DOS
TOS

IBM

Systems Reference Library

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

Program Number 360N-CB-452
360M-CB-402

This publication describes how to compile, linkage
edit, and execute a Disk and Tape COBOL program. The
text also describes the output from each of these
steps. In addition, it explains options of the compil-
er and many available features of the operating system.

The purpose of this publication is to enable programmers to compile, linkage edit, and execute COBOL programs under control of IBM System/360 Disk and Tape Operating Systems. The Disk and Tape COBOL language is described in the publication IBM System/360 Disk and Tape Operating Systems: COBOL Language Specifications, Form C24-3433, which is a corequisite to this publication.

Programmers unfamiliar with the Disk and Tape Operating Systems should read the Introduction and Sections I, II, and III for detailed information about preparing COBOL programs and deck structures for processing by the system.

Programmers who are familiar with the Disk and Tape Operating Systems and wish to know how to run COBOL programs should read Section I. This section contains the control card parameters and specific options needed to prepare deck structures for processing.

Sections IV, V, and VI contain information that is intended to aid programmers in writing efficient programs and in debugging programs that do not execute properly. The remaining sections discuss optional features of the Disk and Tape Operating Systems that are available to COBOL programmers.

This revision provides a new section about using the direct-access methods of the Disk Operating System. It also provides information about the new control cards and a discussion of variable length records.

Wider and more detailed discussions of the Disk and Tape Operating System are given in the following publications:

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

Publications closely related 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 C24-3429.

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 A24-6821.

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

FIGURES

TABLES

A Disk and Tape Operating Systems COBOL program is processed by the IBM
System/360 Disk and Tape Operating Systems. The operating system con-
sists of a number of processing programs and a control program. The
processing programs include the COBOL compiler, service programs, and
user-written programs. The control program supervises the execution of
the processing programs; controls the location, storage, and retrieval
of data; and schedules jobs for continuous processing.

A request to the operating system for facilities and scheduling of
program execution is called a job. For example, a job could request
execution of the COBOL compiler to compile a program. A job consists of
one or more job steps, each of which specifies execution of a program.
A programmer makes these requests to the operating system by use of job
control statements that may be punched into cards.

Each job is preceded by a JOB statement that identifies the job.
Each job step is preceded by an EXEC statement that names the program to
be executed and calls for execution. Included in each job step and pre-
ceding the EXEC statement are other job control statements (such as
ASSGN and XTENT) that describe data or request allocation of input/
output devices.

The data processed by execution of any processing program must be in
the form of a data file. A data file is a named, organized collection
of one or more records that are logically related. Information in a
data file may or may not be restricted to a specific type, purpose, or
storage medium. A data file may be a source program, a library of sub-
routines, or a file of data records that is to be processed by a COBOL
program.

A data file resides in one or more volumes. A volume is a unit of
external storage that is accessible to an input/output device. For
example, a volume may be a reel of tape or a disk pack.

In the Disk and Tape Operating Systems, input/output devices are
given standard symbolic names. A programmer can refer to an input/
output device in his program by using the appropriate symbolic name, and
the program is not dependent on an actual device address. The actual
device address is supplied by a job control statement when the program
is executed or at system generation time.

DATA ORGANIZATION

A data file used by a COBOL program can have one of three types of
organization: sequential, indexed sequential, or direct. The first
type (sequential) may be on any input/output device. All other types
must be on direct-access devices.

1. A sequential data file is one in which records are organized solely
    on the basis of their successive physical positions, as on tape.

2. An indexed sequential data file is one in which records are
    arranged in logical sequence (according to a key that is part of
    every record) on the tracks of a direct-access device. A separate
    index or set of indexes maintained by the system indicates the
    location of each record. This permits random, as well as sequen-
    tial, access to any record.

3. A _direct_ data file in COBOL is one in which records are referred to
    by use of keys. An actual key specifies the actual track address.
    A symbolic key identifies the record on the track.

EXECUTING A COBOL PROGRAM

Three basic operations are performed to execute a COBOL program: com-
pilation, linkage editing, and actual execution.

COMPILATION

Compilation is the process of translating a COBOL source program into a
series of instructions comprehensible to the computer, i.e., machine
language. In operating system terminology, the input to the compiler,
the source program, is called the _source module_. The output from the
compiler, the compiled source program, is called an _object module_.

LINKAGE EDITING

The linkage editor is a service program that prepares object modules for
execution. It can also be used to combine two or more separately com-
piled object modules into a format suitable for execution as a single
program. During the process of linkage editing, external references
between different modules are resolved. The executable output of the
linkage editor is called a _program phase_. The output may consist of one
or more program phases.

EXECUTION

Actual execution is under supervision of the control program, which
obtains a program phase from the core image library, loads it into main
storage, and initiates execution of the machine language instructions
contained in the program phase.

LIBRARIES

Another service program in the Disk and Tape Operating Systems is called
the _librarian_. The librarian consists of a group of maintenance rou-
tines that service the three system libraries. The maintenance routines
provide such operations as adding, deleting, or copying portions of a
library.

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

CORE IMAGE LIBRARY

All permanent programs in the Disk and Tape Operating System must be added to the core image library. The core image library is required, and the programs are stored in the library in the form of program phases. Unless the program phase has been linkage edited in the previous job step, the core image library is searched at execution time to obtain the program phase named in the EXEC statement.

SOURCE STATEMENT LIBRARY

The source statement library is used to store portions of COBOL source programs that are to be copied into a source program when the COPY or INCLUDE clauses of the COBOL language are used. The source statement library is not required. However, use of the source statement library can reduce the amount of coding needed for each individual program, that is, if standard file descriptions are added to the source statement library, they need not be coded again.

RELOCATABLE LIBRARY

The relocatable library is used to store object modules that can be subsequently linkage edited with other object modules. Each object module may also be a complete program that can be linkage edited and then executed. The relocatable library contains the COBOL library subroutines and the input/output modules used by the COBOL compiler.

MULTIPROGRAMMING

The DOS provides the capability of simultaneously processing two or three batched job streams for systems with at least 32K of main storage. This support is referred to as fixed partitioned multiprogramming because each job stream is assigned a different area or partition of main storage. The number and size of the partitions are allocated during system generation and may be altered by the operator.

There are two types of problem programs that can be run in a multiprogramming system: batch job processing and SPI (Single Program Initiator). Batched job processing is initiated by job control from the batched-job input stream. This capability is extended to all three programming partitions (BG, F1, and F2) if sufficient storage and separate input/output devices are available. The batched-job foreground option is selected when the system is generated by specifying MPS=BJF in the supervisor macro instructions. Programs run under SPI do not execute from a stack, but are initiated by the operator from the printer-keyboard. When an SPI program completes processing, the operator must explicitly initiate the next program.

The linkage editor determines whether the program to be executed is either a background or a foreground program. Both types of programs are initiated and terminated asynchronously of the other. Neither is aware of the other's existence.

COBOL source modules must be compiled as background programs. COBOL program phases can be executed as either background or foreground programs.

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

2. Operator communication routine

3. Foreground-one program

4. Foreground-two program

5. Background program

The background program must be a minimum of 10K. The foreground program areas must be in increments of 2K. The maximum size of a foreground program area is 510K.

This section provides information about preparing COBOL source programs for compilation, linkage editing, and execution.   Included are discussions of frequently used job control statements, linkage editor control statements, and librarian control statements.   Some individual examples are given, but Sections II and III should be scrutinized for complete examples of deck setups.

A complete list of job control, linkage editor, and librarian control statements and other options can be found in the publications IBM System/360 Disk Operating System: System Control and System Service Programs and IBM System/360 Tape Operating System: System Control and System Service Programs.

INPUT/OUTPUT DEVICE ASSIGNMENT

The input/output devices used for compilation, linkage editing, and execution are referred to by a standard set of symbolic names.   These symbolic names are used in COBOL programs and in job control statements instead of actual physical device addresses.   This provides several advantages for the programmer.   For example, a programmer uses the ASSIGN TO clause to assign a file to the appropriate symbolic name. Such a program is not dependent on the physical device address and, as such, does not have to be recompiled unless the device type changes. The symbolic names and their usage are shown in Figure 1.

The symbolic names are assigned to physical devices at system generation time, by the operator, or by means of the job control ASSGN statement.

If a programmer wishes to use the assignments made at system generation time, he need not include any ASSGN statements in his job unless he is using his own data files in the COBOL program.

SYMBOLIC NAME	FUNCTION	PERMISSABLE DEVICE TYPES
SYSRDR	Input unit for control statements	Card reader Magnetic tape unit Disk drive
SYSIPT	Input unit for programs	Card reader Magnetic tape unit Disk drive
SYSPCH	Main unit for punched output	Card punch Magnetic tape Disk drive
SYSLST	Main unit for printed output	Printer Magnetic tape Disk drive
SYSLOG	Operator messages and to log job control statements	Printer-keyboard Printer
SYSLNK	Input to the linkage editor	Disk extent Magnetic tape unit
SYSRES	Contains the operating system, the core image library, relocatable library, and source statement library	Tape unit or Area of a disk drive
SYSSLB	The private source statement library	Magnetic tape unit Disk drive
SYSRLB	Private relocatable library	Magnetic tape unit Disk drive
SYSIN	Must be used when SYSRDR and SYSIPT are assigned to the same disk extent. May be used when they are assigned to the same card reader or magnetic tape. This name can only be specified in a job control statement. COBOL SELECT statement must use the other names.	Disk Tape Card reader
SYSOUT	This name must be used when SYSPCH and SYSLST are assigned to the same magnetic tape unit. It must be assigned by the operator ASSGN command.	Tape Disk Punch
SYS000 to SYS222	These names are available to the programmer as work files or for storing data files. These names are called programmer logical units as opposed to the remaining names which are always referred to as system logical units.	Any unit

● Figure 1. Symbolic Names, Their Function, and Permissable Device Type

JOB CONTROL STATEMENTS

Job control statements are read 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.

Statements most likely to be used by the COBOL user are:

Operation	Meaning
JOB	Job name
EXEC	Execute program
ASSGN	Input/output assignments
LBLTYP	Reserve storage for label information
VOL	Volume information
DLAB (Disk only)	Disk file label information
DLBL	Disk file label information
XTENT (Disk only)	Disk file extent
EXTENT	Disk file extent
TPLAB	Tape file label information
TLBL	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 con-
sists 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 state-
ment is that which is discussed here for the label information state-
ments. The following statements can precede the EXEC statement for a
job step and will be frequently used by COBOL programmers.

```
ASSGN                   DLAB
LBLTYP                  XTENT
VOL                     TPLAB
                        DLBL
                        EXTENT
                        TLBL
                        OPTION
                        PAUSE
```

The label statements must be in the order:

```
VOL             VOL
TPLAB    or     DLAB
                XTENT (one for each area or file in volume)
         or     TLBL
                DLBL
                EXTENT (one for each area or file in a volume)
```

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

FORMAT OF JOB CONTROL STATEMENTS

All job control statements are free form, except for a few restrictive rules.

The general format of the job control statements is, as follows:

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:

    a. The end-of-job statement contains /& in columns 1 and 2.
    b. The end-of-file statement contains /* in columns 1 and 2.
    c. The comment statement contains * in column 1 and a blank in column 2.

2. Operation. This field describes the type of control statement. 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.

4. Comments. Comments are permitted anywhere after the trailing blank of the operand field.

CONTINUATION OF JOB CONTROL STATEMENT

Information starts in column 1 and cannot extend past column 71. The exceptions to this are 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 through 15 of the continuation statement are ignored. Begin continuation statement information in column 16.

The ASSGN Statement

The ASSGN statement is used to assign a logical device address to a physical device. The format of the ASSGN statement is as follows:

$$\text{// ASSGN SYSxxx,device-address} \quad \left[\begin{Bmatrix} ,X\text{'ss'} \\ ,ALT \end{Bmatrix} \right]$$

SYSxxx is one of the logical devices listed in Figure 1 (with the exception of SYSOUT, which cannot be assigned by means of ASSGN statements). The system permits programmer logical units in the range from SYS000 to SYS222. The number of units actually permitted in a specific installation is defined at system generation time and, normally, is less than 223. Units SYS000 through SYS009 are the minimum configuration provided by the system.

<u>Device-address</u> permits three different formats:

X'cuu'   where c is the channel number and uu the unit number in
         hexadecimal notation. The values of cuu are determined
         by each installation.

UA   Unassign. The job will be canceled if a file attached
         to this logical unit is referred to by one of the input/
         output statements OPEN, CLOSE, READ, WRITE, or REWRITE.

IGN   Ignore. References to this logical unit are ignored.
         (This does not apply to SYSRDR and SYSIPT.) This format
         may be used in program debugging to ignore files.

<u>X'ss'</u> is the device specification. It is used for specifying mode set-
      tings for 7-track and dual-density 9-track tapes. If X'ss' is not
      specified, the system assumes X'90' for 7-track tapes and X'C0'
      for 9-track tapes. The possible specifications for X'ss' are
      shown in Figure 2.

ss	Bytes per inch	Parity	Translate Feature	Convert Feature
10	200	odd	off	on
20	200	even	off	off
28	200	even	on	off
30	200	odd	off	off
38	200	odd	on	off
50	556	odd	off	on
60	556	even	off	off
68	556	even	on	off
70	556	odd	off	off
78	556	odd	on	off
90	800	odd	off	on
A0	800	even	off	off
A8	800	even	on	off
B0	800	odd	off	off
B8	800	odd	on	off
C0	800	single-density 9-track		
C0	1600	dual-density 9-track		
C0	1600	single-density 9-track		
C8	800	dual-density 9-track		

● Figure 2. Possible Specifications for X'ss' in the ASSGN Statement

<u>ALT</u> indicates an alternate magnetic tape unit that is used when the
capacity of the original assignment is reached. The characteristics of
the alternate unit must be the same as those of the original unit. Mul-
tiple alternates may be assigned to a logical unit.

<u>Note:</u> All device assignments made with ASSGN statements are reset
between jobs to the configuration specified at system generation time
plus any modifications that may have been made by the operator (see the
section "<u>The JOB Statement</u>").

When preparing ASSGN statements for a compilation job step, the pro-
grammer uses the system logical units (SYSIPT, etc.) to refer to input/
output devices used in the system configuration. When preparing the
ASSGN statement for execution time job steps, the programmer uses the
programmer logical units (SYS000 through SYS222) to assign a symbolic
unit to a specified physical device. For example,

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

This example could be used to assign the symbolic unit SYS004 to a card reader at address X'00C'. The first digit specifies the multi-plexor channel and the 0C specifies the unit number.

To specify this file in a COBOL program, the programmer writes

```
SELECT filename ASSIGN TO 'SYS004'
```

Note that only the programmer logical units (SYS000 through SYS222) can be used in a COBOL program.

The EXEC Statement

The execution of a job step is initiated by the following job control statement:

```
// EXEC name
```

Name is the name of the first phase of the program to be fetched for execution from the core-image library. Therefore, execution of a COBOL compilation would be initiated by the statement

```
// EXEC COBOL
```

The name must be omitted if a program that was processed by the linkage editor in the previous job step of the same job is to be executed from SYSLNK.

The JOB Statement

Each job begins with the following job control statement:

```
// JOB job-name
```

Job-name is a user-defined name of 1 to 8 characters, the first of which must be alphabetic.

Note: The JOB statement resets the effect of all previously issued OPTION and ASSGN statements.

LBLTYP Statement

THE LBLTYP statement 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 COBOL program area of main storage. It applies to both background and foreground programs and is required if the file contains standard labels. The format of the LBLTYP statement is:

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

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

VOL Statement

The VOL statement is used to check standard labels for tape or disk
files. It is required for each file on a multiple volume file. The
format of the VOL statement is:

       // VOL SYSxxx,filename

    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
modules, 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.

TPLAB Statement

The TPLAB statement contains file label information for tape label
checking and writing and must immediately follow the VOL statement. The
formats of the TPLAB statements are:

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

    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. These are the only fields used for
        checking the label of an input file. Refer to Appendix C for an
        illustration of standard tape file labels.

    2. When writing output labels, additional fields 11 through 13 may be
        included by use of a continuation statement. (These fields are not
        required for output files.) Refer to the publication IBM
        System/360 Disk Operating System: System Control and System Ser-
        vice Programs, for details about these fields.

OPTION Statement

This statement specifies one or more of the job control options avail-
able. The order in which they appear in the operand field is arbitrary.
The format of the option statement is:

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

where the options are:

LOG   Causes the listing of columns 1 through 80 of all control
statements on SYSLST. Control statements are not listed
until a LOG option is encountered. Once a LOG option state-
ment 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
printed 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. 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 contain-
ing a compiler step.

NOLINK   Suppresses the LINK option. The compiler can also suppress
the LINK option if the program contains an error that would
preclude the successful execution of the program. An EXEC
statement with a blank operand also suppresses the LINK
option.

DECK   Causes the compiler to punch 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.

XREF   Causes the Assembler to writer the symbolic cross-reference
list on SYSLST.

NOXREF   Suppresses the XREF option.

ERRS   Causes the compiler to write the diagnostic messages 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.

MINSYS   Causes the linkage editor to generate minimal modules for
(TOS only)   later runs on systems when linkage editing on systems
greater than 16K.

GO   Indicates that a linkage edited program program exists on
(TOS only)   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   Causes all disk labels submitted after this point to be
(DOS only)   written on the standard label track. Reset to USRLABEL
                option at end-of-job step.

USRLABEL   Causes all disk labels submitted after this point to be
(DOS only)   written at the beginning of the user label track.

SYM   Causes the compiler to print the data division map on
                SYSLST.

PARSTD   Causes all DASD or tape labels submitted after this point to
                be written at the beginning of the partition standard label
                track. Reset to USRLABEL option at end-of-job or end-of-job
                step. All file definition statements submitted after this
                option will be available to any program in the current par-
                tition until another set of partition standard file defini-
                tion statements is submitted. All file definition state-
                ments submitted after OPTION PARSTD will be included in the
                standard file definition set until one of the following
                occurs:

                1. End-of-job step

                2. End-of-job

                3. OPTION USRLABEL is specified.

                4. OPTION STDLABEL is specified.

                For a given filename, the sequence of search for label
                information during an OPEN is the USRLABEL area, followed by
                the PARSTD area, followed by the STDLABEL area.

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 state-
ment, 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

This statement can be used for operator action between jobs. Any mes-
sages to the operator can appear in the operand of a 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 tells the operator 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, B (B = alter code 5), causes processing to continue.

DLAB Statement (DOS only)

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. The format of the DLAB statement is:

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

'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 (DOS only)

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. The format of the XTENT statement is:

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

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 spe-
                cified, 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. Direct files can have up to 5 extents. ISAM files can have up to 11 data extents (9 prime, 1 cylinder index, 1 separate overflow). |

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 and 2314
    0 to 9 for 2321

C_1C_1 is the subcell number. It is equal to:

    00 for 2311 and 2314
    00 to 19 for 2321

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

    000 to 199 for 2311 and 2314
    or strip number:

    000 to 009 for 2321

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

    0 for 2311 and 2314
    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 and 2314

A lower extent of all zeros is invalid.

Note: For 2321, the last 5 strips of subcell 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 alphanumeric character string, contained within apostrophes. 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. If more than one symbolic address is to be specified on separate XTENT cards for the same file, the symbolic addresses must be in consecutive order.

B_2 Currently assigned cell number. Its value is:

0 for 2311 or 2314
0-9 for 2321

This field is optional. If missing, job control assigns $B_2 = B_1$.

The RESET Statement

The RESET statement resets input/output assignments to the standard assignments. The standard assignments are those specified at system generation time plus any modifications made by the operator by means of an ASSGN command (as opposed to using an ASSGN control statement) without the TEMP option. The format of the RESET statement is as follows:

```
                    SYS
// RESET           PROG
                    ALL
                    SYSxxx
```

SYS resets all system logical units to their standard assignments.

PROG resets all programmer logical units to their standard assignments.

ALL resets all programmer and system logical units to their standard assignments.

SYSxxx resets the specified logical unit to its standard assignment.

The End-of-Data-File Statement

The end-of-data-file statement (/* in columns 1 and 2) serves as a delimiter for the input read from the device assigned to SYSIPT. Therefore, COBOL programs must be terminated by an end-of-data-file statement. This statement is also recognized on the programmer logical units that are assigned to a card reader. This causes the ENDFILE condition to be raised for a COBOL input file.

The End-of-Job Statement

The end-of-job statement (/& in columns 1 and 2) indicates that a job has been completed. If this statement is omitted, the Job Control program may skip the next job stacked on the device assigned to SYSRDR and/or SYSIPT. If SYSRDR and SYSIPT are different units, the end-of-job statement must appear on both.

The Comments Statement

A special comments statement (* in column 1 and blank in column 2, followed by the desired comments) is available for longer messages. The comments are printed on SYSLOG, but no halt is effected by this statement.

CBL Statement (COBOL Option Control Card)

Although most options for compilation are specified in the job control
OPTION statement, the COBOL compiler provides an additional option
statement to increase flexibility. The CBL card must be placed between
the EXEC COBOL statement and the first statement in the COBOL program.
The CBL card cannot be continued. However, if specification of options
will continue past column 71, two CBL cards may be used. The format of
this card is:

```
CBL   [DMAP=h]   [,PMAP=h]   [,BUFFSIZ=d]      [,DISPCHK=YES]
                                                        NO
            [,INVED][,NOEXIT=([E],[C])]
```

CBL
: CBL must begin in column 2, be preceded and followed by at least one blank.

DMAP=h
PMAP=h
: DMAP and PMAP specify that the addresses which appear in the coding of both the data division map and the procedure division are to be incremented by the number h. This is only for the listing not for the object module. h is a hexadecimal number of from 1 to 8 digits and is assumed to be zero if one of these options is not specified. If both options are specified, the value for the last h is the value that is used. h is called a relocation factor.

BUFFSIZ=d
: BUFFSIZ specifies the size of the compiler buffer. d is a decimal number from 170 to either 32,760 for magnetic tape or the maximum size of a track for disk. If this option is not specified, 170 is assumed for 16K systems and 1,024 is assumed on 32K or larger systems. Double buffers are always used.

DISPCHK=YES
          NO
: DISPCHK specifies whether or not a diagnostic check is to be made at execution time for displayed items. If YES is specified, the length of all items to be displayed is checked before moving them to the buffer. If an item is too long, it is truncated, but no message is printed. If NO is specified, no check is made and items are moved directly to the buffer. If an item is too long and thus exceeds the buffer size, it will destroy the contents of the storage area following the buffer. The default value is NO.

INVED
: INVED specifies that the roles of the characters "." and "," are to be reversed. This affects report items in the data division, value clauses in the data division, and numeric literals in the procedure division. If this option is not specified, the character "." represents a decimal point and the character "," represents an insertion character.

NOEXIT
: If NOEXIT is specified the compiler does not generate code for program checks. If C is specified, code for program checks is suppressed for CALL coding and if E is specified coding is suppressed for ENTRY coding. Specification of this option will greatly improve the time needed for execution of the program. However, the programmer must be responsible for handling interrupts which occur for an ON SIZE ERROR condition. If this option is not specified, the compiler will handle program checks for arithmetic statements in the program.

Section I: Preparing COBOL Programs for Processing 25

DLBL Statement

This statement replaces the VOL and DLAB statement combination used in previous versions of the system. It contains file label information for DASD label checking and creation. (This release of the system will continue to support the VOL, DLAB, and XTENT combinations currently in use.) The DLBL statement has the following format.

// DLBL	filename,['file-ID'],[date],[codes]
filename	This can be from 1 to 7 characters and is identical to the symbolic name of the program DTF which identifies the file.
'file-ID'	The name associated with the file on the volume. This can be from 1 to 44 bytes of alphanumeric data, contained within apostrophes, including file-ID and, if used, generation number and version number of generatiion. If fewer than 44 characters are used, the field will be left-justified and padded with blanks. If this operand is omitted, "filename" will be used.
date	This can be from 1 to 6 characters indicating either the retention period of the file in the format d-dddd or the absolute expiration date of the file in the format yy/ddd. If this parameter is omitted, a 0-day retention period is assumed. If this operand is present on an input file, it is ignored.
codes	This is a 2 or 3 character field indicating the type of file label, as follows:

SD for Sequential Disk or for DTFPH with
   MOUNTED-SINGLE

DA for Direct Access or for DTFPH with MOUNTED=ALL

ISC for Indexed Sequential using Load Create

ISE for Indexed Sequential using Load Extension, Add,
   or Retrieve

TLBL Statement

This statement replaces the VOL and TPLAB statement combination used in previous versions of the system. (This release will continue to support the VOL, TPLAB combination currently in use.) The TLBL statement contains file label information for tape label checking and writing. Its format follows:

// TLBL	filename,['file-ID'],[date],[file-serial-number], [volume-sequence-number],[file-sequence-number], [generation-number],[version-number]
filename	This can be from 1 to 7 characters and is identical to the symbolic name of the program DTF which identifies the file.
'file-ID'	One to 17 characters, contained within apostrophes, indicating the name associated with the file on the volume. This operand may contain embedded blanks. On output files, if this operand is omitted, the "filename" will be used. On input files, if the operand is omitted, no checking will be done.

date	Four to 6 characters, in the format yy/ddd, indicating the expiration date of the file for output or the creation date for input. (The day of the year may have from 1 to 3 characters.) For output files, a 1 to 4 character retention period (d-dddd) may be specified. If this operand is omitted, a 0 day retention period will be assumed for output files. For input files, no checking will be done if this operand is omitted or if a retention period is specified.
file serial number	One to 6 characters indicating the volume serial number of the first (or only) reel of the file. If less than 6 characters are specified, the field will be right-justified and padded with zeros. If this operand is omitted on output, the volume serial number of the first (or only) reel of the file will be used. If the operand is omitted on input, no checking will be done.
volume sequence number	One to 4 characters in ascending order for each volume of a multiple volume file. This number is incremented automatically by OPEN/CLOSE routines as required. If this operand is omitted on output, BCD 0001 will be used. If i ed i
file sequence number	One to 4 characters in ascending order for each file of a multiple file volume. This number is incremented automatically by OPEN/CLOSE routines as required. If omitted on output, BCD 001 will be used. If omitted on input, no checking will be done.
generation number	One to 4 characters that modify the file-ID. If omitted on output, BCD 0001 is used. If omitted on input, no checking will be done.
version number	One or 2 characters that modify the generation number. If omitted on output, BCD 01 will be used. If omitted on input, no checking will be done.

For output files, the current date will be used as the creation date and "DOS/TOS/360" will be used as the system code.

EXTENT -- DASD Extent Information

The EXTENT command or statement defines each area, or extent, of a DASD file. One or more EXTENT commands or statements must follow each DLBL command or statement except for single volume input files for Sequential Disk or Direct Access, on either a 2311 or a 2314, for which the DEVADDR parameter has been specified in the DTF table.

This command or statement replaces the XTENT command or statement used in previous versions of the system. (Programming support for XTENT will be continued.)

[//] EXTENT,[symbolic-unit],[serial-number],
    [type],[sequence-number],[relative-
    track],[number-of-tracks],[split-
    cylinder-track],[B=bins]

Accepted by SPI

// EXTENT[symbolic-unit],[serial-number],
    [type],[sequence-number],[relative-
    track],[number-of-tracks],[split-
    cylinder-track],[B=bins]

Accepted by JC

symbolic unit	A 6-character field indicating the symbolic unit (SYSxxx) of the volume for which this extent is effective. If this operand is omitted, the symbolic unit of the preceding EXTENT will be used. (This operand is not required for an IJSYSxx filename or for a file defined with the DTF DEVADDR=SYSnnn.)
serial number	From 1 to 6 characters indicating the volume serial number of the volume for which this extent is effective. If fewer than 6 characters are used, the field will be right-justified and padded with zeros. If this operand is omitted, the volume serial number of the preceding EXTENT will be used. If no serial number was provided in the EXTENT command or statement, the serial number will not be checked and it will be the user's responsibility if files are destroyed due to mounting the wrong volume.
type	One character indicating the type of the extent, as follows: 1 - data area (no split cylinder) 2 - overflow area (for indexed sequential file) 4 - index area (for indexed sequential file) 8 - data area (split cylinder) If this operand is omitted, type 1 will be assumed.
sequence number	One to 3 characters containing a decimal number from 0 to 255 indicating the sequence number of this extent within a multi-extent file. 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 the sequence number 1. The extent sequence number for all other types of files begins with 0. If this operand is omitted for the first extent of ISFMS files, the extent will not be accepted. For SD or DA files, this operand is not required.
relative track	One to 5 characters indicating the sequential number of the track, relative to zero, where the data extent is to begin. If this field is omitted on an ISFMS file, the extent will not be accepted. This field is not required for DA input or for SD input files (the extents from the file labels will be used).

Formulas for converting actual to relative track (RT) and relative track to actual for the DASD devices follow.

Actual to Relative

2311 10 x cylinder number + track number = RT
2314 20 x cylinder number + track number = RT
2321 1000 x subcell number + 100 x strip number +
      20 x block number + track number = RT

Relative to Actual

2311 $\dfrac{RT}{10}$ = quotient is cylinder, remainder is track

2314 $\dfrac{RT}{20}$ = quotient is cylinder, remainder is track

2321 $\dfrac{RT}{1000}$ = quotient is subcell, remainder1

$\dfrac{remainder1}{100}$ = quotient is strip, remainder2

$\dfrac{remainder2}{20}$ = quotient is block, remainder is track

                    Example: Track 5, cylinder 150 on a 2311 = 1505 in relative
                    track.

number of   One to 5 characters indicating the number of tracks to be
tracks   allotted to the file. For SD input or DA input, this field-
             may be omitted. For split cylinders, the number of tracks
             must be an even multiple of the number of tracks per cylinder
             specified for the file.

split   One or 2 characters, from 0-19, indicating the upper track
cylinder   number for the split cylinder in SD files.
track

bins   One or 2 characters identifying the 2321 bin that the extent
             was created for or on which the extent is currently located.
             If the field is one character, the creating bin is assumed to
             be zero. There is no need to specify a creating bin for SD
             or ISFMS files. If this operand is omitted, bin zero is
             assumed for both bins. If the operand is included and posi-
             tional operands are omitted, only one comma is required pre-
             ceding the key-word operand. (One comma for each omitted
             positional operand will be acceptable, but not necessary.)

THE LINKAGE EDITOR

The linkage editor prepares an object module for execution. It can also
be used to combine two or more separately compiled object modules into a
format suitable for execution. The output of the linkage editor con-
sists of one or more program phases.

    If linkage editor processing is desired, the job control OPTION
statement specifying the LINK or CATAL option must precede the first
linkage editor control card and the first EXEC statement in the job.
The linkage editor is called for processing by specifying LNKEDT in an
EXEC statement. Processing by the linkage editor is suppressed if
severe programming errors are detected during compilation.

    Input to the linkage editor may consist of any combination of the
following:

1. Object modules compiled in previous job steps (SYSLNK)

2. Object modules from the relocatable library (SYSRES) or (SYSRLB)

3. Object modules in the form of card decks (SYSIPT)

    Output from the linkage editor is placed in the core image library as
a permanent member if the CATAL option has been specified on the job
control OPTION statement. If CATAL has not been specified, the program
phase is placed in the temporary part of the core image library for DOS
or on SYSLNK for TOS.

LINKAGE EDITOR CONTROL STATEMENTS

The execution of the linkage editor is initiated by linkage editor con-
trol statements read from SYSRDR. The general format of linkage editor
control statements is similar to that of the job control statements
except that the linkage editor control statements must have a blank in
column 1 instead of // in columns 1 and 2. The PHASE card and the
INCLUDE card are of special interest in preparing object modules to be
linkage edited.

The PHASE Statement

The PHASE statement must be specified if the output of the linkage edi-
tor is to consist of more than one phase or if the program phase is to
be cataloged in the core image library. Each object module that is to
begin a phase must be preceded by a PHASE statement of the following
format:

        PHASE phase-name,origin

    The phase-name is the name under which the program phase is to be
cataloged. This name does not have to be the program-id name, and in
the case of overlay it should not be the same. It must consist of 1 to
8 alphanumeric characters, the first of which must be alphabetic.

    The origin indicates to the linkage editor the begin address of this
specific phase. An asterisk may be used as an origin specification to
indicate that this phase is to follow the previous phase or the Supervi-
sor at the next double-word boundary. This simple format of the PHASE
statement covers all applications that do not include setting up overlay
structures. See Section VII: Subprograms and Overlay for information
on the PHASE statement for overlay applications.

The INCLUDE Statement

The INCLUDE statement must be specified for each object module deck or
object module in the relocatable library that is to be included in a
program phase. The format of the INCLUDE statement is as follows:

        INCLUDE [module-name]

        module-name is not specified when the module to be included is in
                    the form of a card deck being entered from SYSIPT. If
                    the object module is being included from the relocat-
                    able library, the module-name is the name under which
                    the module was cataloged in the library.

THE AUTOLINK FEATURE

If any references to external names are still unresolved after all
modules have been read from SYSLNK, SYSIPT, and/or the relocatable
library, the autolink feature of the linkage editor searches the relo-
catable library for module-names identical to the unresolved names and
includes these modules in the program phase. This feature is required
in which to include COBOL subroutines that are cataloged in the relocat-
able library.

    Examples:

 1. Linkage edit one object module compiled in a previous job step.

```
// JOB        N0123
// OPTION     LINK
   PHASE      EXAMPLE,*
// EXEC       COBOL
     COBOL
     source module
/*
// EXEC       LNKEDT
// EXEC
     data

/*
/&
```

2. Linkage edit three object modules and catalog the phase (one module
    from previous compilation, one from SYSIPT and one named R from the
    relocatable library).

```
// JOB
// OPTION     CATAL
   PHASE      EXAMPLE,*
// EXEC       COBOL
     COBOL
     source module
/*
   INCLUDE
     (object module)
   INCLUDE R
/*
// EXEC       LNKEDT
/&
```

LIBRARIAN FUNCTIONS

The service program called the librarian takes care of all maintenance
functions (such as adding, deleting, and copying or renaming) for the
system libraries. Cataloging (which simply means adding) of frequently
used program phases, object modules, or source language statements in
one of the system libraries greatly reduces the time required for card
reading and linkage editor processing. Object modules are cataloged in
the relocatable library. Program phases are cataloged in the core image
library. Source module statements are cataloged in the source statement
library. Each sequence of source statements cataloged in the source
statement library is called a book.

   The name of a phase, module, or book must be unique for each library.
When a phase, module or book is cataloged in a library, any module,
phase, or book already contained in the respective library and having
the same name is automatically deleted. This necessitates some naming
conventions for each installation in order to prevent a programmer from
unintentionally deleting programs that are part of the library.

   A complete description of the library maintenance functions and deck
set ups used to specify them is included in the publication IBM System/
360 Tape or Disk Operating System: System Control and System Service
Programs. The following discussions show how to catalog.

