roles of these characters ".", "," be reversed.

LINKAGE EDITOR PROCESSING

The linkage editor processes COBOL object modules, COBOL subroutines, resolves any references to subprograms, and constructs a load module. To communicate with the linkage editor, the programmer supplies an EXEC statement and DD statements that define all required data sets; he may also supply linkage editor control statements.

LINKAGE EDITOR NAME

The program name for the linkage editor is IEWL. If the linkage editor is executed without using cataloged procedures in a job step, the EXEC statement parameter

```
PGM=IEWL
```

must be used.

LINKAGE EDITOR INPUT AND OUTPUT

There are two types of input to the linkage editor: primary and secondary.

Primary input is a sequential data set that contains object modules and linkage editor control statements. Any external references among object modules in the primary input are resolved by the linkage editor as the primary input is processed. Furthermore, the primary input contains references to the secondary input. These references are linkage editor control statements and/or COBOL external references in the object modules.

Secondary input resolves references and is separated into two types: automatic call library and additional input specified by the programmer. The automatic call library must always be the COBOL library (SYS1.COBLIB), which is the PDS that contains the COBOL object time subroutines. Through the use of DD statements, the automatic call library can be concatenated with other partitioned data sets. Three types of additional input may be specified by the programmer:

1. An object module used as the main program in the load module being constructed. This object module, which can be accompanied by linkage editor control statements, is either a member of a PDS or is a sequential data set. The first record in the primary input data set must be a linkage editor INCLUDE control statement that tells the linkage editor to process the main program.

2. An object module used to resolve external references made in another module. The object module, which can be accompanied by linkage editor control statements, is a sequential data set. An INCLUDE statement that defines the data set must be given.

3. A module used to resolve external references made in another module. The load module which can be accompanied by linkage editor control statements, is a member of a PDS. The module can be included from the call library.

In addition, the secondary input can contain external references and linkage editor control statements. If a load module is not in the automatic call library, the linkage editor LIBRARY statement can be used to direct the linkage editor to reference additional libraries during the automatic library call process.

The output load module of the linkage editor is always placed in a PDS as a named member. The name can be provided in the SYSLMOD DD statement for the linkage editor execution. For the execution of the load module, this name can be used. Error messages and optional diagnostic messages are written on an intermediate storage device or a printer. Also, a work data set on a direct-access device is required by the linkage editor to do its processing. Figure 23 shows the I/O flow in linkage editor processing.

LINKAGE EDITOR DDNAMES AND DEVICE CLASSES

The programmer communicates data set information to the linkage editor through DD statements identified by specific ddnames (similar to the ddnames used by the compiler). The ddnames, functions, and requirements for data sets are shown in Table 5.

Any data sets specified by SYSLIB or SYSLMOD must be partitioned data sets. (Additional inputs are partitioned data sets or sequential data sets.) The ddname for the DD statement that defines any additional libraries or sequential data sets is written in INCLUDE and LIBRARY statements and is not fixed by the linkage editor.

[1] If only "name" is specified, the delimiting parentheses may be omitted.

[2] If only one "volume-serial-number" is specified, the delimiting parentheses may be omitted.

[3] SER and REF are keyword subparameters; the remaining subparameters are positional subparameters.

[4] All subparameters are positional subparameters.

[5] EXPDT and RETPD are keyword subparameters; the remaining subparameters are positional subparameters.

[6] All subparameters are keyword subparameters.

Figure 25. DD Parameters for Creating Data Sets (Part 2 of 2)

```
   Example 1:  Creating a Cataloged Data Set

//CALC DD DSNAME=PROCESS,DISP=(NEW,CATLG),LABEL=(,SL,EXPDT=66031),    1
//             UNIT=DACLASS,VOLUME=(PRIVATE,RETAIN,SER=AA69),         2
//             SPACE=(300(100,100),,CONTIG,ROUND)

   Example 2:  Creating a Data Set for a Job

//SYSUT1 DD DSNAME=&TEMP,UNIT=(TAPECLS,3),DISP=(NEW,PASS),            1
//             VOLUME=(,RETAIN,1,9,SER=(777,888,999)),                2
//             DCB=(DEN=2)

   Example 3:  Specifying a SYSOUT Date Set

//SYSPRINT DD SYSOUT=A

   Example 4:  Creating a Data Set that Is Kept, but Not Cataloged

//TEMPFILE DD DSNAME=FILE,DISP=(,KEEP),                               1
//             DCB=(DEN=2)

   Example 5:  Creating a Data Set on a 7-Track Tape

//TEMPFILE DD DSNAME=FILE,DISP=(OLD,KEEP),                            1
//             VOLUME=(PRIVATE,,,SER=222,333),                        2
//             DCB=(DEN=1,TRTCH=ET),UNIT=(2400-2)
```

