SC28-6852-1

Program Product

IBM OS FORTRAN IV (H Extended) Compiler Programmer's Guide

Program Numbers: 5734-FO3
5734-LM3

This publication describes the steps to compile, link edit, and execute a FORTRAN IV program using the FORTRAN IV (H Extended) compiler, an IBM Program Product that operates under the control of the operating system. The methods of invoking each step, input to the steps, and output from the steps, are detailed. In addition, compiler options, features of the operating system used by the FORTRAN programmer, and practices for coding more efficient FORTRAN programs are discussed.

This publication is directed to programmers familiar with the FORTRAN IV language. Previous knowledge of the operating system is not required.

Information in this publciation pertaining to OS/VS2 is for planning purposes until that product is available.

IBM

| Second Edition (June 1972)

This edition corresponds to Release 1 of the FORTRAN IV (H Extended) Compiler.

Changes are periodically made to the specifications herein; any such changes will be reported in subsequent revisions or Technical Newsletters.

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

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

Summary of Amendments — Number 1

Date of Publication: June, 1972

Form of Publication: Revision, SC28-6852-1

OS/VS Information Added

New: Programming and Documentation

The use of the FORTRAN IV (H Extended) compiler with OS/VS1 and OS/VS2 is described. A brief overview of the virtual storage concept has been added to the Introduction, and the section "Using Job Control Statements" has been updated to include descriptions of the VS REGION and ADDRSPC parameters of the JOB and EXEC statements and the VS COPIES, DLM, and SYSOUT parameters of the DD statement. The ORDER and PAGE control statements of the linkage editor are discussed in the "Linkage Editor and Loader" section. In addition, publication references throughout the text have been revised to include the appropriate OS/VS publications.

ERRSET Description Expanded

Maintenance: Documentation Only

The description of the inomes option of the ERRSET subroutine of the Extended Error Handling Facility has been expanded to explain how an unlimited number of error messages may be printed.

Compiler System Requirements Information Removed

Maintenance: Documentation Only

Information about the system requirements of the FORTRAN IV (H Extended) compiler has been removed from the "Programming Considerations" section. This information can be found in the publication *OS FORTRAN IV (H Extended) Installation Reference Material,* Order No. SC28-6861.

Descriptions of SPLIT and SUBALLOC Removed

Maintenance: Documentation Only

Information pertaining to the SPLIT and SUBALLOC parameters of the DD statement has been removed from the section "Using Job Control Statements." Detailed descriptions of these parameters can be found in the job control language reference publications listed in the Preface.

System/360 Dependence of Some Direct Access Units Noted

Maintenance: Documentation Only

The use of the 2303, 2311, and 2321 direct access units with System/360 machines only is indicated in Appendix C: Unit Types.

Description of XCAL Removed

Maintenance: Documentation Only

Information pertaining to the XCAL option of the linkage editor has been removed from the section "Linkage Editor and Loader." A detailed description of this option can be found in the linkage editor and loader publications listed in the Preface.

Asynchronous I/O Information Expanded

Maintenance: Documentation Only

Added information includes the fact that blocked records may not be used with asynchronous I/O, that a BUFNO specification of three buffers in a DCB parameter is limited to asynchronous I/O only, and that a user-exit routine cannot perform any asynchronous I/O operation, even if the error involved was not caused during asynchronous I/O processing.

SIZE Option Information Expanded

Maintenance: Documentation Only

The use of the SIZE option of the PARM parameter of the JOB or EXEC statements to reduce the amount of storage required by the compiler in a multitasking environment is explained. The restriction that SIZE may not be specified on a *PROCESS card is noted.

OPTIMIZE(2) Information Expanded

Maintenance: Documentation Only

The explanation of computational reordering results under OPTIMIZE(2) in the section "Programming Considerations" is expanded.

Editorial changes having no technical significance are not noted here.
Specific changes to the text as of this publishing date are indicated by a vertical bar to the left of the text.
These bars will be deleted at any subsequent republication of the page affected.

This publication describes the use of the operating system and the FORTRAN IV (H Extended) compiler to process programs written in the FORTRAN IV language. The FORTRAN IV language is described in the publication IBM System/360 and System/370: FORTRAN IV Language, Order No. GC28-6515.

This publication is designed for FORTRAN users having various programming needs. The programmer who wants to process a FORTRAN program using the compiler and wants to interpret the output should read the introduction and the first section "How to Submit a FORTRAN Program" in "Part I--Job Input." The introduction explains system concepts to help the programmer understand how the compiler and other operating system resources process a FORTRAN program and also describes the forms of program output. The section "How to Submit a FORTRAN Program" describes in greater detail how a program is processed under control of the operating system.

The remainder of the publication is directed to programmers who required detailed reference material regarding the operation of the compiler and its facilities.

The publication is organized as follows:

- Introduction. The introduction provides an overview of the operating system and how it interacts with the compiler during the processing of a FORTRAN program. Job control language, cataloged procedures, data sets and libraries, and program output are described. The information contained in the introduction is covered in greater detail in subsequent sections.

- Part I--Job Input. Part I describes how a FORTRAN program is processed under control of the operating system. The first section summarizes how a FORTRAN program may be submitted for processing. The remaining sections describe job control language; the three steps in processing a FORTRAN program (compiling, link editing, and executing a load module); and IBM-supplied cataloged procedures.

- Part II--Job Output. Part II describes the output from each step in FORTRAN processing and includes examples of both operating system and FORTRAN program output.

- Part III--Programming Techniques. Part III deals with program efficiency. The first section describes programming considerations for designing and coding more efficient FORTRAN programs. The remaining sections describe such facilities as the compiler automatic precision increase (API) facility, the linkage editor overlay feature, and the load module extended error handling facility.

- Appendixes. The appendixes describe sample FORTRAN programs; the use of assembler language subprograms; the input/output units available with System/360 and System/370; and the carriage control characters available to the FORTRAN programmer.

The FORTRAN programmer using the FORTRAN IV (H Extended) compiler and the FORTRAN IV library (Mod II) should be familiar with the information in the following publications:

IBM System/360 and System/370
FORTRAN IV Language
Order No. GC28-6515

OS FORTRAN IV Library
Mathematical and Service Subprograms
Order No. GC28-6818

OS FORTRAN IV Mathematical and Service Subprograms
Supplement for Mod I and Mod II Libraries
Order No. SC28-6864

OS FORTRAN IV (H Extended) Compiler and FORTRAN IV Library (Mod II) Messages
Order No. SC28-6865

Information about the job control language can be found in the following publications:

OS/MFT and OS/MVT Job Control Language Reference, Order No. GC28-6704

OS/VS Job Control Language Reference, Order No. GC28-0618

Information about IBM-supplied utility programs can be found in the following publications:

OS/MFT and OS/MVT Utilities, Order No. GC28-6586

OS/VS Utilities, Order No. GC35-0005

Information about the linkage editor and loader programs can be found in the following publications:

OS/MFT and OS/MVT Linkage Editor and Loader, Order No. GC28-6538

OS/VS Linkage Editor and Loader, Order No. GC26-3813

Information about debugging techniques can be found in the following publications:

OS/MFT and OS/MVT Programmer's Guide to Debugging, Order no. GC28-6670

OS/VS1 Debugging Guide, Order No. GC24-5093

OS/VS2 Debugging Guide, Order No. GC28-0632

Information about assembler language programming can be found in the following publications:

OS/MFT and OS/MVT Assembler Language, Order No. GC28-6514

OS/VS Assembler Language, Order No. GC33-4010

OS/MFT and OS/MVT Assembler Programmer's Guide, Order No. GC26-3756

OS/VS Assembler Programmer's Guide, Order No. GC33-4021

Information about System/360 and System/370 machine characteristics can be found in the following publications:

IBM System/360 Principles of Operation, Order No. GA22-6821

IBM System/370 Principles of Operation, Order No. GA22-7000

FIGURES

TABLES

A program written for the FORTRAN IV (H Extended) compiler is processed under control of the IBM System/360 Operating System. The compiler is a program product included as part of the operating system in a process called program installation. The operating system consists of a control program and a number of processing programs that perform operations on problem programs.

CONTROL PROGRAM

The control program supervises the execution of the operating system and provides services required by all other programs. One of four control programs may be specified for use with the FORTRAN IV (H Extended) compiler: Multiprogramming with a Fixed Number of Tasks (MFT), Multiprogramming with a Variable Number of Tasks (MVT), and the virtual storage control programs VS1 and VS2. Each control program may handle up to 15 concurrently operating programs, and each control program has these major functions:

1. The supervisor is the control center of the operating system and coordinates all activity within it.
2. The job scheduler reads and analyzes the input job stream (control statements and data entering the system), allocates input/output devices as necessary, initiates the execution of programs, and provides a record of the work processed.
3. Data management routines control input/output operations, regulate the use of input/output devices after the job scheduler allocates them, and provide access to the data held in them.

The MFT and VS1 control programs divide computer storage into areas of fixed sizes called partitions. Data entering the system is directed to these partitions on a priority scheduling basis; that is, data is not processed in the order in which it is encountered in the input stream but according to a priority code assigned by the user.

The MVT and VS2 control programs, like MFT and VS1, process data on a priority scheduling basis, but direct it to storage areas of variable sizes, called regions; each user indicates the amount of storage he needs.

VS1 is the virtual storage version of MFT, and VS2 is the virtual storage version of MVT. Both VS control programs provide the same services as their non-relocatable counterparts, but the partitions (VS1) and regions (VS2) to which job steps are directed are represented in virtual, rather than real, storage. Virtual storage is the name given to the address space that appears to the user as real (main) storage and from which instructions and data are mapped into real storage locations. The size of virtual storage is limited by the addressing range of the computing system, not the number of real storage locations. (Under OS/VS the term real storage is used to designate the physical main storage of System/370 rather than its complete addressing range.)

Virtual storage is provided by space on direct access storage devices (DASDs) and implemented by a combination of the Dynamic Address Translation (DAT) hardware feature of System/370 (often called the "relocate" feature) and the paging function provided by the OS/VS control programs. The contents of virtual storage are formatted into 2048 byte blocks (VS1) or 4096 byte blocks (VS2). These blocks are called pages. During execution of a program, the hardware system translates each virtual storage address into a corresponding real storage address that designates a physical location. The translation process actually amounts to relocation by a value that is a multiple of one page. The VS control program manages the transfer of pages between auxiliary storage and real storage. The number of pages that may reside in real storage at any given time depends on the amount of real storage available. The VS control programs enable the central processing unit to execute programs residing in virtual storage without moving all of the pages into real storage concurrently.

PROCESSING PROGRAMS

Processing programs perform specific tasks, such as language translation, link editing, loading, sorting and merging, and various utility functions. Language translators, or compilers, include FORTRAN, COBOL, and

PL/I. The linkage editor and the loader programs combine many programs into one executable unit. Sort and merge programs sequence data into ordered formats prior to further processing. Utility programs provide the ability to create, print, duplicate, and reformat collections of data.

Processing programs of special concern to the FORTRAN programmer are the FORTRAN IV (H Extended) compiler, which translates FORTRAN statements into executable instructions; the linkage editor, which combines the FORTRAN program with subroutines from the FORTRAN library or user libraries with other routines required by the system; and the loader, which combines a number of programs and subprograms as does the linkage editor and then executes the resulting program.

PROBLEM PROGRAMS

Problem programs are written by the user for the creation and maintenance of files, creation of reports, solution of problems, or for whatever use the user wishes to make of the computer (for instance, a FORTRAN program, designed to do some particular work). Before it can do its work, however, the FORTRAN program must be processed by the operating system.

PROCESSING A FORTRAN PROGRAM

Three basic steps are taken to process a FORTRAN program: the compile step, the link edit step, and the load module execution (or go) step. The input to the compile step is the group of FORTRAN statements called a source module. The output from the compile step is a group of compiler-translated statements called an object module, which is the input to the link edit step. The output of the link edit step is the object module combined with other modules and is called the load module, the program that is executed in the go step. If the loader is used in place of the linkage editor, the last two steps (link edit and load module execution) are combined into one step.

Each step is termed a job step--the execution of one program. Each job step may be executed alone or in combination with other job steps as a job--an application involving one or more job steps. Hence, a job may consist of one step, such as FORTRAN compiler execution, or of many steps, such as compiler execution followed by linkage editor execution and load module execution.

The programmer defines the requirements of each job to the operating system through job control language statements.

JOB CONTROL LANGUAGE STATEMENTS

A job control language statement is identified by the appearance of the characters // or /* in the statement's first two positions. The job control statements most often used by FORTRAN IV programmers are the JOB, EXEC, DD, delimiter, and null statements.

The JOB statement identifies the beginning of a job. It is required for each job, must be the first statement in a job, and must have a name to identify the job. It may also contain other information such as the programmer's name, accounting information, certain system options, and comments.

The EXEC statement identifies the beginning of a job step. It is required for each job step and must be the first statement in the step. The EXEC statement may be named to identify the step. The statement must specify the name of the program to be executed. It may also contain other information such as processor options and comments.

The DD statement defines a data set. A data set is an organized collection of records such as a source program, a set of input records, or a library of subprograms. The DD statement specifies information regarding a data set's characteristics, its location, its name, format of records contained in the data set, and the disposition to be made of the data set at the end of the job step.

The delimiter statement contains the characters /* in the first two positions and indicates the end of a data set contained on cards.

The null statement contains the characters // in the first two positions and indicates the end of the job. The remainder of the card must be blank.

The following examples illustrate the positioning of job control statements within a job for some of the combinations of FORTRAN processing.

Compile Only (One Source Module)

```
// JOB Statement
// EXEC Statement (to execute FORTRAN
      compiler)
// DD Statements for compilation (as
      required)
  ┌----------------------------┐
  |Source module to be compiled|
  └----------------------------┘
/* Delimiter Statement (if source module is
      on cards)
// Null Statement
```

Compile Only (More Than One Source Module)

```
// JOB Statement
// EXEC Statement (FORTRAN compiler)
// DD Statements for compilation 1 (as
      required)
  ┌----------------------------┐
  |Source module to be compiled|
  └----------------------------┘
/* Delimiter Statement (if source module is
      on cards)
// EXEC Statement (FORTRAN compiler)
// DD Statements for compilation 2 (as
      required)
  ┌----------------------------┐
  |Source module to be compiled|
  └----------------------------┘
/* Delimiter Statement (if source module is
      on cards)
// EXEC statement (FORTRAN compiler)
// DD Statements for compilation 3 (as
      required)
  ┌----------------------------┐
  |Source module to be compiled|
  └----------------------------┘
/* Delimiter Statement (if source module is
      on cards)
// Null Statement
```

Compile and Link Edit

```
// JOB Statement
// EXEC Statement (FORTRAN compiler)
// DD Statements for compilation (as
      required)
  ┌----------------------------┐
  |Source module to be compiled|
  └----------------------------┘
/* Delimiter Statement (if source module is
      on cards)
// EXEC Statement (Linkage Editor)
// DD Statements for link editing (as
      required)
// Null Statement
```

Compile, Link Edit, and Execute

```
// JOB Statement
// EXEC Statement (FORTRAN compiler)
// DD Statements for compilation (as
      required)
  ┌----------------------------┐
  |Source module to be compiled|
  └----------------------------┘
/* Delimiter Statement (if source module is
      on cards)
// EXEC Statement (Linkage Editor)
// DD Statements for link editing (as
      required)
// EXEC Statement (Load Module)
// DD Statements for load module execution
      (as required)
  ┌------------------------------------┐
  |Data input statements to be processed|
  └------------------------------------┘
/* Delimiter Statement (if data input is on
      cards)
// Null Statement
```

As these examples show, the same job control statements are used repeatedly. To save programming time and to reduce the possibility of error, IBM supplies sets of job control statements that the programmer may use in place of his own. Each set is given a name and is stored in a data set called the procedure library. The sets of statements are called cataloged procedures.

