File Number S360-24
Form C24-5025-3

DOS
TOS

IBM Systems Reference Library

IBM System/360
Disk and Tape Operating Systems
COBOL Programmer's Guide

This publication is designed to aid the COBOL programmer. Its purpose is to provide guidance and examples in the techniques of COBOL programming in the Disk and Tape Operating Systems, and to expose the user to the components of the Control Program and facilities of IBM System/360 Disk and Tape Operating Systems.

The prerequisites for a thorough understanding of the COBOL language are:

IBM System/360 Disk and Tape Operating Systems COBOL Language Specifications, Form C24-3433.

Publications closely relate to this one are:

IBM System/360 Disk Operating System, System Control and System Service Programs, Form C24-5036.

IBM System/360 Tape Operating System, System Control and System Service Programs, Form C24-5034.

IBM System/360 Disk Operating System, Supervisor and Input/Output Macros, Form C24-5037.

IBM System/360 Tape Operating System, Supervisor and Input/Output Macros, Form C24-5035.

IBM System/360 Disk Operating System, Data Management Concepts, Form C24-3427.

IBM System/360 Tape Operating System, Data Management Concepts, Form C24-3430.

IBM System/360 Disk Operating System, System Generation and Maintenance, Form C24-5033.

IBM System/360 Tape Operating System, System Generation and Maintenance, Form C24-5015.

IBM System/360 Principles of Operation, Form A22-6821.

The titles and abstracts of related publications are listed in the IBM System/360 Bibliography, Form A22-6822.

Fourth Edition, January 1967

This edition, Form C24-5025-3, is a major revision of Form C24-5025-2 and obsoletes it and Technical Newsletter N24-5189. It should be reviewed in its entirety. Changes are indicated in two ways:

1. A dot (●) next to a figure title or page number indicates that the entire figure or page should be reviewed.

2. A vertical line appears to the left of the affected text and to the left of affected parts of figures.

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.

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, New York 10020.

PREFACE

This edition incorporates Direct Access Storage Device (DASD) capabilities and multiprogramming for Disk and Tape Operating Systems. Where material applies only to Disk Operating System, it is so defined.

The purpose of this publication is to help COBOL programmers using COBOL for Disk and Tape Operating Systems. Several basic examples are given with the idea that the beginning programmer is interested in preparing his deck for processing as quickly as possible with a minimum of effort.

Section I describes the information needed in preparation for processing COBOL programs within the framework of Disk and Tape Operating Systems.

Section II contains examples of processing COBOL programs using the Tape Operating System.

Section III contains examples of processing COBOL programs using the Disk Operating System.

Section IV contains information on the types of output that can be expected from IBM System/360 Disk and Tape Operating Systems.

Section V describes how to use the COBOL debugging language.

Section VI discusses the use of Disk and Tape Operating Systems libraries at the COBOL language level.

Section VII discusses calling routines, subprograms and overlay structures.

Section VIII discusses multiprogramming and direct-access data organization considerations as well as techniques and hints for producing effective COBOL coding, which help reduce main storage requirements, execution time, and/or linkage editing time.

Section IX describes the environment within which the COBOL program must operate. Included is a brief discussion of the concepts and configuration of Disk and Tape Operating Systems.

Appendix A contains the subroutine that effects program overlays and a discussion of assembler language subprograms. Included are Disk and Tape Operating Systems linkage conventions for COBOL programs when using assembler language subprograms.

Appendix B contains a table of COBOL reference formats.

Appendix C illustrates Tape Operating System standard tape file labels.

Appendix D illustrates Disk Operating System standard disk file labels.

Appendix E contains a list of options for IBM System/360 Disk and Tape Operating Systems OPTION Job-Control statement.

Appendix F contains examples of COBOL programs.

Appendix G contains a table of subroutines used by COBOL and related operations that cause the subroutines to be invoked.

Appendix H contains a list of compiler diagnostic messages, their descriptions, and the action taken. Also included in this section are a list of object time messages, debugging error messages and an illustration of object storage layout.

Many powerful parameters can be used in the job control cards. Few are required to be used. Generally, only the essential control cards needed (and their basic parameters) for simpler, complete executions are illustrated.

The advanced programmer, who may wish to use some of the options available, but not used in the examples, may refer to the complete list of options in Appendix E.

Where considered necessary, reference is made to the appropriate publication for clarification of a subject.

This section is devoted to preparing a COBOL source statement deck suitable for execution on Disk and Tape Operating Systems. The programmer who wishes to prepare his deck for basic executions should find this section easy to interpret, and fulfilling the requirements for getting his job executed. The examples included illustrate how some of the facilities of Disk and Tape Operating Systems are used. Only a few of the many powerful options available to the programmer are included.

The examples show what types of processing are accomplished at the COBOL Disk and Tape Operating Systems level. When required, Disk and Tape Operating Systems terminology is discussed. Where a programmer wishes to use options not illustrated, reference is made to the appropriate publication in which a complete, detailed discussion on the subject can be found.

SYSTEM LIBRARIES

The three libraries in the system are the core image library, source statement library, and the relocatable library.

All programs in the system (IBM-supplied and user programs) are loaded from the core image library.

The source statement library is used to store source modules. It provides extended program-compilation capability.

The relocatable library is used to store object modules that can be used for subsequent linkage with other program modules. (A module also can be a complete program.)

STAGES OF PROGRAM DEVELOPMENT

In program development, the programmer codes sets of source statements that may be a complete program or part of a program. These source statements are then compiled, or assembled, into a relocatable machine-language program which, in turn, must be edited into an executable program and may be combined with other programs.

A set of source statements processed by a language translator (Assembler, COBOL, FORTRAN, RPG, or PL/I) is referred to as a source module.

The output of a language translator is referred to as an object module. All object modules must be further processed by the linkage editor before they can be executed in the Disk and Tape Operating Systems.

The output of the linkage editor is called a program phase. (Output may consist of more than one phase.)

A program consists of at least one phase and may be composed of many phases. Successive phases of a multi-phase program are often called overlays.

For disk, the output of the linkage editor consists of one or more program phases in the temporary part of the core image library. A phase is in executable, nonrelocatable, core image form.

For tape, the output of the linkage editor consists of one or more program phases on SYSLNK, in executable, nonrelocatable core-image form.

STRUCTURE OF A COBOL PROGRAM

Refer to Figure 1 for an illustration of the procedure for development of a COBOL program.

A COBOL source module is a group of codes, statements, and clauses in the COBOL language which the COBOL compiler accepts as input. When the source module is compiled, the output, which is called an object module, consists of one control section. This control section is a block of machine instructions assigned to contiguous main-storage locations, and consists of control dictionaries and the text of one control section. The control dictionaries contain the information necessary for the linkage editor to resolve cross references between different object modules, and the text is the actual instructions and data fields of the object module. The object module is in relocatable form. The input to the linkage editor for building a phase must consist of at least one complete control section. A phase is a program section loaded as a single overlay, and may be a complete program.

Debug packets, appropriately positioned in the input job stream, automatically become merged with the COBOL source statements before compilation.

Any source statements (modules) that are part of the source statement library, and copied from the library by use of the COBOL library facility statements, automatically become part of the input stream to the COBOL compiler during compilation.

All these statements, combined, are processed by the COBOL compiler to become an object module(s).

The object module(s) [control section(s)], along with any assembler object modules and modules from the relocatable library included in the input job stream through the job-control facility are processed by the linkage editor to become program phases. They can then be executed as a complete program.

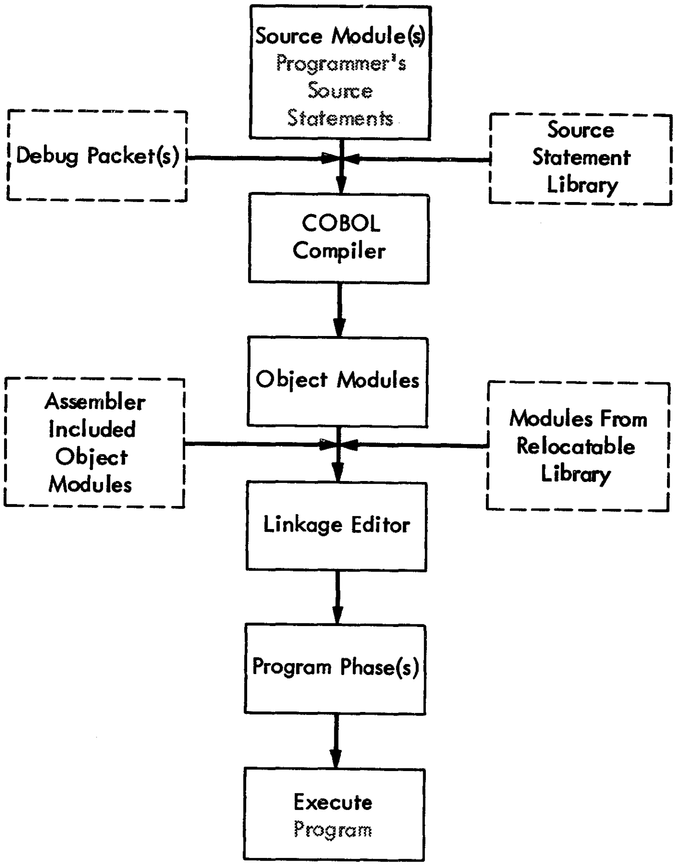

Source Module(s) Programmer's Source Statements
Debug Packet(s)
Source Statement Library
COBOL Compiler
Object Modules
Assembler Included Object Modules
Modules From Relocatable Library
Linkage Editor
Program Phase(s)
Execute Program

Figure 1. Stages of Development of a COBOL Program

MULTIPROGRAMMING

There are two types of problem programs in multiprogramming: background and foreground. Background programs are initiated by job control from the batch-job input stream. Foreground programs are initiated by the operator from the printer-keyboard. Foreground programs do not execute from a stack. When one is completed, the operator must explicitly initiate the next program.

Background and foreground programs initiate and terminate asynchronously from each other. Neither is aware of the other's existence. Main storage equal to or in excess of 32K is required for multiprogramming support.

COBOL source programs cannot be compiled as foreground programs. COBOL object programs can be executed as foreground programs (with certain restrictions), and as background programs with no restrictions. Refer to Multiprogramming Considerations in Section VIII for these restrictions.

At linkage edit time, the program to be executed becomes either a background or a foreground program.

For further details about multiprogramming, refer to the publication System Control and System Service Programs listed on the front cover of this manual.

BASIC TYPES OF OPERATIONS

The three basic operations that can be done are:

- Compilation
- Linkage Editing
- Execution

These operations can be combined to produce a comprehensive program. The discussions that follow explain each of these operations.

COMPILATION

Compilation is a process by which a source module(s) (in the case of COBOL, clauses, sentences, paragraphs, etc) are converted by a language translator (disk and tape COBOL compilers) into an object module (machine language text), which is a form acceptable to the linkage editor.

LINKAGE EDITING

Before the object module from the compiler is executed by System/360, it must be altered (edited) into a form acceptable for execution. This is accomplished by the linkage editor. The control dictionaries (output by the compiler as part of the object module) contain the information needed by the linkage editor to edit the module into executable form. The control dictionaries are resolved (by the linkage editor) to produce an executable program section. The linkage editor recognized these dictionaries and other card types by a 12-2-9 punch in column 1 and the identifier found in columns 2-4 of the following types of object module cards:

Identifier	Contents or Meaning
ESD	External Symbol Dictionary Card
TXT	Text Cards (Object Code)
RLD	Relocation Dictionary Card
END	End Card

All of these cards must be present in an object module in the indicated order.

For a more detailed description of the card types produced by the language translator, refer to the IBM publication, System Control and System Service Programs, listed on the cover of this manual.

EXECUTION

Execution is the process of obtaining a program phase, loading it into main storage, and executing the machine-language instructions contained in the phase. The first phase is called for by an execute (EXEC) job-control card. The following phases are called for by a FETCH (a supervisor call) within a phase. FETCH is not a facility available through the COBOL language. The control program prepares and controls the execution of all COBOL programs.

SYMBOLIC INPUT/OUTPUT ASSIGNMENT

Job control is responsible for assigning physical I/O units. Programs do not reference I/O devices by their actual physical addresses, but rather by symbolic names. The ability to reference an I/O device by a symbolic name rather than by a physical address provides advantages to both programmers and machine operators. The symbolic name of a device is chosen by the programmer from a fixed set of symbolic names. He can write a program that is dependent only on the device type and not on the actual device address. At execution time, the operator or programmer determines the actual physical device to be assigned to a given symbolic name. He communicates this to job control by a control statement (ASSGN). Job control associates the physical device with the symbolic name by which it is referenced.

A fixed set of symbolic names is used to reference I/O devices. No other names can be used. They are:

SYSRDR	Card reader, magnetic tape unit, or disk drive used for job control statements.
SYSIPT	Card reader, magnetic tape unit, or disk drive used as the input unit for programs.
SYSPCH	Card punch, magnetic tape unit, or disk drive used as the main unit for punched output.
SYSLST	Printer, magnetic tape unit, or disk drive used as the main unit for printed output.
SYSLOG	Printer-keyboard used for operator messages and to log job control statements. Can also be assigned to a printer.
SYSLNK	Disk extent used as input to the linkage editor.

SYSRES — System residence tape unit or area on a disk drive.

SYSSLB (Tape System only) — Tape unit used for the source statement library.

SYSRLB (Tape System only) — Tape unit used for the relocatable library.

SYS000-SYS244 — All other units in the system.

The first nine of the above names, termed system logical units, are used by the system control program and system service programs. Of these nine units, user background programs may also use SYSIPT for input, SYSLST and SYSPCH for appropriate output, and SYSLOG for operator communication. Normally, SYSRDR and SYSIPT both refer to the same device. Any additional devices in the system, termed programmer logical units, are referred to by names ranging consecutively from SYS000 to SYS244, with SYS000 to SYS009 being the minimum provided in any system.

Only SYS000-SYS244 are used by COBOL programs. The number of logical assignments that can be made depends on how many the generated system can accommodate. Examples of symbolic assignments for the tape resident system and the disk resident system are given in Figure 2.

For Disk Operating System, programmer logical units are defined at system generation time for each class of program (background, foreground-one, and foreground-two) to be run in the system. For example, in a multiprogramming environment, a unique SYS000 is defined for each class of program, a unique SYS001 is defined for each class of program, etc. The combined number of programmer logical units defined for the system may not exceed 245.

For the convenience of the user, two additional system logical unit names are defined for background programs. These names are used only in job control statements. Reference within a program (such as those given in Sections II and III) must name the particular logical unit to be used (SYSLST or SYSPCH, SYSRDR or SYSIPT). The additional system logical units are:

SYSIN — Name that can be used when SYSRDR and SYSIPT are assigned to the same card reader or magnetic tape unit. This name must be used when SYSRDR and SYSIPT are assigned to the same disk extent.

SYSOUT — Name that must be used when SYSPCH and SYSLST are assigned to the same magnetic tape unit.

With the exception of SYSLOG, foreground programs cannot reference any system logical unit. (System units are reserved for the exclusive use of background programs operating in a stacked-job environment.) Foreground programs may reference any programmer logical unit, SYS000-SYSnnn.

JOB-CONTROL STATEMENTS

Job control reads all control statements from the device identified as SYSRDR. Not all job control statements are needed by COBOL. Those required are JOB, EXEC, /* and /&. If disk labels are used, the VOL, XTENT, and DLAB statements are required. If tape labels are used, the VOL and TPLAB statements are required. All other statements are optional.

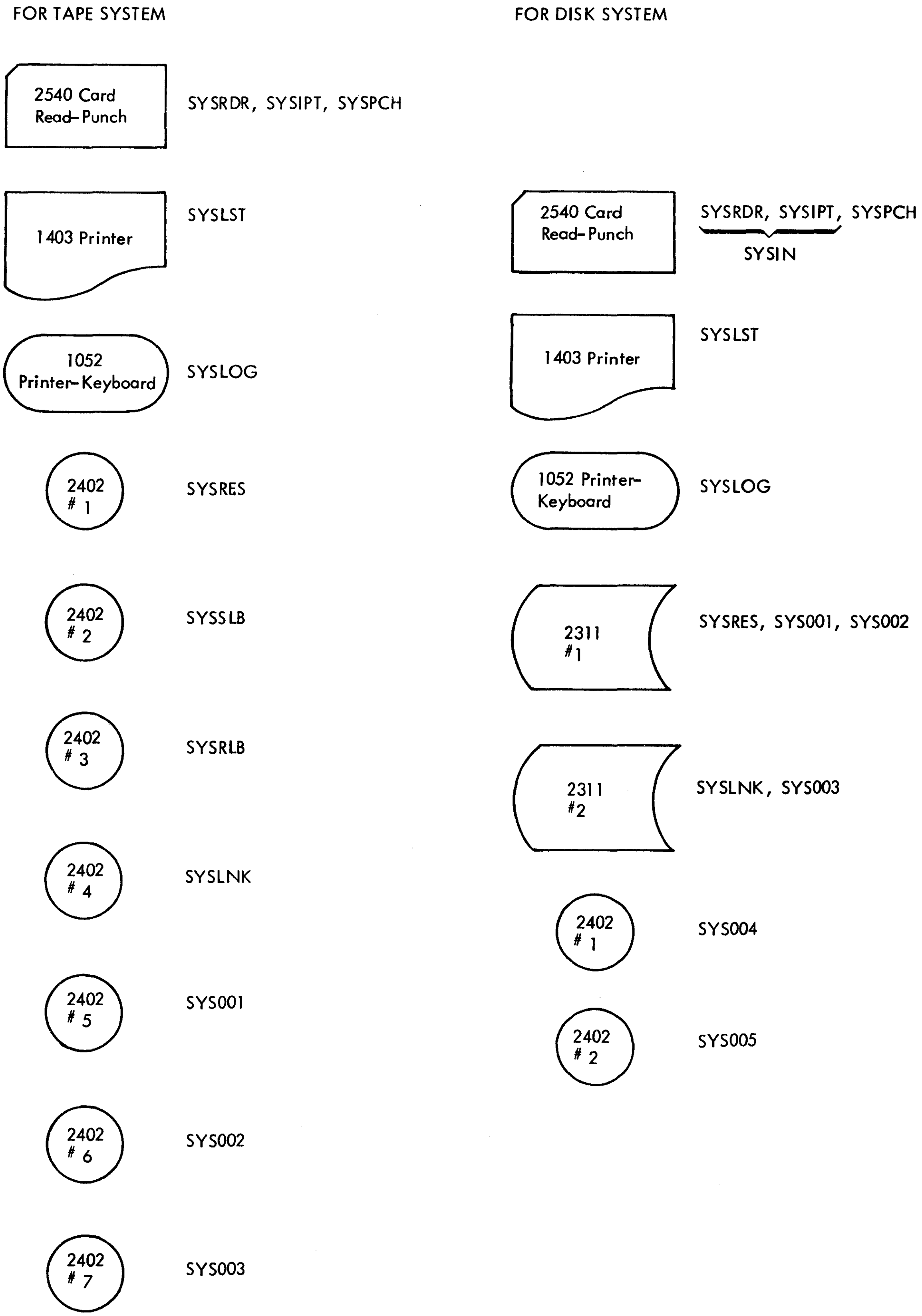


Figure 2. Example of Symbolic Device Assignment

A complete list of statements recognized is given in the IBM publication, *System Control and System Service Programs*, listed on the cover of this manual. A list of the statements most likely to be used by the COBOL user follows:

Operation	Meaning
JOB	Job name
EXEC	Execute program
ASSGN	I/O assignments
LBLTYP	Reserve storage for label information
VOL	Volume information
DLAB (Disk only)	Disk file label information
XTENT (Disk only)	Disk file extent
TPLAB	Tape file label information
OPTION	Option
PAUSE	Pause (for message to operator)
/*	End of data file
/&	End of job
*	Comment

SEQUENCE OF JOB-CONTROL STATEMENTS

The job control statements for a specific *job* always begin with a JOB statement and end with a /& (end of job) statement. A specific *job* consists of one or more *job steps*. Each *job step* is initiated by an EXEC statement. Preceding the EXEC statement are any job-control statements necessary to prepare for the execution of the specific job step. The only limitation on the sequence of statements preceding the EXEC statement is that discussed here for the label information statements. The following statements can precede the EXEC statement for a job step, and will be found most useful to the COBOL programmers.

ASSGN
LBLTYP
VOL
DLAB
XTENT
TPLAB
OPTION
PAUSE

The label statement must be in the order:

VOL		VOL
TPLAB	or	DLAB
		XTENT (one for each area of file in volume)

and must immediately precede the EXEC statement to which they apply.

INTRODUCTION TO JOB CONTROL STATEMENTS

Preparation of a COBOL source deck for execution on disk and tape operating systems entails combining required job control statements (and optional statements) with the source deck in a specific order.

The examples given are typical. Required statements are so indicated. No attempt is made to illustrate all of the options available to the programmer. For a full discussion of the options, refer to the IBM publication, *System Control and System Service Programs*, listed on the cover of this manual.

FORMAT OF JOB CONTROL STATEMENTS

All job-control statements are free form, with a few restrictive rules. The restrictions are pointed out within the examples that follow.

Certain rules must be followed when filling out control statements. The job control statement rules are:

1. Name. Two slashes (//) identify the statement as a control statement. They must be in columns 1 and 2. The second slash must be immediately followed by at least one blank. Exceptions to these rules are: the end-of-job statement (contains /& in columns 1 and 2), the end-of-file statement (contains /* in columns 1 and 2), and the comment statement (contains * in column 1 and a blank in column 2).

2. Operation. This field describes the type of control statement (the operation to be performed). It can be up to eight characters long. At least one blank follows its last character.

3. Operand. This field may be blank or may contain one or more entries separated by commas. The last term must be followed by a blank, unless its last character is in column 71.

CONTINUATION OF JOB CONTROL STATEMENT

Information starts in column 1 and cannot extend past column 71. The exceptions to this are the file-label statements (TPLAB and DLAB). Information for file-label statements can be specified on more than one card, in which case a continuation statement is required. Any non-blank character present in column 72 specifies that information is contained in the card image that follows. Columns 1-15 of the continuation statement are ignored (leave them blank). Begin continuation statement information in column 16. Note that the job-control continuation statements are used only for label statements (no other statements are checked for continuation).

JOB Statement: The format of the JOB card is:

```
// JOB jobname
```

Examples of the job statement follow:

```
// JOB ANYNAME
```

The // identifies the statement as a control statement and must always be in columns 1 and 2, and JOB indicates the type of control statement (the operation to be performed). In this case, it is a JOB card. ANYNAME is the symbolic name given the JOB. It is the operand of this card and must be one to eight characters in length.

If the system is equipped with the timer feature (must have been requested at system generation time), the time of day will be printed at the beginning and end of each job, automatically. Refer to the IBM publication System Generation and Maintenance listed on the cover of this manual for the method of setting the time of day. The time is printed on SYSLOG, and SYSLST, if present.

This means that the printer-keyboard (IBM 1052), and the printer (1403, etc) print the time of day after printing the JOB card, or the /& card.

The printer-keyboard skips to the next line to print the time of day (at the very beginning of the line), whereas the printer records the time of day beginning with print position 73 on the same line.

The programmer could take advantage of this, and use the available space for comments.

An example of a job control statement follows.

```
// JOB ANYNAME JOHN DOE, CALLPROG
```

This card is the same as the first except the operand is followed by comments.

EXEC Statement: The format of the EXEC statement is:

```
// EXEC [progname]
```

An example of this statement is:

```
// EXEC CBLPROG
```

EXEC is the execute control statement. It must be the last statement processed before a job step is executed. It indicates the end of job control statements for a job step, and that execution of the program is to begin. The program name, CBLPROG, is the name of the program to be executed, and must be one to eight alphameric characters long. If a program to be executed was just processed by the linkage editor (which means it is in executable format) the operand of the EXEC statement is left blank. For a COBOL program the program name in the EXEC card must be the phase name specified in a PHASE card of the first phase to be executed.

When control is given to a fetched phase, general register 2 contains the address of the uppermost byte of the appropriate program area.

ASSGN Statement: The format of the ASSGN statement is:

```
// ASSGN SYSxxx,deviceaddress  [{,X'ss'}]
                               [{,ALT  }]
```

An example of this statement is:

```
// ASSGN SYS004,X'00C'
```

During compilation, programs use these symbolic names (SYSIPT, etc) to refer to the I/O devices used in a system configuration. At execution time, the ASSGN statement is used to assign a symbolic unit (logical device SYS004) to a specific physical device (located at the address X'00C').

For example, an object program is to process card input. The preceding ASSGN statement could be used to assign the symbolic unit SYS004 to a card reader at address X'00C'. The first digit in the device address (X'00C') specifies the channel number in hexadecimal. A '0' means multiplexor channel; 1-6 means selector channels 1-6. The second and third digits designate the unit number in hexadecimal. 254 are allowed (00 to FE hexadecimal). For the example given, X'00C', the first 0 specifies the multiplexor channel, and the 0C specifies the unit number.

Example: `// ASSGN SYS005,X'181',X'90'`

This example is the same as the first except that the X'ss' specification is given. It determines the device specifications for seven-track tape. The X'90' specifies that:

- the number of bytes per inch = 800
- parity is odd
- the translate feature is off
- the convert feature is on

For the example given, X'181', the first 1 specifies selector channel 1, and 81 specifies the unit number.

When the COBOL SELECT statement is used, i.e., SELECT FILEA ASSIGN TO 'SYS005' UTILITY 2400. COBOL assigns a file (FILEA) to a logical device (SYS005).

In this example, the job control ASSGN statement assigns logical device SYS005 to the specific physical device located at the address X'181'.

The specific address of each device is determined by the system configuration.

The symbolic unit names that may be used with the ASSGN statement are:

```
SYSRDR
SYSIPT
SYSIN
SYSPCH
SYSLST
SYSLOG
SYSLNK
SYSSLB  (Tape System only)
SYSRLB  (Tape System only)
SYS000--SYS244
```

Note that the assignment for SYSOUT must be permanent, i.e., it is not reset between jobs. For this reason, it is not included in the preceding list. (Also note that COBOL source programs can only use SYS000--SYS244.)

For descriptions of the ALT parameter, refer to the publications Disk or Tape Operating System, System Control and Service Programs, as listed on the cover of this manual.

LBLTYP Statement: The format of the LBLTYP statement is:

```
// LBLTYP  {TAPE[(nn)]}
           {NSD (nn)  }
```

An example of this statement is:

```
// LBLTYP TAPE
```

This statement is required if the programmer's files contain standard labels.

The LBLTYP statement applies to both background and foreground programs. It is used to define the amount of main storage to be reserved at linkage edit time for processing tape and nonsequential disk-file labels in the problem program area of main storage.

TAPE(nn) For Tape Operating System, (nn) is used to specify the decimal number of pairs of VOL, TPLAB statements that appear immediately before the execution of the linkage edited program.

TAPE[(nn)] For Disk Operating System, TAPE is used only if tape files requiring label information are to be processed, and no nonsequential DASD files are to be processed. nn is optional, and is present only for future expansion (it is ignored by job control).

NSD(nn) (Disk only) Used if any nonsequential DASD files are to be processed, regardless of other file types to be used. nn specifies the largest number of extents to be used for a single file.

<u>VOL Statement</u>: The format of the VOL statement is:

```
// VOL SYSxxx,filename
```

An example of the VOL statement is:

```
// VOL SYS004,SYS004
```

VOL identifies this statement as the volume statement. It is used to check standard labels for tape or disk files. This statement is required for each file on a multiple volume file.

SYSxxx (the first operand) is the logical unit referenced. Filename (the second operand) identifies the file for the control program. The occurrence of two identical operands is peculiar to COBOL object programs, because SYS004 is both the filename by which the file is known to the control program, and the logical unit which is assigned to a device.

DLAB Statement (Disk System only): The format of the DLAB statement is:

```
    // DLAB 'label fields 1-3',       C
             xxxx,yyddd,yyddd,'systemcode'[,type]
```

Examples of DLAB statements are given in Section III, under Examples of Processing Using Disk Configuration. DLAB identifies this card as the disk-label statement. The disk-label statement contains file label information for disk label checking and creation. This card must immediately follow the VOL card. The DLAB statement requires (in the case of card input) two cards for completion, therefore, column 72 of the first card requires a character punch other than a blank. The disk-label is known as a FORMAT 1 disk file label. Its format is given in Appendix D.

'label fields 1-3' are defined in Appendix D.

xxxx is the volume sequence number in field 4 of the FORMAT 1 label, and must begin in card column 16.

yyddd,yyddd is the file creation date followed by the file expiration date.

'systemcode' is ignored by Disk and Tape Operating Systems but is required by Operating System. It must be 13 characters long.

[,type] indicates the type of file label

SD - sequential disk
DA - direct access
ISC - indexed sequential (used when creating a file)
ISE - indexed sequential (used when updating or retrieving a file).

XTENT Statement (Disk System only): The format of the XTENT statement is:

```
    // XTENT type,sequence,lower,upper,
             'serial no.',SYSxxx[,B2]
```

Examples of XTENT statements are given in Section III, under Examples of Processing Using Disk Configuration.

This statement is used to define an area in a direct access storage device (DASD). Each DASD statement requires one or more XTENT statements. There are three extent types. Each is identified by a code that informs the control program what the defined area is to be used for.

type — Extent Type - occupies 1 or 3 columns, containing:

1 = data area (no split cylinder)

2 = overflow area (for indexed sequential file)

4 = index area (for indexed sequential file)

128 = data area (split cylinder). If type 128 is specified, the lower head $H_1H_2H_2$ is taken from lower, and the upper head $H_1H_2H_2$ is taken from upper.

sequence

Extent Sequence Number - indicates the sequence number of this extent within a multi-extent file. The sequence number occupies 1 to 3 columns and contains a decimal number from 0 to 255. Extent sequence 0 is used for the master index of an indexed sequential file. If the master index is not used, the first extent of an indexed sequential file has sequence number 1. The extent sequence for all other types of files begins with 0.

lower

Lower Limit of Extent - occupies 9 columns and contains the lowest address of the extent in the form $B_1C_1C_1C_2C_2C_2H_1H_2H_2$ where:

B_1 is the initially assigned cell number. It is equal to:

0 for 2311
0 to 9 for 2321

C_1C_1 is the sub-cell number. It is equal to:

00 for 2311
00 to 19 for 2321

$C_2C_2C_2$ is the cylinder number. It can be:

000 to 199 for 2311

or strip number:

000 to 009 for 2321

H_1 is the head block position. It is equal to:

0 for 2311
0 to 4 for 2321

H_2H_2 is the head number. It can be:

00 to 09 for 2311
00 to 19 for 2321

A lower extent of all zeros is invalid.

Note: For 2321, the last 5 strips of sub-cell 19 are reserved for alternate tracks.

upper

Upper Limit of Extent - occupies 9 columns containing the highest address of the extent, in the same form as the lower limit.

'serial no.'

Volume Serial Number - This is a 6-byte alphameric character string, contained within apostrophies. The number is the same as in the volume label (volume serial number) and the Format 1 label (file serial number).

SYSxxx

This is the symbolic address of the DASD drive.

B_2

Currently assigned cell number. Its value is: 0 for 2311, 0-9 for 2321. This field is optional. If missing, job control assigns $B_2 = B_1$.

TPLAB Statement: The formats of the TPLAB statements are:

```
        // TPLAB 'label fields 3-10'
        // TPLAB 'label fields 3-13'
```

The statement contains file label information for tape label checking and writing, and must immediately follow the VOL statement.

TPLAB identifies the tape-label statement and can be written two ways:

1. Input labels require only one statement, and contain fields 3-10 of the standard tape file label. Refer to Appendix C for an illustration of STANDARD TAPE FILE LABELS. These are the only fields used for checking the label of an input file.

2. When writing output labels, additional fields may be included by use of a continuation statement. The additional fields are 11-13. (These fields are not required for output files.) Refer to the publication, System Control and System Service Programs, listed on the cover of this manual for details on these fields.

OPTION Statement: The format of the option statement is:

```
        // OPTION option1[,option2,...]
```

An example of this statement is:

```
        // OPTION LIST,DUMP,LINK
```

This statement specifies one or more of the job control options available. The order in which they appear in the operand field is arbitrary. A complete list of options useful to the COBOL user is listed in Appendix E.

The options specified in the OPTION statement remain in effect until a contrary option is encountered or until a JOB control statement is read. In the latter case, the options are reset to the standard that was established when the system was originally generated.

Any assignment for SYSLNK, after the occurrence of the OPTION statement, cancels the LINK and CATAL options. These two options are also canceled after each occurrence of an EXEC statement with a blank operand.

PAUSE Statement: The format for the PAUSE statement is:

```
        // PAUSE [comments]
```

An example of this statement is:

```
        // PAUSE SAVE SYS004, SYS005, MOUNT NEW TAPES
```

This statement can be used for operator action between jobs. Any messages to the operator can appear in the operand of a PAUSE statement.

In the example given, the operator is directed to save the output tapes, and mount two new tapes.

When the PAUSE statement is encountered by job control, the printer-keyboard (IBM 1052) is unlocked for operator-message input. The end-of-communication indicator, Ⓑ (Ⓑ = alter code 5), causes processing to continue.

The PAUSE statement is always printed on SYSLOG. If no 1052 is available, the PAUSE statement is ignored.

/* -- End-of-Data-File Statement: This statement must be the last statement of each input data file on SYSRDR and SYSIPT. It is used for separating job steps within a program and testing for end-of-file condition. Its format is:

```
/* ignored
```

Columns 1 and 2 contain a slash and an asterisk.
Column 3 must be blank.

/& -- End-of-Job Statement: This statement must be the last statement of each job. Its format is:

```
/& ignored
```

Columns 1 and 2 contain a slash and an ampersand (12-punch). Column 3 must be blank. If job control attempts to read past the /&, the job is terminated. If the system is equipped with the timer feature this statement can be used for comments. Comments can begin in column 14.

* -- Comments Statement: This statement is a job-control comments statement. Its format is:

```
* any user comments
```

Column 1 contains an asterisk. Column 2 is blank. The remainder of the statement (through column 72) contains any user comments. The content of the comment statement is printed on SYSLOG. If followed by a PAUSE statement, the statement can be used to request operator action.

COBOL OPTIONS (COBOL Control Card): Most options are supplied by the job-control OPTION card (see Appendix E).

The compiler has a CBL option card to provide additional flexibility. Its position in the job stream is between the EXEC COBOL statement and the COBOL IDENTIFICATION DIVISION statement. The format of this card is:

```
CBL          [DMAP = h]
             [,PMAP = h]
             [,BUFFSIZ = d]
             [,DISPCHK = {yes}]
                         {no }
             [,INVED]
```

where:

1. CBL must begin in column two, be preceded by a blank, and followed by at least one blank.

2. DMAP and PMAP set the listing (not object module) relocation factor, i.e. the addresses of both the data division map and procedure division map are incremented by the number h. If both DMAP and PMAP are specified, the value specified for the last parameter (DMAP or PMAP) is that which is used. h is a hexadecimal number of from one to eight hex digits; the assumed value is zero.

3. BUFFSIZ sets the size of each compiler buffer to the number d. d is a decimal number from 170 to 32,760; the assumed values are 170 on a 16K System/360 and 1,024 on a 32K or larger System/360. Double buffers are always used.

4. DISPCHK determines whether or not a diagnostic check is made at object time for DISPLAYed items. If YES is specified, all items are length checked before moving them to the DISPLAY buffer. If they are too long, they are truncated (no message given). If NO is specified, no length check is made and items are moved directly to the DISPLAY buffer. If they are too long, they exceed the buffer size and destroy the information in the following storage area. The assumed value is NO.

5. INVED : When the INVED option is not specified, the character "." represents a decimal point and the character "," represents an insertion character. When the INVED option is specified, the above roles of these characters ".", "," is reversed.

SECTION II. DECK STRUCTURES FOR PROCESSING COBOL PROGRAMS IN A TAPE OPERATING SYSTEM

For each type of processing, certain combinations of job control cards are needed. The examples given illustrate typical basic types of processing within an all tape system configuration.

The examples assume a given tape system configuration, and that the COBOL Tape Compiler is used for processing.

Figure 3 is a diagram of the I/O units used by COBOL in a tape configuration, and should help the user to visualize the logical structure of a configuration. A list of the types of processing discussed, in the order they are presented, follows:

1. Compile and punch

2. Cataloging to the relocatable library

3. Compiling, linkage editing, and executing

4. Executing a previously linkage edited program

5. Cataloging to the source statement library

6. Compiling, linkage editing, and executing.

Examples 3 and 6 differ in that example 3 illustrates how job control is used to link with a module cataloged to the relocatable library, while example 6 illustrates how COBOL copies source statement modules cataloged to the source statement library.