Figure 26. Examples of DD Statements

DDNAME=ddname

indicates a DUMMY data set that will assume the characteristics specified in a following DD statement "ddname". The DD statement identified by "ddname" then loses its identity; that is, it cannot be referred to by an *....ddname parameter. The statement in which the DDNAME parameter appears may be referenced by subsequent *....ddname parameters. If a subsequent statement identified by "ddname" does not appear, the data set defined by the DD statement containing the DDNAME parameter is assumed to be an unused statement. The DDNAME parameter can be used five times in any given job step or procedure step, and no two uses can refer to the same "ddname". The DDNAME parameter is used mainly for cataloged procedures.

SPECIFYING INPUT/OUTPUT DEVICES

The name of an input/output device or class of devices and the number of devices are specified in the UNIT parameter.

```
UNIT=(name[,n])
```

name

is the name assigned to the input/output device classes when the system is generated, or an absolute device address.

[,n]

specifies the number-of devices allocated to the data set. If this parameter is omitted, 1 is assumed.

SPECIFYING VOLUMES

The programmer indicates the volumes used for the data set in the VOLUME parameter.

```
VOLUME=([PRIVATE][,RETAIN]
        [,volume-sequence-number]
        [,volume-count]
      [,SER=(volume-serial-number
          [,volume-serial-number]...)
       ,REF={dsname                        }]  )
            {*.ddname                      }
            {*.stepname.ddname             }
            {*.stepname.procstep.ddname    }
```

identifies the volume(s) assigned to the data set.

PRIVATE

is used only for direct-access volumes. This option indicates that the assigned volume is to contain only the data set defined by this DD statement. PRIVATE is overridden when the DD statement for a data set requests the use of the private volume with the SER or REF subparameter. Volumes other than direct-access volumes are always considered PRIVATE.

RETAIN

indicates that this volume is to remain mounted after the job step is completed. Volumes are retained so that data may be transmitted to or from the data set, or so that other data sets may reside in the volume. If the data set requires more than one volume, only the last volume is retained; the other volumes are previously dismounted. Another job step indicates when to dismount the volume by omitting RETAIN. If each job step issues a RETAIN for the volume, the retained status lapses when execution of the job is completed.

volume-sequence-number

is a one-to-four digit number that specifies the sequence number of the first volume of the data set that is read or written. The volume sequence number is meaningful only if the data set is cataloged and earlier volumes omitted.

volume-count

specifies the number of volumes required by the data set. Unless the SER or REF subparameter is used this subparameter is required for every multi-volume output data set.

SER

specifies one or more serial numbers for the volumes required by the data sets. A volume serial number consists of one to six alphameric characters. If it contains less than six characters, the serial number is left adjusted and padded with blanks. If SER is not specified, and DISP is not specified as NEW, the data set is assumed to be cataloged and serial numbers are retrieved from the catalog. A volume serial number is not required for output data sets.

REF

indicates that the data set is to occupy the same volume(s) as the data set identified by "dsname", "*.ddname", "*.stepname.ddname", or *.stepname.procstep.ddname. Table 7 shows the data set references.

allocated to any of the data sets in the group has been exhausted and more data is to be written. This quantity will not be split.

LABEL INFORMATION

If the programmer wishes to catalog a data set so that he can refer to it without repeating information that was supplied when the data set was created, he must specify certain information in the LABEL parameter. If the parameter is omitted and the data set is cataloged or passed, the label information is retrieved from data set labels stored with the data set.

```
LABEL=([data-set-sequence-number]{,NL}
                                 {,SL}

      [,EXPDT=yyddd]
      [,RETPD=xxxx ] )
```

data-set-sequence-number

is a 4-digit number that identifies the relative location of the data set with respect to the first data set on a tape volume. (For example, if there are three data sets on a magnetic tape volume, the third data set is identified by data set sequence number 3.) If the data set sequence number is not specified, the operating system assumes 1. (This option should not be confused with the volume sequence number, which represents a particular volume for a data set.)

NL
SL

specifies whether standard labels exist for a data set. SL indicates standard labels. NL indicates no labels.

EXPDT=yyddd
RETPD=xxxx

specifies how long the data set shall exist. The expiration date, EXPDT=yyddd, indicates the year (yy) and the day (ddd) the data set can be deleted. The period of retention, RETPD=xxxx, indicates the period of time, in days, that the data set is to be retained. If neither is specified, the retention period is assumed to be zero.