CATALOGED PROCEDURES

To retrieve a cataloged procedure from the library, the programmer specifies its name in an EXEC statement in place of a program name. The effect is the same as though the job control statements in the cataloged procedure appeared in the input stream in place of the EXEC statement that called it.

Note that a cataloged procedure does not execute a program; it merely supplies a set of job control statements. These statements must include appropriate EXEC statements naming programs to be executed.

IBM provides seven cataloged procedures for use with the FORTRAN IV (H Extended) compiler. They are:

- FORTXC, which includes one EXEC statement, to execute the compiler
- FORTXCL, which includes two EXEC statements, to execute the compiler and the linkage editor
- FORTXLG, which includes two EXEC statements, to execute the linkage editor and the load module

- FORTXCLG, which includes three EXEC statements, to execute the compiler, the linkage editor, and the load module

- FORTXCG, which includes two EXEC statements, to execute the compiler and the loader

- FORTXG, which includes one EXEC statement, to execute a load module

- FORTXL, which includes one EXEC statement, to execute the loader

In addition to EXEC statements, cataloged procedures contain DD statements to define data sets needed by each job step.

DATA SETS

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

An input/output device is the piece of equipment that does the recording and/or the reading of data from the volume. For example, a device may be a tape drive or a disk drive.

Several input/output devices may be grouped together into a device class when the system is generated. A device class is referred to by a collective name. For example, the device class SYSDA may consist of all direct access devices in an installation; the device class SYSSQ may consist of magnetic tape and direct-access devices in an installation.

A data set may be a system data set or a user data set. A system data set is one used by the system; for example, to store an object module. A user data set is one defined by the programmer for a specific application; for example, to store intermediate results during program processing.

SYSTEM DATA SETS

System data sets are used in all three steps of FORTRAN processing.

Compiler Data Sets

Compiler data sets are defined in DD statements named SYSIN, SYSPRINT, SYSPUNCH, SYSLIN, SYSUT1, and SYSUT2.

- SYSIN specifies the primary input (the source module) to the compiler.

- SYSPRINT specifies the output data set used to write listings and messages.

- SYSPUNCH specifies the output data set used to punch cards or to write output in card image form.

- SYSLIN specifies the data set in which the object module was produced by the compiler.

- SYSUT1 and SYSUT2 specify utility data sets required by the compiler.

Linkage Editor Data Sets

Linkage editor data sets are defined in DD statements named SYSLIN, SYSPRINT, SYSLIB, SYSLMOD, and SYSUT1.

- SYSLIN specifes the data set in which the object module was created in a compile step and that is to be the input to the link edit step.

- SYSPRINT specifies the output data set used to write listings and messages.

- SYSLIB specifies the system data set, SYS1.FORTLIB, that contains IBM-supplied FORTRAN subprograms. The linkage editor uses this data set to include FORTRAN subprograms called by the compiler.

- SYSLMOD specifies the load module produced by the linkage editor.

- SYSUT1 specifies a utility data set required by the linkage editor.

Load Module Execution Data Sets

Load module execution data sets are defined by DD statements with names in the form FTxxFyyy, where xx is a data set reference number and yyy is a sequence number. A data set reference number may range from 01 to 99; a sequence number may range from 001 to 999. A data set reference number equates the data set defined in the DD statement to the data set referenced in the

READ or WRITE statement in the source program. For example, the name FT08F001 describes the data set referenced by the READ statement:

```
READ (8,x)
```

IBM-supplied cataloged procedures contain DD statements named FT05F001, FT06F001, and FT07F001 which reserve data set reference number 5 for an input data set, 6 for a printer, and 7 for a card punch. All other data set reference numbers are available to the programmer for the definition of other data sets. These standards may be changed by the individual installation or they may be overridden by the programmer. The following discussion describes the data sets as specified in the IBM-supplied cataloged procedures.

FT05F001 defines the input data set. The FORTRAN programmer normally uses 5 as the data set reference number in the READ statement for input to his program.

FT06F001 defines the data set directed to the printer. The FORTRAN programmer uses 6 as the data set reference number in the WRITE statement for printed output.

FT07F001 defines the data set directed to the card punch. The FORTRAN programmer uses 7 as the data set reference number in the WRITE statement to direct output to the punch.

Loader Data Sets

Because the loader combines two job steps, it uses some of the same data sets as the linkage editor and the load module, such as SYSLIN, SYSLIB, FT05F001, FT06F001, and FT07F001.

In addition, it uses a DD statement named SYSLOUT to specify the output data set used to write loader output.

USER DATA SETS

User data sets are those which the user requires for his particular program. For example, the programmer must define data that will be created by a program or read from a previous program as a user data set. A user data set in FORTRAN may be one of three types: sequential, partitioned, or direct-access.

A sequential data set is one in which records are organized solely on the basis of their successive physical positions (i.e., arranged in sequential order). A sequential data set may reside on cards, tape, or a direct-access device. A FORTRAN source module is an example of a sequential data set.

A partitioned data set is one that is composed of groups of sequential data, each of which is called a member. Each member has a name stored in a directory that is part of the data set and that contains the location of each member's starting point. Partitioned data sets are used as libraries (data sets containing a number of programs). SYS1.FORTLIB, which contains FORTRAN subroutines, is an example of a partitioned data set.

A direct-access data set is one in which records may be accessed individually by specifying the position of the record within the data set. For example, if the 100th record of a sequential data set is needed, records 1 through 99 must be transmitted; record 100 of a direct-access data set can be accessed immediately. Records in a direct-access data set may be processed in FORTRAN only by direct-access I/O statements but may be processed in a sequential fashion through use of the associated variable in the direct-access statements. Direct-access data sets may reside only on direct-access devices. A data set containing information on a department store's customers, arranged according to charge account numbers that may be accessed individually, is an example of a direct-access data set.

CATALOGED DATA SETS

Any partitioned or direct-access data set and any sequential data set not on cards, may be cataloged. A cataloged data set is one for which certain information describing the data set has been placed in an index called the catalog. When a data set is cataloged, the serial number of its volume is entered in the catalog together with the data set name, thereby providing a cross-reference. A programmer can subsequently refer to the data set without specifying its physical location. (The term cataloged data set should not be confused with cataloged procedure.)

LIBRARIES OF DATA SETS

A library is a partitioned data set containing a number of programs. Libraries supplied by IBM or the installation are

system libraries; libraries created by the programmer are user libraries.

System Libraries

System libraries most useful to a FORTRAN programmer are SYS1.LINKLIB, SYS1.FORTLIB, and SYS1.PROCLIB.

SYS1.LINKLIB contains frequently-used programs, such as the FORTRAN compiler.

SYS1.FORTLIB contains FORTRAN subprograms used during FORTRAN processing.

SYS1.PROCLIB contains cataloged procedures and may contain user-written cataloged procedures as well as those supplied by IBM.

User Libraries

A user library is created by the programmer to serve a special application, such as storing frequently-used programmer-written programs. The programmer may adopt his own convention in naming his libraries.

A programmer creates a user library and adds programs or subprograms to it by defining it on a SYSLMOD DD statement of a linkage edit job step. He retrieves it through a DD statement named STEPLIB or JOBLIB. STEPLIB makes the library accessible during one step of a job; JOBLIB makes the library accessible for all steps of the job. When JOBLIB is specified, the DD statement must be placed immediately after the JOB statement to ensure that the library remains available for the duration of the job. The STEPLIB DD statement may appear anywhere among the DD statements within a job step.

PROGRAM OUTPUT

In addition to object and load modules, jobs produce other forms of output that help the programmer analyze the job. The compile step informs the user of the success of the compilation by issuing messages, including a severity code for any error encountered. The following severity codes are associated with compilation messages:

- 0 to indicate an informational message; that is, no errors was detected
- 4 to indicate a warning message; that is, a possible minor syntactic error was detected
- 8 to indicate an error message; a syntactic error was detected
- 12 to indicate a serious error message; a serious syntactic error that prevents program execution was detected
- 16 to indicate an unrecoverable error message; compiler processing was terminated because of an error

Severity codes 0 and 4 require no action on the part of the programmer; the program may be executed. (Nevertheless, severity code 4 messages should be corrected.) The other codes require corrections before the program may be executed.

The compiler may also produce output, such as a card deck of the object module, a listing of the statements in an object module (pseudo-assembler listing), and a storage map that lists the names and storage locations of variables appearing in the object module.

The link edit step produces a map of the load module and may also produce a cross-reference listing that shows where the modules making up the program interrelate.

The load module step produces program output as required by the FORTRAN program and also produces any messages describing conditions encountered during execution.

PART I -- JOB INPUT

The simplest way to submit a FORTRAN program is by calling one of the cataloged procedures. The programmer calls a procedure by submitting:

1. A JOB statement
2. An EXEC statement that names the procedure
3. The input to the operating system, such as the FORTRAN program to be compiled
4. A null statement at the end of the job

In addition, if the FORTRAN program is submitted in card deck form, the programmer must supply the following:

1. A DD statement having the name SYSIN immediately preceding the card deck. The name SYSIN defines input data.
2. A delimiter control statement immediately following the card deck to mark the end of card input.

Figure I-1 lists all the cataloged procedures for the FORTRAN IV (H Extended) compiler and an example in submitting card input for each of them.

Note that in Figure I-1, the DD statement named SYSIN may appear more than once within a job. The qualified name FORT.SYSIN defines the source module to the compile step (which is named FORT in cataloged procedures); the qualified name GO.SYSIN defines data records to be processed by the load module (go) step.

Cataloged Procedure Name	Function	Calling the Procedure

FORTXC	Compile	//jobname JOB
		// EXEC FORTXC
		//FORT.SYSIN DD *
		[FORTRAN source module]
		/*
		//

FORTXCL	Compile and Link Edit	//jobname JOB
		// EXEC FORTXCL
		//FORT.SYSIN DD *
		[FORTRAN source module]
		/*
		//

FORTXCLG	Compile, Link Edit and Execute	//jobname JOB
		// EXEC FORTXCLG
		//FORT.SYSIN DD *
		[FORTRAN source module]
		/*
		//GO.SYSIN DD *
		[Data]
		/*
		//

Figure I-1. Submitting FORTRAN Programs Through Cataloged Procedures (Part 1 of 2)

Cataloged Procedure Name	Function	Calling the Procedure

FORTXLG	Link Edit and Execute	//jobname JOB
		// EXEC FORTXLG
		//LKED.SYSIN DD *
		[first FORTRAN object module]
		.
		.
		.
		[last FORTRAN object module]
		/*
		//GO.SYSIN DD *
		[Data]
		/*
		//

FORTXG	Execute	//jobname JOB
		//JOBLIB DD (parameters to describe library)
		// EXEC FORTXG
		//GO.SYSIN DD *
		[Data]
		/*
		//

FORTXCG	Compile and Load	//jobname JOB
		// EXEC FORTXCG
		//FORT.SYSIN DD *
		[FORTRAN source module]
		/*
		//GO.SYSIN DD *
		[Data]
		/*
		//

FORTXL	Load	//jobname JOB
		// EXEC FORTXL
		//LKED.SYSLIN DD (parameters)
		//GO.SYSIN DD *
		[Data]
		/*
		//

Figure I-1. Submitting FORTRAN Programs Through Cataloged Procedures (Part 2 of 2)

The FORTRAN programmer should do one more thing before submitting a program for execution. If input and output data consist solely of cards and printer listing, he should code data set reference numbers for FORTRAN READ and WRITE statements as 5 and 6 respectively. Cataloged procedure FORTXCLG (and FORTXLG) contain DD statements that allocate the card reader to data set reference number 5 and the printer to data set reference number 6.

The programmer need do no more than the foregoing to process a FORTRAN program. Use of the cataloged procedure FORTXCLG, as illustrated, enables the control program to read the source module, read the load module input data set, and write program output and system messages on the printer.

However, if the source module resides on some media other than cards (i.e., on tape or in a direct-access volume) or if the programmer needs specific data sets during program processing, he must make these private data sets available to the control program. To do this, he must define them in DD statements. The following discussion describes how to create and retrieve such data sets.

DEFINING PRIVATE DATA SETS

A private data set may be temporary or permanent. A temporary data set is one that is created and deleted in the same job; it is usually preferred when the programmer needs space temporarily to store data during program execution. The following is an example of a DD statement defining a temporary data set:

```
//GO.FT09F001 DD DSNAME=&&SET,UNIT=2311,
//            SPACE=(TRK,(5,5))
```

The name GO.FT09F001 identifies a data set to be processed by the execute step and links the data set to FORTRAN READ and WRITE statements having 9 as the data set reference number. DSNAME=&&SET assigns the name set to the data set; the symbols && in the first positions signify a temporary data set. The UNIT parameter indicates that the data set is to reside on a 2311 device. SPACE indicates the amount of space allocated to the data set; space is allocated in number of tracks on the 2311 device, five tracks initially and five additional tracks each time the data set needs more space.

A permanent data set is one that is created and kept at the end of the job; it is defined when the data set is to contain information required in later jobs. To respecify the data set in the previous example as a permanent data set, the programmer may submit the following statement:

```
//GO.FT09F001 DD DSNAME=SET,UNIT=2311,
//            SPACE=(TRK,(5,5)),
//            VOLUME=SER=DA003,
//            DISP=(NEW,KEEP)
```

The DSNAME parameter names the data set without the && symbols in the first positions. The VOLUME parameter indicates that the data set is to reside on the disk pack having the serial number DA003. The VOLUME parameter is not mandatory when creating a data set; if it is omitted, the data set is assigned to any available volume. However, to retrieve the data set in a later job, the control program needs to know which volume to access. The DISP parameter indicates that the data set is new and is to be kept. The DISP parameter is mandatory; without it, the data set would be deleted at the end of the job.

To retrieve the data set SET in a later job, the programmer changes the DISP parameter from NEW to OLD; i.e.,

```
//GO.FT09F001 DD DSNAME=SET,UNIT=2311,
//            VOLUME=SER=DA003,
//            DISP=(OLD,KEEP)
```

OLD indicates that the data set existed prior to the job. (The underscore in these examples is for visual aid only and is not part of the statement coding.) Note that the SPACE parameter is not specified; it is used only when creating a data set. Note also that the UNIT and VOLUME parameters are repeated. The programmer can avoid repeating these parameters by cataloging the data set, that is, entering the data set's name and unit and volume information in the control program index. To catalog a data set, the programmer changes the DISP parameter from KEEP to CATLG, i.e.,

```
//GO.FT09F001 DD DSNAME=SET,UNIT=2311,
//            VOLUME=SER=DA003,
//            DISP=(OLD,CATLG)
```

Alternatively, the programmer may catalog the data set when he creates it, that is, DISP=(NEW,CATLG), is valid. After a data set has been cataloged, the programmer may retrieve it by specifying only the DSNAME and DISP parameters.