ASSUMED TAPE RESIDENT SYSTEM CONFIGURATION

The processing examples given herein assume that the following Tape Operating System was generated at system generation time:

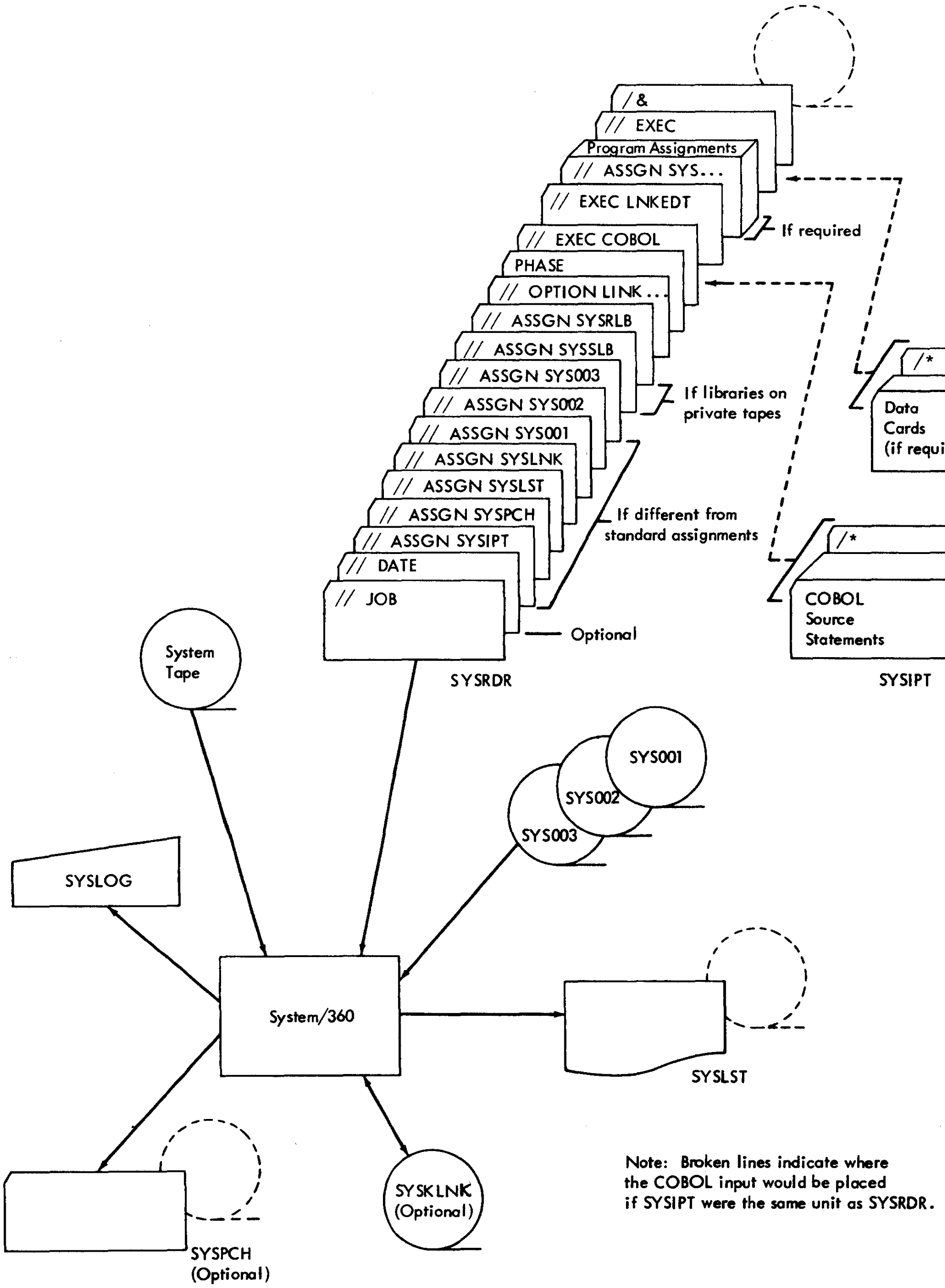
The system includes:

One IBM 1403 Printer
One IBM 2540 Card/Read/Punch
One IBM 1052 Printer-Keyboard
Four IBM Magnetic Tape Units (excluding the resident tape drive)

Assume physical assignments at system generation time are:

1403 Printer assigned to physical unit X'00E'
2540R Reader assigned to physical unit X'00C'
2540P Punch assigned to physical unit X'00D'
1052 Printer-Keyboard assigned to physical unit X'01F'
2402 Magnetic Tape Unit assigned to physical unit X'180'
2402 Magnetic Tape Unit assigned to physical unit X'181',X'90'
2402 Magnetic Tape Unit assigned to physical unit X'182'
2402 Magnetic Tape Unit assigned to physical unit X'183'
2402 Magnetic Tape Unit assigned to physical unit X'184'
(This unit is the resident tape drive.)

The hexadecimal 90 (X'90') in the tape assigned to X'181' determines the device specifications for a seven-track tape.



Note: Broken lines indicate where the COBOL input would be placed if SYSIPT were the same unit as SYSRDR.

Figure 3. I/O Units Used by COBOL Program in a Tape System

Assume logical assignments at system generation time are:

```
   // ASSGN SYSIPT,X'00C'
   // ASSGN SYSRDR,X'00C'            IBM 2540
   // ASSGN SYSPCH,X'00D'
   // ASSGN SYSLST,X'00E'            IBM 1403
   // ASSGN SYSLOG,X'01F'            IBM 1052
   // ASSGN SYSLNK,X'180'
   // ASSGN SYS001,X'181',X'90'
   // ASSGN SYS002,X'182'            IBM 2400's
   // ASSGN SYS003,X'183'
```

Notice that SYSIPT, SYSRDR, and SYSPCH are assigned to the same physical unit (they need not be), and that SYS001 is a 7-track tape. Observe also that four logical tape assignments are made. The COBOL compiler requires three logical work files to compile. The fourth can be used for compile-and-execute functions.

The user can change these assignments by the use of ASSGN cards following his JOB card. Examples of overriding assignments are given in the text that follows. In the examples that follow, whenever an optional statement is used it is identified by the words (optional card).

EXAMPLES OF PROCESSING USING TAPE CONFIGURATION

COMPILE AND PUNCH (EXAMPLE 1)

Assuming that source statements are card input (SYSIPT), and job-control statements are card input (SYSRDR), the set of job-control cards required (and some helpful options) to compile and punch are:

```
   // JOB SUBROTNE
   // OPTION LOG,DECK,LIST,LISTX,ERRS             Input from SYSRDR
   // EXEC COBOL

      SUBROTNE                                    Input from SYSIPT
      SOURCE STATEMENTS

   /*
   /&
   // PAUSE REMOVE OBJECT DECK FROM HOPPER        Input from SYSRDR
      (optional card)
```

The options selected on the option card specify:

LOG -- Requests a listing of all control statements on SYSLST.
DECK -- Requests that a deck (object module) be punched on SYSPCH.
LIST -- Causes compiler to write source statements on SYSLST.
LISTX -- Causes compiler to write a procedure division map on SYSLST in hexadecimal.
ERRS -- Causes compiler to write all diagnostics related to the source program on SYSLST.

CATALOGING AN OBJECT MODULE TO RELOCATABLE LIBRARY (EXAMPLE 2)

In this example, an object module generated by the compiler (refer to example 1) is cataloged to the relocatable library. It is assumed that the relocatable library is on SYSRES (similarly for the source statement library). Another tape drive may be used as a private library for the relocatable library, in which case the system logical unit SYSRLB is used.

The job-control cards required to catalog an object module to an existing relocatable library are:

```
// JOB RELOCATE
// EXEC MAINT
      CATALR SUBROTNE
     {Object deck to be    }
     {cataloged goes here. }
/*
/&
*             OBJECT MODULE 'SUBROTNE' IS NOW
*             CATALOGED TO NEW SYSRES TAPE ON
// PAUSE      SYS002                                  (Optional Card)
```

When an object module is cataloged to the relocatable library residing on SYSRES, the following points must be considered.

1. SYS002 is the device on which the newly updated library is located (SYSRES is now outdated).

2. If SYS002 is to be established as new SYSRES, it must be mounted on the tape drive assigned to "old" SYSRES, and initial program loaded (IPL). This automatically establishes it as "new" SYSRES. SYS002 can then be reassigned.

SYS001 is used as a work file.

COMPILE, LINKAGE EDIT, AND EXECUTE (EXAMPLE 3)

This example illustrates how an object module cataloged to the relocatable library is included in a compilation, linkage edited with the main program and executed.

The job control statements required to compile, linkage edit, and execute are:

```
// JOB CALLPROG
// OPTION LINK,LIST,LISTX,ERRS
   PHASE MAIN,*
// EXEC COBOL
  {COBOL SOURCE STATEMENTS}
/*
   INCLUDE SUBROTNE  {Retrieves SUBROTNE from relocatable library}
// EXEC LNKEDT
// EXEC
  {DATA DECK}
  { (if any)}
/*
/&
```

This program consists of one phase that includes the object module SUBROTNE and permits immediate execution of the program. (The name provided in the PHASE statement (MAIN) has no relationship to the external-name given in the COBOL Program-ID statement.)

It is possible to process this program with only three work files; however, the procedure requires special instructions to the operator for making two passes through the system. In this example, such instructions are conveyed to the operator on comment cards.

The output of the first pass (Pass 1) is a punched object deck, which is used in the second pass (Pass 2). To accomplish Pass 2, linkage edit and execute, the punched object deck must be positioned in the job stream to precede the EXEC LNKEDT and EXEC statements. (This is done when the PAUSE statement is encountered.)

The complete job stream to accomplish both Pass 1 and Pass 2 is as follows:

```
// JOB CALLPROG                                  }
// ASSGN SYS001,X'180' } Work                    }
// ASSGN SYS002,X'182' } Files                   }
// ASSGN SYS003,X'183' }                         }
// OPTION DECK,LIST,LISTX,ERRS                   } Pass 1
// EXEC COBOL                                    }
                                                 }
   [COBOL SOURCE STATEMENTS]                     }
                                                 }
/*                                               }
// ASSGN SYS001,X'180' } Assignments
// ASSGN SYS002,X'182' } For Linkage
// ASSGN SYSLNK,X'183' } Edit and Execute
// OPTION LINK
* PLACE THE OUTPUT OF SYSPCH INTO SYSRDR.
* PLACE THE INCLUDE SUBROTNE STATEMENT
* THROUGH THE /& STATEMENT, INCLUSIVE,
* (LABELED PASS 2 IN THIS EXAMPLE)
* BEHIND THE PUNCHED OBJECT DECK JUST
* PUT INTO SYSRDR.
* CONTINUE
// PAUSE
   INCLUDE
   PHASE MAIN,*
   [The punched object deck will be  ]
   [positioned here in the job stream]
/*
                   {Retrieves SUBROTNE}  }
   INCLUDE SUBROTNE{from RELOCATABLE  }  }
                   {LIBRARY           }  }
                                         }
// EXEC LNKEDT                           }
// EXEC                                  } Pass 2
  [DATA DECK]                            }
  [(if any) ]                            }
/*                                       }
/&                                       }
```

The new option card is needed to accomplish the linkage editing. The entire set of control statements, and source statements from // JOB card through /& card are submitted as one job.

Note that the SYS001, SYS002 and SYSLNK are required to execute the linkage editor.

EXECUTING A PROGRAM (EXAMPLE 4)

The job control statements required to simply execute a program, assuming it is in the core image library, are:

```
   // JOB CALLPROG
   // ASSGN SYS006,X'00C'
   // ASSGN SYS004,X'182'
   // ASSGN SYS005,X'183'
   // EXEC MAIN
     ⌈DATA⌉
     ⌊DECK⌋
   /*
   /&
   // PAUSE MESSAGE TO OPERATOR IF ANY.     (optional card)
```

The example can be used for validating data, or test runs, where many runs might be made with different sets of data decks.

CATALOGING SOURCE MODULES TO SOURCE STATEMENT LIBRARY (EXAMPLE 5)

The procedural steps, and the job control statements required to catalog two source statement routines to the source statement library follow.

It is assumed that a source statement library is on the system residence volume, SYSRES.

The job control statements are:

```
   // JOB CATLSORC

   // EXEC MAINT
      CATALS C.DATAIN
      BKEND C.DATAIN
     FILEB, DATA RECORDS ARE CAPACITOR-RECORD1,     ⎫
            INDUCTOR-RECORD1,                        ⎬     ROUTINE 1
      LABEL RECORDS ARE STANDARD, BLOCK              ⎪
      CONTAINS 12 RECORDS, RECORDING MODE IS F.      ⎭
      BKEND C.DATAIN
      CATALS C.INOUT
      BKEND C.INOUT
      BEGIN. OPEN INPUT FILEB, FILED OUTPUT FILEA.  ⎫
      DATA. READ FILEB AT END GO TO CYCLE.          ⎪
      MASTER.  READ FILED AT END GO TO LABA.        ⎬     ROUTINE 2
            GO TO PROCESS.                          ⎪
      LABA. CLOSE FILEA, FILEB, FILED, STOP RUN.    ⎭
      BKEND C.INOUT
   /*
   /&
   // PAUSE REMOVE NEW SYSRES ESTABLISHED ON X'182'.
```

The open and close routine is now cataloged to the source statement library under the name 'INOUT', and the file description under the name 'DATAIN'. Notice that DATAIN is cataloged before INOUT. This is because books to be cataloged must be in alphanumeric sequence.

The message is an interruption in the job stream to inform the operator to perform some task. In this example he is instructed to remove the tape for protection.

COMPILE (USING SOURCE STATEMENT LIBRARY), LINKAGE EDIT, AND EXECUTE (EXAMPLE 6)

This example illustrates:

1. How two previously written routines, that were cataloged in the source statement library, are utilized. In this example, the source statement library is on SYSRES.

2. How assignments can be made to process an inventory file with four tapes (not including SYSRES).

Assume an electronics firm stocks quantities of electrical components that are to be maintained at a minimum quantity level, and an input data file is used to check against a master file to determine stock item reorder points.

For the purposes of illustration, only two of its many components are treated here. They are:

CAPACITORS PART NUMBER	VALUE	QUANTITY ON HAND	REORDER POINT
C61	.010MFD	47	50
C62	.020MFD	60	50
C65	.050MFD	50	50
C121	.001MMFD	90	50
C122	.002MMFD	100	50
C125	.005MMFD	22	50

INDUCTORS PART NUMBER	VALUE	QUANTITY ON HAND	REORDER POINT
L10	.10H	18	35
L20	.20H	15	35
L40	.40H	30	35
L61	10.00MH	60	35
L62	20.00MH	70	35
L64	40.00MH	69	35

Assume further, that an input update file called "DATAIN" (example 5, ROUTINE 1) was created on tape and cataloged to the source statement library, and its records look like:

```
01 CAPACITOR-RECORD1.
   02 CAPACITOR OCCURS 6.
        03 PART-NUMBER PICTURE XXXX.
        03 VALUE1 PICTURE V999.
        03 VALUE2 PICTURE XXXX.
        03 QUANTITY-ON-HAND PICTURE IS S999.
        03 REORDER-PT PICTURE IS 99.
```

```
01 INDUCTOR-RECORD1.
   02 INDUCTOR OCCURS 6.
        03 PART-NUMBER PICTURE XXXX
        03 VALUE1 PICTURE 99V99.
        03 VALUE2 PICTURE XX.
        03 QUANTITY-ON-HAND PICTURE IS S999.
        03 REORDER-PT PICTURE IS 99.
```

Also assume a program called ORDERPT (to be compiled) was written to process these records (against the master file) to reorder parts when their respective QUANTITY-ON-HAND falls below REORDER-PT.

The following source statements portray, in skeleton form, the program ORDERPT. Included is the INPUT-OUTPUT section for the program.

```
IDENTIFICATION DIVISION.

PROGRAM-ID.'ORDERPT'.

       .
       .
ENVIRONMENT DIVISION.
       .
       .

INPUT-OUTPUT SECTION.

FILE-CONTROL.

   SELECT FILEB ASSIGN TO 'SYS004' UTILITY 2400 UNITS.
   SELECT FILEA ASSIGN TO 'SYS005' UTILITY 2400 UNITS, RESERVE NO
    ALTERNATE AREA.
   SELECT FILEC ASSIGN TO 'SYS006' UNIT-RECORD 1403.
   SELECT FILED ASSIGN TO 'SYS007' UTILITY 2400 UNITS.
```

Notice that FILEC is assigned to an IBM 1403 Printer. This enables printing out the REORDER-PT, PART NUMBER of the component, and its VALUE (in MFD or MH) when the QUANTITY-ON-HAND falls below REORDER-PT.

In order to do this, a file description or FD must be written for FILEC in the data division:

```
DATA DIVISION.

FD FILEC....

   01 REORDER.
      02 REORDER-PT PICTURE IS 99 USAGE IS DISPLAY.
      02 VALUE-OF-PART PICTURE IS ZZ.999.
      02 PART-NUMBER PICTURE IS XXXX.
      02 QUANTITY PICTURE IS 999.
```

Before printing out FILEC, the appropriate values are moved into REORDER-PT (50 or 35), VALUE-OF-PART (.999MFD or ZZ.999H) PART-NUMBER (CXXX or LXXX), and QUANTITY (999).

Specifically, four files are required to process this problem:

FILEA	Updated master file.
FILEB	Updating input file (DATAIN).
FILEC	Output print file.
FILED	Master file.

The control cards to compile, link edit, and execute the problem are:

```
// JOB INVNTORY
// OPTION LINK,LIST,DUMP
   PHASE INVNTORY,*
// EXEC COBOL

            .
            .
            .
   DATA DIVISION.          (Refer to
   FD FILEB COPY 'DATAIN'.{ Example 5
            .              (for expansion
            .
            .
   PROCEDURE DIVISION.
            .
            .
            .

START. INCLUDE 'INOUT'.    (Refer to
               .          { Example 5
               .           (for expansion
               .
PROCESS.  (Records on FILEB are processed)

/*
// LBLTYP TAPE(03)
// EXEC LNKEDT
// ASSGN SYS004,X'181',X'90' (DATAIN)
// ASSGN SYS005,X'182'       (OUTPUT FILE,NEW MASTER)
// ASSGN SYS006,X'00E'       (PRINT FILE)
// ASSGN SYS007,X'183'       (MASTER FILE)
*        MOUNT INPUT (SYS004) ON X'181',
*        OUTPUT (SYS005) ON X'182',
// PAUSE MASTER (SYS007) ON X'183'.
// VOL SYS004,SYS004
// TPLAB HDR,1,DATAIN, etc.....
// VOL SYS007,SYS007
// TPLAB HDR,1,MASTER, etc.....
// VOL SYS005,SYS005
// TPLAB HDR,1,NEWMASTER, etc.....
// EXEC
// PAUSE SAVE SYS007 ON X'183' and SYS005 ON X'182'
/*
/&
```

Note that the program that processes the files takes advantage of two previously written routines (routine 1, and routine 2) that were cataloged to the source statement library.

Note also that the VOL and TPLAB job-control statements were used to check header records and write trailer labels on input and output files.

SECTION III. DECK STRUCTURES FOR PROCESSING COBOL PROGRAMS IN A DISK OPERATING SYSTEM

For each type of processing, certain combinations of job control cards are needed. The examples given illustrate basic types of processing within a Disk Operating System.

The examples assume a given Disk Operating System configuration that includes tape, and that the COBOL disk compiler is used for processing.

Because the COBOL disk compiler permits the use of disk or tape work files, some of the examples given in this section use tape work files while others use disk work files. Figure 4 is a diagram of the I/O units used by COBOL in a disk configuration with tape, and should help the user to visualize the logical structure of such a configuration.

Preceding the types of processing discussed is a procedure for establishing labels for COBOL disk work files and SYSLNK on the Standard Label Track. A list of the types of processing discussed, in the order they are presented, follows:

1. Compile and punch

2. Cataloging to the relocatable library

3. Compiling, linkage editing, and executing

4. Executing a previously linkage edited program

5. Cataloging to the source statement library

6. Compiling, linkage editing, and executing.

Examples 3 and 6 differ in that example 3 illustrates how job control is used to link with a module cataloged to the relocatable library, while example 6 illustrates how COBOL copies source statement modules cataloged to the source statement library.

ASSUMED DISK RESIDENT SYSTEM CONFIGURATION

The processing examples given here assume that the following Disk Operating System configuration with tape was generated at system generation time for the COBOL disk compiler.

The system includes:

- One IBM 2540 Card/Read/Punch
- One IBM 1052 Printer-Keyboard
- One IBM 1403 Printer
- Two IBM 2311 Disk Drives
- Four IBM 2400 Magnetic Tape Units

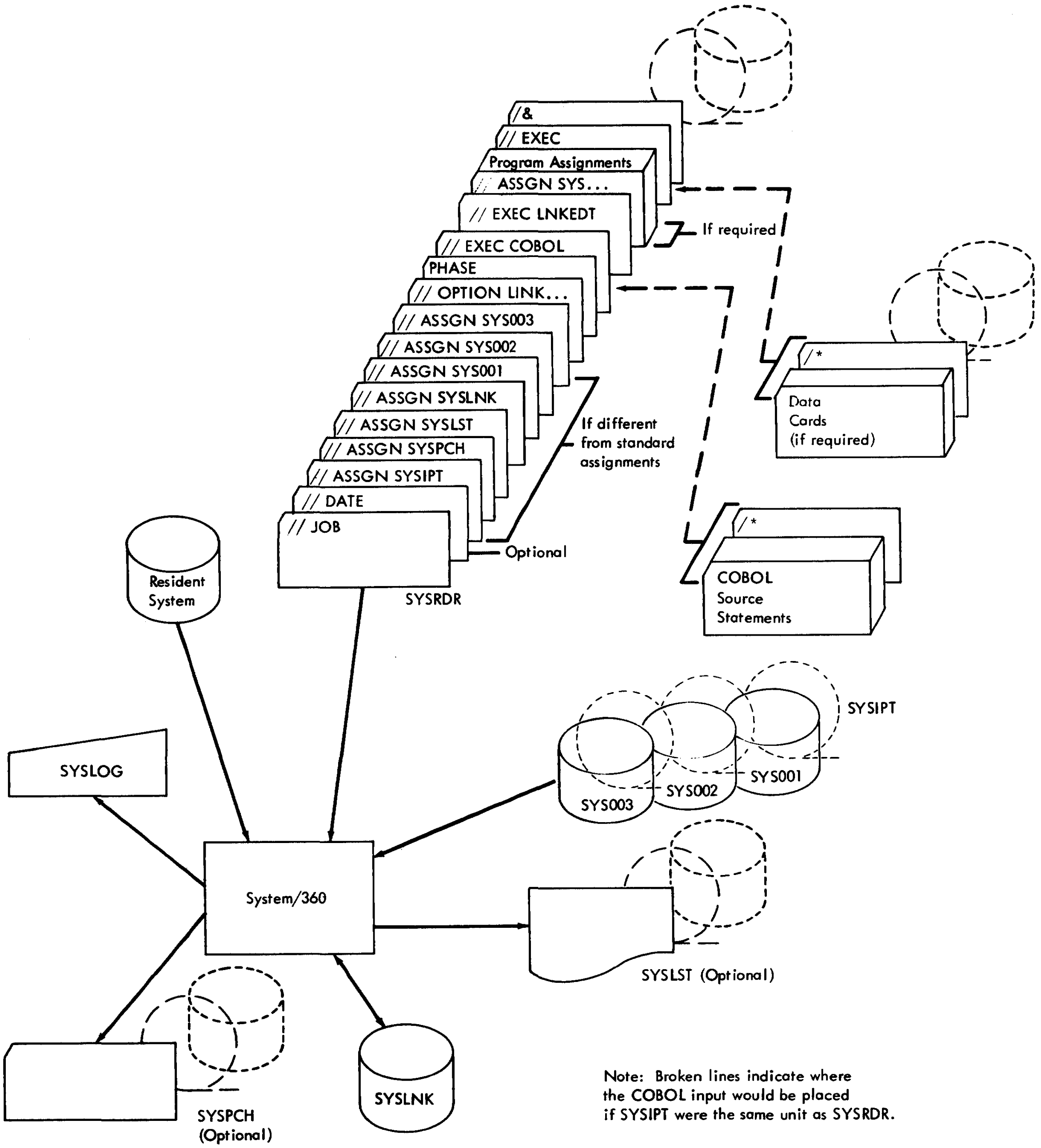


Figure 4. I/O Units Used by COBOL Program in a Disk System

Assume physical assignments at system generation time are:

- 2540R Reader assigned to physical unit X'00C'
- 2540P Punch assigned to physical unit X'00D'
- 1052 Printer-keyboard assigned to physical unit X'01F'
- 1403 Printer assigned to physical unit X'00E'
- 2311 Disk pack assigned to physical unit X'190'
- 2311 Disk pack assigned to physical unit X'191'
- 2402 Magnetic tape unit assigned to physical unit X'281'
- 2402 Magnetic tape unit assigned to physical unit X'282'
- 2402 Magnetic tape unit assigned to physical unit X'283'
- 2402 Magnetic tape unit assigned to physical unit X'284', X'90'

The hexadecimal 90 (X'90') in the last tape assignment determines the device specifications for a seven-track tape.

Assume logical assignments at system generation time are:

```
// ASSGN SYSIPT,X'00C'
// ASSGN SYSRDR,X'00C'   IBM 2540
// ASSGN SYSPCH,X'00D'
// ASSGN SYSLST,X'00E'   IBM 1403
// ASSGN SYSLOG,X'01F'   IBM 1052
// ASSGN SYSLNK,X'190'
// ASSGN SYS003,X'190'   IBM 2311's
// ASSGN SYS001,X'191'
// ASSGN SYS002,X'191'
```

When logical assignments are made at system generation time for the disk compiler, the following must be considered:

- SYSLNK must be assigned to disk.
- SYS001, SYS002, and SYS003 (work files) can be assigned to disk or tape, but must all be assigned to the same device type.
- When the linkage editor function is being performed, work file SYS001 can be assigned to either disk or tape.

When tape work files are to be used instead of the given logical assignments for disk work files (SYS001, SYS002, SYS003), the user must assign tape work files at system generation time. For example:

```
// ASSGN SYS001,X'281'
// ASSGN SYS002,X'282'
// ASSGN SYS003,X'283'
```

Notice that SYSIPT, SYSRDR and SYSPCH are assigned to the same physical unit.

The programmer can change these assignments using ASSGN cards following his JOB card. Examples of overriding assignments are given in the text that follows. In the examples that follow, whenever an optional statement is used it is identified by the words "optional card."

EXAMPLES OF PROCESSING USING DISK CONFIGURATION

When processing programs with the COBOL disk compiler, the information provided by the VOL, DLAB, and XTENT statements for the work files SYS001, SYS002 and SYS003 must be available for each job processed. This information can be supplied by the programmer with each job processed, or is provided for the programmer on the Standard Label Track for each job processed as required. In addition to establishing the labels required for the disk work files SYS001, SYS002, and SYS003, the lables required for SYSLNK can also be established on the Standard Label Track, where they will be available for subsequent use.

The following procedure enables setting up the Standard Label Track for COBOL disk compiler work files and SYSLNK. Once established, the labels remain in effect for use with subsequent jobs processed, until overridden.

```
// JOB BUILD STANDARD LABELS
*   ALL VOL, DLAB, AND XTENT STATEMENTS SUBMITTED IN THIS JOB
*   WILL BE PERMANENTLY WRITTEN ON TRACK 0 OF THE LABEL STORAGE
*   CYLINDER OF DOS SYSTEM RESIDENCE FILE SYSRES. THUS THESE
*   LABELS NEED NOT BE SUBMITTED FOR EVERY JOB THAT REQUIRES
*   SYSLNK AND SYS001-SYS003
// OPTION STDLABEL
// VOL SYS000,IJSYS00
// DLAB 'SYSTEM WORK FILE SYSLNK                  1111111',      C
               0001,66001,66001,'DISK OPER SYS',SD
// XTENT 1,0,000190000,000198009,'111111',SYSLNK
// VOL SYS001,IJSYS01
// DLAB 'SYSTEM WORK FILE NO. 1                   1111111',      C
               0001,66001,66001,'DISK OPER SYS',SD
// XTENT 128,0,000142000,000189003,'111111',SYS001
//  VOL SYS002,IJSYS02
// DLAB 'SYSTEM WORK FILE NO. 2          02.G0000V001111111',      C
               0001,66001,66001,'SYSTEM CODE 1',SD
// XTENT 128,0,000142004,000189007,'111111',SYS002
// VOL SYS003,IJSYS03
// DLAB 'SYSTEM WORK FILE NO. 3          02.G0000V001111111',      C
               0001,66001,66001,'SYSTEM CODE 1',SD
// XTENT 128,0,000142008,000189009,'111111',SYS003
```

COMPILE AND PUNCH (EXAMPLE 1)

Assuming that source statements are card input (SYSIPT), and job control statements are card input (SYSRDR), the job control cards required (and some helpful options) to compile and punch are:

```
// JOB SUBROTNE                              }
// OPTION LOG,DECK,LIST,LISTX,ERRS           } Input from SYSRDR
// EXEC COBOL                                }

  SUBROTNE                                   }
  SOURCE STATEMENTS                          } Input from SYSIPT

/*                                           }
/&                                           } Input from SYSRDR
// PAUSE REMOVE OBJECT DECK FROM HOPPER      } (Optional card)
```

The options selected specify:

LOG - Requests a listing of all control statements on SYSLST.

DECK - Requests that a deck (object module) be punched on SYSPCH.

LIST - Causes the compiler to write source statements on SYSLST.

LISTX - Causes the compiler to write a procedure division map on SYSLST in hexadecimal.

ERRS - Causes the compiler to write all diagnostics related to the source program on SYSLST.

CATALOGING AN OBJECT MODULE TO RELOCATABLE LIBRARY (EXAMPLE 2)

In this example, an object module generated by the compiler (refer to example 1) is cataloged to the relocatable library.

Note: The relocatable library is on SYSRES.

The job control cards required to catalog an object module to an existing relocatable library are:

```
// JOB RELOCATE
// EXEC MAINT
      CATALR SUBROTNE
     {Object deck to be     }
     {cataloged goes here   }
/*
/&
*          OBJECT MODULE 'SUBROTNE' IS
*          NOW CATALOGED TO THE RELOCATABLE
// PAUSE   LIBRARY ON SYSRES           (optional card)
```

COMPILE, LINKAGE EDIT, AND EXECUTE (EXAMPLE 3)

This example illustrates how an object module cataloged to the relocatable library is included in a compilation, linkage edited with the main program and executed.

```
// JOB CALLPROG
// OPTION LINK,LIST,LISTX,ERRS
   PHASE MAIN,*
// EXEC COBOL
   {COBOL SOURCE STATEMENTS}
/*
   INCLUDE SUBROTNE   {Retrieves SUBROTNE from relocatable library}
// EXEC LNKEDT
// EXEC
  {DATA DECK}
  { (if any)}
/*
/&
```

This program consists of one phase that includes the object module SUBROTNE, and can be executed immediately. (The name provided in the PHASE statement (MAIN) has no relationship to the external-name given in the COBOL Program-ID statement.)

EXECUTING A PROGRAM (EXAMPLE 4)

The job control statements required to simply execute a program, assuming it has been cataloged into the core image library, are:

```
// JOB CALL PROG
// ASSGN SYS006,X'00C'
// ASSGN SYS004,X'191'
// ASSGN SYS005,X'191'
// VOL SYS004,SYS004
// DLAB 'THIS IS THE JOB INPUT FILE etc, ...
// XTENT  Enter track specification here ...
// VOL SYS005,SYS005
// DLAB 'THIS IS THE JOB OUTPUT FILE etc, ...
// XTENT Enter track specification here ...
// EXEC MAIN
   [DATA DECK]
/*
/&
// PAUSE  MESSAGE TO OPERATOR, IF ANY (optional card)
```

The example can be used for validating data, or for test runs, where many runs might be made with different sets of data. Note that the VOL, DLAB and XTENT statements specify areas in the disk pack (assigned to X'191') that are used by the job input and output files, SYS004 and SYS005 respectively.

CATALOGING SOURCE MODULES TO SOURCE STATEMENT LIBRARY (EXAMPLE 5)

The procedural steps and the job-control statements required to catalog two source statement modules to the source statement library follow.

Note: The source statement library is on the system residence volume SYSRES.

The job control statements are:

```
// JOB CATLSORC
// EXEC MAINT
   CATALS C.INOUT
   BKEND C.INOUT
   BEGIN. OPEN INPUT FILEB, FILED OUTPUT FILEA.  ⎫
   DATA. READ FILEB AT END GO TO CYCLE.          ⎪
   MASTER. READ FILED AT END GO TO LABA.         ⎬ ROUTINE 1
         GO TO PROCESS.                          ⎪
   LABA. CLOSE FILEA, FILEB, FILED, STOP RUN.    ⎭
   BKEND C.INOUT
   CATALS C.DATAIN
   BKEND C.DATAIN
   FILEB, DATA RECORDS ARE CAPACITOR-RECORD1,    ⎫
          INDUCTOR-RECORD1,                      ⎪
   LABEL RECORDS ARE STANDARD, BLOCK             ⎬ ROUTINE 2
   CONTAINS 12 RECORDS, RECORDING MODE           ⎪
   IS F.                                         ⎭
   BKEND C.DATAIN
/*
/&
```

The open and close routine is now cataloged to the source statement library under the name 'INOUT', and the file description under the name 'DATAIN'.

COMPILE (USING SOURCE STATEMENT LIBRARY), LINKAGE EDIT, AND EXECUTE (EXAMPLE 6)

This example illustrates:

1. How two previously written routines, that were cataloged in the source statement library, are utilized. In this example, the source statement library is on SYSRES.

2. How assignments can be made to process an inventory file using disk.

Assume an electronics firm stocks quantities of electrical components that are to be maintained at a minimum quantity level, and an input data file is used to check against a master file to determine stock item reorder points.

For the purposes of illustration, only two of its many components are treated here. They are:

CAPACITORS		QUANTITY	REORDER
PART NUMBER	VALUE	ON HAND	POINT
C61	.010MFD	47	50
C62	.020MFD	60	50
C65	.050MFD	50	50
C121	.001MMFD	90	50
C122	.002MMFD	100	50
C125	.005MMFD	22	50

INDUCTORS		QUANTITY	REORDER
PART NUMBER	VALUE	ON HAND	POINT
L10	.10H	18	35
L20	.20H	15	35
L40	.40H	30	35
L61	10.00MH	60	35
L62	20.00MH	70	35
L64	40.00MH	69	35

Assume further, that an input update file called "DATAIN" (example 5, ROUTINE 2) was created on disk and cataloged to the source statement library, and its records look like:

```
01 CAPACITOR-RECORD1.
   02 CAPACITOR OCCURS 6.
        03 PART-NUMBER PICTURE XXXX.
        03 VALUE1 PICTURE V999.
        03 VALUE2 PICTURE XXXX.
        03 QUANTITY-ON-HAND PICTURE IS S999.
        03 REORDER-PT PICTURE IS 99.


01 INDUCTOR-RECORD1.
   02 INDUCTOR OCCURS 6.
        03 PART-NUMBER PICTURE XXXX.
        03 VALUE1 PICTURE 99V99.
        03 VALUE2 PICTURE XX.
        03 QUANTITY-ON-HAND PICTURE IS S999.
        03 REORDER-PT PICTURE IS 99.
```

Also assume a program called ORDERPT (to be compiled) was written to process these records (against the master file) to reorder parts when their respective QUANTITY-ON-HAND falls below REORDER-PT.

The following source statements portray in skeleton form, the program ORDERPT. Included is the INPUT-OUTPUT section for the program.

```
IDENTIFICATION DIVISION.

PROGRAM-ID.   'ORDERPT'.

      .
      .

ENVIRONMENT DIVISION.
      .
      .

INPUT-OUTPUT SECTION.

FILE-CONTROL.

    SELECT FILEB ASSIGN TO 'SYS004' UTILITY 2400 UNITS.
    SELECT FILEA ASSIGN TO 'SYS005' UTILITY 2311 UNITS, RESERVE NO
     ALTERNATE AREA.
    SELECT FILEC ASSIGN TO 'SYS006' UNIT-RECORD 1403.
    SELECT FILED ASSIGN TO 'SYS007' UTILITY 2311 UNITS.
```

Notice that FILEC is assigned to an IBM 1403 Printer. This enables printing out the REORDER-PT, PART NUMBER of the component, and its VALUE (in MFD or MH) when the QUANTITY-ON-HAND falls below REQRDER-PT.

In order to do this, a file description or FD must be written for FILEC in the data division:

```
DATA DIVISION.

FD FILEC....

    01 REORDER.
       02 REORDER-PT PICTURE IS 99 USAGE IS DISPLAY.
       02 VALUE-OF-PART PICTURE IS ZZ.999.
       02 PART-NUMBER PICTURE IS XXXX.
       02 QUANTITY PICTURE IS 999.
```

Before printing out FILEC, the appropriate values are moved into REORDER-PT (50 or 35), VALUE-OF-PART (.999 MFD or ZZ.999H) PART-NUMBER (CXXX or LXXX), and QUANTITY (999).

Specifically, four files are required to process this problem:

FILEA	Updated master file.
FILEB	Updating input file (DATAIN).
FILEC	Output print file.
FILED	Master file.

The control cards to compile, link edit, and execute the problem are:

```
// JOB INVNTORY
// OPTION LINK,LIST,DUMP
   PHASE INVNTORY,*
// EXEC COBOL

            .
            .
            .
   DATA DIVISION.          (Refer to
   FD FILEB COPY 'DATAIN'. {Example 5
            .              (for expansion
            .
            .
   PROCEDURE DIVISION.
            .
            .
            .

START. INCLUDE 'INOUT'.    (Refer to
               .           {Example 5
               .           (for expansion
               .
PROCESS.  (Records on FILEB are processed)

/*
// LBLTYP TAPE
// EXEC LNKEDT
// ASSGN SYS004,X'284',X'90'     (DATAIN)
// ASSGN SYS005,X'190'           (OUTPUT FILE, NEW MASTER)
// ASSGN SYS006,X'00E'           (PRINT FILE)
// ASSGN SYS007,X'191'           (MASTER FILE)
*          MOUNT INPUT (SYS004) ON X'284',
// PAUSE   X'90'.
// VOL SYS004,SYS004
// TPLAB HDR,1,DATAIN, etc, ....
// VOL SYS005,SYS005
// DLAB 'THIS IS THE JOB NEW MASTER FILE etc, .....
// XTENT Enter the track specification here ....
// VOL SYS007,SYS007
// DLAB 'THIS IS THE JOB(OLD) MASTER FILE etc, ....
// XTENT Enter the track specification here ...
// EXEC
/*
/&
```

Note that the program that processes the files takes advantage of two previously written routines (routines 1 and 2) that were cataloged to to the source statement library.