DISPOSITION OF A DATA SET

The disposition of a data set is specified by the DISP parameter; see Data Definition (DD) Statement. The same options are used for both creating data sets and using previously created data sets. When a data set is created, the subparameters used are NEW, KEEP, PASS, and CATLG.

WRITING A UNIT RECORD DATA SET ON THE PRINTER

A printed output data set may be written using the following parameter.

```
SYSOUT=A
```

DCB PARAMETER

For load module execution, the COBOL programmer may specify the details of a data set by using COBOL source statements and DD statement subparameters of the DCB parameter. The illustrations given in the following are examples of DCB subparameters for processing these file organizations:

- Sequential
- Indexed Sequential
- Direct or Relative

Sequentially organized data sets may reside on magnetic tape or direct-access volumes. Direct relative or indexed files must reside on direct-access volumes. Note that some DCB subparameter values (see Tables 10, 11, and 12) may be supplied by DD statements; other values are supplied either by certain COBOL source statements or by the COBOL compiler.

DCB FOR PROCESSING SEQUENTIAL DATA SET

```
DCB= ([DEN={0|1|2}]
  [,TRTCH={C|E|T|ET|U|UC}]
  [,PRTSP={0|1|2|3}]
  [,MODE={C|E}][,STACK={1|2}]
  [,OPTCD={W|C|WC}][ERROPT={ACC|SKP|ABE}]
  [,DSORG=PS][,MACRF=({GL|PL|GL,PL})
  [,DDNAME=symbol][,RECFM={F|U|V}
  [,LRECL=absexp][,BLKSIZE=absexp]
  [,BFTEK=S][,BUFNO=absexp]
  [,BFALN= F D ][,BUFL=absexp]
  [,BUFCB=relexp][,EODAD=relexp]
  [,SYNAD=relexp])
```

A description of the DCB subparameters follows.

DEN={0|1|2}

can be used with magnetic tape, and specifies a value for the tape recording density in bits per inch as listed in Table 8.

Table 8. DEN Values

DEN Value	TAPE RECORDING DENSITY (BITS/INCH)	
	Model 2400	
	7 Track	9 Track
0	200	-
1	556	-
2	800	800

TRTCH={C|E|T|ET|U|UC|}

is used as with 7-track tape to specify the tape recording technique, as follows:

C - specifies that the data conversion feature is to be used; if data conversion is not available, only format-F and -U records are supported by the control program.

E - specifies that even parity is to be used; if omitted, odd parity is assumed.

T - specifies that BCDIC to EBCDIC translation is required.

ET- specifies that even parity is to be used and BCDIC to EBCDIC translation is required.

U - unblock (permit) data checks on a printer with the Universal Character Set feature.

UC- unblock data checks on a printer and use chained scheduling.

PRTSP={0|1|2|3}

specifies the line spacing on a printer as 0, 1, 2, or 3.

MODE={C|E}

can be used with a card reader, a card punch, or a card read punch and specifies the mode of operation as follows:

C - the card image (column binary) mode

E - the EBCDIC code

If this information is not supplied by any source, E is assumed.

STACK={1|2}

can be used with a card reader, a card punch, or a card read punch and specifies which stacker bin is to receive the card. Either 1 or 2 is specified. If this information is not supplied by any source, 1 is assumed.

OPTCD={W|C|WC}

specifies an optional service to be performed by the control program, as follows.

W - perform a write validity check (on direct-access devices only).

C - process using the chained scheduling method.

WC- perform a validity check and use chained scheduling.

If this information is not supplied by any source, none of the services are provided, except in the case of the IBM 2321 direct-access device where OPTCD=W is specified by the operating system.

EROPT={ACC|SKP|ABE}

specifies the option to be executed if an error occurs and either there is no synchronous exceptional error (SYNAD) exit routine or there is a SYNAD routine and the programmer wishes to return from it to his processing program. One of the following is specified:

ACC - accept error block

SKP - skip error block

ABE - terminate the task

Table 9 indicates the choices that are permitted for each type of data set processing.

Table 9. Error Options for QSAM

OPERAND	PROCESS DATA SET FOR	
	INPUT,RDBACK	OUTPUT
ACC	X	X[1]
SKP	X	
ABE	X	X

[1]Valid for printer only.

DSORG=PS

specifies the organization of the data set

ALLOCATING SPACE FOR INDEXED SEQUENTIAL DATA SETS

Indexed sequential data sets consist of one, two, or three areas:

- Prime area. This area contains data records and the accompanying track indexes. It exists in all indexed sequential data sets.
- Overflow area. This area contains data records that overflow from tracks of the prime area when records are added to the data set. This area may or may not exist in an indexed sequential data set.
- Index area. This area contains the master and cylinder indexes for an indexed sequential data set. It exists for any data set that has a prime area on more than one cylinder.