Finally, to delete the data set in a later job, the programmer changes the DISP parameter from CATLG (or KEEP) to DELETE, i.e.,

```
//GO.FT09F001 DD DSNAME=SET,
//            DISP=(OLD,DELETE)
```

DELETE indicates that the space occupied by the data set is to be released and made available for other uses at the end of the job.

The operating system contains many more options that can be specified in the job control language. The programmer may choose those options he needs to tailor the system's facilities to a particular application. The following sections describe some of those options and how he may use them.

Job control statements provide a communications link between the FORTRAN programmer and the operating system. The FORTRAN programmer uses these statements to define a job, a job step within a job, and data sets required by the job. A full description of job control language can be found in the appropriate job control language reference publication, as listed in the Preface.

This section describes the job control statements most used by the FORTRAN programmer: the JOB, DD, EXEC, delimiter, comments and null statements. Another job control statement, the PROC statement, has a special application in FORTRAN processing; it assigns default values to program options specified in cataloged procedures. Because of its limited applicability to FORTRAN programs, the PROC statement is not further discussed in this section (see "IBM-Supplied Cataloged Procedures" for a discussion of this statement).

Job control statements are identified by the characters // in columns 1 and 2, except for the delimiter statement, which is identified by the characters /* in columns 1 and 2, and the comments statement, identified by the characters //* in columns 1, 2, and 3. The delimiter statement may contain optional comments preceded by one or more blanks. The null statement may contain only the two slashes; the remainder of the statement must be blank. The comments statement contains notes written by the programmer. The other three statements discussed in this section can contain up to four fields: name, operation, operand, and comments.

Table I-1 summarizes the function of job control statements. Figure I-2 illustrates the format of job control statements. The brackets around some of the items illustrated indicate that those items are optional when using the statement.

Table I-1. Job Control Statement Functions

Statement	Function
JOB	Indicates the beginning of a new job and describes that job
EXEC	Marks the beginning of a job step and indicates the program or cataloged procedure to be used
DD	Describes data sets and controls device and volume assignment
Delimiter	Separates data sets in the input stream from control statements; it appears after each data set in the input stream
Null	Indicates the end of a job
Comment	Contains miscellaneous remarks written by the programmer; it can appear before or after any control statement

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

Figure I-2. Job Control Statement Formats

The name field begins in column 3, immediately following the //.

The name field assigns a name to the statement, identifying it to other statements and to the operating system. A name consists of from one to eight alphameric characters or the characters #, $, or @. The first character of the name must be alphabetic. A DD statement may contain a qualified name which is two names joined together with periods; the name FORT.SYSIN is an example of a qualified name.

The operation field begins in any column after the name field and is preceded and followed by one or more blanks.

The operation field identifies the type of control statement, e.g., JOB, DD, or EXEC.

The operand field begins in any column after the operation field and is preceded and followed by one or more blanks. The operand field identifies system options requested by the programmer. Options are specified through one or more parameters separated by commas.

Parameters may be either positional or keyword. Positional parameters must appear in a fixed order and are identified, or given meaning, by their position in that order. A keyword parameter is composed of an identifying keyword, an equal sign (=), and a value. Parameters may also comprise a number of subparameters, which can be either positional or keyword.

The comments field begins in any column after the operand field (or the /* in the delimiter statement) and is preceded by one or more blanks. The comments field may contain any information that is considered helpful. There is no required syntax for comments.

All statements (except the null statement) may be continued onto succeeding cards, using the following rules:

1. Interrupt operands after a completed parameter or subparameter, including the comma that follows it, at or before column 71.

2. If comments are desired in the same statement as an interrupted operand field and there is sufficient room, leave one or more spaces after the comma that follows the last parameter, and then code the comments.

3. For an interrupted comment, code any nonblank character in column 72. For an interrupted operand, the nonblank character in column 72 is optional. If a nonblank character is not coded in column 72 of an interrupted operand, but the conventions outlined in the next two items are followed, the job scheduler will treat the next statement as a continuation statement.

4. Code the identifying characters // in columns 1 and 2 of the following card or card image.

5. Continue the interrupted operand beginning in any column from 4 through 16.

Note that excessive continuation cards (or card images) should be avoided whenever possible to reduce processing time for the control program.

SYNTAX FOR PARAMETER DESCRIPTION

In describing the syntax for parameters, this publication follows the conventions listed in the following paragraphs.

1. The set of symbols listed below are used to define control statements, but are never coded in a control statement.

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

   The special uses of these symbols are explained in paragraphs numbered from 4 to 10.

2. Uppercase letters and words, numbers, and the set of symbols listed below are coded in a control statement exactly as shown in the statement definition.

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

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

   Example:

   DSNAME=filename

   may be coded as

   DSNAME=MYNAME

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

   Example:

   PGM=program-name

   may be coded as

   PGM=MYNAME

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

   For example:

   or an alternate designation:

   {A|B|C}

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

6. Brackets also group related items; but everything within the brackets is optional and may be omitted.

   Example:

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

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

7. An underscore indicates the standard default option. If a default option is selected, it need not be coded in the actual statement.

   PARM=[DECK / NODECK]

   If the user wants the DECK option, he will have to specify it; if he wants the NODECK option, he may specify it, but he does not need to do so.

8. Braces group related items, such as alternatives.

   Example:

   ALPHA=({A|B|C},D)

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

9. An ellipsis indicates that the preceding item or group of items can be repeated many times.

   Example:

   ALPHA[,BETA]...

   The preceding example indicates that ALPHA can appear alone or it can appear followed by ,BETA optionally repeated many times in succession.

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

    Example:

11. Blanks are used to improve the readability of control statement definitions. Unless otherwise noted, blanks may not appear in a statement definition.

JOB STATEMENT

A JOB statement initiates the beginning of a job and assigns a name to it.

The JOB statement may also contain the following information:

1. Accounting information relative to the job
2. Programmer's name
3. The type of system messages to be written

4. Conditions for terminating the execution of the job

5. Assignment of input and output classes

6. Job priority

7. Main storage requirements

8. A time limit for the job

Figure I-3 illustrates the format of the JOB statement. Table I-2 summarizes the functions of the JOB statement. Figure I-4 shows sample JOB statements.

The job name is always required. All other information is optional unless made mandatory by each installation.

Naming the Job

Jobname identifies the job to the operating system. Because a multiprogramming environment permits many jobs to operate concurrently, the programmer should select a unique jobname for each job.

The accounting information and programmer's name are positional parameters and, if used, must be the first ones specified in the operand field. All other parameters are keyword parameters and may appear in any order.

Specifying Accounting Information

Accounting information is used to store installation-required accounting procedures. It is specified as the first parameter in the operand field. The parameter has no predefined format; it consists of a string of up to 142 characters. If it contains any special characters (valid members of the character set not alphabetic or numeric) other than a hyphen, it is enclosed in apostrophes.

An example of the accounting parameter is:

'215,46WX819'

If this parameter is not present but the programmer's name is, a comma is required to indicate the omission. If both parameters are omitted, no commas are required. (The accounting field may be required, at the installation's option.)

Name	Operation	Operand
		Positional Parameters
//jobname	JOB	[,accounting-information]
		[,programmer-name]
		Keyword Parameters
		[MSGLEVEL=(x,y)]
		[COND=((code,operator)...)]
		[CLASS=job-class]
		[PRTY=job-priority]
		[MSGCLASS=x]
		[REGION=region-size][1]
		[TIME=(minutes,seconds)][2]
		[ADDRSPC=REAL|VIRT][3]

[1]MVT and VS only.
[2]MVT and VS2 only.
[3]VS only.

Figure I-3. Job Statement Format

```
        //CLH1     JOB  NY237413,C.L.HARVEY,MSGLEVEL=1

        //CLH2     JOB  ,HARVEY,COND=(8,EQ),REGION=100K,TIME=60,
        //               CLASS=H,PRTY=10

        //PROGRAM JOB   873,'COVER-RAMSEY',MSGLEVEL=(1,1),MSGCLASS=C,
        //               PRTY=10
```

Figure I-4. Sample JOB Statements

Table I-2. JOB Statement Functions

Specification	Purpose	How to Specify
jobname	To identify the job to the operating system.	One to eight alphameric (alphabetic or numeric) characters, the first of which must be alphabetic or one of the extended alphabetic characters #, @, or $.
accounting-information	To identify the account number or other accounting information relating to the job.	Up to 142 characters; if any special characters other than a hyphen, enclose in apostrophes. See "Specifying Accounting Information" for details.
programmer-name	To identify the person submitting the job.	Up to 20 characters; if special characters other than a period, enclose in apostrophes.
MSGLEVEL	To specify the type of system messages (i.e., job control statements, diagnostic messages, termination messages) to be written as part of the output listing.	MSGLEVEL=(x,y), where x is 0, 1, or 2, to indicate which job control statements and diagnostic messages are to be written, and y is 0 or 1 to indicate whether termination messages are to be written. See "Specifying System Messages" for details.
COND	To specify those conditions that are to result in program termination.	COND=((code,operator)...), where code is a number between 0 and 4095 (usually 0, 4, 8, 12, or 16) and operator is a 2-character value that represents a comparison to be made between code and a return code issued by the operating system. If the comparison is true, the job is terminated. See "Specifying Condition Codes to Terminate a Job" for details.
CLASS	To assign an input class to the job.	CLASS=job-class, where job-class is an alphabetic character A thru O assigning the class represented by that character to the job.
PRTY	To assign a priority to the job.	PRTY=job-priority, where job-priority is a number 0 through 13 assigning the priority represented by that number to the job; the higher the number, the greater the priority.
MSGCLASS	To assign an output class to the job.	MSGCLASS=x, where x is an alphabetic or numeric character assigning the output class represented by that character to the job.
TIME	To assign a time limit to the job.	TIME=(minutes,seconds), where minutes is a number up to 1439, and seconds is a number up to 59.

(Part 1 of 2)

Table I-2. JOB Statement Functions (Part 2 of 2)

Specification	Purpose	How to Specify
REGION	To assign storage to a job operating in the MVT or VS environment.	For MVT, REGION=(nnnnn$_1$K[,nnnnn$_2$K]), where nnnnn$_1$K is a value up to 16383K and indicates the amount of storage to be allocated (K=1024 bytes); nnnnn$_2$K is a value up to 2048K and indicates the amount of additional storage to be allocated, usually from an IBM 2361 Core Storage device. For an IBM 2361 Model 1, nnnnn$_2$K may be a value up to 1024K; for an IBM 2361 Model 2, nnnnn$_2$K may be a value up to 2048K. For VS, REGION=valueK, where valueK represents the number of contiguous 1024-byte areas of real or virtual storage to be allocated. See "Assigning Storage to a Job Under MVT" for details.
ADDRSPC	To specify the mode of operation, real or virtual, of the VS control program.	ADDRSPC=REAL for real mode, and ADDRSPC=VIRT for virtual mode.

Specifying the Programmer's Name

The programmer's name is specified as the second parameter in the operand field. It consists of a string of up to 20 characters. If it contains any special characters other than a period, it is enclosed in apostrophes.

Examples of the programmer-name parameter are:

```
J.SMITH
'COVER-RAMSAY'
```

If this parameter is not present, no comma is required to note the omission.

Specifying System Messages

The MSGLEVEL parameter is used to specify whether job control statements and termination messages are to be written.

MSGLEVEL has the format:

```
MSGLEVEL=(x,y)
```

The letter x represents a job control message code and indicates whether job control statements are to be written along with program statements. The value of x may be 0, 1, or 2. When x=0, the JOB statement, job control statement errors, and diagnostic messages are written. When x=1, all job control statements, including those appearing in called cataloged procedures, are written. When x=2, only job control statements submitted with the job (not taken from cataloged procedures) are written.

The letter y represents a termination message code and indicates whether termination messages are to be produced. The value of y may be 0 or 1. When y=0, no termination messages appear for normal completion of the job; termination messages appear for abnormal termination. When y=1, termination messages appear under any circumstances.

If the parameter is omitted, the default values for x and y are those established by the installation.

An example of MSGLEVEL is:

```
MSGLEVEL=(2,0)
```

The example states that only control statements submitted with the job are to be written, and that termination messages are to be written only if abnormal termination occurs.

Specifying Condition Codes to Terminate a Job

The COND parameter is used to specify which condition codes terminate processing. It is useful in helping the programmer reduce computing time by making job continuation dependent upon successful completion of a previous job step.

The system issues a number, called a return code, at the end of each job step. The return code is an indication of how well the job step ran, i.e., whether it completed processing normally or whether error conditions were detected. The COND parameter indicates a comparison to be made between the return code and the condition code specified in COND; if the condition is met, the job is terminated.

COND has the format:

```
COND=((code,operator)[,(code,operator)]...)
```

Code consists of a number. Operator consists of two alphabetic characters indicating a mathematical relationship between the condition code and the return code. There are six operators, as follows:

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

Up to eight sets of codes and operators may be specified.

An example of COND is:

```
COND=((8,LT))
```

The example states that if 8 is less than the return code issued by the system, the job is to be terminated (any return code less than or equal to 8 allows the job to continue).

Assigning Job Priority

The CLASS parameter is used to assign a job class; the PRTY parameter assigns a priority within a class.

When a job is introduced to the system it is assigned to an input queue according to a code called a class. Main storage is divided into a number of areas, each of which handles jobs assigned a certain class. When one job has completed processing, that storage is given to the next job having the same class. The programmer determines the order in which jobs are entered into the system by specifying the CLASS parameter. He can further refine the order in which jobs within a class are entered into the system by specifying the PRTY parameter.

CLASS has the format:

```
CLASS=a
```

The letter a indicates an alphabetic character A through O. The meaning of the characters is determined by each installation at system generation time. If the parameter is omitted, the default class A is assumed.

PRTY has the format:

```
PRTY=n
```

The letter n indicates a number from 0 through 13; the higher the number, the greater the priority. Whenever possible, priority 13 should be avoided. This number is used by the system for special processing. If the parameter is omitted, a standard default established by each installation is assumed.

An example of the CLASS and PRTY parameters is:

```
CLASS=C,PRTY=10
```

Assigning an Output Writer

The MSGCLASS parameter is used to assign messages to a specific output writer through a class code.

The system assigns messages to a variety of output devices according to the output class. By assigning an output class, the programmer directs output to a specific device.

MSGCLASS has the format:

```
MSGCLASS=x
```

The letter x indicates an alphabetic or numeric character. The meaning of the characters is determined by each installation at program installation time. If the parameter is omitted, the default class A (usually directed to a printer) is assumed.

An example of MSGCLASS is:

```
MSGCLASS=B
```

Assigning a Time Limit to a Job

The TIME parameter is used to assign a processing time limit. If the job is not

completed in the time specified, it is terminated.

TIME has the format:

TIME=(minutes,seconds)

Minutes and seconds are expressed in numeric characters. Minutes cannot exceed 1439; seconds cannot exceed 59. If a job is expected to run longer than 24 hours, the programmer may code TIME=1440 to eliminate job timing. If the TIME parameter is omitted, a default limit established by each installation is assumed.

If time is specified in minutes only, the delimiting parentheses are not required (e.g., TIME=10). If time is specified in seconds only, the parentheses and a comma denoting the omission of minutes are required (e.g., TIME=(,50)).

An example of TIME is:

TIME=(10,30)

The example states that the job may run up to a maximum time of ten minutes and thirty seconds.