Note also that the LBLTYP job control statement was used (for SYS004) because it is required when label information for tape files is processed.

This section describes what the output from the computer and compiler looks like and how to use it for debugging. Included are examples of the types of output the compiler provides. Where output from the linkage editor is concerned, reference to the appropriate publication is made.

Included are:

1. Examples of compiler output and related explanations.

2. Reasons for initiating a program dump.

3. A discussion of how to use a dump (what to do when a dump occurs).

4. An explanation of diagnostic error messages (how they are determined, and how to work with them).

A complete list of the error messages (pre-compile time (debug packet), compile time and object time) are included in Appendix H. Also included in Appendix H is an illustration of object storage layout.

SOURCE LISTING (LIST)

Figure 5 is an example of a source module listing. It is obtained when LIST is specified on the job-control OPTION card. The listing is given on SYSLST.

GENERATED COBOL SOURCE LISTING (The heading appearing at the top of the listing is for explanation purposes only, and does not actually appear on the listing.)

The components of a source listing are:

1. A compiler generated line number which is shown in the left-most column. This line number is used in diagnostic and LISTX references. The generated line numbers for the sample program are 11-391).

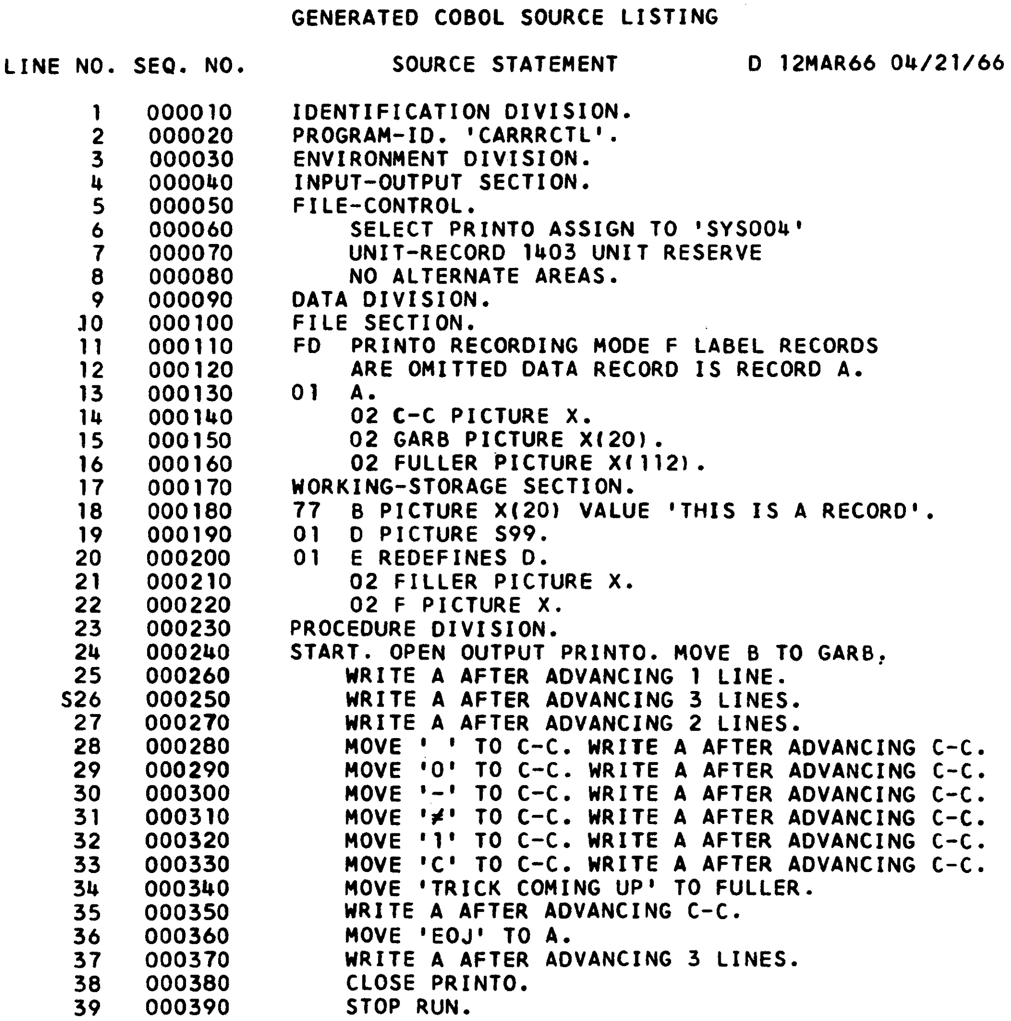
   The programmer provides the statement sequence numbers. They appear in the second column (SEQ. NO.).

2. All COBOL words and punctuation. Words, punctuation, and other groups of characters on each line are referenced as elements on the line in LISTX listings so that a specific entry may be defined.

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 beside a number not in logical ascending order. Example: Assume that in the sample listing, statement number 26 (generated line number) was out of sequence. The compiler would list the source statement as:

   ```
   S26  000250  WRITE A AFTER ADVANCING 3 LINES.
   ```

4. Debug packet card inserts. Cards inserted as part of a DEBUG packet are identified with the character "D" alongside the generated sequence number.



```
                      GENERATED COBOL SOURCE LISTING

LINE NO. SEQ. NO.           SOURCE STATEMENT        D 12MAR66 04/21/66

      1   000010     IDENTIFICATION DIVISION.
      2   000020     PROGRAM-ID. 'CARRRCTL'.
      3   000030     ENVIRONMENT DIVISION.
      4   000040     INPUT-OUTPUT SECTION.
      5   000050     FILE-CONTROL.
      6   000060         SELECT PRINTO ASSIGN TO 'SYS004'
      7   000070         UNIT-RECORD 1403 UNIT RESERVE
      8   000080         NO ALTERNATE AREAS.
      9   000090     DATA DIVISION.
     10   000100     FILE SECTION.
     11   000110     FD  PRINTO RECORDING MODE F LABEL RECORDS
     12   000120         ARE OMITTED DATA RECORD IS RECORD A.
     13   000130     01  A.
     14   000140         02 C-C PICTURE X.
     15   000150         02 GARB PICTURE X(20).
     16   000160         02 FULLER PICTURE X(112).
     17   000170     WORKING-STORAGE SECTION.
     18   000180     77  B PICTURE X(20) VALUE 'THIS IS A RECORD'.
     19   000190     01  D PICTURE S99.
     20   000200     01  E REDEFINES D.
     21   000210         02 FILLER PICTURE X.
     22   000220         02 F PICTURE X.
     23   000230     PROCEDURE DIVISION.
     24   000240     START. OPEN OUTPUT PRINTO. MOVE B TO GARB.
     25   000260         WRITE A AFTER ADVANCING 1 LINE.
    S26   000250         WRITE A AFTER ADVANCING 3 LINES.
     27   000270         WRITE A AFTER ADVANCING 2 LINES.
     28   000280         MOVE ' ' TO C-C. WRITE A AFTER ADVANCING C-C.
     29   000290         MOVE '0' TO C-C. WRITE A AFTER ADVANCING C-C.
     30   000300         MOVE '-' TO C-C. WRITE A AFTER ADVANCING C-C.
     31   000310         MOVE '≠' TO C-C. WRITE A AFTER ADVANCING C-C.
     32   000320         MOVE '1' TO C-C. WRITE A AFTER ADVANCING C-C.
     33   000330         MOVE 'C' TO C-C. WRITE A AFTER ADVANCING C-C.
     34   000340         MOVE 'TRICK COMING UP' TO FULLER.
     35   000350         WRITE A AFTER ADVANCING C-C.
     36   000360         MOVE 'EOJ' TO A.
     37   000370         WRITE A AFTER ADVANCING 3 LINES.
     38   000380         CLOSE PRINTO.
     39   000390         STOP RUN.
```

Figure 5. Example of a COBOL Source Listing

5. Library cards. Cards coming from the library as a result of a COPY or INCLUDE statement are noted with an asterisk.

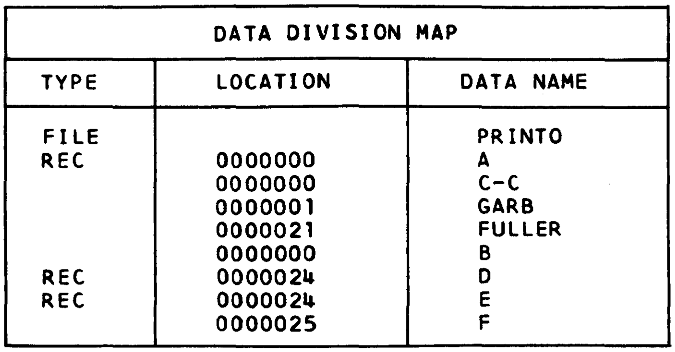
DATA MAP (SYM)

Figure 6 is an example of a data map. It is a portion of the data map generated for the program given in Figure 5, and is obtained when SYM is specified on the job-control OPTION card. The data map is output by SYSLST.

This optional listing shows the name of each non-procedure name defined in the program. File-names, record-names and condition names are identified in the column headed TYPE. (In this particular example no condition names were used, therefore, none are listed.) 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 0. The addresses given are 24-bit addresses.

The column headed DATA NAME gives the names of the non-procedure name specified in the program.

The working-storage addresses may be offset by CBL option card parameter. If the load address is known, it may be used as the hexadecimal offset parameter in a CBL option card parameter (DMAP = h). This would result in adjusted addresses on the listing.



DATA DIVISION MAP

TYPE	LOCATION	DATA NAME
FILE		PRINTO
REC	0000000	A
	0000000	C-C
	0000001	GARB
	0000021	FULLER
	0000000	B
REC	0000024	D
REC	0000024	E
	0000025	F

Figure 6. Example of a Data Map Generated for a COBOL Program

PROCEDURE MAP (LISTX)

Figure 7 is an example of a procedure map. It is a portion of the procedure map generated for the program given in Figure 5, and is obtained when LISTX is specified on the job-control option card. The listing is obtained on SYSLST. The details of LISTX are given for their debugging value.

LINE/POS - Contains the generated line number and the 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. A look at source statement 28 shows that MOVE is the first COBOL verb on the line, hence, its location is 28 01. Counting from left to right, each element on the line (for definition of an element, refer to the discussion, Error Messages (ERRS)) it is found that the COBOL verb "WRITE" occupies position 6 on the line, hence, it is location 28 06. The MOVE verb required only one System/360 machine instruction to effect its action. However, the WRITE verb required five System/360 machine instructions to effect its action. This accounts for "28 06" appearing five times in the listing. It should be noted that qualified words count as one element. The line counter cannot exceed 4095. At this point it resets to 0.

ADDR - Contains the relative address of each instruction in the procedure division in hexadecimal. The addresses are relative to the program's load point. The address may be offset as described for the data map.

INSTRUCTION - Contains the actual instruction (in hexadecimal) generated for the COBOL statement.

LINE/POS	ADDR	INSTRUCTION
00028 01	003270	D2 00 5 000 4 14D
00028 06	003276	D2 00 5 000 5 000
00028 06	003270	41 10 4 088
00028 06	003280	58 F0 1 010
00028 06	003284	45 E0 F 00C
00028 06	003288	58 50 4 088
00029 01	00328C	D2 00 5 000 4 14E
00029 06	003292	D2 00 5 000 5 000
00029 06	003298	41 10 4 088
00029 06	00329C	58 F0 1 010
00029 06	0032A0	45 E0 F 00C
00029 06	0032A4	58 50 4 088

Figure 7. Example of a Procedure Map for a COBOL Program

ERROR MESSAGES (ERRS)

Figure 8 is an example of a list of error messages that are obtained when ERRS is specified on the job-control option card. These diagnostics were generated by the compiler for the program shown in Figure 5. The list is generated on SYSLST.

LINE/POS - Contains the internal line numbers of the source statements, and the position of the COBOL verb or element on the line where the error was detected. An element is a word, punctuation, picture, name, literal, or any other similar unit of COBOL syntax.

When the compiler cannot locate the item in error on the line, it identifies the line at fault by generating the SEQUENCE NUMBER X-0.

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 H.

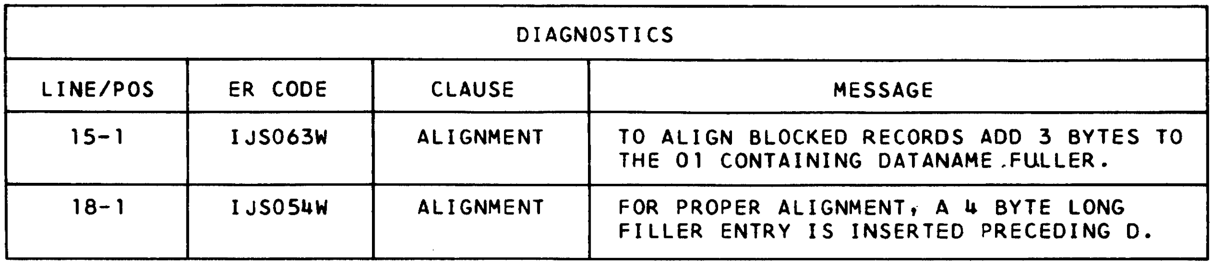
Severity Code,

W = WARNING - This calls attention 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 statement should not execute as intended.

E = ERROR - This condition seriously affects execution of the job. Execution is not 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.



DIAGNOSTICS			
LINE/POS	ER CODE	CLAUSE	MESSAGE
15-1	IJS063W	ALIGNMENT	TO ALIGN BLOCKED RECORDS ADD 3 BYTES TO THE 01 CONTAINING DATANAME .FULLER.
18-1	IJS054W	ALIGNMENT	FOR PROPER ALIGNMENT, A 4 BYTE LONG FILLER ENTRY IS INSERTED PRECEDING D.

Figure 8. Example of Source Module Diagnostics

MESSAGE - The actual message is given here. For specific details of these messages, refer to Appendix H.

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 missing 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.

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.

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 $\underline{\text{MOVE}} \left\{ \begin{array}{l} \text{data-name} \\ \text{literal} \end{array} \right\} \underline{\text{TO}}$ data-name ...

```
Error 1              MOVE FIELDA TOO FIELDB
023
                     ERROR #178

                     INSERT1 TO       Information
                                      passed to
                     INSERT2 TOO      diagnostic
                                      out of phase
```

Skeleton Message #178 C SYNTAX REQUIRES WORD "Insert1". FOUND "Insert2".

Message appears as: 23-3 IJS178I C SYNTAX REQUIRES WORD "TO". FOUND "TOO".

Example 2:

```
Error 2
023                  NOVE FIELDA TO FIELDB

                     ERROR #549

                     INSERT1 NOVE
```

Skeleton Message #549 E WORD INSERT1 WAS EITHER INVALID OR SKIPPED DUE TO ANOTHER DIAGNOSTIC.

Message appears as: 23-1 IJS549E "NOVE" UNHANDLED. WORD NOVE WAS EITHER INVALID OR SKIPPED DUE TO ANOTHER DIAGNOSTIC.

LINKAGE EDITOR OUTPUT

The linkage editor also produces error diagnostic messages and console messages. For a description of output and error messages from the Linkage Editor see the IBM publication, System Control and System Service, referenced on the cover of this manual.

OBJECT TIME MESSAGES

When an error condition that is recognized by compiler generated code occurs during execution, an error message is written on SYSLST or SYSLOG. Any messages normally written on SYSLST that result from an error in the foreground program are written on SYS000. Messages normally appearing on SYSLOG are provided with a code indicating whether the message originated in a foreground or background program. These messages and their descriptions are contained in Appendix H.

OBJECT PROGRAM DUMPS

An object program may dump as part of an abort procedure. A dump is caused by one of many errors. Several of these errors may 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 may cause an interrupt and a dump.

3. Invalid data in a numeric field resulting from redefinition.

4. Input/output errors that are nonrecoverable.

5. Moving data fields into the procedure division could destroy a machine instruction in the program. This could happen, for example, when using a subscript whose value exceeds its defined maximum value.

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 e.g., a CALL using a procedure-name and ENTRY using a data name.

10. Incorrect tape record length. Causes the compiler to generate an invalid supervisor call SVC32. This initiates the dump terminating the job.

11. 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.

HOW TO USE A DUMP

Information regarding the location of the error and the reason for an interrupt precedes the dump.

The instruction address can be compared to the procedure division map. Such a map is produced in the listing by the LISTX option. The load address of the module (can be obtained from the map of main storage generated by the linkage editor) must be subtracted from the instruction address to obtain the relative instruction address as shown in the procedure map. The contents of LISTX provides a relative address for each statement. By use of the error address and LISTX, 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. A more involved analysis would involve a deeper knowledge of Disk and Tape Operating Systems and control programs.

OBJECT STORAGE LAYOUT

The relative position, in main storage, of all the components of a COBOL program is illustrated in the object storage layout given in Appendix H.

SECTION V. THE DEBUGGING LANGUAGE

DEBUGGING TECHNIQUES.

The DEBUG option in the COBOL Disk and Tape Operating Systems language allows the programmer to use three new verbs for the purpose of debugging COBOL source programs. These verbs are EXHIBIT, TRACE, and ON. They can appear anywhere in the disk and tape COBOL program or in a compile-time debugging packet. The formats used for them and a description of their use is contained in the IBM publication, COBOL Language Specifications, listed on the cover of this manual. However, this section is included in the publication to give the programmer an idea of when to use the debugging language, how to construct a debugging packet, and what job control cards are needed to use the debugging packet. A complete list of precompile error messages is included in Appendix H. These messages reflect errors in the debug packet(s) only. They are not associated with compiling.

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 it 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 may be restricted to display the field only if it has changed since the last time the program fell 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 printout 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 compiler counter generated for each one. The counter starts at 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 can be used to count records. Then a READY TRACE can be set as the counter approaches the point where the error occurred. This eliminates tracing each statement up to that point.

Note: 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 can only be used in background type processing. It is a tool used for debugging COBOL object modules, and 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.

A number of debugging packets are permitted for a program depending on the size of the machine used. In practice, the number of packets required by a programmer should not exceed Disk and Tape Operating Systems storage facilities.

Each compile-time debugging packet is headed by the control card:

```
1      8
*DEBUG location
```

An example of the deck setup for executing a debugging packet, including all the required job control cards is given in Figure 9.

Note that the deck setup provides for the assignment of SYSIPT (for the COBOL compilation) to the drive currently assigned to SYS004 for the packet. This must be assigned to a tape unit. This is required by job control, because SYSIPT (card reader or magnetic tape unit) is used as the input for the COBOL program.

If a disastrous error occurs, a message followed by "RUN TERMINATED" is displayed and listed. If the job runs to completion, a message saying that SYSIPT for the COBOL compilation should be assigned to the current SYS004 is displayed and listed.

At the conclusion of a compilation, SYSIPT should be reassigned to the original device if the job stream contains additional job steps.

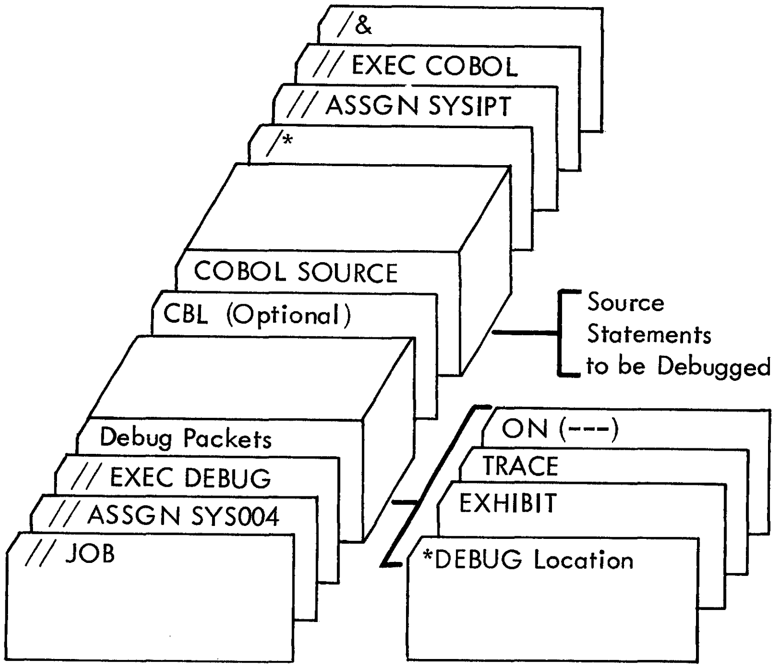


Figure 9. Example of a Debugging Packet

SECTION VI. USE OF LIBRARIES AT COBOL LANGUAGE LEVEL

SOURCE STATEMENT LIBRARY

Incorporated in the COBOL language are facilities for utilizing the source statement library.

Prewritten source program entries in the source statement 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 clause (in the case of the data division) or an INCLUDE clause (in the case of the procedure division). Examples of cataloging are given in Sections II and III.

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, which illustrates what actually gets copied when the COBOL COPY clause is written, follows:

Assume a book called 'CFILEA' is in the source statement library, and the source module statements written to have made this entry were:

```
// JOB ANYNAME
// EXEC MAINT
   CATALS C.CFILEA
   BKEND C.CFILEA
      BLOCK CONTAINS 13 RECORDS
      RECORD CONTAINS 120 CHARACTERS.
   BKEND C.CFILEA
/*
/&
```

Note: This will be 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.

To retrieve the cataloged entry 'CFILEA' from the source statement library, the source COBOL statement written is:

```
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 library is followed by the actual library entries, except for data entries which have a duplicate level number and dataname. 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 dataname, the following results are obtained when issuing a COBOL COPY statement.

Assume the statements written to catalog a file are:

```
// JOB ANYNAME
// EXEC MAINT
   CATALS C.XFILEY
   BKEND C.XFILEY
      01 PAYFILE USAGE IS DISPLAY.
            02 CALC PICTURE 99.
            02 GRADE PICTURE 9
                 OCCURS 1 DEPENDING ON CALC OF PAYFILE.
   BKEND C.XFILEY
/*
/&
```

and, the source statement is written:

```
   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 GROSS.
```

INCLUDE (Procedure Division)

The procedure for copying from the source statement library from within the procedure division is the same as that described for the data division. The results are identical.

Assume a book named PROCESS is in the source statement library, and was cataloged as follows:

```
// JOB ANYNAME
// EXEC MAINT
   CATALS C.PROCESS
   BKEND C.PROCESS
      COMPUTE QTY-ON-HAND = TOTAL-USED-NUMBER-ON-HAND.
   BKEND C.PROCESS
/*
/&
```

To retrieve catalog entry PROCESS, write:

```
Paragraph-name.  INCLUDE 'PROCESS'.
```

It is the user's responsibility to supply the name for paragraph-name.

RELOCATABLE LIBRARY

Linkage editor must be used to retrieve object modules from the relocatable library. Before execution, an object module must be linkage edited. It can be linkage edited with:

- I/O modules
- COBOL subroutine modules
- Subprogram modules
- Main program modules (where this module is a subprogram)

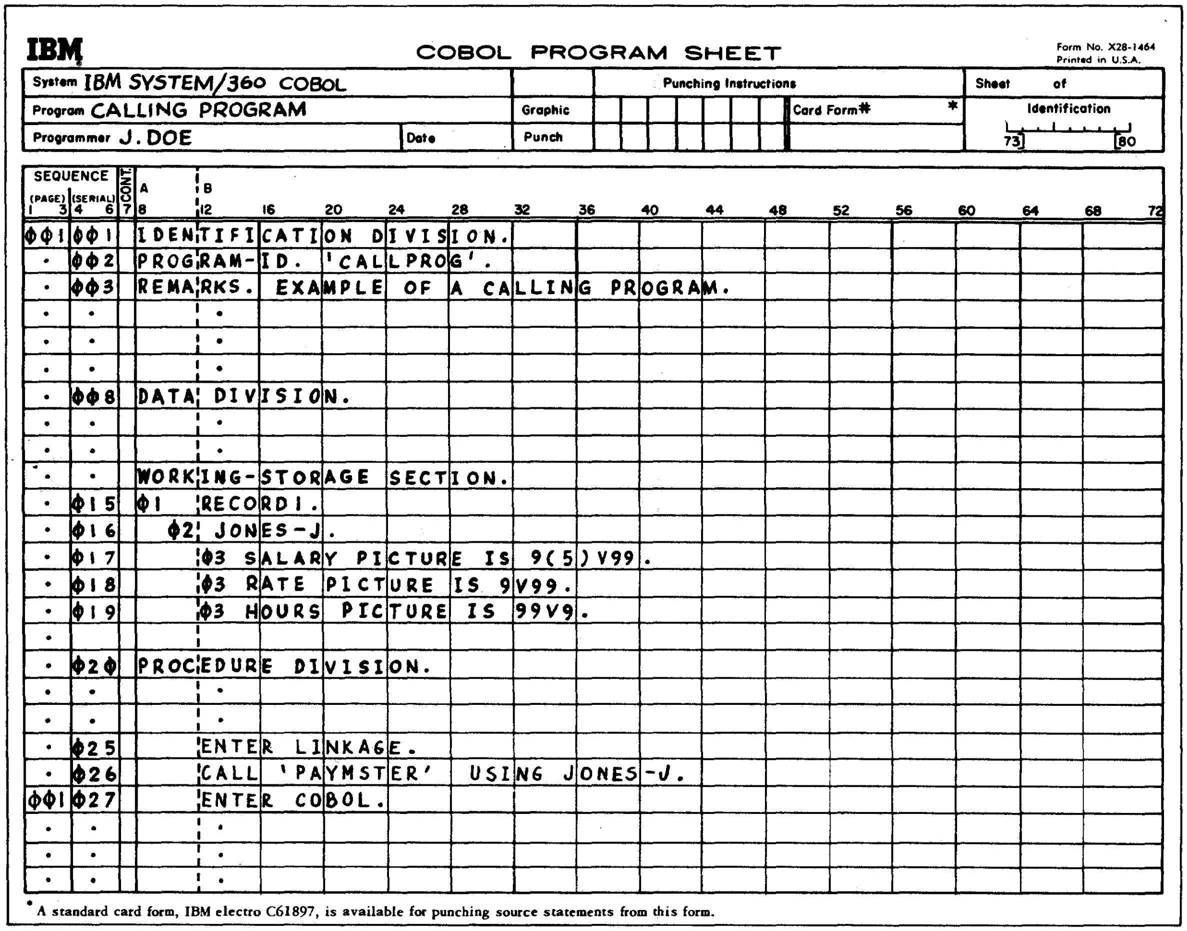
These modules can be located in the relocatable library, from which one can be retrieved and combined with the object module by the linkage editor at linkage edit time.

LINKAGE EDITOR

The output of a COBOL compilation is an object module. Before the program can be executed it must be altered to a form acceptable for execution. The linkage editor edits the object module and produces a program phase. The structure of a program phase makes it suitable for execution. The COBOL program itself is produced as one control section. However, there may be external references, such as entry points to subroutines or subprograms to be resolved. The subroutines that the COBOL compiler calls for in the object program e.g., for conversion from COMPUTATIONAL to COMPUTATIONAL-3, are obtained from the relocatable library. The subprograms that a user CALLS in his COBOL source program can be obtained from SYSIPT or from the relocatable library.

CALLING A SUBPROGRAM

Figure 10 illustrates how a subprogram is called and what data definitions are required to support the CALL.



IBM COBOL PROGRAM SHEET

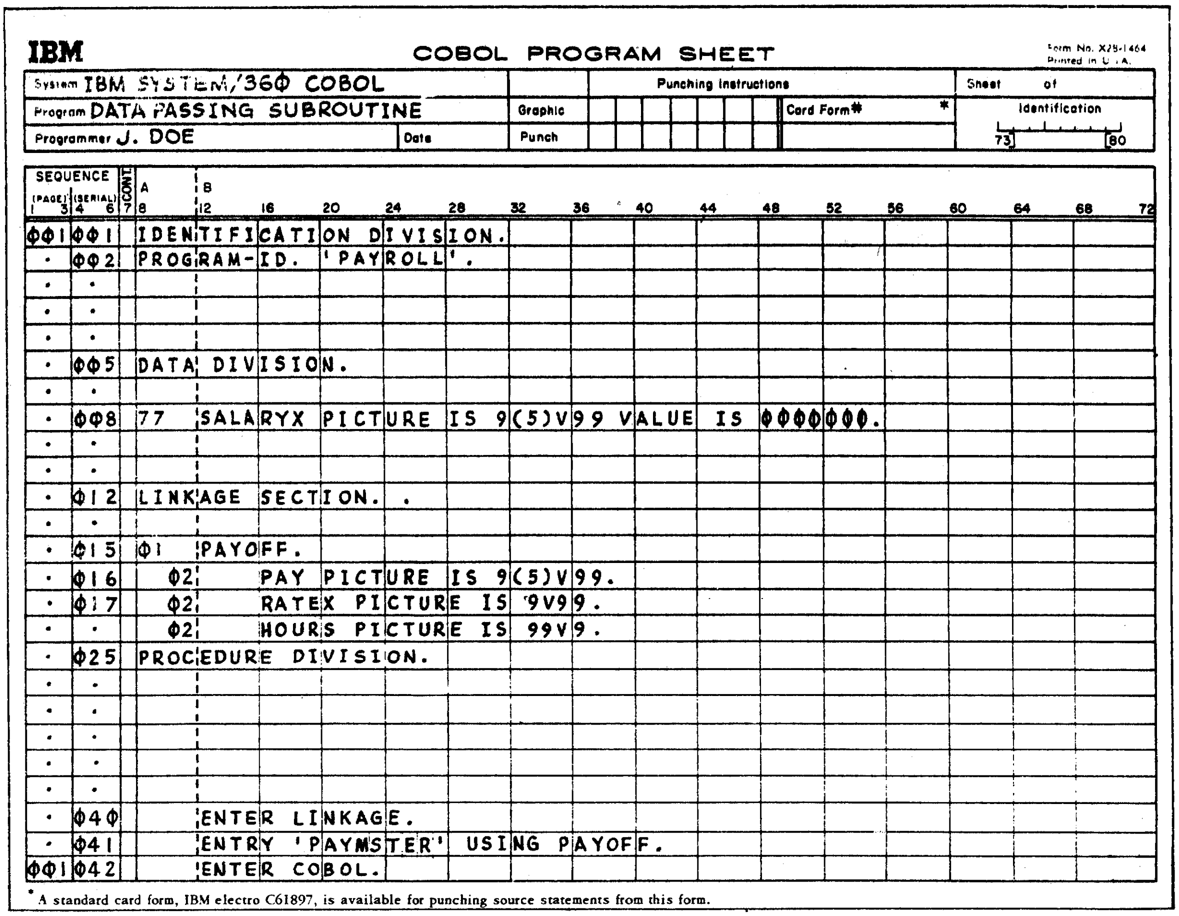
Form No. X28-1464
Printed in U.S.A.

System IBM SYSTEM/360 COBOL		Punching Instructions	Sheet of
Program CALLING PROGRAM	Graphic	Card Form# *	Identification
Programmer J.DOE Date	Punch		73 80

```
SEQUENCE  A   B
(PAGE)(SERIAL)
ΦΦ1ΦΦ1 IDENTIFICATION DIVISION.
   ΦΦ2 PROGRAM-ID. 'CALLPROG'.
   ΦΦ3 REMARKS. EXAMPLE OF A CALLING PROGRAM.
    .          .
    .          .
    .          .
   ΦΦ8 DATA DIVISION.
    .          .
    .          .
    .  WORKING-STORAGE SECTION.
   Φ15 Φ1  RECORD1.
   Φ16     Φ2 JONES-J.
   Φ17         Φ3 SALARY PICTURE IS 9(5)V99.
   Φ18         Φ3 RATE PICTURE IS 9V99.
   Φ19         Φ3 HOURS PICTURE IS 99V9.

   Φ2Φ PROCEDURE DIVISION.
    .          .
    .          .
   Φ25         ENTER LINKAGE.
   Φ26         CALL 'PAYMSTER' USING JONES-J.
ΦΦ1Φ27         ENTER COBOL.
    .          .
    .          .
    .          .
```

* A standard card form, IBM electro C61897, is available for punching source statements from this form.

Figure 10. Example of a Calling Subprogram (Part 1 of 3)



IBM COBOL PROGRAM SHEET

Form No. X28-1464
Printed in U.S.A.

System IBM SYSTEM/36Φ COBOL
Program DATA PASSING SUBROUTINE
Programmer J. DOE
Date
Punching Instructions
Graphic
Punch
Card Form# *
Sheet of
Identification 73 80

```
SEQUENCE    A   B
(PAGE)(SERIAL) CONT.
1  3 4  6 7 8   12  16  20  24  28  32  36  40  44  48  52  56  60  64  68  72
ΦΦ1ΦΦ1 IDENTIFICATION DIVISION.
 . ΦΦ2 PROGRAM-ID. 'PAYROLL'.
 .  .
 .  .
 .  .
 . ΦΦ5 DATA DIVISION.
 .  .
 . ΦΦ8 77  SALARYX PICTURE IS 9(5)V99 VALUE IS ΦΦΦΦΦΦΦ.
 .  .
 .  .
 . Φ12 LINKAGE SECTION. .
 .  .
 . Φ15 Φ1  PAYOFF.
 . Φ16   Φ2    PAY PICTURE IS 9(5)V99.
 . Φ17   Φ2    RATEX PICTURE IS 9V99.
 .  .    Φ2    HOURS PICTURE IS 99V9.
 . Φ25 PROCEDURE DIVISION.
 .  .
 .  .
 .  .
 .  .
 .  .
 . Φ4Φ     ENTER LINKAGE.
 . Φ41     ENTRY 'PAYMSTER' USING PAYOFF.
ΦΦ1Φ42     ENTER COBOL.
```

*A standard card form, IBM electro C61897, is available for punching source statements from this form.

Figure 10. Example of a Calling Subprogram (Part 2 of 3)

The calling program 'CALLPROG' calls the subprogram 'PAYROLL', handing 'PAYMSTER' the address of the group item, JONES-J. The elementary data items subordinate to JONES-J i.e., SALARY, RATE, HOURS can be operated on by 'PAYMSTER' through its using statement parameter, PAYOFF.

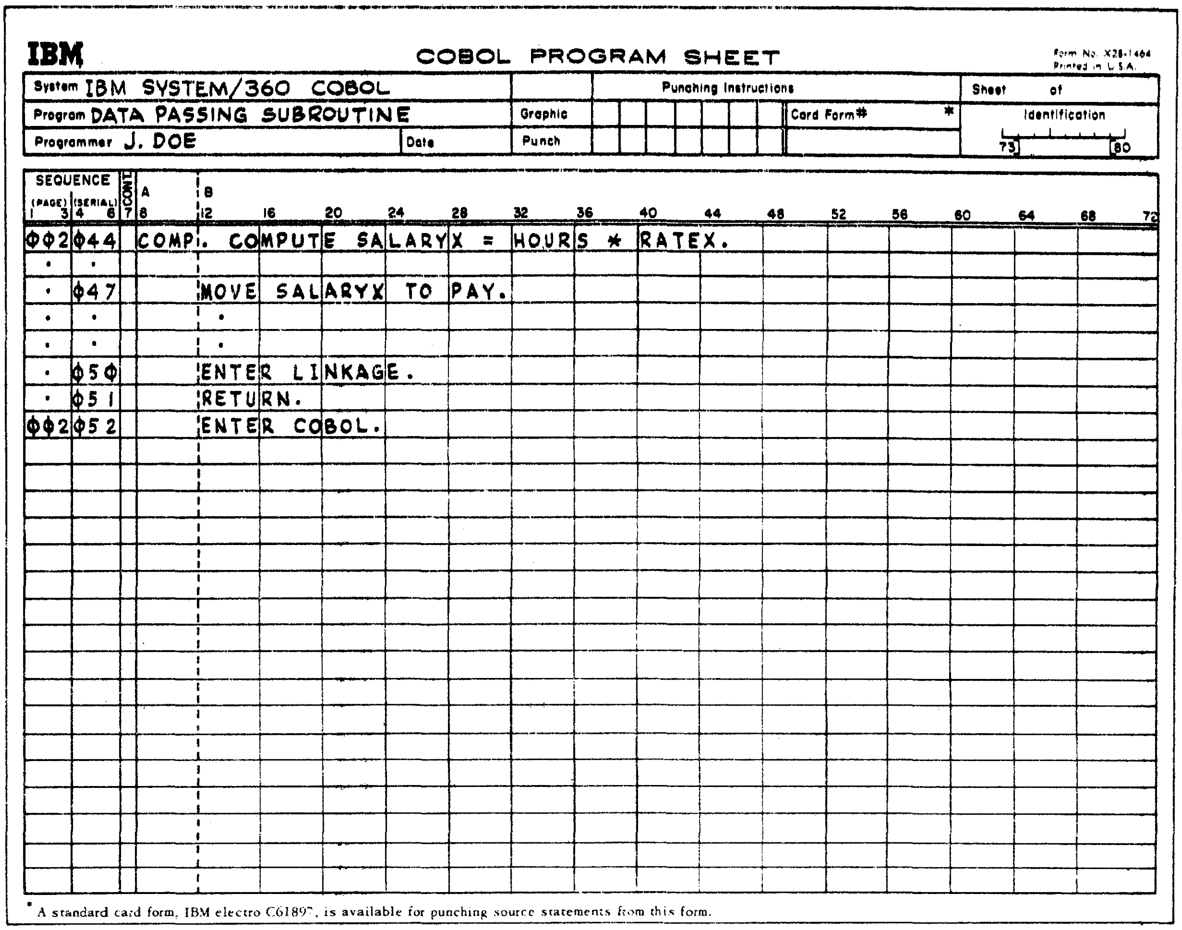
In effect, JONES-J is "equated" to PAYOFF. Any operation performed on data items subordinate to PAYOFF are really done to those items subordinate to JONES-J.

Be sure the PICTURE descriptions of equated data items are identical. There is no necessary relationship between the names of data items between calling programs and subprograms.

Notice that the PICTURES for the data items under JONES-J and PAYOFF are identical. This is good technique.

It is not required, necessarily, that the number of characters in a PICTURE of a data item be identical to its related data item, but it is required that the number of characters per record description be equal to its associated record. (The record in the calling program and the record in the called subprogram.)

Note: The entry-name (ENTRY 'PAYMSTER') must not be the same as the Program-ID ('PAYROLL').



IBM COBOL PROGRAM SHEET

Form No. X28-1464
Printed in U.S.A.

System IBM SYSTEM/360 COBOL		Punching Instructions	Sheet of
Program DATA PASSING SUBROUTINE	Graphic	Card Form # *	Identification 73 80
Programmer J. DOE Date	Punch		

```
SEQUENCE  A   B
(PAGE)(SERIAL)
ØØ2Ø44 COMP. COMPUTE SALARYX = HOURS * RATEX.
   .  .
   . Ø47     MOVE SALARYX TO PAY.
   .  .      .
   .  .      .
   . Ø5Ø     ENTER LINKAGE.
   . Ø51     RETURN.
ØØ2Ø52     ENTER COBOL.
```

* A standard card form, IBM electro C61897, is available for punching source statements from this form.

Figure 10. Example of a Calling Subprogram (Part 3 of 3)

ACCESSING CALL PARAMETERS

When a call is issued, the address of JONES-J is passed forward to a special table generated by COBOL and reserved for USING statement parameters. (The user need not declare storage for this table because it is taken care of by COBOL.) Control is then transferred to the subprogram (entry point) which accesses a special register that contains the address of the generated table. The table contains all the addresses of USING statement items declared by the calling program (JONES-J). The subprogram, having obtained the address of the parameter table, can operate on any parameter (JONES-J) in the table. (Therefore, it can operate on any item subordinate to a parameter.)

Any procedural statements referencing using parameters written in the subprogram actually operate on the data items declared in the calling program as though they (data items) were located within their own data division. The subprogram makes a salary computation: COMP. COMPUTE SALARYX = HOURS * RATEX and moves SALARYX into the elementary item called PAY. Since JONES-J is equated to PAYOFF; SALARY (under JONES-J) is equated to PAY, and SALARY (under JONES-J) contains the result of the computation COMP.

As illustrated, procedures previously written as subprograms can be used by employing the calling statement. This eliminates the need for repeated coding of frequently used procedures.

A programmer may want to prepare subprograms written in assembler language for use with COBOL programs. For a description of the conventions used in System/360 for preparing and using assembler language subprograms with COBOL, refer to Appendix A.

RESTRICTIONS OF THE USING STATEMENT

The maximum number of parameters permitted in a USING statement is 40. The total number of distinct paragraph names used in all the USING statements in the entire program is limited to 90. There is no upper limit to the number of data-names and file-names used throughout the program although 40 parameters per USING statement also applies to data name. Exceeding the limits specified causes diagnostics.

OVERLAY STRUCTURES

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 the overlay facility. 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.

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: SUBPRGA, SUBPRGB, SUBPRGC, SUBPRGD, and SUBPRGE. Also assume that the module sizes for the main program and the subprograms given are:

PROGRAM	MODULE SIZE (IN BYTES)
COBMAIN	20,000
SUBPRGA	4,000
SUBPRGB	5,000
SUBPRGC	6,000
SUBPRGD	3,000
SUBPRGE	4,000

Through the linkage mechanism, ENTER LINKAGE, CALL SUBPRGA..., 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 11 illustrates the storage layout for nonoverlay processing.

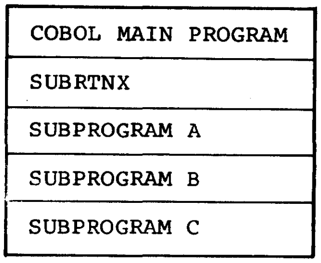


Figure 11. Storage Layout for Nonoverlay

LINKAGE EDITING WITHOUT OVERLAY

Following is an example of the job control cards needed for the COBOL call structure without overlay. In this example all the subprograms (including the main program NOVERLAY) fit into main storage.

Figure 12 portrays the flow of data as a result of the call structure.