The areas allocated and their locations depend on the parameters specified in the DD statement or statements that define the data set. For a description of the parameters and subparameters that can be used in DD statements defining a new indexed sequential data set or specifying an existing one, refer to the publication IBM System/360 Operating System: Job Control Language, Form C28-6539.

DCB FOR CREATING INDEXED SEQUENTIAL DATA SETS

```
DCB=( [,OPTCD={WLI}] ,DSORG=IS
  [,MACRF=(PL)] [,DDNAME=symbol]
  [,RECFM={F|FB}] [,LRECL=absexp]
  [,BLKSIZE=absexp] [,RKP=absexp]
  [,KEYLEN=absexp]
  [,BUFNO=absexp]
  [,SYNAD=relexp])
```

OPTCD

OPTCD={WLI}

specifies an optional service to be performed by the program, as follows:

W - a write validity check (on direct-access devices only)

L - delete option: user marks records for deletion; records so marked may actually be deleted when new records are added to the data set.

I - use independent overflow area.

If this information is not supplied by any source, none of the services are provided, except in the case of the IBM 2321 direct-access device where OPTCD=W is specified by the operating system.

DSORG=IS

specifies the organization of the data set as IS (an indexed sequential organization). This subparameter is required to be supplied by the programmer in the DD statement.

MACRF=(PL)

specifies the macro instruction that will be used in processing the data sets, as follows:

PL - indicates that locate mode PUT macro instructions are to be used.

DDNAME=symbol

specifies the name of the DD statement that will be used to describe the data set to be processed.

RECFM={F|FB}

specifies the characteristics of the record in the data sets, as follows:

F - fixed-length records

FB - fixed-length, blocked records

LRECL=absexp

specifies the length of a logical record in bytes.

BLKSIZE=absexp

specifies the maximum length of a block in bytes. For fixed-length records, the block must be an integral multiple of the LRECL value.

RKP=absexp

specifies the relative position of the first byte of the record key within each logical record. The value specified cannot exceed the logical record length minus the record key length.

KEYLEN=absexp

specifies the length of the record key, in bytes, associated with a logical record. The maximum length of the record key is 255 bytes.

BUFNO=absexp

specifies the number of buffers to be assigned to the data control block. The maximum number that can be specified is 255; however, the number must not exceed the limit on input/output requests established during system generation. This information can be supplied by the DD statement or the user's problem program.

SYNAD=relexp

specifies the address of the user's synchronous error exit routine. The routine is entered if input/output errors result from an attempt to process data records. If no routine is specified and an error occurs, the option specified by the EROPT parameter is executed.

ACCESSING INDEXED SEQUENTIAL DATA SETS

When accessing and/or updating indexed sequential data sets, the DCB subparameters specified for creating indexed sequential data sets are applicable with the following differences, and additions.

DIFFERENCES

[,MACRF={(GL)|(GL,PU)|(R)|(RU,WUA)}]

- G - indicates GET macro instruction
  L - indicates locate mode
- P - indicates PUT macro instruction
  U - indicates sequential updating
- R - indicates READ macro instruction
  U - indicates read for update
- W - indicates WRITE macro instruction
  UA - indicates add new records, update existing records.

ADDITIONS

[,NCP=1]

specifies the number of channel programs to be established for this data control block. The value 1 is supplied by the compiler.

[,MSWA=relexp]

specifies the address of a main storage work area reserved for the control program.

If specified when fixed-length records are being added to the data set, the control program uses the work area to speed up record insertion.

[,SMSW=absexp]

specifies the number of bytes reserved for the main storage work area. For unblocked records, the work area must be large enough to contain the count, key, and data fields of all the blocks on one track. For blocked records, the work area must be large enough to contain one logical record plus the count and data fields of all the blocks on one track. The maximum number of blocks on one track is 32,767.

[,EODAD=relexp]

specifies the address of the user's end-of-data set exit routine for input data sets. This routine is entered when the user requests a record and there are no more records to be retrieved. If no routine has been provided, the task is abnormally terminated.

Table 11 shows the values supplied for DCB subparameters by the COBOL compiler, by statements in the COBOL source program, and those subparameters that may be supplied by a DD statement for an indexed sequential data set.