CATALOGING PROGRAM PHASES--CORE IMAGE LIBRARY

If a program is to be cataloged in the core image library, the job control statement // OPTION with the CATAL option must be given prior to linkage editor processing and it must precede the first PHASE card of the program to be cataloged in the case of compile and link edit runs. Upon successful completion of the linkage editor job step, the program phase(s) are automatically cataloged. The program phase can be executed in the next job step in the same job by specifying the // EXEC statement with a blank name field. When it is executed in a subsequent job, the EXEC statement that calls for execution must specify the name under which it has been cataloged. Note that the phase is cataloged under the name specified in the PHASE statement. The following is an example of cataloging a single phase in the core image library.


```
// JOB FOURA                      // JOB FOURB
// OPTION CATAL                   // OPTION CATAL
   PHASE FOURA,*                     PHASE FOURB,*
   INCLUDE                           INCLUDE MOD4B
                                  // LBLTYP NSD(nn)  or TAPE
                                  // EXEC LNKEDT
      object deck                 /&
/*
// LBLTYP NSD(nn)  or TAPE
// EXEC LNKEDT
/&
```


CATALOGING OBJECT MODULES--RELOCATABLE LIBRARY

   Object modules are cataloged in the relocatable library in a job step that specifies execution of a library maintenance program. This program is called by specifying MAINT in the operand field of the EXEC statement. Each object module to be cataloged must be preceded by the CATALR control statement. The format of this statement is:

      CATALR module-name

where:

module-name
      must be used in the linkage editor INCLUDE statement. The module may be preceded but not followed by linkage editor control statements.

   The object modules named in CATALR statements must be on SYSIPT. If a programmer wishes to compile and catalog in one job, he could assign SYSPCH to a magnetic tape drive and reassign the same drive to SYSIPT for the catalog step. This would eliminate unnecessary card handling.

   Note that CATALR statements are read from SYSIPT for DOS and SYSRDR for TOS. Therefore, the CATALR statements must be put on SYSRDR in the same sequence as the object modules on SYSIPT. The following is an example of cataloging two modules in the relocatable library.

```
// JOB EIGHT
// EXEC MAINT
   CATALR MOD8A


     object deck

   CATALR MOD8B


     object deck

/*
/&
```

The following is an example of compiling and cataloging a module in the relocatable library.

```
// JOB NINE
// ASSGN SYSPCH,X'183'  (TAPE DRIVE)
// OPTION DECK
// EXEC COBOL


   source deck

/*
   CLOSE SYSPCH,X'00D'  (CARD PUNCH)
// ASSGN SYSIPT,X'183'  (TAPE ABOVE)
// EXEC MAINT
   CATALR MOD9
/*
/&
```


CATALOGING BOOKS--SOURCE STATEMENT LIBRARY

Frequently used data division, environment division, and procedure division entries can be cataloged in the source statement library. A book in the source statement library might consist, for example, of a file description or a paragraph of the procedure division. Such source language statements are cataloged in the source statement library by using the library maintenance program MAINT. Each part to be entered must be preceded by a control statement of the format:

    CATALS C.library-name

   In addition, a control statement of the form BKEND C.library-name must precede and follow the book to be cataloged. Note that the CATALS statement is read from SYSIPT for DOS and SYSRDR for TOS but the BKEND statements are entered on SYSIPT before and after the book. The library-name must follow the rules for external-names in the COBOL language.

   The following is an example of cataloging a file description in the source statement library.

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

This file description can be included in a COBOL source module by writing the following statement:

    FD FILEB COPY 'FILEA'

Note that the library entry does not include FD or the filename. It begins with the first clause that is actually to follow the filename. This is true for all options of COPY or INCLUDE. However, data entries in the library may have level numbers (01 or 77) identical to the level number of the data name that precedes the COPY clause. In this case, all information about the library data-name is copied but the library data-name and all references to it are replaced by the data-name in the program. For example, assume the following data entry is cataloged under the library name DATAR.

        01   PAYFILE USAGE IS DISPLAY.
              02 CALC PICTURE 99.
              02 GRADE PICTURE 9
                  OCCURS 1 DEPENDING ON CALC OF PAYFILE.

and the following statement is written in a COBOL source module

    01 GROSS COPY 'DATAR'

The compiler interprets this as:

    01 GROSS USAGE IS DISPLAY.
        02 CALC PICTURE 99.
        02 GRADE PICTURE 9
                OCCURS 1 DEPENDING ON CALC OF GROSS.

Note also that the library-name is used to identify the book in the library. It has no other use in the COBOL program.

For both the relocatable library and the source statement library, several library maintenance operations can be performed in one job step. Except in the case of adding, this is also true for the core image library.

CATALOGING BOOKS--USER PRIVATE LIBRARY

The procedure for cataloging books in a private library is the same as the procedure for cataloging books in the source statement library except that the logical device SYSSLB must be assigned and defined by DLBL, EXTENT or VOL, DLAB, and XTENT control statements. SYSSLB is the logical device used for private libraries.

A private library is defined by the CORGZ program. The following example defines a private library on physical unit 191.

    // JOB PRIVLIB
    // ASSGN SYSSLB,X'191'
    // DLBL IJSYSSL, 'DATASET ID', DATE INFORMATION, SD
    // EXTENT extent information
    // EXEC CORGZ
            NEWVOL SL=cylin (tracks)
    /*
    /&

where:

    cylin=number of cylinders allocated to the library and tracks=
          number of tracks allocated to the directory.

To use this private library for COPY and INCLUDE, the ASSGN, DLBL, and EXTENT job control statements which define the private library must be included in the deck structure for compilation. When these cards are present, a search for the book is made in the private library. If it is not there, the system library is searched. If the cards for the private library are not there, only the system library is searched. A programmer may create several private libraries, but only one private library can be used in a given job.

CHECKPOINTING A PROGRAM

When a program is expected to run for an extended period of time, provision should be made for taking checkpoint information periodically during the run. This information describes the status of the job and the system (main storage, input/output status, general and floating-point registers) at the time the records are written. Thus, it provides a means of restarting at a checkpoint position rather than at the beginning of the entire job, if processing is terminated for any reason before the normal end-of-job. Checkpoints are taken using the COBOL RERUN statement.

In designing a program for which checkpoints are to be taken, the user should consider the fact that, upon restarting, the program must be able to continue as though it had just reached that point in the program at which termination occurred. Hence, the user should ensure that:

1. File handling is such as to permit easy reconstruction of the status of the system as it exists at the time of each checkpoint. For example, when multifile reels are used, the operator should be informed (by message) as to which file is in use at checkpoint time. He requires this information at restart time.

2. The contents of files are not significantly altered between the time of the checkpoint and the time of the restart:

   • For sequential files, all records written on the file at checkpoint time should be unaltered at restart time.

   • For nonsequential files, care must be taken to design the program so that a restart will not duplicate work that has been completed between checkpoint time and restart time. For example, suppose that Checkpoint 5 is taken. By adding an amount representing interest due, account XYZ is updated on a direct-access file that was opened with the input/output clause. If the program is restarted from Checkpoint 5 and if the interest is recalculated and again added to account XYZ, incorrect results will be produced.

If the program is modular in design, RERUN statements must be included in all modules that handle files for which checkpoints are to be taken. (When an entry point of a module containing a RERUN statement is encountered, a COBOL subroutine -- IHD03800 -- is called. IHD03800 enters the files of the module into the list of files to be repositioned.) Repositioning to the proper record will not occur for any files that were defined in modules other than those containing RERUN statements. Moreover, a restart from any given checkpoint may not reposition other tapes on which checkpoints are stored. Note, too, that only one disk checkpoint file can be used.

Restarting a Program: If the programmer includes checkpoints in his job by means of the COBOL RERUN statement, the message

OC00I CHKPT nnnn HAS BEEN TAKEN ON SYSxxx

is given each time a checkpoint is taken. (nnnn is the 4-character identification of the checkpoint record.) To restart a job from a checkpoint, the following actions are required:

1. Replace the // EXEC statement with a // RSTRT statement whose format is:

    // RSTRT SYSxxx,nnnn [,SYSxxx]

    where:

    SYSxxx is the symbolic name of the device on which the checkpoint records are stored

    nnnn is the 4-character identification of the checkpoint record to be used for restarting

    SYSxxx can be any value from SYS000 through SYS222. (If the checkpoint records are recorded on a direct-access device, SYSxxx must be repeated following the four-character identification of the checkpoint record.) All other job control statements applicable to the job step should be the same as when the job was originally run. If necessary, the channel and unit addresses for the // ASSGN statements may be changed.

2. Rewind all tapes used by the program being restarted and mount them on devices assigned to the symbolic units required by the program. (If multireel files are involved, mount (on the primary unit) the reel in use at the time that the checkpoint was taken and rewind it. If multifile reels are involved, position the reel to the start of the file referred to at the time of the checkpoint.)

3. Reposition any card file so that only cards not yet read when the checkpoint was taken are in the card reader.

4. Execute the job.

    Disk and Tape Operating System COBOL provides a linkage to the system checkpoint routine and re-entry point to the COBOL program from the system restart routine. Therefore, any restrictions applying to these system routines also apply to COBOL's restart procedures except as noted above. These restrictions are outlined in the publications IBM System/360 Disk Operating System: Supervisor and Input/Output Macros and IBM System/360 Tape Operating System: Supervisor and Input/Output Macros.

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 pro-
cessing 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 input/output 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 dis-
cussed, 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 7-track tape.

```
                                                    /&
                                                  // EXEC
                                                Program Assignments
                                              // ASSGN SYS...
                                            // EXEC LNKEDT            If required
                                          // EXEC COBOL
                                        PHASE
                                      // OPTION LINK...
                                    // ASSGN SYSRLB
                                  // ASSGN SYSSLB
                                // ASSGN SYS003
                              // ASSGN SYS002          If libraries on
                            // ASSGN SYS001            private tapes
                          // ASSGN SYSLNK
                        // ASSGN SYSLST
                      // ASSGN SYSPCH
                    // ASSGN SYSIPT                If different from
                  // DATE                          standard assignments
                // JOB
                                           Optional
                    SYSRDR
```

Data
Cards
(if required)

COBOL
Source
Statements

SYSIPT

System
Tape

SYSLOG

SYS001
SYS002
SYS003

System/360

SYSLST

SYSLNK
(Optional)

SYSPCH
(Optional)

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

Figure 3.   Input/output 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 phys-
ical 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 fol-
lowing his JOB card. Examples of overriding assignments are given in
the text that follows. In the examples that follow, whenever an option-
al statement is used it is identified by the words (optional card).

EXAMPLES OF PROCESSING USING TAPE CONFIGURATION

EXAMPLE 1--COMPILE AND PUNCH

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.

EXAMPLE 2--CATALOGING AN OBJECT MODULE IN RELOCATABLE LIBRARY

In this example, an object module generated by the compiler (see Example
1) is cataloged in 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 a "new" SYSRES. SYS-002 can then be reassigned.

   SYS001 is used as a work file.

EXAMPLE 3--COMPILE, LINKAGE EDIT, AND EXECUTE

This example illustrates how an object module cataloged in 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,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 units are required to execute the linkage editor.

EXAMPLE 4--EXECUTING A PROGRAM

The job control statements required simply to 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 making test runs, where many runs might be made with different sets of data decks.

EXAMPLE 5--CATALOGING SOURCE MODULES TO SOURCE STATEMENT LIBRARY

The procedural steps and the job control statements required to catalog two source statement routines in 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,                           |
LABEL RECORDS ARE STANDARD, BLOCK                 }    ROUTINE 1
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 in 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.

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

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--
capacitors and inductors--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 the source statement
library. Assume its records are, as follows:

```
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 pro-
gram 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, linkage edit, and execute the problem are:

```
// JOB INVNTORY
// OPTION LINK,LIST,DUMP
   PHASE INVNTORY,*
// EXEC COBOL
            .
            .
            .
   DATA DIVISION.              (see
   FD FILEB COPY 'DATAIN'.     Example 5
            .                  for expansion)
            .
            .
   PROCEDURE DIVISION.
            .
            .
            .
   START.  INCLUDE 'INOUT'.    (see
            .                  Example 5
            .                  for expansion)
            .
   PROCESS.   (Records on FILEB are processed)


/*
// LBLTYP TAPE (03)
// EXEC LNKEDT
// ASSGN SYS0004,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 'DATAIN,etc...'
// VOL SYS007,SYS007,
// TPLAB 'MASTER,etc...'
// VOL SYS005,SYS005
// TPLAB '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 cata-
loged 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 input/ output 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 in the relocatable library

3. Compiling, linkage editing, and executing

4. Executing a previously linkage edited program

5. Cataloging in 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 in the relocatable library, whereas example 6 illustrates how COBOL copies source statement modules cataloged to the source statement library.
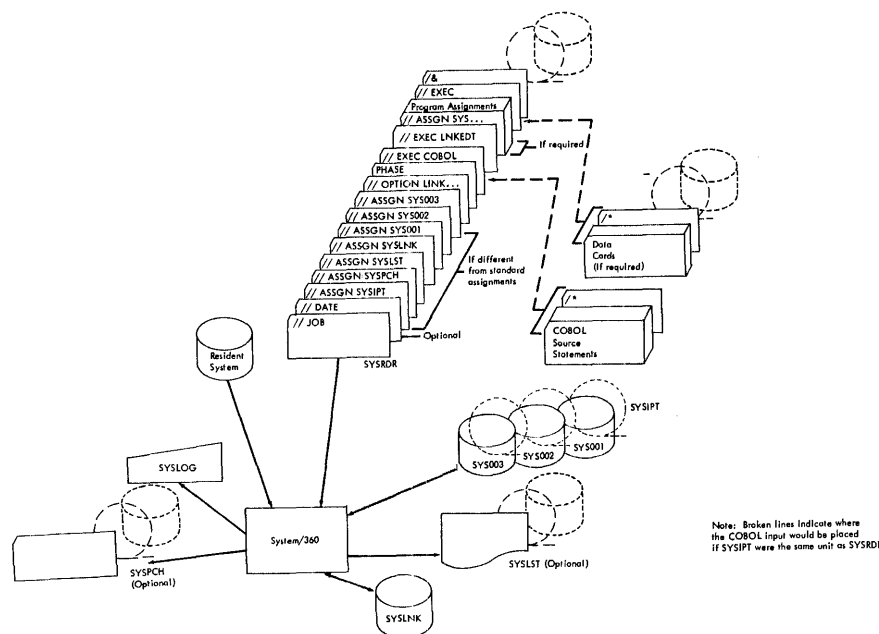
Figure 4. Input/Output Units Used by COBOL Program in a Disk System

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

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 7-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 SYS-001 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 SYS-001, 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 labels 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 EXTENT 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
```

EXAMPLE 1--COMPILE AND PUNCH

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         │                      ⎬  Input from SYSIPT
     │SOURCE STATEMENTS│                      ⎭
     └─────────────────┘
/*
/&                                            ⎫  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.

EXAMPLE 2--CATALOGING AN OBJECT MODULE IN RELOCATABLE LIBRARY

In this example, an object module generated by the compiler
(see 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)
```


EXAMPLE 3--COMPILE, LINKAGE EDIT, AND EXECUTE

This example illustrates how an object module cataloged to the relocat-
able libeary 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

EXAMPLE 4--EXECUTING A PROGRAM

The job control statements required simply to execute a program, assum-
ing it has been cataloged in 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 making 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.

EXAMPLE 5--CATALOGING SOURCE MODULES IN SOURCE STATEMENT LIBRARY

The procedural steps and the job-control statements required to catalog
two source statement modules in 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,                               |   ROUTINE 2
   LABEL RECORDS ARE STANDARD, BLOCK                     }
   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.

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

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
   direct-access storage.

   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--
capacitors and inductors--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 than an input update file called DATAIN (Example 5,
ROUTINE 2), was created on disk and cataloged to the source statement
library. Assume that the records are, as follows:

```
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 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 PICUTRE 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, linkage edit, and execute the problem are:

```
// JOB INVNTORY
// OPTION LINK,LIST,DUMP
   PHASE INVNTORY,*
// EXEC COBOL
          .
          .
          .
   DATA DIVISION.              (see
   FD FILEB COPY 'DATAIN'.     Example 5
          .                    for expansion.)
          .
          .
   PROCEDURE DIVISION.
          .
          .
          .
   START. INCLUDE 'INOUT'.     (see
          .                    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'.
// TLBL DATAIN
// DLBL SYS005, 'THIS IS THE NEW JOB MASTER FILE etc., ...'
// EXTENT Enter the track specification here ...
// DLBL SYS007, 'THIS IS THE JOB(OLD) MASTER FILE etc., ...'
// EXTENT 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 in 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.

SECTION IV: INTERPRETING OUTPUT

The compiler, linkage editor, COBOL program phases, and other system components can produce output in the form of printed listings, punched card decks, diagnostic or informative messages, and data files directed to tape or direct-access devices. This section describes the output listings that can be used to document and debug programs. Included are explanations of compiler output, a list of conditions that can cause a dump, a brief discussion of how to best use a dump, and an explanation of how diagnostic messages are determined. A complete list of diagnostic messages is contained in Appendix H.

COMPILER OUTPUT

The output of the compilation job step may include:

- A printed listing of the control statements

- A printed listing of the statements contained in the source module

- A printed listing of a data map

- A printed listing of a procedure map

- Compiler diagnostic messages

- An object module

All forms of output must be requested by means of the job control OPTION statement. For example, DECK specifies that the object module is to be punched. A complete list of the options for this statement is given in Section I. All output to be listed is printed on the device whose symbolic name is SYSLST.

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 statement. The listing is given on SYSLST.

> GENERATED COBOL SOURCE LISTING (The heading that appears at the top of the listing is explanatory only. It does not actually appear on the listing.)

The source listing consists of:

LINE NO.   A compiler generated line number that is shown in the left-most column. This line number is used in diagnostic messages and LISTX references. The generated line numbers for the sample program are 1 through 39.

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

SOURCE   All COBOL words and punctuation. Words, punctuation, and
STATEMENT 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.

S   Sequence numbers out of order. If columns 1 through 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.

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

*   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 statement. The data map is printed on the SYSLIST unit.

The data map shows the name of each nonprocedure name defined in the program. File-names, record-names, and condition names are identified in the column headed TYPE. (In this 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 level 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 nonprocedure name specified in the program.

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.

```
r--------------------------------------------------------------------------
|                   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 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.                                    |
L--------------------------------------------------------------------------
```
Figure 5.   Example of a COBOL Source Listing

```
r--------------------------------------------------------------------------
|                        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                               |
L--------------------------------------------------------------------------
```
Figure 6.   Example of a Data Map

Figure 7 is an example of a procedure map. It is a portion of the pro-
cedure 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
printed on the SYSLST unit. 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 instruc- tion(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 each element in
                        the line from left to right (for definigion of an
                        element, see "e (for definigion of an element, see
                        "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 the fact that
                        "28 06" appears 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 by specifying PMAP=h on
                        the COBOL CBL option statement.

INSTRUCTION   Contains the machine language instruction (in hexa-
                        decimal) 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	00329B	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

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 diagnostic messages 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	Explanation
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 message was discovered or the basic area that was involved, such as ALIGNMENT, FD, or similar items.

MESSAGE

The actual message is given here. These messages are listed in Appendix H.

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

Figure 8. Example of Source Module Diagnostics

Working with Diagnostic Messages

1. Handle the diagnostic messages in the order in which they appear on the source listing. It is possible to get compound diagnostic messages. Frequently, an earlier diagnostic indicates the reason for a later diagnostic message. 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 an apparently valid clause to be diagnosed as invalid because it is not complete, or because it 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. Diagnostic messages 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. Errors uncovered in this manner can produce a slightly different message 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 that appear 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, which 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 message in one case than in another.

If a given compilation produces more than 25 diagnostic messages, they are presented in a batched sequence. The first 25 messages are sorted in order, followed by the second series, which is also sorted in order.

If an error occurs after the 4095 source statement, the line sequence of the source statement in error can usually be determined by adding 4095 to the sequence number given in the diagnostic message. A message frequently suggests the divison of a COBOL source program in which the error occurred.

Examples of How Diagnostic Messages 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 <u>MOVE</u> data-name   <u>TO</u> data-name ...
                      literal

Error 1   MOVE FIELDA TOO FIELDB
023

                ERROR #178

                INSERT1 TO   Information
                                                  passed to
                INSERT2 TOO   diagnostic
                                                  out of phase I19

                Skeleton Message #178 CSYNTAX REQUIRES WORD "Insert1". FOUND "Insert2".

                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 produces diagnostic messages, console messages and a storage map. For a description of output and error messages from the linkage editor see the IBM publications <u>IBM System/360 DOS System Control and System Service Programs</u> and <u>IBM System/360 TOS System Control and System Service Programs</u>.

EXECUTION TIME MESSAGES

When an error condition that is recognized by compiler generated code occurs during execution, an error message is written on SYSLST or SYS-LOG. Any messages normally written on SYSLST that result from an error

in the foreground program are written on SYS000. Messages that normally appear on SYSLOG are provided with a code indicating whether the message originated in a foreground or background program. These messages and their descriptions are listed in Appendix H.

PROGRAM PHASE DUMPS

Execution of a program phase may produce a 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.

Examples of COBOL language errors that can cause a dump follow.

1. A GO TO statement with no procedure name following it may have been in properly initialized with an ALTER statement. The execution of this statement will cause an invalid branch.

2. Arithmetic calculations or moves on numeric fields that have not been properly initialized can cause an interrupt and a dump.

   For example, neglecting to initialize an OCCURS...DEPENDING ON clause, or referencing data fields prior to the first read may cause an interrupt and a dump.

3. Invalid data placed in a numeric field as a result of redefinition.

4. Input/output errors that are nonrecoverable.

5. Subscripts whose values exceed the defined maximum value will, when moved into the procedure division, can destroy machine instructions in the program.

6. Attempting to execute an invalid operation code through a systems error or invalid program.

7. Generating an invalid address for 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 an ENTRY using a data name.

10. Incorrect tape record length. Causes the compiler to generate an invalid supervisor call SVC32. This inititates 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.

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 infromation as to the specific nature of the error. A more detailed 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, as follows:

* COBOL subroutines

* Working storage data items

* Edit masks

* DTF tables

* Buffers

* Procedure literals

* Work area and global table

* Instructions

* Input/output subroutines

* Subprograms

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 COBOL program or in a compile-time debugging packet. Their formats and a description of their use is contained in the publication IBM System/360 DOS-TOS COBOL Language Specifications. 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 G. These messages reflect errors in the debug packet(s) only. They are not associated with compiling.

TRACE STATEMENT

When a job does not execute properly 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 is passed to 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 STATEMENT

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 to 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 STATEMENT

It is possible, where large volumes of data are involved, to sample spe-
cific portions of a program by use of the ON statement. The ON state-
ment 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. This type of example could also have
been done by a counter or a PERFORM statement, but this method is
easier.

Note: An ON statement with an UNTIL or ELSE option cannot be used in an
IF statement.

THE DEBUG PACKET

The debug packet can be used only 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
with the COBOL source module before compilation begins. Where the pack-

et 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 is required by job control, because SYSIPT 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.
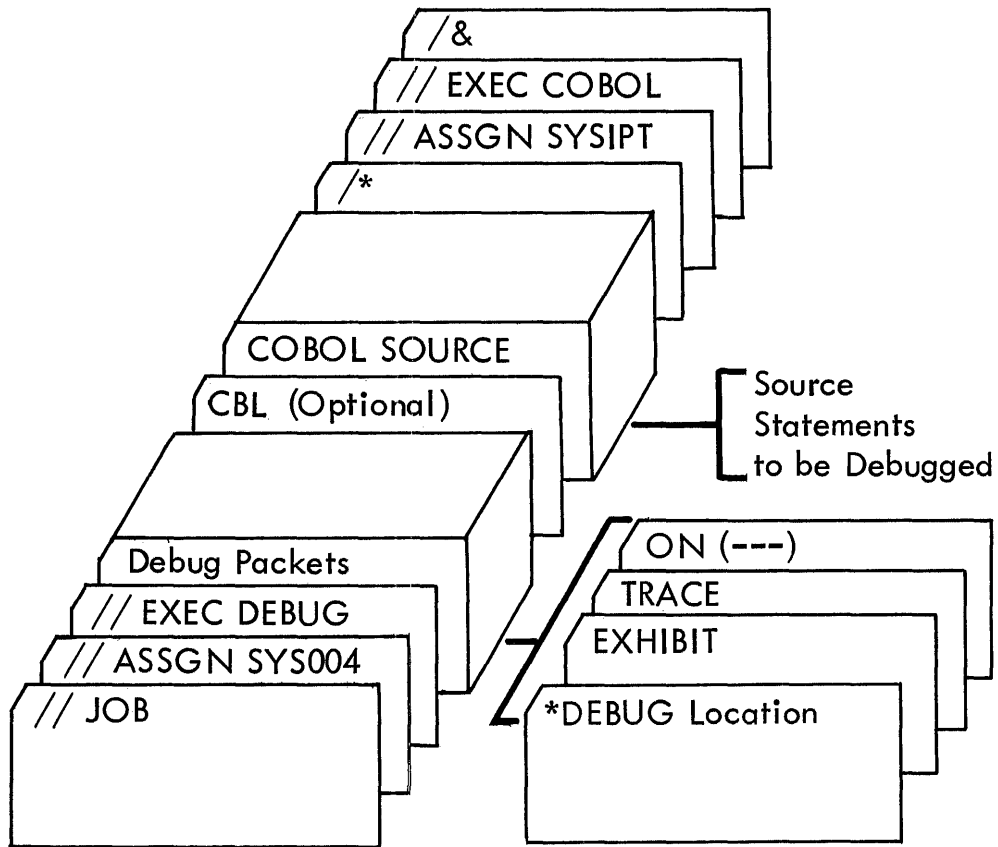
Figure 9. Example of a Debug Packet

This section is intended to aid the programmer in preparing efficient
COBOL programs. Also included are discussions on error declaratives,
variable length records, and processing buffers.

CONSERVING STORAGE

Entries in the data division can significantly affect the amount of core
storage required by the program.

   For example, saving one byte in the data division can cause a signi-
ficant 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 saving 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
will resolve all of the allowable mixed data usages encountered. The
required additional instructions are generated and additional storage is
used.

   However, if a programmer uses the methods illustrated in the follow-
ing discussions, a significant amount of storage can be saved. For
example, attention to decimal alignment saves storage, as follows:

1. 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.

2. 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 following
    COBOL statements.

   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 (arithmetic calculations, input/output, PERFORMS,
          etc.)   - 15%

   Assume that the number of move statements, out of a total of 250 pro-
cedural 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 simply by 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 examples given in the following discussion illustrate efficient coding techinques to conserve core storage.

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

To get efficient code, the programmer should align decimal points wherever possible. If he cannot align data in the data division, he can move it to a work area. 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. No additional bytes are required for field B. (Note that System/360 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 alphanumeric fields in order to pad out to the proper length. To avoid these operations, the number of integer digits in fields used together should be 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 inserting zeros in the high-order positions (numeric fields are justified right) is 10 bytes. To eliminate these 10 bytes, SENDFLD should be defined 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 longer 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 comparison. 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 one-time 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 statement are defined:

    A COMPUTATIONAL-1.
    B PICTURE S99V9 COMPUATIONAL-3.
    C PICTURE S9999V9 COMPUATIONAL-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 format) 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 Arithmetic Calculations: Before arithmetics are performed, DISPLAY data is converted to COMPUTATIONAL-3 data format.

DISPLAY to COMPUTATIONAL

To Execute a MOVE: Before the MOVE is executed, DISPLAY data is converted to COMPUTATIONAL-3 data 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 Arithmetic Calculations: Before arithmetic calculations 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 Arithmetic Calculations: Before arithmetics calculations 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 Arithmetic Calculations: Before arithmetic calculations 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 Arithmetic Calculations: Before arithmetic calculations 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 Arithmetic Calculations: Before arithmetic calculations 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 Arithmetic Calculations: Before arithmetic calculations 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 COMPUTATIONAL-2 Data

For efficient object code, use of floating-point (COMPUTATIONAL-1 or
COMPUTATIONAL-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 format, causing
the object program to perform conversions. For example, assume a COM-
PUTE is specified as:

        COMPUTE A = B * C + D + E.

Assume B is COMPUTATIONAL-1 or COMPUTATIONAL-2 data and all other fields
are defined as COMPUTATIONAL-3 data. Fields C, D, and E are converted
to COMPUTATIONAL-1 or COMPUTATIONAL-2 data format, the calculation per-
formed, and the result converted back from COMPUTATIONAL-1 or
COMPUTATIONAL-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 com-
putational data that is practically all COMPUTATIONAL-1 or
COMPUTATIONAL-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 compiler attempts to ensure that a special
positive sign (F) is present so that the values are treated as absolute.

   The compiler moves in a hexadecimal F whenever the possibility of the
sign changing exists. Examples are: substracting unsigned fields, mov-
ing 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 statements 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 8 decimal positions, the final result actually obtained in this example contains 6 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 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 comparisons, the shorter item must be moved to a work area where zeros or blanks are inserted before the COMPARE statement can be executed.

COMPUTATIONAL-3 (Internal Decimal) Fields

The decimal feature provides for the automatic insertion of high-order zeros on additions, subtractions, and comparisons.

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 number 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 COMPUTATIONAL-2 Fields

A full set of short- and long-precision instructions is provided that 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 for-
mat (or mode). Figure 10 illustrates the number of bytes required for
each class of elementary items.

   If files and working storage are organized so that all halfwords,
fullwords, and double words are grouped together, essentially no addi-
tional storage is used. However, if these items are not grouped togeth-
er properly, the amount of storage required for alignment is:

                Halfword - 1 byte
                Fullword - 1 to 3 bytes
             Double word - 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 ⎬       Bytes = Number of characters in picture             │
│ ⎝ point    ⎠                                                           │
│                                                                        │
│  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                      │
│                                                                        │
│                        ⎛ 2 if 1≤N≤4    Halfword Machine Address         │
│  {Binary} Bytes =      ⎨ 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      ⎛ 4 if short- ⎞                                  │
│COMPUTATIONAL-2         ⎜ precision    ⎜ Fullword Machine Address        │
│ ⎛ Internal ⎞           ⎜ (computa-    ⎜                                 │
│ ⎨ floating ⎬ Bytes =   ⎨ tional-1)   ⎬                                  │
│ ⎝ point    ⎠           ⎜ 8 if long   ⎜                                  │
│                        ⎜ precision    ⎜ Double word Machine Address     │
│                        ⎜ (computa-    ⎜                                 │
│                        ⎝ tional-2)   ⎠                                  │
└──────────────────────────────────────────────────────────────────────┘
```
Figure 10. Number of Bytes Required for Each Class of Elementary Item

It is possible for two group items defined in exactly the same way 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 6 bytes (it does not contain slack bytes).

Case 2 group (02 MINERALS) consists of a total of 8 bytes, including 2 slack bytes.

In case 2, 03 QUARTZ will be preceded by 2 slack bytes; thus, if case 2 group (02 MINERALS) is moved to case 1, the last 2 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 11 lists the common characteristics and special characteristics of numeric data.

Type of Data	No. of Bytes Required	Typical Usage	Converted in Arithmetic Calculation	Boundary Alignment Required	Special Characteristics
DISPLAY (External decimal)	1 per digit	Input from cards Output to card. 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≤N≤4 4 if 5≤N≤9 8 if 10≤N≤18	Subscripting Arithmetic	Yes/No--for mixed usages No--for unmixed usage	Yes	Rounding and on size error tests are cumbersome. Always must be signed. Fields of over 8 digits require more handling.
COMPUTATIONAL-1 COMPUTATIONAL-2 (Floating-Point)	4 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 11. 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 the 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 the publication IBM System/360 Principles of Operation listed
in the preface 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 STATEMENT

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, the bytes
used 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.
     24 bytes total

Move C to D is: (The sign requires a separate move.)

      6 bytes for a simple move, and
     18 bytes for decimal alignment.
     24 bytes total

DISPLAY Fields

If data items A, B, C, and D are defined as DISPLAY fields, the bytes
used 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
       12 bytes total

MOVE DISPLAY TO COMPUTATIONAL-3

The bytes used for 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 bytes used for 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 bytes used to execute an IF statement when data is defined as DIS-
PLAY 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 bytes used 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 unpre-
dictable when the number of decimal digits is from 10 to 18.

IF A * B = C * D, ETC.

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

ARITHMETIC COMPUTATIONS

ADD COMPUTATIONAL-3 TO COMPUTATIONAL-3

The bytes used 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 deci-
mals, and 4 bytes for blanking the sign.

GENERAL TECHNIQUES FOR CODING

The following examples illustrate how COBOL data fields can be manipu-
lated. Some of the techniques illustrated are basic and can be used in
most programs, whereas 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 intermedi-
ate results may be obtained. The rules for truncation are in the publi-
cation IBM System/360 COBOL Language Specifications.

A method of avoiding unexpected intermediate results is to make crit-
ical 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 statement 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 programmer attempts to use a field without it being initialized, the contents of the field are unpredictable: thus, 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. Example 1 requires the use of the floating-point feature. Example 2 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 = 00.0 - 2

An unexpected result could be:

FLD = 02.0

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.

COMPARISONS

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

   Because compiler inserted slack bytes can contain meaningless data,
group comparisons 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 desig-
nators 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 <u>is not</u> required for low-order characters, much
more efficient coding <u>is</u> generated by pictures such as:

    ZZZ.99

OPENING FILES

The OPEN statement does not require a work area. Less storage is used
if multiple files are opened with one OPEN statement rather than speci-
fying an OPEN statement for each file. A single OPEN requires approxim-
ately 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 pro-
grammer must provide his own end card (some card other than /*) which he
can test to detect an end-of-file condition.

PARAGRAPH NAMES

Paragraph names use storage when the PERFORM verb is used in the pro-
gram. 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 iden-
tifying in-line procedures where paragraph names are not required.

<u>Example</u>:

    Avoid writing the following:

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

The recommended coding is:

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

TRAILING CHARACTERS

Pictures with a trailing period or comma require that punctuation fol-
low, otherwise 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 the REDEFINES
clause. 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:

```
   |    |    |           |
   |____|____|_____|
    S1      7 8      15
    ^   ~~~~   ~~~~
    |   Byte 1   Byte 2
    |
    A
```
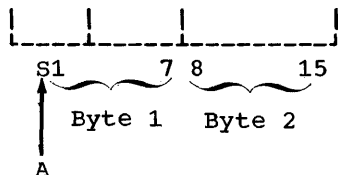
   02 FILLER REDEFINES A., states that A is to be redefined
      as follows:

   • 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 the assembler can perform
logic operations operated on logically at the assembler level, or logic-
al comparisons can be made by the COBOL program. It can be stored in a
file, and later moved back to its point in a similarly defined field.
However, using 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 the REDEFINES clause 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:

```
           IF-NUM-FIELD
         ⌒⌒⌒⌒⌒⌒⌒⌒⌒
 | 0 | 0 | 0 | 0 | 0 |
 L___L___L___L___L___J
  1   2   3   4   5    Byte Positions
 ⌣⌣⌣⌣⌣⌣⌣⌣     ⌣⌣

 REAL-FIELD     FILLER
```
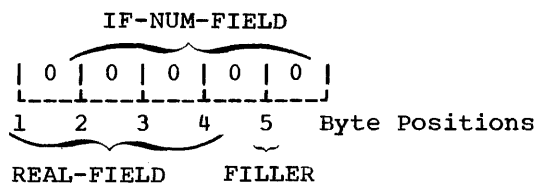
To make an IF NUMERIC test true for only unsigned fields:

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:

```
              IF-NUM-FIELD
            ⌒⌒⌒⌒⌒⌒⌒⌒⌒
Case A    | F1 | F2 | F3 | C4 | F0 |
          L____L____L____L____L____J
            1    2    3    4    5      Byte Positions
          ⌣⌣⌣⌣⌣⌣⌣⌣⌣⌣      ⌣⌣

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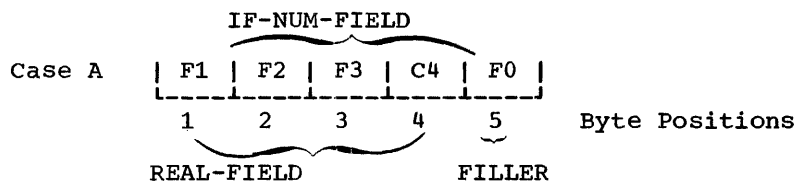
    Note that the low-order byte (rightmost byte) of IF-NUM-FIELD
    retains its initial value of 0.