```
// JOB NOVERLAY
// OPTION LINK,LIST,DUMP
   ACTION MAP
   PHASE EXAMP1,*
   INCLUDE

   {Object Module A }

/*
   INCLUDE SUBRTNC
   INCLUDE SUBRTND
   INCLUDE

   { SUBRTN    }
   { OBJMOD  B }
```

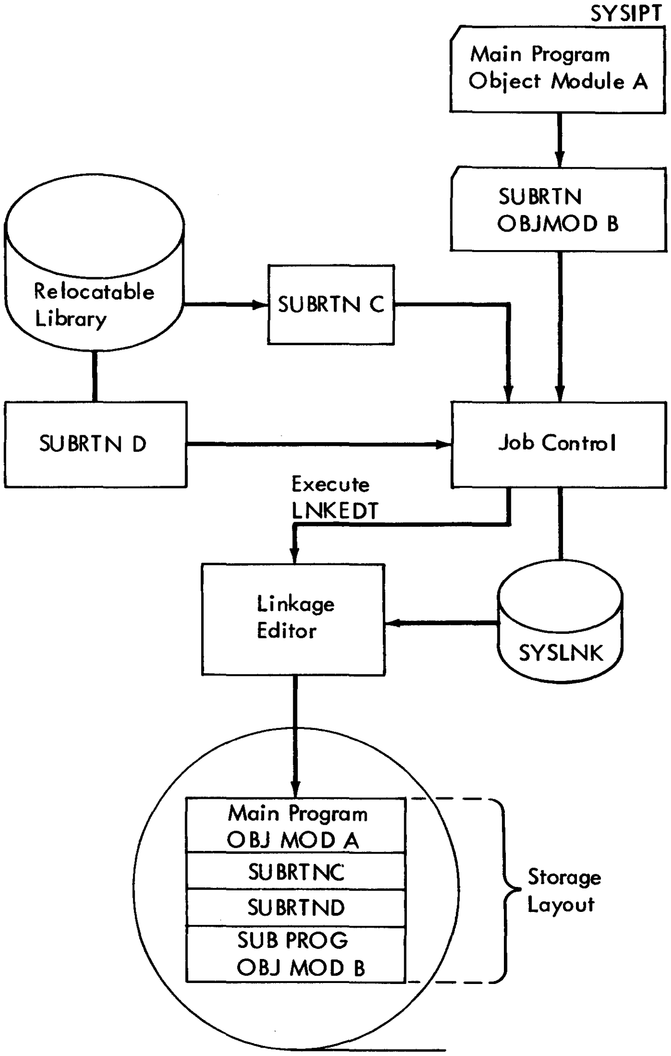


Figure 12. Example of Data Flow Logic in a Call Structure

```
/*
   ENTRY
// EXEC LNKEDT
// EXEC

      {DATA FOR PROGRAM}

/*
/&
```

Note: For the example given, it is assumed that SYSLNK is a standard assignment. The flow diagram illustrates how the various program seqments are linkage edited into storage in a sequential arrangement.

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 13 illustrates storage layout for overlay processing.

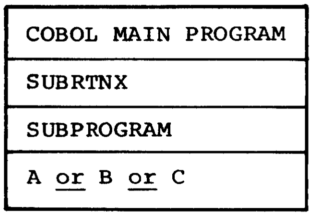


Figure 13. Storage Layout for Overlay Processing

Linkage Editing with Overlay

The linkage editor facility permits the reuse of storage locatic.ıs already occupied. By judiciously segmenting a program, and using the 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 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 an assembler language subroutine (such as the assembler language subroutine OVRLAY given in Appendix A). 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 System Control and System Service publication listed on the cover of this manual.

In a linkage editor run, the programmer specifies the overlay points in a program by using PHASE 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. An example is given to illustrate how the overlay facility is used.

The intent of the overlay example given is merely to show how an overlay structure is coded; therefore, no processing of the related parameters are illustrated.

In order to process the parameters in the respective subprograms, the USING parameters specified therein must be appropriately defined in the pertinent working storage sections. Procedural statements can then be written in each subprogram for the respective parameters.

It is the programmer's responsibility to write the entire overlay procedure i.e., the COBOL main (or calling) program, an assembler language subroutine that fetches and overlays the subprograms desired, and the overlay subprograms themselves. The linkage conventions for using assembler language subroutines with COBOL subprograms are given in Appendix A. A calling sequence to obtain an overlay structure between two COBOL programs follows.

COBOL Program Main (Root or Main Program):

```
IDENTIFICATION DIVISION.
PROGRAM-ID. 'OVERLAY'.

        .
        .
        .

ENVIRONMENT DIVISION.

        .
        .
        .
```

```
DATA DIVISION.

        .
        .
        .

WORKING-STORAGE SECTION.
   77 PROCESS-LABEL PICTURE IS X(8) VALUE IS 'OVERLAYB'.
   77 PARAM-1   PICTURE IS X.
   77 PARAM-2   PICTURE IS XX.
   77 COMPUTE-TAX PICTURE IS X(8) VALUE IS 'OVERLAYC'.
   01 NAMET.
      02 EMPLY-NUMB PICTURE IS 9(5).
      02 SALARY PICTURE IS 9(4)V99.
      02 RATE PICTURE IS 9(3)V99.
      02 HOURS-REG PICTURE IS 9(3)V99.
      02 HOURS-OT PICTURE IS 9(2)V99.
   01 COMPUTE-SALARY PICTURE IS X(8) VALUE IS 'OVERLAYD'.
   01 NAMES.
      02 RATES   PICTURE IS 9(6).
      02 HOURS   PICTURE IS 9(3)V99.
      02 SALARYX PICTURE IS 9(2)V99.

          .
          .
          .

PROCEDURE DIVISION.

         .
         .

    ENTER LINKAGE.
    CALL 'OVRLAY' USING PROCESS-LABEL, PARAM-1, PARAM-2.
    ENTER COBOL.

             .
             .

    ENTER LINKAGE.
    CALL 'OVRLAY' USING COMPUTE-TAX, NAMET.
    ENTER COBOL.

             .
             .
             .
             .

    ENTER LINKAGE.
    CALL 'OVRLAY' USING COMPUTE-SALARY, NAMES.
    ENTER COBOL.

             .
             .

    ENTER LINKAGE.
    CALL 'OVRLAY' USING COMPUTE-TAX, NAMET.
    ENTER COBOL.

             .
             .
```

COBOL Subprogram B:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  'OVERLAY1'.
        .
        .
        .

ENVIRONMENT DIVISION.

        .
        .

DATA DIVISION.

        .
        .

LINKAGE-SECTION.
01 PARAM-10   PICTURE IS X.
01 PARAM-20   PICTURE IS XX.
PROCEDURE DIVISION.
   ENTER LINKAGE.
   ENTRY 'OVERLAYX' USING PARAM-10, PARAM-20.
   ENTER COBOL.

        .
        .
        .

   ENTER LINKAGE.
   RETURN.
   ENTER COBOL.
```

COBOL Subprogram C:

```
IDENTIFICATION DIVISION.
PROGRAM-ID  'OVERLAY2'.

        .
        .

ENVIRONMENT DIVISION.

        .
        .

DATA DIVISION.
LINKAGE SECTION.
01 NAMEX.
   02 EMPLY-NUMBX PICTURE IS 9(5).
   02 SALARYX PICTURE IS 9(4)V99.
   02 RATEX PICTURE IS 9(3)V99.
   02 HOURS-REGX PICTURE IS 9(3)V99.
   02 HOURS-OTX PICTURE IS 9(2)V99.
PROCEDURE DIVISION.

        .
        .
   ENTER LINKAGE.
   ENTRY 'OVERLAYY' USING NAMEX.
   ENTER COBOL.

        .
        .
        .
        .

   ENTER LINKAGE.
   RETURN.
   ENTER COBOL.
```

COBOL Subprogram D:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  'OVERLAY3'.

        .
        .

ENVIRONMENT DIVISION.

        .
        .

DATA DIVISION.
LINKAGE SECTION.
01 NAMES.
   02 RATES PICTURE IS 9(6).
   02 HOURS PICTURE IS 9(3)V99.
   02 SALARYX PICTURE IS 9(2)V99.
PROCEDURE DIVISION.
   ENTER LINKAGE.
   ENTRY 'OVERLAYZ' USING NAMES.
   ENTER COBOL.

        .
        .
        .

   ENTER LINKAGE.
   RETURN.
   ENTER COBOL.
```

An assembly program called OVRLAY effects the overlay. It FETCHES the COBOL subprogram called by the COBOL main program and puts it in the overlay area.

Appendix A contains the assembly routine that accomplishes the overlay. Figure 14 is a flow diagram of the overlay logic.

Notice: If OVERLAYB were known to be in storage the CALL would be:

CALL 'OVERLAYX' USING PARAM-1, PARAM-2. But when using the OVRLAY subroutine it becomes:

CALL 'OVRLAY' USING PROCESS-LABEL, PARAM-1, PARAM-2.

where PROCESS-LABEL contains the external name 'OVERLAYB' of the subprogram.

However, the ENTRY statement of the subprogram is the same for both cases i.e. ENTRY' OVERLAYX' USING PARAM-10, PARAM-20, whether it is called indirectly by the main program through the OVERLAY program or called directly by the main program.

Note: An ENTRY which is to be called by OVRLAY must precede the first executable statement in the subprogram.

The job control statements required to accomplish overlay follow. The PHASE statements specify to the linkage editor that the overlay structure to be established is one in which subprograms OVERLAYB, OVERLAYC and OVERLAYD overlay each other when called during execution.

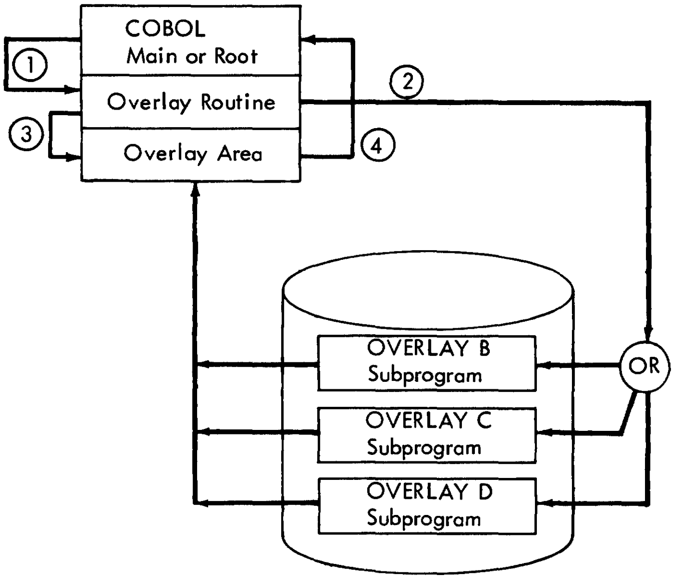


Figure 14. Flow Diagram of Overlay Logic

```
// JOB OVERLAYS
// OPTION LINK
   PHASE OVERLAY,ROOT
// EXEC COBOL
   ⎧COBOL Source for ⎫
   ⎨Main Program     ⎬
   ⎩'OVERLAY'        ⎭
/*
   PHASE OVERLAYB,*
// EXEC COBOL
   ⎧COBOL Source for ⎫
   ⎨Subprogram       ⎬
   ⎩'OVERLAYB'       ⎭
/*
   PHASE OVERLAYC,OVERLAYB
// EXEC COBOL

   ⎧COBOL Source for ⎫
   ⎨Subprogram       ⎬
   ⎩'OVERLAYC'       ⎭
/*
   PHASE OVERLAYD,OVERLAYB
// EXEC COBOL
   ⎧COBOL Source for ⎫
   ⎨Subprogram       ⎬
   ⎩'OVERLAYD'       ⎭
/*
// EXEC LNKEDT
// EXEC
/*
/&
```

Note: PHASE cards reorigin C and D overlays at the same origin as OVERLAYB. The sequence of events is:

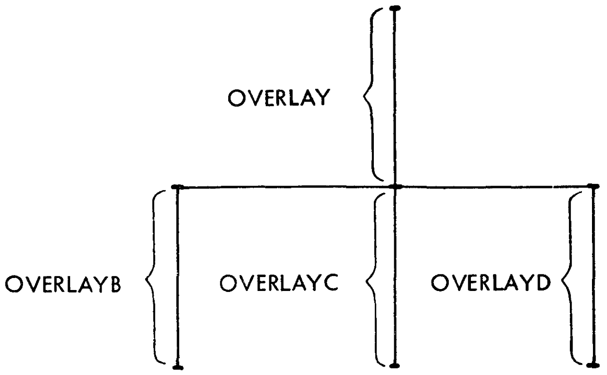
1. The main program calls the overlay routine.

2. The overlay routine fetches the particular COBOL subprogram and puts it in the overlay area, and then

3. Transfers to the first instruction of the subprogram.

4. The subprogram returns to the COBOL calling program (not the overlay subroutine).

The sequence of PHASE statements given in this example causes the linkage editor to structure a module as follows:



The phase name specified in the PHASE card must be the same as the value contained in the first argument for CALL 'OVRLAY' i.e., PROCESS-LABEL, COMPUTE-TAX etc contain OVERLAYB, OVERLAYC, respectively, which are the names given in the PHASE card.

PASSING PARAMETERS TO ASSEMBLER LANGUAGE ROUTINE

A subprogram may be written in assembler language to take advantage of the systems FETCH function or other control program options not available directly through the COBOL language. Thus, a main program in COBOL may link to a subprogram in assembler language, passing a data name that contains the name of the specific entry point desired.

An example of a COBOL program passing parameters to the assembler language follows:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  'COBOL'.

         .
         .

DATA DIVISION.

         .
         .
         .
         .
         .

WORKING STORAGE.
01 FIELD-1.
   02 FIELD1A PICTURE IS XXX.
   02 FIELD2A PICTURE IS XXX, VALUE IS 'ABC'.

   .
   .
   .

PROCEDURE DIVISION.
   MOVE 'ABC' TO FIELD1A.

   .
   .
   .

   ENTER LINKAGE.
   CALL 'ASC' USING FIELD1A, FIELD2A.
   ENTER COBOL.

   .
   .
```

```
SQRT START 0

   USING *, 15
   R14 EQU 14
   R1  EQU  1
   R2  EQU  2
   R3  EQU  3
   R0  EQU  0
   R15 EQU 15
   ENTRY ASC

     .
     .

   SAVE R14, R12

     .
     .

   L       2,0(R1)

   L       3,4(R1)

   CLC     0(3,R2),0(R3)

     .
     .

   BE CORRECT

     .
     .

CORRECT - - -
   RETURN R14,R12
```

Note: If SQRT is in the relocatable library, an INCLUDE SQRT card must be added to the input job stream.

If the assembler program were named ASC (the same as its ENTRY point) instead of SQRT, no INCLUDE card would be needed.

In this example the COBOL program calls an assembler program, passing it two parameters (FIELD1A and FIELD2A). The assembler program compares them logically (for any reason) finds them logically equal, and then returns to the COBOL program.

The purposes of register R1, R14, and R15 are:

Register R1 - Points to table of parameter addresses supplied by COBOL call statement.
Register R15 - Points to first executable instruction of assembler program.
Register R14 - Points to return statement in calling program.

The results of the assembler instructions in the SQRT routine are:

Start 0 - Informs the assembler to start assembling the program instruction, at the first available storage location for problem program.

USING *,15 - Specifies that the next instruction address be stored in the first operand (*) of this instruction.

R14 EQU 14 - Equates the Symbol R14 to the number 14 allowing user's coding to be more descriptive.

ENTRY ASC - Is the entry point into the assembler program.

SAVE R14,R12 - A macro instruction that saves the contents of all the general registers. (This protects any data the user might want to use when the COBOL program is re-entered.)

L 2,0(R1) - Loads the address of FIELD1A into register R2 (since FIELD1A is the first parameter in the table of parameters).

L 3,4(R1) - Loads the address of FIELD2A into register R3. (The second parameter in the table.)

CLC 0(3,R2),0(R3) - Compares first three characters of FIELD1A to first three characters of FIELD2A.

BE CORRECT - A conditional branch instruction that transfers to the address CORRECT if the CLC instruction proves true.

RETURN R14,R12 - Restores the general registers with their original contents (that which they contained at ENTRY time), and returns to the next statement after the CALL in the COBOL program.

SECTION VIII. PROGRAMMING CONSIDERATIONS

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. Discussed are:

- 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.
- Labeling considerations, multiprogramming considerations and DASD considerations.
- A technique for altering the DTF (Define the File) table.

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 in 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 operations.

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 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 MOVE is executed, COMPUTATIONAL-3 data is converted to DISPLAY 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 conditional 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 S999V99999999 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

System/360 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

In order 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). Figure 15 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

GROUP ITEM

Group moves of 256 or less bytes cost less than a series of single alphanumeric moves of the elementary items 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 half-word, or full-word boundary alignment. This is necessary for the elementary item to be handled properly in binary arithmetic. However, using group items that include slack bytes could cause problems.

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

Figure 15. Number of Bytes Required for Each Class of Elementary Item

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. Because 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, 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 16 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 first 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 if $1\leq\overset{**}{N}\leq4$ 4 if $5\leq\overset{**}{N}\leq9$ 8 if $10\leq\overset{**}{N}\leq18$	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 8	Fractional exponentiation, or very large or very small values	No	Yes	Tends to produce less accuracy. Computational-2 is more accurate than Computational-1. Requires floating-point feature.

Figure 16. 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:           Machine Representation is:

   PICTURE 9999.                 | F1 | F2 | F3 | F4 |
                                                 Byte
              or

   PICTURE S9999.                | F1 | F2 | F3 | D4 |
                                                 Byte
```

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:                 Machine Representation is:

   PICTURE S9999 COMPUTATIONAL-3.        | 01 | 23 | 4C |
                                                   Byte
               or

   PICTURE 9999 COMPUTATIONAL-3.         | 01 | 23 | 4F |
                                                   Byte
```

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:                 Machine Representation is:

   PICTURE S9999 COMPUTATIONAL.        | 0000 | 0100 | 1101 | 0010 |
                                         ↑
                                        sign              Byte
```

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:                      Machine Representation is:

COMPUTATIONAL-1.  | 0 | 1000011 | 0100 1101 0010 0000 0000 0000 |
                    S   1     7   8                            31
```

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 listed on the cover of this manual.

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:
   18 bytes for conversion and for the compare and branch instruction, and
   18 bytes for decimal alignment.

IF 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, COBOL Language Specifications, listed on the cover of this manual.

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 up 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, i.e.:

```
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 are unpredictable: 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 = VALUE1 ** (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:
      ┌─────────┐
    ┌─┴─┐    ┌──┴┐
    S99V9 =  100.0 - 2.
```

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

```
    FLD = 00.0̲ - 2
    FLD = 02.0̄, 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:

```
      ┌─────────┐
    ┌─┴──┐   ┌──┴┐
    S999V9 = 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 GO TO ERR.
Bad   MOVE TAB-FIELD (A,B,C) TO F.
Code: COMPUTE TAB-FIELD (A,B,C) = TAB-FIELD (A,B,C) + F / G.
```

This coding could be improved. by writing:

```
      MOVE TAB-FIELD (A,B,C) TO WORK-FLD.  ADD
      D TO WORK-FLD.  IF WORK-FLD = LIMIT-FLD
Good  GO TO ERR.
Code:
      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		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 does not require a work area. Less storage is used if multiple files are opened with one open than when an open statement is used for each file. A single open requires approximately 100 bytes of additional storage for each file-name.

To conserve storage, use:

```
OPEN INPUT FILEA, FILEB.
```

rather than:

```
OPEN INPUT FILEA OPEN INPUT FILEB.
```

ACCEPT Verb

The ACCEPT verb does not provide for recognition of the last card being read from a card reader. When COBOL detects a /* card it drops through to the next statement. Because no indication of this is given by COBOL, an end of file detection requires special treatment. Thus, the programmer must provide his own end card (some card other than /*) which he can test 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

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

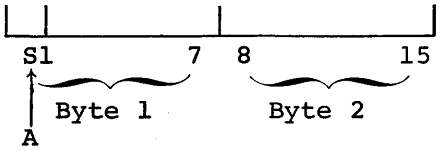
REDEFINITION

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

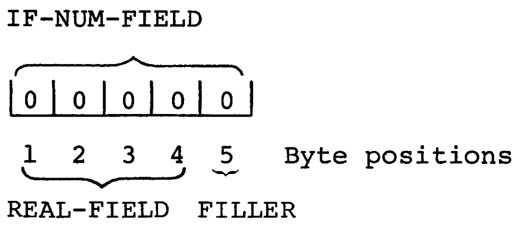
If a field is to be considered numeric only when it is unsigned, the sign position must be tested. A technique for relocating the sign (or "shifting") so that it can be tested as an unsigned numeric value follows.

Assume a field is defined:

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

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

The fields appear in storage as follows:

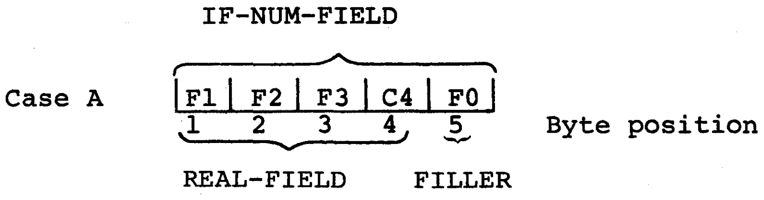


To make an IF NUMERIC, test true for only unsigned fields.

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:

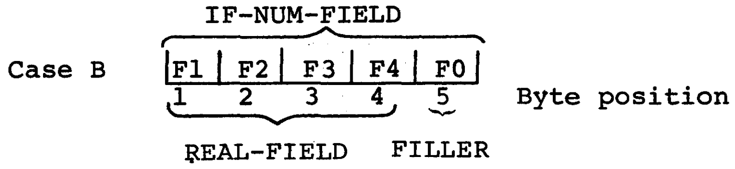


Case A

	IF-NUM-FIELD				
	F1	F2	F3	C4	F0
Byte position	1	2	3	4	5
	REAL-FIELD				FILLER

Note that the low-order byte (righmost 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:



Case B

	IF-NUM-FIELD				
	F1	F2	F3	F4	F0
Byte position	1	2	3	4	5
	REAL-FIELD				FILLER

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 an 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 not 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.

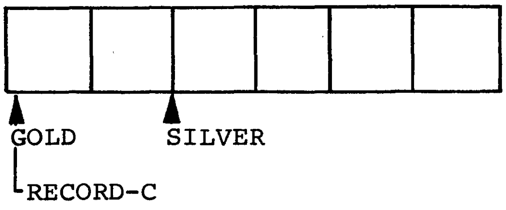
A warning diagnostic is given when slack bytes are inserted by the compiler.

Because COBOL alignes 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 that is 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.

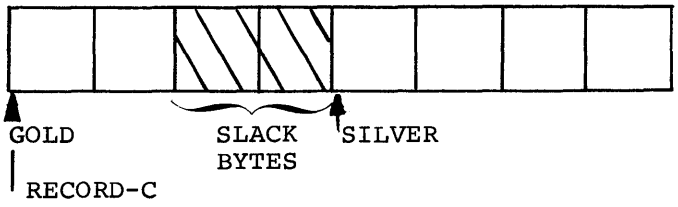
Assume a group record called RECORD-C exists on a file and consists of 2-bytes of alphanumeric data called GOLD, and 4-bytes of binary data called SILVER. The record on the file would appear as follows:



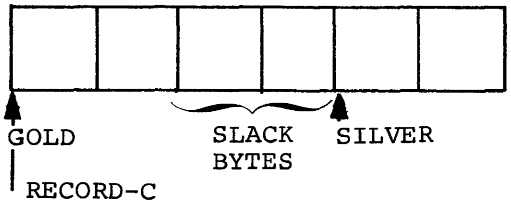
If an FD were defined:

```
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:



(This is the compiler generated address for SILVER)

Thus the first 2-bytes of the 02 SILVER are lost because of improper alignment. 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.
```

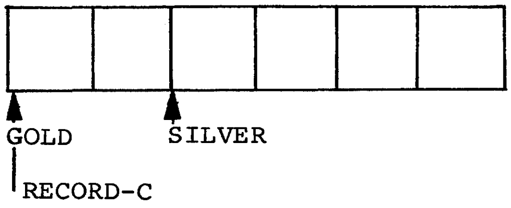
Now, access RECORD-C. This places it in the buffer, properly aligned. Then move the 4-byte elementary 02 SILVER (defined as alphanumeric but is actually binary data) to the record 01 LEAD. Because the 01 LEAD is a group item, the data moved retains its original form (no data conversion takes place) and 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, there is an alternate method of obtaining proper alignment when reading the record.

Define a record in an FD as follows:

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

The area defined would appear:

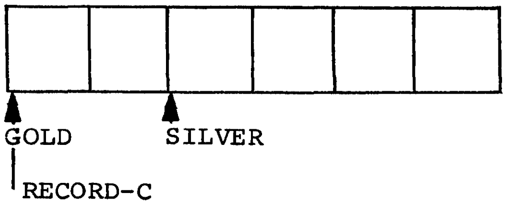


Then define a record in the WORKING-STORAGE section as:

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

As before, when the record is accessed, it is placed properly aligned in the buffer.

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 (DIAMONDS) 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, COBOL Language Specifications listed on the cover of this manual.

GENERAL INFORMATION--FILE HANDLING

BUFFERS

In IBM System/360 COBOL, a buffer is 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 a READ or WRITE statement 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 variability 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 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, the block is written out and a new one is started. Thus, use of the APPLY WRITE ONLY results in ignoring the maximum record size specified.

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 ERROR PROCESSING CONSIDERATIONS--USE AFTER STANDARD ERROR

The USE AFTER STANDARD ERROR clause provides the programmer with a means for investigating input/output processing errors. Depending upon the presence or absence of the declarative section, IOCS provides certain error processing procedures when an I/O error occurs. The following points should be considered when the USE AFTER STANDARD ERROR is used with the various types of file organization.

SEQUENTIAL TAPE FILE ORGANIZATION

1. If the declarative section is not included in the program and a wrong length record occurs, IOCS issues an illegal supervisor CALL of 32 which causes a storage dump.

   If the declarative section is not included in the program and a parity error is detected when a block of tape records is read, the tape is backspaced and reread 100 times. If the parity error persists, the tape block within which the error occurred is considered a tape error block, and the block is added to the block count found in the DTF table. IOCS indicates an I/O error (by a diagnostic) and cancels the job.

2. If the declarative section is included in the program and a parity error is detected when a block of tape records is read (described in 1 above), the tape is backspaced and reread 100 times. If the error persists, the tape block is considered a tape error block, and the block is added to the block count found in the DTF table. However, instead of canceling the job (this occurs when a declarative section is not included in the program), IOCS transfers control to the declarative section procedures to be followed on an error condition.

3. The address of the tape error block is stored by COBOL in register 3 + 192, and is accessible through an assembler subprogram.

   Normal return (to the main program) from the declarative section is through the IOCS subroutine invoked, thus bringing the next sequential block into main storage and permitting continued processing of the file (the bad block is bypassed).

   The programmer can interrogate the DTF table further, and display any pertinent data desired (such as block number) by using a CALL statement USING filename.

   A return through the use of GO TO does not bring the next block into main storage, therefore continued processing of the file is impossible.

4. In the case of tapes, the error declarative is only entered for read errors. For write errors, IOCS automatically retries 15 times (including skips and erases) and then cancels the job.

SEQUENTIAL DISK FILE ORGANIZATION

1. If the declarative section is not included in the program and a parity error occurs when a block of records is read, the disk block is reread 10 times. If the read error persists, the disk block within which the error occurred is considered a disk error block, and the job is terminated. If a parity error occurs when a block of records is written, IOCS attempts to write the block on an alternate track, and continued processing of the file is permitted.

   If the declarative section is not included in the program and a wrong length record occurs, IOCS issues an illegal supervisor CALL of 32 which causes a storage dump.

2. If the declarative section is included in the program, and a read or write error occurs that the programmer does not want canceled, the declarative section is entered.

   If a parity error occurs when a block of records is read (described in 1 above), the disk block is reread 10 times. If the read error persists, the disk block within which the error occurred is considered a disk error block and a READ operation cannot be issued to the error block. IOCS transfers control to the declarative section procedures to be followed on an error condition.

   In the case of a READ operation, normal return from the declarative is to the IOCS subroutine invoked, thus bringing the next sequential block into storage and permitting continued processing of the file.

   If a parity error occurs when a block of records is written, IOCS transfers control to the declarative section procedures to be followed on an error condition.

   In the case of a WRITE operation, normal return from the declarative is to the next instruction in the problem program. The disk block that was to be written is bypassed.

3. In the case of a READ error, a return from the declarative through the use of GO TO does not bring the next block into main storage. Continued processing of the file is impossible and the file must be closed.

   In the case of a WRITE error, a return from the declarative through the use of GO TO permits continued processing of the file. A normal return from the declarative results in the record to be written being bypassed.

DIRECT ACCESS FILE ORGANIZATION

1. If the declarative section is not included in the program and an error is detected in the execution of a direct access operation, the error is normally handled by COBOL object time subroutine IHD03400. A diagnostic message is output indicating the type of error and the file on which the error occurred.

2. If the declarative section is included in the program and an error is detected in the execution of a direct access operation, the programmer can interrogate bytes 254 and 255 of the DTF table to determine the type of error. (See Altering DTF Table for a technique of accessing the DTF table.) Interrogation can be accomplished by using the CALL statement USING filename (thus exiting from the declarative) and processing the error in a subprogram. The user is responsible for writing the error checking subprogram.

The error indications are:

For byte 254:	X'40' - wrong length record X'08' - no room found
For byte 255:	X'80' - data check count X'10' - data check in key X'08' - no record found

3. Normal return from the declarative is to the next sequential instruction in the problem program following the I/O operation.

4. Return from a declarative through the use of GO TO can be to any location but the programmer should be aware that the record for the last I/O operation is not located.

INDEXED SEQUENTIAL FILE ORGANIZATION

1. If the declarative section is not included in the program and an error is detected in the execution of an indexed sequential access operation, the error is normally handled by COBOL object time subroutine IHD03500. A diagnostic message is output indicating the type of error and the file on which the error occurred.

2. If the declarative section is included in the program and an error is detected in the execution of an indexed sequential access operation, the programmer can interrogate byte 30 of the DTF table to determine the type of error. (See Altering DTF Table for a technique of accessing the DTF table.) Interrogation can be accomplished by using the CALL statement USING filename (thus exiting from the declarative) and processing the error in a subprogram. The user is responsible for writing the error checking subprogram.

   The error indications are:

byte 30	ADD, RETRVE, ADDRTR	LOAD
X'80'	Direct access device error	Direct access device error
X'40'	Wrong length record	Wrong length record
X'20'	End of file	Prime data area full
X'10'	No record found	Cylinder index area full
X'08'		Master index area full
X'02'	Overflow area full	

3. Normal return from the declarative is to the next sequential instruction in the problem program following the I/O operation.

4. Return from a declarative through the use of GO TO can be to any location but the programmer should be aware that the record for the last I/O operation is not located.

LABELING CONSIDERATIONS

PROCEDURE FOR BYPASSING NON-STANDARD LABELS

COBOL requires that labels be either standard or omitted. When non-standard labels are used, a technique is required to bypass (or process) them. To bypass non-standard labels on input, the following procedure could be used:

- Include an MTC FSF SYSnnn job control statement before the EXEC statement in the job control stream.

- Specify LABELS ARE OMITTED and OPEN NO REWIND statements in the COBOL source program.

An example of COBOL statements and the job control statements for bypassing non-standard labels follows:

COBOL Statements.

```
DATA DIVISION.

FD  FILEA DATA RECORD IS RECORD-A, LABEL RECORDS ARE OMITTED,
    RECORDING MODE IS F.

PROCEDURE DIVISION.

    OPEN INPUT FILEA WITH NO REWIND.
```

Job Control Statements

```
// JOB MAIN

// ASSGN SYSnnn ...

// EXEC COBOL

   COBOL Source statements

// EXEC LNKEDT

// MTC FSF,SYSnnn[,01]

// EXEC

/*

/&
```

Note: The parameter 01 in the MTC statement implies that the tape mark following the file label will be detected.

MULTIPROGRAMMING CONSIDERATIONS

When the COBOL programmer uses the multiprogramming capability, the following points must be considered.

1. The COBOL compiler must always be executed as a background program.

2. Object programs produced by the COBOL compiler (linkage edited) can operate as foreground programs with the following restrictions.

   - With the exception of SYSLOG, no system logical units can be referenced in foreground programs.

   - The EXHIBIT and TRACE statements cannot be used in foreground programs. (The output of these statements is given on the system logical unit SYSLST.)

   - The CONSOLE option of the ACCEPT and/or DISPLAY statements must be specified when these statements are used.

Note: In addition to the restrictions given above, object time messages are output on SYS000 instead of SYSLST.

The programmer interested in the structure of main storage, and storage requirements related to multiprogramming should read the discussion Main Storage Organization given in Section IX.

PROCESSING INDEXED AND DIRECT FILES

Following is a discussion of what happens in Disk Operating System COBOL when creating, retrieving, adding, or updating a file whose data organization is specified as indexed or direct.

CREATING A SEQUENTIAL, INDEXED SEQUENTIAL FILE

To create a sequential, indexed sequential file, the following clauses are required:

- ORGANIZATION IS INDEXED
- ACCESS IS SEQUENTIAL
- ASSIGN TO DIRECT-ACCESS
- RECORD KEY IS data-name

(The SYMBOLIC KEY may be specified.)