Table 11. DCB Subparameter Values For Indexed Sequential Data Set

DCB Parameter	Value Supplied Unconditionally By COBOL Compiler	Value Supplied by COBOL Source Statement	Value Supplied By DD Statement
OPTCD	WLI		
DSORG	IS		DSORG=IS
MACRF			
Sequential	GL	OPEN INPUT	
	GL,PU	OPEN I-O	
	PL	OPEN OUTPUT	
Random	R	OPEN INPUT	
	RU,WUA	OPEN I-O	
DDNAME		External-name in ASSIGN Clause	
RECFM		RECORDING MODE Clause	
LRECL		RECORD CONTAINS Clause	
BLKSIZE		BLOCK CONTAINS Clause	
RKP		RECORD KEY Clause	
KEYLEN		RECORD KEY Clause	
NCP	1		
MSWA		TRACK AREA Clause	
BUFNO		RESERVE Clause	BUFNO=nnn
SMSW		TRACK AREA Clause	
EODAD		AT END Clause	
SYNAD		USE Statement Option 5	

DCB FOR CREATING DIRECT OR RELATIVE ORGANIZATION DATA SET

```
DCB=([,OPTCD=W] [,DSORG=PS]
  [,MACRF=(WL)] [,DDNAME=symbol]
  [,RECFM={F|V|U}] [,LRECL=absexp]
  [,BLKSIZE=absexp] [,DEVD=DA,KEYLEN=value]
  [,NCP=1] [,EODAD=relexp]
  [,SYNAD=relexp])
```

OPTCD=W

specifies an optional service to be performed by the program, as follows:

W - a write validity check (on direct-access devices only)

If this information is not supplied by any source, the service is not provided, except in the case of the IBM 2321 direct-access device where OPTCD=W is specified by the operating system.

DSORG=PS

specifies the organization of the data set as PS (a physical sequential organization)

MACRF=(WL)

specifies the macro instruction that will be used in processing the data sets, as follows:

W - indicates use of WRITE macro instruction
L - indicates LOAD mode for direct data set

DDNAME=symbol

specifies the name of the DD statement that will be used to describe the data set to be processed.

RECFM={F|V|U}

specifies the characteristics of the record in the data set, as follows:

F - fixed-length records

V - variable-length records
U - undefined records

LRECL=absexp

specifies the length of a format-F logical record in bytes or the maximum length of a format-V or format-U logical record.

BLKSIZE=absexp

specifies the maximum length of the block in bytes for format-F records. The length must be an integral multiple of the LRECL value. For format-V records, the length must include the four-byte block length field that is recorded at the beginning of each block.

DEVD=DA,KEYLEN=value

specifies the device or devices on which the data set resides
DA - specifies a direct-access device
KEYLEN - specifies the length of the key, in bytes, associated with a physical block.

NCP=1

specifies the maximum number of READ or WRITE macro instructions that are issued before a CHECK macro instruction.

SYNAD=relexp

specifies the address of the user's synchronous error exit routine. The routine is entered if input/output errors result from an attempt to process data records. If no routine is specified and an error occurs, the option specified by the EROPT parameter is executed.

ACCESSING DIRECT OR RELATIVE ORGANIZATION DATA SETS

When accessing and/or updating direct data sets, the DCB subparameters specified for creating direct data sets are applicable, with the following differences and additions.

DIFFERENCES

[,OPTCD={WE|WR}]

W - indicates a write validity check be performed
E - indicates an extended search be performed
R - indicates that relative block addresses be used

[,DSORG=DA]
DA - indicates direct or relative organization

```
          R
[,MACRF=  WL
          (RKC,[WAKC])
          (RIC,[WAIC])
```

R - indicates use of READ macro instruction
K - indicates that search argument is a key
I - indicates that search argument is a block identification
W - indicates use of WRITE macro instruction
A - indicates that blocks are to be added to the data set
C - indicates use of check macro instruction

ADDITIONS

[,KEYLEN=absexp]

specifies the length of the key for each physical record in bytes

[,LIMCT=absexp]

specifies the maximum number of blocks or tracks searched when the extended search option is chosen

[,EODAD=relexp]

specifies the address of the user's end-of-data set exit routine for input data sets. This routine is entered when the user requests a record and there are no more records to be retrieved. If no routine has been provided, the task is abnormally terminated.

Table 12 shows the values supplied for DCB subparameters by the COBOL compiler, by statements in the COBOL source program, and those subparameters that may be supplied by a DD statement for a direct-access data set.