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

```
              IF-NUM-FIELD
            ⌒⌒⌒⌒⌒⌒⌒⌒⌒
Case B    | F1 | F2 | F3 | F4 | F0 |
          L____L____L____L____L____J
            1    2    3    4    5      Byte Position
          ⌣⌣⌣⌣⌣⌣⌣⌣⌣⌣      ⌣⌣

          REAL-FIELD          FILLER
```
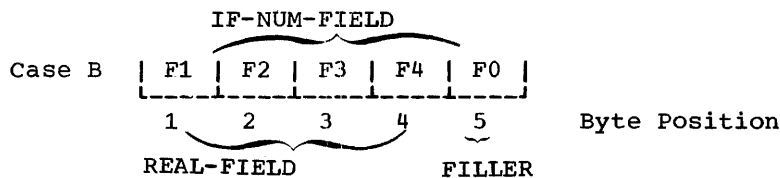
2. Test IF-NUM-FIELD FOR NUMERIC.

    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 character.

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

Unless binary (COMPUTATIONAL) or floating-point data (COMPUTATIONAL-1 or COMPUTATIONAL-2) is used, the programmer need not be concerned with slack bytes and alignment. Slack bytes are bytes inserted either by the programmer or the compiler in order to align data items.

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 message is given when slack bytes are inserted by the compiler.

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

A file 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 in file would appear, as follows:

GOLD   SILVER
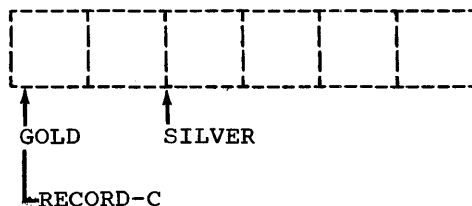
RECORD-C

If an FD were defined:

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

The compiler assumes the following structure:

```
 _____
|    |    |▓▓▓▓|▓▓▓▓|    |    |    |    |    |    |
|    |    |▓▓▓▓|▓▓▓▓|    |    |    |    |    |    |
|____|____|▓▓▓▓|▓▓▓▓|____|____|____|____|____|____|
 ↑          _____/  ↑
GOLD          SLACK   SILVER
|             BYTES
RECORD-C
```
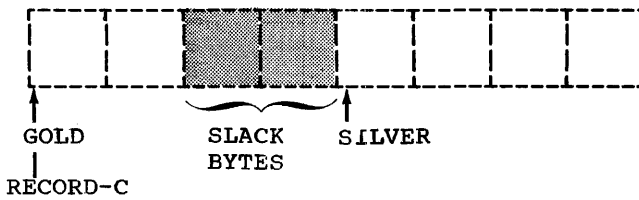
When the record on the file is read, it is placed in the area
defined, left justified. The area thus contains the following:

```
 _____
|    |    |▓▓▓▓|▓▓▓▓|    |    |    |
|    |    |▓▓▓▓|▓▓▓▓|    |    |    |
|____|____|▓▓▓▓|▓▓▓▓|____|____|____|
 ↑          SLACK  ↑
GOLD        BYTES  SILVER       (This is the compiler-
|                               generated address for
RECORD-C                        SILVER)
```
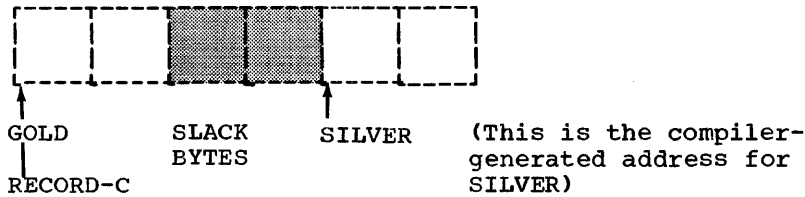
Thus, the first 2 bytes of the 02 SILVER are lost because of improper
alignment. Hence, when the 02 SILVER is accessed, only the last 2 bytes
are available.

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

        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 02 SILVER (defined as alphanumeric but which 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 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 SLIVER PICTURE XXXX.

```
 _____
|    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |
|____|____|____|____|____|____|____|
 ↑         ↑
GOLD    SILVER
|
    RECORD-C
```
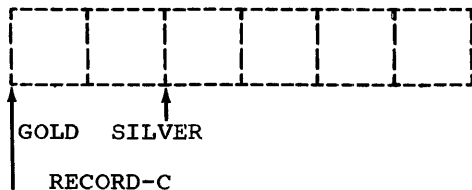
then define a record in the WORKING-STORAGE section as:

```
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:

GOLD   SILVER

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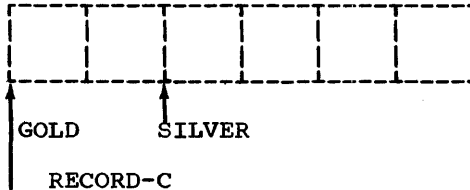
Now move the 4-byte 02 SILVER to the 02 LEAD. Because the 02 SILVER and 02 LEAD are both defined as display, the data retains its original form and are properly aligned. By accessing the REDEFINES (DIAMONDS) the binary data can be operated on as desired. The same problem could exist when reading or writing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

For a complete discussion of slack bytes, refer to the publication _IBM System/360 COBOL Language Specifications_.

VARIABLE LENGTH RECORDS

Variable length records can be specified for standard sequential files only. The OCCURS...DEPENDING ON clause describes the part of the record that is to be variable. An example of specifying a variable length record is, as follows:

```
01 VARIABLE-REC.
    05 FIXED.
        10 A PICTURE X(46).
        10 CONSTANT PICTURE 99.
    05 VARIABLE-PART OCCURS 10 TIMES DEPENDING ON CONSTANT.
        10 V-1 PICTURE X(38).
        10 V-2 PICTURE 9(10).
```

The record consists of a fixed portion and a variable portion. The variable portion must be the last part of the record. The variable portion in the example can occur a maximum of 10 times depending on CONSTANT. In this example, CONSTANT is part of the record but it need not be. However, the programmer is always responsible for initializing and updating the value of CONSTANT before referring to data items that are part of the variable portion of the record (such as V-1 and V-2 in the example). The value of CONSTANT may be 0, in which case only the fixed portion of the record exists. However, the value of CONSTANT cannot be negative.

References to the variable portion must always be subscripted. For example, V-1 (1) refers to the first occurrence of the field in the record. If the subscript is a data-name, for example V-1 (N), the programmer must be sure that N has the appropriate value. He may wish to initialize it to 1, increment it to refer to subsequent portions of the record, and check it for maximum size.

If a subscript is represented by a literal, the location of the subscripted data item is resolved at compile time. If a subscript is represented by a data name, the location is resolved at execution time for each occurrence of the data item. Thus, if a data item subscripted by a variable is to be used frequently, it is more efficient to move the data item to a work area. It is also more efficent to define subscripts as COMPUTATIONAL with pictures of not more than five integers.

BLOCKING VARIABLE LENGTH RECORDS

When blocking variable length records, the programmer must consider how much the records will vary and the size of the buffer area. A buffer is a designated area in main storage used for input/output transactions. When file processing begins, a block is placed into a buffer where the records are directly addressed. Execution of a READ or WRITE statement directs a pointer to the appropriate record in the buffer. When writing a file, the buffer is filled and then written out as a block.

When a variable length record is written, it actually contains the record itself and a 4-byte control field indicating the record length. An additional 4-byte control field containing the block size precedes each block. The following illustration shows the layout of both blocked and unblocked records.


```
UNBLOCKED      BLOCKSIZE|RECORDSIZE|RECORD  |IRG|BLOCKSIZE|RECSIZE|RECORD
               ---------+----------+-------+---+---------+-------+-------
               4-bytes  |  4-bytes |x-bytes|   |  4-bytes |4-bytes|x-bytes


BLOCKED        BLOCKSIZE|RECORDSIZE|RECORD  |RECORDSIZE|RECORD  |RECSIZE|RECORD
               ---------+----------+-------+----------+-------+-------+-------
               4-bytes  |  4-bytes |x-bytes|  4-bytes |x-bytes|4-bytes|x-bytes
```
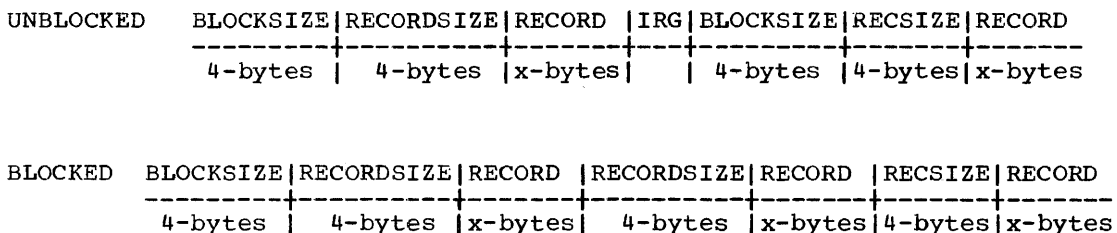

These control fields are supplied by the system and are not available to the programmer. However, they are a consideration when determining the buffer size which is specified by means of the BLOCK CONTAINS clause. If the BLOCK CONTAINS integer CHARACTERS form of the clause is used, integer must equal the size of the largest record defined for the file (RECORD CONTAINS) plus an additional 4-bytes for each record for the control field that precedes each record. (The compiler adds the 4-byte block count field. The programmer does not include this field in his count.) Note that if the file contains records with COMPUTATIONAL, COMPUTATIONAL-1, or COMPUTATIONAL-2 entries, it is the programmer's responsibility to add necessary intra-record slack bytes. These slack bytes are part of the record description and must be included in the value of the integer.

Thus, if two types of records are to be written, one of 400 characters in length and the other 200 characters in length (RECORD CONTAINS 200 TO 400 CHARACTERS), the minimum integer that can be specified is 404 (BLOCK CONTAINS 404).

However, if a record 200 characters long was placed in the block specified, there would not be enough space allocated for another record even if the next record was also 200 characters long, because the 4-byte count field preceding each variable length record could not be accommodated. Therefore, given the above facts, the programmer should at least specify BLOCK CONTAINS 408 optimum and use the APPLY WRITE-ONLY option. This option is specified to make optimum use of buffer space. When it

is specified, the length of the next record to be written is checked against the space remaining in the buffer. If the space is sufficient, the record is written. If APPLY WRITE-ONLY is not specified, the buffer is truncated and the block is written out whenever the space remaining in the buffer is not sufficent for the maximum record (400 characters in the above example) defined for the file.

The programmer can use the RECORDS option instead of the CHARACTERS option in the BLOCK CONTAINS clause to specify how many maximum size records are to fit into a block. The compiler then computes the buffer size by multiplying the length of the maximum size record by the number of records specified and adding 4 bytes for the block count field and 4 bytes for a count field for each record. This option is more efficient if the records do not vary in size considerably. However, if the following is specified:

        RECORD CONTAINS 200 TO 400 CHARACTERS
        BLOCK CONTAINS 3 RECORDS

the compiler reserves a buffer area of 1216 characters. Depending on the actual size of the records, more records could probably be contained in the buffer area. Given the above facts, it is possible for a block to contain five 200-character records (5*204+4<1216).

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 alternatives:

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 ON CLAUSE

Records are processed in the file's buffer area. The first record starts on a doubleword boundary. If there is no OCCURS...DEPENDING ON

clause, a diagnostic message 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 ON
clause in the buffer:

1. Determine the largest alignment factor with 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 align-
   ment factor.

3. For an alignment factor of eight, move the record, as a group, to
   01 in the working storage section.

MULTIVOLUME INPUT FILE WITH STANDARD LABELS, ALTERNATE TAPE DRIVE
SPECIFIED

When volume switching occurs for a multivolume file with standard
labels, the volume just completed will not rewind.

INPUT/OUTPUT ERROR PROCESSING CONSIDERATIONS

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 input/output error occurs. The fol-
lowing points should be considered when the USE AFTER STANDARD ERROR
clause 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, the program is abnormally terminated
   and a storage dump is produced.

   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 per-
   sists, 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 input/output error (by a diag-
   nostic message) 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 declara-

tive 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 entered only 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 invalid 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.

Refer to Section VIII: Processing COBOL Files on Direct Access Devices for information on error processing for other direct access organizations.

LINKAGE EDITOR

The output of a COBOL compilation is an object module. Before the pro-
gram can be executed, it must be altered to a form acceptable for execu-
tion. The linkage editor edits the object module and produces a program
phase. The structure of a program phase makes it suitable for execu-
tion. The COBOL program itself is produced as one control section.
However, there may be external references, such as entry points to sub-
routines or subprograms to be resolved. The subroutines that the COBOL
compiler calls for in the object program, e.g., for conversion from COM-
PUTATIONAL 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 12 illustrates how a subprogram is called and what data defini-
tions are required to support the CALL.

   The calling program 'CALLPROG' calls the subprogram 'PAYROLL', hand-
ing '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 sub-
ordinate 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 CALLING PROGRAM               Graphic              Card Form#  *    Identification
Programmer J. DOE            Date     Punch                              73        80

SEQUENCE
(PAGE) (SERIAL)  A        B
1   3 4  6 7 8  12   16   20   24   28   32   36   40   44   48   52   56   60   64   68   72
001 001  IDENTIFICATION DIVISION.
  . 002  PROGRAM-ID. 'CALLPROG'.
  . 003  REMARKS. EXAMPLE OF A CALLING PROGRAM.
  .  .
  .  .
  .  .
  . 008  DATA DIVISION.
  .  .
  .  .
  .  .   WORKING-STORAGE SECTION.
  . 015  01  RECORD1.
  . 016      02  JONES-J.
  . 017          03  SALARY PICTURE IS 9(5)V99.
  . 018          03  RATE PICTURE IS 9V99.
  . 019          03  HOURS PICTURE IS 99V9.
  .  .
  . 020  PROCEDURE DIVISION.
  .  .
  .  .
  .  .
  . 025      ENTER LINKAGE.
  . 026      CALL 'PAYMSTER' USING JONES-J.
001 027      ENTER COBOL.
  .  .
  .  .
  .  .
```

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

Figure 12. Example of a Calling Program (Part 1 of 3).

IBM — COBOL PROGRAM SHEET — Form No. X28-1464 Printed in U.S.A.

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

```
001001    IDENTIFICATION DIVISION.
   002    PROGRAM-ID. 'PAYROLL'.

   005    DATA DIVISION.

   008    77  SALARYX PICTURE IS 9(5)V99 VALUE IS 0000000.

   012    LINKAGE SECTION.

   015    01  PAYOFF.
   016        02    PAY PICTURE IS 9(5)V99.
   017        02    RATEX PICTURE IS 9V99.
             02    HOURS PICTURE IS 99V9.
   025    PROCEDURE DIVISION.



   040          ENTER LINKAGE.
   041          ENTRY 'PAYMSTER' USING PAYOFF.
001042          ENTER COBOL.
```

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

Figure 12. Example of a Calling Program (Part 2 of 3).

IBM — COBOL PROGRAM SHEET — Form No. X28-1464 Printed in U.S.A.

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

```
002044 COMP.  COMPUTE SALARYX = HOURS * RATEX.

   047          MOVE SALARYX TO PAY.


   050          ENTER LINKAGE.
   051          RETURN.
002052          ENTER COBOL.
```

Figure 12. Example of a Calling Program (Part 3 of 3).

When a call is issued, the address of JONES-J is passed 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 subpro-
gram, 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, then 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 conven-
tions used in System/360 for preparing and using assembler language sub-
programs with COBOL, refer to Appendix A.

RESTRICTIONS ON 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 sta-
tements 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 diagnostic error messages.

OVERLAY STRUCTURES

The following discussion illustrates the procedures available for pro-
cessing COBOL subprograms. The first technique employs the linkage edi-
tor 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.

Linkage Editing without Overlay

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 13 illustrates the storage layout for nonoverlay processing.

```
+-----------------------------------------------+
|    COBOL MAIN PROGRAM                          |
+-----------------------------------------------+
|    SUBRTNX                                     |
+-----------------------------------------------+
|    SUBPROGRAM A                                |
+-----------------------------------------------+
|    SUBPROGRAM B                                |
+-----------------------------------------------+
|    SUBPROGRAM C                                |
+-----------------------------------------------+
```
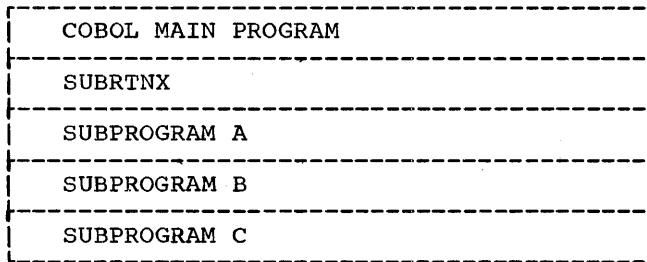
Figure 13. Storage Layout for Nonoverlay

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 14 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

/*
   ENTRY
// EXEC LNKEDT
// EXEC

   (data for program)
/*
/&
```
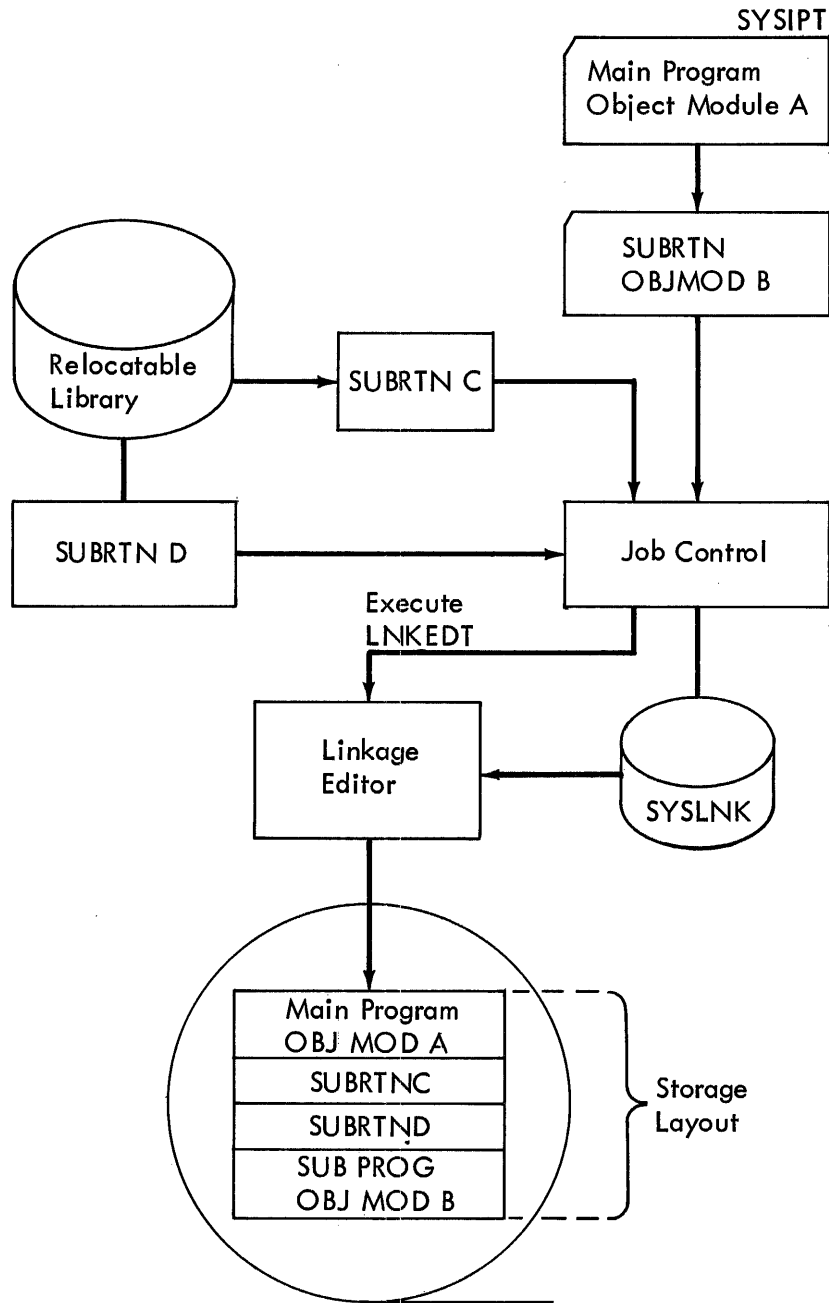
Figure 14. Example of Data Flow Logic in a Call Structure

Note: For the example given, it is assumed that SYSLNK is a standard
assignment. The flow diagram illustrates how the various program seg-
ments 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 the programmer to use
subprograms that do not fit into main storage (along with the main pro-
gram) is called overlay.

Figure 15 illustrates storage layout for overlay processing.

```
+--------------------------------------------+
|    COBOL MAIN PROGRAM                       |
+--------------------------------------------+
|    SUBRTNX                                   |
+--------------------------------------------+
|    SUBPROGRAM                                |
+--------------------------------------------+
|    A or B or C                               |
+--------------------------------------------+
```
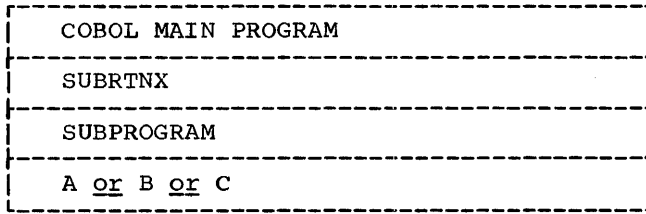
Figure 15. Storage Layout for Overlay Processing

Linkage Editing with Overlay

The linkage editor facility permits the reuse of storage locations
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 lin-
kage editor, which subprograms are to overlay each other. The subpro-
grams 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 edi-
tor 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 publication System Control and System Service listed
in the preface 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 lan-
guage 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.
       .
       .
       .
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 16 is a flow diagram of the overlay logic.

Note that 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 that is to be called by OVRLAY must precede the first executable statement in the subprogram.
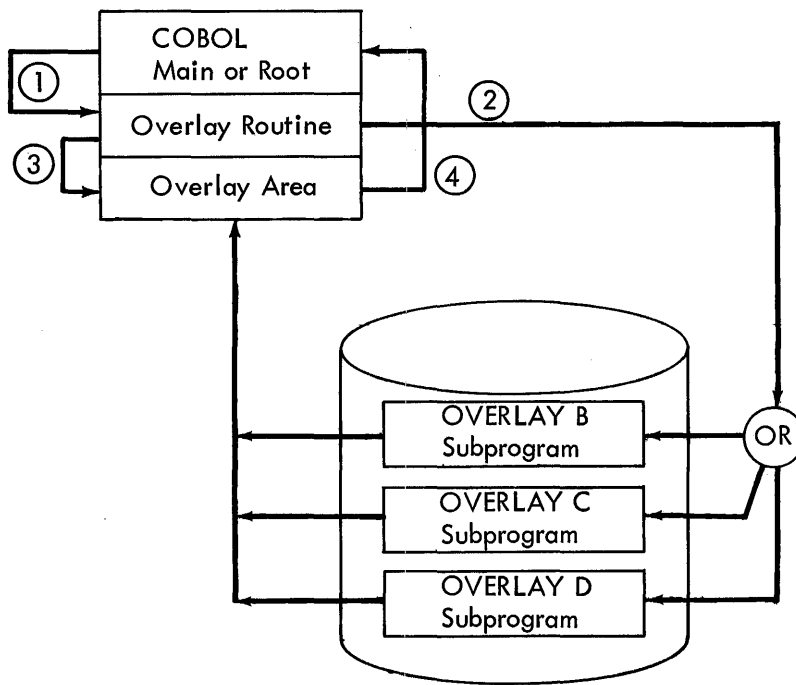
Figure 16. Flow Diagram of Overlay Logic

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, OVER-LAYC, and OVERLAYD overlay each other when called during execution.

```
// JOB OVERLAYS
// OPTION LINK
   PHASE OVERLAY,ROOT
// EXEC COBOL
   COBOL Source for
   Main Program
    'OVERLAY'
/*
// EXEC ASSEMBLY
      (source deck for OVRLAY)
/*
   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
/*
/&
```

The PHASE cards indicate the begin address of each phase. The phases OVERLAYC and OVERLAYD will have the same begin address 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:
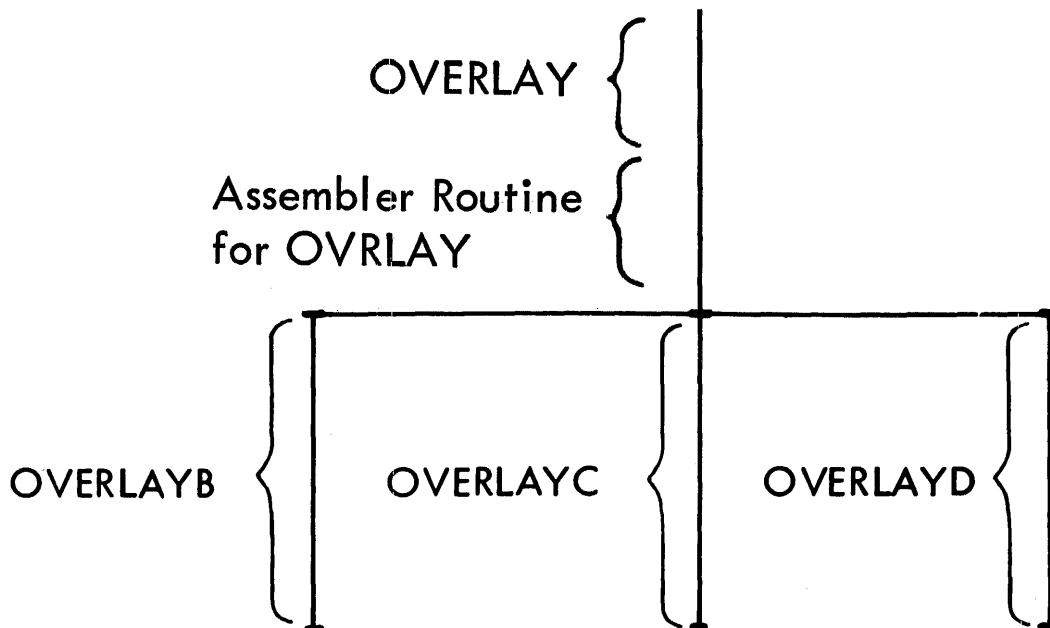
The phase name specified in the PHASE card must be the same as the value contained in the first argument for CALL 'OVRLAY', i.e., PROCESS-LABEL, COMPUTE-TAX, etc., contain OVERLAYB, OVERLAYC, respectively, which are the names given in the PHASE card. The phase name specified in the PHASE statement and the ENTRY name in the called subprogram must not be the same.

PASSING PARAMETERS TO ASSEMBLER LANGUAGE ROUTINE

A subprogram may be written in assembler language to take advantage of the system's 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 and the assembler language routine follow:

```
IDENTIFICATION DIVISION.
    PROGRAM-ID.  'COBOL'.
         .
         .
         .
    DATA DIVISION.
         .
         .
         .
    WORKING-SECTION.
    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 instruc-
tion, 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.

The data management facilities of the Disk Operating System are provided
by routines that are referred to as the input/output control system
(IOCS). A distinction is made between two types of routines:

1. Physical IOCS (PIOCS) -- Input/output routines that are included in
    the supervisor.

    Physical IOCS controls the actual transfer between the external
    medium and main storage. It performs the functions of initiating
    the execution of channel commands and handling associated input/
    output interrupts.

2. Logical (LIOCS) -- Input/output routines that are linked with a
    COBOL program.

    LIOCS performs those functions that a programmer needs to locate
    and access a logical record for processing. A logical record is
    one unit of information in a file of similar units -- for example,
    one employee's record in a master payroll file, one part-number
    record in an inventory file, or one customer account record in an
    account file. One or more logical records may be included in one
    physical record. LIOCS refers to the routines that perform the
    following functions:

    a. Blocking and deblocking records.

    b. Switching between input/output areas when two areas are speci-
        fied for a file.

    c. Handling end-of-file and end-of-volume conditions.

    d. Checking and writing labels.

An understanding of how LIOCS functions may be of help to a COBOL
programmer preparing files. Briefly, certain input/output statements
and file description entries in the Data Division of the COBOL program
such as ACCESS IS, RECORD KEY IS, etc., are used to build a unique DTF
(Define the File) table for each file. The information in this table
will be different based on what is stated in the COBOL program; or bas-
ically, it will contain the information particular to the file for pro-
cessing input/output and a series of constants that describe the charac-
teristics of a particular file, such as record size, block size, record
format, etc.

One of the constants in the DTF table names a logic module that is to
be used at execution time to process that file. A logic module is a
generalized routine that modifies itself based on the constants stored
in the DTF tables. It contains the coding necessary to perform data
management functions required by the file such as blocking and deblock-
ing, initiating label checking, etc.

Generally, these logic modules are separately assembled and cataloged
in the relocatable library under a standard name. Then, at linkage
editing time, the linkage editor searches the relocatable library using
the name in the DTF table to find the logic module. The logic module is
then included as part of the program phase. Note that the autolink fea-
ture of the linkage edit takes care of including the logic modules. The
COBOL programmer does not specify any INCLUDE statements.

The type of DTF table prepared by the compiler depends on the data organization of the file. COBOL provides two types of data organization that are used for direct-access files only.

- Indexed Sequential -- direct-access only.

- Direct -- direct-access only.

The rest of this section provides information on preparing files with indexed sequential and direct data organizations. Included for both are general descriptions of the organization, the COBOL statements that must be specified in order to build the correct DTF tables, error recovery techniques, how to modify the DTF tables, and coding examples.

INDEXED SEQUENTIAL

Files having indexed sequential organization may be processed randomly or sequentially. Indexed sequential files may be created, added to, read from, or updated.

The records of an indexed sequential file are organized on the basis of a collating sequence determined by control fields called keys. A key precedes the data portion of a record and is from 1 to 255 bytes in length.

An indexed sequential file exists in space allocated on direct-access volumes as prime areas, overflow areas, and index areas.

Prime Areas and Overflow Areas

Prime areas are areas on a cylinder allotted for data records which consist of a key and the actual data. The track format for the 2311 or 2314 Disk Storage Drive and the 2321 Data Cell Drive shown in Appendix E may be helpful in visualizing the formats of the records for an indexed sequential file.

A new record added to an indexed sequential file is placed into a location on a track determined by the value of its key field. If records were inserted in precise physical sequence, insertion would require shifting all records of the file with keys higher than that of the one inserted. However, because an overflow area exists, indexed sequential file organization allows a record to be inserted into its proper position with only the records on the track in which the insertion is made being shifted.

Overflow areas are areas on the disk reserved to accommodate additions to the file. When a record is to be inserted, the records already on the track that are to follow the new record are written back on the track after the new record. The last record on the track is written onto an overflow track. Track index entries (see the next discussion) are adjusted to indicate records in an overflow area. The COBOL programmer may choose among three options in determining where records will be placed. They are:

1. Cylinder Overflow Area Only -- When creating an indexed sequential file two 2311 tracks or four 2314 or 2321 tracks will automatically be reserved by the COBOL compiler for cylinder overflow. These tracks will accommodate overflow records that occur within the specified cylinder. To decrease or increase the number of tracks reserved for overflow, the user must modify a DTF table entry (see

"Modifying the DTF Table" and the coding examples at the end of this section).

2. Independent Overflow Area -- An additional extent for the exclusive purpose of storing overflow records can be implemented by submitting the proper job control EXTENT cards. IOCS will then create this independent overflow area. No COBOL entry is needed to implement it. If, however, the programmer wishes an independent overflow area only (that is, no cylinder overflow area), the DTF table must be modified to specify no cylinder overflow (see "Modifying the DTF Table" and the coding examples at the end of this section).

3. Both Cylinder and Independent Overflow Areas -- In this case, the cylinder overflow will be used until it becomes full. At that time, further additions to the file will result in overflow records being placed in the independent overflow area. This option again can be implemented by specifying job control EXTENT cards.

   The advantage of the cylinder overflow area is that of time. Arm motion is minimized during sequential or random retrieval and when making additions to the file.

   The advantage of the independent overflow area is that of space. The number of tracks allocated to an independent overflow area may justifiably be less than the total number of tracks allocated by the cylinder overflow option. This alternative would then minimize the total disk pack area obligated for overflow records.

Index Areas

The ability to read and write records from anywhere in a file with indexed-sequential organization is provided by indexes that are part of the file itself. There are always two types of indexes: a cylinder index for the whole file, and a track index for each cylinder. A third type of index, the master index, can be created and used if desired.

Cylinder Index: For each cylinder of data, an entry is made to the cylinder index. This entry consists of the address of the track index for that cylinder and the highest key of a record on the entire cylinder.

   The cylinder index will have its own extents that must be defined by job control EXTENT cards. These extents must be outside the limits of any data extents. (Note that at least two sets of extent information must be defined. They are a data extent and a cylinder index extent. If the file exceeds one disk pack, additional data extents will be provided and, at the user's option, an independent overflow area extent may be defined.) The cylinder extent must be on-line when the file is processed. It may be on the same pack as the data file, or it may be on a separate pack.

Track Index: A track index exists for each cylinder of the file. The track index always begins on track 0. For each track of data, an entry is made to the track index. The entry consists of two parts: (1) for the prime area, the entry is the address of the lowest record on that track and the highest key of a record on that track; (2) for the overflow area, the entry consists of the highest key associated with that track and the address of the lowest record in the overflow area. If no overflow entry has been made, the address is X'FF'.