The programmer must then specify:

- OPEN OUTPUT file-name
- WRITE record-name [INVALID KEY]
- CLOSE file-name

OPEN Statement

The OPEN causes the label information for the file to be recorded in a Volume Table of Contents (VTOC). It then initiates a checking procedure that prevents writing on an existing file that might still be active. In addition, OPEN establishes the area that is to be read on the disk as specified in the XTENT statement (by the LOWER and UPPER parameters). Finally, OPEN initializes the cylinder and track index tables, which are eventually filled with the RECORD keys provided by the programmer when the file is being created.

WRITE Statement

The WRITE enters into the track and cylinder index tables the keys (RECORD KEY'S) specified by the programmer and writes the actual data on the portion of the track defined by the XTENT parameters (see OPEN statement discussion for these parameters). The records are placed on the track sequentially in an area referred to as the "prime data area".

If the user specifies INVALID KEY, control is given to the invalid key routine whenever a duplicate record or a record out of sequence is detected. The user is responsible for writing the invalid key routine.

CLOSE Statement

The CLOSE removes the reference to the labels in the VTOC, updates indexes (track and cylinder) and writes the end-of-file record. Once the reference to the labels is removed from the VTOC, the file must be opened again to be accessed. The index tables are updated each time a block is written out (in the case of blocked records) or each time a record is written (in the case of unblocked records). For a short block, the CLOSE results in truncation of the area not used in the block and in updating of the indexes (with the RECORD KEYs of those records in the block).

Key Handling

During the creation of a sequential, indexed sequential file, the user can control the RECORD KEY with certain restrictions:

- The RECORD KEY must be provided before execution of the WRITE. It is part of the record and identifies the particular record in the file.
- The RECORD KEY values must be given in ascending collating sequence.
- No two keys can be the same.

To extend a file previously created, the same clauses and control statements (VOL, XTENT) used to create the file are required, with the following exception: the parameter ISE should be used for the 'type' code in the DLAB statement instead of ISC (used for creating the file).

Note that the record to be added must fit within the limits originally specified for the file by the XTENT statement. If it does not fit, the file must be recreated.

The SYMBOLIC KEY is not required when creating a sequential, indexed sequential file.

SEQUENTIAL RETRIEVAL OF AN INDEXED SEQUENTIAL FILE OR UPDATING AN INDEXED SEQUENTIAL FILE

Sequential Retrieval

To retrieve an indexed sequential file sequentially, the following clauses are required:

- ORGANIZATION IS INDEXED
- ACCESS IS SEQUENTIAL
- RECORD KEY IS data-name

(The SYMBOLIC KEY may be specified.)

To simply read the file, the programmer must specify:

- OPEN INPUT file-name
- READ file-name AT END
- CLOSE file-name

OPEN Statement: OPEN checks the labels of the files to be opened and initializes the VTOC to indicate an active file. It also establishes the area to be read as specified by the LOWER and UPPER limit parameters of the XTENT statement. This initializes processing of the file as follows:

- If the SYMBOLIC KEY is omitted, processing begins with the first record of the file, and progresses sequentially.
- If the SYMBOLIC KEY is used and binary zeros are specified therein, processing begins with the first record of the file, and progresses sequentially.
- If the SYMBOLIC KEY is used and other than binary zeros are specified, processing begins with the specified key and progresses sequentially.

READ Statement: The READ causes sequential retrieval of logical records from the file until the end-of-file record is detected. At this time control is given to the user routine specified by the AT END statement.

CLOSE Statement: The file is reset for future use.

Updating

To update an existing indexed sequential file, the same clauses needed to retrieve the file are required (ORGANIZATION IS INDEXED, ACCESS IS SEQUENTIAL, RECORD KEY IS) and the programmer must specify:

- OPEN I-O file-name
- READ file-name AT END
- REWRITE record-name [INVALID KEY]
- CLOSE file-name

The OPEN and CLOSE statements function the same as for sequential retrieval of an indexed sequential file. The READ also functions the same (as for sequential retrieval) but must be used in conjunction with the REWRITE as follows.

REWRITE Statement: The REWRITE writes the logical record (read by a preceding READ statement) back into the same physical location from which it was originally retrieved. Thus, the REWRITE provides the facility to update records in a file. Under no circumstances should the user modify the RECORD KEY of the record being updated. Because the INVALID KEY check is not exercised for sequential retrieval of an indexed sequential file, results caused by modification of the RECORD KEY prior to return of the record to the file are unpredictable.

Key Handling

During sequential retrieval of a file, limited control of the SYMBOLIC and RECORD KEY is permitted. Thus, the SYMBOLIC KEY can be set before the OPEN is executed, allowing processing to begin with any record within the file. (Once the OPEN is completed, the SYMBOLIC KEY is not needed.) The RECORD KEY, which must not be modified when updating a file, can be referenced when retrieving a record for the purpose of recognizing a particular record in the file.

RANDOM RETRIEVAL OF AN INDEXED SEQUENTIAL FILE

To retrieve, randomly update, or add to an indexed sequential file, the following clauses are required:

- ORGANIZATION IS INDEXED
- ACCESS IS RANDOM
- SYMBOLIC KEY IS data-name
- RECORD KEY IS data-name

Random Retrieval

To retrieve randomly, the following clauses must be specified:

- OPEN INPUT file-name
- READ file-name INVALID KEY
- CLOSE file-name

The OPEN and CLOSE statements function the same as for sequential retrieval of an indexed sequential file. The clauses specified allow random retrieval only. Before retrieval of each record, the SYMBOLIC KEY must be provided.

Updating Randomly

To update an indexed sequential file randomly, the following clauses must be specified:

- OPEN I-O file-name
- READ file-name INVALID KEY
- REWRITE record-name [INVALID KEY]
- CLOSE file-name

The OPEN and CLOSE statements function the same as for sequential retrieval of an indexed sequential file. The READ retrieves the record identified by the SYMBOLIC KEY. This key must be specified for every READ and must be within the limits of the file (UPPER and LOWER limits of XTENT), otherwise a 'NO RECORD FOUND' condition results. If this occurs, control is given to the user's INVALID KEY routine.

The REWRITE clause permits random updating of records in a file. It must be preceded by a READ, and the SYMBOLIC and RECORD KEYs must not be modified before the REWRITE is executed. (NO INVALID KEY check is available for the update function.)

Adding Randomly

To add to an indexed sequential file randomly, the same clauses needed to retrieve the file are required, and in addition, the programmer must specify:

- OPEN I-O file-name
- WRITE record-name [INVALID KEY]
- CLOSE file-name

The OPEN and CLOSE statements function the same as for sequential retrieval of an indexed sequential file. Records can be added to an existing file by means of the WRITE clause. The WRITE requires that the RECORD KEY be initialized before the operation.

A duplicate key error results when a record being added has the same RECORD KEY value as a record already in the file. This condition causes control to be given to the user's invalid key routine.

Key Handling

The programmer must initialize the SYMBOLIC KEY with a key value prior to every READ. The value must be equal to the record key within the record to be retrieved. This key must be within the file limit (UPPER and LOWER of XTENT), otherwise a 'NO RECORD FOUND' error condition results. The RECORD KEY can only be used for record reference during the retrieve and update functions. For the add function, this key must be initialized before each write.

CREATING A DIRECT ORGANIZATION FILE

To create a direct file, the following clauses are required:

- ORGANIZATION IS DIRECT
- ACCESS IS SEQUENTIAL
- SYMBOLIC KEY IS data-name
- ACTUAL KEY IS data-name

The programmer must then specify:

- OPEN OUTPUT file-name
- WRITE record-name [INVALID KEY]
- CLOSE file-name

OPEN Statement

The OPEN initializes the VTOC to indicate the presence of the labels and checks the label area for a vaild output file. It also establishes the limits of the file as defined in the XTENT statement. It checks to be sure that the file limits specified do not overlap with an existing file and completes the DTF (Define The File) table for the file opened. Thus, it enters the system logical unit specified for the file into the table. In addition, the OPEN initializes the capacity records (R0) over the entire area of the XTENT(s) for the output file.

WRITE Statement

The WRITE transfers the record to the DASD address specified in the ACTUAL KEY. The specified SYMBOLIC KEY becomes a part of the record in the file.

Key Handling

When handling keys, the following restrictions are imposed:

1. The programmer must provide the SYMBOLIC KEY for every record loaded.

2. When creating a file, no provision is made to prevent the addition of a duplicate record.

CLOSE Statement

The CLOSE returns the track address of the end-of-file record in the ACTUAL KEY.

SEQUENTIAL RETRIEVAL OF A DIRECT ORGANIZATION FILE

To retrieve a direct file sequentially, the following clauses are required:

- ORGANIZATION IS DIRECT
- ACCESS IS SEQUENTIAL
- SYMBOLIC KEY IS data-name
- ACTUAL KEY IS data-name

The programmer must then specify:

- OPEN INPUT file-name
- READ file-name AT END
- CLOSE file-name

READ Statement

The READ retrieves the file sequentially beginning with the lower XTENT.

OPEN, CLOSE

The OPEN checks labels on the label track and initializes the VTOC. The limits of the XTENTs are established at this time. CLOSE is a no-operation.

Key Handling

During sequential retrieval of a direct file, the SYMBOLIC and ACTUAL KEY are ignored.

RANDOM RETRIEVAL, UPDATING AND ADDING TO A DIRECT FILE

To retrieve a direct file randomly, the following clauses are required:

- ORGANIZATION IS DIRECT
- ACCESS IS RANDOM
- SYMBOLIC KEY IS data-name

- ACTUAL KEY IS data-name

  The programmer must then specify:

- OPEN INPUT file-name

- READ file-name INVALID KEY

- CLOSE file-name

Random Retrieval

The READ retrieves a record from the information given in the SYMBOLIC KEY. The search begins at the DASD address specified in the ACTUAL KEY.

Updating Randomly

To update randomly, the programmer must specify:

- OPEN I-O file-name

- READ file-name INVALID KEY

- REWRITE record-name [INVALID KEY]

- CLOSE file-name

  When updating a file, the keys must not be modified.

Adding Randomly

The WRITE allows new records to be added to the file. When adding records to an existing file, both the ACTUAL and SYMBOLIC KEYs must be supplied. The record is written into the specified location. When adding randomly to a direct file, no provision is made to prevent the addition of a duplicate record.

OPEN, CLOSE

For random retrieval, the OPEN and CLOSE functions are the same as for sequential retrieval of a direct file.

Key Handling

When a file is accessed randomly, both the ACTUAL and SYMBOLIC KEYs must be initialized by the user before the READ or WRITE is specified. The ACTUAL KEY contains the DASD address and the SYMBOLIC KEY identifies the record within the file.

DIRECT ACCESS DATA ORGANIZATION CONSIDERATIONS

When the COBOL programmer defines files for direct access storage devices (DASD) the following points must be considered.

1. For processing a file whose organization is direct, up to five extents are permitted. For processing a file whose organization is indexed, up to 11 extents are permitted.

2. When sequential retrieval is indicated, a record must appear on the first track of every cylinder.

3. For direct organization files, an end-of-file (EOF) is written on the last track if the last extent specified for the file.

4. The verify option assumed by the compiler (verification consists of IOCS checking to be sure that a record written out is correct) can be changed so that verification is suppressed. Suppression of the verify option could result in improved program performance. Prior to OPEN, the fourth byte of the Define The File (DTF) table controls the verify option for direct access and indexed sequential organized files. Changing the value of the fourth byte to X'00' suppresses the verify option. See Altering DTF Table for the procedure to change this byte.

5. For an indexed sequential file, COBOL presumes the absence of a master cylinder index. Its presence can be indicated by changing the value in the twenty-first byte of the DTF table prior to OPEN. For a 2311, change the contents of the twenty-first byte to X'F2', and for 2321 change its contents to X'02'.

6. Prior to open, the twenty-second byte of the DTF table for indexed sequential data organization contains the number of overflow tracks assumed per cylinder. The number of overflow tracks is equal to 20% of the tracks per cylinder. The programmer may change this value if he wishes by writing a subprogram that changes the value of the twenty-second byte in the DTF table before the OPEN. See Altering DTF Table for the procedure to change this byte.

7. For direct files, the ACTUAL KEY must be provided before the record is processed. (See Coding ACTUAL KEY for 2311 Disk Pack and 2321 Data Cell, and Updating ACTUAL KEY for 2321 Data Cell.)

8. When processing files, the following IOCS error indications will cause COBOL to go to the INVALID KEY routine.

   When organization is direct and:

   a. no room is found during an update (WRITE or REWRITE).

   b. no record is found during retrieval (READ).

   When organization is indexed sequential and:

   a. a duplicate record exists (duplicate key) (WRITE).

   b. when building a file, a record is out of sequence (sequence check) (WRITE).

   c. no record is found during retrieval (READ).

MULTIPLE ENTRY POINTS

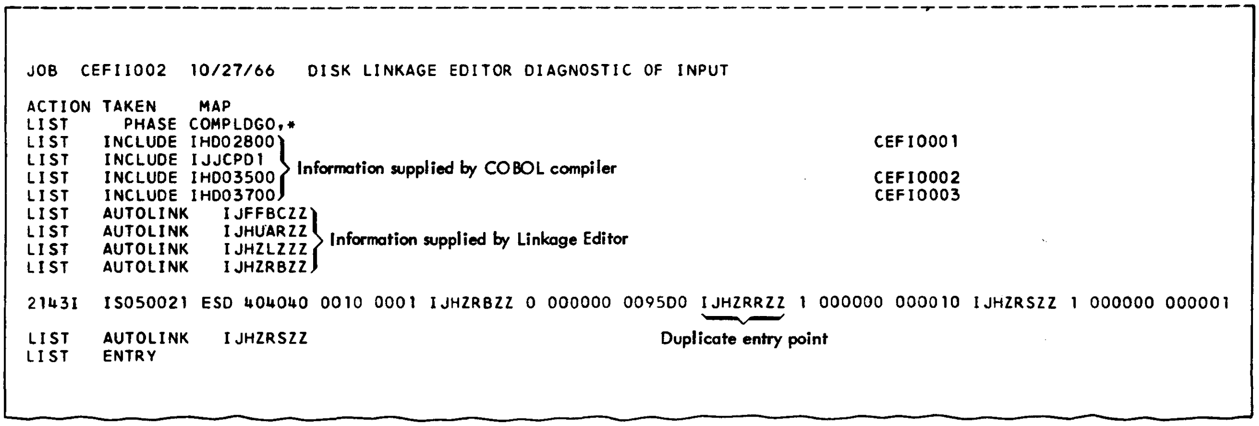
When more than one type of retrieval is specified for direct access files in a program, an indication of duplicate entry points may be given at linkage edit time. If duplicate entry points occur, the user must construct and include a supersetted LIOCS module that contains the individual modules.

The following example shows how:

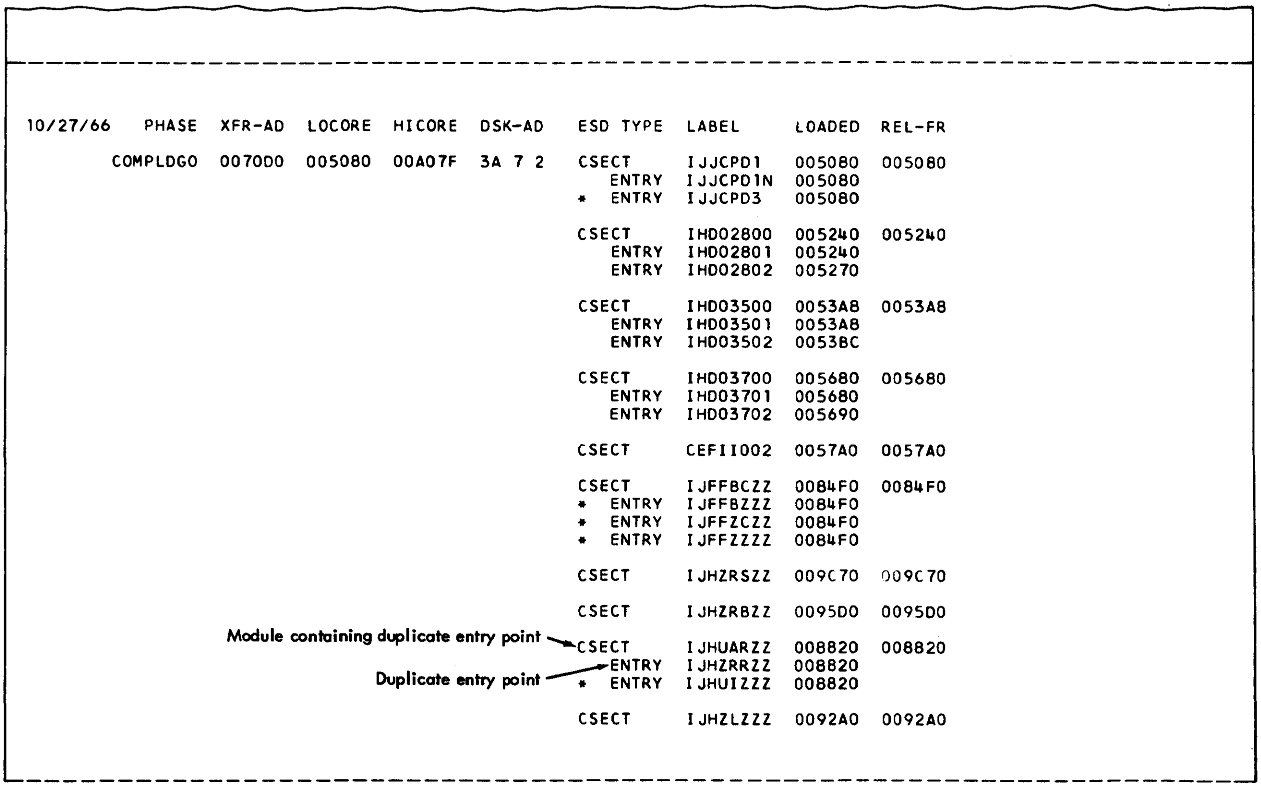
1. The module containing the duplicate entry point can be identified, and

2. the supersetted module is built, and included in the COBOL object program in place of the individual modules.

If a number of direct files are defined to be used by the same program, the following linkage editor diagnostics might be obtained (they are included in the DISK LINKAGE EDITOR DIAGNOSTIC OF INPUT).



```
JOB  CEFII002  10/27/66   DISK LINKAGE EDITOR DIAGNOSTIC OF INPUT

ACTION TAKEN    MAP
LIST     PHASE COMPLDGO,*
LIST   INCLUDE IHD02800                                                 CEFI0001
LIST   INCLUDE IJJCPD1    Information supplied by COBOL compiler
LIST   INCLUDE IHD03500                                                 CEFI0002
LIST   INCLUDE IHD03700                                                 CEFI0003
LIST   AUTOLINK    IJFFBCZZ
LIST   AUTOLINK    IJHUARZZ   Information supplied by Linkage Editor
LIST   AUTOLINK    IJHZLZZZ
LIST   AUTOLINK    IJHZRBZZ

2143I  IS050021 ESD 404040 0010 0001 IJHZRBZZ 0 000000 0095D0 IJHZRRZZ 1 000000 000010 IJHZRSZZ 1 000000 000001

LIST   AUTOLINK    IJHZRSZZ                              Duplicate entry point
LIST   ENTRY
```



```
10/27/66   PHASE   XFR-AD  LOCORE  HICORE  DSK-AD   ESD TYPE  LABEL     LOADED  REL-FR

        COMPLDGO  0070D0  005080  00A07F  3A 7 2   CSECT     IJJCPD1   005080  005080
                                                      ENTRY  IJJCPD1N  005080
                                                   *  ENTRY  IJJCPD3   005080

                                                   CSECT     IHD02800  005240  005240
                                                      ENTRY  IHD02801  005240
                                                      ENTRY  IHD02802  005270

                                                   CSECT     IHD03500  0053A8  0053A8
                                                      ENTRY  IHD03501  0053A8
                                                      ENTRY  IHD03502  0053BC

                                                   CSECT     IHD03700  005680  005680
                                                      ENTRY  IHD03701  005680
                                                      ENTRY  IHD03702  005690

                                                   CSECT     CEFII002  0057A0  0057A0

                                                   CSECT     IJFFBCZZ  0084F0  0084F0
                                                   *  ENTRY  IJFFBZZZ  0084F0
                                                   *  ENTRY  IJFFZCZZ  0084F0
                                                   *  ENTRY  IJFFZZZZ  0084F0

                                                   CSECT     IJHZRSZZ  009C70  009C70

                                                   CSECT     IJHZRBZZ  0095D0  0095D0

            Module containing duplicate entry point CSECT     IJHUARZZ  008820  008820
                                                      ENTRY  IJHZRRZZ  008820
                            Duplicate entry point  *  ENTRY  IJHUIZZZ  008820

                                                   CSECT     IJHZLZZZ  0092A0  0092A0
```

Notice that the LIOCS modules are separately included in the program (see AUTOLINK IJ..... entries near the top of the listing). When the modules are linkage edited with the COBOL program, an indication of a duplicate entry point may be given. The duplicate entry point is included in the line of print identified by the message number 2143I and belongs to the module IJHZRBZZ. This message number is listed in the operating guide for the system, and indicates an invalid duplication of entry point label.

The user can identify the module containing the duplicate entry point and build a supersetted module as follows.

Compare the IJH..... (entry points) given in the line next to the message number, to the ENTRY points given in the LABEL column part of the listing.

In this example, the duplicate ENTRY point is ENTRY IJHZRRZZ (the second one in the 2143I line of print, and the third one from the bottom in the LABEL column listing). Thus, this duplicate entry point is in the module CSECT IJHUARZZ (see the entry just above IJHZRRZZ in the LABEL column listing). The module should also be among those given in the AUTOLIST list.

From this module (IJHUARZZ) and module IJHZRBZZ, a supersetted module must be formed as follows. Use the first three characters of the module name for the functions used. In this case, they would by IJH. Then use the lowest letter, between the two modules, for each of the next five character positions, as follows:

```
I J H U A R Z Z
      ↓ ↓
I J H U A B Z Z    Supersetted module
          ↑
I J H Z R B Z Z
```

Thus, the name of the supersetted LIOCS module that contains the individual modules (IJHUARZZ and IJHZRBZZ) is IJHUABZZ.

The supersetted module can then be included with the COBOL object program at linkage edit time instead of the individual modules (IJHUARZZ and IJHZRBZZ) by inserting an INCLUDE card before the linkage edit function as follows:

```
   INCLUDE IJHUABZZ

// EXEC LNKEDT
```

CODING ACTUAL KEY FOR 2311 DISK PACK AND 2321 DATA CELL.

When creating or processing direct access files, the programmer is responsible for providing the ACTUAL KEY, in binary, for each record to be processed. The ACTUAL KEY is an eight-byte field that contains specific byte specifications and limits depending on whether the device used is a 2311 Disk Pack or 2321 Data Cell.

The ACTUAL KEY field must be defined <u>before</u> a record can be processed. The structure and examples of code for the eight-byte ACTUAL KEY field for both the 2311 and 2321 direct access devices follow.

Specification for 2311 Disk Pack:

	Pack Number (M)	Cell (BB)	Cylinder (CC)	Head (HH)	Record (R)
Byte Position	0	1 2	3 4	5 6	7
	0-244	0 0	0 0-199	0 0-9	0-255

An example of a method of coding the eight-byte ACTUAL KEY in binary, using COBOL, for the 2311 Disk Pack is:

```
01  BINARY-KEY-RECORD.
  02 MM USAGE IS COMPUTATIONAL PICTURE IS  999 VALUE IS 0.
  02 BB USAGE IS COMPUTATIONAL PICTURE IS  9 VALUE IS 0
  02 CC USAGE IS COMPUTATIONAL PICTURE IS  999 VALUE IS 10.
  02 HH USAGE IS COMPUTATIONAL PICTURE IS  9 VALUE IS 0.
  02 REC-R PICTURE IS X VALUE IS LOW-VALUE.
01   KEY-AS-ACTUAL REDEFINES BINARY-KEY-RECORD.
  02 FILLER PICTURE IS X.
  02 THE-ACTUAL-KEY PICTURE IS X(8).
```

Although the ACTUAL KEY field consists of eight bytes, nine bytes are defined by the given code. The 02 MM defines two bytes, the first byte of which is disposed of by the 02 FILLER PICTURE IS X in the REDEFINITION statement. Thus, the code defines an eight-byte binary field called THE-ACTUAL-KEY which is used by IOCS to access records. A pictorial structure of THE-ACTUAL-KEY field defined by the code follows:

Pack Number (M)	Cell (BB)		Cylinder (CC)		Head (HH)		Record (R)
0	1	2	3	4	5	6	7
0	0	0	0	10	0	0	0

Specification for 2321 Data Cell is:

	Pack Number (M)	Cell (BB)		Cylinder (CC)		Head (HH)		Record (R)
				Sub Cell	Strip	Head Bar	Head Element	
Byte Position	0	1	2	3	4	5	6	7
	0-244	0	0-9	0-19	0-9	0-4	0-19	0-255

An example of a method of coding the eight-byte ACTUAL KEY in binary, using COBOL, for the 2321 Data Cell is:

```
01  BINARY-KEY-RECORD.
  02 MM USAGE IS COMPUTATIONAL PICTURE IS  999 VALUE IS 0.
  02 BB USAGE IS COMPUTATIONAL PICTURE IS  9 VALUE IS 0.
  02 CC USAGE IS COMPUTATIONAL PICTURE IS  9999 VALUE IS 1.
  02 HH USAGE IS COMPUTATIONAL PICTURE IS  999 VALUE IS 0.
  02 REC-R PICTURE IS X VALUE IS LOW-VALUE.
01   KEY-AS-ACTUAL REDEFINES BINARY-KEY-RECORD.
  02 FILLER PICTURE IS X.
  02 THE-ACTUAL-KEY PICTURE IS X(8).
```

Notice that just as for the 2311, nine bytes are defined and then redefined to eliminate the first byte, leaving eight bytes. Thus, the code defines the ACTUAL KEY which is used by IOCS to access records. A pictorial structure of THE-ACTUAL-KEY field as defined by the given code follows:

	Pack Number (M)	Cell (BB)		Cylinder (CC)		Head (HH)		Record (R)
				Sub Cell	Strip	Head Bar	Head Element	
Byte Position	0	1	2	3	4	5	6	7
	0	0	0	0	1	0	0	0

As records are processed, IOCS automatically updates Record (R) (REC-R). When the desired number of records are processed within the defined area of a strip, or no more room is available in a strip area, the next head element must be accessed in order to continue processing on that strip. When all head elements have been used, the next head bar must be accessed, making 20 new head elements available. Thus, 256 records can be accessed per head element and 20 head elements per head bar. Also, 5 head bars can be accessed per strip, 10 strips per subcell, and 10 subcells per pack. The number of packs available are 255.

EXAMPLE OF UPDATING ACTUAL KEY

Once the programmer defines an ACTUAL KEY for the 2311 Disk Pack or 2321 Data Cell, it must be updated before each record is processed. The ACTUAL KEY for the 2311 can be updated by conventional methods of incrementing (adding one to the byte position of interest). However, when updating the ACTUAL KEY for the 2321, conventional methods of incrementing do not allow use of the high order byte positions for the head bar, strip etc. These can be accessed only by forcing a binary overflow into their high order byte positions. The following is an example of a method for updating the ACTUAL KEY for the 2321 (which must be a binary number). Assume that an ACTUAL KEY was defined so that cylinder strip byte position 4 equals 1, and all other bytes are equal to zero, the code to increment the head element is:

```
A2.
   ADD 1 TO HH.    (Increments the head element.  This is defined as the
                   head element in the code defining the ACTUAL-KEY for
                   the 2321 Data Cell.)

A3.
   PERFORM A2 20 TIMES.   (The maximum number of head elements is 20.)
   ADD 236 TO HH.   (This causes a binary overflow from head element
                     byte position 6 to head bar position 5.)
A4.
   PERFORM A3 4 TIMES.   (The maximum number of head bars is 5.)
   ADD 1 TO CC.    (Increments strip.)
   MOVE ZERO TO HH.   (Resets head element to zero.)
```

When all the strips are used (there are 10 strips), the next subcell must be accessed. This can be done by forcing a binary overflow similar to that demonstrated for the head (HH). To accomplish this add 236 to 10 (strips), forcing a one into byte position 3 (subcell). Cells (BB), and Packs (M) can be accessed through conventional methods of incrementing.

ALTERING DTF (DEFINE THE FILE) TABLE (Disk System Only)

The programmer interested in suppressing the verify option or changing the number of overflow tracks per cylinder may do so by changing the appropriate byte(s) in the DTF table in one of the following ways:

- Writing a COBOL main program to access the DTF table and a COBOL subprogram to change the DTF table.

- Writing a COBOL main program to access the DTF table and an assembler subprogram to change the DTF table.

Example Using COBOL

Define, CALL and operate on a dummy DTF table as follows.

In a COBOL subprogram called 'CHGPRG', define a dummy DTF and entry point:

```
   In the LINKAGE SECTION, write:
          01  DTF-FOR-FILEA               }
                                          }  Dummy DTF
              02  BYTE PICTURE X OCCURS 125. }

   In the PROCEDURE DIVISION, write:

   ENTER LINKAGE.
   ENTRY 'CHGPRG' USING DTF-FOR-FILEA.
   ENTER COBOL.
   .
   .
   .
   MOVE ZERO TO BYTE(4).
```

In the main COBOL program define and CALL the subprogram:

```
   In the DATA DIVISION, write:
   FD FILEA.

   In the PROCEDURE DIVISION, write:
   ENTER LINKAGE.
   CALL 'CHGPRG' USING FILEA.
   ENTER COBOL.
```

When the CALL is executed, the address of the first byte of the DTF table is stored in the location for FILEA. The MOVE ZERO TO BYTE(4) statement in the COBOL subprogram initializes the fourth byte in the DTF table. To alter the twenty-second byte of the DTF table, a similar MOVE statement could be written. For example:

```
   MOVE ANY-VALUEX TO BYTE(22).
```

Note: The byte count of a DTF table begins with zero, thus byte 22 is actually the twenty-third byte of the table.

Example Using Assembler

Define and CALL a file in a COBOL main program.

```
   In the DATA DIVISION, write:
   FD FILEA                          (Defined file)
```

In the PROCEDURE DIVISION, write a CALL that specifies the file in the USING portion of the CALL statement:

```
   PROCEDURE DIVISION.
       ENTER LINKAGE.                      (Call that links to
       CALL 'CHGPRG' USING FILEA.           assembler subprogram
       ENTER COBOL.                         'CHGPRG')
```

When the CALL is executed, the first address of the DTF table is stored at the storage location for FILEA. Access to any byte in the DTF table can be accomplished by an assembler subprogram. It is the programmers responsibility to write all the code (assembler language subprogram included) for both methods of accessing and altering the COBOL DTF table.

The Disk and Tape Operating Systems are a group of processing programs with the control programs necessary to maintain their continuous operation. They are self-contained systems and require a minimum of operator intervention. Figure 17 is a flow diagram of Disk and Tape Operating Systems.

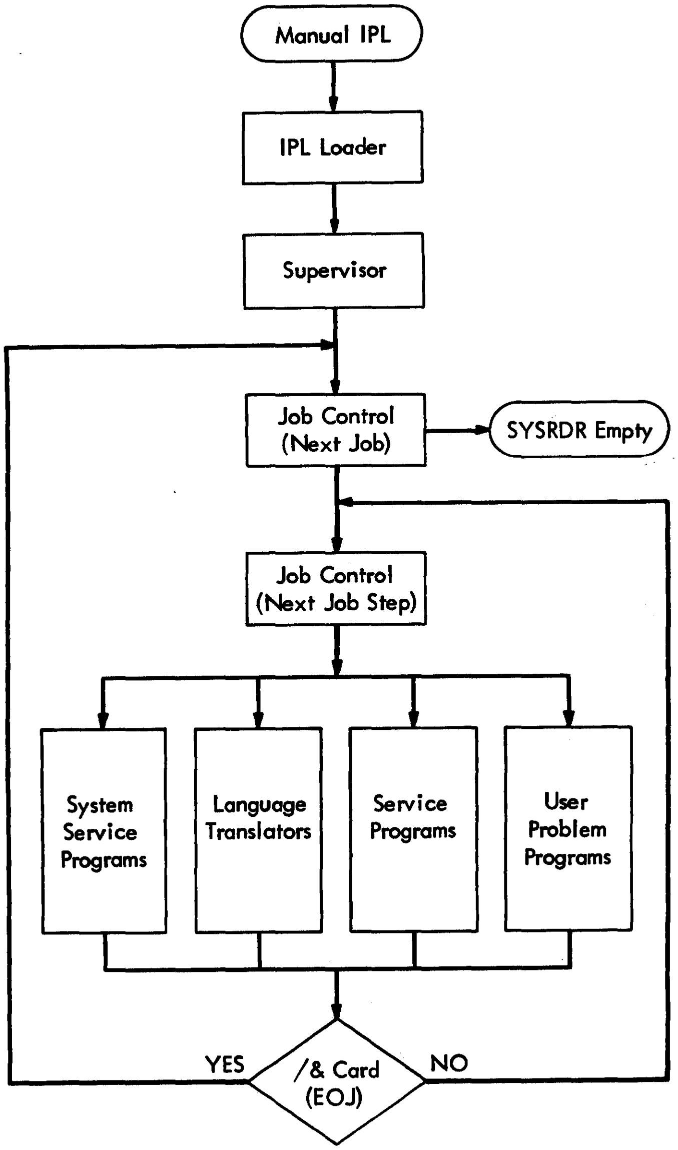


Figure 17. Disk and Tape Operating System Flow

The processing programs consists of language translators and service programs. The group of processing programs can be expanded by adding user-written problem programs (Figure 18). The control program, which constitutes the framework of the Disk and Tape Operating Systems, consists of three components: supervisor, job control, and initial program loading (IPL) loader. These components prepare and control the execution of all processing programs and problem programs within the system. The system service programs consists of the linkage editor and the librarian. These programs are used in generating the systems, and creating, editing, and maintaining the libraries in the resident systems.

For disk and tape operating systems having a main storage equal to or in excess of 32K, multiprogramming support is available. There are two types of problem programs in multiprogramming: background and foreground.

Foreground programs do not execute from a stack, but are expicitly initiated by the operator. Background and foreground programs begin and end asynchronously from each other. The systems are capable of concurrently operating one background and one or two foreground programs.

The structure of the control program, system service programs, and processing programs is depicted in Figure 18.

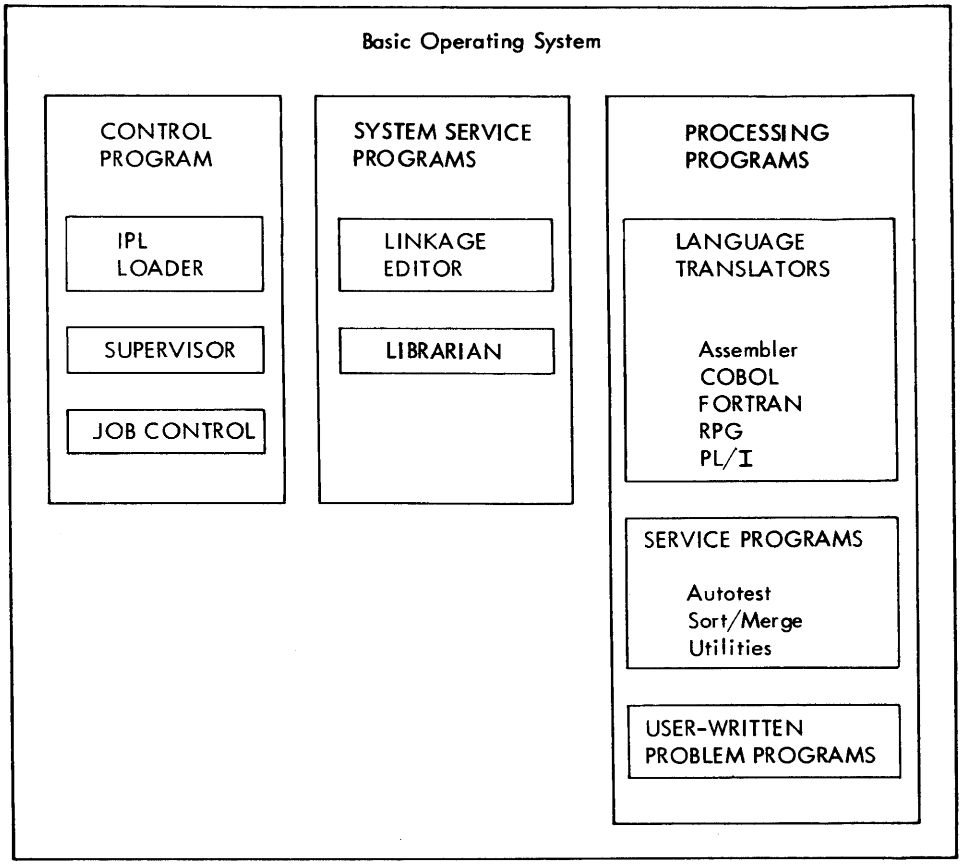


Figure 18. Disk and Tape Operating Systems

FUNCTIONAL RELATIONSHIPS OF THE SYSTEM COMPONENTS

To make full use of the Disk and Tape Operating Systems COBOL, the programmer should be familiar with:

1. Each system component

2. The function of each component

3. The interaction of the components in the total system.

The remainder of this section is intended to acquaint the reader with each of these three. For a thorough understanding of these components, the user should familiarize himself with the publication, System Control and System Service Programs, listed on the cover of this manual.

CONTROL PROGRAM

To provide optimum operating efficiency, some programmed control over the operation of the system is required. Without such programmed control, the system is frequently idle and requires the intervention of an operator to locate and load successive programs in addition to performing other required setup functions, such as changing tape reels. An orderly and efficient flow of jobs through the system is maintained by using a control program that provides the job-to-job transition.