Assigning Storage to a Job Under MVT

Under MVT, storage is assigned according to the needs of each particular job. The REGION parameter is used to specify the amount of storage to be allocated. If the parameter is omitted, a default size established by each installation is assumed.

The programmer may assign storage to individual job steps rather than to the entire job by coding the REGION parameter in each EXEC statement instead.

REGION has the format:

REGION=(nnnnn$_1$K[,nnnnn$_2$K])

The term nnnnn$_1$K indicates the number of contiguous 1024-byte areas to be allocated. The number specified may be from one to five digits but cannot exceed the total storage available. Parentheses are not required if only nnnnn$_1$K is specified (e.g., REGION=200K).

The term nnnnn$_2$K is used only if the installation has specified hierarchy support. Hierarchy support indicates that storage may be allocated from two regions, one known as hierarchy 0, the other as hierarchy 1. Hierarchy 0 is always assigned from main storage and is specified by nnnnn$_1$K. Hierarchy 1 may be assigned either from main storage or from the large core storage device, IBM 2361 Model 1 or 2, and is specified by nnnnn$_2$K. If hierarchy 1 is assigned from main storage, the combined value of nnnnn$_1$K and nnnnn$_2$K cannot exceed the total storage available; if hierarchy 1 is assigned from the IBM 2361, nnnnn$_1$K cannot exceed 16383 and nnnnn$_2$K cannot exceed 1024 for an IBM 2361 Model 1 or 2048 for an IBM 2361 Model 2.

If storage is requested only from hierarchy 1, a comma is required to indicate the absence of nnnnn$_1$K (e.g., REGION=(,200K)).

An example of REGION is:

REGION=(100K,100K)

The example states that 200K bytes of storage are to be assigned, 100K from hierarchy 0 and 100K from hierarchy 1.

Assigning Storage to a Job Under VS

Storage can be allocated under the VS control programs by use of the ADDRSPC and REGION parameters.

The ADDRSPC parameter is used with the VS control programs to indicate whether a job or job step is to be run in real or virtual mode. This parameter takes the form ADDRSPC=REAL|VIRT, where REAL specifies real mode and VIRT specifies virtual mode. Most FORTRAN programs are run in virtual mode as described under "Operating System Control Programs" in the Introduction, and the default value for this parameter is ADDRSPC=VIRT. When ADDRSPC=REAL is coded on a JOB or EXEC card, all the pages of the job or job step in question are brought into real storage simultaneously and remain there for the duration of the job or job step.

The REGION parameter for the VS control programs takes the form REGION=valueK, where valueK indicates the number of contiguous 1024-byte areas of either real or virtual storage (depending on the mode of operation of the control program) to be allocated to the job or job step in question. There is no hierarchy support under VS.

Under VS1, in virtual mode, virtual storage is automatically allocated in partitions of installation-determined size in a manner analogous to the allocation of main storage under MFT. The REGION

parameter, if coded, is ignored when VS1 is run in virtual mode. When VS1 is run in real mode, the REGION parameter must be coded to indicate the number of contiguous 1024-byte areas of real storage to be allocated, and valueK may not exceed the total real storage available. If the REGION parameter is not coded, a default region of installation-determined size will be allocated.

Under VS2, in virtual mode, the REGION parameter may be used to specify the number of contiguous 1024-byte areas of virtual storage to be allocated, and valueK is limited only by the addressing range of the System/370. Under VS2 in real mode, the REGION parameter must be coded to indicate the number of contiguous 1024-byte areas of real storage to be allocated, and valueK may not exceed the total real storage available. If REGION is not coded, in either virtual or real mode, a default region of installation-determined size will be allocated.

For a further discussion of storage allocation under the VS control programs, see the publication OS/VS Job Management Services, Order No. GC28-0617.

EXEC STATEMENT

An EXEC statement indicates the beginning of a job step and names the program or cataloged procedure to be executed.

The parameter indicating the program or cataloged procedure to be executed is the only one required and must be the first one specified; all others are optional and may appear in any order.

The EXEC statement also contains the following information:

1. A job step name (a step name is required only when it is necessary for a later job step to reference information from this job step)
2. Compiler, linkage editor, or other options passed to the job step
3. Conditions for bypassing the execution of the job step
4. Accounting information relative to the job step
5. A time limit for the step or cataloged procedure
6. Main storage requirements

Figure I-5 illustrates the format of the EXEC statement. Table I-3 summarizes the function of the EXEC statement. Figure I-6 shows sample EXEC statements.

Name	Operation	Operand
//[stepname]	EXEC	Positional Parameter {[PROC=]procedure-name} {PGM=program-name} Keyword Parameters [PARM='option[,option]...' [ACCT=(accounting-information)] [COND=((code,operator[,stepname])(...)) [DPRTY=step-priority][1] [TIME=(minutes,seconds)][1] [REGION=region-size][2] [ADDRSPC=REAL|VIRT][3]

[1]MVT and VS2 only.
[2]MVT and VS only.
[3]VS only

Note: To indicate the step of a cataloged procedure to which an option applies, any of the keyword parameters can be followed by a period and the name of the step to be executed; e.g., PARM.procstep=option

Figure I-5. EXEC Statement Format

Table I-3. EXEC Statement Functions (Part 1 of 2)

Specification	Purpose	How to Specify
stepname	To permit other job steps to refer to: 1. The condition code resulting from this step, and 2. The data sets in this step.	One to eight alphameric characters, the first of which must be alphabetic or one of the extended alphabetic characters @, #, or $.
procedure-name	To name the cataloged procedure to be executed. A procedure-name initiates the processing of a series of job control statements that has been previously written in the system library, SYS1.PROCLIB, and cataloged in the system catalog.	procedure-name, or PROC=procedure-name, where procedure-name is the name of the cataloged procedure. See "Naming the Cataloged Procedure or Program to be Executed" for details.
program-name	To name the program to be executed.	PGM=program-name, where program-name is the name of a program. See "Naming the Cataloged Procedure or Program to be Executed" for details.
PARM	To specify program options.	PARM='option[,option]...', where option names a particular program option that is to be in effect during processing. See "Specifying Program Options" for details in how to specify options. See the sections "Compilation" and "Linkage Editor and Loader" for descriptions of available options.
ACCT	To identify accounting information relating to the job step.	ACCT=(value[,value]) where value indicates accounting information. See "Specifying Accounting Information" for details.
COND	To specify those condition codes from previous job steps that will cause this job step to be bypassed.	COND= ((code,operator[,stepname])(...)), where code is a number between 0 and 4095, operator is a two-character value that represents a comparison to be made between code and a return code issued by a preceding job step or the operating system, and stepname is the name of the preceding job step issuing the return code. See "Specifying Condition Codes to Bypass a Job Step" for details.

Table I-3. EXEC Statement Functions (Part 2 of 2)

Specification	Purpose	How to Specify
DPRTY	To assign a priority to the job step.	DPRTY=(step-priority,n), where step-priority is a number from 0 to 15 that the system converts into an internal priority, and n is a number from 0 to 15 that is added to the internal priority to establish a step priority.
TIME	To assign a time limit to the job step.	TIME=(minutes,seconds), where minutes is a number up to 1439, and seconds is a number up to 59.
REGION	To specify the amount of main storage to be allocated to a job step operating in the MVT environment.	For MVT, REGION=(nnnnn$_1$K[,nnnnn$_2$K]), where nnnnn$_1$K is a value up to 16383K and indicates the amount of storage to be allocated; nnnnn$_2$K is a value that usually indicates the amount of additional storage to be allocated from an IBM 2361 Core Storage Device. For an IBM 2361 Model 1, nnnnn$_2$K may be any value up to 1024K; from an IBM 2361 Model 2, nnnnn$_2$K may be any value up to 2048K. The combined value of of nnnnn$_1$K and nnnnn$_2$K may not exceed 16383. For VS, REGION=valueK, where valueK represents the number of contiguous 1024-byte areas of real or virtual storage to be allocated.
ADDRSPC	To specify the mode of operation, real or virtual, of the VS control program.	ADDRSPC=REAL for real mode, ADDRSPC=VIRT for virtual mode.

```
//STEPA   EXEC   PARM=(MAP,DECK)

//DO      EXEC   PROC=FORTXCLG,PARM.LKED=DECK,ACCT=248

//        EXEC   PGM=MYPROG,ACCT=NY12345,REGION=128K,TIME=3,
//               COND=(8,GT,HISPROG)
```

Figure I-6. Sample EXEC Statements

Naming an EXEC Statement

Stepname identifies the EXEC statement to other control statements and permits them to refer to information contained in the job step. Steps within cataloged procedures should be given unique names so that identification to the correct step can be easily made.

Naming the Cataloged Procedure or Program to be Executed

The PROC parameter is used to specify a cataloged procedure; PGM, to specify a program.

The format of PROC is:

PROC=procedure-name

where:

procedure-name
names the cataloged procedure to be executed.

The word PROC is optional; if it is omitted, PROC is assumed. For example, the following are equivalent:

```
// EXEC PROC=FORTXCLG
// EXEC FORTXCLG
```

Note that a cataloged procedure does not in itself execute a program; it permits previously written job control statements to be inserted into the job stream at the point that the procedure was called. The job control statements in the cataloged procedure should contain at least one EXEC statement having the PGM parameter which names the program to be executed.

The PGM parameter may specify any load module name accessible to the operating system. For example, PGM identifies the FORTRAN IV (H Extended) compiler as PGM=IFEAAB; PGM identifies the linkage editor as PGM=IEWL.

The format of PGM is:

```
      ⎧program-name                  ⎫
PGM=  ⎨*.stepname.ddname             ⎬
      ⎩*.stepname.procstep.ddname    ⎭
```

where:

program-name
names a program that resides in either the system library or a private library. The system library is a partitioned data set named SYS1.LINKLIB that stores frequently-used programs such as IFEAAB and IEWL. Private libraries are partitioned data sets that store groups of programs that are used by individual users.

*.stepname.ddname
names a program found on a data set defined in a previous job step of the current job. The * indicates the current job, "stepname" is the name of the job step, and "ddname" is the name of the DD statement defining the program. (The "stepname" cannot refer to a job step in another job.) The program referred to must be a member of a partitioned data set.

This form of a program name is most familiar to a FORTRAN programmer in an execution step following a link edit step. The linkage editor stores the load module in the data set defined by the SYSLMOD DD statement. For example, in the following statements, STEP4 executes the linkage editor. The linkage editor stores the resulting load module named ARCTAN into the partitioned data set, MATH, defined by the SYSLMOD DD statement. STEP5 indicates that the program to be executed (ARCTAN) is defined in the DD statement SYSLMOD in the job step named STEP4 of the current job.

```
//XYZ       JOB   ,JSMITH,COND=(7,LT)
                  .
                  .
                  .
//STEP4     EXEC  PGM=IEWL
//SYSLMOD   DD    DSNAME=MATH(ARCTAN)
                  .
                  .
                  .
//STEP5     EXEC  PGM=*.STEP4.SYSLMOD
```

*.stepname.procstep.ddname
names a program found on a data set defined in a previously executed step of a cataloged procedure. The * indicates the current job, "stepname" is the name of the job step that invoked the cataloged procedure, "procstep" is the name of a step within the cataloged procedure, and "ddname" is the name of a DD statement that defines the data set within that procedure step. Consider a cataloged procedure, FORT, containing the following statements:

```
//COMPFIL   EXEC  PGM=IExxx
//SYSPUNCH  DD    SYSOUT=B
//SYSPRINT  DD    SYSOUT=A
//SYSLIN    DD    DSNAME=LINKINP
                  .
                  .
                  .
```

```
//LKED      EXEC  PGM=IEWL
//SYSLMOD   DD    DSNAME=RESULT(ANS)
            .
            .
            .
```

Assume that the following statements appear in the input stream:

```
//MAINJOB   JOB   ,SMITH,COND=(7,LT)
//S1        EXEC  PROC=FORT
            .
            .
            .
//S2      EXEC  PGM=*.S1.LKED.SYSLMOD
//FT03F001  DD    UNIT=PRINTER
//FT01F001  DD    UNIT=INPUT
```

Statement S1 calls the cataloged procedure FORT.
nt S2 indicates that the program to be executed is found on a data set described by the DD statement SYSLMOD, located in the procedure step LKED of the cataloged procedure FORT, which was invoked by the statement S1. Consequently, the load module ANS in the data set RESULT is executed.

Specifying Program Options

The PARM parameter is used to specify program options applicable during execution of a job step. Program options increase the flexibility of a program by allowing the programmer to choose the form of input and output, the type of output, and otherwise tailor the program to his needs. A FORTRAN programmer may specify options for the FORTRAN compiler, the linkage editor, and the system loader.

See the sections "Compilation" and "Linkage Editor and Loader" for a discussion of these options.

If a PARM parameter is not specified, the program being executed assumes default values established at program installation time. When the user's installation receives a copy of the program product, it will contain the default options indicated in this publication. (Default values are shown underlined in this publication.) The user's installation can customize the compiler for its needs and default options may be permanently established at that time.

PARM has the format:

```
PARM='option[,option]...'
PARM.procstep='option[,option]...'
```

where:

procstep
: is used when a cataloged procedure has been specified in the PROC parameter. Specified PARM options are to apply to the job step in the cataloged procedure identified by procstep. For example, the following indicates that changes are to be made to the FORT step:

```
PARM.FORT='NOOBJECT'
```

option
: indicates either a keyword that has an intrinsic value to the program (e.g., LIST, SOURCE) or a keyword that is assigned a value by means of an equal sign or a pair of parentheses, for example, LINECOUNT(nn). The option field may contain up to 100 characters of information.

The field of PARM options may be enclosed in either apostrophes or parentheses according to the following rules:

1. If no individual option contains a special character (such as an equal sign or parenthesis), the programmer may enclose the field in either apostrophes or parentheses. For example, either of the following formats is valid:

   ```
   PARM='LIST,SOURCE,NODECK'
   PARM=(LIST,SOURCE,NODECK)
   ```

2. If an option contains a special character, the programmer may enclose either that option in apostrophes and the entire field in parentheses, or the entire field in apostrophes. For example, either of the following is valid:

   ```
   PARM=(LIST,'LINECOUNT(50)',SOURCE)
   PARM='LIST,LINECOUNT(50),SOURCE'
   ```

   (This publication adopts the convention of enclosing the entire field in apostrophes.)

3. If the PARM parameter is continued onto a following card, the programmer must enclose the entire field in parentheses and any individual options containing special characters in apostrophes. For example, only the following is valid:

```
PARM=(LIST,'LINECOUNT(50)',SOURCE,
     NODECK)
```

4. If only one option is specified, the programmer need not enclose the option in either apostrophes or parentheses (except if that option contains a special character, in which case apostrophes are mandatory). For example, any of the following is valid:

```
PARM=LIST
PARM='LIST'
PARM=(LIST)
```

Specifying Accounting Information

The ACCT parameter is used to specify accounting information for a job step.

Like the accounting parameter in the JOB statement, the field may contain up to 142 characters and is enclosed in apostrophes if it contains any special characters other than a hyphen. Unlike the JOB statement parameter, ACCT is a keyword parameter and may appear anywhere in the operand field.

ACCT has the format:

```
ACCT=(value[,value]...)
ACCT.procstep=(value[,value]...)
```

where:

procstep
: indicates in which step of a cataloged procedure the accounting information is to apply.

value
: indicates some accounting information, such as the account number to be charged for machine time. If value contains any special characters, it must be enclosed in apostrophes. If only one value is specified, it need not be enclosed in parentheses.