Master Index: If a file occupies many cylinders, a search of the cylinder index for a key is inefficient. Thus, the user can create a master index (by specification on the EXTENT card) that indexes the cylinder index. An entry for the master index consists of the address

of the cylinder index track and the highest key of a data record on that
cylinder index track. One entry is made to the master index for each
track of data on the cylinder index. It is advantageous to construct a
master index if the cylinder index occupies four or more tracks.

Figure 17 shows an indexed sequential file with two levels of index-
ing and no overflow records. The entries for track 1 of the cylinder
are shown. The address of the first record on the track is track 1,
record-0 at the beginning of track 1. The highest key on track 1 is 8.
The address of the lowest record in the overflow area is FF because
there are no overflow records. The entry to the cylinder index for
track 1 shows the address of the track index and the highest key on the
cylinder which is 32. The COCR (cylinder overflow control record),
which is maintained in the data portion of record zero, indicates that
no records are in the cylinder overflow area after loading. (Note that
if the track index does not occupy all of track 0, track 0 will also
contain data records).
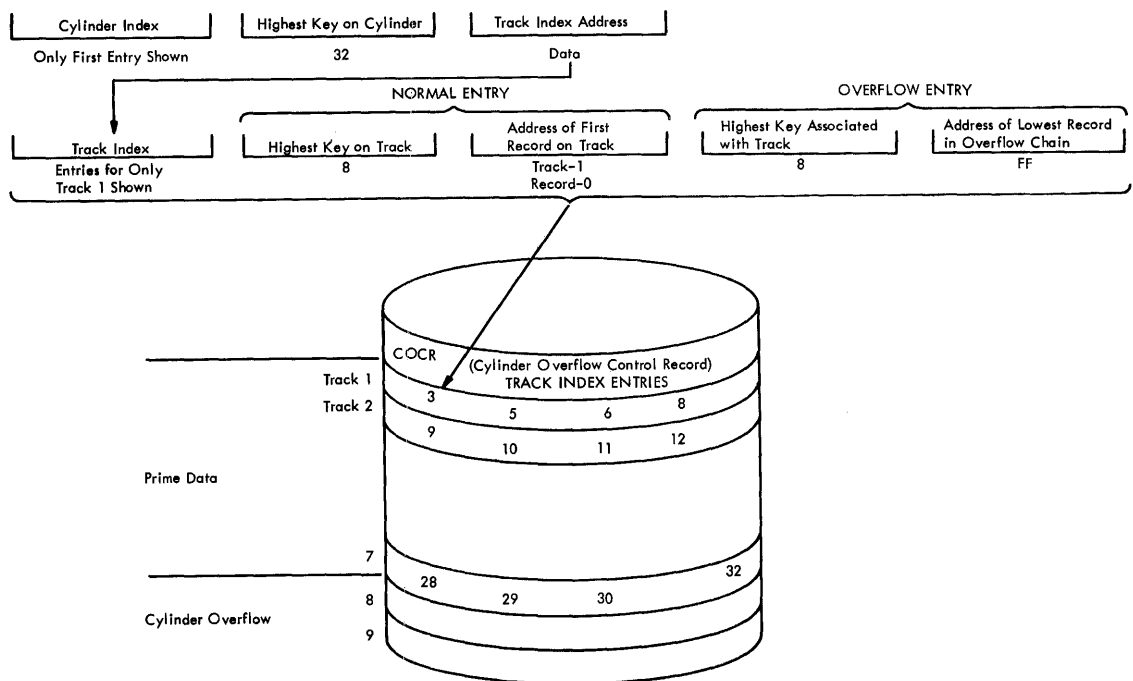
• Figure 17.   Indexed Sequential File Without Overflow

Figure 18 shows what happens when another record, record 7, is added,
and forces record 8 into an overflow area. In this case, the key of the
normal entry is changed to 7 and the address of the overflow entry indi-
cates the location of record 8, which is on track 8 record 1. No other
changes to the indexes are required. The COCR will be updated to indi-
cate that the last record in the cylinder overflow area is on track 8,
record 1.

Cylinder Index	32	Track Index Address
Points to Track Index	Key	Data

NORMAL ENTRY   OVERFLOW ENTRY

Track Index	⑦	Track-1 Record-0	8	Track-8 Record-1
Points to Data Tracks	Key	Data	Key	Data

Prime Data:
COCR TRACK INDEX ENTRIES
3 5 6 7
9 10 11 12
28 29 30 32
8

Track 1, Track 2 ... 7, 8, 9

Cylinder Overflow
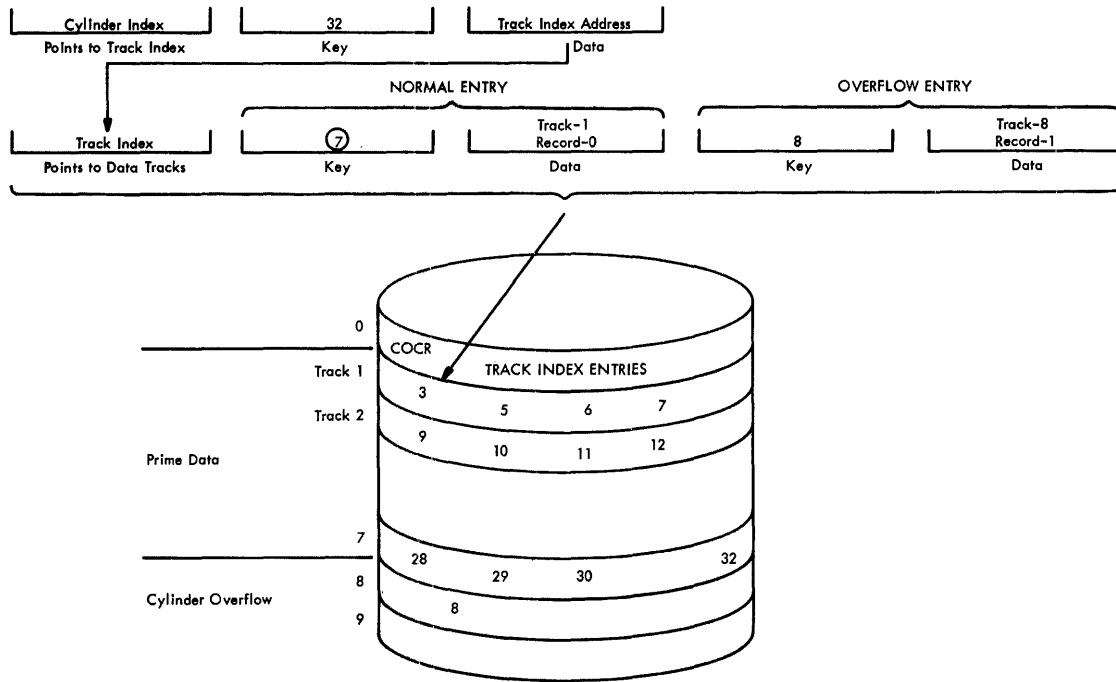
● Figure 18. Indexed Sequential File With Overflow

The indexed sequential organization enables the programmer to:

1. Read or write (in a manner similar to that for sequential organization) logical records whose keys are in ascending collating sequence.

2. Read or write individual logical records randomly.

3. Add logical records with new keys. The system locates the proper position in the file for the new record and makes all necessary adjustments to the indexes.

The important advantages are the ability to retrieve records randomly as well as sequentially, and to make additions to the file without having to sort and merge the additions while copying the entire file.

These advantages are possible because overflow areas are available to provide the additional space required when additions are made to the file, and indexes which are built and maintained by LIOCS.

COBOL STATEMENTS USED TO SPECIFY AN INDEXED SEQUENTIAL FILE

The following is a list of the COBOL statements required for creating, adding, or updating an indexed sequential file. In addition, the list is followed by a brief explanation of the functions IOCS performs as a result of executing these statements.

Creating an 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 statement 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, the OPEN statement establishes the area that is to be written on the disk as specified in the EXTENT statement by the LOWER and UPPER parameters. Finally, the OPEN statement 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 statement enters the record keys specified by the programmer into the track and cylinder index tables and writes the actual data on the portion of the track defined by the EXTENT parameters. The records are placed on the track sequentially in the prime data area.

If the programmer specifies INVALID KEY, control is given to the invalid key routine whenever a duplicate record or a record out of sequence is detected. The programmer is responsible for writing the invalid key routine (see the coding examples).

CLOSE Statement

The CLOSE statement 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 statement 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 an indexed sequential file, the programmer can control the RECORD KEY with certain restrictions:

• The RECORD KEY must be provided before execution of the WRITE statement. 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 (DLBL, EXTENT) used to create the file are required, with the following exception: the parameter ISE should be used for the 'type' code in the DLBL 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 EXTENT statement. If it does not fit, the file must be recreated.

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

Sequential Retrieval of an Indexed Sequential File

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: The OPEN statement 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 EXTENT 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 Sequentially

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) used for creating the file.

- OPEN I-O file-name

- READ file-name AT END

- REWRITE record-name [INVALID KEY]

- CLOSE file-name

    The OPEN and CLOSE statements function in the same manner for updating as they do for retrieving an indexed sequential file. The READ statement also functions in the same manner (as for sequential retrieval) but must be used in conjunction with the REWRITE statement, as follows.

REWRITE Statement: The REWRITE statement 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 statement 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 KEY and RECORD KEY is permitted. Thus, the SYMBOLIC KEY can be set before the OPEN statement is executed, allowing processing to begin with any record within the file. Once the OPEN statement is completed, the SYMBOLIC KEY is not needed. The RECORD KEY, which must not be modified when updating a file, can be referred to 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

- OPEN INPUT file-name

- READ file-name INVALID KEY

- CLOSE file-name

The OPEN and CLOSE statements function in the same manner for updating as they do for retrieving 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 in the same manner as for sequential retrieval of an indexed sequential file. The READ statement retrieves the record identified by the SYMBOLIC KEY. This key must be specified for every READ statement and must be within the limits of the file; 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 KEY and RECORD KEY 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 as are
needed to retrieve the file are required and, in addition, the programm-
er must specify:

- OPEN I-O file-name

- WRITE record-name [INVALID KEY]

- CLOSE file-name

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

A duplicate key error results when a record that is being added has
the same RECORD KEY value as a record already in the file. This condi-
tion causes control to be given to the programmer's INVALID KEY routine.

Key Handling: The programmer must initialize the SYMBOLIC KEY with a
key value prior to every READ statement. The value must be equal to the
record key within the record to be retrieved. This key must be within
the file limits; otherwise, a 'NO RECORD FOUND' error condition results.
The RECORD KEY can be used only for record reference during the retrieve
and update functions. When adding to the file, this key must be
initialized before each write.

ERROR RECOVERY TECHNIQUES FOR INDEXED SEQUENTIAL FILES

Recovery from input/output error conditions raised for an indexed
sequential file may be attempted in three ways. They are:

1. The INVALID KEY clause in the COBOL program.

2. A COBOL library subroutine (IHD03500).

3. The USE AFTER STANDARD ERROR clause in the COBOL program.

The relationship among these three types of recovery attempts is
shown in the flowchart that follows. If an INVALID KEY error is raised,
the INVALID KEY routine is executed. If it is not, the COBOL subroutine
will analyze the other error indicators. If no USE AFTER STANDARD ERROR
is specified, the job is terminated after printing a message that iden-
tifies the type of error. If a user Declarative Section does exist,
control is given to that routine. The COBOL programmer can prepare such
a routine to make a further analysis of the DTF table to determine
further information about the error status. Note that program execution
can continue only if an INVALID KEY or USE AFTER STANDARD ERROR clause
exists.

Table 1 summarizes the error conditions by function and indicates
which errors will raise an INVALID KEY, which will be analyzed by the
subroutine, and which errors could be profitably analyzed by a USE AFTER
STANDARD ERROR routine (see Supervisor and Input/Output Macro Instruc-
tions for complete list of error conditions).

Does error raise INVALID KEY? → INVALID KEY ? — YES → Execute Statements following INVALID KEY

NO ↓

Execute COBOL subroutine IHD03500

↓

Use After Standard Error ? — Does a declarative section exist in the program?

YES → Branch to User Error Routine

NO → Print Error Message on SYSLST or SYS000
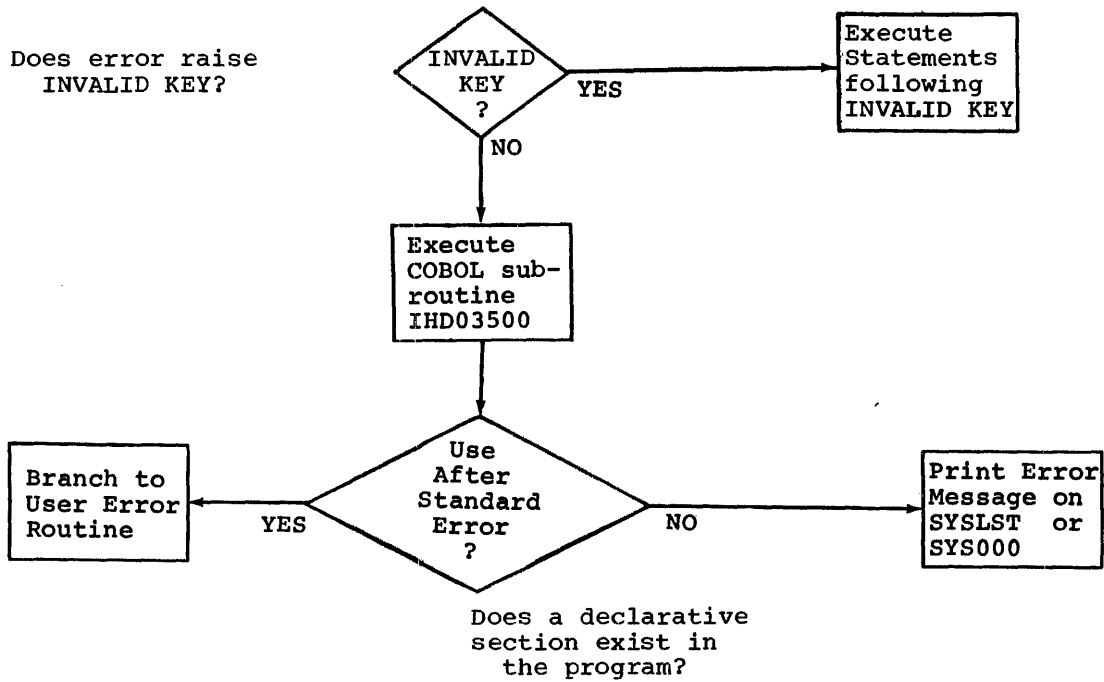
ABORT

Table 1. Error Functions

Function / Error	Create	Add	Random Retrieval	Sequential Retrieval
Duplicate Record	INVALID KEY	INVALID KEY		
Sequence Check	INVALID KEY			
DASD error	USE AFTER STANDARD ERROR			
Wrong length Record	USE AFTER STANDARD ERROR			
Prime data Area Full	USE AFTER STANDARD ERROR			
Cylinder Index Full	USE AFTER STANDARD ERROR			
Master Index Full	USE AFTER STANDARD ERROR			
No Record Found			INVALID KEY	USE AFTER STANDARD ERROR
Overflow Area Full / Area Full		USE AFTER STANDARD ERROR		

INVALID KEY Errors

The invalid key is raised when:

1. A duplicate record exists (duplicate key) (WRITE).

2. When building a file, a record is out of sequence (sequence check) (WRITE).

3. No record is found during retrieval (READ).

These are the conditions from which the programmer can most logically recover. In almost all other cases, the only choice is to abort the job; therefore, most programmers should be able to handle almost all of their error recovery procedures within the INVALID KEY routines.

When creating the file, the program will execute the imperative statements following the INVALID KEY clause for either a duplicate record or sequence check. In either case the record is not added to the file. The INVALID KEY routine could identify the error record, and either close the files, or continue with the next transaction.

When adding to the file, the INVALID KEY is raised for a duplicate record. In this case, the record will not be added. The user could identify the error record and continue with the next transaction in his INVALID KEY routine.

For random retrieval, the INVALID KEY is raised if the record cannot be found on the file. The INVALID KEY routine could identify the missing record and continue with the next transaction.

During sequential retrieval no INVALID KEY IS raised. The user must specify the AT END option to branch to end-of-job steps when the end of file is encountered.

USE AFTER STANDARD ERROR Routines

When creating a file, prime data area full would be indicated if the user did not reserve a large enough extent for all data records. In this case, the user could close the file in his USE AFTER STANDARD ERROR routine. He could then submit new job control EXTENT cards reserving more room for prime data, and extend the file with the remaining data cards by resubmitting the same program.

DASD errors and wrong-length records could be bypassed, if feasible, in the USE AFTER STANDARD ERROR routine. However, in most cases, the job must be terminated.

When adding to the file, if the cylinder overflow area cannot contain the overflow record, an Overflow Area Full error will be indicated. The record will not be added. The user could continue with the next transaction in his STANDARD ERROR routine by going to a routine to read the next input transaction, since it may be possible to add records that will be located on a different cylinder. However, the best procedure would be to close the file and stop the run.

Note that a programmer may elect to start sequential retrieval at a specified key rather than at the beginning of the file. If the key specified is not in the file, and if there is no error declarative section, the COBOL subroutine will print a message indicating this and abort the job.

INVALID KEY Errors

The invalid key is raised when:

1. A duplicate record exists (duplicate key) (WRITE).

2. When building a file, a record is out of sequence (sequence check) (WRITE).

3. No record is found during retrieval (READ).

These are the conditions from which the programmer can most logically recover. In almost all other cases, the only choice is to abort the job; therefore, most programmers should be able to handle almost all of their error recovery procedures within the INVALID KEY routines.

When creating the file, the program will execute the imperative statements following the INVALID KEY clause for either a duplicate record or sequence check. In either case the record is not added to the file. The INVALID KEY routine could identify the error record, and either close the files, or continue with the next transaction.

When adding to the file, the INVALID KEY is raised for a duplicate record. In this case, the record will not be added. The user could identify the error record and continue with the next transaction in his INVALID KEY routine.

For random retrieval, the INVALID KEY is raised if the record cannot be found on the file. The INVALID KEY routine could identify the missing record and continue with the next transaction.

During sequential retrieval no INVALID KEY IS raised. The user must specify the AT END option to branch to end-of-job steps when the end of file is encountered.

USE AFTER STANDARD ERROR Routines

When creating a file, prime data area full would be indicated if the user did not reserve a large enough extent for all data records. In this case, the user could close the file in his USE AFTER STANDARD ERROR routine. He could then submit new job control EXTENT cards reserving more room for prime data, and extend the file with the remaining data cards by resubmitting the same program.

DASD errors and wrong-length records could be bypassed, if feasible, in the USE AFTER STANDARD ERROR routine. However, in most cases, the job must be terminated.

When adding to the file, if the cylinder overflow area cannot contain the overflow record, an Overflow Area Full error will be indicated. The record will not be added. The user could continue with the next transaction in his STANDARD ERROR routine by going to a routine to read the next input transaction, since it may be possible to add records that will be located on a different cylinder. However, the best procedure would be to close the file and stop the run.

Note that a programmer may elect to start sequential retrieval at a specified key rather than at the beginning of the file. If the key specified is not in the file, and if there is no error declarative section, the COBOL subroutine will print a message indicating this and abort the job.

In general, a declarative section may be included in a program in order to find out more information about an error condition that occurs. Such a routine can interrogate byte 30 of the DTF table to determine the type of error that exists. (For more information and a coding example, see "Modifying the DTF Table.") The error indicator can be passed to a subprogram by specifying the CALL statement including USING file-name. The programmer is responsible for writing the error checking subprogram. The error indicators are shown in Table 2. Normal return from the declarative is to the next sequential instruction in the program following the input/output operation. Return by means of a GO TO statement may be made to any location. Note that the programmer must realize that the record of the last input/output operation was not located.

Table 2. Error Indicators

BYTE 30	Add, Retrieve, or Both	Create
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	

The following example illustrates a declarative routine when combining random retrieval and additions. Assuming that INVALID KEY clauses were written, control will be given to the declarative section in this example only when DASD, wrong-length record, or Overflow Area Full error indicators are raised. The purpose of the routine is to continue processing in the case of an Overflow Area Full indicator and to close the file and end the job for DASD and wrong-length record indicators.

MAIN PROGRAM

```
    WORKING-STORAGE SECTION.
    01  INDICATOR PICTURE 9.
            .
            .
            .
    PROCEDURE DIVISION.
            DECLARATIVES.
            ERR-TEST SECTION.
            USE AFTER STANDARD ERROR PROCEDURE ON IS-FILE.
            ENTER LINKAGE.
            CALL 'ERRSUB' USING IS-FILE, INDICATOR.
            ENTER COBOL.

            IF INDICATOR IS EQUAL TO 1 GO TO READ-ROUTINE.
            GO TO END-JOB.
            END DECLARATIVES.
```

SUBPROGRAM

```
    WORKING-STORAGE SECTION.
    77 FULL PICTURE 9 USAGE COMPUTATIONAL VALUE 2.

    LINKAGE SECTION.
        01  DTF-FOR-DISK.
            02 BYTE PICTURE 9 OCCURS 31 TIMES.
        01  INDICATOR PICTURE 9.
```

```
PROCEDURE DIVISION.
    ENTER LINKAGE.
    ENTRY 'ERRSUB' USING DTF-FOR-DISK, INDICATOR.
    ENTER COBOL.
    MOVE LOW-VALUE TO INDICATOR.
    IF BYTE (31) IS NOT EQUAL TO FULL GO TO EXT.
    MOVE 1 TO INDICATOR.
    EXT.
    ENTER LINKAGE.
    RETURN.
    ENTER COBOL.
```

When a DASD error, wrong-length record, or Overflow Area Full error occurs on the IS-FILE, control is given to the first instruction after the USE AFTER STANDARD ERROR clause. The subprogram is called. The address of the IS-FILE DTF table and the working storage indicator are passed to the subprogram.

The subprogram sets the indicator in working storage to zeros. If byte 30 contains an X'02' indicating an Overflow Area Full error, a 1 is moved to the working storage indicator. Return is made to the main program. The main program checks to see if the indicator is a 1. If it is, a branch is made to read the next transaction. If the indicator is not a 1, a DASD error or wrong-length record error has occurred, and the program branches to the end-of-job routine to close the files and stop the run.

Note that the COBOL subroutine will also print out an error message that provides another source of information about the type of error that has occurred. (See "Modifying the DTF Table" for more information about using COBOL and assembler language subprograms to modify the DTF table.)

MODIFYING THE DTF TABLE FOR INDEXED SEQUENTIAL ORGANIZATION FILES

The DOS COBOL compiler builds a skeleton DTF table. At execution time when the file is opened, a transient routine is called to construct the complete table. The programmer may wish to change the contents of the skeleton DTF table prior to execution of the OPEN statement in order to change the overflow tracks, suppress the verify option, etc. The contents of the skeleton DTF table are shown in Table 3.

The COBOL programmer can change the contents of a certain byte in the DTF table in one of two ways, as follow:

1. By writing statements in the main COBOL program to refer to the DTF table and by writing a COBOL subprogram to change the DTF table.

2. By writing statements in the COBOL main program to refer to the DTF table and by writing an assembler language subprogram to change the DTF table.

Table 3. Contents of Skeleton DTF Table

```
COBOL TRANSIENT PREPHASE TO OPEN.  THIS ROUTINE WILL
      BUILD THE DTFIS TABLES AT OPEN-TIME


INPUT TO THIS ROUTINE IS AS FOLLOWS
SPACE MUST BE RESERVED FOR THE BIGGEST POSSIBLE TABLE
THIS IS DEPENDING ONLY ON THE KEYSIZE, THE TOTAL NUMBER OF
BYTES USED BY THE TABLE IS 600&KEYSIZE.  THE CONTENTS OF
THIS TABLE IS BEFORE THIS ROUTINE


BYTE 0      Xa00a#RANDOM              XaF0a#SEQUENTIAL
BYTE 1      Xa00a#2311 and 2314      XaF0a#2321        (PRIME DATA)
BYTE 2      Xa00a#UNBLOCKED          XaF0a#BLOCKED
BYTE 3      Xa00a#NO VERIFY          XaF0a#VERIFY
BYTE 4-5    RECORD SIZE
BYTE 6-7    NUMBER OF RECORDS IN BLOCK
BYTE 8-9    KEYLOCATION %FIRST BYTE IN RECORD 31
BYTE 10-11  RECORD KEY-LENGTH
BYTE 12-15  ADDRESS OF IOAREAL
BYTE 16-19  ADDRESS OF WORKL
BYTE 20     Xa00a#INDEX ON 2321 or 2314 XaF0a#INDEX ON 2311
BYTE 21     NUMBER OF OVERFLOW-TRACKS IN CYLINDER
BYTE 22-29  FILENAME
BYTE 30-31  DUMMY
BYTE 32-35  ADDRESS OF KEYARG (SYMBOLIC KEY)


THE REST OF THE TABLE MUST BE SET TO Xa00a
NOTICE THAT BYTES 3,20,21 MAY BE CHANGED BY THE USER
BEFORE THE OPEN STATEMENT IS EXECUTED
```

The following are examples of the options in the DTF table that can be changed. (Note that the byte count of a DTF table begins with zero; thus, for example, byte 22 actually is the 23rd byte of the table.)

1. Byte 3 (the 4th byte) of the DTF table controls the verify option. Changing the value of the 4th byte to X'00' suppresses this option. Verification consists of IOCS checking to be sure that the record written out is correct. Suppression of the verify option can result in improved program performance.

2. The presence of a master index can be indicated by changing the value of byte 20 (the 21st byte) to X'F2' for the 2311, X'82' for the 2314, and X'02' for the 2321.

3. Byte 21 (the 22nd byte) of the DTF can be altered to change the number of overflow tracks assumed per cylinder. The number of overflow tracks assumed by the compiler is 2 for the 2311 or 4 for the 2321 or the 2314. The programmer may change this to a maximum of 8 for the 2311, 16 for the 2314, or 18 for the 2321. If the value is changed to LOW VALUE, cylinder overflow is suppressed.

4. By replacing the contents of byte 20 (the 21st byte) with X'F0' the user can locate his indexes on the 2311 or the 2314 when the prime data is on the 2321.

Example of COBOL Main Program and COBOL Subprogram Modifying DTF
_____

In the COBOL main program, define and call the subprogram, as follows:

1. Data Division

                    FD ISFILE

2. Procedure Division

                        ENTER LINKAGE.
                        CALL 'CHGDTF' USING ISFILE.
                        ENTER COBOL.
                        OPEN OUTPUT ISFILE.

When the call is executed, the address of the skeleton DTF is passed
to the subprogram. Note that COBOL permits only data names to be passed
to a COBOL subprogram. In this case, a filename is passed but only to
use the address of the DTF for the file. No input/output operations can
be performed on the file in the subprogram.

The subprogram could move an appropriate value into one or more bytes
that the programmer wishes to modify. For example, the following sub-
program will eliminate the verify option.

        DATA DIVISION.
        LINKAGE SECTION.
        01 DTF-FOR-DISK.   DUMMY DTF TABLE
            02 BYTE PICTURE X OCCURS 35 TIMES.

        PROCEDURE DIVISION.
            ENTER LINKAGE.
            ENTRY 'CHGDTF' USING DTF-FOR-DISK.
            ENTER COBOL
            MOVE LOW-VALUE TO BYTE (4).
            ENTER LINKAGE.
            RETURN.
            ENTER COBOL.

The LINKAGE SECTION in the subprogram defines a DTF table. Control
is passed to byte 3 to eliminate the verify option and then control is
returned to the main program at the OPEN statement.

Note that an assembler language subprogram can be called by using the
same coding in the main COBOL program.

CODING EXAMPLES USING INDEXED SEQUENTIAL FILES

The following examples illustrate how a COBOL program can be prepared in
order to create and retrieve an indexed sequential file. The job con-
trol cards specifying label information for these examples follow. They
define a cylinder index located on cylinder 196, and a small prime data
extent located on cylinders 197 and 198.

// ASSGN SYS004,X'19'

// DLBL SYS004,'INDEXED SEQ FILE,' 67365,ISC

// EXTENT SYS004,111111,4,1,00010,00196

// EXTENT SYS004,111111,1,2,00020,00197

// EXEC

Creating an Indexed Sequential File

This program example shows how to create an indexed sequential file from the card reader. The file consists of records of 100 characters, 5 records in a block.

When the file is opened, the labels are checked, the track indexes are formed, and the extents are reserved for the cylinder index.

After the files are opened, cards are read and data is moved to the print file and disk file. Movement of the KEY-ID in the cards to the REC-ID of disk is mandatory since the REC-ID is the data name specified by the RECORD KEY clause.

A branch to the INVALID KEY routine will occur on sequence errors or duplicate records. This routine identifies the error type and error record, and then gets the next transaction.

The COBOL error subroutine identifies all other errors and aborts the job. If the INVALID KEY routine were not present, it would also identify and abort the job for sequence errors. It would identify the error and continue with the next sequential instruction for duplicate records.

At end-of-job, the files are closed and the job terminated.

```
01 001001 IDENTIFICATION DIVISION.
02      1 PROGRAM-ID.  'LOADIS'.
03      3 AUTHOR.
04      4 INSTALLATION.
05      5 DATE WRITTEN.
06      6 REMARKS.   ILLUSTRATE CREATING OF INDEXED SEQUENTIAL FILE.
07      8 ENVIRONMENT DIVISION.
08      9 CONFIGURATION SECTION.
09     10 SOURCE-COMPUTER.  IBM-360.
10     11 OBJECT-COMPUTER.  IBM-360.
11 002001 INPUT-OUTPUT SECTION.
12      2 FILE-CONTROL.
13      3     SELECT IS-FILE ASSIGN TO 'SYS004' DIRECT-ACCESS 2311
14      4        ACCESS IS SEQUENTIAL ORGANIZATION IS INDEXED
15      5        RESERVE NO ALTERNATE AREA
16      7        RECORD KEY  IS REC-ID.
17     10     SELECT CARD-FILE ASSIGN TO 'SYS005' UNIT-RECORD 2540R
18     11        RESERVE NO ALTERNATE AREA.
19     12     SELECT PRINT-FILE ASSIGN TO 'SYS006' UNIT-RECORD 1403
20     13        RESERVE NO ALTERNATE AREA.
21 003001 DATA DIVISION.
22    002 FILE SECTION.
23      4 FD IS-FILE      DATA RECORD IS DISK
24      5                 RECORDING MODE IS F
25      6                 LABEL RECORDS ARE STANDARD
26      7                 BLOCK CONTAINS 5 RECORDS.
27     10 01 DISK.
28     11    02  DISK-FLD1    PICTURE   X(10).
29     12    02  REC-ID       PICTURE   X(10).
30     13    02  DISK-NAME    PICTURE   X(20).
31     14    02  DISK-BAL     PICTURE   99999V99.
32     15    02  FILLER       PICTURE   X(53).
33 004001 FD CARD-FILE    DATA RECORD IS CARDS
34      2                 RECORDING MODE IS F
35      3                 LABEL RECORDS ARE OMITTED.
36      5 01 CARDS.
37      6    02  KEY-ID       PICTURE   X(10).
38      7    02  CD-NAME      PICTURE   X(20).
39      8    02  CD-BAL       PICTURE   99999V99.
40 005001 FD PRINT-FILE   DATA RECORD IS PRINTER
41      2                 RECORDING MODE IS F
```

```
42       3              LABEL RECORDS ARE OMITTED.
43       6  01 PRINTER.
44       7      02   PRINT-ID     PICTURE   X(10).
45       8      02   FILLER       PICTURE   X(10).
46       9      02   PRINT-NAME   PICTURE   X(20).
47      10      02   FILLER       PICTURE   X(10).
48              02   PRINT-BAL    PICTURE   ZZZ,ZZZ.99-.
49  006001  PROCEDURE DIVISION.
50       3  START.
51       4      OPEN INPUT CARD-FILE PRINT-FILE OUTPUT IS-FILE.
52       6  RD.
53       7      READ CARD-FILE AT END GO TO END-JOB.
54       9      MOVE KEY-ID TO PRINT-ID   REC-ID.
55      10      MOVE CD-NAME TO PRINT-NAME   DISK-NAME.
56      11      MOVE CD-BAL TO PRINT-BAL   DISK-BAL.
57              WRITE DISK INVALID KEY GO TO ERR.
58      13      WRITE PRINTER.
59      17      GO TO RD.
60          ERR.
61              DISPLAY 'DUPLICATE OR SEQ-ERR' UPON CONSOLE.
62              DISPLAY KEY-ID UPON CONSOLE.
63              GO TO RD,
64      19  END-JOB.
65      20      CLOSE CARD-FILE PRINT-FILE IS-FILE.
66      21      DISPLAY 'END JOB' UPON CONSOLE.
67      22      STOP RUN.
```

Random Retrieval

This program illustrates random retrieval and updating for the IS-FILE created in the previous example.

The Data Division is basically the same as in loading the file except for the ACCESS IS RANDOM clause. Both the SYMBOLIC KEY and the RECORD KEY clause are required.

The IS-FILE is opened as input. This allows both retrieval and updating. The OPEN verb does label checking, and it also stores statistic fields from the format 2 label into the DTF table.

The SYMBOLIC KEY clause data name (KEY-ID) is defined within the card read area. Reading the IS-FILE causes a search of the indexes for a record matching the KEY-ID. If the record is found, data is moved to the print-file and printed, the IS-FILE is updated, and a branch is made to the next transaction.

If a matching record is not found, the COBOL subroutine turns control to the INVALID KEY routine (No-Record). This routine identifies the error and branches to the next transaction.

If for some reason the record retrieved cannot be located again when the REWRITE verb is issued, control will be passed to ERR-ROUTINE. This condition should never occur.