Disk and Tape Operating Systems contain such a control program. It provides automatic transition from program phase to program phase within a processing program, and from processing program to processing program within a continuous processing environment. Once the system has been initialized, job after job can be included in the input job stream.

The component parts of the control program are:

- IPL Loader

- Supervisor

- Job Control.

IPL LOADER

Operation of IBM System/360 Disk and Tape Operating Systems is initiated through an initial program load (IPL) procedure from the resident system. The IPL loader is loaded into main storage from the resident system simply by selecting the address of the unit in the load-unit switches on the system console, and pressing the load key. The loader then reads the nucleus of the supervisor into low main storage from the resident system. After successfully reading in the supervisor nucleus, the IPL loader performs certain initializing and housekeeping functions, then, control is transferred to the supervisor, which uses the system loader routine to issue a call for job control.

SUPERVISOR

The supervisor is the control program that operates with the problem programs. It consists of two types of routines: Permanent, which are loaded into main storage during the IPL process and remain there throughout system operation until main storage is cleared, and transient routines, which remain on the disk or tape system until needed, and are then retrieved and loaded into a common transient area. For example: The interruption processing routines are permanent, and the OPEN and CLOSE routines are transient.

During execution of a processing program, control alternates between the processing program and the supervisor.

In a multiprogramming environment, control always passes to the program with the highest priority. Priority is assigned according to classification of programs as follows:

1. Supervisor (highest priority)

2. Operator communication routine

3. Foreground-one program

4. Foreground-two program

5. Background program (lowest priority)

The area occupied by the background program begins just past the transient area. The background program area must be a minimum of 10K bytes. (Disk Operating System COBOL requires a minimum of 14K bytes to perform its functions.) Following the background program area is the foreground-two program area. This area must be defined in increments of 2K. (Storage protection requires that main storage be divided into blocks of 2K bytes.) Following the foreground-two program area is the foreground-one program area. The foreground-one area must also be defined in increments of 2K. The minimum size of a foreground area is 0K; the maximum is 510K. Figure 19 illustrates the relationship between the supervisor and the problem program areas.

JOB CONTROL

The job control program provides job-to-job transition within Disk and Tape Operating Systems. It also is called into main storage to prepare each job step to be run. (One or more programs can be executed within a single job. Each such execution is called a job step.) It performs its functions between job steps and is not present while a problem program is being executed.

SYSTEM SERVICE PROGRAM

The system service programs provide the functions of generating the system, creating and maintaining the library sections, and editing programs on disk or tape before execution. Minimum systems can be built that do not include the system service program. Such minimum systems will require disk or tape residence.

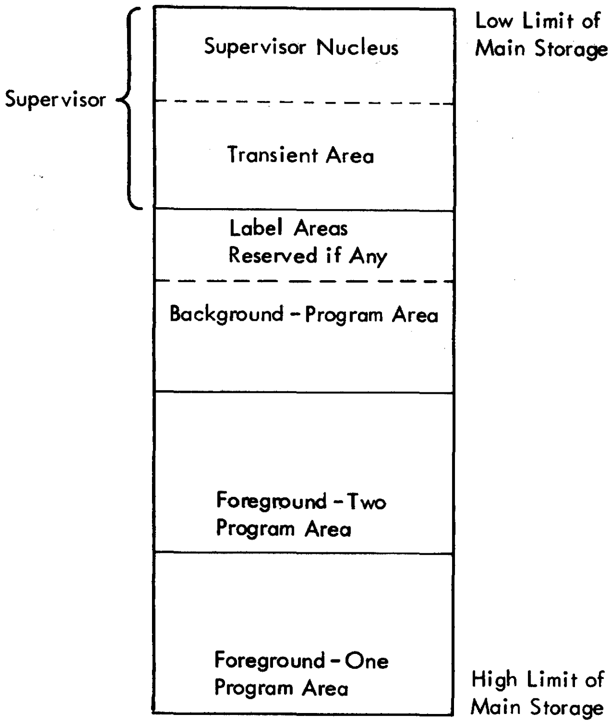

Supervisor
Supervisor Nucleus
Transient Area
Label Areas Reserved if Any
Background - Program Area
Foreground - Two Program Area
Foreground - One Program Area
Low Limit of Main Storage
High Limit of Main Storage

Figure 19. Major Divisions of Main Storage for Disk and Tape Operating System

The system service programs are:

1. Linkage Editor
2. Librarian
   a. Core Image Library
   b. Source Statement Library
   c. Relocatable Library

LINKAGE EDITOR

Disk Operating System

All programs executed in the Disk Operating System environment must be edited by the linkage editor. The linkage editor reads the relocatable output of the language translators and edits it into executable, non-relocatable programs in the core image library. Once a program has been edited, it can be executed immediately, or it can be cataloged as a permanent entry in the core image library. When a program has been cataloged in the core image library, the linkage editor is no longer required for that program. The program is run as a distinct job step and is loaded directly from the resident pack by the system loader.

Tape Operating System

Programs executed in the Tape Operating System environment are processed the same as those residing in the disk system environment with the following exception.

All programs to be linkage edited in the Tape Operating System are written onto SYSLNK tape. This tape is the input to linkage editor. After all programs are linked and edited, these programs are written back onto SYSLNK for execution.

LIBRARIAN

This is actually a group of programs used for maintaining tape libraries, and providing printed and punched output from the libraries.

The three libraries are:

Core image library
Source statement library
Relocatable library.

Core Image Library

All permanent programs in the system (IBM-supplied and user programs) are loaded from this library by the system loader routine of the supervisor. The core image library is required for each disk or tape resident system. The core image library contains any number of programs, each of which is edited to run with the resident supervisor. Each program is made up of one or more separate phases (defined in Section I). Associated with each phase is a header record that contains a complete description of the phase.

Source Statement Library

This library is used to store IBM-supplied macro definitions and user-defined source statements (such as COBOL data definitions) on the disk or tape built to provide extended program compilation capability. The source statement library is not required for operating a system. The purpose of the source-statement library is to provide such information as standard installation file descriptions to reduce the required amount of coding for each individual program. The source statement library contains any number of books. Each book in the source statement library is made up of a sequence of source language statements. Each book is classified as belonging to a specific sub-library. Sub-libraries are defined for two programming languages, assembler and COBOL. Books entered into the source statement library are copied into the source program when the COPY or INCLUDE function of a COBOL language compiler are used.

Associated with the source statement library are a source statement directory and a header record for each book in the library. Entries in the directory define each sub-library and the books associated with each sub-library. Associated with each book is a header record that contains a complete description of the book.

Relocatable Library

This library is used to store object modules (defined in Section I) that can be used for subsequent linkage with other program modules. A module also can be a complete program. The relocatable library is not required for operating a system.

The purpose of the relocatable library is to allow the user to maintain frequently used routines in residence and combine them with other modules without requiring recompilation. The routines from the relocatable library are edited on SYSLNK by the linkage editor.

The relocatable library contains any number of modules. Each module is a complete object deck in the relocatable format.

Librarian Functions

Each system-residence device contains one to three libraries: core image (required), relocatable, and source statement. As their names imply, executable programs (core-image format) are stored in the core image library; relocatable object decks are stored in the relocatable library; and, source language routines are stored in the source statement library.

The librarian is a group of routines that maintain and service the libraries of the system. The maintenance routine in both systems, disk and tape, provides for cataloging (adding) and deleting. In the disk system, the maintenance function also provides the ability to rename, condense, and reallocate. The service routines provide for displaying (printing) and punching elements of the relocatable and source statement libraries. Since the system does not load absolute card decks, no facility is provided for punching or displaying elements of the core image library. Both systems provide for copying any or all of the library components.

PROCESSING PROGRAMS

Three types of processing programs are contained in the Disk and Tape Operating Systems:

- Language translators
- Service programs
- User programs

Language Translators

Several translators are available for translating user-written source programs into relocatable object programs. All of the translators requested by the user are stored in the Relocatable Library when the system disk or tape arrives at an installation. The assembler (required for system generation) is in both the Relocatable Library and the Core Image Library. When the system is tailored at the installation, the desired translators must be cataloged into the core image library. All translators take advantage of the IOCS included in the Disk and Tape Operating Systems.

The language translators for the disk/tape system are: assembler, COBOL, FORTRAN, RPG (report program generator), and PL/I.

Service Programs

The service programs are initially contained in the relocatable library. They must be cataloged into the core image library before being executed.

1. Autotest: The autotest program provides debugging functions within the Disk and Tape Operating Systems. The COBOL user would not be concerned with autotest because COBOL makes available to the user its own debugging packet in a form familiar to COBOL users. The COBOL debugging language is described in the publication COBOL Language Specifications, listed on the cover of the manual. For a complete description of the autotest program, see IBM System/360 Disk and Tape Operating Systems, Autotest Specifications, Form C24-3441.

2. Sort/Merge: The IBM System/360 Disk and Tape Operating Systems Tape Sort/Merge Programs enable the user to sort files of random records, or merge multiple files of sequenced records, into one sequential file. The program is designed to satisfy the sorting and merging requirements of disk or tape-oriented installations with 16K and above bytes of main storage. For a basic understanding of the use of this program see IBM System/360 Disk and Tape Operating Systems Tape Sort/Merge Program Specifications, Form C24-3438, or IBM System/360 Disk Operating System, Sort/Merge Program Specifications, Form C24-3444.

3. Utility Programs: These programs provide for file-to-file transition for data files of almost any format. They are generalized programs and must be tailored by control-statement information to fit specific data files. For a complete description of the utility programs, see IBM System/360 Disk and Tape Operating Systems Utility Programs Specifications, Form C24-3465.

User Programs

These programs, supplied by the user, are programs written in the language of the user's choice.

INSTALLATION--TAILORED SYSTEMS

The value of the Disk and Tape Operating Systems depends to a large extent on the specific requirements of an installation and how closely the services provided by the system meet such requirements. If a facility provided by a system is not required for a particular application, it need not use storage space. Therefore, the disk and tape systems are designed to enable individual facilities to be selected on the basis of whether they are required at a particular installation or for a particular application within an installation.

The optimum system for a given application or group of applications is influenced by a number of factors. Foremost among these factors is the system configuration.

The larger systems, those with a disk or tape drive allocated primarily to system residence, permit the user to take full advantage of the complete Disk and Tape Operating Systems.

For a complete description of the features included in a system refer to the IBM publication System Generation and Maintenance listed on the cover of this manual.

APPENDIX A. CONSIDERATIONS WHEN USING ASSEMBLER WITH COBOL FOR OVERLAYS

This appendix contains:

- An example of a printout of an assembler routine effecting overlays specified by a COBOL Disk and Tape Operating Systems program.
- Explanations of the functions performed by the assembler overlay subroutine instructions. The explanations are keyed to the instructions in the listing.
- Information needed to prepare and use subprograms written in assembler language with a main program written in COBOL.

ASSEMBLER ROUTINE FOR EFFECTING OVERLAYS

The following overlay subroutine is an example and is governed by the following restrictions:

1. The example is a suggested technique, and not the only technique.
2. It can be used for assembler overlays if statement 30 is deleted and if the user has a desired entry point in his end card.
3. The subroutine cannot be used for entry points other than at the first instruction of the procedure division. A suggested technique for diverse entry points is a table lookup employing V-type constants.
4. Deletion of statement 30, i.e., LA 15,40(15) could result in looping or a process error in the subprogram.
5. The number of bytes of initialization generated by the compiler (i.e., the 40 in statement 30 of the example) may change in subsequent modification of the compiler. This number was 32 in version 1 of DOS/TOS COBOL.

```
STMNT   SOURCE STATEMENT

0001    OVRLAY START 0
               ENTRY OVRLAY
0002
0003    *  AT ENTRY TIME
0004    *  .  R1 = POINTER TO ADCON LIST OF USING ARGUMENTS FIRST ARGUMENT
0005    *        IS PHASE OR SUBROUTINE NAME, MUST BE 8 BYTES
0006    *     R13 = ADDRESS OF SAVE AREA
0007    *     R14 = RETURN POINT OF CALLING PROGRAM
0008    *     R15 = ENTRY POINT OF OVERLAY PROGRAM
0009    *  AT EXIT
0010    *     R1 = POINTER TO SECOND ARGUMENT OF ADOCN LIST OF USING ARGUMENTS
0011    *     R14 = RETURN POINT OF CALLING PROGRAM--NOT THIS PROG
0012    *     R15 = ENTRY POINT OF PHASE OR SUBPROGRAM
0013    *  R0 IS DESTROYED BY THIS ROUTINE
0014           USING *,15
0015           ST    1,SAVE
0016           L     1,0(1)                R1 = ADDRESS OF PHASE NAME
0017           CLC   0(8,1),CORSUB         IS IT IN CORE
0018           BE    SUBIN                      YES
0019           MVC   CORSUB(8),0(1)             NO, CORSUB = PHASE NAME
0020           SR    0,0                   R0 = 0
0021    *                                  LOAD REQUIRES R0 = 0 IF LOAD ADDRESS
0022    *                                       ISNT SPECIFIED, R1 = ADDRESS OF
0023    *                                       PHASE NAME.  R1 = PHASE ENTRY
0024    *                                       UPON RETURN
```

```
STMNT   SOURCE STATEMENT

0025            SVC    4                     LOAD PHASE
0026            ST     1,ASUB                ASUB=ENTRY POINT OF PHASE
0027    SUBIN   L      1,SAVE                R1 = POINTER TO SECOND ADCON OF
0028            LA     1,4(1)                    USING LIST--BYPASSES PHASE NAME
0029            L      15,ASUB               R15 = ENTRY POINT OF PHASE
0030            LA     15,40(15)             BYPASS COBOL INITIALIZATION IN SUBPROGRAM
0031            BR     15                    BRANCH TO SUBROUTINE, RETURN WILL BE
0032    *                                         TO PROGRAM WHICH CALLED OVRLAY
0033            DS     0F
0034    SAVE    DC     4X'FF'                REGISTER SAVE AREA
0035    ASUB    DC     4X'FF'                ADDRESS OF SUBROUTINE
0036    CORSUB  DC     8X'FF'                NAME OF SUBROUTINE IN CORE
0037            END
```

FUNCTIONS OF OVERLAY ROUTINE INSTRUCTIONS

The instructions of the overlay routine perform the following functions:

0015 - Saves the address of the PARAMETER LIST
0016 - Loads the address of the PARAMETER LIST
0017 - Checks to see if program is already in overlay area
0018 - If it is, OVERLAY branches directly to subprogram
0025 - OVERLAY the issues OVERLAY CALL
0026 - Saves 1st address of overlaying and subprogram
0027 - Loads address of parameter table
0028 - Indexes and loads address of first parameter
0029, 0030, 0031 - Branches to subprogram to execute procedural steps
0033,37 - Defines storage, defines constants and end of routine instruction.

ASSEMBLER LANGUAGE SUBPROGRAMS

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 20, 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.

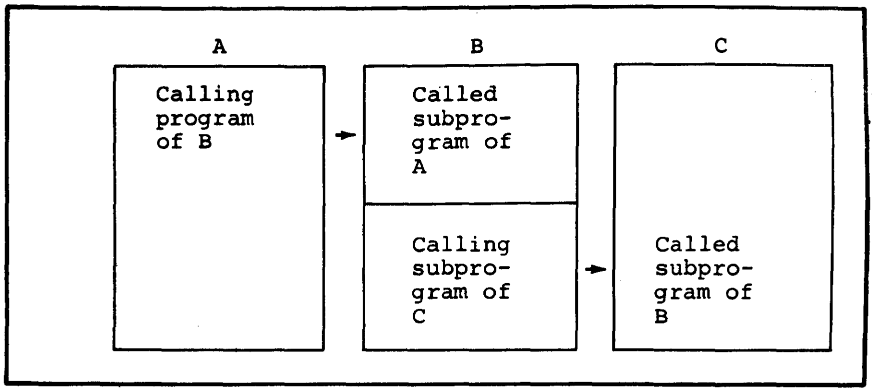


Figure 20. Called and Calling Programs

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 Disk and Tape Operating Systems have 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 Disk and Tape Operating Systems have assigned functions to certain registers used in linkages. The function of each linkage register is shown in Figure 21.

REGISTER NUMBER	REGISTER NAME	FUNCTION
1	Argument List Register	Address of the argument list passed to the called subprogram.
13	Save Area 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.

Figure 21. Linkage Registers

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 22 shows the layout of the save area and the contents of each word.

AREA (word 1)	This word is a part of the standard linkage convention established under the disk and tape operating systems. 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 22. Save Area Layout and Word Contents

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.

Example

The linkage conventions used by an assembler subprogram that calls another subprogram are shown in Figure 23. 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.

```
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 con-
*                                 trol 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 23. Sample Linkage Routines Used with a Calling Subprogram

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 23. If the assembler subprogram uses any registers, it must save them. Figure 24 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 24. 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 25 for that shown in Figure 23.

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 descrepancies 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.

```
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 25. Sample In-line Parameter List

APPENDIX B. TABLE OF REFERENCE FORMATS FOR DISK AND TAPE OPERATING SYSTEMS COBOL

```
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.                                                   ]
[FILE-CONTROL.  [COPY library-name.]                                     ]
[ SELECT file-name [COPY library-name.]                                  ]
[                                                                        ]
[                          (DIRECT-ACCESS)                               ]
[  ASSIGN TO external-name{ UTILITY       }device-number UNIT[S]         ]
[                          (UNIT-RECORD  )                               ]
[           { NO      }                                                  ]
[  [RESERVE { integer } ALTERNATE AREA[S]                                ]
[                                                                        ]
[  [ACCESS IS{SEQUENTIAL}]                                               ]
[            {RANDOM    }                                                ]
[                                                                        ]
[  [ORGANIZATION IS {INDEXED}]                                           ]
[                   {DIRECT }                                            ]
[                                                                        ]
[  [SYMBOLIC KEY IS data-name]                                           ]
[  [ACTUAL KEY IS data-name]                                             ]
[  [RECORD KEY IS data-name]                                             ]

I-O CONTROL.

[SAME AREA FOR file-name-1 file-name-2 [file-name-3...].]

                                       {REEL}
[RERUN ON external-name EVERY END OF {UNIT} of file-name.]

[APPLY overflow-name to FORM-OVERFLOW ON file-name.]

[APPLY WRITE-ONLY ON file-name.....]

[APPLY RESTRICTED SEARCH OF integer TRACKS ON file-name.....]


DATA DIVISION.

FILE SECTION.

FD  file-name   [COPY library-name.]

  [BLOCK CONTAINS integer    {CHARACTERS} ]
                             {RECORDS   }
```

```
                            (U)
   [RECORDING MODE IS      {F}]
                            (V)

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

         (RECORD IS  ) (STANDARD  )
   LABEL {           } {OMITTED   }
         (RECORDS ARE) (data-name )

DATA  (RECORDS ARE) record-name...
      (RECORDS IS )

Record Description Entry.

WORKING-STORAGE SECTION.

Record Description entries

LINKAGE SECTION.

Record Description entries

level-number   (data-name)  [REDEFINES data-name-2]   [COPY library-name.]
               (FILLER   )
 [                                     ]
 [               (alpha-form    )      ]
 [               (an-form       )      ]
 [ PICTURE IS    {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


DECLARATIVES.
 {section-name SECTION.  USE-sentence.
 {paragraph-name.   sentence...   .}  ... }   ...
END DECLARATIVES.

USE FOR CREATING   [BEGINNING]   LABELS ON OUTPUT  file-name...
                   [ENDING   ]

USE FOR CHECKING   [BEGINNING]   LABELS ON INPUT file-name...
                   [ENDING   ]

USE AFTER STANDARD ERROR PROCEDURE ON file-name.
```

Conditionals.

IF Statement.

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

Relation Test.

```
                                                  ( >            )  ( data-name-2    )
                                                  ( <            )  (                )
{ data-name-1                 }                   ( =            )  ( arithmetic     )
{ arithmetic-expression-1     }   IS [NOT]        { GREATER THAN }  ( expression-2   )
{ figurative-constant-1       }                   ( LESS THAN    )  (                )
{ literal-1                   }                   ( EQUAL TO     )  { figurative     }
                                                                    ( constant-2     )
                                                                    (                )
                                                                    ( literal-2      )
```

Sign Test.

```
{ data-name             }                 { POSITIVE }
{ arithmetic-expression }    IS [NOT]     { ZERO     }
                                          { NEGATIVE }
```

Class Test.

```
                           { NUMERIC    }
{ data-name IS [NOT] }     { ALPHABETIC }
```

Condition Name Test.

```
[NOT] condition-name
```

Overflow Test.

```
[NOT] overflow-name
```

Open and Close Statements.

```
        ( 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]]} ...]               )

                      [UNIT]    [      { NO REWIND } ]
CLOSE  { file-name    [REEL]    [ WITH { LOCK      } ] }...
```

Input/Output Verbs

```
READ file-name RECORD [INTO data-name] AT END
     imperative statement...
```

```
READ file-name RECORD [INTO data-name]

       (AT END      )    imperative statement...
       (INVALID KEY )

WRITE record-name [FROM data-name-1]
       [INVALID KEY imperative statement...]

WRITE record-name [FROM data-name-1]

     [AFTER ADVANCING (data-name-2) LINES]
                      (integer    )
```

permissible values for data-name-2

Value	Interpretation
b (blank)	single spacing
0	double spacing
-	triple spacing
+	suppress spacing
1 through 9	skip to channels 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 2540 and P1 or P2 on the IBM 2540

Permissible Integer

0 - skip to next-page
1 - skip 1 line
2 - skip 2 lines
3 - skip 3 lines

```
REWRITE record-name [FROM data-name]
       [INVALID KEY imperative-statement...]

         (data-name)          [UPON CONSOLE ]
DISPLAY  (literal  )    ...   [UPON SYSPUNCH]

ACCEPT data-name   [FROM CONSOLE]
```

Data Manipulation Verbs

```
      (data-name-1)
MOVE  (literal    )  TO data-name-2  ...
```

Option 1

```
                              (ALL        )
EXAMINE data-name TALLYING    (LEADING    )   'character-1'
                              (UNTIL FIRST)

       [REPLACING BY 'character-2']
```

Option 2

```
                               (ALL        )
                               (LEADING    )
EXAMINE data-name REPLACING    (UNTIL FIRST)  'character-1'
                               (FIRST      )

       BY 'character-2'
```

```
TRANSFORM data-name-3 CHARACTERS

         { figurative-constant-1 }          { figurative-constant-2 }
FROM     { non-numeric-literal-1 }    TO    { non-numeric-literal-2 }
         { data-name-1           }          { data-name-2           }
```

Arithmetic Verbs

```
      { numeric-literal        }                 
ADD   { floating-point-literal }   ...    { TO     }    data-name-n
      { data-name-1            }          { GIVING }

       [ROUNDED]   [ON SIZE ERROR imperative-statement...]


        { data-name-1              }
SUBTRACT{ numeric-literal-1        }       ...
        { floating-point-literal-1 }

        { data-name-m      [GIVING data-name-n]                }
    FROM{ numeric-literal-m  GIVING data-name-n                }
        { floating-point-literal-m  GIVING data-name-n         }

       [ROUNDED]   [ON SIZE ERROR imperative statement...]


         { data-name-1              }
MULTIPLY { numeric-literal-1        }
         { floating-point-literal-1 }

         { data-name-2      [GIVING data-name-3]               }
      BY { numeric-literal-2  GIVING data-name-3               }
         { floating-point-literal-2 GIVING data-name-3         }

       [ROUNDED] [ON SIZE ERROR imperative statement...]

         { data-name-1              }
DIVIDE   { numeric-literal-1        }
         { floating-point-literal-1 }

         { data-name-2  [GIVING data-name-3]                   }
  INTO   { numeric-literal-2 GIVING data-name-3                }
         { floating-point-literal-2  GIVING data-name-3        }

       [ROUNDED] [ON SIZE ERROR imperative statement...]


                                         { data-name-2            }
COMPUTE data-name-1   [ROUNDED] =        { numeric-literal        }
                                         { floating-point-literal }
                                         { arithmetic-expression  }

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

Procedure Branching Statements.

```
         STOP { RUN     }
              { literal }
```

```
Option 1

GO TO [procedure-name]

Option 2

GO TO procedure-name-1 [procedure-name-2...] DEPENDING ON data-name

ALTER  {procedure-name-1 TO PROCEED TO procedure-name-2 }    ...

Option 1

PERFORM procedure-name-1 [THRU procedure-name-2]

Option 2

PERFORM procedure-name-1 [THRU procedure-name-2]  {integer   }  TIMES
                                                  {data-name }

Option 3

PERFORM procedure-name-1 [THRU procedure-name-2]
           UNTIL test-condition


Option 4

PERFORM procedure-name-1  [THRU procedure-name-2]
   VARYING data-name-1    FROM  {numeric-literal-2 }
                                {   data-name-2    }

   BY  {numeric-literal-3}      UNTIL test-condition-1
       {data-name-3      }

   [AFTER data-name-4]    FROM   {numeric-literal-4 }
                                 {    data-name-5   }

   BY   {numeric-literal-6 }     [UNTIL test-condition-2]
        {   data-name-6    }

   [AFTER data-name-7     FROM  { numeric-literal-8 }
                                {    data-name-8    }

   BY  {numeric-literal-9}     UNTIL test-condition-3]
       { data-name-9     }
```

Compiler-Directing Statements.

```
ENTER LINKAGE.
CALL entry-name [USING argument...].
ENTER COBOL.

ENTER LINKAGE.
ENTRY entry-name [USING data-name...].
ENTER COBOL.

ENTER LINKAGE.
RETURN.
ENTER COBOL.
```

EXIT Statement.

```
paragraph-name.  EXIT.
```

NOTE Statement.

```
NOTE comment...
```

Option 1.

```
paragraph-name.  INCLUDE library-name.
```

Option 2.

```
section-name SECTION.  INCLUDE library-name.
```

COPY Statement.

(Within the Input-Output Section):

```
{FILE-CONTROL.}  COPY library-name.
{I-O-CONTROL. }
```

(Within the File-Control paragraph):

```
SELECT file-name COPY library-name.
```

(Within the File Section):

```
FD file-name COPY library-name.
```

(Within a File, Working Storage or Linkage Section):

```
01 data-name COPY library-name.
```

(Within Working Storage or Linkage Section):

```
77 data-name COPY library-name.
```

COBOL Debugging Statements.

TRACE Statement.

```
{READY}
{RESET}  TRACE
```

EXHIBIT Statement.

```
         {NAMED        }
EXHIBIT  {CHANGED NAMED}  {data-name          }
         {CHANGED      }  {non-numeric-literal}     ...
```

On (Count-Conditional) Statement

```
ON integer-1 [AND EVERY integer-2] [UNTIL integer-3]

   { imperative-statement...} [{ELSE       statement...  }]
   { NEXT SENTENCE          } [{OTHERWISE  NEXT SENTENCE }]
```

Debug Packet Statement.

```
1      8
*DEBUG location
```

PERMISSIBLE COMPARISONS

		SECOND OPERAND									
		GR	AL	AN	ED	ID	BI	EF	IF	RP	FC
FIRST OPERAND	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.

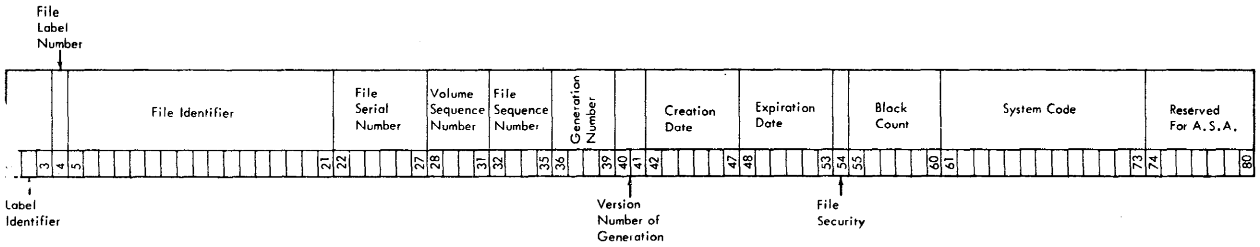
[4] Not permitted with figurative constant QUOTE.

[5] External decimal field must consist of integers.

PERMISSIBLE MOVES

	Receiving Field								
Source 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

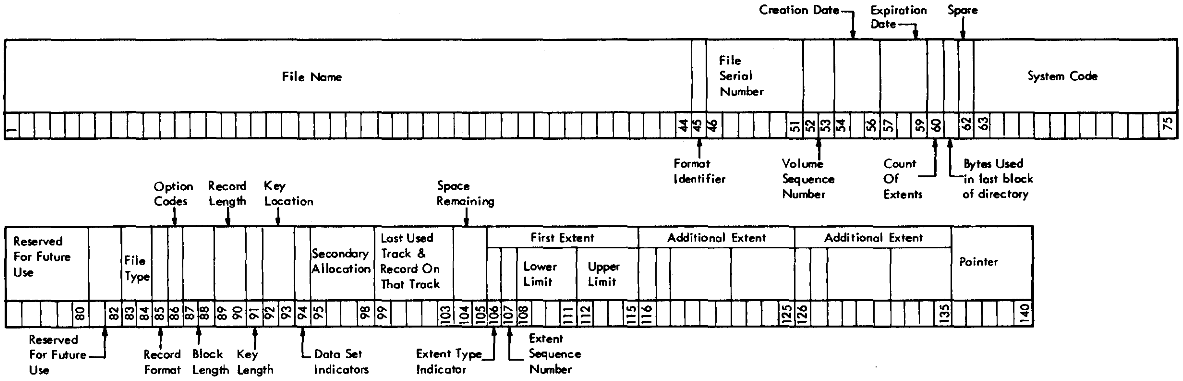
[1] For integers only.



The standard tape file label format and contents are as follows:

FIELD	NAME AND LENGTH	DESCRIPTION
1.	LABEL IDENTIFIER 3 bytes, EBCDIC	identifies the type of label HDR = Header -- beginning of a data file EOF = End of File -- end of a set of data EOV = End of Volume -- end of the physical reel
2.	FILE LABEL NUMBER 1 byte, EBCDIC	Always a 1
3.	FILE IDENTIFIER 17 bytes, EBCDIC	uniquely identifies the entire file, may contain only printable characters.
4.	FILE SERIAL NUMBER 6 bytes, EBCDIC	uniquely identifies a file/volume relationship. This field is identical to the Volume Serial Number in the volume label of the first or only volume of a multi-volume file or a multi-file set. This field will normally be numeric (000001 to 999999) but may contain any six alphameric characters.
5.	VOLUME SEQUENCE NUMBER 4 bytes	indicates the order of a volume in a given file or multi-file set. The first must be numbered 0001 and subsequent numbers must be in proper numeric sequence.
6.	FILE SEQUENCE NUMBER 4 bytes	assigns numeric sequence to a file within a multi-file set. The first must be numbered 0001.
7.	GENERATION NUMBER 4 bytes	uniquely identifies the various editions of the file. May be from 0001 to 9999 in proper numeric sequence.
8.	VERSION NUMBER OF GENERATION 2 bytes	indicates the version of a generation of a file.
9.	CREATION DATE 6 bytes	indicates the year and the day of the year that the file was created: Position Code Meaning 1 blank none 2-3 00-99 Year 4-6 001-366 Day of Year (e.g., January 31, 1965 would be entered as 65031)
10.	EXPIRATION DATE 6 bytes	indicates the year and the day of the year when the file may become a scratch tape. The format of this field is identical to Field 9. On a multifile reel, processed sequentially, all files are considered to expire on the same day.
11.	FILE SECURITY 1 byte	indicates security status of the file. 0 = no security protection 1 = security protection. Additional identification of the file is required before it can be processed.
12.	BLOCK COUNT 6 bytes	indicates the number of data blocks written on the file from the last header label to the first trailer label exclusive of tape marks. Count does not include checkpoint records. This field is used in Trailer Labels.
13.	SYSTEM CODE 13 bytes	uniquely identifies the programming system.
14.	RESERVED 7 bytes	Reserved for American Standards Association (A.S.A.). At present, should be recorded as blanks.

APPENDIX D. STANDARD DASD FILE LABELS, FORMAT 1



Format 1: This format is common to all data files on disk.

FIELD	NAME AND LENGTH	DESCRIPTION
1.	FILE NAME 44 bytes, alphameric EBCDIC	This field serves as the key portion of the file label. It can consist of three sections: 1. File ID is an alphameric assigned by the user and identifies the file. Can be 1 - 35 bytes if generation and version numbers are used, or 1 - 44 bytes if they are not used. 2. Generation Number. If used, this field is separated from File ID by a period. It has the format Gnnnn, where G identifies the field as the generation number and nnnn (in decimal) identifies the generation of the file. 3. Version Number of Generation. If used, this section immediately follows the generation number and has the format Vnn, where V identifies the field as the version of generation number and nn (in decimal) identifies the version of generation of the file. Note: IBM System/360 Disk and Tape Operating Systems compares the entire field against the file name given in the DLAB card. The generation and version numbers are treated differently by Operating System/360.

The remaining fields comprise the DATA portion of the file label:

FIELD	NAME AND LENGTH	DESCRIPTION
2.	FORMAT IDENTIFIER 1 byte, EBCDIC numeric	1 = Format 1
3.	FILE SERIAL NUMBER 6 bytes, alphameric EBCDIC	Uniquely identifies a file/volume relationship. It is identical to the Volume Serial Number of the first or only volume of a multi-volume file. It is the disk pack number identification.
4	VOLUME SEQUENCE NUMBER 2 bytes, binary	Identifies each volume in a multi-volume file. Each volume is relative to the first volume on which the data file resides.
5	CREATION DATE 3 bytes, discontinuous binary	Indicates the year and the day of the year the file was created. It is of the form YDD, where Y signifies the year (0- 99) and DD the day of the year (1- 366).
6	EXPIRATION DATE 3 bytes, discontinuous binary	Indicates the year and the day of the year the file may be deleted. The form of this field is identical to that of Field 5.
7A	EXTENT COUNT 1 byte, binary	Contains a count of the number of extents for this file on this volume. If user labels are used, the count includes the user label track as a separate extent. This field is maintained by the Disk and Tape Operating Systems programs.
7	BYTES USED IN LAST BLOCK OF DIRECTORY 1 byte, binary	Used by Operating System/360 only for partitioned (library structure) data sets. Not used by Disk and Tape Operating Systems.
7C	SPARE 1 byte	Reserved for future use.
8	SYSTEM CODE 13 bytes	Uniquely identifies the programming system.
9	RESERVED 7 bytes	This field is reserved for future use.
10	FILE TYPE 2 bytes	The contents of this field uniquely identify the type of data file: Hex 4000 = Consecutive organization Hex 2000 = Direct-access organization Hex 8000 = Indexed-sequential organization Hex 0200 = Library organization Hex 0000 = Organization not defined in the file label.
11	RECORD FORMAT 1 byte	The contents of this field indicate the type of records contained in the file: Bit Position / Content / Meaning 0 and 1 / 01 / Variable-length records 10 / Fixed-length records 11 / Undefined format 2 / 0 / No track overflow 1 / File is organized using track overflow (Operating System/360 only) 3 / 0 / Unblocked records 1 / Blocked records

FIELD	NAME	DESCRIPTION
		Bit Position / Content / Meaning 4 / 0 / No truncated records 1 / Truncated records in file 5 and 6 / 01 / Control character ASA code 10 / Confrol Character machine code 00 / Control Character not stated 7 / 0 / Records have no keys 1 / Records are written with keys
12	OPTION CODES 1 byte	Bits within this field are used to indicate various options used in building the file. BIT 0 If on, indicates data file was created using Write Validity Check. 1 - 7 unused
13.	BLOCK LENGTH 2 bytes, binary	Indicates the block length for fixed length records or maximum block size for variable length blocks.
14.	RECORD LENGTH 2 bytes, binary	Indicates the record length for fixed length records or the maximum record length for variable length records.
15.	KEY LENGTH 1 byte, binary	Indicates the length of the key portion of the data records in the file.
16.	KEY LOCATION 2 bytes, binary	Indicates the high order position of the data record.
17.	DATA SET INDICATORS 1 byte	Bits within this field are used to indicate the following: BIT 0 If on, indicates that this is the last volume on which this file normally resides. This bit is used by the Disk and Tape Operating Systems DTFSR routine only. None of the other bits in this byte are used by Disk and Tape Operating Systems. 1 If on, indicates that the data set described by this file must remain in the same absolute location on the direct access device. 2 If on, indicates that Block Length must always be a multiple of 8 bytes. 3 If on, indicates that this data file is security protected; a password must be provided in order to access it. 4 - 7 Spare. Reserved for future use.
18.	SECONDARY ALLOCATION 4 bytes, binary	Indicates the amount of storage to be requested for this data file at End of Extent. This field is used by Operating System 360 only. It is not used by Disk and Tape Operating Systems routines. The first byte of this field is an indication of the type of allocation request. Hex code "C2" (EBCDIC "B") indicates bytes, hex code "E3" (EBCDIC "T") indicates tracks, and hex code "C3" (EBCDIC "C") indicates cylinders. The next three bytes of this field is a binary number indicating how many bytes, tracks or cylinders are requested.
19.	LAST USED TRACK AND RECORD ON THAT TRACK 5 bytes discontinuous binary	Indicates the last occupied track in a consecutive file organization data file. This field has the format CCHHR. It is all binary zeros if the last track in a consecutive data file is not on this volume or if it is not consecutive organization.
20.	AMOUNT OF SPACE REMAINING ON LAST TRACK USED 2 bytes, binary	A count of the number of bytes of available space remaining on the last track used by this data file on this volume.
21.	EXTENT TYPE INDICATOR 1 byte	Indicates the type of extent with which the following fields are associated: HEX CODE 00 Next three fields do not indicate any extent. 01 Prime area (indexed Sequential); or Consecutive area, etc., (i.e., the extent containing the user's data records.) 02 Overflow area of an indexed Sequential file. 04 Cylinder index or master index area of an Indexed Sequential file. 40 User label track area 80 Shared cylinder indicator.
22.	EXTENT SEQUENCE NUMBER 1 byte, binary	Indicates the extent sequence in a multi- extent file.
23.	LOWER LIMIT 4 bytes, discontinuous binary	The cylinder and the track address specifying the starting point (lower limit) of this extent component. This field has the format CCHH.
24.	UPPER LIMIT 4 bytes	The cylinder and the track address specifying the ending point (upper limit) of this extent component. This field has the format CCHH.
25 - 28	ADDITIONAL EXTENT 10 bytes	These f elds have the same format as the fields 21 - 24 above.
29 - 32	ADDITIONAL EXTENT 10 bytes	These fields have the same format as fields 21 - 24 above.
33	POINTER TO NEXT FILE LABEL WITHIN THIS LABEL SET 5 bytes, discontinuous binary	The disk address (format CCHHR) of a continuation label if needed to further describe the file. If field 9 indicates Indexed Sequential organization, this field will point to a Format 2 file label within this label set. Otherwise, it points to a Format 3 file label, and then only if the file contains more than three extent segments. This field contains all binary zeros if no additional file label is pointed to.

APPENDIX E. LIST OF OPTIONS FOR DISK AND TAPE OPERATING SYSTEMS, JOB CONTROL OPTION STATEMENT

The options that can appear in the operand field of the OPTION statement are as follows. Selected options can be in any order. Options are reset to the standards established at system generation time upon encountering a JOB and a /& statement.

LOG	Causes the listing of columns 1-80 of all control statements on SYSLST. Control statements are not listed until a LOG option is encountered. Once a LOG options statement is read, logging continues from job-step to job-step until a NOLOG option is encountered or until either the JOB or /& control statement is encountered.
NOLOG	Suppresses the listing of all control statements on SYSLST except JOB and /& statements until a LOG option is encountered.
DUMP	Causes a dump of the registers and main storage to be output on SYSLST in the case of an abnormal program end (such as program check).
NODUMP	Suppresses the DUMP option.
LINK	Indicates that the object module is to be linkage edited after compilation or assembly. When the LINK or CATAL option is used, the output of the compiler is written on SYSLNK. The LINK option must always precede an EXEC LNKEDT statement containing a compiler step.
NOLINK	Suppresses the LINK option. The compiler can also suppress the LINK option if the problem program contains an error that would preclude the successful execution of the problem program. An EXEC statement with a blank operand also suppresses the LINK option.
DECK	Causes the compiler to output object modules on SYSPCH. If LINK is specified, the DECK option is ignored.
NODECK	Suppresses the DECK option.
LIST	Causes the compiler to write source statements on SYSLST.
NOLIST	Suppresses the LIST option.
LISTX	Causes the compiler to write the procedure division map in hexadecimal on SYSLST.
NOLISTX	Suppresses the LISTX option.
ERRS	Causes the compiler to write the diagnostics related to the source program on SYSLST.
NOERRS	Suppresses the ERRS option.
CATAL	Causes the cataloging of a phase or program in the core image library at the completion of a linkage-editor run. CATAL also causes the LINK option to be set.
48C	Specifies the 48-character set on SYSIPT (for PL/I).

60C Specifies the 60-character set on SYSIPT (for PL/I).

MINSYS (Tape only) Causes the linkage editor to output minimal modules for later runs on systems when linkage editing on systems greater than 16K.

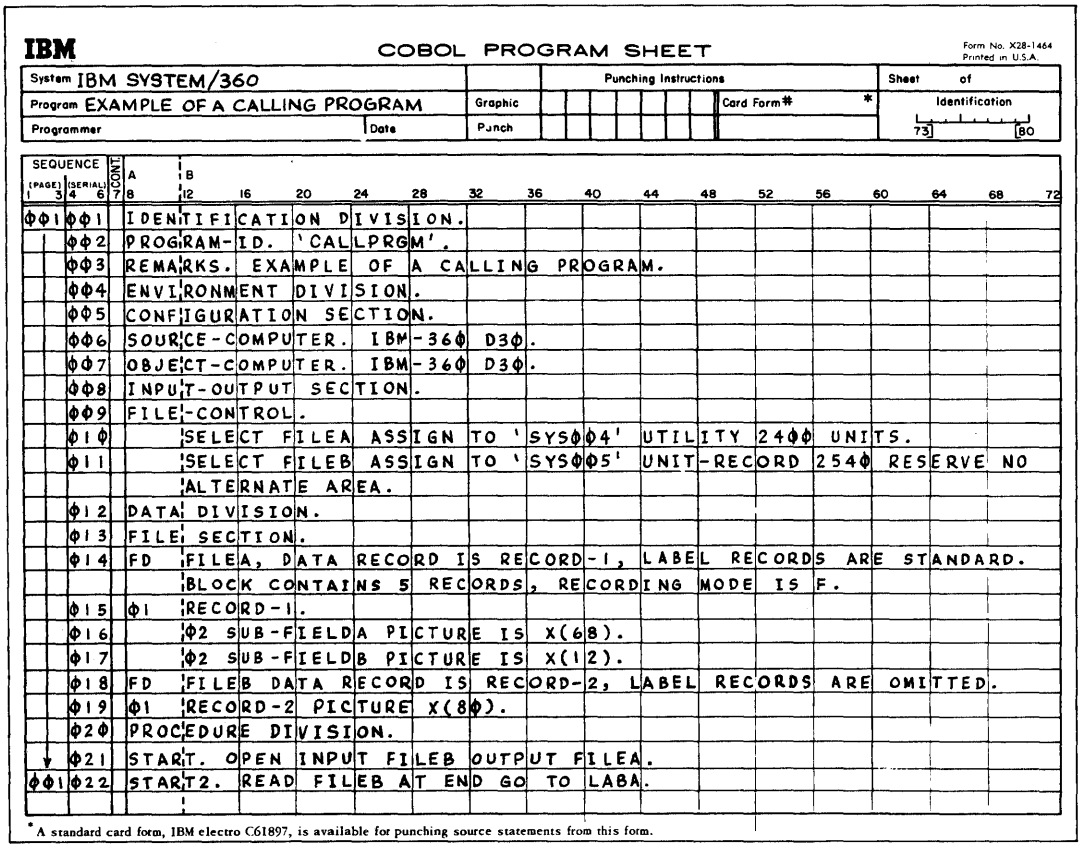
GO (Tape only) Indicates that a linkage-edited program exists on SYSLNK. The program either can be cataloged in the core image library or executed. To catalog the program, specify GO, CATAL in the OPTION statement. To execute the program, specify GO in the OPTION statement and follow it with an EXEC statement with a blank operand. When GO is specified, job control does not open SYSLNK or check the content of SYSLNK.

STDLABEL (Disk only) Causes all disk labels submitted after this point to be written on the standard label track. Reset to USRLABEL option at end-of-job step.

USRLABEL (Disk only) Causes all disk labels submitted after this point to be written at the beginning of the user label track.

SYM Causes the compiler to output the data division map on SYSLST.

This appendix contains two sample COBOL programs. Figure 26 is a calling program, the other, Figure 27, is a subprogram which is linked by the calling program. The linkage subprogram illustrated need not be a COBOL program. However, COBOL assumes option 2 of the standard CALL, SAVE, and RETURN macros.



IBM COBOL PROGRAM SHEET

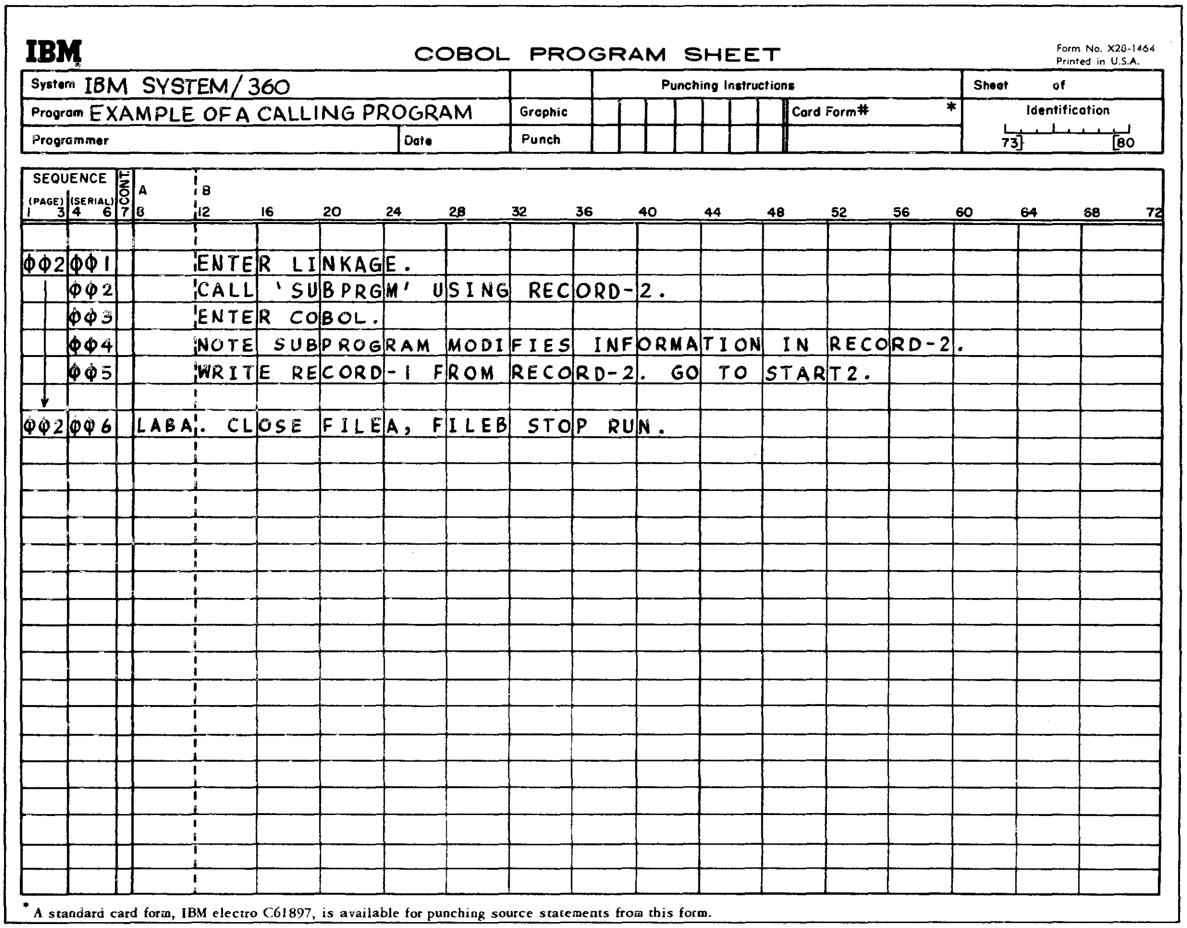
Form No. X28-1464
Printed in U.S.A.

System IBM SYSTEM/360		Punching Instructions	Sheet of
Program EXAMPLE OF A CALLING PROGRAM	Graphic	Card Form# *	Identification
Programmer Date	Punch		73 80