An example of ACCT is:

```
ACCT='NY432 777'.
```

Note that a blank is considered a special character, requiring the use of apostrophes.

Specifying Condition Codes to Bypass a Job Step

The COND parameter is used to specify which condition codes are to cause job step processing to be bypassed.

Like the COND parameter in the JOB statement, the parameter lists codes and operators to test the return code issued by the system. Unlike the JOB statement parameter, if the condition is met, the job step is not terminated but bypassed.

COND has the format:

```
COND=((code,operator[,stepname])(...))
COND.procstep=((code,operator[,stepname])
                (...))..))
```

where:

procstep
: is used when a cataloged procedure has been specified in the PROC parameter, to indicate in which step of the procedure the condition applies.

stepname
: indicates the name of a previous job step that issued the return code to be tested against the code in COND.

The meanings of code and operator are the same as described for the COND parameter of the JOB statement.

An example of COND is:

```
COND.GO=((4,LT,FORT)(4,LT,LKED))
```

The example states that if 4 is less than the return code issued by FORT or LKED, the job is to be terminated (any return code less than or equal to 4 allows the job to continue).

Assigning Step Priority

The DPRTY parameter is used to assign a dispatching priority to a job step. The dispatching priority determines the order in which job steps will use main storage and CPU (central processing unit) resources. If the parameter is omitted, each job step is assigned the same priority as the job, either through the JOB statement PRTY parameter or by default.

DPRTY has the format:

```
DPRTY=(step-priority,n)
DPRTY.procstep=(step-priority,n)
```

where:

procstep
is used when a cataloged procedure has been specified in the PROC parameter to indicate in which step of the procedure the dispatching priority applies.

step-priority
is a number from 1 through 15 representing a priority. This subparameter has the same meaning as the PRTY parameter in the JOB statement; that is, if PRTY=10 is coded in the JOB statement and DPRTY=10 is coded in the EXEC statement, job and step priority are the same. If the step priority is to be different from that assigned to the job, a different number is assigned. The step-priority is converted by the system into an internal priority; the higher the number the greater the priority. Whenever possible, the number 15 should be avoided; this number is reserved for certain system tasks.

n
is a number from 0 to 15 that is added to the internal priority to establish the dispatching priority.

If the first value is omitted but the second is specified, a comma is required to indicate the absence (e.g., DPRTY=(,12)). If the second value is omitted, the delimiting parentheses are not required (e.g., DPRTY=3).

An example of DPRTY is:

```
DPRTY=(10,9)
```

The example states that the number 10 is to be converted into an internal priority and the number 9 is to be added to the resulting priority.

Assigning a Time Limit to a Job Step

The TIME parameter is used to assign a time limit to the step. If not completed in the time specified, the step is terminated.

TIME has the format:

```
TIME=(minutes,seconds)
TIME.procstep=(minutes,seconds)
```

where:

procstep
is used when a cataloged procedure has been specified in the PROC parameter to indicate in which step of the procedure timing considerations apply.

If procstep is omitted, the time limit applies to the entire procedure.

The meanings of minutes and seconds are the same as described for the TIME parameter of the JOB statement.

An example of TIME is:

```
TIME.FORT=5
```

The example states that the FORT step of a cataloged procedure is to have a time limit of five minutes.

Assigning Storage to a Job Step Under MVT

Under MVT the REGION parameter is used to specify the amount of storage to be allocated to a job step. A REGION parameter specified on a JOB statement overrides any EXEC statement REGION parameters.

REGION has the format:

```
REGION=(nnnnn1K[,nnnnn2K])
REGION.procstep=(nnnnn1K[,nnnnn2K])
```

where:

procstep
is used when a cataloged procedure has been specified in the PROC parameter, to indicate in which step of the procedure storage allocation applies.

If procstep is omitted, the allocation applies to all steps of the procedure.

The meanings of nnnnn$_1$K and nnnnn$_2$K are the same as described for the REGION parameter of the JOB statement.

An example of REGION is:

```
REGION=100K
```

The example states that 100K bytes of storage are to be allocated to the job step.

Assigning Storage to a Job Step Under VS

Under the VS control programs, the REGION and ADDRSPC parameters can be used to specify the amount and type of storage to be allocated to individual job steps. The

REGION and ADDRSPC parameters specified on JOB statements override those specified on EXEC statements. Rules for coding them are the same as those described for the JOB statement.

DD STATEMENT

DD statements describe and identify data sets and the volumes in which they reside. They also provide instructions for their proper handling and disposition. DD statements supply information which is used by the job scheduler to allocate input/output devices and by data management to supervise the input/output operations.

The DD statement is concerned with the data set, records within the data set, and the location of the data set.

The DD statement specifies the following information:

1. A name for the DD statement
2. The location of the data set in the system's resources
3. The name of the data set
4. The status of the data set at the beginning and end of the job step
5. Labeling information for the data set volume
6. Data set allocation to optimize the use of input/output channels
7. The type of input/output device on which the data set resides
8. Space allocation for data sets on direct access devices
9. Characteristics of the records in the data set

Figure I-7 illustrates the format of the DD statement. Figure I-8 shows sample DD statements. Table I-4 summarizes the functions of the DD statement parameters.

Name	Operation	Operand
// {ddname / procstep.ddname}	DD	Positional Parameters [* DATA DUMMY] Keyword Parameters [{VOLUME / VOL}={SER=serial-number / REF=ddname}] [DDNAME=ddname {DSNAME / DSN}=data-set-name] [DISP=(subparameter-list) SYSOUT=x] [LABEL=(subparameter-list)] [COPIES=number][1] [SEP=(ddname[,ddname]...)] [DLM=delimiter][1] [UNIT=(device[,SEP=(ddname,...)])] [SPACE=(subparameter-list)] [DCB=(subparameter-list)]

[1]VS only.

Figure I-7. DD Statement Format

Example 1: Directing a data set to the printer:

```
//SYSPRINT  DD SYSOUT=A
```

Example 2: Creating a temporary data set (created and deleted within the same job):

```
//FT14F001  DD  DSNAME=&&TEMP,UNIT=SYSSQ
```

Example 3: Creating a permanent data set:

```
//FT89F001  DD  DSNAME=MINE,VOLUME=SER=8342,UNIT=2400,
//              DISP=(NEW,KEEP),LABEL=(,SL,EXPDT=71365)
```

Example 4: Creating a permanent cataloged data set:

```
//FT31F001  DD  DSNAME=MATRIX,DISP=(NEW,CATLG,DELETE),
//              UNIT=2311,VOLUME=SER=AA69,SPACE=(TRK,(50,5,ROUND)),
//              DCB=(RECFM=FB,LRECL=604,BLKSIZE=1208)
```

Example 5: Retrieving the permanent uncataloged data set defined in Example 3:

```
//FT89F001  DD  DSNAME=MINE,VOLUME=SER=8342,UNIT=2400,DISP=(OLD)
```

Example 6: Retrieving the cataloged data set defined in Example 4:

```
//FT37F001  DD  DSNAME=MATRIX,DISP=(OLD)
```

Figure I-8. Sample DD Statements

Table I-4. DD Statement Functions (Part 1 of 3)

Specification	Purpose	How to Specify
ddname	To identify the DD statement to other job control statements that may need to refer to it.	One to eight alphameric characters, the first of which must be alphabetic or one of the extended alphabetic characters #, @, or $.
procstep.ddname	To identify the DD statement in the particular jobstep identified as procstep.	Procstep and ddname are each specified as one to eight alphameric characters, the first of which must be alphabetic or one of the extended alphabetic characters. Procstep and ddname are separated by a period (.).
*	To indicate that the data set appears in the input stream	When * is used, • The data set must appear immediatedly following the DD * statement; • No other parameter except the DCB parameter has meaning on the DD statement. See "Data Set Location" for details.
DATA	To indicate that the data set appears in the input stream and contains job control statements that are to be read as data and not as instructions. Used, for example, when job control statements are being entered into the system catalog as a cataloged procedure.	When DATA is used, • The data set must appear immediately following the DD DATA statement; • No other parameter except the DCB parameter has meaning on the DD statement. See "Data Set Location" for details.
DLM	To specify a delimiter other than /* to terminate a group of data in the input stream.	DLM=delimiter, where delimiter is any combination of two characters that will indicate the end of a group of data in the input stream.
DUMMY	To identify a data set on which no operations are to be performed (such as to defer processing a data set in a program segment that has already been tested).	See "Data Set Location" for details.
VOLUME	To identify the volume in which the data set resides.	VOLUME=SER=serial-number, where serial-number consists of one through six alphameric characters identifying the volume, or VOLUME=REF=ddname, where ddname is the name of another DD statement and indicates that the data set is to share the same volume as the data set defined in ddname. See "Data Set Location" for details.

Table I-4. DD Statement Functions (Part 2 of 3)

Specification	Purpose	How to Specify
DDNAME	To indicate that the data set is to have the same characteristics as a data set defined in another DD statement.	DDNAME=ddname, where ddname is the name of another DD statement or the characteristics of the data set defined in ddname.
DSNAME	To name the data set or a member of a partitioned data set.	DSNAME=data-set-name, where data-set-name is the name of a permanent data set, a temporary data set, a member of a partitioned data set, or a reference to a data set defined in another DD statement. See "Data Set Identification" for details.
DISP	To indicate whether the data set is new or old, and whether it is to be kept or released at the end of the job step.	DISP=(subparameter-list), where subparameter-list indicates: • The disposition of the data set at the beginning of the job step (i.e., whether new or old) • The disposition to be made of the data set at the end of the job step (i.e., whether to be kept, deleted, or passed to another job step), • The disposition to be made of the data set if abnormal termination occurs (i.e., whether to be kept or deleted). See "Data Set Disposition" for details.
SYSOUT	To assign the output of the data set to an output class.	SYSOUT=x, where x is an alphabetic or numeric character assigning the output class represented by the character to the data set. SYSOUT=A is usually specified for printer output, SYSOUT=B for card punch output. See "Data Set Disposition" for details and VS options.
COPIES	To obtain between 1 and 255 hard copies of the output data set.	COPIES=number, where number is between 1 and 255.
LABEL	To assign such information as labels, whether the data set is protected from unauthorized processing, and how long the data set should be kept.	See "Data Set Labels" for details.
SEP	To assign the data set to a different input/output channel from the data set defined in ddname.	SEP=(ddname[,ddname]...), where ddname is the name of another DD statement.

Table I-4. DD Statement Functions (Part 3 of 3)

Specification	Purpose	How to Specify
UNIT	To identify the input/output device or device class on which the data set resides.	UNIT=(device[,SEP=(ddname,...])), where device is a number or name identifying the device and ddname is the name of another DD statement. SEP=ddname... is used only for data sets on direct access devices and only when the data set is not to share device access arms with the data set defined in ddname. See "Assigning a Data Set to an Input/Output Device" for details.
SPACE	Used only for a data set in a direct access volume, to allocate space to the data set.	SPACE=(subparameter-list), where subparameter-list indicates whether space is to be allocated to a specific address or to any location in the direct access volume, and the amount of space to be allocated. See "Assigning Space to a Data Set on a Direct Access Volume" for details.
DCB	To identify the characteristics of the records in a data set.	See "Defining Record Characteristics" for details.

DD Statement Uses

DD statements define the following types of data sets:

1. System data sets. These include SYSIN, SYSPRINT, SYSPUNCH, SYSLIN, SYSLMOD, SYSUT1, and SYSUT2. These data sets are used by various system components as work areas and temporary storage areas, and are necessary for the execution of the compiler, linkage editor, and loader.

2. FORTRAN load module data sets. If the FORTRAN programmer uses cataloged procedures and the standard data set reference numbers (5 for card input, 6 for printer output, and 7 for punched-card output), there is no need to supply DD statements for those data sets; they are contained in the cataloged procedures.

   If the FORTRAN programmer uses other forms of input/output, such as magnetic tape or disk, it is necessary to define those data sets with DD statements. The DD statements are equated to the data set reference numbers through the DDNAME parameter which identifies the DD statement. The ddname format for data sets used in FORTRAN load module execution is:

   FTxxFyyy

   where xx is the data set reference number and yyy is a FORTRAN sequence number (usually 001). The data set reference number is nothing more than a numeric means of equating FORTRAN input/output statements with the proper data set definition; it has nothing to do with the physical address of the device or its type.

3. A sequential data set on which a dump can be written in the event of an abnormal termination. Definition of this data set automatically requests the dump facility; if no data set has been defined for dump output, it is bypassed. A ddname of SYSUDUMP specifies a dump of the problem program area. A ddname of SYSABEND

specifies a dump of the problem program area and the system nucleus.

4. Concatenated data sets (data sets temporarily joined together), usually consisting of one or more private libraries and the system library SYS1.LINKLIB. In other words, the system library and the specified libraries will be combined temporarily to form one library. A ddname of STEPLIB will retain the concatenated library for the duration of the job step; a ddname of JOBLIB will retain the concatenated library for the duration of the job.

In theory, all DD statement parameters are optional; that is, no one parameter is required for all DD statements. In practice, however, some parameters are required to describe a function properly. For example, to create a permanent data set, the DISP, UNIT, and DSNAME parameters are required; to create a permanent data set on a particular direct access volume, the VOLUME and SPACE parameters are also required.

Naming a DD Statement

The ddname identifies the statement to other control statements and relates the data set to I/O statements in the FORTRAN source module. The name may be a qualified name with the format procstep.ddname to associate the DD statement with a cataloged procedure job step, such as FORT.SYSIN.

Data Set Location

The *, DATA, or VOLUME parameters are used to specify a data set's location. The DUMMY parameter is used to inhibit I/O operations on a data set.

The * parameter defines a data set in the input stream, usually a card deck or a data set in card image form. An example of the * parameter is:

```
//DSET1 DD *
```

The data set is physically placed after the DD * statement, and it must be followed by a /* statement to denote the end of the data set.

The DATA parameter also defines a data set in the input stream. It is used in place of the * when the data set contains records having the characters // in columns 1 and 2.

An example of DATA is:

```
//DSET2 DD DATA
```

When either the * or the DATA parameter is used, no other operand in the DD statement has meaning except the DCB parameter which may specify block and buffer information for the data set, e.g.,

```
DCB=(BLKSIZE=800,BUFNO=2)
```

(The DCB parameter is discussed in the section "Defining Record Characteristics.")

In addition, under VS, the DLM parameter may be coded with the * or DATA parameter to allow the programmer to specify a delimiter other than /* to terminate a group of data in the input stream. By assigning a different delimiter in the DLM parameter, the programmer can include a standard delimiter (/*) as data in the input stream.

The DLM parameter has the format:

```
DLM=delimiter
```

where delimiter is any combination of two characters that will indicate the end of a group of data in the input stream. (If the delimiter contains any special characters, it must be enclosed in apostrophes.)

An example of the use of the DLM parameter is:

```
//DD1  DD  *,DLM=AA
          .
          .
        Data
          .
          .
AA
```

This example shows the DLM parameter being used to assign the characters AA as the valid delimiter for the data defined in the input stream by DD1.