The end of job routine closes the files, and terminates the job. The CLOSE verb also returns the updated statistics in the DTF table back to the format 2 label. Additions could have been made to the file in the same program if the IS-FILE were opened as I-O.

```
01         IDENTIFICATION DIVISION.
02 001002 PROGRAM-ID.  'RANDOMIS'.
03      3 AUTHOR.
04      4 INSTALLATION.  360 PROGRAMMING CENTER.
05      5 DATE WRITTEN.
06 001006 REMARKS.  ILLUSTRATE RANDOM RETRIEVAL FROM  IS-FILE.
07      8 ENVIRONMENT DIVISION.
08      9 CONFIGURATION SECTION.
09     10 SOURCE-COMPUTER.  IBM-360.
10     11 OBJECT-COMPUTER.  IBM-360.
11 002001 INPUT-OUTPUT SECTION.
12      2 FILE-CONTROL.
13      3     SELECT IS-FILE ASSIGN TO 'SYS004' DIRECT-ACCESS 2311
14 002004     ACCESS IS RANDOM ORGANIZATION IS INDEXED
15      5     RESERVE NO ALTERNATE AREA
16      6     SYMBOLIC KEY IS KEY-ID
17      7     RECORD KEY   IS REC-ID.
18     10     SELECT CARD-FILE ASSIGN TO 'SYS005' UNIT-RECORD 2540R
19     11     RESERVE NO ALTERNATE AREA.
20     12     SELECT PRINT-FILE ASSIGN TO 'SYS006' UNIT-RECORD 1403
21     13     RESERVE NO ALTERNATE AREA.
22 003001 DATA DIVISION.
23    002 FILE SECTION.
24      4 FD IS-FILE      DATA RECORD IS DISK
25      5                 RECORDING MODE IS F
26      6                 LABEL RECORDS ARE STANDARD
27      7                 BLOCK CONTAINS 5 RECORDS.
28     10 01 DISK.
29     11    02  DISK-FLD1   PICTURE  X(10).
30     12    02  REC-ID      PICTURE  X(10).
31     13    02  DISK-NAME   PICTURE  X(20).
32     14    02  DISK-BAL    PICTURE  99999V99.
33     15    02  FILLER      PICTURE  X(53).
34 004001 FD CARD-FILE   DATA RECORD IS CARDS
```

```
35      2                   RECORDING MODE IS F
36      3                   LABEL RECORDS ARE OMITTED.
37      5 01 CARDS.
38      6      02   KEY-ID      PICTURE   X(10).
39      7      02   CD-NAME     PICTURE   X(20).
40      8      02   CD-AMT      PICTURE   99999V99.
41 004009      02   CD-CODE     PICTURE   X.
42         FD PRINT-FILE  DATA RECORD IS PRINTER
43                        RECORDING MODE IS F
44                        LABEL RECORDS ARE OMITTED.
45 005006 01 PRINTER.
46 005007      02   PRINT-ID     PICTURE   X(10).
47 005008      02   FILLER       PICTURE   X(10).
48 005009      02   PRINT-NAME   PICTURE   X(20).
49 002010      02   FILLER       PICTURE   X(10).
50 005011      02   PRINT-BAL    PICTURE   $ZZZ,999.99-.
51 005012      02   FILLER       PICTURE   X(10).
52 005013      02   PRINT-AMT    PICTURE   $ZZZ,ZZZ.99-.
53 005014      02   FILLER       PICTURE   X(10).
54 005015      02   PRINT-NEW-BAL PCITURE   $ZZZ,ZZZ.99-.
55 006001 PROCEDURE DIVISION.
56 006003 START.
57 006004      OPEN INPUT   CARD-FILE IS-FILE
58 006005           OUTPUT  PRINT-FILE.
59 006006 RD.
60 006007      READ CARD-FILE AT END GO TO END-JOB.
61 006009      READ IS-FILE   INVALID KEY GO TO NO-RECORD.
62 006010      MOVE REC-ID     TO PRINT-ID.
63 006011      MOVEDISK-NAME    TO PRINT-NAME.
64 006012      MOVEDISK-BAL     TO PRINT-BAL.
65 006013      MOVECD-AMT       TO PRINT-AMT.
66 006014      ADD CD-AMT       TO DISK-BAL.
67 006015      MOVE DISK-BAL    TO PRINT-NEW-BAL.
68 006017      WRITE PRINTER.
69 006019      REWRITE DISK INVALID KEY GO TO ERR-ROUTINE.
70 006021      GO TO RD.
71 007001 NO-RECORD.
72 007002      DISPLAY 'NO RECORD FOUND' UPON CONSOLE.
73 007003      DISPLAY KEY-ID UPON CONSOLE.
74 007004      GO TO RD.
75 007008 ERR-ROUTINE.
76 007009      DISPLAY 'NO UPDATE' UPON CONSOLE.
77 007010      DISPLAY KEY-ID UPON CONSOLE.
78 007011      GO TO RD.
79 007017 END-JOB.
80 007018      CLOSE CARD-FILE PRINT-FILE IS-FILE.
81 007019      DISPLAY 'END JOB' UPON CONSOLE.
82 007020      STOP RUN.
```

Sequential Retrieval

This program illustrates a sequential retrieval of the IS-FILE in order
to print its contents.

The Data Division is basically the same as before. The file is
opened as input. To update, it would have been opened as an INPUT-
OUTPUT file.

Prior to opening the file, the program requests the operator to enter
the starting key. The information he types will enter the KEY-ID data
name specified by the SYMBOLIC KEY clause.

The OPEN verb will then open the file and start sequential retrieval
based on the entered key. If the operator enters blanks or zeros,
retrieval will start at the beginning of the file. If he enters a key
that is not on the file, the COBOL error subroutine will identify the
error and abort the job.

During execution of the job, the COBOL error subroutine will identify
any DASD error or WLR indication and abort the job.

When the end-of-file indicator is detected, a branch is made to the
END-JOB procedures.

```
01 001001 IDENTIFICATION DIVISION.
02 001002 PROGRAM-ID.  'SEQIS'.
03      3 AUTHOR.
04      4 INSTALLATION.   360 PROGRAMMING CENTER.
05      5 DATE WRITTEN.
06 001006 REMARKS.   ILLUSTRATE SEQUENTIAL RETRIEVAL FROM IS-FILE.
07      8 ENVIRONMENT DIVISION.
08      9 CONFIGURATION SECTION.
09     10 SOURCE-COMPUTER.   IBM-360.
10     11 OBJECT-COMPUTER.   IBM-360.
11 002001 INPUT-OUTPUT SECTION.
12      2 FILE-CONTROL.
13      3     SELECT IS-FILE ASSIGN TO 'SYS004' DIRECT-ACCESS 2311
14 002004     ACCESS IS SEQUENTIAL ORGANIZATION IS INDEXED
15      5        RESERVE NO ALTERNATE AREA
16      6        SYMBOLIC KEY IS KEY-ID
17      7     RECORD KEY   IS REC-ID.
18     12     SELECT PRINT-FILE ASSIGN TO 'SYS006' UNIT-RECORD 1403
19     13        RESERVE NO ALTERNATE AREA.
20 003001 DATA DIVISION.
21    002 FILE SECTION.
22      4 FD IS-FILE       DATA RECORD IS DISK
23      5                     RECORDING MODE IS F
24      6                     LABEL RECORDS ARE STANDARD
25      7                     BLOCK CONTAINS 5 RECORDS.
26     10 01 DISK.
27     11    02  DISK-FLD1   PICTURE  X(10).
28     12    02  REC-ID      PICTURE  X(10).
29     13    02  DISK-NAME   PICTURE  X(20).
30     14    02  DISK-BAL    PICTURE  99999V99.
31     15    02  FILLER      PICTURE  X(53).
32 005001 FD PRINT-FILE   DATA RECORD IS PRINTER
33      2                     RECORDING MODE IS F
34      3                     LABEL RECORDS ARE OMITTED.
35      6 01 PRINTER.
36      7    02  PRINT-ID    PICTURE  X(10).
37      8    02  FILLER      PICTURE  X(10).
38      9    02  PRINT-NAME  PICTURE  X(20).
39     10    02  FILLER      PICTURE  X(10).
40     11    02  PRINT-BAL   PICTURE  $ZZZ,ZZZ.99-.
41        WORKING-STORAGE SECTION.
```

```
42      6    77 KEY-ID       PICTURE  X(10).
43           PROCEDURE DIVISION.
44 001000 START.
45              DISPLAY 'ENTER STARTING KEY' UPON CONSOLE.
46              ACCEPT KEY-ID FROM CONSOLE.
47 001002
48 001003    OPEN OUTPUT PRINT-FILE
49 001004        INPUT   IS-FILE.
50 001006 RD.
51 001007    READ IS-FILE AT END GO TO END-JOB.
52 001008
53 001009    MOVE REC-ID      TO PRINT-ID.
54 001010    MOVE DISK-NAME   TO PRINT-NAME.
55 001011    MOVE DISK-BAL    TO PRINT-BAL.
56 001013    WRITE PRINTER.
57 001015    GO TO RD.
58 001017 END-JOB.
59 001018    CLOSE PRINT-FILE  IS-FILE.
60 001019    DISPLAY 'END JOB'  UPON CONSOLE.
61 001020    STOP RUN.
```

DIRECT ORGANIZATION

When files are created using direct organization, the positioning of the logical records in a file is determined by keys.

The two important characteristics of direct files are:

1. Records are stored at a physical address on disk that has some mathematical relationship to the record key (in COBOL this is supplied by the SYMBOLIC KEY clause).

2. The records are not arranged in any logical sequence by key.

   Some tracks may be only partially filled, or they may have no records at all.

SPECIFYING KEYS

Both the SYMBOLIC KEY clause and the ACTUAL KEY clause must be specified for files having direct organization.

The SYMBOLIC KEY must be specified for every record and actually becomes part of that record. The ACTUAL KEY specifies the physical track address. It must be specified as an eight-byte field, and it must be defined before a record can be processed. The structure and examples of code for the 8-byte ACTUAL KEY field for both the 2311 and 2321 direct access devices follow.

The elements of the field for the 2311 are:

M	M indicates the relative number of a disk pack. It will be 0 for the first disk pack, 1 for the second disk pack, etc.
	It corresponds to the symbolic address (SYS000-SYSxxx), which is specified on the EXTENT card for the file. Thus, if three EXTENT cards with SYS011, SYS012, and SYS013, respectively, are specified, M can range from 1 to 3.
BB	This 16-bit field will always be zero for the 2311 or 2314. It refers to the cell number when addressing the data cell (2321).
CC=0-199	This 16-bit field must contain the cylinder number in binary notation.
HH=0-9 for 2311 =0-19 for 2314	This 16-bit field must contain the head number in binary notation.
R=0	R refers to the record number or sector number. When using COBOL, this field will always be zero.

An example of a method of coding the 8-byte ACTUAL KEY in binary using COBOL for the 2311 disk pack is, as follows:

```
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 IS PICTURE IS X(8).
```

Although the ACTUAL KEY field really consists of 8 bytes, 9 bytes are
defined by the given code. The 02 MM defines 2 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 8-byte binary field named THE-
ACTUAL-KEY, which is used by IOCS to access records. A pictorial struc-
ture of the THE-ACTUAL-KEY field defined by the code is, as follows:

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

The elements of the field are the same for the 2311 Data Cell except
that CC (bytes 3 and 4) is divided into subcell and strip, and HH (bytes
I7I75 and 6) is divided into headbar and head element.

An example of a method of coding the 8-byte ACTUAL KEY in binary
using COBOL for the 2321 Data cell is, as follows:

```
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 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, 9 bytes are defined and then rede-
fined to eliminate the first byte, leaving 8 bytes. Thus, the code
defines the ACTUAL KEY that is used by IOCS to access records. A pic-
torial structure of THE-ACTUAL-KEY field as defined by the code is, as
follows:

	Pack Number (M)	Cell (BB)	Cylinder (CC)		Head (HH)		Record (R)
			Sub Cell	Strip	Head Bar	Head Element	
Byte	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) (in
the example, it is defined by REC-R). When the desired number of
records is 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, thus making 20 new head
elements available.

Space is allocated on the 2321, as follows:

- 256 records per head element

- 20 head elements per head bar

- 5 head bars per strip

- 10 strips per subcell

- 10 subcells per pack

- 255 packs are available

The examples in the foregoing discussion may be used to specify the fields of the actual key. Another point of consideration is how to determine what the value of the ACTUAL KEY should be and then how to update it. One method is to prepare what is called a directly addressed file. With direct addressing, every symbolic key must convert to a unique address that is used as the value of the actual key. To use this method, records must be of fixed length and the symbolic key must be a numeric character.

Direct addressing disk time when processing is random. It is also convenient when processing is sequential because the records are written in key sequence. A possible disadvantage is that there may be a large amount of unused direct-access storage because a location must be reserved for every key in the file's range even though many of the possible keys may not be used.

Another method of determining the value of the ACTUAL KEY is called indirect addressing. Indirect addressing generally is used when the range of keys for a file includes a high percentage of unused ones so that direct addressing is not feasible. For example, employee numbers may range from 0001 to 9999, but only 3000 of the possible 9999 numbers are actually assigned. Indirect addressing is also used for nonnumeric keys.

Indirect addressing means that the symbolic key is converted to a value for the actual key by use of some algorithm intended to compress the range of addresses. Such an algorithm is usually called a randomizing technique. Randomizing techniques need not produce a unique address for every record and, in fact, such techniques usually produce synonyms. Synonyms are records whose symbolic keys randomize to the same address.

Two objectives must be considered in selecting a randomizing technique:

1. Every possible key in the file must randomize to an address in the allotted range.

2. The addresses should be distributed evenly across the range so that there are as few synonyms as possible.

Note that one way to minimize synonyms is to allot more space for the file than is actually required to hold all the records. For example, the percentage of locations that are actually used might be 80%-85% of the allotted space.

A RANDOMIZING TECHNIQUE

This randomizing technique is sometimes referred to as the division/ remainder method. For examples of other randomizing techniques, refer to the publication Introduction to IBM System/360 Direct Access Storage Devices and Organization Methods, Form C20-1649.

The ACTUAL KEY field can be thought of as a "discontinuous binary address." This is important to the programmer because it describes two conditions of which he must be aware. First, the cylinder and head number must be in binary notation, so the results of the randomizing formula must be in binary format. Second, the address is "discontinuous" because a mathematical overflow from one element (e.g., head number) does not increment the adjacent element (e.g., cylinder number).

Randomizing for the 2311 Disk Pack

When randomizing to a 2311, it is possible to circumvent the discontinuous binary address, by coding the randomizing formula in decimal arithmetic, and then converting the results to binary. This can be done by setting aside a decimal field with the low-order byte reserved for head number, and the high-order bytes reserved for cylinder number. A mathematical overflow from the head number will now increment the cylinder number and produce a valid address. The low-order byte should then be converted to binary and stored in the HH field, and the high-order bytes converted to binary and stored in the CC field of the ACTUAL KEY field.

Randomizing to the 2311 should present no significant problems if the programmer using direct organization is completely aware that the cylinder and head number gives him a unique track number. To illustrate, the 2311 could be thought of as consisting of tracks numbered as follows:

```
          Cylinder 0   Cylinder 1   Cylinder 2

Track        ___0         ___10        ___20
numbers      ___          ___          ___
             ___          ___          ___
             ___          ___          ___
             ___          ___          ___
             ___          ___          ___
             ___          ___          ___
             ___          ___          ___
             ___          ___          ___
             ___9         ___19        ___29
```
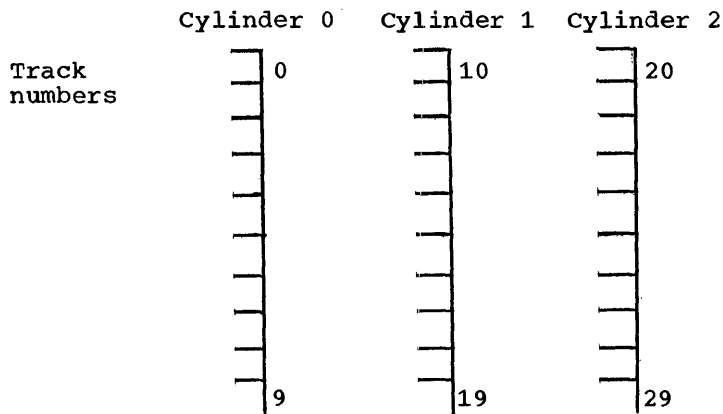
        Now if the randomizing formula resulted in an address of Cylinder 001, Head 9:

        0   0   1   9

        Cylinder   Head
           #   #

this would be a reference to track 19. This fact allows the programmer to ignore the discontinuous cylinder and head number. If his formula resulted in an address of 0 0 2 0, this would result in accessing cylinder 2, head 0, and this is where track 20 is located.

The programmer can make another use of this decimal track address.
He may wish to reserve the bottom track of each cylinder for synonyms.
If this is the case, he is, in effect, redefining the cylinder to con-
sist of 9 tracks rather than 10 tracks. The 2311 cylinder could then be
thought of as consisting of tracks numbered, as follows:

```
        Cylinder 0      Cylinder 1      Cylinder 2

        ─┐ 0            ─┐ 9            ─┐ 18
         │               │               │
        ─│              ─│              ─│ 19
         │               │               │
        ─│              ─│              ─│ 20
        ─│              ─│              ─│
        ─│              ─│              ─│
        ─│              ─│              ─│
        ─│              ─│              ─│
        ─│              ─│              ─│
        ─│              ─│              ─│
         │ 8             │ 17            │ 26
```
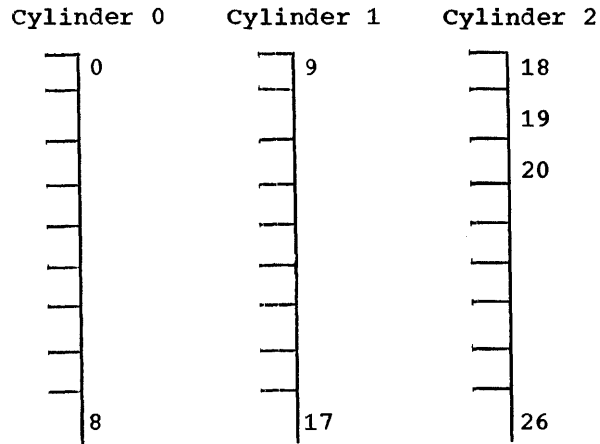
     If he randomizes to relative track number 20, he can access it by
dividing the track address by the number of tracks in a cylinder.

```
        2=Cylinder number
       ┌────
   9  │0020
       18
       ┌─
       │ 2=Head number
```

     The quotient now becomes the cylinder number, and the remainder
becomes the head number. As can be seen from the illustration, relative
track number 20 is on cylinder 2, head 2.

     To simplify randomizing,an algorithm must be developed to generate a
decimal track address. This track address can then be converted to a
binary cylinder number and head number. In addition, tracks can be
reserved by dividing the track address by the number of tracks in a
cylinder. The same concepts will hold true for devices such as the
2314. For example, an algorithm can be developed using 20 tracks per
cylinder and dividing by an approximate prime number.

Randomizing for the 2321 Data Cell

The track reference field for the data cell is composed of the following
discontinuous binary address:

```
                0-9    0-19   0-9     0-4    0-19
         │    │    │      │     │       │      │     │
     M   │ B  │ B  │  C   │  C  │   H   │  H   │  R  │
    ─────┴────┴────┴──────┴─────┴───────┴──────┴─────┘
              cell  sub    strip  cyl.  Head   Record
                    cell
```
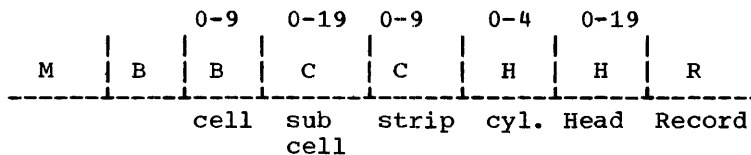
     At first glance, this presents an almost impossible randomizing task,
but since each strip comprises 100 tracks that are accessible through
cylinder and head number, the 2321 can be considered to be composed of
consecutively numbered tracks.

```
      Tracks            Strip
     0        99          0
   100──────►199          1
      ↗
   900─────►999          etc.

  1000──────►1099
      ↗
  1900──────►1999

 10000─────►10099
      ↗
 19900─────►19999
      ↗
199900─────►199999
```

On inspection, it can be seen that relative track 20 is located on cylinder 1, head 0 of some particular strip. Its address can be calculated by dividing by 20

```
          1=Cylinder #
       ┌──
  20 │ 20
       20
       ┌─
       │ 0=Head #.
```

Thus, relative track number 120 will be located on strip 1, cylinder 1, head 0 of some subcell. Note that the strip number is given by the hundreds digit, and the cylinder and head number is derived by dividing the low-order two digits by 20.

The same relationship holds true for relative track number 900. It is located on strip 9, cylinder 0, track 0. Again the hundreds digit gives the strip number, and dividing the low-order two digits by 20 results in a quotient and remainder of zero.

This relationship holds true through a relative track number of 19999, which is the number of tracks that can be contained on one cell of a data cell array. By applying the foregoing rules, an address of subcell 19, strip 9, cylinder 4, Head 19 is derived.

```
  19         9         99
  subcell    strip       4=Cylinder #
                       ───
                    20 99
                       80
                       ──
                       19=Head #
```

Thus, by randomizing to a 5-digit decimal track number, the programmer will be able to access the 20,000 tracks (40,000,000 characters) contained in a cell.

The thousands digits would represent the subcell number, the hundreds digit, the strip number, and the quotient and remainder of the two low-order digits divided by 20 would represent the cylinder and head number. Each one of these resulting decimal digits would then be converted to binary and placed in the appropriate place in the track reference field.

There is a total of 200,000 tracks per data cell array. To derive valid addresses that cross cell boundaries, the user should randomize to a 6-digit decimal track address. The highest address possible should be 199,999. To convert this to a data cell address, similar rules apply. In this case, the user must divide the three high-order digits by 20:

```
                  9=Cell
                  ┌──
          20 │199
             180
             ┌──
             │ 19=Subcell
```

The quotient becomes the cell number and the remainder becomes the
subcell number. The hundreds digit is still the strip number, and the
cylinder and head number can be derived as before. The resulting
address would be:

```
 │    │    │    │    │      │      │      │      │
 │  0 │  0 │  9 │ 19 │  9   │  4   │ 19   │  0   │
 └────┴────┴────┴────┴──────┴──────┴──────┴──────┘
   M    B    B    C     C      H      H      R
 ───────────────┬────┬──────┬──────┬──────┬─────────
                │    │      │      │      │
                │    │      │      │      │
                cell sub    strip  cyl.  Head
                     cell
```

Randomizing to the data cell can be accomplished by developing an
algorithm to generate decimal track addresses. The use of the foregoing
rules makes it possible to convert these generated track addresses to
the appropriate discontinuous binary address.

COBOL STATEMENTS USED TO SPECIFY DIRECT ORGANIZATION FILES

The following discussions show the COBOL statements required to create,
retrieve, or update a direct organization file. Also included are dis-
cussions of what functions the operating system performs when each
statement is executed.

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 statement initializes the VTOC to indicate the
presence of the labels and checks the label area for a valid output
file. It also establishes the limits of the file as defined in the
EXTENT 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 that was opened. Thus, it enters the system

logical unit specified for the file into the table. In addition, the OPEN statement initializes the capacity records (R0) over the entire area of the EXTENT for the output file.

WRITE Statement: The WRITE statement 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 statement returns the track address of the end-of-file record to 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 statement retrieves the file sequentially beginning with the lower EXTENT.

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

Key Handling: During sequential retrieval of a direct field, the SYMBOLIC KEY 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 statement 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 statement allows new records to be added to the file. When adding records to an existing file, both the ACTUAL KEY and SYMBOLIC KEY 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 Statements: 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 KEY and SYMBOLIC KEY must be initialized by the user before the READ or WRITE statement is specified. The ACTUAL KEY contains the DASD address and the SYMBOLIC KEY identifies the file.

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 programmer
must construct and include a supersetted LIOCS module that contains the
individual modules.

The following example shows how (1) the module containing the duplic-
ate entry point can be identified and (2) the supersetted module is
built and included in the COBOL object module in place of the individual
modules.

If a number of direct files are defined to be used by the same pro-
gram, the linkage editor diagnostic messages shown in Table 4 might be
obtained (they are included in the DISK LINKAGE EDITOR DIAGNOSTIC OF
INPUT).

Table 4. Linkage Editor Diagnostic Output

```
JOB  CEFI1002  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
LINK   AUTOLINK  IJHZLZZZ
LINK   AUTOLINK  IJHZRBZZ

21431  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 007000 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  00538C

                                                   CSECT    IHD03700  005680  005680
                                                   ENTRY    IHD03701  005680
                                                   ENTRY    IHD03702  005690

                                                   CSECT    CEFI1002  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
```

Note 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 21431 and

belongs to the module IJHZRBZZ. This message number is listed in the operating guide for the system and indicates an invalid duplication of an entry point label.

The programmer 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 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 be 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 module 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

ERROR RECOVERY TECHNIQUES FOR DIRECT FILES

As with indexed sequential files, error recovery may be attempted in one of three ways, as follows:

1. The INVALID KEY clause

2. COBOL error subroutine (IHD03400)

3. The USE AFTER STANDARD ERROR clause

Table 4 summarizes, by function, the conditions that cause control to be passed to the INVALID KEY, which error conditions are analyzed by the COBOL subroutine, and, optionally, those error conditions for which a USE AFTER STANDARD ERROR routine can be useful.

Table 5. Error Functions

FUNCTION ERROR	Create or Add	Random Retrieve	Sequential Retrieve
No Room Found	INVALID KEY		
No Record Found		INVALID KEY	
Wrong Length Record	USE AFTER STANDARD ERROR		
Data Check	USE AFTER STANDARD ERROR		
End of File			AT END

INVALID KEY

When creating or adding to a file, the INVALID KEY is raised if the track does not contain room enough for the record to be written. A user routine could contain coding to place the synonym on some other track based on the randomizing technique being used.

During random retrieval, an INVALID KEY error is raised if the record is not found. Depending upon how the file was created, this condition could mean that the record may be located on some other track or cylinder, or it could mean that the record is truly missing. The INVALID KEY routine could determine this.

During sequential retrieval, there is no INVALID KEY. The AT END option after the READ statement is used to determine when the end-of-file condition is reached.

USE AFTER STANDARD ERROR

For data check and wrong-length record error conditions, the user may wish to have control passed to him by writing a USE AFTER STANDARD ERROR routine in the Declarative Section of his program. This routine can interrogate bytes 244 and 255 of the DTF in a subprogram to determine the type of error that has occurred. These error indicators are:

Byte 254

X'40'	wrong-length record
X'08'	no room found

Byte 255

X'80'	data check in count
X'10'	data check in key or data
X'08'	no record found

Note that the "no room found" and the "no record found" conditions should normally be handled by INVALID KEY routines.

Normal return from the Declarative Section is to the next sequential instruction following the input/output operation (which caused the interrupt). Return by means of a GO TO statement may be made to any location within the program. It is important to remember that the last input/output operation was not completed.

Refer to the "Use After Standard Error" discussion under indexed sequential for an example showing how to interrogate the DTF table. Also, see the following discussion "Modifying the DTF Table for Drect Files."

MODIFYING THE DTF TABLE FOR DIRECT FILES

The COBOL compiler builds a skeleton DTF table for files having direct organization. When the OPEN statement is executed, a transient routine uses this information to build a complete DTF table. The contents of the skeleton DTF table are illustrated in Table 6.

Table 6. Skeleton DTF Table for Direct Organization File

```
┌─────────────────────────────────────────────────────────────────────────┐
│BYTE   0        -X'00'=RANDOM ACCESS                                       │
│                 X'F0'=SEQUENTIAL ACCESS                                    │
│       1        -X'00'=2311                                                 │
│                 X'F0'=2321                                                 │
│       2        -X'00'=RESTRICTED SEARCH                                    │
│                 X'F0'=SEARCH MULTIPLE                                      │
│       3        -X'00'=NO VERIFY                                            │
│                 X'F0'=VERIFY                                               │
│       4-7      -I/O AREA ADDRESS                                           │
│       8-11     -SYMBOLIC KEY ADDRESS                                       │
│       12-15    -ACTUAL KEY ADDRESS                                         │
│       16-19    -USER LABEL                                                 │
│       20-21    -LENGTH OF DATA FIELD                                       │
│       22-29    -SYMBOLIC FILE-NAME                                         │
│       30       -SYMBOLIC KEY LENGTH                                        │
│       31       -X'00'=INPUT                                                │
│                 X'F0'=OUTPUT                                               │
│                 X'FF'=I/O                                                  │
│       32-35    -GLOBAL TABLE ADDRESS                                       │
│       36       -X'00'=FIXED LENGTH                                         │
│                       RECORDS                                              │
│                 X'FF'=UNDEFINED LENGTH                                     │
│                       RECORDS                                              │
│       37       -X'FF'=USER LABELS                                          │
│                 X'00'=NO USER LABELS                                       │
└─────────────────────────────────────────────────────────────────────────┘
```

The programmer may modify the skeleton DTF table prior to execution of the OPEN statement in one of two ways.

1. By writing statements in the main COBOL program to access the DTF table and by writing a COBOL subprogram to modify the DTF table.

2. By writing statements in the COBOL main program to access the DTF table and by writing an assembler language subprogram to change the DTF table.

See the discussion "Modifying the DTF Table" under indexed sequential for an example of a COBOL main program and COBOL subprogram that modifies the DTF table.

Byte 3 (the 4th byte) can be changed to X'00' to suppress the verify option. Verification consists of IOCS checking to be sure that a record written out is correct. Suppression of the verify option could result in improved program performance.

CODING EXAMPLES FOR DIRECT ORGANIZATION FILES

The following examples illustrate how to create and retrieve, sequentially or randomly, a file with direct data organization.

The following job control cards can be used for the examples:

// DLBL   SYS004',DIRECT ACCESS FILE',67300,DA

// EXTENT SYS004,111111,1,1,00001,00030

// EXEC

Creating the File

This example illustrates how to create a direct organization file from cards. The DA-FILE is composed of unblocked 100 character records that are preceded by a 10-byte key. The key contains the customer identification and the data portion contains customer name balance.

The file control section defines the DA-FILE. The SYMBOLIC KEY clause defines the data name KEY-ID, which is a field in the CARD-FILE. This field will become the record key used by LIOCS to create the key portion of the disk record.

The ACTUAL KEY clause defines the data name address that is in working storage. This field will contain the binary disk address used by LIOCS for its track reference field.

To properly align the binary CC and HH portions of this track reference field on a half-word boundary, the data name address is preceded by a 01 FILLER. This forces FILLER to begin on a double-word boundary. The M portion of the address is not aligned, but the BB, CC, and HH portions of the address are all on half-word or full-word boundaries.

The M, BB, and R portions of the address are assigned a picture of X with a value of LOW-VALUE to cause a binary zero to be inserted in these portions of the address. The BB and R portions should always be binary zeros in 2311 applications. The M portion is always zero in this example, since the data file is contained on one disk pack.

If the data file extends over several disk packs, the M portion of the address would have to be changed to access the correct disk pack. For example, if the file required 250 cylinders, it could be contained on two disk packs. The first 199 cylinders could be located on the first pack (assuming that cylinder zero contains the VTOC). A cylinder address in excess of 199 would indicate that the record is to be located on the second disk pack. To illustrate, a cylinder address of 229 should be located on cylinder 30 of pack 2. This can be accomplished by subtracting 199 from the cylinder address. If the results are positive,

move a binary one to the M portion of the address, and to the results of
the subtraction to the CC portion of the address. This results in an
address, as follows:

```
|   |   |   |   |   |   |   |   |   |
| 1 | 0 | 0 | 3 | 0 | 0 | 0 | 0 |
|___|___|___|___|___|___|___|___|___|
  M   B   B   C   C   H   H   R
```
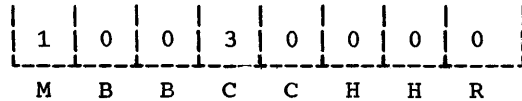
   To assure that a binary one is moved to the M field, the following
procedure may be used.

```
      WORKING-STORAGE SECTION.
      01 M-FLD PICTURE99USAGE COMPUTATIONAL.
      01 M-FLD-2 REDEFINES M-FLD.
         02 FILLER PICTURE X.
         02 M-2 PICTURE.
```

   M-FLD defines a half-word binary field. Move 1 to M-FLD. Then move
M-2 to M. This will move the eight low-order bits of M-FLD to M.

   If the literal 1 were moved directly to M, it would be stored there
as a display item with a hexadecimal notation of 'F1'. This will pro-
duce an invalid address. The recommended procedure will cause a X'01'
to be stored in M field, yeilding the desired result.

   The PROCEDURE DIVISION starts by opening the files. This will cause
the transient open routine to construct the complete DTF table. In
addition to the open functions of checking labels and creating VTOC
entries, COBOL subroutine IHD03600 is entered at this time. The purpose
of this subroutine is to issue the WRITE RECORD ZERO macro-instruction
to all the tracks specified by the EXTENT cards for the DA-FILE. This
will assure that the track is cleared and that record zero is initial-
ized with the correct information.

   This subroutine relieves the user of the responsibility of running
clear disk utilities or of rerunning the Initialize Disk program.

   After the card is read, the randomizing routine is entered. This
routine uses division by prime number. It is assumed that this small
file fits on 2 cylinders, and a prime number of 19 is used as a divisor.
The results of the routine (the remainder) are stored in TRACK, giving a
decimal track address. Ten is added to the track address to avoid any
references to cylinder zero.

   The low-order byte must now be moved to the binary HH field of the
ACTUAL KEY and the three high-order bytes to the binary CC field. This
is done by redefining the 4-digit track address into a 3-byte cylinder
number and a 1-byte head number. The move is then accomplished by
accessing the appropriate data-names.

   If there is room for the record on the accessed track, the record is
written, an audit trail is printed, and a branch is made to get the next
transaction.

   If there is no room on the track, a branch to the synonym routine is
made. This synonym routine uses the "spilling" technique. It checks
the HH field for 9 to determine whether or not the end of cylinder has
been reached. If it has not been reached, 1 is added to the head number
and the write is repeated; this time, attempting to write the synonym on
the following track.

   If the end of cylinder is reached without successfully writing the
record, a branch is made to the end-of-cylinder routine. This routine
will write synonyms on an overflow track located on cylinder 3, head 0.
If the overflow track becomes full, the job is terminated. Abnormal

termination, for this reason, would indicate the need for a better randomizing formula or more track reserved for overflow.

The END-JOB routine closes the file, and terminates the run. It will also use information obtained by COBOL subroutine IHD03100 to write an end of file record. This subroutine has kept track of the last record location. It will write the EOF record on either the last track of the extent, or after the last record, whichever is greater.