```
SEQUENCE    A   B
(PAGE)(SERIAL) CONT.
1   3 4   6 7 8   12  16  20  24  28  32  36  40  44  48  52  56  60  64  68  72
ϕϕ1ϕϕ1 IDENTIFICATION DIVISION.
   ϕϕ2 PROGRAM-ID. 'CALLPRGM'.
   ϕϕ3 REMARKS. EXAMPLE OF A CALLING PROGRAM.
   ϕϕ4 ENVIRONMENT DIVISION.
   ϕϕ5 CONFIGURATION SECTION.
   ϕϕ6 SOURCE-COMPUTER. IBM-36ϕ D3ϕ.
   ϕϕ7 OBJECT-COMPUTER. IBM-36ϕ D3ϕ.
   ϕϕ8 INPUT-OUTPUT SECTION.
   ϕϕ9 FILE-CONTROL.
   ϕ1ϕ     SELECT FILEA ASSIGN TO 'SYSϕϕ4' UTILITY 24ϕϕ UNITS.
   ϕ11     SELECT FILEB ASSIGN TO 'SYSϕϕ5' UNIT-RECORD 254ϕ RESERVE NO
           ALTERNATE AREA.
   ϕ12 DATA DIVISION.
   ϕ13 FILE SECTION.
   ϕ14 FD  FILEA, DATA RECORD IS RECORD-1, LABEL RECORDS ARE STANDARD.
           BLOCK CONTAINS 5 RECORDS, RECORDING MODE IS F.
   ϕ15 ϕ1  RECORD-1.
   ϕ16     ϕ2 SUB-FIELDA PICTURE IS X(68).
   ϕ17     ϕ2 SUB-FIELDB PICTURE IS X(12).
   ϕ18 FD  FILEB DATA RECORD IS RECORD-2, LABEL RECORDS ARE OMITTED.
   ϕ19 ϕ1  RECORD-2 PICTURE X(8ϕ).
   ϕ2ϕ PROCEDURE DIVISION.
   ϕ21 START. OPEN INPUT FILEB OUTPUT FILEA.
ϕϕ1ϕ22 START2. READ FILEB AT END GO TO LABA.
```

*A standard card form, IBM electro C61897, is available for punching source statements from this form.

Figure 26. Example of a Calling Program (Part 1 of 2)



IBM COBOL PROGRAM SHEET

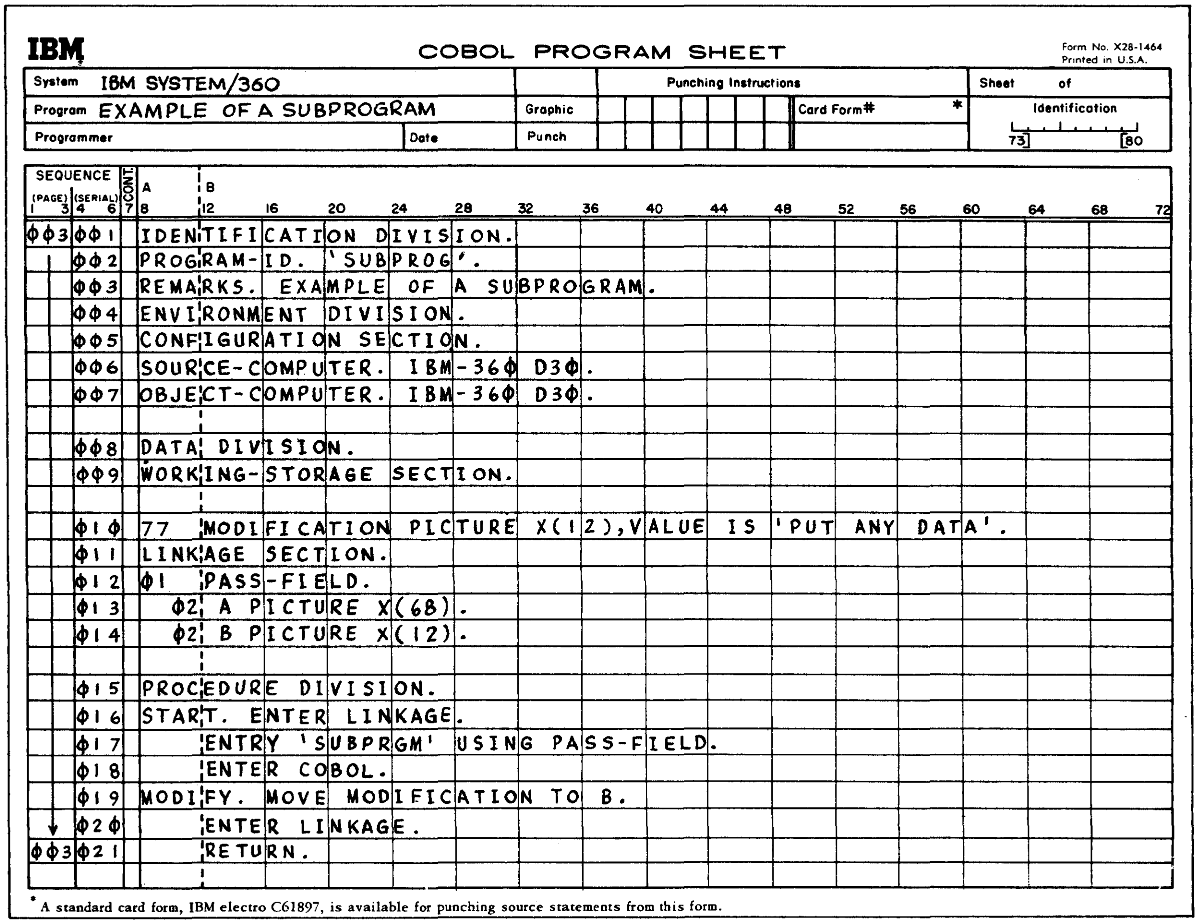
Form No. X28-1464
Printed in U.S.A.

System IBM SYSTEM/360		Punching Instructions	Sheet of
Program EXAMPLE OF A CALLING PROGRAM	Graphic	Card Form # *	Identification 73 80
Programmer / Date	Punch		

```
SEQUENCE  C A   B
(PAGE)(SERIAL)
1   3 4   6 7 8   12
ØØ2ØØ1     ENTER LINKAGE.
   ØØ2     CALL 'SUBPRGM' USING RECORD-2.
   ØØ3     ENTER COBOL.
   ØØ4     NOTE SUBPROGRAM MODIFIES INFORMATION IN RECORD-2.
   ØØ5     WRITE RECORD-1 FROM RECORD-2. GO TO START2.

ØØ2ØØ6 LABA. CLOSE FILEA, FILEB STOP RUN.
```

* A standard card form, IBM electro C61897, is available for punching source statements from this form.

Figure 26. Example of a Calling Program (Part 2 of 2)



IBM COBOL PROGRAM SHEET

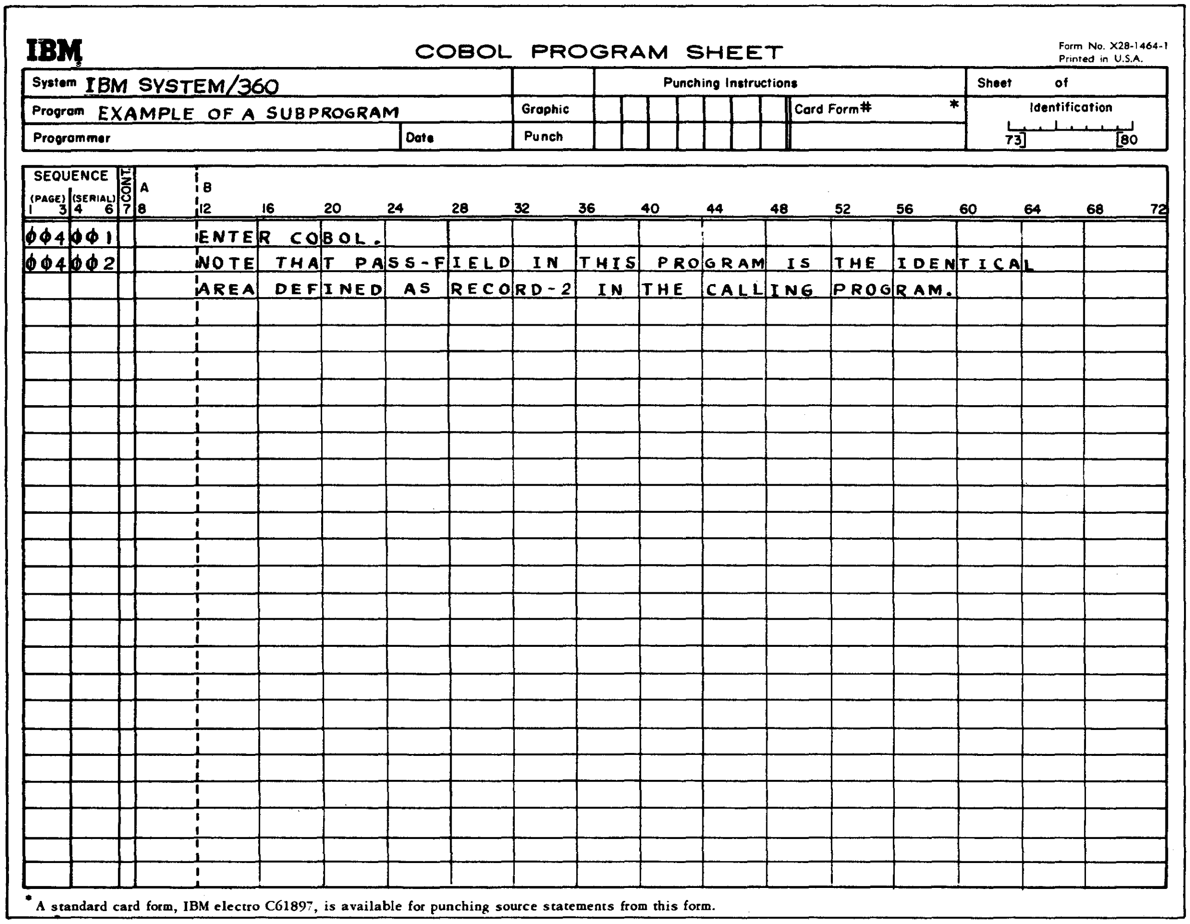
Form No. X28-1464
Printed in U.S.A.

System IBM SYSTEM/360		Punching Instructions	Sheet of
Program EXAMPLE OF A SUBPROGRAM	Graphic	Card Form# *	Identification
Programmer / Date	Punch		73 80

```
SEQUENCE  C
(PAGE)(SERIAL) O A   B
1   3 4   6   N
              T
              7 8   12  16  20  24  28  32  36  40  44  48  52  56  60  64  68  72
ØØ3ØØ1 IDENTIFICATION DIVISION.
   ØØ2 PROGRAM-ID. 'SUBPROG'.
   ØØ3 REMARKS. EXAMPLE OF A SUBPROGRAM.
   ØØ4 ENVIRONMENT DIVISION.
   ØØ5 CONFIGURATION SECTION.
   ØØ6 SOURCE-COMPUTER. IBM-36Ø D3Ø.
   ØØ7 OBJECT-COMPUTER. IBM-36Ø D3Ø.

   ØØ8 DATA DIVISION.
   ØØ9 WORKING-STORAGE SECTION.

   Ø1Ø 77  MODIFICATION PICTURE X(12),VALUE IS 'PUT ANY DATA'.
   Ø11 LINKAGE SECTION.
   Ø12 Ø1  PASS-FIELD.
   Ø13   Ø2 A PICTURE X(68).
   Ø14   Ø2 B PICTURE X(12).

   Ø15 PROCEDURE DIVISION.
   Ø16 START. ENTER LINKAGE.
   Ø17     ENTRY 'SUBPRGM' USING PASS-FIELD.
   Ø18     ENTER COBOL.
   Ø19 MODIFY. MOVE MODIFICATION TO B.
   Ø2Ø     ENTER LINKAGE.
ØØ3Ø21     RETURN.
```

* A standard card form, IBM electro C61897, is available for punching source statements from this form.

Figure 27. Example of a Subprogram (Part 1 of 2)



IBM COBOL PROGRAM SHEET

Form No. X28-1464-1
Printed in U.S.A.

System IBM SYSTEM/360

Punching Instructions

Sheet of

Program EXAMPLE OF A SUBPROGRAM

Graphic

Card Form# *

Identification 73 80

Programmer

Date

Punch