Table 12. DCB Subparameter Values for Direct-Access Data Sets

DCB Parameter	Value Supplied Unconditionally by COBOL Compiler	Value Supplied by COBOL Source Statement	Value Supplied by DD Statement
OPTCD			
Direct organization	WE		
Relative organization	WR		
DSORG			
Sequential access	PS		
Random-access	DA		
MACRF			
Sequential-access	R	OPEN INPUT	
	WL	OPEN OUTPUT	
Random-access			
Direct organization	RKC	OPEN INPUT	
	RKC,WAKC	OPEN I-O	
Relative organization	RIC	OPEN INPUT	
	RIC,WAIC	OPEN I-O	
DDNAME		External-name in ASSIGN clause	
DEVD	DA,KEYLEN=nnn (nnn=0 - 255)	SYMBOLIC KEY Clause	
RECFM		RECORDING MODE Clause	
LRECL		RECORD CONTAINS Clause	
BLKSIZE		BLOCK CONTAINS Clause	
NCP	1		
KEYLEN		SYMBOLIC KEY Clause	
LIMCT		APPLY Clause Option 1	
EODAD		AT END Clause	
SYNAD		USE Statement Option 5	

The following DD statements are examples for processing Indexed Sequential, Direct, Relative sets.

Example of DD statements for Indexed Sequential organization:

```
//GO.SYSUT5 DD DSNAME=ISAM(INDEX),        X
//             UNIT=2311,                 X
//             VOLUME=SER=111111,         X
//             DCB=(,DSORG=IS),           X
//             SPACE=(CYL,(1)),           X
//             DISP=(NEW,KEEP)
//          DD DSNAME=ISAM(PRIME),        X
//             UNIT=2311,                 X
//             VOLUME=SER=111111,         X
//             DCB=(,DSORG=IS),           X
//             SPACE=(CYL,(3)),           X
//             DISP=(NEW,KEEP)
//          DD DSNAME=ISAM(OVERFLOW),     X
//             UNIT=2311,                 X
//             VOLUME=SER=111111,         X
//             DCB=(,DSORG=IS),           X
//             SPACE=(CYL,(1)),           X
//             DISP=(NEW,KEEP)
```

This example specifies:

- that an indexed sequential data set (named ISAM), is to be processed on a 2311 disk pack;
- that the volume serial number of the volumes required by the data set is 111111;
- that the data set is to be kept after execution of the run;
- that the prime area consists of three cylinders, the overflow area, and the index area of one cylinder each, and
- that the COBOL external name for the data set is SYSUT5.

Example of DD statement for Direct or Relative organizations:

```
//GO.SYSUT6 DD DSNAME=&RANDOM,UNIT=SYSDA,X
//             SPACE=(TRK,(10,5))
```

This example specifies:

- that a temporary data set (named RANDOM) is to be processed on a direct-access device;
- that the data set be allocated a space of 10 tracks, with a secondary allocation of 5 tracks, if needed;
- that the COBOL external name for this data set is SYSUT6.

Example of DD statement for sequential organization:

```
//GO.SYSUT7 DD DSNAME=SEQUENTIAL,         X
//             UNIT=2311,                 X
//             DISP=(NEW,DELETE),         X
//             DCB(,OPTCD=W),             X
//             SPACE=(TRK,(20,5))
```

This example specifies:

- that a data set (named SEQUENTIAL) is to be processed on a 2311 disk pack;
- that the data set is to be deleted after execution;
- that the data set be allocated 20 tracks with a secondary allocation of 5 tracks, if needed; and
- that the COBOL external name for the data set is SYSUT7

Note: For sequential, direct, and relative organizations, essentially the same DD statements can be used.

Table 17. Linkage Registers

REGISTER NUMBER	REGISTER NAME	FUNCTION
1	Argument Register	Address of the argument list passed to the called subprogram.
13	Save Register	Address of the area reserved by the calling program in which the contents of certain registers are stored by the called program.
14	Return Register	Address of the location in the calling program to which control is returned after execution of the called program.
15	Entry Point Register	Address of the entry point in the called subprogram.

ARGUMENT LIST

Every assembler-written subprogram that calls another subprogram must reserve an area of storage (argument list) in which the argument list used by the called subprogram is located. Each entry in the parameter list occupies four bytes and is on a full-word boundary.

In the first byte of each entry in the parameter list, bits 1 through 7 contain zeros. However, bit 0 may contain a 1 to indicate the last entry in the parameter area.

The last three bytes of each entry contain the 24-bit address of the argument.

SAVE AREA

An assembler subprogram that calls another subprogram must reserve an area of storage (save area) in which certain registers (i.e., those used in the called subprogram and those used in the linkage to the called subprogram) are saved.

The maximum amount of storage reserved by the calling subprogram is 18 words. Figure 47 shows the layout of the save area and the contents of each word.

A called COBOL subprogram does not save floating-point registers. The programmer is responsible for saving and restoring the contents of these registers in the calling program.