```
01 001001 IDENTIFICATION DIVISION.
02         PROGRAM-ID. 'LOADDA'.
03         REMARKS.     ILLUSTRATE CREATION OF DIRECT ACCESS FILE.
04       3 AUTHOR.
05       4 INSTALLATION. 360 PROGRAMMING CENTER.
06         ENVIRONMENT DIVISION,
07         CONFIGURATION SECTION.
08         SOURCE-COMPUTER.  IBM-360.
09         OBJECT-COMPUTER.  IBM-360.
10 002001 INPUT-OUTPUT SECTION.
11         FILE-CONTROL.
12             SELECT  DA-FILE ASSIGN TO 'SYS004' DIRECT-ACCESS 2311
13                ACCESS IS SEQUENTIAL ORGANIZATION IS DIRECT
14                RESERVE NO ALTERNATE AREA
15                SYMBOLIC KEY IS KEY-ID
16                ACTUAL KEY IS ADDRESS.
17             SELECT CARD-FILE ASSIGN TO 'SYS005' UNIT-RECORD 2540R
18                RESERVE NO ALTERNATE AREA.
19             SELECT PRINT-FILE ASSIGN TO 'SYS006' UNIT-RECORD 1403
20                RESERVE NO ALTERNATE AREA.
21 003002 DATA DIVISION.
22         FILE SECTION.
23         FD  DA-FILE      DATA RECORD IS DISK
24                          RECORDING MODE IS F
25                          LABEL RECORDS ARE STANDARD.
26         01  DISK.
27             02  DISK-NAME    PICTURE X(20).
28             02  DISK-BAL     PICTURE  99999V99.
29             02  FILLER       PICTURE X(73).
30 004001 FD  CARD-FILE    DATA RECORD IS CARDS
31                          RECORDING MODE IS F
32                          LABEL RECORDS ARE OMITTED.
33         01  CARDS.
34             02  KEY-ID   PICTURE  9(10).
35             02  CD-NAME  PICTURE  X(20).
36             02  CD-BAL   PICTURE  99999V99.
37 005001 FD  PRINT-FILE   DATA RECORD IS PRINTER
38                          RECORDING MODE IS F
39                          LABEL RECORDS ARE OMITTED.
40         01  PRINTER.
41             02  PRINT-ID     PICTURE  X(10).
42             02  FILLER       PICTURE  X(10).
43             02  PRINT-NAME   PICTURE  X(20).
44             02  FILLER       PICTURE  X(10).
45         02  PRINT-BAL        PICTURE  $ZZ,ZZ9.99-.
46         WORKING-STORAGE SECTION.
47             77  NINE       PICTURE   99      USAGE COMPUTATIONAL VALUE 09.
48             77  SAVE       PICTURE S9(10)    USAGE COMPUTATIONAL-3.
49             77  QUOTIENT   PICTURE S9999     USAGE COMPUTATIONAL-3.
50             77  PRODUCT    PICTURE S9999     USAGE COMPUTATIONAL-3.
51             01  TRACK      PICTURE S9999.
52             01  TRACK2                       REDEFINES TRACK.
53                 03  CYL    PICTURE  999.
54                 03  HEAD   PICTURE    9.
55         01  FILLER.
56             02  FILLER PICTURE X.
57             02  ADDRESS.
58                 03  M         PICTURE  X    VALUE LOW-VALUE.
```

```
59                    03  BB      PICTURE  XX    VALUE LOW-VALUE.
60                    03  CC      PICTURE  999     USAGE COMPUTATIONAL.
61                    03  HH      PICTURE  99      USAGE COMPUTATIONAL.
62                    03  R       PICTURE  X     VALUE LOW-VALUE.
63 006001 PROCEDURE DIVISION.
64          START.
65              OPEN INPUT   CARD-FILE
66                   OUTPUT PRINT-FILE  DA-FILE.
67          RD.
68              READ CARD-FILE AT END GO TO END-JOB.
69              MOVE KEY-ID TO SAVE.
70              DIVIDE 19 INTO SAVE    GIVING QUOTIENT.
71              MULTIPLY QUOTIENT BY 19 GIVING PRODUCT.
72              SUBTRACT PRODUCT FROM SAVE GIVING TRACK.
73              ADD 10 TO TRACK.
74              MOVE HEAD TO HH.
75              MOVE CYL TO CC.
76              MOVE CD-NAME TO DISK-NAME.
77              MOVE CD-BAL  TO DISK-BAL.
78          WR.
79              WRITE DISK INVALID KEY GO TO SYNONYMN-ROUTINE.
80          PRT.
81              MOVE CD-NAME TO PRINT-NAME.
82              MOVE CD-BAL  TO PRINT-BAL.
83              MOVE KEY-ID  TO PRINT-ID.
84              WRITE PRINTER.
85              GO TO RD.
86 007001 SYNONYMN-ROUTINE.
87              IF HH IS EQUAL TO NINE GO TO END-OF-CYLINDER.
88              ADD 1 TO HH.
89              GO TO WR.
90          END-OF-CYLINDER.
91              MOVE 3 TO CC.
92              MOVE ZERO TO HH.
93              WRITE DISK INVALID KEY GO TO NO-ROOM.
94              GO TO PRT.
95          NO-ROOM.
96              DISPLAY 'CYLINDER OVERFLOW FULL' UPON CONSOLE.
97          END-JOB.
98              CLOSE  CARD-FILE  PRINT-FILE  DA-FILE
99              DISPLAY 'END JOB' UPON CONSOLE.
00              STOP RUN.
```

Random Retrieval -- Direct Organization

This coding example illustrates random retrieval from the file created in the previous example.

The Data Division contains basically the same information as that in the previous example. The DA-FILE is opened as input, which permits updating. By opening the file as INPUT-OUTPUT, additions to the file could be made in the same run, or in a separate processing program.

The same randomizing formula is used as in loading the file. The SYMBOLIC KEY data-name (KEY-ID) within the card file specifies the key of the record to be found. The READ verb starts a search for an equal key on the track specified by the ACTUAL KEY address.

If the record is found, data is printed, the disk record is updated, and the next transaction is read.

Since the search was not restricted, the record will be found if it is either on the track specified or on any following tracks of the same cylinder. Only if the record is not located prior to reaching the end of the cylinder does a "no record found" condition occur. This will cause a branch to the END-OF-CYLINDER routine.

The END-OF-CYLINDER routine initiates a search for the record on the overflow cylinder. If the record is located, a branch is made to the processing routine. If another "no record found" condition occurs, the error and error record are identified, and processing continues with the next transaction.

The END-JOB routine terminates the run.

```
01 001001 IDENTIFICATION DIVISION.
02         PROGRAM-ID.   'RANDA'.
03         REMARKS.   ILLUSTRATE RANDOM RETRIEVAL FROM DA FILE.
04       3 AUTHOR.
05       4 INSTALLATION.   360 PROGRAMMING CENTER.
06         ENVIRONMENT DIVISION.
07         CONFIGURATION SECTION.
08         SOURCE-COMPUTER.   IBM-360.
09         OBJECT-COMPUTER.   IBM-360.
10 002001 INPUT-OUTPUT SECTION.
11         FILE-CONTROL.
12             SELECT  DA-FILE ASSIGN TO 'SYS004' DIRECT-ACCESS 2311
13               ACCESS IS RANDOM      ORGANIZATION IS DIRECT
14               RESERVE NO ALTERNATE AREA
15               SYMBOLIC KEY IS KEY-ID
16               ACTUAL KEY IS ADDRESS.
17             SELECT CARD-FILE ASSIGN TO 'SYS005' UNIT-RECORD 2540R
18               RESERVE NO ALTERNATE AREA.
19             SELECT  PRINT-FILE ASSIGN TO 'SYS006' UNIT-RECORD 1403
20               RESERVE NO ALTERNATE AREA.
21 003002 DATA DIVISION.
22         FILE SECTION.
23      FD DA-FILE     DATA RECORD IS DISK
24                     RECORDING MODE IS F
25                     LABEL RECORDS ARE STANDARD.
26      01 DISK.
27         02 DISK-NAME    PICTURE  X(20).
28         02 DISK-BAL     PICTURE  99999V99.
29         02 FILLER       PICTURE  X(73).
30 004001 FD CARD-FILE  DATA RECORD IS CARDS
31                     RECORDING MODE IS F
32                     LABEL RECORDS ARE OMITTED.
```

```
33          01  CARDS.
34              02  KEY-ID   PICTURE   9(10).
35              02  CD-NAME  PICTURE   X(20).
36              02  CD-AMT   PICTURE   S99999V99.
37 005001 FD   PRINT-FILE  DATA RECORD IS PRINTER
38                         RECORDING MODE IS F
39                         LABEL RECORDS ARE OMITTED.
40          01  PRINTER.
41              02  PRINT-ID      PICTURE   X(10).
42              02  FILLER        PICTURE   X(10).
43              02  PRINT-NAME    PICTURE   X(20).
44              02  FILLER        PICTURE   X(10).
45              02  PRINT-BAL     PICTURE   $ZZ,ZZ9.99-.
46              02  FILLER        PICTURE   X(10).
47              02  PRINT-AMT     PICTURE   ZZ,ZZ9.99-.
48              02  FILLER        PICTURE   X(10).
49              02  PRINT-NEW-BAL PICTURE   ZZ,ZZ9.99-.
50          WORKING-STORAGE SECTION.
51              77  NINE      PICTURE   99      USAGE COMPUTATIONAL VALUE 09.
52              77  SAVE      PICTURE S9(10)    USAGE COMPUTATIONAL-3.
53              77  QUOTIENT  PICTURE S9999     USAGE COMPUTATIONAL-3.
54              77  PRODUCT   PICTURE S9999     USAGE COMPUTATIONAL-3.
55              01  TRACK     PICTURE S9999.
56              01  TRACK2                      REDEFINES TRACK.
57                  03  CYL   PICTURE   999.
58                  03  HEAD  PICTURE     9.
59          01  FILLER.
60              02  FILLER PICTURE X.
61              02  ADDRESS.
62                  03  M     PICTURE   X       VALUE LOW-VALUE.
63                  03  BB    PICTURE   XX      VALUE LOW-VALUE.
64                  03  CC    PICTURE   999     USAGE COMPUTATIONAL.
65                  03  HH    PICTURE   99      USAGE COMPUTATIONAL.
66                  03  R     PICTURE   X       VALUE LOW-VALUE.
67 006001 PROCEDURE DIVISION.
68          START.
69              OPEN   INPUT    CARD-FILE
70                     OUTPUT   PRINT-FILE
71                     INPUT DA-FILE.
72          RD.
73              READ CARD-FILE AT END GO TO END-JOB.
74              MOVE KEY-ID TO SAVE.
75              DIVIDE 19 INTO SAVE    GIVING QUOTIENT.
76              MULTIPLY QUOTIENT BY 19 GIVING PRODUCT.
77              SUBTRACT PRODUCT  FROM   SAVE   GIVING TRACK.
78              ADD 10 TO TRACK.
79              MOVE HEAD TO HH.
80              MOVE CYL  TO CC.
81              READ DA-FILE   INVALID KEY GO TO END-OF-CYLINDER.
82          ADD.
83              MOVE DISK-BAL    TO PRINT-BAL.
84              ADD   CD-AMT     TO DISK-BAL.
85              MOVE CD-AMT      TO PRINT-AMT.
86              MOVE DISK-NAME   TO PRINT-NAME.
87              MOVE DISK-BAL    TO PRINT-NEW-BAL.
88              MOVE KEY-ID      TO PRINT-ID.
89              WRITE PRINTER.
90              REWRITE DISK.
91              GO TO RD,
92          END-OF-CYLINDER.
93              MOVE 3 TO CC.
94              MOVE ZERO TO HH.
95              READ DA-FILE INVALID KEY GO TO NO-RECORD.
96              GO TO ADD.
97          NO-RECORD.
98              DISPLAY 'NO RECORD FOUND' UPON CONSOLE.
99              DISPLAY KEY-ID  UPON CONSOLE.
00              GO TO RD.
```

```
01          END-JOB.
02                  CLOSE CARD-FILE  PRINT-FILE  DA-FILE
03                  DISPLAY 'END JOB' UPON CONSOLE.
04                  STOP RUN.
```

Sequential Retrieval -- Direct Organization

This program illustrates sequential retrieval of the direct-access
records in their physical sequence.

The starting disk address, Cylinder 1, Head 0, is moved to the ACTUAL
KEY address.

The READ statement fetches the first record and locates the address
of the next record. Both the key and data portion of the DASD record
are read since the SYMBOLIC KEY clause is specified.

A listing is made of the DA-FILE. Each READ statement issued to the
DA-FILE retrieves another record and locates the following record. This
continues until the end-of-file record is reached, at which time the
program branches to the END-JOB routine.

```
01 001001 IDENTIFICATION DIVISION.
02         PROGRAM-ID. 'SEQDA'.
03         REMARKS.        ILLUSTRATE SEQ. RETRIEVAL OF DIRECT ACCESS FILE.
04       3 AUTHOR.
05       4 INSTALLATION.   360 PROGRAMMING CENTER.
06         ENVIRONMENT DIVISION.
07         CONFIGURATION SECTION.
08         SOURCE-COMPUTER.   IBM-360.
09         OBJECT-COMPUTER.   IBM-360.
10 002001 INPUT-OUTPUT SECTION.
11         FILE-CONTROL.
12             SELECT  CA-FILE ASSIGN TO 'SYS004' DIRECT-ACCESS 2311
13                ACCESS IS SEQUENTIAL ORGANIZATION IS DIRECT
14                RESERVE NO ALTERNATE AREA
15                SYMBOLIC KEY IS KEY-ID
16                ACTUAL KEY IS ADDRESS.
17             SELECT  PRINT-FILE ASSIGN TO 'SYS006' UNIT-RECORD 1403
18                RESERVE NO ALTERNATE AREA.
19 003002 DATA DIVISION.
20         FILE SECTION.
21         FD  DA-FILE      DATA RECORD IS DISK
22                          RECORDING MODE IS F
23                          LABEL RECORDS ARE STANDARD.
24         01  DISK.
25             02  DISK-NAME    PICTURE   X(20).
26             02  DISK-BAL     PICTURE   99999V99.
27             02  FILLER       PICTURE   X(73).
28 005001 FD  PRINT-FILE   DATA RECORD IS PRINTER
29                          RECORDING MODE IS F
30                          LABEL RECORDS ARE OMITTED.
31         01  PRINTER.
32             02  PRINT-ID     PICTURE   X(10).
33         02  FILLER           PICTURE   X(10).
34             02  PRINT-NAME   PICTURE   X(20).
35             02  FILLER       PICTURE   X(10).
36             02  PRINT-BAL    PICTURE   $ZZ,ZZ9.99-.
37         WORKING-STORAGE SECTION.
38             77  KEY-ID       PICTURE   X(10).
39         01  FILLER.
40             02  FILLER  PICTURE X.
41             02  ADDRESS.
42                 03  M            PICTURE   X    VALUE LOW-VALUE.
43                 03  BB           PICTURE   XX   VALUE LOW-VALUE.
```

```
44                  03  CC      PICTURE  999    USAGE COMPUTATIONAL.
45                  03  HH      PICTURE  99     USAGE COMPUTATIONAL.
46                  03  R       PICTURE  X   VALUE LOW-VALUE.
47 006001 PROCEDURE DIVISION.
48         START.
49             MOVE  ZERO TO HH.
50             MOVE  1     TO CC.
51             OPEN  INPUT  DA-FILE
52                   OUTPUT PRINT-FILE.
53         RD.
54             READ DA-FILE AT END GO TO END-JOB.
55             MOVE DISK-NAME  TO PRINT-NAME.
56             MOVE DISK-BAL   TO PRINT-BAL.
57             MOVE KEY-ID     TO PRINT-ID.
58         PRT.
59             WRITE PRINTER.
60
61             GO TO RD.
62         END-JOB.
63             DISPLAY 'END OF JOB' UPON CONSOLE.
64             CLOSE  DA-FILE  PRINT-FILE.
65             STOP RUN.
```

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 15,48(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 48 in statement 30 of the example) may change in subsequent modification of the compiler.


```
STMNT   SOURCE STATEMENT

0001    OVRLAY START 0
0002          ENTRY OVRLAY
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 ADCON 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          CLC0(8,1),CORSUBIS IT IN CORE
0018          BE SUBIN              YES
0019          MVCCORSUB(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   *
0025       SVC4                 LOAD PHASE
0026       ST 1,ASUB            ASUB=ENTRY POINT OF PHASE
0027   SUBLN 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 15,48(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   SAVDC 4X'FF'                 REGISTER SAVE AREA
0035   ASUDC 4X'FF'            ADDRESS OF SUBROUTINE
0036   CORDCB8X'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 then 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
0033,0037 - Defines storage, defines constants and end of routine
            instruction.

ASSEMBLER LANGUAGE SUBPROGRAMS

CALLED AND CALLING PROGRAMS

Any program referred to by another program is called subprogram. If
this called subprogram refers to another subprogram, it is both a called
and calling subprogram. In Figure 19, 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.

```
+-------------------------------------------------------------------------+
|  +----------------------+    +------------------+    +----------------+  |
|  |          A           |    |        B         |    |       C        |  |
|  |                      |    |                  |    |                |  |
|  |   Calling            |------>|  Called          |    |                |  |
|  |   program            |    |  subprogram      |    |                |  |
|  |   of B               |    |  of A            |    |                |  |
|  |                      |    +------------------+    |                |  |
|  |                      |    |  Calling         |------>|  Called        |  |
|  |                      |    |  subprogram      |    |  subprogram    |  |
|  |                      |    |  of C            |    |  of B          |  |
|  +----------------------+    +------------------+    +----------------+  |
+-------------------------------------------------------------------------+
```
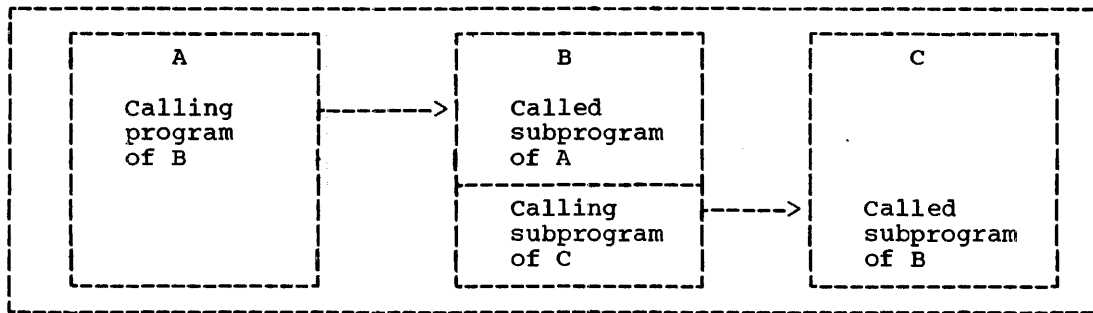
Figure 19. Called and Calling Programs

     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 conven-
tions to give control to and return control from assembler-written sub-
programs. These conventions, called linkage conventions, are described
in the following text.

LINKAGE CONVENTIONS

The save and return routines for assembler subprograms need not be writ-
ten 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 regis-
    ters 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 20.

156 Disk and Tape Operating Systems COBOL Programmer's Guide

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 20. 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 maximun amount of storage reserved by the calling subprogram is 18 words. All registers are saved. Figure 21 shows the layout of the save area and the contents of each word.

```
r------------------------------------------------------------------------------
| AREA      r--------------------------------------------------------------------
|  (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.                           |
|            L--------------------------------------------------------
L------------------------------------------------------------------------------
```

Figure 21. 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 22. 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 full-word boundary.

```
deckname    START    0
            ENTRY    name₁
            EXTRN    name₂



            USING    name₁,15
*  Save Routine
name,       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       r₃,15
            DROP     15
            USING    name₁,r₃         where r₃ and r₂ have been saved
            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)       Stores the address of this program's
*                                    save area into word 3 of the save area
*                                    of the callng 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
            1,ARGLST                  First statement in calling sequence.
            L        15,ADCON
            BALR     14,15
*           Remainder of user-written program statements
*  Return Routine
            L        13,4(0,13)       First statement in return routine.
*                                    Loads the address of the previous save
*                                    area back into register 13.
            LM       2,r₁,28(13)      The contents of registers 2 through r₁,
*                                    are restored from the previous save area
            L        14,12(13)        Loads the return address, which is in
*                                    word 4 of the calling program's save
*                                    area, into register 14.
            MVI      12(13),X'FF'     Sets flag FF in the save area of the
*                                    calling program to indicate that 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 22. Sample Linkage Routines Used with a Calling 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 22. If the
assembler subprogram uses any registers, it must save them. Figure 23
shows the appropriate linkage conventions used by an assembler subpro-
gram 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 23. 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 an out-of-line list. In this case, he may substitute the calling
sequence and parameter list shown in Figure 24 for that shown in Figure
22.

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 dif-
ferent 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 24. Sample In-line Parameter List

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.]

     ASSIGN TO external-name $\left\{\begin{array}{l}\text{DIRECT-ACCESS} \\ \text{UTILITY} \\ \text{UNIT-RECORD}\end{array}\right\}$ device-number UNIT [S]

     [RESERVE $\left\{\begin{array}{l}\text{NO} \\ \text{1}\end{array}\right\}$ ALTERNATE AREA[S]]

     [ACCESS IS $\left\{\begin{array}{l}\text{SEQUENTIAL} \\ \text{RANDOM}\end{array}\right\}$]
     [ORGANIZATION IS $\left\{\begin{array}{l}\text{INDEXED} \\ \text{DIRECT}\end{array}\right\}$]

     [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...].]

RERUN ON external-name $\left\{\begin{array}{l}\text{DIRECT-ACCESS} \\ \text{UTILITY}\end{array}\right\}$ device-number UNIT[S]]

     EVERY integer RECORDS 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 $\left\{\begin{array}{l}\text{CHARACTERS} \\ \text{RECORDS}\end{array}\right\}$]

     [RECORDING MODE IS $\left\{\begin{array}{l}\text{U} \\ \text{F} \\ \text{V}\end{array}\right\}$]

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

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

DATA {RECORDS ARE / RECORD IS} record-name...

Record Description Entry.

WORKING-STORAGE SECTION.

Record Description entries

LINKAGE SECTION.

Record Description entries

level-number {data-name / FILLER} [REDEFINES data-name-2] [COPY library-name.]

PICTURE IS {alpha-form / an-form / numeric-form / report-form / fp-form}

[OCCURS integer TIMES [DEPENDING ON data-name]]

[JUSTIFIED RIGHT]

[BLANK WHEN ZERO]

[VALUE IS literal]

USAGE IS {DISPLAY / COMPUTATIONAL / COMPUTATIONAL-1 / COMPUTATIONAL-2 / COMPUTATIONAL-3}

PROCEDURE DIVISION

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

END DECLARATIVES.

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

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

USE AFTER STANDARD ERROR PROCEDURE ON file-name.

Conditionals.

IF Statement.

IF condition [THEN] $\left\{\begin{array}{l}\text{statement-1...}\\ \underline{\text{NEXT SENTENCE}}\end{array}\right\}$ $\left[\left\{\begin{array}{l}\underline{\text{ELSE}}\\ \underline{\text{OTHERWISE}}\end{array}\right\}\left\{\begin{array}{l}\text{statement-2...}\\ \underline{\text{NEXT SENTENCE}}\end{array}\right\}\right]$

Relation Test.

$\left\{\begin{array}{l}\text{data-name-1}\\ \text{arithmetic-expression-1}\\ \text{figurative-constant-1}\\ \text{literal-1}\end{array}\right\}$ IS [NOT] $\left\{\begin{array}{l}>\\ <\\ =\\ \underline{\text{GREATER}}\ \text{THAN}\\ \underline{\text{LESS}}\ \text{THAN}\\ \underline{\text{EQUAL TO}}\end{array}\right\}$ $\left\{\begin{array}{l}\text{data-name-2}\\ \text{arithmetic}\\ \text{expression-2}\\ \text{figurative}\\ \text{constant-2}\\ \text{literal-2}\end{array}\right\}$

Sign Test.

$\left\{\begin{array}{l}\text{data-name}\\ \text{arithmetic-expression}\end{array}\right\}$ IS [NOT] $\left\{\begin{array}{l}\underline{\text{POSITIVE}}\\ \underline{\text{ZERO}}\\ \underline{\text{NEGATIVE}}\end{array}\right\}$

Class Test.

{data-name IS [NOT]} $\left\{\begin{array}{l}\underline{\text{NUMERIC}}\\ \underline{\text{ALPHABETIC}}\end{array}\right\}$

Condition Name Test.

[NOT] condition-name

Overflow Test.

[NOT] overflow-name

Open and Close Statements.

OPEN $\left(\begin{array}{l}\underline{\text{INPUT}}\quad\{\text{file-name}\ [\text{WITH}\ \underline{\text{NO REWIND}}\quad[\underline{\text{REVERSED}}]]\}\ ...\\ \qquad[\underline{\text{OUTPUT}}\{\text{file-name}\ [\text{WITH}\ \underline{\text{NO REWIND}}]\}....]\\ \qquad[\underline{\text{I-O}}\ \{\text{file-name}\}\ ...]\\ \\ \underline{\text{OUTPUT}}\ \{\text{file-name}\ [\text{WITH}\ \underline{\text{NO REWIND}}]\}...\\ \qquad[\underline{\text{INPUT}}\ \{\text{file-name}\ [\text{WITH}\ \underline{\text{NO REWIND}}\ [\underline{\text{REVERSED}}]]\}...]\\ \qquad[\underline{\text{I-O}}\ \{\text{file-name}\}\ ...]\\ \\ \underline{\text{I-O}}\ \{\text{file-name}\}...[\underline{\text{OUTPUT}}\{\text{file-name}\ [\text{WITH}\ \underline{\text{NO REWIND}}]\}...]\\ \qquad[\underline{\text{INPUT}}\ \{\text{file-name}\ [\text{WITH}\ \underline{\text{NO REWIND}}[\underline{\text{REVERSED}}]]\}...]\end{array}\right)$

$\underline{\text{CLOSE}}\left\{\text{file-name}\quad\begin{array}{l}[\underline{\text{UNIT}}]\\ [\underline{\text{REEL}}]\end{array}\left[\text{WITH}\left\{\begin{array}{l}\underline{\text{NO REWIND}}\\ \underline{\text{LOCK}}\end{array}\right\}\right]\right\}...$

Input/Output Verbs

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

READ file-name RECORD [INTO data-name]

    $\left\{\begin{array}{l}\text{AT}\ \underline{\text{END}}\\ \underline{\text{INVALID}}\ \text{KEY}\end{array}\right\}$ imperative statement...

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

164 Disk and Tape Operating Systems COBOL Programmer's Guide

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

    [AFTER ADVANCING $\begin{Bmatrix} \text{data-name-2} \\ \text{integer} \end{Bmatrix}$ LINES]

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...]

DISPLAY $\begin{Bmatrix} \text{data-name} \\ \text{literal} \end{Bmatrix}$ ... $\begin{bmatrix} \underline{\text{UPON CONSOLE}} \\ \underline{\text{UPON SYSPUNCH}} \end{bmatrix}$

ACCEPT data-name   [FROM CONSOLE]

Data Manipulation Verbs

MOVE $\begin{Bmatrix} \text{data-name-1} \\ \text{literal} \end{Bmatrix}$ TO data-name-2 ...

Option 1

EXAMINE data-name TALLYING $\begin{Bmatrix} \underline{\text{ALL}} \\ \underline{\text{LEADING}} \\ \underline{\text{UNTIL FIRST}} \end{Bmatrix}$ 'character-1'

    [REPLACING BY 'character-2']

Option 2

EXAMINE data-name REPLACING $\begin{Bmatrix} \underline{\text{ALL}} \\ \underline{\text{LEADING}} \\ \underline{\text{UNTIL FIRST}} \\ \underline{\text{FIRST}} \end{Bmatrix}$ 'character-1'

    BY 'character-2'

TRANSFORM data-name-3 CHARACTERS

FROM $\begin{Bmatrix} \text{figurative-constant-1} \\ \text{non-numeric-literal-1} \\ \text{data-name-1} \end{Bmatrix}$ TO $\begin{Bmatrix} \text{figurative-constant-2} \\ \text{non-numeric literal-2} \\ \text{data-name-2} \end{Bmatrix}$

Arithmetic Verbs

ADD $\left\{\begin{array}{l}\text{numeric-literal} \\ \text{floating point literal} \\ \text{data-name-1}\end{array}\right\}$ ... $\left\{\begin{array}{l}\underline{\text{TO}} \\ \underline{\text{GIVING}}\end{array}\right\}$ data-name-n

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

SUBTRACT $\left\{\begin{array}{l}\text{data-name-1} \\ \text{numeric-literal-1} \\ \text{floating-point-literal-1}\end{array}\right\}$ ...

  FROM $\left\{\begin{array}{ll}\text{data-name-m} & [\underline{\text{GIVING}} \text{ data-name-n}] \\ \text{numeric-literal-m} & \underline{\text{GIVING}} \text{ data-name-n} \\ \text{floating-point-literal-m} & \underline{\text{GIVING}} \text{ data-name-n}\end{array}\right\}$

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

MULTIPLY $\left\{\begin{array}{l}\text{data-name-1} \\ \text{numeric-literal-1} \\ \text{floating-point-literal-1}\end{array}\right\}$

  BY $\left\{\begin{array}{ll}\text{data-name-2} & [\underline{\text{GIVING}} \text{ data-name-3}] \\ \text{numeric-literal-2} & \underline{\text{GIVING}} \text{ data-name-3} \\ \text{floating-point-literal-2} & \underline{\text{GIVING}} \text{ data-name-3}\end{array}\right\}$

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

DIVIDE $\left\{\begin{array}{l}\text{data-name-1} \\ \text{numeric-literal-1} \\ \text{floating-point-literal-1}\end{array}\right\}$

  INTO $\left\{\begin{array}{ll}\text{data-name-2} & [\underline{\text{GIVING}} \text{ data-name-3}] \\ \text{numeric-literal-2} & \underline{\text{GIVING}} \text{ data-name-3} \\ \text{floating-point-literal-2} & \underline{\text{GIVING}} \text{ data-name-3}\end{array}\right\}$

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

COMPUTE data-name-1   [ROUNDED] = $\left\{\begin{array}{l}\text{data-name-2} \\ \text{numeric-literal} \\ \text{floating-point-literal} \\ \text{arithmetic-expression}\end{array}\right\}$

   [ON SIZE ERROR imperative-statement...]

Procedure Branching Statements.

       STOP $\left\{\begin{array}{l}\underline{\text{RUN}} \\ \text{literal}\end{array}\right\}$

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]

<u>Option 2</u>

<u>PERFORM</u> procedure-name-1 [<u>THRU</u> procedure-name-2] $\left\{ \begin{array}{c} \text{integer} \\ \text{data-name} \end{array} \right\}$ <u>TIMES</u>

<u>Option 3</u>

<u>PERFORM</u> procedure-name-1 [<u>THRU</u> procedure-name-2]
        <u>UNTIL</u> test-condition

<u>Option 4</u>

<u>PERFORM</u> procedure-name-1 [<u>THRU</u> procedure-name-2]
    <u>VARYING</u> data-name-1   <u>FROM</u> $\left\{ \begin{array}{c} \text{numeric-literal-2} \\ \text{data-name-2} \end{array} \right\}$

    <u>BY</u> $\left\{ \begin{array}{c} \text{numeric-literal-3} \\ \text{data-name-3} \end{array} \right\}$ <u>UNTIL</u> test-condition-1

    [<u>AFTER</u> data-name-4]   <u>FROM</u> $\left\{ \begin{array}{c} \text{numeric-literal-4} \\ \text{data-name-5} \end{array} \right\}$

    <u>BY</u> $\left\{ \begin{array}{c} \text{numeric-literal-6} \\ \text{data-name-6} \end{array} \right\}$ [<u>UNTIL</u> test-condition-2]

    [<u>AFTER</u> data-name-7   <u>FROM</u> $\left\{ \begin{array}{c} \text{numeric-literal-8} \\ \text{data-name-8} \end{array} \right\}$

    <u>BY</u> $\left\{ \begin{array}{c} \text{numeric-literal-9} \\ \text{data-name-9} \end{array} \right\}$ <u>UNTIL</u> test-condition-3

<u>Compiler-Directing Statements</u>.

<u>ENTER LINKAGE</u>.
<u>CALL</u> entry-name   [<u>USING</u> argument...]
<u>ENTER COBOL</u>.

<u>ENTER LINKAGE</u>.
<u>ENTRY</u> entry-name   [<u>USING</u> data-name...]
<u>ENTER COBOL</u>.

<u>ENTER LINKAGE</u>.
<u>RETURN</u>.
<u>ENTER COBOL</u>.

<u>EXIT Statement</u>.

paragraph-name.   <u>EXIT</u>.

<u>NOTE Statement</u>.

<u>NOTE</u> comment...

<u>Option 1</u>.

paragraph-name.   <u>INCLUDE</u> library-name.

<u>Option 2</u>.

section-name <u>SECTION</u>.   <u>INCLUDE</u> library-name.

COPY Statement.

(within the Input-Output Section):

$$\left\{ \begin{array}{l} \underline{\text{FILE-CONTROL}}. \\ \underline{\text{I-O-CONTROL}}. \end{array} \right\} \underline{\text{COPY}} \text{ library-name.}$$

(within the File-Control Paragraph):

   $\underline{\text{SELECT}}$ file-name $\underline{\text{COPY}}$ library-name.

(within the File Section):

   FD file-name $\underline{\text{COPY}}$ library-name.

(within a file, Working Storage or Linkage Section):

   01 data-name $\underline{\text{COPY}}$ library-name.

(within Working Storage or Linkage Section):

   77 data-name $\underline{\text{COPY}}$ library-name.

COBOL Debugging Statements.

TRACE Statement.

$$\left\{ \begin{array}{l} \underline{\text{READY}} \\ \underline{\text{RESET}} \end{array} \right\} \quad \underline{\text{TRACE}}$$

EXHIBIT Statement.

$$\underline{\text{EXHIBIT}} \left\{ \begin{array}{l} \underline{\text{NAMED}} \\ \underline{\text{CHANGED NAME}} \\ \underline{\text{CHANGED}} \end{array} \right\} \left\{ \begin{array}{l} \text{data-name} \\ \text{non-numeric-literal} \end{array} \right\} \quad \ldots$$

ON (Count-Conditional) Statement

$\underline{\text{ON}}$ integer-1 [$\underline{\text{AND EVERY}}$ integer-2] [$\underline{\text{UNTIL}}$ Integer-3]

$$\left\{ \begin{array}{l} \text{imperative-statement } \ldots \\ \underline{\text{NEXT SENTENCE}} \end{array} \right\} \left[\left\{ \begin{array}{ll} \underline{\text{ELSE}} & \text{statement } \ldots \\ \underline{\text{OTHERWISE}} & \underline{\text{NEXT SENTENCE}} \end{array} \right\} \right]$$

Debug Packet Statement