```
SEQUENCE     A   B
(PAGE)(SERIAL) CONT.
1  3 4  6 7 8   12  16  20  24  28  32  36  40  44  48  52  56  60  64  68  72
ØØ4ØØ1     ENTER COBOL.
ØØ4ØØ2     NOTE THAT PASS-FIELD IN THIS PROGRAM IS THE IDENTICAL
           AREA DEFINED AS RECORD-2 IN THE CALLING PROGRAM.
```

* A standard card form, IBM electro C61897, is available for punching source statements from this form.

Figure 27. Example of a Subprogram (Part 2 of 2)

A table of subroutines used by COBOL to accomplish the statements or actions specified follows. The 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.)

SUBROUTINE NAME	ACTION
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 (TOBIN), which converts the number from packed decimal to binary, and IHD01500 (BINFL), 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 (FRFLPT), which converts the number from floating point to binary, and IHD01800 (BINZN), 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 visa 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 then 9 decimal digits.

SUBROUTINE NAME	ACTION
	- 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, and 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 to a floating point number. Conversion is accomplished by calling the same subroutine used by FLPZND (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.

SUBROUTINE NAME	ACTION
IHD01300 Converts a floating point number to packed decimal format. Conversion is accomplished by calling IHD01100 (FRFLPT), which converts a floating-point number to binary, and IHD01000 (BINPK), 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 (FRFLPT), 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 for: 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.

SUBROUTINE NAME	ACTION
	- 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, zoned field is 9. 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 (FRFLPT) which converts internal floating point to binary, and IHD01500 (BINFL) which converts binary to external floating point.	MOVE - Required when receiving field is external floating point. MISCELLANY - Required if user displays interal floating point.
IHD02000	Used to move group items longer than 256 bytes.
IHD02100	Performs the class test on alphameric fields, as specified in the IBM publication, Specifications COBOL Language, listed on the cover of this manual.
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.

SUBROUTINE NAME	ACTION
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 when linkage editing a version I object deck with a version II system.
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 SYSPCH. Used when display on SYSPCH is specified.

SUBROUTINE NAME	ACTION
IHD02800	Writes out display data on SYSLST. Required when EXHIBIT, TRACE, or standard DISPLAY statements are used (i.e., not UPON CONSOLE or UPON SYSPCH).
IHD02900	Reads a record from SYSIPT and moves data to the field specified by data-name. Required when ACCEPT is specified (not ACCEPT FROM CONSOLE).
IHD03000	Used for display on console.
IHD03100	Used for execution of direct-access statements. Required when any direct-access statement is used.
IHD03200	If problem program has user labels, this subroutine is the linkage with the declaratives section.
IHD03300	If one field is divided by another and the divisor is zero, this subroutine links to the on size error routine.
IHD03400	Prints out object time diagnostics when errors are encountered in direct-access processing. Required when IHD03100 is used.
IHD03500	Produces object time diagnostics for indexed sequential organization of files. Required when indexed sequential data organization is indicated.
IHD03600	Required to write record number zero on all tracks for an output operation when using direct access method.
IHD03700	Used for initializing tape or disk when using read and write operations.

APPENDIX H. IBM SYSTEM/360 DISK AND TAPE OPERATING SYSTEMS COMPILER DIAGNOSTICS

This appendix contains a detailed description of diagnostics. They consist of:

- Compiler diagnostic messages
- Object time messages
- Debug packet error messages

These messages are produced during compilation.

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, System Control and System Service Programs listed on the cover of this manual.

Also included in this appendix is an illustration of object storage layout.

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.

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.

All messages preceded by a # are self-explanatory.

IJS001I C - LITERAL EXCEEDS 120 CHARACTERS.

The element count begins following the next quote on the line, if there is one, or following the element beginning after the 120th character.

IJS002I W - LITERAL CONTINUATION QUOTE INVALID IN MARGIN A.

Continuation is allowed.

IJS003I C - LITERAL IMPROPERLY CONTINUED OR CONTINUATION QUOTE IS MISSING.

The non-numeric literal is truncated at the end of the preceding line. The syntax scan resumes with the first element on the next line. This may be the result of a missing quote sign on the preceding line.

#IJS004I C - SYNTAX REQUIRES A BLANK AFTER A PERIOD OR PERIOD IS INVALID DECIMAL POINT.

IJS005I C - XXX EXCEEDS 30 CHARACTERS.

Any element that is not a non-numeric literal is truncated after 30 characters.

IJS006I C - QUALIFICATION - XXX REQUIRES QUALIFICATION.

This indicates that the name is defined in more than one location, and requires qualification in order to be unique. The first name defined is assumed.

IJS007I C - QUALIFICATION - XXX HAS UNDEFINED QUALIFICATION.

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. The first name defined is assumed.

IJS008I C - QUALIFICATION - XXX REQUIRES MORE QUALIFICATION.

The number of qualifiers or the names are not sufficient to make the subject name unique. Another name could have the same qualification. The first name defined is assumed.

#IJS009I E - SUBSCRIPTED 88 MUST HAVE A RIGHT PARENTHESIS. WILL BE TREATED AS A DATA NAME.

#IJS010I W - SYNTAX REQUIRES A BLANK AFTER A RIGHT PAREN, SEMICOLON AND OR COMMA.

#IJS011I C - XXX IS UNDEFINED.

IJS012I E - XXX HAS MORE SUBSCRIPTS THAN DECLARED IN THE DATA DIVISION.

The PROCEDURE DIVISION reference to the data-name has too many subscripts. The number of subscripts must match the number of OCCURS clauses in the definition hierarchy in the DATA DIVISION.

#IJS013I C - RECORD NAME IS ASSOCIATED WITH INVALID FD.

IJS023I C - COPY/INCLUDE - COPY AND INCLUDE MUST NOT BE USED WITHIN LIBRARY ENTRIES.

Words following the library name are diagnosed according to the clause being processed, up to the next required clause.

IJS024I C - COPY/INCLUDE - PERIOD MISSING FOLLOWING XXX. THE NEXT CARD MAY BE SKIPPED.

A period should be inserted following library book name. 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.

IJS025 C - COPY - XXX IS AN INVALID LIBRARY NAME OR NOT FOUND ON LIBRARY

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.

IJS026I C - VALUE - FLOATING-POINT NUMBER XXX IS BELOW OR ABOVE VALID RANGE.

The value of zero is assumed.

IJS027I W - VALUE - NUMBER OF DECIMALS IN LITERAL XXX AND DATA ENTRY DISAGREE.

Truncation or padding is performed according to the rules governing the MOVE verb.

IJS028I C - VALUE - LITERAL XXX IS INVALID AND IS DROPPED.

The value clause conflicts with the description of the entry and is dropped.

IJS029I W - VALUE - LITERAL XXX AND PICTURE SIZES DISAGREE.

This diagnostic points out a literal larger than its picture. 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.

IJS030I W - VALUE - LITERAL XXX WAS SIGNED, ENTRIES PICTURE WAS UNSIGNED.

The literal encountered in this entry contains a sign, it does not appear as part of the entry because the picture is unsigned.

IJS031I W - VALUE - NUMBER OF INTEGERS IN LITERAL XXX AND DATA ENTRY DISAGREE.

Same as message 27.

#IJS032I C - INCLUDE - LIBRARY NAME IS AN INVALID EXTERNAL NAME OR NOT IN THE LIBRARY.

#IJS041I C - OCCURS - THIS CLAUSE IGNORED AT THE 01 LEVEL IN XXX ENTRY.

#IJS042I C - OCCURS - THIS CLAUSE IGNORED IN XXX ENTRY AS IT PROVIDES MORE THAN 3 LEVLES OF SUBSCRIPTING.

IJS043I C - OCCURS - DEPENDING ON OPTION IN XXX ENTRY IS IGNORED DUE TO PRIOR USE.

The occurs depending-on option can appear only once in a given record and it must contain the last entry within that 01.

#IJS044I C - OCCURS - DEPENDING ON OPTION IN XXX ENTRY IS IGNORED BECAUSE IT IS SUBORDINATE TO A PREVIOUS OCCURS CLAUSE.

IJS045I C - OCCURS - THE LEVEL OF XXX ENTRY INVALIDATES THE DEPENDING OPTION AT THE PRECEDING XXX ENTRY. THE DEPENDING OPTION IS DROPPED.

The level number just encountered indicates that there was an occurs depending that did not include the last entry within the 01.

IJS046I C - XXX ENTRY CONTAINS AN ILLEGAL LEVEL NUMBER OR REDEFINES CLAUSE WHICH IS IGNORED.

A redefines clause must redefine an entry at the same level number.

IJS047I E - INTERNAL QUALIFIER TABLE OVERFLOWED WHEN HANDLING XXX. RESTARTED QUALIFIERS WITH XXX.

Qualification: 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.

#IJS048I W - REDEFINES - ENTRY PRECEDING XXX IS OF VARIABLE LENGTH.

IJS049I W - REDEFINES - XXX IS LARGER THAN ENTRY REDEFINED.

The current entry is larger than the area redefined. The area is assumed to be expanded.

IJS050I W - REDEFINES - XXX ENTRY PRECEDING XXX IS LARGER THAN ENTRY REDEFINED.

Same as for message 049, only for a group entry.

IJS051I C - REDEFINES - THIS CLAUSE INVALID IN XXX ENTRY AS REDEFINED AREA IS SUBSCRIPTED.

It is invalid to redefine an area containing an occurs clause. The redefinition clause is dropped.

IJS052I C - VALUE - THIS CLAUSE IGNORED IN XXX ENTRY DUE TO REDEFINES OR OCCURS CLAUSE IN PRECEDING XXX LEVEL.

A value clause cannot appear in an entry subordinate to a redefines clause. The value clause is dropped.

IJS053I W - ALIGNMENT - FOR PROPER ALIGNMENT, A XXX BYTE LONG FILLER ENTRY IS INSERTED PRECEDING XXX.

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. The number of slack bytes required can be reduced by the use of a different data format such as: packed, 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, COBOL Language Specifications listed on the cover of this manual.

IJS054I W - ALIGNMENT - FOR PROPER ALIGNMENT, A XXX BYTE LONG XXX FILLER ENTRY IS INSERTED PRECEDING XXX.

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. Refer to message 53.

IJS055I E - XXX ENTRY PRECEDING XXX EXCEEDS MAXIMUM SIZE OF 4092 BYTES.

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.

IJS056I W - XXX ENTRY PRECEDING XXX EXCEEDS MAXIMUM LENGTH OF 32,768 BYTES.

Same as 55 except for the working storage section.

IJS057I E - PROGRAM EXCEEDS 240 BASE LOCATORS MAXIMUM AT XXX.

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. The base locator counter wraps around and the results are unpredictable.

IJS058I E - ERRONEOUS OR MISSING DATA DIVISION.

No data division entries were present or all data division entries were dropped because of errors.

IJS060I W - REDEFINES - XXX LEVEL PRECEDING XXX IS OF VARIABLE LENGTH.

The entry, defined at the level indicated, that preceded this clause, contained an occurs depending clause. The redefined clause is dropped because it is illegal to redefine a variable-length entry.

IJS061I C - XXX ENTRY EXCEEDS MAXIMUM LENGTH FOR ITS DATA TYPE.

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. The maximum size is assumed.

IJS062I W - REDEFINES - XXX REQUIRED ALIGNMENT AND STARTS XXX BYTES PAST THE START OF THE ENTRY IT REDEFINED.

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.

IJS063I W - ALIGNMENT - TO ALIGN BLOCKED RECORDS ADD XXX BYTES TO THE 01 CONTAINING DATANAME XXX.

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.

IJS064I W - ALIGNMENT - IF THE PRECEDING RECORD IS BLOCKED, IT MAY BE ALIGNED BY MOVING TO AN 01 IN THE WORKING-STORAGE SECTION.

When records are variable and blocked, only the first record can be aligned.

IJS076I W - RERUN - INTEGER OPTION IS NOT PERMITTED.

This clause is dropped.

#IJS077I E - USER LABELS NOT SUPPORTED IN THIS VERSION.

IJS078I C - SELECT - INTERNAL FILE-NAME AND DESCRIPTION TABLE OVERFLOWED. XXX NOT PROCESSED.

There is a fixed number of files that can be handled by a given COBOL compilation. If additional files must be handled, they can be processed in a subprogram and accessed via the linkage facility. Any files that are encountered after the maximum permitted are dropped.

#IJS079I C - APPLY - RESTRICTED SEARCH INTEGER TOO LARGE ON XXX. CLAUSE DROPPED.

#IJS080I C - APPLY - MORE THAN THREE FORMS OVERFLOW CLAUSES. OVERFLOW-NAME XXX ENTRY IS DROPPED.

IJS081I W - SAME - XXX APPEARED PREVIOUSLY IN A 'SAME' CLAUSE. REMAINDER OF SAME CLAUSE DROPPED.

A given filename can appear in only one same-area clause. If a duplication is encountered, the entire same-area clause is dropped.

IJS082I W - SAME - INTERNAL 'SAME' TABLE OVERFLOW. ENTRIES AFTER XXX DROPPED.

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.

IJS083I W - RECORD - RECORD LENGTH SPECIFIED DISAGREES WITH CALCULATED MAX. RECORD LENGTH OF XXX ON XXX. CALCULATED RECORD LENGTH ASSUMED.

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.

IJS084I W - BLOCK - BLOCK SIZE FOR XXX TOO BIG. 32K ASSUMED.

The integer specifying block size for the referenced files is too large. The maximum size allowed is assumed.

IJS085I E - INDEXED - SYMBOLIC KEY MUST BE SPECIFIED FOR XXX IF INPUT.

This message is used for a direct access storage device only.

IJS086I E - RELATIVE - ACTUAL KEY MUST BE SPECIFIED FOR XXX.

This message is used for a direct access storage device only.

IJS087I C - THE XXX FILE MUST BE DESCRIBED IN A SELECT CLAUSE. CURRENT ENTRY IGNORED.

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.

IJS088I C - LABEL - LABEL·RECORD DATA-NAME MUST BE DEFINED IN LINKAGE SECTION.

Label records are assumed standard.

#IJS089I C - ASSIGN - UNIT IS MISSING FOR XXX FILE. 2400 IS ASSUMED.

IJS090I C - I/O FD - THE DESCRIPTION OF XXX FILE CONFLICTS ON THE FOLLOWING POINTS --- XXX.

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. The points in conflict are defined by the trailing clauses of the diagnostic.

IJS091I E - INDEXED - INDEXED ORGANIZATION ON XXX NOT VALID FOR THIS LEVEL COMPILER.

This message is used for a direct access storage device only.

IJS092I E - DIRECT - DIRECT ORGANIZATION ON XXX NOT VALID FOR THIS LEVEL COMPILER.

This message is used for a direct access storage device only.

#IJS093I E - XXX NOT HANDLED WITH PRESENT RELEASE.

IJS094I E - SELECT - XXX FILE WAS NOT DEFINED BY AN FD ENTRY.

No DTF is built for this file, therefore, it cannot be used.

#IJS095I C - IF - ARITHMETIC EXPRESSION CANNOT BE USED IN NON-NUMERIC COMPARISON. TEST IS DROPPED.

#IJS096I W - RERUN - ONLY ONE CHECKPOINT FILE MAY BE SPECIFIED.

#IJS097I E - DIRECT ACCESS - STANDARD LABELS ARE REQUIRED ON XXX FILE.

#IJS098I C - ASSIGN - XXX FILE ASSUMED ASSIGNED TO UTILITY.

#IJS099I C - ASSIGN - XXX FILE UNIT MISSING AND ASSUMED TO BE 1403 PRINTER.

#IJS100I E - ASSIGN - DIRECT-ACCESS ASSIGNED TO XXX NOT SUPPORTED IN THIS VERSION.

IJS101I C - RECORDING - XXX FILE IS ASSIGNED TO UNIT RECORD AND MUST BE RECORDING MODE IS F.

Unit record must be fixed length. Largest described length is assumed.

IJS102I C - RESERVE - A MAXIMUM OF 1 ALTERNATE AREA IS ALLOWED FOR XXX FILE.

One alternate area is reserved.

IJS103I E - ASSIGN - XXX IS NOT A VALID SYSTEM ASSIGNMENT.

Must be SYS001 - SYS244, SYS001 is assumed.

#IJS104I E - RECORD/BLOCK SIZE ON XXX IS GREATER THAN 3625.

#IJS105I E - INVALID DEVICE NUMBER SPECIFIED. DISK 2311 ASSUMED.

#IJS106I W - ONLY ONE AREA SUPPORTED FOR INDEXED OR DIRECT ORGANIZATION. ONE AREA ASSIGNED FOR XXX.

#IJS107I C - RECORD KEY REQUIRED FOR INDEXED ORGANIZATION FILE XXX.

#IJS108I E - LENGTH OF SYMBOLIC/RECORD KEY GREATER THAN 256.

#IJS109I C - LENGTH OF ACTUAL KEY IS GREATER/LESS THAN 8.

#IJS110I E - INCORRECT DATA ITEM TYPE SPECIFIED FOR KEY.

#IJS111I W - TRACK AREA CLAUSE NOT SUPPORTED IN DOS.

#IJS112I C - SYMBOLIC AND RECORD KEY LENGTH FOR XXX DISAGREE.

#IJS113I E - ORGANIZATION - RELATIVE ORGANIZATION ASSIGNED TO XXX NOT SUPPORTED IN THIS VERSION. COMPLETE SELECT STATEMENT DROPPED.

#IJS114I E - RECORD/BLOCK ON XXX IS GREATER THAN 2000.

IJS176I C - WORD RECORD OR RECORDS IS REQUIRED. FOUND 'XXX.'

Syntax skips until the next clause, level number, or period at the end of the file description is encountered.

#IJS177I W - FILE - PERIOD REQUIRED AFTER WORD 'SECTION'.

IJS178I C - SYNTAX REQUIRES 'XXX'. FOUND 'XXX'.

This clause is ignored.

IJS179I W - FD - 'XXX' IS AN INVALID FILE-NAME FORMAT.

A filename must follow the format rules for data-names. Invalid names are truncated to 30 characters and assumed to be valid.

IJS180I E - XXX EXCEEDS 30 CHARACTERS AND IS DROPPED.

The picture is too long, therefore, it is dropped.

IJS181I W - THE OPTION WORD IS MISSPELLED OR OMITTED. FOUND XXX.

Usage is assumed DISPLAY.

IJS183I C - 'XXX' IS AN INVALID OR EXCESSIVE INTEGER.

The integer indicated in this clause is determined to be invalid and, therefore, not used.

IJS184I W - XXX IS AN INVALID LEVEL NUMBER.

The level number found is changed to 01, syntax scanning proceeds.

#IJS185I W - LABEL RECORD IS OMITTED. LABELS ASSUMED STANDARD.

IJS186I W - SYNTAX REQUIRES DATA RECORD CLAUSE.

Syntax scanning proceeds.

IJS187I C - MODE MUST BE 'V', 'F', or 'U'. FOUND XXX.

If V, F, or U was specified, check the element number on this line for a misspelled optional word.

IJS190I W - LABEL - 'XXX' IS AN INVALID DATA-NAME FORMAT.

Invalid data-names are truncated to 30 characters and assumed valid.

IJS191I W - SD OR SA ENTRY REQUIRES F LEVEL COMPILER.

Syntax skips to next margin A entry.

#IJS192I W - 'XXX' IS AN INVALID RECORD NAME FORMAT.

IJS194I C - 'XXX' IS INVALID AT THIS POINT. CHECK FOR SYNTAX ERROR ON CURRENT/PREVIOUS STATEMENT.

While processing a given clause or sentence, an unexpected element was encountered. The clause may be valid but misplaced. Check for prior diagnostics, extra or missing period, invalid continuation of non-numeric literals, or a misspelled word. This diagnostic is also given for clauses that are not valid source input to this level compiler.

#IJS195I E - SYNTAX REQUIRES AN FD ENTRY. FOUND XXX.

#IJS196I W - SYNTAX REQUIRES AN 01 LEVEL ENTRY. FOUND XXX.

#IJS197I W - NOT VALID FOR THIS LEVEL COMPILER.

IJS201I C - XXX IS AN INVALID DATA-NAME FORMAT BUT ASSUMED VALID.

Invalid data-names are truncated to 30 characters and assumed valid.

#IJS202I C - 'XXX' IS INVALID AT THIS POINT. CHECK FOR SYNTAX ERROR ON CURRENT/PREVIOUS STATEMENT.

#IJS203I C - THIS USAGE XXX CONFLICTS WITH THE GROUP USAGE AND IS IGNORED.

IJS204I C - XXX IS AN INVALID OR EXCESSIVE INTEGER.

Invalid integer is dropped.

#IJS205I W - XXX IS AN INVALID DATA-NAME FORMAT, BUT ASSUMED VALID.

IJS206I W - WORD ZERQ REQUIRED. FOUND XXX.

The clause is ignored.

IJS207I W - WORD RIGHT IS REQUIRED. FOUND XXX.

The clause is ignored. Right justified.

IJS210I C - THIS ENTRY CONFLICTS WITH THE FOLLOWING DESCRIPTIONS--XXX.

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. This diagnostic could be produced when (for example) the picture clause for the second entry is encountered, and automatically conflicts with the picture clause for the previous entry.

#IJS211I C - XXX EXCEEDS 30 CHARACTERS AND IS TRUNCATED.

IJS212I C - ONLY LEVELS 77 OR 01 ARE PERMITTED AT THIS POINT. FOUND XXX.

Syntax skips until a section name or level number is found.

IJS213I W - THE FOLLOWING DESCRIPTIONS INVALID AT GROUP LEVEL---XXX.

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.

IJS214I C - ELEMENTARY - XXX DATA ENTRY REQUIRES A PICTURE, COMPUTATIONAL-1 OR COMPUTATIONAL-2.

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. Check for missing periods or other diagnostic messages. Any statement in the procedure division containing a reference to this entry is diagnosed and dropped.

IJS215I W - SYNTAX REQUIRES AN ENTRY IN MARGIN A. FOUND XXX IN MARGIN B.

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.

IJS216I W - SYNTAX REQUIRES AN ENTRY IN MARGIN B. FOUND XXX IN MARGIN A. CHECK FOR MISSING PERIODS.

All entries in margin A must be preceded by a period. The compiler was in the middle of processing a clause or sentence and encountered the indicated word in margin A. A diagnostic is given and the word is processed as if valid.

#IJS217I W - LEVEL 77 ENTRIES MUST PRECEDE OTHER LEVELS AND ARE ASSUMED TO BE 01 LEVEL.

#IJS218I W - SYNTAX PERMITS ONLY LEVELS 77, 88, OR 01 AFTER A 77 LEVEL. THE LEVEL WAS CHANGED XXX TO 01.

IJS221I C - SYNTAX FOR ALL REQUIRES XXX BE A SINGLE CHARACTER IN QUOTES.

The value clause is dropped.

IJS222I C - PICTURE XXX WAS FOUND INVALID WHILE PROCESSING XXX. THE PICTURE IS DROPPED.

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. The picture is dropped, and the entry identified as an error.

#IJS227I E - FILE SECTION OUT OF SEQUENCE.

IJS228I E - SYNTAX PERMITS ONLY ONE XXX IN SOURCE PROGRAM.

Syntax proceeds.

#IJS229I E - WORKING STORAGE SECTION OUT OF SEQUENCE.

#IJS231I E - ENVIRONMENT DIVISION MISSING.

#IJS233I C - REPORT SECTION REQUIRES F LEVEL COMPILER.

#IJS234I W - WORD 'SECTION' MISSING.

#IJS235I W - 'PERIOD' MUST FOLLOW WORD SECTION.

IJS238I W - 'XXX' IS AN INVALID SECTION NAME OR INVALID/MISPLACED LEVEL INDICATOR.

Syntax skips until a valid section-name or level number is found.

#IJS239I W - SYNTAX REQUIRES WORD 'DIVISION'.

IJS241I C - 88-LEVEL PRECEDING 88 MUST BE AN ELEMENTARY.

Any level number preceding an 88 must be an elementary. If it is not, it is assumed to be an elementary and is processed.

#IJS242I W - THE 88 ENTRY DOES NOT HAVE A VALUE, THEREFORE, IT IS DROPPED.

IJS301I W - SYNTAX REQUIRES 'XXX' IN MARGIN A. FOUND 'XXX'. RESTART WITH 'XXX'.

Syntax requires the specific entry indicated to be in margin A. If the entry is found in margin B, compilation resumes.

IJS302I C - SYNTAX REQUIRES 'XXX'. FOUND 'XXX'. RESTART WITH 'XXX'. IF WORDS REQUIRED AND FOUND ARE THE SAME, THE ENTRY IS IN THE WRONG MARGIN.

Syntax skips to the restart clause.

#IJS303I W - 'XXX' IS AN INVALID CONDITION-NAME FORMAT

IJS304I E - 'XXX' IS AN INVALID EXTERNAL-NAME FORMAT. RESTART WITH 'XXX'.

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.

IJS305I C - SYNTAX REQUIRES SAME, RERUN, APPLY, OR 'XXX' DIVISION. FOUND 'XXX'. RESTART WITH 'XXX'.

Check for invalid sequence of source program cards or extra periods.

#IJS306I W - SYNTAX REQUIRES 'ENVIRONMENT' OR 'XXX' DIVISION IN MARGIN A. FOUND 'XXX'. RESTART WITH 'XXX'.

IJS307I E - SYNTAX REQUIRES I-O-CONTROL, INPUT-OUTPUT, OR 'XXX' DIVISION IN MARGIN A. FOUND 'XXX'. RESTART WITH 'XXX'.

Same as 305.

IJS308I W - 'XXX' IS AN INVALID DATA-NAME FORMAT. RESTART WITH 'XXX'.

A data-name was expected at this point in the scan of the subject clause. Invalid format is truncated to 30 characters and processed as if valid.

IJS309I C - ENVIRONMENT PARAGRAPHS OUT OF ORDER.

Statements are handled anyway.

IJS310I W - 'XXX' IS AN INVALID 360 MODEL-NUMBER. RESTART WITH 'XXX'.

Syntax skips to the restart clause.

IJS311I E - SYNTAX REQUIRES 'FILE-CONTROL', 'XXX' OR 'DATA DIVISION' IN MARGIN A. FOUND 'XXX'. RESTART WITH 'XXX'.

Same as 305.

IJS312I C - 'XXX' IS AN INVALID OR EXCESSIVE INTEGER. RESTART WITH 'XXX'.

The syntax at this point of scan of the specified clause requires an integer. The element found was invalid and dropped.

IJS313I W - 'XXX' IS AN INVALID FILE-NAME FORMAT. RESTART WITH 'XXX'.

The syntax scan of the subject clause requires a filename at this point. The element found was invalid. It was truncated to 30 characters and used as if valid.

IJS314I E - 'XXX' IS AN INVALID LIBRARY-NAME FORMAT. RESTART WITH 'XXX'.

A library name is required at this point. It is an invalid format, and is dropped.

IJS315I W - APPLY - MORE THAN THREE OVERFLOW OPTION CLAUSES ARE USED.

An internal table permits a maximum of three form overflow names to be assigned in any compilation. Any more are dropped.

IJS316I C - SYNTAX REQUIRES 'INDEXED' OR 'XXX'. FOUND 'XXX'. RESTART WITH 'XXX'.

This message is used for a direct access storage device only.

IJS317I C - SYNTAX REQUIRES 'SEQUENTIAL' OR 'XXX'. FOUND 'XXX'. RESTART WITH 'XXX'.

This message is used for a direct access storage device only.

IJS318I E - SYNTAX REQUIRES 'XXX' OR DATA DIVISION IN MARGIN A. FOUND 'XXX'. RESTART WITH 'XXX'.

The syntax for the subject clause requires specific entries at this point. Check for misspelled words.

IJS319I C - SYNTAX REQUIRES 'UTILITY', 'DIRECT-ACCESS' OR 'XXX'. FOUND 'XXX'. RESTART WITH 'XXX'.

See message 318.

IJS320I W - 'XXX' IS AN INVALID I-O-DEVICE-NUMBER. RESTART WITH 'XXX'.

See message 318.

IJS321I E - NO PROCESSING OF THIS MULTIPLE SPECIFIED DIVISION OR SECTION. RESTART WITH 'XXX'.

A section or division was encountered more than once. It was dropped, rather than disturb the internal sequence of the compilation.

#IJS322I W - FILE-NAME OR DATA-NAME EXCEEDS 30 CHARACTERS. TREATED AS 30-CHARACTER NAME.

IJS323I W - SYNTAX REQUIRES 'XXX' OR CLAUSE-NAME. FOUND 'XXX'. RESTART WITH 'XXX'.

Syntax skips to the restart clause.

IJS324I E - SYNTAX REQUIRES 'REEL' OR 'XXX'. FOUND 'XXX'. RESTART WITH 'XXX'.

Syntax skips to the restart clause.

IJS401I C - SYNTAX REQUIRES A DATA-NAME. FOUND 'XXX'.

The syntax of the indicated clause requires 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. 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.

IJS402I C - SYNTAX REQUIRES NEXT ITEM BE 'XXX'.

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. The element found is displayed unless it was a name, in which case the word <u>invalid</u> name or <u>data</u> name is indicated. 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.

IJS403I C - SYNTAX REQUIRES A DATA-NAME OR NUMERIC-LITERAL. FOUND 'XXX'.

See message 402.

IJS404I C - SYNTAX REQUIRES EITHER WORD 'TO', OR 'GIVING'. FOUND 'XXX.

See message 402.

IJS405I C - SYNTAX REQUIRES A SINGLE CHARACTER IN QUOTES OR A FIGCON. FOUND 'XXX'.

See message 402.

IJS406I C - SYNTAX REQUIRES A FILE-NAME. FOUND 'XXX'.

See message 402.

IJS407I C - SYNTAX REQUIRES DATA-NAME OR INTEGER. FOUND 'XXX'.

See message 402.

IJS408I C - SYNTAX REQUIRES WORD 'INPUT', 'OUTPUT', OR 'I-O'. FOUND 'XXX'.

See message 402.

IJS409I C - SYNTAX REQUIRES A PROCEDURE-NAME. FOUND 'XXX'.

See message 402.

IJS410I C - SYNTAX REQUIRES A DATA-NAME OR LITERAL. FOUND 'XXX'.

See message 402.

IJS411I C - SYNTAX REQUIRES WORD 'CALL', 'ENTRY', OR 'RETURN'. FOUND 'XXX'.

See message 402.

IJS412I E - SYNTAX REQUIRES AN EXTERNAL-NAME. FOUND 'XXX'.

See message 402.

IJS413I C - SYNTAX REQUIRES '='. FOUND 'XXX'.

See message 402.

IJS414I 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.

See message 402.

IJS415I C - SYNTAX REQUIRES CALL PARAMETERS TO BE EITHER DATA-NAME, PROCEDURE-NAME OR FILE-NAME. FOUND 'XXX'.

See message 402.

IJS416I C - SYNTAX REQUIRES DATA-NAME, LITERAL FIGCON, '+', '-', '(' OR 'NOT'. FOUND 'XXX'.

See message 402.

IJS417I C - SYNTAX REQUIRES AN ARITHMETIC OPERATOR OR RELATIONAL. FOUND 'XXX'.

See message 402.

IJS418I C - SYNTAX REQUIRES A DATA-NAME, NUMERIC-LITERAL, OR '(' AFTER AN OPERATOR. FOUND 'XXX'.

See message 402.

IJS419I C - SYNTAX REQUIRES A DATA-NAME, LITERAL, FIGCON, '(', '+' OR '-' AFTER A RELATIONAL. FOUND 'XXX'.

See message 402.

IJS420I C - SYNTAX REQUIRES A VERB, PERIOD, ELSE OR OTHERWISE. FOUND XXX.

The end of a valid clause was encountered. The element that followed the valid termination of this clause is not valid. 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.

IJS421I C - ENTRY PARAMETER MUST BE A DATA-NAME. FOUND XXX.

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.

IJS422I C - SYNTAX REQUIRES A RELATIONAL. FOUND XXX.

Syntax requires that the next element be a relational. Check for invalid punching or a preceding error.

#IJS423I C - SYNTAX REQUIRES WORD 'INPUT' OR 'OUTPUT'. FOUND XXX.

IJS424I C - SYNTAX REQUIRES WORDS 'TO PROCEED TO'. FOUND XXX.

See message 402.

IJS425I C - SYNTAX REQUIRES WORD 'CONSOLE' OR 'SYSPCH'. FOUND XXX.

See message 402.

#IJS426I E - SYNTAX REQUIRES 'AT END' OR 'INVALID KEY'. FOUND XXX.

IJS427I C - SYNTAX REQUIRES A DATA-NAME, FIGCON OR NON-NUMERIC LITERAL. FOUND XXX.

See message 402.

IJS428I C - SYNTAX REQUIRES A PROCEDURE-NAME AFTER 'GO TO' NOT PRECEDED BY A PARAGRAPH-NAME. FOUND XXX.

See message 402.

IJS429I C - SYNTAX REQUIRES 'ALL', 'LEADING', 'UNTIL', OR 'FIRST'. FOUND XXX.

See message 402.

IJS430I C - SYNTAX REQUIRES WORD 'TALLYING' OR 'REPLACING'. FOUND XXX.

See message 402.

IJS431I C - SYNTAX REQUIRES WORD 'DEPENDING ON'. FOUND XXX.

See message 402.

IJS432I C - DATA TYPE MUST BE ED, ID, OR BI.

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. The statement is dropped.

*IJS433I C - SYNTAX REQUIRES WORD TRACE. FOUND XXX.

See message 402.

*IJS434I C - SYNTAX REQUIRES THAT A PERIOD OR 'SECTION' FOLLOWS PARAGRAPH-NAME.

See message 402.

*IJS435I E - DATA NAME AND ANY QUALIFIER MUST APPEAR WITHIN THE FIRST SEVEN OPERANDS OF STATEMENT FOR CHANGED OPTION.

See message 402.

*IJS436I C - SYNTAX REQUIRES A DATA-NAME, FIGCON OR LITERAL. FOUND XXX.

See message 402.

*IJS437I C - SYNTAX REQUIRES A FIGCON. FOUND XXX.

See message 402.

*IJS438I C - SYNTAX REQUIRES DATA-ITEM TO BE NO LONGER THAN FOUR.

See message 402.

IJS439I C - WRONG SUBSCRIPT SPECIFICATION.

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. There cannot be 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.

IJS440I C - INCORRECT SPECIFICATION IN DECLARATIVE-SECTION. FOUND XXX.

See message 402.

IJS441I C - SYNTAX REQUIRES AN INTEGER NOT LONGER THAN 5. FOUND XXX.

The integer exceeds the size permitted by language specifications. The statement is dropped.

IJS442I C - THE DECLARATION OF THIS DATA-NAME CAUSES IT TO BE FLAGGED AS AN ERROR.

The data-name encountered was flagged by the data division as containing an error in its declaration. Correct the declaration as indicated by the data division diagnostics and recompile. The statement in the procedure division is dropped.

IJS443I E - SYNTAX REQUIRES A VERB. FOUND XXX.

A point was reached where a verb was required, and was missing. For example 'IF A = B.' requires a verb between B and the period. The statement is skipped from the point of the error.

* The entire statement from the point of error is dropped, and is not compiled. This applies to messages IJS433I, IJS434I, IJS436I - IJS438I.

IJS444I E - SYNTAX REQUIRES A RECORD NAME. FOUND XXX.

See message 402.

IJS500I W - DISPLAY - AN OPERAND'S LENGTH EXCEEDS AND IS TRUNCATED TO 256 BYTES.

A maximum of 256 bytes can be displayed.

IJS501I W - DISPLAY - IF THIS VARIABLE-LENGTH ENTRY EXCEEDS 256, RESULTS WILL BE UNPREDICTABLE.

A maximum of 256 bytes can be displayed.

IJS502I W - STOP - LITERAL EXCEEDS AND IS TRUNCATED TO 72 BYTES.

In a stop-literal statement only the first 72 bytes of a longer field are typed on the console.

IJS503I W - ACCEPT - DATA EXCEEDS AND IS TRUNCATED TO 72 BYTES.

A maximum of one line (72 bytes) can be retrieved using the ACCEPT FROM CONSOLE statement.

IJS504I W - ACCEPT - DATA EXCEEDS AND IS TRUNCATED TO 256 BYTES.

A maximum of 256 bytes can be accepted from SYSIPT.

IJS505I C - RELATIONAL - FILENAMES OR STERLING-DATATYPE NOT ALLOWED IN COMPARE.

See message 506.

IJS506I C - RELATIONAL - USAGE OF DATA-TYPES CONFLICT. THE TEST DROPPED.

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 alphameric.

#IJS507I W - EXIT MUST BE ONLY STATEMENT IN PARAGRAPH.

#IJS508I E - THE STATEMENT CONTAINS AN UNDEFINED DATANAME.

See message 402.

#IJS509I C - AN ALPHABETIC DATANAME CAN BE TESTED ONLY FOR ALPHABETIC OR NOT ALPHABETIC, AND NUMERIC DATANAME ONLY FOR NUMERIC OR NOT NUMERIC.

IJS510I C - COMPARISON OF TWO LITERALS OR FIGCONS IS INVALID.

See message 506.

IJS511I C - DATA-TYPE IN ARITHMETIC STATEMENT IS NOT NUMERIC OR RECEIVING FIELD IS NOT NUMERIC OR REPORT.

See message 506.

IJS512I C - DATA-NAME IN CLASS-TEST MUST BE AN, ED, OR ID.

See message 506.

IJS513I C - DATA-NAME IN SIGN-TEST MUST BE NUMERIC.

See message 506.

IJS514I W - DISPLAY - DATA EXCEEDS AND IS TRUNCATED TO 72 BYTES.

If the data is longer than 72 bytes, only the first 72 bytes are printed for a DISPLAY ON CONSOLE statement.

IJS515I W - DISPLAY - DATA EXCEEDS AND IS TRUNCATED TO 120 BYTES.

If the data is longer than 120 bytes, only the first 120 bytes are printed for a DISPLAY statement.

#IJS516I C - OPEN 'NO REWIND' OR 'REVERSED' CANNOT BE SPECIFIED FOR A UNIT-RECORD OR DISK UTILITY FILE.

#IJS517I C - CLOSE - 'NO REWIND' OR 'LOCK' CANNOT BE SPECIFIED FOR A UNIT RECORD OR DISK UTILITY FILE.

#IJS518I E - MORE THAN 40 PARAMETERS ARE NOT ALLOWED WITH THE STATEMENT.

IJS519I C - SYNTAX ALLOWS ZERO AS ONLY VALID FIGCON IN A COMPARISON WITH BI, ID, EF, AND IF.

See message 506.

IJS520I C - SYNTAX ALLOWS SPACE OR ALL AS ONLY VALID FIGCONS IN COMPARISON WITH AN ALPHABETIC FIELD.

See message 506.

IJS521I C - DATATYPE MUST BE ED, EF, AL, AN OR GF. FOUND XXX.

The data types indicated are the only valid ones that can be used in the clause indicated.

IJS522I C - SYNTAX REQUIRES WORD RUN OR LITERAL. FOUND XXX.

Syntax skips the rest of the statement.

IJS523I C - RECEIVING FIELD IN PRECEDING STATEMENT IS A LITERAL.

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.

IJS524I C - SYNTAX REQUIRES AT LEAST TWO OPERANDS BEFORE GIVING OPTION.

For example, ADD A GIVING B. The statement is skipped.

IJS525I C - THE EXPRESSION HAS MORE RIGHT PARENS THAN LEFT PARENS TO THIS POINT. FOUND XXX.

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. 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.

IJS526I C - THE EXPRESSION HAS UNEQUAL NUMBER OF RIGHT AND LEFT PARENS.

See message 525.

IJS527I C - DATA-TYPE MUST BE ED, ID, OR BI. FOUND XXX.

The statement is skipped from the point of error.

IJS528I C - VARYING OPTION EXCEEDS THREE LEVELS.

With the varying option of the PERFORM verb, a maximum of three levels is permitted. The statement is dropped.

IJS529I C - DATA-TYPE MUST BE ED, ID, BI, EF, OR IF.

The data types shown are the only valid ones. The data-name found is not one of these types.

IJS530I C - NUMBER OF ELSES EXCEEDS NUMBER OF IFS.

Nested ifs must balance out with the appropriate number of else or otherwise. Recount and make corrections.

#IJS531I E - OCCURS TABLE - INTERNAL OCCURS - DEPENDING - ON TABLE OVERFLOWED AVAILABLE CORE.

IJS532I E - STATEMENT HAS TOO MANY OPERANDS.

The statement referenced is too large or complex for the internal tables needed for compilation. The statement should be divided into more than one statement.

IJS533I E - PARENTHESIZING REQUIRES SAVING TOO MANY OPERANDS.

See message 532.

IJS534I E - PARENTHESIZING REQUIRES SAVING TOO MANY INTERNALLY GENERATED LABELS.

See message 532.

IJS535I E - PARENTHESIZING REQUIRES SAVING TOO MUCH OF STATEMENT.

See message 532.

IJS536I E - ARITHMETIC EXPRESSION REQUIRES MORE THAN 9 INTERMEDIATE RESULT FIELDS.

See message 532.

#IJS537I C - USE - NOT HANDLED IN THIS VERSION.

IJS549I E - WORD XXX WAS EITHER INVALID OR SKIPPED DUE TO ANOTHER DIAGNOSTIC.

The majority of these messages will probably be caused by words skipped because of another diagnostic that occurred earlier in the statement. Correct the previous error. This diagnostic will also occur because of misspelled words. Correct the misspellings.

IJS550I C - A FIGURATIVE CONSTANT IS NOT ALLOWED AS A CALL OR ENTRY PARAMETER.

The statement is skipped from the point of the error.

IJS551I C - SYNTAX REQUIRES WORD TO. FOUND XXX.

Syntax skips the rest of the statement.

IJS552I C - RECEIVING FIELD MUST BE A DATA-NAME. FOUND XXX.

The statement is skipped from the point of the error.

#IJS553I E - A FIGURATIVE CONSTANT IS NOT ALLOWED AS A RECEIVING FIELD.

IJS554I C - THE XXX DATA-TYPE IS NOT A LEGAL RECEIVING FIELD.

The statement is skipped from the point of the error.

IJS555I C - OVERFLOW NAME IS NOT A VALID SENDING FIELD.

The statement is skipped.

#IJS556I E - END DECLARATIVES IS MISSING IN PROGRAM.

IJS557I W - FLOATING-POINT CONVERSION MAY RESULT IN TRUNCATION.

Conversion of floating-point numbers can result in truncation of low-order digits.

#IJS558I E - I-O OPTION FOR FILE CONFLICTS WITH NO REWIND.

IJS559I E - OUTPUT OPTION FOR FILE CONFLICTS WITH 'REVERSED'.

The output option conflicts with an opening of a file, reversed.

IJS560I C - SYNTAX REQUIRES WORD 'NAMED', 'CHANGED', OR 'CHANGED NAMED'. FOUND XXX.

The statement is skipped from the point of error.

IJS561I C - DATA TYPE MUST BE ED, ID, BI, EF, IF, RP, AL, AN, OR GF. FOUND XXX.

A filename, condition name, figcon, or variable-length group is not valid at this point.

IJS562I C - DATA ENTRY MUST NOT EXCEED 120 CHARACTERS.

The data entry specified exceeds the maximum permitted for this type of output.

#IJS563I C - DATA ENTRY MUST BE DISPLAY.

#IJS564I C - SYNTAX REQUIRES ONE OF THE ALLOWABLE CHARACTERS. FOUND XXX.

IJS565I C - AN IF STATEMENT MUST BE TERMINATED BY A PERIOD.

This diagnostic is obtained when the IF statement is the last statement of a paragraph and a label is detected. instead of a period.

#IJS566I C - DATA TYPE MUST BE AL, AN, RP, OR GROUP.

#IJS567I C - DATA TYPE MUST BE AL, AN, FIGCON OR FIXED-LENGTH GROUP.

#IJS568I C - DATA ITEM MUST NOT EXCEED 256 CHARACTERS.

#IJS569I C - DATA ENTRIES MUST BE OF EQUAL LENGTH.

#IJS570I C - THE LENGTH OF THE SECOND OPERAND MUST BE EQUAL TO THE FIRST OR A SINGLE CHARACTER.

#IJS571I E - A RECORD NAME MUST BE ASSOCIATED WITH THIS FILE. FOUND XXX.

#IJS572I C - ONLY ONE DATA-NAME MAY BE ASSOCIATED WITH THE CHANGED OPTION.

IJS573I C - DATA TYPE MUST BE ED, ID, BI, EFP, IFP, SNR, SR, RI, AL, AN, FIGCON, OR GROUP. FOUND XXX.

The statement is skipped from the point of error.

IJS601I W - NO SIGNIFICANT POSITION MATCHES BETWEEN SENDING AND RECEIVING FIELDS IN MOVE. RECEIVING FIELD IS SET TO ZERO.

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. Receiving field is set to zeros.

IJS602I W - DESTINATION FIELD DOES NOT ACCEPT THE WHOLE SENDING FIELD IN MOVE.

The sending field is larger than the receiving field in either its integer or decimal positions or both. Truncation of the sending field results.

#IJS603I C - AFTER ADVANCING OPTION NOT ALLOWED WITH REWRITE.

IJS604I E - SOURCE PROGRAM EXCEEDS LIMITS.

The program is too large. The user should do one of the following and then try again.

- Divide the program into two or more parts
- Simplify compound conditional statements.

IJS605I E - PROCEDURE NAME XXX MULTIPLY DEFINED.

Procedure name indicated was multiply defined and was not qualified properly by the appropriate section name when used.

IJS606I E - PROCEDURE-NAME XXX NOT DEFINED.

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 at which they are defined.

IJS607I E - LITERAL - INVALID, LITERAL XXX.

Check for multiple decimal points, non-numeric characters not enclosed in quotes.

IJS608I E - IT IS NOT ALLOWED TO HANDLE MORE THAN 25 FILES IN ONE STATEMENT.

The rest of the statement is skipped. Only 25 files are handled.

#IJS609I E - PROCEDURE-NAME XXX HAS ILLEGAL CONTENT AND IS DROPPED.

#IJS610I E - XXX WAS EITHER NOT ALLOWED IN THIS STATEMENT OR SKIPPED DUE TO ANOTHER DIAGNOSTIC.

#IJS611I E - TOO MANY PARAGRAPH NAMES HAVE BEEN USED IN CALL STATEMENTS.

#IJS612I W - OPEN STATEMENT CONTAINS MORE THAN 9 FILENAMES. OPEN WILL SPLIT.

#IJS614I E - THIS CONDITIONAL HAS A MISSING CONDITIONAL OPERATOR.

#IJS700I E - SOURCE PROGRAM NOT FOUND. COMPILATION CANCELED.

#IJS701I E - DATA DIVISION NOT FOUND. COMPILATION CANCELED.

#IJS702I E - PROCEDURE DIVISION NOT FOUND. COMPILATION CANCELED.

IJS703I E - SOURCE PROGRAM EXCEEDS INTERNAL LIMITS. COMPILATION CANCELED.

The above message is printed on SYSLST when the size of the assembler phase tables exceeds the core storage available for these tables. When this message appears, the programmer may be able to modify the source program to allow compilation on the source computer. There are essentially three variables that can be modified.

- The length and number of source labels could be reduced as the table for source labels must reserve (3 + L) bytes per source label.
- The number of literals could be reduced as 3 bytes are reserved for each literal.
- The size of the buffer can be reduced in machines above 16K storage size.

#IJS704I E - DATA-NAME TABLE OVERFLOW. COMPILATION CANCELED.

#IJS705I NO DIAGNOSTICS IN THIS COMPILATION.

#IJS706I EXECUTION CANCELED DUE TO E LEVEL DIAGNOSTIC.

(D) IJS707I CONFLICTING I/O ASSIGNMENTS.

SYS001, SYS002 and SYS003 are not assigned to the same type of device. Compilation is canceled. This message applies to Disk Operating System only.

(T) #IJS708I STORAGE ALLOCATED TO THE COMPILER IS LESS THAN 10K. COMPILATION CANCELED.

(D) #IJS708I STORAGE ALLOCATED TO THE COMPILER IS LESS THAN 14K. COMPILATION CANCELED.

#IJS709I W - INCORRECT COBOL OPTION 'XXX'.

#IJS710I W - BUFFSIZ CANNOT BE LESS THAN 170. ASSUMED 170.

(T) #IJS711I W - BUFFSIZ CANNOT BE GREATER THAN 32000. ASSUMED 32000.

(D) #IJS712I W - BUFFSIZ CANNOT BE GREATER THAN 3625 FOR WORK FILES ON DISK. ASSUMED 3625.

#IJS713I W - BUFFSIZ IS TOO LARGE FOR SIZE OF STORAGE ALLOCATED TO THE COMPILER. ASSUMED XXX.

A (T) preceding a message indicates that the message is applicable to Tape Operating System only.

A (D) preceding a message indicates that the message is applicable to Disk Operating System only.

OBJECT STORAGE LAYOUT

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 follows.

COBOL SUBROUTINES
WORKING STORAGE AND DATA LITERALS
EDIT MASKS
DTF TABLES
BUFFERS
PROCEDURE LITERALS
WORK AREA & GLOBAL TABLE
INSTRUCTIONS
SUBROUTINES (I/O)
SUBPROGRAMS

OBJECT TIME MESSAGES

A list of object time messages follows. Most of them are self-explanatory. Where deemed necessary, examples are included to explain the message. When COBOL is operating as a foreground program, these messages are output on SYS000.

#IHD901I*	AN UNCORRECTABLE DASD ERROR HAS OCCURRED.
#IHD902I*	WRONG LENGTH RECORD.
#IHD903I*	NO RECORD FOUND.
#IHD904I*	ILLEGAL ID SPECIFIED.
#IHD905I*	DUPLICATE RECORD.
#IHD906I*	CYLINDER OVERFLOW AREA FULL.
#IHD907I*	PRIME DATA AREA FULL.
#IHD908I*	CYLINDER INDEX AREA FULL.
#IHD909I*	MASTER INDEX AREA FULL.
#IHD910I*	RECORD OUT OF SEQUENCE.
#IHD911I	WRONG LENGTH RECORD.
#IHD912I	NO MORE ROOM FOUND ON TRACK.
#IHD913I	DATA CHECK IN COUNT AREA.
#IHD914I	DATA CHECK WHEN READING KEY OR DATA.
#IHD915I	NO RECORD FOUND.
#IHD993I	ZERO BASE-MINUS EXPONENT-PACKED RESULT MADE ALL NINES.
#IHD996I	RESULT TOO BIG-FLOATING POINT RESULT MADE MAX FP NUMBER.
#IHD997I	ZERO BASE-MINUS EXPONENT-FLOATING POINT RESULT IS MAX FP NUMBER.
#IHD998I	ZERO BASE-PLUS EXPONENT-FLOATING POINT RESULT MADE ZERO.
#IHD999I	MINUS BASE MADE PLUS AND FLOATING POINT EXPONENTIATION CONTINUED.
-------	AWAITING REPLY.

*These messages pertain to indexed sequential data organization only.

DEBUG PACKET ERROR MESSAGES

The following is a complete list of precompile error messages. They apply to errors in the debugging packet only.

IJS850I TABLE OF DEBUG REQUESTS OVERFLOWED. RUN TERMINATED.

IJS851I THE FOLLOWING CARD DUPLICATES A PREVIOUS *DEBUG CARD. THIS PACKET WILL BE IGNORED.

IJS852I THE FOLLOWING PROCEDURE DIVISION NAMES WERE NOT FOUND. INCOMPLETE DEBUG EDIT IS NOT TERMINATED.

IJS853I THE FOLLOWING *DEBUG CARD DOES NOT CONTAIN A VALID LOCATION FIELD. THIS PACKET WILL BE IGNORED.

IJS854I IDENTIFICATION DIVISION NOT FOUND. RUN TERMINATED.

IJS855I DEBUG EDIT RUN COMPLETE. INPUT FOR COBOL COMPILATION ON SYS004.

INDEX

READER'S COMMENT FORM

IBM System/360
Disk and Tape Operating Systems
COBOL Programmer's Guide

C24-5025-3

- 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

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

Cut Along Line

IBM S/360 Printed in U. S. A. C24-5025-3

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®

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]