AREA	
(word 1)	This word is a part of the standard linkage convention established under the operating system. The word must be reserved for proper addressing of the succeeding entries. However, an assembler subprogram may use the word for any desired purpose.
AREA+4 (word 2)	The address of the previous save area; that is, the save area of the subprogram that called this one.
AREA+8 (word 3)	The address of the next save area; that is, the save area of the subprogram to which this subprogram refers.
AREA+12 (word 4)	The contents of register 14; that is, the return address.
AREA+16 (word 5)	The contents of register 15; that is, the entry address.
AREA+20 (word 6)	The contents of register 0.
AREA+24 (word 7)	The contents of register 1.
. . .	. . .
AREA+68 (word 18)	The contents of register 12.

Figure 47. Save Area Layout and Word Contents

```
deckname  START  0
          ENTRY  name1
          EXTRN  name2
          USING  *,15
*  Save Routine
name1     STM    14,r1,12(13)        The contents of registers 14, 15, and 0 through
*                                      r1 are stored in the save area of the calling
*                                      program (previous save area). r1 is any number
*                                      from 0 through 12.
          LR     r2,13               Loads register 13, which points to the save area
*                                      of the calling program, into any general
*                                      register, r2, except 0 and 13.
          LA     13,AREA             Loads the address of this program's save area
*                                      into register 13.
          ST     13,8(0,r2)          Store the address of this program's save area
*                                      into word 3 of the save area of the calling
*                                      program.
          ST     r2,4(0,13)          Stores the address of the previous save area
*                                      (i.e., the same area of the calling program)
*                                      into word 2 of this program's save area.
          BC     15,prob1
AREA      DS     18F                 Reserves 18 words for the save area.  This is
*                                      last statement of save routine.
prob1     User-written program statements
*  Calling Sequence
          LA     1,ARGLST            First statement in calling sequence.
          L      15,ADCON
          BALR   14,15
*         Remainder of user-written program statements
*  Return Routine
          L      13,4(0,13)          First statement in return routine.  Loads the
*                                      address of the previous save area back into
*                                      register 13.
          LM     2,r1,28(13)         The contents of registers 2 through r1, are
*                                      restored from the previous save area.
          L      14,12(13)           Loads the return address, which is in word 4 of
*                                      the calling program's save area, into register
*                                      14.
          MVI    12(13),X'FF'        Sets flag FF in the save area of the calling
*                                      program to indicate that control has retunred
*                                      to the calling program.
          BCR    15,14               Last statement in return routine.
ADCON     DC     A(name2)            Contains the address of subprogram name2.
*  Parameter List
ARGLST    DC     AL4(arg1)           First statement in parameter area setup.
          DC     AL4(arg2)
          DC     X'80'               First byte of last argument sets bit 0 to 1.
          DC     AL3(argn)           Last statement in parameter area setup.
```

•Figure 48. Sample Linkage Routines Used with a Calling Subprogram

Example

The linkage conventions used by an assembler subprogram that calls another subprogram are shown in Figure 48. The linkage should include:

1. The calling sequence.

2. The save and return routines.

3. The out-of-line parameter list. (An in-line parameter list may be used; see In-line Parameter List.)

4. A save area on a fullword boundary.

LOWEST LEVEL SUBPROGRAM

If an assembler subprogram does not call any other subprogram (i.e., if it is at the lowest level), the programmer should omit the save routine, calling sequence, and

IEP615I E READ 'AT END' REQUIRED FOR FILES WITH ACCESS SEQUENTIAL

System Action: The entire statement is skipped.

IEP616I E 'INVALID KEY' REQUIRED FOR FILES WITH ACCESS RANDOM

System Action: The entire statement is dropped.

IEP617I E WRITE 'FROM' OPTION REQUIRED WITH APPLY WRITE-ONLY

System Action: The entire statement is dropped.

IEP618I E REWRITE INVALID ON DIRECT OR RELATIVE SEQUENTIAL FILES

System Action: The entire statement is dropped.

IEP619I E WRITE INVALID FOR RELATIVE RANDOM FILE

System Action: The entire statement is dropped.

IEP620I E WRITE 'INVALID KEY' REQUIRED FOR INDEXED SEQUENTIAL FILE

System Action: The entire statement is dropped.

IEP621I E OPEN 'I-O' INVALID FOR DIRECT OR RELATIVE SEQUENTIAL FILES

Explanation: On OPEN and CLOSE no code is generated for the file in error.

System Action: Syntax scan skips to the next file in the statement.

IEP622I C OPEN 'OUTPUT' INVALID FOR FILES WITH ACCESS RANDOM, I-O ASSUMED.

Explanation: See message IEP621I.

IEP623I E OPEN 'REVERSED' VALID ONLY ON STANDARD SEQUENTIAL FILES

Explanation: See message IEP621I.