VOLUME specifies the location of a data set not in the input stream (i.e., it is used for data sets residing on tape or in direct-access volumes). The parameter is required when creating a new data set if the programmer wants the data set assigned to a specific volume; if the parameter is omitted, the data set is assigned to any available volume. The parameter is required when retrieving an existing data set except if the data set has been cataloged. The VOLUME parameter contains many subparameters. Two subparameters most useful to the FORTRAN programmer are SER and REF. SER specifies the volume serial number. REF specifies the name of another data set or the name of another DD

statement and indicates that more than one data set is to share the same volume.

The format of VOLUME=SER is:

VOLUME=SER=serial-number

where:

serial-number
is a 1 to 6 character serial number.

An example is:

VOLUME=SER=DA1234

The format of VOLUME=REF is:

VOLUME=REF= { dsname | *.ddname | *.stepname.ddname | *.stepname.procstep.ddname }

See Table I-5 for an explanation of the data set name formats.

An example of VOLUME=REF is:

VOL=REF=*.DDNAM

The example states that the data set is to share the same volume as the data set described on the DD statement DDNAM in the current job. Note that VOLUME may be abbreviated VOL.

For a discussion of the other subparameters available in the VOLUME parameter, see the appropriate job control language reference publication, as listed in the Preface.

The DUMMY parameter is used to prevent input/output operations on the data set. A WRITE statement is recognized but no data is transmitted. A READ statement is recognized but permits further processing only if the END= option is specified; if the option is not specified, a read causes an end of data set condition and termination of load module execution.

Data Set Identification

DSNAME and DDNAME are used to identify a data set.

DSNAME specifies either the name of the data set or the name of an earlier defined DD statement that identifies this data set. DDNAME specifies the name of a DD statement to be defined later which is to identify this data set.

DSNAME has the format:

{DSNAME / DSN} = { dsname | dsname(member) | &&dsname | &&dsname(member) | *.ddname | *.stepname.ddname | *.stepname.procstep.ddname }

See Table DD3 for an explanation of the data set name formats.

Examples of DSNAME are:

DSNAME=DSET
DSN=LIB(PROG1)
DSNAME=*.FORT.SYSLIN

The first example states that the data set name is DSET. The second example states that the data set is a partitioned data set named LIB and that the DD statement defines the member named PROG1. The third example states that the data set is the one defined in the DD statement named SYSLIN occurring in the job step named FORT. (This last example is how loader cataloged procedures define the object module resulting from the compilation job step.)

DDNAME has the format:

DDNAME=ddname

where:

ddname
is the name of another DD statement.

When this parameter is used, no other parameter except the DCB parameter may be specified on the DD statement.

An example of DDNAME is:

DDNAME=FT08F001

The example states that data set definition is to be found on the DD statement named FT08F001.

Table I-5. Data Set Names

Format	Description	Example
dsname[1]	A data set named dsname	NAME
dsname(element)[2]	A member of a partitioned data set	MYPROG(PROGA)
&&dsname[2]	A temporary data set, to be deleted at the end of the job	&&TEMP
&&dsname(element)[2]	A member of a temporary partitioned data set	&&TEMPP(PROG)
*.ddname[1]	A data set defined in another DD statement within the current job step	*.SYSPRINT
*.stepname.ddname[1]	A data set defined in another DD statement but in an earlier job step named stepname of the current job	*.STEPA.SYSPRINT
*.stepname.procstep.ddname[1]	A data set defined in a DD statement in the cataloged procedure step named procstep called by the job step named stepname.	*.STEPA.LKED.SYSPRINT

[1]This format may appear in the DSNAME, VOLUME, and DCB parameters
[2]This format may appear in the DSNAME parameter only

Data Set Disposition

The SYSOUT and DISP parameters indicate the status of a data set.

SYSOUT directs data set output to a unit record device such as a printer or a card punch device class. Table I-6 summarizes device classes.

The format of SYSOUT is:

SYSOUT=x

where:

x
is an alphabetic or numeric character that assigns the data set to the device class represented by the character.

An example of SYSOUT is:

SYSOUT=B

The example states that the data set is to be directed to the device class B, usually a card punch.

Under the VS1 and VS2 control programs, the programmer has the additional options of using the SYSOUT parameter to specify a program in the system library which will write the output data set, as well as a special output form on which the data set is to be printed or punched.

The format of the VS SYSOUT parameter is:

SYSOUT=(x[,program name][,form number])
[,]

where:

x
is as described above.

,program name
is the member name of a program in the system library (other than the system output writer) that is to write the output data set to a unit record device.

,
specifies that the system output writer is to write the output data set to a unit record device, and a form number follows.

,form number
is from 1 through 4 alphabetic or numeric characters (including @, $, and #) which specify that the output

data set is to be printed or punched on a special output form.

The parentheses are optional if the program name and form number are omitted.

An example is:

```
SYSOUT=(F,,7402)
```

The example specifies that the data set is to be written by the system output writer to a unit record device corresponding to class F. The data set is to be printed on a special form. The form number is 7402.

Under VS, the programmer may also use the COPIES parameter in conjunction with the SYSOUT parameter to obtain between 1 and 255 copies of the output data set.

The format of the COPIES parameter is:

```
COPIES=number
```

where number may be between 1 and 255.

An example is:

```
//RECORD  DD SYSOUT=W,COPIES=32
```

The example is a request for 32 copies of the data set RECORD to be produced by a unit record device corresponding to class W.

If fewer than 1 or more than 255 copies are specified, or if the COPIES parameter is coded without an associated SYSOUT parameter, the job will be canceled.

The DISP parameter indicates the status of a data set not on a unit record device. It contains three subparameters, the first describes the data set status at the beginning of the job step, the second the disposition of the data set at the end of the step, and the third the data set's disposition if an abnormal termination occurs.

The format of DISP is:

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

where:

NEW
: describes a data set that is being created in the current job step. If the status is NEW, that subparameter may be omitted (NEW is the assumed value) by indicating its absence with a comma.

OLD
: describes a data set that existed before the current job step.

SHR
: describes an existing data set which resides on a direct-access volume that is also available to other concurrently-operating jobs.

MOD
: describes a sequential or partitioned data set that is to be added to. Before the first input/output operation occurs, the data set will be automatically positioned after the last record.

KEEP
: specifies that the data set is to be retained for future use in other jobs. KEEP is the default value for old data sets.

DELETE
: specifies that the space occupied by this data set is to be released and made available for other uses. If the data set was cataloged, the entry for it will be removed from the catalog. If the data set resides on a direct-access volume, the entry for the data set will be removed from the volume table of contents. Once this disposition has taken place, the data set will have ceased to exist. DELETE is the default value for new data sets.

CATLG
: specifies that a catalog entry that points to the data set is to be entered in the system catalog. The data set can then be referred to by name in subsequent jobs or job steps. CATLG implies KEEP.

UNCATLG
: specifies that the catalog entry that points to the data set is to be removed from the system catalog. UNCATLG implies KEEP. If the data set resides on a direct-access volume, the entry for the data set will remain in the volume table of contents. The area occupied by the data set is still reserved for the data set, and is not released.

PASS
: specifies that the data set is to be used in a later job step. A DD statement in a later job step can reference the data set using the DSNAME parameter format *.stepname.ddname. The final disposition of the data set should be given in the last job step that uses

the data set. When a data set is in the PASS status, the volume on which it resides remains mounted.

If the third subparameter is not specified, the disposition is the same as that specified in the second subparameter; if the second subparameter specifies PASS, the status of the data set reverts to the status it had before the job step that passed it. In other words, if the data set had an initial disposition of OLD, MOD, or SHR, the data set is kept; if it had an initial disposition of NEW, it is deleted.

Examples of the DISP parameter are:

```
DISP=(NEW,CATLG,DELETE)
DISP=(OLD,DELETE,KEEP)
DISP=SHR
```

The first example specifies a new data set that is to be cataloged; if abnormal termination occurs, information in the data set is considered useless and the data set is deleted. The second example specifies an existing data set that is to be deleted; if abnormal termination occurs, information in the data set is needed and the data set is retained so that it may be resubmitted. The third example specifies an existing data set that is to be shared with other jobs; it is implicitly retained. Note that if only the first subparameter is specified the delimiting parentheses may be omitted.

Table I-6. Device Class Names

Class Name	Class Function	Device Type
SYSSQ	Reading, Writing (sequential)	Magnetic Tape Direct Access
SYSDA	Reading, Writing, Updating (direct)	Direct Access
A	Printed Line Output	Printer Magnetic Tape Direct Access
B	Card Image Output	Card Punch Magnetic Tape Direct Access

Assigning a Data Set to an Input/Output Device

The UNIT parameter assigns a data set to a device. The device may be identified by its identification number (e.g., 2400 for tape, 2311 for a disk), or by its device class name (e.g., SYSSQ for devices containing sequential data sets). A device class is a group of input/output devices performing similar functions, which is given a collective name. Device class names are assigned at program installation time. See Table I-6 for a summary of device class names supplied by IBM.

For data sets residing on direct access devices, the UNIT parameter may also specify those data sets that are not to share the same device access arms, thereby increasing operating efficiency for data sets whose input/output operations occur at the same time; these other data sets are identified by naming the DD statements defining them.

The format of the UNIT parameter most applicable to FORTRAN programs is:

$$\text{UNIT} \left(\left\{ \begin{matrix} \text{device-type} \\ \\ \text{group-name} \end{matrix} \right\} [,\text{SEP=(ddname,...)}] \right)$$

where:

device type
: identifies an input/output unit by its device number. For example, to request a 2400 Magnetic Tape Device, UNIT=2400 is specified; to request a 2311 Disk Storage Unit, UNIT=2311 is specified. See "Appendix C: Unit Types" for a list of device types that can be specified in the UNIT parameter.

groupname
: identifies a device class name, such as SYSSQ.

ddname
: identifies a DD statement defining a data set that is to be on a separate device access arm from the data set defined in the current DD statement. Up to eight ddnames may be specified.

Examples of UNIT are:

```
UNIT=SYSDA
UNIT=(2311,SEP=DDNAME1,DDNAME3)
```

The first example states that the data set may be assigned to any unit in the SYSDA device class. Note that parentheses are not required if only one value is specified. The second example states that the data set is to be assigned to a 2311 device and is to use different access arms from the data sets described in DD statements DDNAME1 and DDNAME3.

For a discussion of the other subparameters available in the UNIT parameter, see the appropriate job control

language reference publication, as listed in the Preface.

Assigning Space to a Data Set on a Direct Access Volume

The SPACE parameter is used to allocate space to data sets residing on direct access volumes. This parameter is required when defining a new data set on a direct access volume.

To assign space anywhere within a volume, the SPACE parameter has the format:

```
SPACE=( {TRK       }
        {CYL       }
        {block-length}

       (,primary[,secondary][,directory])
       [,RLSE]
       [,CONTIG]
       [,ROUND])
```

where:

TRK
: specifies that space is to be allocated in number of tracks.

CYL
: specifies that space is to be allocated in number of cylinders.

block-length
: is a number indicating the average length of a block of records in the data set; the system is to allocate space according to the block length specified.

primary
: is a number indicating the amount of tracks, cylinders, or blocks to be allocated.

secondary
: is a number indicating the amount of tracks, cylinders, or blocks to be allocated if additional space is required. Additional space is allocated up to fifteen times.

directory
: is a number indicating the amount of blocks of 256-byte areas to be allocated for the directory of a partitioned data set.

RLSE
: is a keyword indicating the unused space may be released at the end of the job step.

CONTIG
: is a keyword indicating that space is to be assigned contiguously within the volume.

ROUND
: is a keyword indicating that space to be allocated must be equal to one or more cylinders.

Examples of this form of the SPACE parameter are:

```
SPACE=(TRK,(10,10,2))
SPACE=(400,(5,5),CONTIG,RLSE)
```

The first example states that space is to be allocated in tracks; 10 tracks are to be allocated initially; 10 additional tracks are to be allocated as needed; additional space is to be reserved for 2 records of a directory.

The second example states that space is to be allocated according to the size of each block of records (400 bytes). Space to accomodate five blocks is to be allocated initially. Space to accomodate five more blocks is to be allocated as needed up to a maximum of 15 times. Space is to be assigned contiguously. Unused space may be released at the end of the job step.

Data Set Labels

The LABEL parameter is used to specify label information for a data set. Labels are used by the control program to store information about the data set and to identify the volume on which it is contained. Data sets on direct access volumes always have labels, data sets on magnetic tape volumes usually have labels, and data sets on unit record devices never have labels.

The LABEL parameter consists of four positional subparameters and one keyword subparameter (any subparameter not specified must have its position noted by a comma, except the last subparameter).

The first subparameter indicates the position of the data set relative to other data sets sharing the volume (tape only). The second subparameter identifies the type of label associated with the data set. The third specifies whether the data set is protected against unauthorized processing. The fourth specifies whether the data set is to be processed for input or output operations only. The fifth specifies a date when the data set may be released.

LABEL has the format:

```
LABEL=( [sequence number]
        [,              ]

        [,SL  ]        [,PASSWORD]  [,IN  ]
        [,NL  ]        [,        ]  [,OUT ]
        [,AL  ]                     [,    ]
        [,BLP ]
        [,    ]

        [,EXPDT=yyddd]
        [,RETPD=nnnn ]
```

where:

sequence-number
: is used only for data sets residing on magnetic tape to specify which data set is to be processed in a volume containing more than one data set (e.g., 3 specifies the third data set). The subparameter may consist of up to four digits. If the data set is the first or only data set in the volume, the subparameter need not be coded; the default value is 1.

: specifies the absence of a subparameter when following subparameters are specified.

SL
: specifies that the data set has standard system-created labels.

NL
: specifies that the data set has no labels.

AL
: specifies that the data set has ASCII labels (American National Standard Code for Information Interchange, a system of coding computer-processed characters differently from the IBM standard EBCDIC). ASCII data sets can reside only on magnetic tape.

BLP
: specified that label processing for the data set is to be bypassed.

PASSWORD
: specifies that the data set is protected by a password. In order to access the data set the operator must issue the correct password on the system console. Password-protected data sets must have standard labels.

IN
: specifies that the data set is to be processed for input operations only. IN will be recognized only if the first input/output operation is a READ. If it is not a READ, IN is ignored and both input and output operations are permitted; if it is a READ, any subsequent WRITE will be treated as an error and job processing is terminated. IN also permits a password-protected data set to be read (if the correct password is supplied) and avoids the need of operator intervention when reading a data set having a high expiration date (for an explanation of the expiration date, see the description of EXPDT). IN must be specified to read a partitioned data set or number.

OUT
: specifies that the data set is to be processed for output operations only. OUT will be recognized only if the first input/output operation is a WRITE. If it is not a WRITE, OUT is ignored and both input and output operations are permitted. If it is a WRITE, any subsequent READ will be treated as an error and job processing is terminated. OUT must be specified to create a partitioned data set or member and WRITE operations only may be specified.

EXPDT=yyddd
: specifies a date when the data set can be deleted. The date is expressed as a 2-digit number for the year and a 3-digit number for the number of the day in the year (e.g., 72100 indicates that the data set may be released on the 100th day of the year 1972).

RETPD=nnnn
: specifies the number of days that the data set is to be kept.

Examples of LABEL are:

```
LABEL=(3,SL)
LABEL=(,SL,,OUT,EXPDT=71196)
```

The first example states that the third data set of a magnetic tape volume has standard labels. The second example specifies a data set with standard labels which is used only for output operations and which may be released on the 196th day of 1971.

Assigning Channel Use

The SEP parameter indicates a data set that is to be assigned to a different channel from other data sets.