```
1      8
*DEBUG location
```

PERMISSIBLE COMPARISONS

	SECOND OPERAND									
	GR	AL	AN	ED	ID	BI	EF	IF	RP	FC
Group Item (GR)	NN	NN	NN	NN	NN	NN	NN	NN	NN	NN
Alphabetic Item (AL)	NN	NN	NN							NN[1]
Alphanumeric (non-report) Item (AN)	NN	NN	NN	NN[5]					NN	NN
External Decimal Item (ED)	NN		NN[5]	NU	NU	NU	NU	NU		NN[3]
Internal Decimal Item (ID)	NN			NU	NU	NU	NU	NU		NU[2]
Binary Item (BI)	NN			NU	NU	NU	NU	NU		NU[2]
External Floating-point Item (EF)	NN			NU	NU	NU	NU	NU		NU[2]
Internal Floating-point Item (IF)	NN			NU	NU	NU	NU	NU		NU[2]
Report Item (RP)	NN		NN						NN	NN[4]
Figurative Constant (FC)	NN	NN[1]	NN	NN[3]	NU[2]	NU[2]	NU[2]	NU[2]	NN[4]	

Abbreviations for Types of Comparison
NN - Comparison as described for non-numeric items
NU - Comparison as described for numeric items
[1] Permitted with the figurative constants SPACE and ALL 'character'
  where character must be alphabetic.
[2] Permitted only if figurative constant is ZERO.
[3] Permitted only if figurative constant is ZERO or ALL 'character'
  where character must be numeric.
[4] Not permitted with figurative constant QUOTE.
[5] External decimal field must consist of integers.

Note: The first column of the table is labelled vertically "FIRST OPERAND".

PERMISSIBLE MOVES

Source Field	Receiving Field								
	GR	AL	AN	ED	ID	BI	EF	IF	RP
Group (GR)	Y	Y	Y	N	N	N	N	N	N
Alphabetic (AL)	Y	Y	Y	N	N	N	N	N	N
Alphanumeric (AN)	Y	Y	Y	N	N	N	N	N	N
External Decimal (ED)	Y	N	Y¹	Y	Y	Y	Y	Y	Y
Internal Decimal (ID)	Y	N	Y¹	Y	Y	Y	Y	Y	Y
Binary (BI)	Y	N	Y¹	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

¹For integers only.

File
Label
Number

		File Identifier		File Serial Number	Volume Sequence Number	File Sequence Number	Generation Number		Creation Date	Expiration Date		Block Count		System Code		Reserved For A.S.A.

Label
Identifier

Version
Number of
Generation
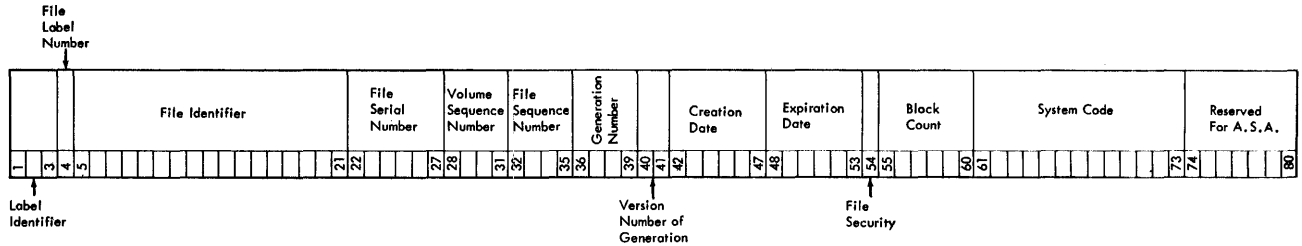
File
Security

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.

FIELD	NAME AND LENGTH	DESCRIPTION
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)

FIELD	NAME AND LENGTH	DESCRIPTION
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.

Creation Date — Expiration Date — Spare

| File Name | File Serial Number | System Code |

Format Identifier — Volume Sequence Number — Count Of Extents — Bytes Used in last block of directory

Option Codes — Record Length — Key Location — Space Remaining

| Reserved For Future Use | File Type | | | | Secondary Allocation | Last Used Track & Record On That Track | First Extent | | Additional Extent | Additional Extent | Pointer |

First Extent: Lower Limit, Upper Limit

Reserved For Future Use — Record Format — Block Length — Key Length — Data Set Indicators — Extent Type Indicator — Extent Sequence Number

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.

FIELD	NAME AND LENGTH	DESCRIPTION
		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

Bit Position	Content	Meaning
4	0	No truncated records
	1	Truncated records in file
5 and 6	01	Control character ASA code
	10	Control 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 fields 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.

The track format for the 2311, 2314, and 2321 is illustrated in Figure
25.   The names of the fields are described in the following discussion.

Index Marker: All tracks start with an index marker.   It is a signal to
the hardware indicating the beginning of the track.

Home Address: The home address, preceded by a gap, follows the index
marker.   The home address uniquely identifies each track by specifying
the cylinder and head number.

Track Descriptor Record (Record Zero):   Record zero consists of two
parts: a count portion and a data portion.   The count portion is the
same as it is for any other record (see the following description of
count for record one).   The 8-byte data portion is used to record infor-
mation used by LIOCS.   The information in the data portion depends on
the data organization (direct or indexed sequential) that is being used.

For direct organization, this portion in the form of CCHHR contains
the address of the last record on the track and the number of bytes
remaining on the track.   This information is used to determine if there
is room for another record on the track.   For indexed sequential, the
data portion contains the address of the last record in the cylinder
overflow area and the number of tracks remaining in the cylinder over-
flow area.   Record zero is then used as the cylinder overflow control
record.

Address Marker: All records after record zero will be preceded by a
2-byte address marker.   The address marker is a signal to the hardware
that a record is starting.

Data Records: Data records (see R1 in Figure 25) can consist of a count
and data portion for sequential organization, or a count, key, and data
portion for direct and indexed sequential organizations.

1.   Count Portion.   The count portion contains the identification of
     each record, the key length, and the data length.

     Identification.   Each record is identified with its cylinder
     number, head number, or record number.   The cylinder and head num-
     bers will be the same as those of the home address.   The record
     number will indicate which record this is on the track.   That is,
     the first record after record zero will be record 1, followed by
     record 2, etc.   This 5-byte binary field in the form of CCHHR is
     often referred to as the record ID.

     Key Length.   The key length is specified in an 8-bit byte; its
     length can range from zero to 255.   This field will contain a zero
     if there is no key.

     Data Length.   The data length is specified in the 16 bits of the
     next 2 bytes.

     Note: It is the count portion that identifies the presence or
     absence of a key, as well as indicates the data length.   In this
     way, each record is unique and self formatting.

2.   Key Portion.   The key portion of the record is normally used to
     store the control field of the data record such as a man number.
     Direct and indexed sequential files must have a key portion.

3.   Data Portion.   The data portion of the record contains the data
     record.

Note that all records, including the data record, end with a 2-byte cyclic check. The hardware uses this cyclic check to assure that it correctly reread what it had written. The cyclic check is cumulative and is appended to each record when it is written. Upon reading the record, the cyclic check is again accumulated and then compared with the appended cyclic check. If they do not agree, a data check is initiated.

The first byte of the count portion of each record and the home address is reserved for a flag byte. If a track becomes defective, a utility may be used to transfer the data to an alternate track. (Cylinders 200 through 202 are reserved for alternate tracks on the 2321. Strips 6 through 9 of subcell 19 of each cell are reserved for alternate tracks on the 2321.) In this case, a flag bit within the byte is set on to indicate that this is a defective track and the address of an alternate track will be placed in the record ID of record zero. Subsequent references to this defective track will result in the supervisor accessing record zero for the address of the alternate track.

● Figure 25.   Track Format

APPENDIX F: EXAMPLES OF COBOL PROGRAMS

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.

Figure 26. Example of a Calling Program (Part 1 of 2)

Figure 26. Example of a Calling Program (Part 2 of 2)

COBOL PROGRAM SHEET

System	IBM SYSTEM/360		Punching Instructions		Sheet 3 of 4
Program	EXAMPLE OF A SUBPROGRAM	Graphic		Card Form#	*	Identification
Programmer		Date	Punch			73 — 80

```
SEQUENCE
(PAGE)(SERIAL)  A  B
003001  IDENTIFICATION DIVISION.
  002   PROGRAM-ID. 'SUBPROG'.
  003   REMARKS. EXAMPLE OF A SUBPROGRAM.
  004   ENVIRONMENT DIVISION.
  005   CONFIGURATION SECTION.
  006   SOURCE-COMPUTER. IBM-360 D30.
  007   OBJECT-COMPUTER. IBM-360 D30.

  008   DATA DIVISION.
  009   WORKING-STORAGE SECTION.

  010   77  MODIFICATION PICTURE X(12),VALUE IS 'PUT ANY DATA'.
  011   LINKAGE SECTION.
  012   01  PASS-FIELD.
  013       02 A PICTURE X(68).
  014       02 B PICTURE X(12).

  015   PROCEDURE DIVISION.
  016   START. ENTER LINKAGE
  017       ENTRY 'SUBPRGM' USING PASS-FIELD
  018       ENTER COBOL.
  019   MODIFY. MOVE MODIFICATION TO B.
  020       ENTER LINKAGE.
003021     RETURN.
```

Figure 27. Example of a Subprogram (Part 1 of 2)

COBOL PROGRAM SHEET

System	IBM SYSTEM/360		Punching Instructions		Sheet 4 of 4
Program	EXAMPLE OF A SUBPROGRAM	Graphic		Card Form#	*	Identification
Programmer		Date	Punch			73 — 80

```
SEQUENCE
(PAGE)(SERIAL)  A  B
004001        ENTER COBOL.
004002        NOTE THAT PASS-FIELD IN THIS PROGRAM IS THE IDENTICAL
              AREA DEFINED AS RECORD-2 IN THE CALLING PROGRAM.
```

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 to isolate a troublespot (debugging).

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
   16-byte 30-character
   divisor producing a 16-byte
   30-character quotient.
   No registers are used.
	Required for division of complex computations, 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 computations, COMPUTATIONAL fields of over 9, or COMPUTATIONAL-3 of over 16 digits.

IHD00400
   Error message subroutine.
   It generates execution time messages.
	Required with floating-point and non-integer exponentiation.

IHD00500
   Packed exponentiation
   subroutine.
	Required for exponentiation to an integer power. [Used with IEP00700 (floating-point exponentiation) subroutine.]

IHD00600
   Floating-point
   logarithm subroutine.
	Required whenever floating-point conversion is needed. Used with IEP00700 (floating-point exponentiation) subroutine.

IHD00700 Floating-point exponentiation subroutine.	Required to set up floating-point conversion routines for nonfloating point exponentiation.

SUBROUTINE NAME	ACTION

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 than 9 decimal digits.
	  - If sending field is binary and receiving field is packed, and sending field is greater than 9 decimal digits.
	  COMPUTATIONAL - If one field is binary and the other is zoned.
	  - If one field is binary and the other is packed.
	  - If both fields are binary and A is less than 10, and B is less than 10 and the scales of both fields are equal.

SUBROUTINE NAME	ACTION
	- 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.
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 IHD0100 (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.

SUBROUTINE NAME	ACTION
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.	Required for: MOVE - Required if the sending field is external decimal, and receiving field is packed, receiving field must be 9 decimal digits. COMPUTATIONAL - If one field is binary or zoned and one field is packed. - If both fields are binary and the following conditions are not met: • the length of the fields are unequal • A and B are both less than 10 and the scales of the fields are equal - If the scale of the sending field is greater than the scale of the receiving field and the real or implied integer positions of the receiving field plus the scale of the sending field is less than 10. - If the scale of the sending field is less than the scale of the receiving field and the real or implied decimal positions plus the scale of the receiving field is less than 10.

SUBROUTINE NAME	ACTION
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 same 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 internal 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 <u>IBM System/360 Disk and Tape Operating Systems: COBOL Language Specifications</u>, Form C24-3433.

SUBROUTINE NAME	ACTION
IHD02200 Converts a packed decimal number to a zoned decimal number.	ARITHMETIC - Required when the operations are performed in packed decimal and the receiving field is zoned. MISCELLANY - Required if user displays packed decimal format.
IHD02300	This subroutine consists of three parts: 1. The first part builds a table of the beginning and end addresses of the PERFORM or nested PERFORM statements 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 clauses.
	  Requirements:
	    R1 points to FIELD1.
	    R2 points to FIELD2.
	    WORKA is the same length as FIELD1.
	    WORKA+2 is the same length as FIELD2.

SUBROUTINE NAME	ACTION
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 execution 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.
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.

SUBROUTINE NAME	ACTION
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 diagnostic messages when errors are encountered in direct-access processing. Required when IHD03100 is used.
IHD03500	Produces object time diagnostic messages 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.
IHD03800	Used for maintaining a list of tapes to be repositioned, linking to the system's checkpoint routine, and providing a restart entry point.
IHD03900	Converts internal decimal to sterling non-report.
IHD04000	Converts sterling non-report to internal decimal.
IHD04100	Edits internal decimal into sterling report.

APPENDIX H: DIAGNOSTIC MESSAGES

This appendix contains a detailed description of the diagnostic messages that are generated during processing. They consist of:

- Compiler diagnostic messages

- Execution time messages

- Debug packet error messages

Certain conditions that may occur when a module is being processed will generate linkage editor diagnostic messages. For a complete description of these messages, see the publications IBM System/360 Disk Operating System, System Control and System Service Programs, Form C24-5036, and IBM System/360 Tape Operating System, System Control and System Service Programs, Form C24-5034.

COMPILER DIAGNOSTIC MESSAGES

IJS001I C   LITERAL EXCEEDS 120 CHARACTERS.

        System Action: The element count begins following the next quote on the line if there is one, or following the element beginning after the 120th character.

        User Response: Change the length of the literal so it does not exceed the allowed maximum, or insert the missing quote, or define the literal with two statements; execute the compilation again.

IJS002I W   LITERAL CONTINUATION QUOTE INVALID IN MARGIN A.

        Explanation: The literal continuation quote should appear in margin B.

        System Action: The continuation is allowed.

IJS003I C   LITERAL IMPLY CONTINUED OR CONTINUATION QUOTE IS MISSING.

        Explanation: This may be the result of a missing quote sign on the preceding line.

        System Action: The non-numeric literal is truncated at the end of the preceding line. The syntax scan resumes with the first element of the next line.

        User Response: Check for missing quote, column 7 continuation hyphen, or improper formation of the non-numeric literal.

IJS004I   SYNTAX REQUIRES A BLANK AFTER A PERIOD OR THIS PERIOD IS INVALID DECIMAL POINT.

        System Action: The inverted print edit word with the invalid decimal point is dropped, and processing continues with the next word.

        User Response: Check syntax of statement in error, and try again.

```
IJS005I C    XXX EXCEEDS 30 CHARACTERS.

             Explanation:  Any element that is not a non-numeric literal
             is truncated after 30 characters.

             System Action:  Normal processing continues with a literal
             made up of the first 30 characters.

             User Response:  Alter the length of the literal to conform
             with the specifications for this class of literal.

IJS006I C    XXX REQUIRES QUALIFICATION.

             Explanation:  This indicates that the name is defined in
             more than one location, and requires qualification in order
             to be unique.

             System Action:  The first name defined is used and the com-
             pilation continues.  If it is the name desired, the run
             compiles as desired.  For further system action, see mes-
             sage IJS013I.  It explains the handling for the procedure
             division statement.

             User Response:  Correct the procedural statements in error,
             or change the duplicate data names so they are unique.
             Execute the job again.

IJS007I C    XXX HAS UNDEFINED QUALIFICATION.

             Explanation:  One or more of the names in the qualification
             hierarchy are not defined as a group containing the data-
             name.  This may have resulted from the dropping of a data-
             name because of an error at its point of declaration, or
             because of a misspelling.

             System Action:  The first name defined is used.  If it is
             the name desired, the run compiles as desired.

             User Response:  Check for misspelling of the data-name, or
             the data-name's qualifier in the hierarchy order.

IJS008I C    XXX REQUIRES MORE QUALIFICATION.

             Explanation:  The number of qualifiers or the names are not
             sufficient to make the subject name unique.  Another name
             could have the same qualification.

             System Action:  The first name defined is used and the com-
             pilation continues.  If it is the name desired, the run
             compiles as desired.  For further system action, see mes-
             sage IJS013I.  It explains the handling for the procedure
             division statement.

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.

             Explanation:  Normal processing continues.

IJS011I B    XXX IS UNDEFINED.

IJS012I C    XXX HAS MORE SUBSCRIPTS THAN DECLARED IN THE DATA DIVISION.