IEP624I W A FILE WHOSE ORGANIZATION IS INDEXED AND ACCESS IS SEQUENTIAL (QISAM) OPENED AS OUTPUT MAY NOT ALSO BE OPENED AS INPUT OR I-O IN THE SAME PROGRAM WITH THE SAME FILE-NAME.

IEP625I E OPEN 'REVERSED' INVALID FOR FILES WITH FORMAT V RECORDS

IEP626I E CLOSE 'UNIT' OR 'REEL' VALID ONLY FOR STANDARD SEQUENTIAL FILES

Explanation: See message IEP621I.

IEP627I E 'INVALID KEY' INVALID FOR STANDARD, DIRECT OR RELATIVE SEQUENTIAL FILES.

System Action: The clause is skipped.

IEP628I E 'ACTUAL KEY' REQUIRED FOR DIRECT SEQUENTIAL OUTPUT FILES

IEP700I E IDENTIFICATION DIVISION NOT FOUND

IEP701I E DATA DIVISION NOT FOUND. COMPILATION CANCELED.

IEP702I E PROCEDURE DIVISION NOT FOUND. COMPILATION CANCELED.

IEP703I E SOURCE PROGRAM EXCEEDS INTERNAL LIMITS. COMPILATION CANCELED.

IEP704I E DATA-NAME TABLE OVERFLOW. COMPILATION CANCELED.

Explanation: The data-name attribute table has a maximum size of 64K bytes.

User Response: Reduce the length of data-names, and recompile.

IEP705I NO DIAGNOSTICS IN THIS COMPILATION.

IEP709I W INCORRECT EXECUTE PARAMETER - XXX.

IEP710I W BUFSIZE GREATER THAN BLKSIZE - TRUNCATED TO BLKSIZE

Explanation: For the devices specified, the BUFSIZE exceeded the maximum allowable block size. A BUFSIZE equal to the allowable block size is used instead.

For additional information, see "Compiler Options" under "Job Processing."

LOAD MODULE EXECUTION DIAGNOSTIC MESSAGES

Load module execution diagnostic messages are of two types: object time messages, and operator messages.

OBJECT TIME MESSAGES

Most object time messages are self explanatory. Where necessary, examples are included to explain the message.

IEP999I MINUS BASE MADE POSITIVE & FLOATING POINT EXPONENTIATION CONTINUED.

IEP998I ZERO BASE TO POSITIVE EXPONENT - FLOATING-POINT ANSWER MADE ZERO.

IEP997I ZERO BASE TO MINUS EXPONENT - FLOATING-POINT ANSWER IS MAX F.P. NUMBER.

IEP996I RESULT TOO BIG - FLOATING-POINT EXPONENTIATION ANSWER IS MAX F.P. NUMBER.

IEP993I ZERO BASE TO MINUS EXPONENT - PACKED EXPONENTIATION RESULT MADE ALL NINES.

OPERATOR MESSAGES

In addition to system diagnostic and object time messages the COBOL load module may issue operator messages.

The following message is generated by STOP 'literal'.

IEP000D text provided by object program.

Explanation: This message is issued at the programmer's discretion to indicate possible alternative action to be taken by the operator.

Operator Response: Follow the instructions given both by the message and on the job request form supplied by the programmer.

If the job is to be resumed, issue a REPLY command with a text field that contains any 1-character message.

The following message is generated by an ACCEPT ... FROM CONSOLE.

IEP990D 'AWAITING REPLY'

Explanation: This message is issued by the object program when operator intervention is required.

Operator Response: Issue a REPLY command. (The contents of the text field should be supplied by the programmer on the job request form.)

DEBUG PACKET ERROR MESSAGES

The following is a complete list of precompile error messages. They apply to errors in the debugging packets only.

IEP850I TABLE OF DEBUG REQUESTS OVERFLOWED. RUN TERMINATED.

IEP851I THE FOLLOWING CARD DUPLICATES A PREVIOUS *DEBUG CARD. THIS PACKET WILL BE IGNORED.

IEP852I THE FOLLOWING PROCEDURE DIVISION NAMES WERE NOT FOUND. INCOMPLETE DEBEG EDIT IS NOT TERMINATED.

IEP853I THE FOLLOWING *DEBUG CARD DOES NOT CONTAIN A VALID LOCATION FIELD. THIS PACKET WILL BE IGNORED.

IEP854I IDENTIFICATION DIVISION NOT FOUND. RUN TERMINATED.

IEP855I DEBUG EDIT RUN COMPLETE. INPUT FOR COBOL COMPILATION ON SYSUT4.