A channel is a small control unit that manages data transmission operations between a number of input/output devices

and the system's central processing unit. Processing time may be shortened by the use of separate channels; assigning data sets whose input/output operations occur at the same time to separate channels increases the speed of input/output operations.

SEP has the format:

```
SEP=(ddname,...)
```

where:

ddname
: is the name of a DD statement whose data set is not to share the same channel as the data set being defined. Up to eight ddnames may be specified. If only one ddname is specified, the enclosing parentheses are not required.

An example of SEP is:

```
SEP=(NAMEA,NAMEB,NAMED)
```

The example states that the data set being defined is to be assigned to a separate channel from the data sets defined on DD statements NAMEA, NAMEB, and NAMED.

Defining Record Characteristics

The DCB parameter defines characteristics of records in a data set. The parameter may specify the following:

- Record format, i.e., whether records are fixed-length (all records in the data set are of equal length), variable-length (records are of different length), or undefined-length (no record length information has been stated).
- Record length. For variable-length records, the record length specifies the length of the largest record.
- Block information, i.e., whether records are accessed individually (unblocked) or in groups (blocked), and the block size of blocked records.
- The number of buffers to be assigned to a data set when data is transmitted between system devices.
- Information identifying ASCII data sets.
- Options to describe characteristics of data sets residing on tape.
- Options to describe characteristics of data sets residing on direct access devices.
- Characteristics of data sets to be taken from another data set.

If the parameter is not specified, default values are supplied according to the data set description presented by other parameters in the DD statement. Table C1 in the section "Compilation" lists DCB defaults for compiler data sets; Table I-12 in the section "Load Module Execution" lists DCB defaults for load module data sets.

The format of the DCB subparameter is:

```
DCB=([data-set-name]
     [,RECFM=record-format]
     [,LRECL=record-length]
     [,BLKSIZE=block-length]
     [,BUFNO=number-of-buffers]
     [,BUFOFF=block-prefix]
     [,DEN=tape-density]
     [,TRTCH=tape-recording-technique]
     [,DSORG=direct-access-organization]
     [,OPTCD=optional-services]
```

The data-set-name subparameter indicates that DCB attributes are to be taken from a data set defined earlier. Other DCB subparameters may be coded on the same DD statement to override any of the attributes taken from the earlier data set. The data-set-name subparameter is specified as:

```
data-set-name= { dsname                          }
               { *.ddname                        }
               { *.stepname.ddname               }
               { *.stepname.procstep.ddname      }
```

The form that data-set-name may take is summarized in Table I-5.

An example of dsname is:

```
DCB=*.STEP1.SMITH2
```

The example states the data set is to have the same characteristics as the data set defined in the DD statement SMITH2 appearing in job step STEP1.

RECFM is specified as follows:

```
RECFM= [F]      [B]  [S]  [A]  [T]
       [V]           [M]
       [D]
       [U]
```

where:

F
: indicates fixed-length records.

V
indicates variable-length EBCDIC records.

D
indicates variable-length ASCII records.

U
indicates undefined-length records.

B
indicates that records are blocked. B may not be specified for undefined-length records.

S
indicates spanned records, i.e., a record spans over two or more blocks. S may be specified only for variable-length EBCDIC records.

A
indicates that carriage control characters are used for formatting purposes.

M
indicates that machine code control characters are used. M may not be coded for ASCII records.

T
indicates that the track overflow feature is to be used. This feature permits records to be written even though the block size exceeds the track size of a direct access device. This feature results in more efficient track utilization.

Examples of RECFM are:

```
RECFM=V
RECFM=FBA
```

The first example specifies variable-length records. The second example specifies fixed-length records which are blocked and written according to carriage control characters.

LRECL is specified as:

```
LRECL=record-length
```

where:

record-length
is a number indicating the length of the largest record to be found in the data set. The maximum value that may be specified is 32756.

For fixed-length records, LRECL indicates the actual length of the record. For variable-length records, LRECL indicates the length of the longest record, plus four for a segment control word. A four-byte Segment control word precedes each variable-length record and specifies the actual length of the record. For undefined records LRECL is omitted.

An example of LRECL is:

```
LRECL=84
```

This example states that the record length is 84 bytes.

BLKSIZE is specified as:

```
BLKSIZE=block-length
```

where:

block-length
is a number indicating the length of the block of records. The number specified also determines the length of the buffer, i.e., the number of bytes of data transmitted between an I/O unit and main storage in one operation. The maximum value that may be specified is 32760.

For fixed-length unblocked records, BLKSIZE is the same as the number specified in LRECL (LRECL may be omitted; if it is, the operating system determines the record length from BLKSIZE). For fixed-length blocked records, BLKSIZE is an integral multiple of LRECL.

For variable-length unblocked records, BLKSIZE is equal to LRECL plus four for a block control word. A four-byte block control word precedes each block of variable-length records (whether the block contains a single unblocked record or a number of blocked records) and specifies the actual number of bytes contained in the block (including the length of the records, the segment control words preceding each record, and the block control word itself). For variable-length blocked records, data management will place records being written in an output block until there is no more room left for the next record and segment control word. It will then write the completed block (which does not have to be as large as BLKSIZE) and will start a new one with the new record.

For undefined-length records, BLKSIZE should be specified to indicate the largest record that might be encountered.

An example of BLKSIZE is:

```
BLKSIZE=300
```

This example states that the block size is 300 bytes.

BUFNO is specified as:

BUFNO=n

where:

n
is a number indicating how many buffers are to be assigned to the data set. One, two, or three buffers may be assigned. (Three buffers may be specified only with asynchronous I/O.) If the subparameter is omitted, two buffers are assigned. If the programmer specifies a number greater than three, three buffers are assigned.

An example of BUFNO is:

BUFNO=1

This example states that only one buffer is to be assigned to the data set.

BUFOFF is used with ASCII data sets to indicate the size of an optional block prefix. The block prefix is a field that, if specified, precedes each unblocked record or the first record in a block and contains information describing the record or block. It may be up to 99 bytes in length, and may be specified for fixed-length, undefined-length, and variable-length records. No use is made of the block prefix by IBM System/360 Operating System except in certain cases where the information contained in it is used to determine the length of a block of variable-length records.

BUFOFF is specified as:

BUFOFF= {n / L}

where:

n
is a number from 0 to 99 indicating the size of the block prefix. BUFOFF=n may be specified only for input data sets; the operating system makes no use of the block prefix but skips the number of bytes specified before beginning record processing. If specified for output data sets, BUFOFF=n causes an error message to be written with abnormal termination resulting (unless the extended error handling facility is in force).

L
indicates that the block prefix is four bytes long and that it is to be used to compute the block size. BUFOFF=L may be specified for both input and output data sets but only for variable-length records, that is, when RECFM=D (or RECFM=DB) is also specified.

The BUFOFF subparameter is optional; if not specified, BUFOFF defaults to 0.

An example of BUFOFF is:

BUFOFF=40

This example states that 40 bytes are to be skipped before record processing is to begin.

The DEN and TRTCH subparameters define characteristics of data sets residing on magnetic tape.

DEN indicates the recording density of a tape volume. It is specified as:

DEN=density

where:

density
is the number 0, 1, 2, or 3. The number 0 indicates density of 200 bits per inch of tape (bpi); 1 indicates 556 bpi; 2 800 bpi; and 3 1600 bpi.

Data may be stored on seven-track tape or nine-track tape. Seven-track tape may be written 200 bpi, 556 bpi, or 800 bpi. Nine-track tape may be written 800 bpi or 1600 bpi. DEN is optional; if it is not specified, 800 bpi is assumed for both seven-track and nine-track tape.

An example of DEN is:

DEN=3

The example states that the data set has a density of 1600 bpi.

TRTCH is used for seven-track tape to indicate the recording technique to be used.

TRTCH has the format:

TRTCH= {C / E / T / ET}

where:

C
: indicates the data conversion feature. The data conversion feature converts binary data between seven-track tape and eight-bit main-storage. Because of the difference in the number of bits per character (seven-track tape has six data bits and one parity-check bit), four tape characters are stored as three main-storage bytes.

E
: indicates even parity. Parity is a technique that determines whether any bits were lost during data transmission by counting the number of bits in each character. The mode may require the parity check to be even or odd; the default value is odd parity.

T
: indicates data translation from BCD to EBCDIC.

ET
: indicates even parity and data translation.

If this subparameter is not specified, the default value is odd parity with no conversion or translation.

DSORG defines the organization of a direct-access data set. As applicable to a FORTRAN program, it is specified as:

DSORG= {DA / PS}

where:

DA
: indicates direct access organization. DA should be specified for direct-access data sets that will be processed by non-FORTRAN programs; this specification causes a label to be created indicating that the data set has direct organization.

PS
: indicates physical sequential organization. PS is specified for direct access data sets that will be processed only by FORTRAN programs.

OPTCD indicates optional services to be performed by the control program. As applicable to a FORTRAN program, it is specified as:

OPTCD= {Q / C}

where:

Q
: indicates an unlabeled ASCII data set. If an ASCII data set has been specified with a label (LABEL=(,AL)), this option need not be specified.

C
: indicates chained scheduling. Chained scheduling is a technique whereby the control program receives several separate read or write operations as one continuous operation; in a program having extensive input/output operations, chained scheduling may result in a significant reduction in processing time, particularly when creating a FORTRAN DA data set.

Examples of the DCB parameter are:

```
DCB=(RECFM=F,BLKSIZE=100,DEN=2)
DCB=(RECFM=VB,LRECL=54,
     BLKSIZE=850,BUFNO=1)
```

The first example describes fixed-length unblocked records 100 bytes long on tape written in 800 bpi. The second example describes variable-length blocked records with a maximum record size of 50 bytes plus four for segment control word, a maximum block size (including record control words and block control word) of 850, and one buffer to transmit this data.

The FORTRAN IV (H Extended) compiler is a processing program that translates a FORTRAN IV source module into an object module.

The EXEC statement calls the compiler by name, IFEAAB, in the PGM parameter, i.e.,

```
//  EXEC  PGM=IFEAAB
```

The EXEC statement may also specify other parameters and compiler options.

COMPILER OPTIONS

Compiler options are specified in the PARM parameter of the EXEC statement. They increase the flexibility of the compiler. For example:

- The SOURCE option lists the statements in the source module
- The LIST option writes the object module
- The MAP option writes a table of names used in the source module
- The DECK option produces a card deck of the object module.

All the options available are illustrated in Figure I-9. Default options are indicated by an underscore and need never be specified explicitly. The default options shown are standard IBM defaults; when the compiler is installed, each installation may establish its own set of default options.

Options may be coded in any order and may be separated by blanks or commas. As many as 100 characters may be coded in the PARM field. The options are enclosed in quotation marks except when the PARM field is continued onto a following card; the field must then be enclosed in parentheses rather than quotation marks, and each option containing a parenthesis must be enclosed in quotation marks. The following is an example of a continued PARM field:

```
PARM=(NOSOURCE,
   'LINECOUNT(50)',LIST)
```

For purposes of simplicity, the discussion below lists only one affirmative and negative form of each option. For examples of output that these options produce, see the chapter "Compiler Output."

SOURCE
NOSOURCE
: indicates whether the source module listing is to be written. If SOURCE is specified, the listing is produced in the data set defined by the SYSPRINT DD statement.

LINECOUNT(number)
: indicates the maximum number of lines to be assigned per page of the source listing. The number may be in the range 1 to 99. If the option is omitted, the compiler assumes 60.

LIST
NOLIST
: indicates whether the object module listing is to be written. The object module listing consists of statements written in pseudo-assembler language format. If LIST is specified, the listing is produced in the data set defined by the SYSPRINT DD statement.

OBJECT
NOOBJECT
: indicates whether the object module (not the listing) is to be written. OBJECT must be specified if the linkage editor job step is to be called in the current job. If the object module is to be written, it is written in the data set defined by the SYSLIN DD statement, which defines primary input to the linkage editor job step.

DECK
NODECK
: indicates whether the object module in card image form is to be produced in the data set specified by the SYSPUNCH DD statement. DECK is usually specified when the compilation step is not immediately followed by the linkage editor step; the deck is used to supply input to the linkage editor in a subsequent job.

OPTIMIZE({0|1|2})
NOOPTIMIZE
: indicates what optimizing level is to be in force. NOOPTIMIZE indicates that no optimization is to be performed, and is equivalent to the specification OPTIMIZE(0). OPTIMIZE(1) specifies that each source

module is to be treated by the compiler as a single program loop and that the single loop is to be optimized with regard to register allocation and branching. OPTIMIZE(2) specifies that each source module is to be treated as a collection of program loops and that each loop is to be optimized with regard to register allocation, branching, common expression elimination, and replacement of redundant computations. Optimizing techniques are discussed in greater detail in the section "Programming Considerations."

FORMAT
NOFORMAT
indicates whether a structured source module listing is to be written. A structured source module listing indicates loop structures and the logical continuity of a source program. This option is effective only when OPTIMIZE(2) is in effect, and a DD statement named SYSUT1 is present; the listing is written in the data set specified by the SYSPRINT DD statement.

GOSTMT
NOGOSTMT
indicates whether internal sequence numbers (ISN) are to be generated for the calling sequence to subroutines for a traceback map. (A traceback map is a tool used in diagnosing execution errors; it is discussed in "Load Module Output.")

MAP
NOMAP
indicates whether a table of names and statement labels used by the source program is to be written. If MAP is specified, the table is written in the data set specified by the SYSPRINT DD statement.

XREF
NOXREF
indicates whether a cross reference listing of variables and labels used in the source program is to be written. If XREF is specified, ISNs are generated for each statement in which a variable or label is used. If XREF is specified, a DD statement named SYSUT2 must be supplied; the listing is written in the data set specified by the SYSPRINT DD statement.

NAME(name)
indicates the name to be given to the main source program. The name may be from one to six characters. If NAME is not specified, the compiler assumes the name MAIN.

BCD
EBCDIC
indicates whether the source module is written in BCD (Binary Coded Decimal) or EBCDIC (Extended Binary Coded Decimal Interchange Code). If BCD and EBCDIC statements are intermixed in the source module, BCD should be specified. BCD characters are not supported by the compiler as print control characters or in literal data. For example, the carriage control character to specify same line printing, +, is specified as a 12-8-6 punch in EBCDIC and as a 12 punch in BCD; the compiler recognizes only the EBCDIC code. Therefore, programs keypunched in BCD should be carefully screened for potential errors before job submission.

SIZE
MAX
nnnnk
indicates the amount of main storage to be allocated to the compilation step. The symbol nnnnK represents the number, multiplied by K (1024-bytes), to be allocated. The number may range from 160 to 9999.

If SIZE(MAX) is specified, or if the option is omitted, the compiler uses all available storage in the environment in which it is operating, except for approximately 3K bytes which are left for system routines. SIZE should only be specified to reduce the amount of storage required by FORTRAN in a multitasking environment.

AUTODBL(value)
calls the Automatic Precision Increase (API) facility and indicates whether data items are to be converted to higher precision. API provides an automatic means of converting single precision floating point calculations to double precision accuracy and double precision calculations to extended precision accuracy. The AUTODBL option indicates which particular data types are to be converted. The AUTODBL option is discussed in detail in the section "Automatic Precision Increase".

If AUTODBL is omitted, no precision increase is performed.