             Explanation:  The procedure division reference to the data-
             name has too many subscripts.  The number of subscripts
```

must match the number of OCCURS...DEPENDING ON clauses in
the definition hierarchy in the Data Division.

System Action: Normal processing continues with the next
word.

IJS013I C   RECORD NAME 'XXX' IS ASSOCIATED WITH INVALID FD ENTRY.

Explanation: The FD associated with the SELECT clause is
invalid.

System Action: The error attribute for the record is
generated, and normal processing continues with the next
word.

User Response: Check FD entries for proper device labels,
required clauses, missing period terminator, etc.

IJS023I C   COPY AND INCLUDE MUST NOT BE USED WITHIN LIBRARY ENTRIES.

System Action: Words following the library name are diag-
nosed according to the clause being processed, up to the
next required clause.

IJS024I C   PERIOD MISSING FOLLOWING XXX. THE NEXT CARD MAY BE
SKIPPED.

System Action: For the Data Division COPY statement -- Any
other entry following the name is diagnosed as the missing
period and the return is made to the phase. The phase dia-
gnoses all entries up to the next period according to the
current clause string. Normal processing continues.

For the Procedure Division INCLUDE statement -- Interroga-
tion of the library name continues to determine its validi-
ty and whether or not it is in the library. If the library
name is valid and it is found, normal processing continues.

User Response: A period should be inserted following
library book name.

IJS025I C   XXX IS AN INVALID LIBRARY NAME OR NOT FOUND ON LIBRARY.

Explanation: The library name may have been misspelled,
not previously cataloged or not properly terminated with a
quote.

System Action: Any word other than period immediately fol-
lowing the library name is diagnosed according to the cur-
rent clause string up to the next period. This includes
the current card and the next card if read.

User Response: Check for the possible causes given in the
explanation.

IJS026I C   FLOATING-POINTING NUMBER XXX IS BELOW OR ABOVE VALID RANGE.

System Action: The value zero is assumed.

IJS027I W   NUMBER OF DECIMALS IN LITERAL XXX AND DATA ENTRY DISAGREE.

System Action: Truncation or padding is performed accord-
ing to the rules governing the MOVE verb.

IJS028I C   LITERAL XXX IS INVALID AND IS DROPPED.

Explanation: The value clause conflicts with the descrip-
tion of the entry.

<u>System Action:</u> The value clause is dropped.

IJS029I W   LITERAL XXX AND PICTURE SIZE DISAGREE.

<u>Explanation:</u> This message indicates a literal that is larger than its picture.

<u>System Action:</u> The literal is truncated to picture size from left to right, unless right justification is specified. The scan is continued as though no error occurred.

IJS030I W   LITERAL XXX WAS SIGNED, ENTRIES PICTURE WAS UNSIGNED.

<u>Explanation:</u> 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   NUMBER OF INTEGERS IN LITERAL XXX AND DATA ENTRY DISAGREE.

<u>System Action:</u> Same as for message IJS027I.

IJS032I C   LIBRARY NAME IS AN INVALID EXTERNAL NAME OR NOT IN THE LIBRARY.

<u>Explanation:</u> The library name may have been misspelled, not cataloged, or not properly terminated with a quote.

<u>System Action:</u> The invalid or not found library name is dropped and the next card is read.

IJS041I C   THIS CLAUSE IGNORED AT THE 01 LEVEL IN XXX ENTRY.

<u>Explanation:</u> The OCCURS...DEPENDING ON clause not valid as a level 01 or level 88 entry.

<u>System Action:</u> The clause is dropped.

<u>User Response:</u> Alter the clause level number to one that is valid or remove the OCCURS...DEPENDING ON clause from the statement in error.

IJS042I C   THIS CLAUSE IGNORED IN XXX ENTRY AS IT PROVIDES MORE THAN 3 LEVELS OF SUBSCRIPTING.

IJS043I C   DEPENDING ON OPTION IN XXX ENTRY IS IGNORED DUE TO PRIOR USE.

IJS044I C   DEPENDING ON OPTION IN XXX ENTRY IS IGNORED BECAUSE IT IS SUBORDINATE TO A PREVIOUS CLAUSE.

IJS045I C   THE LEVEL OF XXX ENTRY INVALIDATES THE DEPENDING OPTION AT THE PRECEDING XXX ENTRY. THE DEPENDING OPTION IS DROPPED.

<u>Explanation:</u> The level number just encountered indicates that there was an OCCURS...DEPENDING ON clause that did not include the last entry within the level 01.

<u>System Action:</u> The OCCURS...DEPENDING ON option is dropped.

IJS046I C   XXX ENTRY CONTAINS AN ILLEGAL LEVEL NUMBER OR REDEFINES CLAUSE WHICH IS IGNORED.

<u>Explanation:</u> A redefines clause must redefine an entry at the same level number.

<u>System Action:</u> The level number or the redefines clause is ignored.

User Response: Alter the level number or relocate the
                    redefines clause to conform with the specification.

IJS047I E   INTERNAL QUALIFIER TABLE OVERFLOWED WHEN HANDLING XXX.
             RESTARTED QUALIFIERS WITH XXX.

             Explanation: The sum of all the characters in the data-
             name and all its qualifiers + 4 times (the number of quali-
             fiers + 1) must not exceed 300.

IJS048I W   ENTRY PRECEDING XXX IS OF VARIABLE LENGTH.

IJS049I W   XXX IS LARGER THAN ENTRY REDEFINED.

             Explanation: The current entry is larger than the area
             redefined.

             System Action: The area is assumed to be expanded.

             User Response: The redefined area may be expanded.

IJS050I W   XXX ENTRY PRECEDING XXX IS LARGER THAN ENTRY REDEFINED.

             Explanation: Same as for message IJS049I, only for a group
             entry.

             System Action: Same as for message IJS049I.

IJS051I W   THIS CLAUSE INVALID IN XXX ENTRY AS REDEFINED AREA IS
             SUBSCRIPTED.

             Explanation: It is invalid to redefine an area containing
             an OCCURS...DEPENDING ON clause.

             System Action: The redefinition clause is dropped.

IJS052I C   THIS CLAUSE IGNORED IN XXX ENTRY DUE TO REDEFINES OR OCCURS
             CLAUSE IN PRECEDING XXX LEVEL.

             Explanation: A value clause cannot appear in an entry sub-
             ordinate to a redefines clause.

             System Action: The value clause is dropped.

IJS053I W   FOR PROPER ALIGNMENT, A XXX BYTE LONG FILLER ENTRY IS
             INSERTED PRECEDING XXX.

             Explanation: Binary or floating-point data improperly
             aligned for computations.

             System Action: Binary and floating-point data are aligned
             on an appropriate boundary by the compiler. The alignment
             is performed by inserting an assumed filler entry preceding
             the item requiring alignment.

             User Response: The number of slack bytes required can be
             reduced by the use of a different data format such as:
             internal decimal, grouping aligned items to the beginning
             of a record, or otherwise positioning them so that they
             will have the proper alignment within the record. A dis-
             cussion of slack bytes can be found in the publication IBM
             System/360 Disk and Tape Operating System: COBOL Language
             Specifications, Form C24-3433.

IJS054I W   FOR PROPER ALIGNMENT, A XXX BYTE LONG XXX FILLER ENTRY IS
             INSERTED PRECEDING XXX.

             Explanation: Binary or floating-point data is improperly
             aligned for computations.

             System Action: Groups are aligned according to the align-
             ment requirements of the first elementary within that
             group. The level number indicated in the diagnostic mes-
             sage shows exactly where the implied filler entry was
             inserted. For further explanation, see message IJS053I.

IJS055I E   XXX ENTRY PRECEDING XXX EXCEEDS MAXIMUM SIZE OF 4092 BYTES.

             Explanation: The group defined at the indicated level pre-
             ceding the point where this message was generated exceeded
             the maximum size permitted in the file or linkage section.

             System Action: The compilation is continued, but execution
             is not attempted.

             User Response: Reduce the record size to the allowable
             maximum size.

IJS056I W   XXX ENTRY PRECEDING XXX EXCEEDS MAXIMUM LENGTH OF 32,768
             BYTES.

             Explanation: See message IJS055I. It applies to working-
             storage section.

             System Action: See message IJS055I. It applies to
             working-storage section.

IJS057I E   PROGRAM EXCEEDS 240 BASE LOCATORS MAXIMUM AT XXX.

             Explanation: A base locator is assigned for each file for
             each level 01 or level 77 in the linkage section, and for
             every 4,096 bytes in the working-storage section.

             System Action: The base locator counter wraps around and
             the results are unpredictable.

             User Response: Reduce the number of base locators.

IJS058I E   ERRONEOUS OR MISSING DATA DIVISION.

             Explanation: No data division entries were present.

             System Action: All data division entries were dropped
             because of errors.

IJS060I W   XXX LEVEL PRECEDING XXX IS OF VARIABLE LENGTH.

             Explanation: The entry, defined at the level indicated,
             that preceded this clause contained an OCCURS...DEPENDING
             ON clause.

             System Action: The redefined clause is dropped because it
             is invalid to redefine a variable-length entry.

IJS061I C   XXX ENTRY EXCEEDS MAXIMUM LENGTH FOR ITS DATA TYPE.

             Explanation: The maximum permitted length of an entry
             depends on the type of data defined for that entry. Numer-
             ic data cannot exceed 18 digit positions, report entries
             cannot exceed 127 character positions.

             System Action: The maximum size is used.

IJS062I W   XXX REQUIRED ALIGNMENT AND STARTS XXX BYTES PAST THE START
               OF THE ENTRY IT REDEFINED.

               Explanation: The entry containing the redefines clause
               requires alignment that differs from the alignment of the
               clause redefined. If alignment is required, insert a fill-
               er the size of the number of bytes indicated in the message
               before the item being redefined.

IJS063I W   TO ALIGN BLOCKED RECORDS ADD XXX BYTES TO THE 01 CONTAINING
               DATANAME XXX.

               Explanation: The first record in a buffer is aligned on a
               double-word boundary. All level 01 records 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 a
               level 01 record in the dorking-storage section before
               processing the computational field. The pointer to this
               diagnostic message indicates the last element within a
               record. The padding must go into the preceding level 01
               record, not the level 01 record that may immediately follow
               the indicated data name.

IJS064I W   IF THE PRECEDING RECORD IS BLOCKED, IT MAY BE ALIGNED BY
               MOVING TO AN 01 IN THE WORKING-STORAGE SECTION.

               Explanation: When records are variable and blocked, only
               the first record can be aligned.

IJS076I W   INTEGER OPTION IS NOT PERMITTED.

               System Action: The clause is dropped.

IJS078I C   INTERNAL FILE-NAME AND DESCRIPTION TABLE OVERFLOWED.   XXX
               NOT PROCESSED.

               Explanation: There is a fixed number of files that can be
               handled by a given COBOL compilation (25). If additional
               files must be handled, they can be processed in a subpro-
               gram and accessed via the linkage facility.

               System Action: Any files encountered after the maximum
               permitted are dropped. The maximum permitted is 25.

IJS079I C   RESTRICTED SEARCH INTEGER TOO LARGE ON XXX.   CLAUSE
               DROPPED.

IJS080I C   MORE THAN THREE FORMS OVERFLOW CLAUSES. OVERLOW-NAME XXX
               ENTRY IS DROPPED.

IJS081I W   XXX APPEARED PREVIOUSLY IN A 'SAME' CLAUSE.   REMAINDER OF
               'SAME' CLAUSE DROPPED.

               Explanation: A given filename can appear in only one same-
               area clause. Any duplication encountered is dropped.

               System Action: The entire same-area clause is dropped.

               User Response: Eliminate the duplicate statement.

IJS082I W   INTERNAL 'SAME' TABLE OVERFLOW. ENTRIES AFTER XXX DROPPED.

Explanation: A fixed number of filenames and combinations of filenames are allowed in an internal same-area table. If reducing the number of filenames or the number of same-area clauses does not relieve the situation, it may require an entry to a subprogram to permit a large number of files to be referenced in this manner.

IJS083I W   RECORD LENGTH SPECIFIED DISAGREES WITH CALCULATED MAX. RECORD LENGTH OF XXX ON XXX. CALCULATED RECORD LENGTH ASSUMED.

Explanation: The actual length of each record is calculated during compilation time by totaling all its components. If the length disagrees with the specified maximum, this warning message is given to indicate that the specified record size is ignored.

IJS084I W   BLOCK SIZE FOR XXX TOO BIG. 32K ASSUMED.

Explanation: The integer specifying block size of the referenced files is too large.

System Action: The maximum size allowed is used.

IJS085I W   SYMBOLIC KEY MUST BE SPECIFIED FOR XXX IF INPUT.

Note: This message is used only for a direct-access storage device.

IJS086I E   ACTUAL KEY MUST BE SPECIFIED FOR XXX.

Note: This message is used only for a direct-access storage device.

IJS087I C   THE XXX FILE MUST BE DESCRIBED IN A SELECT CLAUSE. CURRENT ENTRY IGNORED.

Explanation: The subject file was referenced in the Environment Division or in an FD clause. There is no select clause to define this file. The filename referenced may be an invalid entry encountered at the point that a filename was expected.

IJS088I C   LABEL RECORD DATA-NAME MUST BE DEFINED IN LINKAGE SECTION.

System Action: Label records are assumed standard.

IJS089I C   UNIT IS MISSING FOR XXX FILE. 2400 IS ASSUMED.

IJS090I C   THE DESCRIPTION OF XXX FILE CONFLICTS ON THE FOLLOWING POINTS -- XXX.

Explanation: The description of the file referenced contains factors that conflict with each other. The factors can be in the description of the file in the Environment Division, in the FD of the file section, or in other areas such as the record description for that file.

System Action: The points in conflict are defined by the trailing clauses of the diagnostic message.

IJS091I E   INDEXED ORGANIZATION ON XXX NOT VALID FOR THIS LEVEL COMPILER.

IJS092I E   DIRECT ORGANIZATION ON XXX NOT VALID FOR THIS LEVEL COMPILER.

IJS093I E   XXX NOT HANDLED WITH PRESENT RELEASE.

IJS094I E   XXX FILE WAS NOT DEFINED BY AN FD ENTRY.

Explanation: No DTF table is built for this file, there-
fore, it cannot be used.

IJS096I W   ONLY ONE CHECKPOINT FILE MAY BE SPECIFIED.

IJS097I E   STANDARD LABELS ARE REQUIRED ON XXX FILE.

IJS098I C   XXX FILE ASSUMED TO BE UTILITY.

IJS099I C   XXX FILE UNIT MISSING AND ASSUMED TO BE 1403 PRINTER.

IJS100I E   DIRECT-ACCESS ASSIGNED TO XXX NOT SUPPORTED IN THIS
VERSION.

IJS101I C   XXX FILE IS ASSIGNED TO UNIT RECORD AND MUST BE RECORDING
MODE IS F.

Explanation: Unit record must be fixed length.

System Action: The largest described length is assumed.

IJS102I C   A MAXIMUM OF 1 ALTERNATE AREA IS ALLOWED FOR XXX FILE.

System Action: One alternate area is reserved.

IJS103I E   XXX IS NOT A VALID SYSTEM ASSIGNMENT.

Explanation: Must be SYS000 to SYS244.

System Action: SYS000 is assumed.

IJS104I E   RECORD/BLOCK SIZE ON XXX GREATER THAN 3625.

IJS105I C   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 INDEX ORGANIZATION FILE XXX.

IJS108I E   LENGTH OF SYMBOLIC/RECORD KEY GREATER THAN 255.

IJS109I E   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   RELATIVE ORGANIZATION ASSIGNED TO XXX NOT SUPPORTED IN THIS
VERSION. COMPLETE SELECT STATEMENT DROPPED.

IJS114I E   RECORD/BLOCK ON XXX IS GREATER THAN 2000.

IJS117I E   SYMBOLIC KEY MUST BE SPECIFIED FOR XXX.

IJS118I W   DIRECT ACTUAL KEY MUST BE SPECIFIED FOR OUTPUT FILES.

IJS176I C   WORD RECORD OR RECORDS IS REQUIRED. FOUND 'XXX'.

Explanation: Syntax skips until the next clause, level
number, or period at the end of the file description is
encountered.

| IJS177I W | PERIOD REQUIRED AFTER WORD 'SECTION'. |

| IJS179I W | 'XXX' IS AN INVALID FILE-NAME FORMAT. |

Explanation: A filename must follow the format rules for data-names.

System Action: Invalid names are truncated to 30 characters and treated as valid names.

| IJS180I E | XXX EXCEEDS 30 CHARACTERS AND IS DROPPED. |

System Action: The picture is too long and is dropped.

| IJS181I W | THE OPTION WORD IS MISSPELLED OR OMITTED. FOUND XXX. |

System Action: The usage assumed is DISPLAY.

| IJS183I C | 'XXX' IS AN INVALID OR EXCESSIVE INTEGER. |

Explanation: The integer indicated in this clause is determined to be invalid.

System Action: The integer is not used.

| IJS184I W | XXX IS AN INVALID LEVEL NUMBER. |

| IJS185I W | LABEL RECORDS IS OMITTED. LABELS ASSUMED STANDARD. |

| IJS186I W | SYNTAX REQUIRES DATA RECORD CLAUSE. |

System Action: Syntax scanning proceeds.

| IJS187I C | MODE MUST BE 'V', 'F', OR 'U'. FOUND XXX. |

User Response: If V, F, or U was specified, check the element number on this line for a misspelled optional word.

| IJS190I W | 'XXX' IS AN INVALID DATA-NAME FORMAT. |

System Action: The invalid data-name(s) are truncated to 30 characters and used.

| IJS191I W | SD OR SA ENTRY REQUIRES F LEVEL COMPILER. |

System Action: Syntax skips to next margin A entry.

| IJS192I W | 'XXX' IS AN INVALID RECORD-NAME FORMAT. |

System Action: Invalid record names are truncated to 30 characters and treated as valid names.

| IJS194I C | 'XXX' IS INVALID AT THIS POINT. CHECK FOR SYNTAX ERROR ON CURRENT/PREVIOUS STATEMENT. |

Explanation: While processing a given clause or sentence, an unexpected element was encountered. The clause may be valid but misplaced. This message is also given for clauses that are not valid source input to this level compiler.

User Response: Check for prior diagnostic messages, an extra or missing period, invalid continuation of non-numeric literals, or a misspelled word.

| 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.

             System Action: Invalid data-names are truncated to 30
             characters and treated as valid names.

IJS202I C   Same as diagnostic message IJS194I.

IJS203I C   THIS USAGE XXX CONFLICTS WITH THE GROUP USAGE AND IS
             IGNORED.

IJS204I C   XXX IS AN INVALID OR EXCESSIVE INTEGER.

             System Action: The invalid integer is dropped.

IJS205I W   XXX IS AN INVALID DATA-NAME FORMAT, BUT ASSUMED VALID.

IJS206I W   WORD ZERO IS REQUIRED. FOUND XXX.

             System Action: The clause is ignored.

IJS207I W   WORD RIGHT IS REQUIRED. FOUND XXX.

             System Action: The clause is ignored.

IJS210I C   THIS ENTRY CONFLICTS WITH THE FOLLOWING
             DESCRIPTIONS --- XXX.

             Explanation: Various clauses specified for a data entry
             are compared with previous specifications for the entry.
             If there is any factor that conflicts with the subject
             clause, it is listed as a trailer to this entry. Factors
             included that are not themselves clauses would be elemen-
             tary or group item usage, specified at a group level in
             previous clauses. This message can appear if a period is
             missing at the end of a data entry or (for example) when
             the PICTURE clause for the second entry is encountered and
             automatically conflicts with the PICTURE clause for the
             previous entry.

IJS211I C   XXX EXCEEDS 30 CHARACTERS AND IS TRUNCATED.

IJS212I C   ONLY LEVELS 77 OR 01 ARE PERMITTED AT THIS POINT. FOUND
             XXX.

             System Action: Syntax skips until a section name or level
             number is found.

IJS213I W   THE FOLLOWING DESCRIPTIONS INVALID AT GROUP LEVEL --- XXX.

             Explanation: The data entry described is determined to be
             a group, although the entries specified as trailers to this
             message are invalid at the group level. This diagnostic
             message can be produced by an invalid level number that was
             changed to a level 01, or because of 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 diag-
             nostic message.

IJS214I C   XXX DATA ENTRY REQUIRES A PICTURE, COMPUTATIONAL-1 OR
             COMPUTATIONAL-2.

             Explanation: This diagnostic message can be produced by an
             error in the following level number that caused its level
             to be changed to a level 01, thereby making this entry an
             elementary.

System Action: Any statement in the Procedure Division containing a reference to this entry is diagnosed and dropped.

User Response: Check for missing periods or other diagnostic messages.

IJS215I W SYNTAX REQUIRES AN ENTRY IN MARGIN A. FOUND XXX IN MARGIN B.

System Action: Following certain entries in a source program, a specific clause must be encountered in margin A. If it is found in margin B, it is diagnosed but handled by the compiler.

IJS216I W SYNTAX REQUIRES AN ENTRY IN MARGIN B. FOUND XXX IN MARGIN A CHECK FOR MISSING PERIOD.

Explanation: All entries in margin A must be preceded by a period.

System Action: The compiler was in the middle of processing a clause or sentence and encountered the indicated word in margin A. Thus, a diagnostic message is issued and the word is processed as though it were 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. CHANGED XXX TO 01.

IJS221I C SYNTAX FOR 'ALL' REQUIRES 'XXX' BE A SINGLE CHARACTER IN QUOTES.

System Action: The value clause is dropped.

IJS222I C PICTURE XXX WAS FOUND INVALID WHILE PROCESSING XXX. THE PICTURE IS DROPPED.

Explanation: Any element that follows the word PICTURE in a data description, other than the word that is dropped, is assumed to be a PICTURE and is passed to a later phase for analysis. The analysis proceeds from left to right on a character-by-character basis. The character identified in the message is the one processed at the time the PICTURE is determined to be invalid. The specific character itself may be invalid or may have indicated that a previous character or condition is invalid. For example, an E encountered in an external floating-point PICTURE may indicate that a preceding decimal was omitted in the mantissa.

System Action: The PICTURE is dropped and the entry identified as an error.

IJS277I W FILE SECTION OUT OF SEQUENCE.

IJS228I E SYNTAX PERMITS ONLY ONE XXX IN SOURCE PROGRAM.

System Action: 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.

IJS237I E   'XXX' IS MISPLACED.

         Explanation: The statement is probably out of place in the source deck; that is, FD is working-storage.

         System Action: The statement is processed as it is; however, execution may not be as desired.

         User Response: Properly locate the misplaced statement.

IJS238I W   'XXX' IS AN INVALID SECTION NAME, A MISSING FD OR AN INVALID/MISPLACED LEVEL INDICATOR.

         System Action: Syntax skips until a valid section-name or level number is found.

IJS239I W   SYNTAX REQUIRES WORD 'DIVISION'.

IJS240I C   'UNIT' OR 'REEL' CANNOT BE SPECIFIED FOR UNIT-RECORD FILE.

IJS241I C   LEVEL PRECEDING 88 MUST BE AN ELEMENTARY.

         Explanation: Any level number preceding a level 88 entry must be an elementary.

         System Action: If the level number preceding the level 88 entry is not an elementary, it is assumed to be one and is processed as such.

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'.

         Explanation: 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.

         System Action: Syntax scan skips to the restart clause.

IJS303I W   'XXX' IS AN INVALID CONDITION-NAME FORMAT.

         Explanation: The name shown is an invalid condition name.

         System Action: The name is truncated to 30 characters and processed as though it were valid.

IJS304I E   'XXX' IS AN INVALID EXTERNAL-NAME FORMAT.   RESTART WITH 'XXX'.

         Explanation: An external name was expected at this point in the scan of the subject clause.   An external name must be enclosed in quotes.   It must start with an alphabetic character, cannot contain more than eight characters, and letters and numerals are the only valid characters.   A dash is not permitted.

IJS305I C   SYNTAX REQUIRES SAME, RERUN, APPLY, OR 'XXX' DIVISION.
             FOUND 'XXX'. RESTART WITH 'XXX'.

             User Response: Check for invalid sequence of source pro-
             gram cards or extra periods.

IJS306I W   SYNTAX REQUIRES ENVIRONMENT OR 'XXX' DIVISION IN MARGIN A.
             FOUND 'XXX'. RESTART WITH 'XXX'.

             User Response: Same as for message IJS305I.

IJS307I E   SYNTAX REQUIRES I-O-CONTROL, INPUT-OUTPUT, OR 'XXX' DIVI-
             SION IN MARGIN A. FOUND 'XXX'. RESTART WITH 'XXX'.

             User Response: Same as for message IJS305I.

IJS308I W   'XXX' IS AN INVALID DATA-NAME FORMAT. RESTART WITH 'XXX'.

             Explanation: A data-name was expected at this point in the
             scan of the subject clause.

             System Action: Invalid format is truncated to 30 charac-
             ters and processed as though it were valid.

IJS309I C   ENVIRONMENT PARAGRAPHS OUT OF ORDER.

             System Action: Statements are handled anyway.

IJS310I W   'XXX' IS AN INVALID 360 MODEL-NUMBER. RESTART WITH 'XXX'.

             System Action: Syntax scan skips to the restart clause.

IJS311I E   SYNTAX REQUIRES 'FILE-CONTROL', 'XXX' OR 'DATA DIVISION' IN
             MARGIN A. FOUND 'XXX'. RESTART WITH 'XXX'.

             User Response: Same as for message IJS305I.

IJS312I C   'XXX' IS AN INVALID OR EXCESSIVE INTEGER. RESTART WITH
             'XXX'.

             Explanation: The syntax at this point of scan of the spe-
             cified clause requires an integer.

             System Action: The element found was invalid and is
             dropped.

IJS313I W   'XXX' IS AN INVALID FILE-NAME FORMAT. RESTART WITH 'XXX'.

             Explanation: The syntax scan of the subject clause
             requires a file name at this point.

             System Action: The element found was invalid. It was
             truncated to 30 characters and processed as though it were
             valid.

IJS314I E   'XXX' IS AN INVALID LIBRARY-NAME FORMAT. RESTART WITH
             'XXX'.

             Explanation: A library name is required at this point.

             System Action: The format is invalid. It is dropped.

IJS315I W   MORE THAN THREE OVERFLOW OPTION CLAUSES ARE USED.

             Explanation: An internal table permits a maximum of three
             form overflow names to be assigned in any compilation.

System Action: All form overflow names in excess of the maximum allowed (three) are dropped.

IJS316I C   SYNTAX REQUIRES 'INDEXED' OR 'XXX'. FOUND 'XXX'. RESTART WITH 'XXX'.

Explanation: This message applies to a direct access storage device only.

IJS317I C   SYNTAX REQUIRES 'SEQUENTIAL' OR 'XXX'. FOUND 'XXX'. RESTART WITH 'XXX'.

Explanation: This message applies to a direct access storage device only.

IJS318I W   SYNTAX REQUIRES 'XXX' OR DATA DIVISION IN MARGIN A, OR SELECT IN MARGIN B. FOUND 'XXX'. RESTART WITH 'XXX'.

Explanation: The syntax for the specific clause requires specific entries at this point.

User Response: Check for misspelled words, or excessive periods.

IJS319I C   SYNTAX REQUIRES 'UTILITY', 'DIRECT-ACCESS' OR 'XXX'. FOUND 'XXX'. RESTART WITH 'XXX'.

Explanation: Same as for message IJS316I.

IJS320I W   'XXX' IS AN INVALID I-O-DEVICE-NUMBER. RESTART WITH 'XXX'.

Explanation: Same as for message IJS316I.

IJS321I E   NO PROCESSING OF THIS MULTIPLE SPECIFIED DIVISION OR SECTION. RESTART WITH 'XXX'.

Explanation: A section or division was encountered more than once.

System Action: The additional section or division is dropped, rather than disturb the internal sequence of the compilation.

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'.

System Action: Syntax scan skips to the restart clause.

IJS324I E   SYNTAX REQUIRES 'REEL' OR 'XXX'. FOUND 'XXX'. RESTART WITH 'XXX'.

System Action: Syntax scan skips to restart clause.

IJS401I C   SYNTAX REQUIRES A DATA-NAME. FOUND 'XXX'.

Explanation: The syntax of the indicated clause requires a data-name. The element found was not defined as a valid data-name. The element may be indicated here, or an indication given that it was an invalid name such as, file name, condition name, figure configuration, or overflow name.

System Action: The compilation continues at the next verb or paragraph label.

User Response: Check for misspelled data-name in diagnostic messages, 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'.

Explanation: The syntax for this clause requires a specific word that was not found. The item encountered was probably a data-name. The next item indicates that the syntax requires a specific word or words. None were found.

System Action: The element found is displayed unless it was a name, in which case the word invalid name or data name is indicated. Compilation continues at the next verb or paragraph label.

User Response: The reference format for the clause specified should be consulted if the meaning of the message is not immediately clear. Also check for: missing periods, preceding diagnostic messages, invalid non-numeric literals, and COBOL words used as data-names.

IJS403I C   SYNTAX REQUIRES A DATA-NAME OR NUMERIC-LITERAL. FOUND 'XXX'.

Explanation: See message IJS402I.

IJS404I C   SYNTAX REQUIRES EITHER WORD 'TO', OR 'GIVING'. FOUND 'XXX'.

Explanation: See message IJS402I.

IJS405I C   SYNTAX REQUIRES A SINGLE CHARACTER IN QUOTES OR A FIGCON. FOUND 'XXX'.

Explanation: See message IJS402I.

IJS406I C   SYNTAX REQUIRES A FILE-NAME. FOUND 'XXX'.

Explanation: See message IJS402I.

IJS407I C   SYNTAX REQUIRES DATA-NAME OR INTEGER. FOUND 'XXX'.

Explanation: See message IJS402I.

IJS408I C   SYNTAX REQUIRES WORD 'INPUT', 'OUTPUT', OR 'I-3'. FOUND 'XXX'.

Explanation: See message IJS402I.

IJS409I C   SYNTAX REQUIRES A PROCEDURE-NAME. FOUND 'XXX'.

Explanation: See message IJS402I.

IJS410I C   SYNTAX REQUIRES A DATA-NAME OR LITERAL. FOUND 'XXX'.

Explanation: See message IJS402I.

IJS411I C   SYNTAX REQUIRES WORD 'CALL', 'ENTRY', OR 'RETURN'. FOUND 'XXX'.

Explanation: See message IJS402I.

IJS412I E   SYNTAX REQUIRES AN EXTERNAL-NAME. FOUND 'XXX'.

Explanation: See message IJS402I.

IJS413I C   SYNTAX REQUIRES '='. FOUND 'XXX'.

             Explanation: See message IJS402I.

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.

             Explanation: See message IJS402I.

IJS415I C   SYNTAX REQUIRES CALL PARAMETERS TO BE EITHER DATA-NAME,
             PROCEDURE-NAME OR FILE-NAME. FOUND 'XXX'.

             Explanation: See message IJS402I.

IJS416I C   SYNTAX REQUIRES DATA-NAME, LITERAL, FIGCON, '+', '-', '('
             OR 'NOT'. FOUND 'XXX'.

             Explanation: See message IJS402I.

IJS417I C   SYNTAX REQUIRES ARITHMETIC OPERATOR OR RELATIONAL. FOUND
             'XXX'.

             Explanation: See message IJS402I.

IJS418I C   SYNTAX REQUIRES A DATA-NAME, NUMERIC-LITERAL, OR '(' AFTER
             AN OPERATOR. FOUND 'XXX'.

             Explanation: See message IJS402I.

IJS419I C   SYNTAX REQUIRES A DATA-NAME, LITERAL, FIGCON, '(', '+' OR
             '-' AFTER A RELATIONAL. FOUND 'XXX'.

             Explanation: See message IJS402I.

IJS420I C   SYNTAX REQUIRES A VERB, PERIOD, 'ELSE' OR 'OTHERWISE'.
             FOUND 'XXX'.

             Explanation: The end of a valid clause was encountered.
             The element that followed the valid termination of this
             clause is not valid.

             System Action: Compilation continues at the next verb or
             paragraph label.

             User Response: If the preceding clause had some options,
             check the reference format to determine whether or not the
             options were specified correctly. A COBOL word used as a
             data-name or an extra period can also produce this diag-
             nostic message.

IJS421I C   ENTRY PARAMETER MUST BE A DATA-NAME. FOUND 'XXX'.

             Explanation: The only parameters that can be passed to a
             COBOL subprogram are data-names. The data-names must be
             defined in the linkage section of the subprogram.

             System Action: Compilation continues at the next verb or
             paragraph label.

IJS422I C   SYNTAX REQUIRES A RELATIONAL. FOUND XXX.

             Explanation: Syntax requires that the next element be a
             relational.

             System Action: Compilation continues at the next verb or
             paragraph label.

IJS423I C   SYNTAX REQUIRES WORD 'INPUT' OR 'OUTPUT'. FOUND XXX.

Explanation: See message IJS402I.

IJS424I C   SYNTAX REQUIRES WORDS 'TO PROCEED TO'. FOUND XXX.

Explanation: See message IJS402I.

IJS425I C   SYNTAX REQUIRES WORD 'CONSOLE' OR 'SYSPCH'. FOUND XXX.

Explanation: See message IJS402I.

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.

Explanation: See message IJS402I.

IJS428I C   SYNTAX REQUIRES A PROCEDURE-NAME AFTER 'GO TO' NOT PRECEDED
            BY A PARAGRAPH-NAME. FOUND XXX.

Explanation: See message IJS402I.

IJS429I C   SYNTAX REQUIRES 'ALL', 'LEADING', 'UNTIL', OR 'FIRST'.
            FOUND XXX.

Explanation: See message IJS402I.

IJS430I C   SYNTAX REQUIRES WORD 'TALLYING' OR 'REPLACING'. FOUND XXX.

Explanation: See message IJS402I.

IJS431I C   SYNTAX REQUIRES WORD 'DEPENDING ON'. FOUND XXX.

Explanation: See message IJS402I.

IJS432I C   DATA TYPE MUST BE ED, ID OR BI.

Explanation: Valid syntax for the subject verb permits
only specific data types. The data type as determined by
the definition in the Data Division is invalid for its use
here.

System Action: The statement is dropped from the point of
error.

IJS433I C   SYNTAX REQUIRES WORD 'TRACE'. FOUND XXX.

Explanation: See message IJS402I.*

IJS434I C   SYNTAX REQUIRES THAT A PERIOD OR SECTION FOLLOWS PARAGRAPH-
            NAME. FOUND XXX.

Explanation: See message IJS402I.*

IJS435I E   DATA NAME AND ANY QUALIFIER MUST APPEAR WITHIN THE FIRST
            SEVEN OPERANDS OF STATEMENT FOR CHANGED OPTION.

Explanation: See message IJS402I.

----
*Also, the entire statement from the point of error is dropped and is
not compiled.

IJS436I C   SYNTAX REQUIRES A DATA-NAME, FIGCON OR LITERAL. FOUND XXX.

             Explanation: See message IJS402I.*

IJS437I C   SYNTAX REQUIRES A FIGCON. FOUND XXX.

             Explanation: See message IJS402I.*

IJS438I C   SYNTAX REQUIRES DATA-ITEM TO BE NO LONGER THAN FOUR.

             Explanation: See message IJS402I.*

IJS439I C   WRONG SUBSCRIPT SPECIFICATION.

             Explanation: Data names and condition names can be sub-
             scripted to a depth of three. A subscript is required for
             each OCCURS...DEPENDING ON clause specified at the speci-
             fied data-name or in groups containing that data-name.

             System Action: The compilation continues at the next verb
             or paragraph label.

             User Response: Check for fewer or more subscripts than
             OCCURS...DEPENDING ON 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.

             Explanation: See message IJS402I.

IJS441I C   SYNTAX REQUIRES AN INTEGER NOT LONGER THAN 5. FOUND XXX.

             Explanation: The integer exceeds the size permitted by
             language specifications.

             System Action: The compilation continues at the next verb
             or paragraph label.

IJS442I C   THE DECLARATION OF THIS DATA-NAME CAUSED IT TO BE FLAGGED
             AS AN ERROR.

             Explanation: The data-name encountered was flagged by the
             data division as containing an error in its declaration.

             System Action: Compilation continues at the next verb or
             paragraph label.

             User Response: Correct the declaration as indicated by the
             data division diagnostics and recompile.

IJS443I E   SYNTAX REQUIRES A VERB. FOUND XXX.

             Explanation: A point was reached where a verb was required
             and was missing. For example, 'IF = B.' requires a verb
             between the B and the period.

             System Action: The statement is skipped from the point of
             the error.

IJS444I E   SYNTAX REQUIRES A RECORD NAME. FOUND XXX.

             Explanation: See message IJS402I.

----
*Also, the entire statement from the point of error is dropped and is
not compiled.

IJS500I W   AN OPERAND'S LENGTH EXCEEDS AND IS TRUNCATED TO 256 BYTES.

             Explanation: The maximum number of bytes that can be dis-
             played is 256.

             System Action: The operand is truncated to 256 bytes and
             displayed.

IJS501I W   IF THIS VARIABLE-LENGTH ENTRY EXCEEDS 256, RESULTS WILL BE
             UNPREDICTABLE.

             Explanation: A maximum of 256 bytes can be displayed.

             System Action: The entry is truncated to 256 bytes and
             displayed.

IJS502I W   LITERAL EXCEEDS AND IS TRUNCATED TO 72 BYTES.

             System Action: In a stop-literal statement only the first
             72 bytes of a longer field are typed on the console.

IJS503I W   DATA EXCEEDS AND IS TRUNCATED TO 72 BYTES.

             Explanation: A maximum of one line (72 bytes) can be
             retrieved using the ACCEPT FROM CONSOLE statement.

IJS504I W   DATA EXCEEDS AND IS TRUNCATED TO 256 BYTES.

             Explanation: A maximum of 256 bytes can be accepted from
             SYSIPT.

IJS505I C   FILENAMES OR STERLING-DATA TYPE NOT ALLOWED IN COMPARE.

             Explanation: See message IJS506I.

IJS506I C   USAGE OF DATA-TYPES CONFLICT. THE TEST DROPPED.

             Explanation: Only certain data types can be compared to
             each other. The types specified are invalid. Reference
             can be made to the compared table to determine the valid
             combinations. Logical comparisons of fields that are
             classified as invalid comparisons 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.

             System Action: Compilation continues normally.

IJS508I E   THE STATEMENT CONTAINS AN UNDEFINED DATA NAME.

             Explanation: See message IJS402I.

IJS509I C   AN ALPHABETIC DATA-NAME CAN BE TESTED ONLY FOR ALPHABETIC
             OR NOT ALPHABETIC, AND NUMERIC DATA-NAME ONLY FOR NUMERIC
             OR NOT NUMERIC. THE TEST IS DROPPED.

IJS510I C   COMPARISON OF TWO LITERALS OR FIGCONS IS INVALID.

             Explanation: See message IJS506I.

IJS511I C   DATA-TYPE IN ARITHMETIC STATEMENT IS NOT NUMERIC OR RECEIV-
             ING FIELD IS NOT NUMERIC OR REPORT.

             Explanation: See message IJS506I.

IJS512I C   DATA-NAME IN CLASS-TEST MUST BE AN, ED, OR ID.

        Explanation: See message IJS506I.

IJS513I C   DATA-NAME IN SIGN-TEST MUST BE NUMERIC.

        Explanation: See message IJS506I.

IJS514I W   DATA EXCEEDS AND IS TRUNCATED TO 72 BYTES.

        System Action: If the data is longer than 72 bytes, only
        the first 72 bytes are printed for a DISPLAY ON CONSOLE
        statement.

IJS515I W   DATA EXCEEDS AND IS TRUNCATED TO 120 BYTES.

        System Action: If the data is longer than 120 bytes, only
        the first 120 bytes are printed for a DISPLAY statement.

IJS516I C   OPEN 'NO REWIND' OR 'REVERSED' CANNOT BE SPECIFIED FOR A
        UNIT RECORD, DIRECT-ACCESS OR DISK/DATA CELL UTILITY FILE.

        System Action: The options are ignored.

IJS517I C   'NO REWIND' OR 'LOCK' CANNOT BE SPECIFIED FOR A UNIT
        RECORD, DIRECT-ACCESS OR DISK/DATA CELL UTILITY FILE.

        System Action: The options are ignored.

IJS518I E   MORE THAN FORTY 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.

        Explanation: See message IJS506I.

IJS520I C   SYNTAX ALLOWS SPACE OR ALL AS ONLY VALID FIGCONS IN COM-
        PARISON WITH AN ALPHABETIC FIELD.

        Explanation: See message IJS506I.

IJS521I C   DATATYPE MUST BE ED, EF, AL, AN OR GF. FOUND XXX.

        Explanation: The data types indicated are the only valid
        ones that can be used in the clause indicated.

        System Action: Compilation continues at the next verb or
        paragraph label.

IJS522I C   SYNTAX REQUIRES WORD RUN OR LITERAL. FOUND XXX.

        System Action: The syntax scan skips the rest of the
        statement.

IJS523I C   RECEIVING FIELDS IN PRECEDING STATEMENT IS A LITERAL.

        Explanation: A Procedure Division literal cannot be
        changed as the result of arithmetic or a move. The state-
        ment, SUBTRACK data name FROM literal, would specify
        invalid action of this type.

        System Action: Compilation continues at the next verb or
        paragraph label.

IJS524I C   SYNTAX REQUIRES AT LEAST TWO OPERANDS BEFORE GIVING OPTION.

        Explanation: For example, ADD A GIVING B.

IJS525I C   THE EXPRESSION HAS MORE RIGHT PARENS THAN LEFT PARENS TO
             THIS POINT. FOUND XXX.

Explanation: The number of right parentheses and left
parentheses in a statement must agree. At no point in time
can there be more right parentheses than left parentheses.

System Action: The statement is skipped from the point of
the error.

User Response: Check for extra periods or missing periods,
an error in a non-numeric literal, 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.

Explanation: See message IJS525I.

IJS527I C   DATA-TYPE MUST BE ED, ID, OR BI. FOUND XXX.

System Action: The statement is skipped from the point of
error.

IJS528I C   VARYING OPTION EXCEEDS THREE LEVELS.

Explanation: A maximum of three levels is permitted with
the varying option of the PERFORM verb.

System Action: The statement is dropped from the point of
error.

IJS529I C   DATA-TYPE MUST BE ED, ID, BI, EE, OR IF.

Explanation: The data types shown are the only valid ones.
The data-name found is not one of these types.

System Action: The statement is skipped from the point of
error.

IJS530I C   NUMBER OF ELSES EXCEEDS NUMBER OF IFS.

Explanation: Number of ELSE clauses must balance out with
the appropriate number of ELSE or OTHERWISE clauses.

System Action: Statement is skipped from the point of
error.

User Response: Recount and make corrections.

IJS531I E   INTERNAL OCCURS-DEPENDING-ON TABLE OVERFLOWED AVAILABLE
             CORE.

IJS532I E   STATEMENT HAS TOO MANY OPERANDS.

Explanation: The statement referenced is too large or com-
plex for the internal tables needed for compilation.

System Action: The statement is skipped from the occur-
rence of this condition.

User Response: The statement should be divided into more
than one statement.

IJS533I E   PARENTHESIZING REQUIRES SAVING TOO MANY OPERANDS.

             Explanation: See message IJS532I.

IJS534I E   PARENTHESIZING REQUIRES SAVING TOO MANY INTERNALLY
             GENERATED LABELS.

             Explanation: See message IJS532I.

IJS535I E   PARENTHESIZING REQUIRES SAVING TOO MUCH OF STATEMENT.

             Explanation: See message IJS532I.

IJS536I E   ARITHMETIC EXPRESSION REQUIRES MORE THAN 9 INTERMEDIATE
             RESULT FIELDS.

             Explanation: See message IJS532I.

IJS538I W   'OUTPUT' CANNOT BE SPECIFIED FOR INDEX/DIRECT ORGANIZATION
             WITH RANDOM ACCESS. 'I-O' IS ASSUMED.

IJS539I C   'I-O' IS AN INVALID SPECIFICATION FOR DIRECT ORGANIZATION
             WITH SEQUENTIAL ACCESS.

IJS540I W   'NO REWIND' CANNOT BE SPECIFIED WITH 'REVERSED' OPTION.
             'REVERSED' IS ASSUMED.

IJS549I E   WORD XXX WAS EITHER INVALID OR SKIPPED DUE TO ANOTHER
             DIAGNOSTIC.

             Explanation: The majority of these messages will probably
             be caused by words skipped because of another diagnostic
             message that occurred earlier in the statement. This diag-
             nostic message also occurs because of misspelled words.

             User Response: In the case of words skipped, correct the
             previous error, or correct the current misspellings.

IJS550I C   A FIGURATIVE CONSTANT IS NOT ALLOWED AS A CALL OR ENTRY
             PARAMETER.

             System Action: The statement is skipped from the point of
             error.

IJS551I C   SYNTAX REQUIRES WORD 'TO'. FOUND XXX.

             System Action: Syntax scan skips the rest of the
             statement.

IJS552I C   RECEIVING FIELD MUST BE A DATA-NAME. FOUND XXX.

             System Action: The statement is skipped from the point of
             error.

IJS553I E   FIGURATIVE CONSTANT IS NOT ALLOWED AS A RECEIVING FIELD.

             System Action: The statement is skipped from the point of
             the error.

IJS554I C   THE 'XXX' DATA-TYPE IS NOT LEGAL RECEIVING FIELD.

             System Action: The statement is skipped from the point of
             the error.

             User Response: Check the table of permissible moves in the
             COBOL specification.

IJS555I C   OVERFLOW NAME IS NOT A VALID SENDING FIELD.

               System Action: The statement is skipped from the point of
               the error.

IJS556I E   END DECLARATIVES IS MISSING FROM PROGRAM.

               Explanation: The entire Procedure Division is treated as a
               declarative section.

IJS557I W   FLOATING-POINT CONVERSION MAY RESULT IN TRUNCATION.

               Explanation: Conversion of floating-point numbers can
               result in truncation of low-order digits.

IJS558I E   I-O OPTION FOR FILE CONFLICTS WITH NO REWIND.

               System Action: The statement is skipped from the point of
               the error.

IJS559I E   OUTPUT OPTION FOR FILE CONFLICTS WITH REVERSED.

               Explanation: The OUTPUT option conflicts with an opening
               of a file that has a reversed option specified.

               System Action: The statement is skipped from the point of
               the error.

IJS560I C   SYNTAX REQUIRES WORD 'NAMED', 'CHANGED', OR 'CHANGED
               NAMED'.   FOUND XXX.

               System Action: 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.

               Explanation: A file-name, condition name, figure config-
               uration, or variable-length group is not valid at this
               point.

               System Action: The statement is skipped from the point of
               the error.

IJS562I C   DATA ENTRY MUST NOT EXCEED 120 CHARACTERS.

               Explanation: The data entry specified exceeds the maximum
               permitted for this type of output.

               System Action: The statement is skipped from the point of
               the error.

IJS563I C   DATA ENTRY MUST BE DISPLAY.

               System Action: The statement is skipped from the point of
               the error.

IJS564I C   SYNTAX REQUIRES ONE OF THE ALLOWABLE CHARACTERS.   FOUND
               XXX.

               System Action: The statement is skipped from the point of
               the error.

IJS565I C   IF STATEMENT MUST BE TERMINATED BY A PERIOD.

               Explanation: This message is obtained when the IF state-
               ment is the last statement of a paragraph and a label is
               detected instead of a period.

```
              System Action:  The statement is skipped from the point of
              error.

IJS566I C     DATA TYPE MUST BE AL, AN, RP, OR GROUP.


              System Action:  The statement is skipped from the point of
              error.

IJS567I C     DATA TYPE MUST BE AL, AN, FIGCON OR FIXED-LENGTH GROUP.


              System Action:  The statement is skipped from the point of
              the error.

IJS568I C     DATA ITEM MUST NOT EXCEED 256 CHARACTERS.


              System Action:  The statement is skipped from the point of
              the error.

IJS569I C     DATA ENTRIES MUST BE OF EQUAL LENGTH.

              System Action:  The statement is skipped from the point of
              the error.

IJS570I C     THE LENGTH OF THE SECOND OPERAND MUST BE EQUAL TO THE FIRST
              OR A SINGLE CHARACTER.

              System Action:  The statement is skipped from the point of
              the error.

IJS571I E     A RECORD NAME MUST BE ASSOCIATED WITH THIS FILE.   FOUND
              XXX.

              System Action:  The statement is skipped from the point of
              the error.

IJS572I C     ONLY ONE DATA-NAME MAY BE ASSOCIATED WITH THE CHANGED
              OPTION.

              System Action:  The statement is skipped from the point of
              the error.

IJS573I C     DATA TYPE MUST BE ED, ID, BI, EF, IF, SN, SR, RP, AL, AN,
              FC, OR GROUP.

              System Action:  The statement is skipped from the point of
              error.

IJS601I W     NO SIGNIFICANT POSITION MATCHES BETWEEN SENDING AND RECEIV-
              ING FIELDS IN MOVE.   RECEIVING FIELD IS SET TO ZERO.

              Explanation:  There are no digit positions in common
              between the sending and receiving fields.  This can be
              illustrated by moving a field with PICTURE 99 to a receiv-
              ing field with PICTURE V99.

              System Action:  The receiving field is set to zero.

IJS602I W     DESTINATION FIELD DOES NOT ACCEPT THE WHOLE SENDING FIELD
              IN MOVE.

              Explanation:  The sending field is larger than the receiv-
              ing field in either its integer or decimal positions, or
              both.

              System Action:  The sending field is truncated.
```

IJS603I C   AFTER ADVANCING OPTION NOT ALLOWED WITH REWRITE.

             System Action: The statement is skipped from the point of
             the error.

IJS604I E   SOURCE PROGRAM EXCEEDS INTERNAL LIMITS.

             Explanation: The program is too large.

             User Response: The user should do one of the following,
             then retry:

               • Divide the program into two or more parts

               • Simplify compound conditional statements.

IJS605I E   PROCEDURE NAME MULTIPLY DEFINED.

             Explanation: 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.

             Explanation: The name indicated was incorporated into a GO
             TO or a PERFORM statement, and was never defined. Proce-
             dure names must begin in columns 8 through 11 at the point
             where they are defined.

IJS607I E   INVALID LITERAL XXX.

             User Response: Check for multiple decimal points, non-
             numeric characters that have not been enclosed in quotes.

IJS608I E   XXX IS NOT ALLOWED TO HANDLE MORE THAN 25 FILES IN ONE
             STATEMENT.

             System Action: The rest of the statement is skipped. Only
             25 files are handled.

IJS609I E   PROCEDURE-NAME XXX HAS ILLEGAL CONTENT AND IS DROPPED.

IJS610I E   'CONDITION NAME' 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.

             System Action: Handles multiple OPEN statements each con-
             taining nine filenames.

IJS613I W   USING STATEMENT HAS BEEN INCORRECTLY SPECIFIED.

IJS614I E   THIS CONDITIONAL HAS A MISSING RELATIONAL OPERATOR.

             System Action: The statement is skipped from the point of
             the error.

IJS615I E   READ 'AT END' REQUIRED FOR FILES WITH ACCESS SEQUENTIAL.

IJS617I E   WRITE 'FROM' REQUIRED WITH APPLY WRITE ONLY.

IJS618I E   REWRITE INVALID ON DIRECT OR RELATIVE SEQUENTIAL FILES.

IJS621I E   OPEN 'I-O' INVALID FOR DIRECT OR RELATIVE SEQUENTIAL FILES.

```
IJS622I E    OPEN 'OUTPUT' INVALID FOR FILES WITH ACCESS RANDOM, I-O
             ASSUMED.

IJS623I E    OPEN 'REVERSED' VALID ONLY ON STANDARD SEQUENTIAL FILES.

IJS625I E    OPEN 'REVERSED' INVALID FOR FILES WITH FORMAT V RECORDS.

IJS626I E    CLOSE 'UNIT' OR 'REEL' VALID ONLY FOR STANDARD SEQUENTIAL
             FILES.

IJS627I E    'INVALID KEY' INVALID FOR STANDARD, DIRECT, OR RELATIVE
             SEQUENTIAL FILES, OR FOR REWRITE ON INDEXED SEQUENTIAL
             FILES.

IJS628I E    OPEN 'ACTUAL KEY' REQUIRED FOR DIRECT SEQUENTIAL OUTPUT
             FILES.

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.
```

Explanation: The size of the assembler phase tables
exceeds the core storage available for these tables.

User Response: Modify the source program to allow compila-
tion 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 E    EXECUTION CANCELED DUE TO E LEVEL DIAGNOSTIC.

IJS707I E    CONFLICTING I/O ASSIGNMENTS.
```

Explanation: SYS001, SYS002, and SYS003 are not assigned
to the same type of device.

System Action: Compilation is canceled.

```
IJS708I E    STORAGE ALLOCATED TO THE COMPILER IS LESS THAN 10K.  compi-
             lation canceled.

IJS708I E    STORAGE ALLOCATED TO THE COMPILER IS LESS THAN 14K.  COMPI-
             LATION CANCELED.

IJS709I W     INCORRECT COBOL OPTION 'XXX'.

IJS710I W     BUFFSIZ CANNOT BE LESS THAN 170.  ASSUMED 170.

IJS711I W     BUFFSIZ CANNOT BE GREATER THAN 32000.  ASSUMED 32000.
```

IJS712I W   BUFFSIZ CANNOT BE GREATER THAN 3625 (7294) FOR WORK FILES
             ON DISK. ASSUMED 3625 (7294).

             Explanation: 3625 for 2311
                           7294 for 2314

IJS713I W   BUFFSIZ IS TOO LARGE FOR SIZE OF STORAGE ALLOCATED TO THE
             COMPILER. ASSUMED XXX.

EXECUTION TIME MESSAGES

A list of execution time messages follows. Most of them are self-
explanatory. Where deemed necessary, examples are included to explain
the message.

#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 OR COBOL COMPILATION ON
             SYS004.

218