ALC
NOALC
indicates whether data items are to be aligned on proper storage boundaries. It is often used with the AUTODBL option to restore proper storage boundaries when a conversion is performed. ALC is discussed in detail in the section "Automatic Precision Increase."

ANSF
NOANSF
indicates whether the compiler is to recognize only those library and built-in functions specified by the American National Standards Institute, (ANS), or the entire range of functions specified by IBM in the publication IBM System/360 and System/370 FORTRAN IV Language, Order No. GC28-6515. If ANSF is specified, any function not supported by ANS is considered to be user-supplied.

FLAG(I)
FLAG(E)
FLAG(S)
indicates the level of diagnostic messages to be printed. FLAG(I) indicates that information messages, warning messages (those generating a return code of 4), error messages (those generating a return code of 8), and severe error messages (generating a return code of 12 or higher), are to be printed. FLAG(E) indicates that only error messages and severe error messages are to be printed. FLAG(S) indicates that only severe error messages are to be printed.

DUMP
NODUMP
indicates whether the contents of registers, storage, and files associated with the compiler are to be printed if an abnormal termination occurs. If DUMP is specified, a DD statement named SYSUDUMP or SYSABEND must be supplied; the dump is written in the data set specified by the DD statement.

Changing Program Options During a Batch Compilation

A batch compilation permits the programmer to compile more than one source module in a single job. The PARM options specified in the EXEC statement apply to each of the source modules unless the programmer specifies different options for a source module. To change the options for any source module, the programmer precedes the source module with a card containing the characters *PROCESSb in columns 1 through 9 followed by the options up to column 72, which must be left blank and which denotes the end of the *PROCESS card.

An example of the *PROCESS card is:

```
*PROCESS LIST,MAP
```

If succeeding source modules are not preceded by a *PROCESS card, options revert to those specified in the EXEC statement. Any option except SIZE may be specified on the *PROCESS card.

COMPILER DATA SETS

The compiler uses up to seven data sets.

Table I-7 lists the function, device types, and allowable device classes for each data set. Table I-8 lists the DCB default values for data set characteristics.

Data Sets Defined in Cataloged Procedures

If the programmer uses a cataloged procedure, he need not define the DD statements SYSPRINT, SYSPUNCH, SYSLIN, SYSUT1, and SYSUT2; these are defined in the cataloged procedure.

SYSPRINT defines printed output. Such output may be directed to a tape, direct access device, or to a printer. To specify output to a printer, the DD statement is coded:

```
//SYSPRINT DD SYSOUT=A
```

SYSOUT is the disposition for system data sets, and A is the standard output class for a printer.

SYSPUNCH defines punched output (output in card image format). Such output may be directed to a tape, direct access device, or to a card punch. To specify output to a card punch, the DD statement is coded:

```
//SYSPUNCH DD SYSOUT=B
```

B is the standard output class for a card punch.

COMPILER OPTIONS	
Form	Abbreviated Form
SOURCE|NOSOURCE	S|NOS
LINECOUNT(number)[1]	LC(number)
LIST|NOLIST	
OBJECT|NOOBJECT[2]	OBJ|NOOBJ
DECK|NODECK	
OPTIMIZE({0|1|2})[3] NOOPTIMIZE	OPT({0|1|2}) NOOPT
FORMAT|NOFORMAT[4]	FMT|NOFMT
GOSTMT|NOGOSTMT[5]	
MAP|NOMAP	
XREF|NOXREF	
NAME(name)[6]	
EBCDIC|BCD	EB|BCD
SIZE({MAX / nnnnK})	
AUTODBL(value)	AD(value)
ALC|NOALC	
ANSF|NOANSF	
FLAG(I)|FLAG(E)|FLAG(S)	
DUMP|NODUMP	

Notes:
[1]Compiler also accepts the old form: LINECNT=xx
[2]Compiler also accepts the old form: LOAD|NOLOAD
[3]Compiler also accepts the old form: OPT={0|1|2}
[4]Compiler also accepts the old form: EDIT|NOEDIT
[5]Compiler also accepts the old form: ID|NOID
[6]Compiler also accepts the old form: NAME=name

Figure I-9. Compiler Options

SYSLIN defines the object module created by the compiler. The object module may be directed to a tape or direct access device. The following is a typical SYSLIN DD statement:

```
//SYSLIN DD DSNAME=&&LOADSET,
//          DISP=(MOD,PASS),UNIT=SYSSQ,
//          SPACE=(400,(200,50))
```

In this example, DSNAME=&&LOADSET specifies a temporary data set that is to contain the object module. DISP=(MOD,PASS) specifies that the data set is new or is to be modified (in the event of multiple compilations) and is to be passed to a succeeding job step, in this case the link edit step. UNIT=SYSSQ specifies that the data set is to reside in a sequential device class. SPACE=(400,(200,50)) specifies that space is to be allocated for records whose average block length is 400 bytes; space is allocated initially for 200 such blocks and additional space is allocated as necessary for 50 more such blocks.

SYSUT1 and SYSUT2 define utility data sets used by the compiler. SYSUT1 is used if the compiler option FORMAT (structured source listing) is requested. SYSUT2 is used if the compiler option XREF (produce cross reference listing) is requested. Both data sets may reside on a tape or direct access device. The following is a typical DD statement for a utility data set:

```
//SYSUT1 DD UNIT=SYSSQ,SPACE=(3465,(10,10))
```

In this example, UNIT=SYSSQ specifies that the data set is to reside in the sequential device class SYSSQ. SPACE=(3465,(10,10)) specifies that space is to be allocated for records whose average block length is 1050 bytes; space is allocated initially for 10 such blocks, and additional space is allocated as necessary for 10 more such blocks.

Data Sets That Must Be Defined by the Programmer

Cataloged procedures do not supply the DD statements SYSIN, SYSUDUMP, and SYSABEND; the programmer must define these, as required, when he submits a program to be compiled.

SYSIN defines the source module. The source module may reside on a tape, on a direct-access device, or on cards. To specify a source module in the input stream, the DD statement is coded:

```
//SYSIN DD *
```

The asterisk indicates that the source module statements physically follow the SYSIN DD* statement.

Table I-7. Compiler Data Sets.

ddname	Function	Device Types	Applicable Device Class	Defined in Cataloged Procedures Calling the Compiler
SYSIN	Reading input source module	Card reader Magnetic tape Direct access	Input stream (defined as DD * or DD DATA) SYSSQ	No
SYSPRINT	Writing listings, storage maps, messages	Printer Magnetic tape Direct access	A SYSSQ	Yes
SYSPUNCH	Punching the object module deck	Card punch[1] Magnetic tape Direct access	B SYSCP SYSSQ SYSDA	Yes
SYSLIN	Creating an object module data set as compiler output and linkage editor input	Direct access Magnetic tape Card punch[1]	SYSDA SYSSQ SYSCP	Yes
SYSUT1	Work data set for structured source listing; required if compiler option EDIT is requested	Magnetic tape Direct access	SYSSQ	Yes
SYSUT2	Work data set for compiler cross reference listing; required if compiler option XREF is requested	Magnetic tape Direct access	SYSSQ	Yes
SYSUDUMP or SYSABEND	Writing dump in event of abnormal termination	Printer Magnetic tape Direct access	A SYSSQ	No

[1]SYSPUNCH and SYSLIN may not be directed to the same card punch.

Table I-8. DCB Default Values for Compiler Data Sets

ddname	LRECL	RECFM	BLKSIZE	BUFNO
SYSIN	80	FB	80	2
SYSPRINT	137	VBA	141	2
SYSLIN	80	FB	3200	2
SYSPUNCH	80	FB	3440	2
SYSUT1	105	FB	3465[1]	2
SYSUT2	1024-4096[2]	FB	1024-4096[1]	2

[1]This value is fixed by the compiler and may not be overridden. (Values for other entries may be overridden through the DCB parameter in the DD statement.)
[2]The value is within this range and the actual value is calculated during compiler execution.

SYSUDUMP or SYSABEND define data sets on which abnormal termination dumps may be written if the DUMP compiler option has been specified. SYSUDUMP is requested when the user wants to display the problem program area in the event of an abnormal termination. SYSABEND is requested when the user wants to display the system nucleus and trace table in addition to the problem program area. Abnormal termination data sets may be directed to tape, direct access devices, or to a printer. To specify the data set to a printer, the DD statement is coded:

```
//SYSUDUMP  DD  SYSOUT=A
```

or

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

The linkage editor and the loader are two of the processing programs in the operating system. They both perform the link edit function, i.e., combining the object module with other modules to form one executable load module. They differ in the way they store the load module. The linkage editor places the load module into a library, where it is called for execution by another job step; the loader places the load module directly into storage for execution in the same job step.

CHOOSING THE PROPER LINKAGE PROGRAM

The choice of the proper linkage program depends upon the facilities required. The linkage editor provides the following facilities:

1. The overlay feature. The overlay feature separates a program into two or more segments that do not have to be in main storage at the same time, thereby reducing storage size requirements for large programs.

2. Control statements to provide additional processing flexibility. Linkage editor control statements define additional libraries available to the linkage editor, define the structure of segments of a program, and serve other uses.

3. Placement of the load module into a library for execution at a later time.

The loader is the more efficient program when:

1. a small load module, not requiring the use of overlay, is to be executed,

2. no linkage editor control statements are needed, and

3. the load module is to be executed immediately.

LINK EDIT JOB STEP

The EXEC statement calls the linkage editor by name, IEWL, in the PGM parameter, i.e.,

```
//  EXEC PGM=IEWL
```

The EXEC statement may also specify other parameters and linkage editor options.

LINKAGE EDITOR OPTIONS

Linkage editor options increase the flexibility of the linkage editor. At the time the operating system is generated, each installation chooses its set of default options. At execution time, the programmer may specify other options through the EXEC statement PARM parameter. Options available are illustrated in Figure I-10. Options may appear in any order. For examples of output that these options produce, see the section "Linkage Editor and Loader Output."

MAP
XREF
: indicates that a map of the load module is to be produced, showing the location and length of main programs and subprograms. The map is produced in the data set defined by the SYSPRINT DD statement. XREF indicates that a cross reference listing is also to be produced. If neither option is specified, no map or cross-reference listing is produced.

LET
: indicates that the linkage editor is to mark the load module executable even though abnormal conditions, which could cause execution to fail, have been detected. The LET option is useful in testing segments of a large program that refer to segments not yet coded; as long as the calls to absent segments are not executed, the user can still test the finished segments.

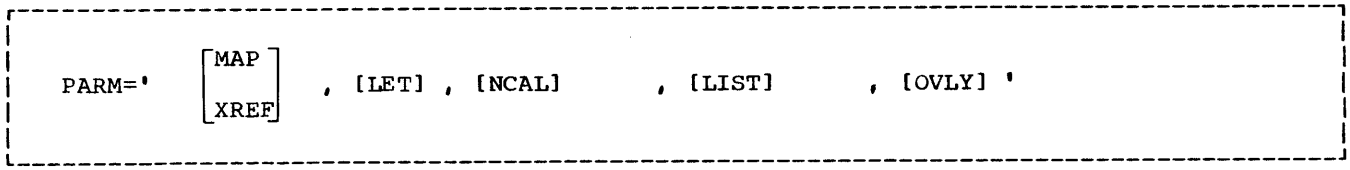


Figure I-10. Linkage Editor Options

NCAL
indicates that the linkage editor is to call no system libraries to resolve external references. (The SYSLIB DD statement, which defines system libraries, need not be submitted.) The load module is marked executable even though references to other programs may have been detected.

LIST
indicates that any linkage editor control statements are to be listed in the data set defined by the SYSPRINT DD statement.

OVLY
indicates that the load module is to be in the format of an overlay program, i.e., segments of the program may share the same storage area at different times during processing. Overlay programs are described in detail in the section "Linkage Editor Overlay Feature."

LINKAGE EDITOR DATA SETS

The linkage editor normally uses five data sets; others may be necessary if secondary input is specified.

Table I-9 lists the function, device types, and allowable device classes for each data set.

Data Sets Defined in Cataloged Procedures

Cataloged procedures calling the linkage editor contain the DD statements SYSLIN, SYSLIB, SYSLMOD, SYSPRINT, and SYSUT1. (Note that the SYSUT1 DD statement for the linkage editor should not be confused with the SYSUT1 DD statement used in the compile job step.)

SYSLIN defines the object module used as input to the linkage editor. The following is a linkage editor SYSLIN DD statement that is the counterpart to the compiler SYSLIN DD statement illustrated in the section "Compilation":

```
//SYSLIN DD DSNAME=&LOADSET,
//          UNIT=SYSSQ,DISP=(OLD,DELETE)
```

In this example, DISP=(OLD,DELETE) specifies that the data set existed prior to this job step and that it is to be deleted at the end of the step.

SYSLIB defines the system library, SYS1.FORTLIB, from which IBM-supplied FORTRAN subroutines may be obtained by the linkage editor to resolve references made by the object module to other programs (such references are called external references). SYSLIB is specified if there is a possibility that the compiler may have generated calls to any FORTRAN subroutine. SYS1.FORTLIB exists prior to the job and may be called simply by name. To specify the library, the DD statement is coded:

```
//SYSLIB DD DSNAME=SYS1.FORTLIB,DISP=SHR
```

In this example, DISP=SHR is coded to permit other jobs to have access to the library while this job step is still in progress. If the NCAL option is specified, the SYSLIB DD statement is not required.

SYSLMOD defines the load module created by the link edit job step. The load module may be directed only to a direct access device and must be stored in a library as a named member. The library may be the system library, SYS1.LINKLIB, a temporary library, or a private library. The following is a typical SYSLMOD DD statement:

```
//SYSLMOD  DD  DSNAME=&GOSET(MAIN),
//             UNIT=SYSDA,DISP=(,PASS),
//             SPACE=(3072,(30,10,1),RLSE)
```

In this example, DSNAME=&GOSET specifies a temporary library. MAIN specifies the member name of the load module. UNIT=SYSDA specifies that the data set is to reside in a direct access device class, DISP=(,PASS) specifies that the data set is new (by default) and that it is to be passed to a later job step. SPACE=(3072,(30,10,1),RLSE) allocates space for 30 record blocks, whose size is 3072 bytes, allocates space for 10 additional blocks as necessary, allocates 1 block of 256 bytes for the directory, and specifies that unused space may be released at the end of the step.

The following example indicates how a programmer may store the load module into a private library:

```
//SYSLMOD DD DSNAME=USERLIB(PROG1),
//           UNIT=SYSSQ,DISP=(,CATLG),
//           SPACE=(TRK,(50,30,3),
//           VOLUME=SER=34345
```

In this example, USERLIB specifies the name of the library. PROG1 the member name of the load module. UNIT=SYSSQ specifies that the library (or data set) is to reside in a sequential device class. DISP=(,CATLG) specifies a new data set that is to be cataloged. SPACE allocates 50 tracks initially, 30 tracks if needed subsequently, and 3 blocks of 256 bytes for

the directory. VOLUME specifies the specific volume which is to hold the data set.

To execute the load module PROG1 in a later job, the programmer must submit job control statements containing the following minimum information:

```
//jobname JOB
//JOBLIB  DD DSNAME=USERLIB,DISP=(OLD,KEEP)
//        EXEC PGM=PROG1
```

The DD statement defining a private library must be named JOBLIB and must follow the JOB statement and precede any EXEC statements. It makes the private library available. The EXEC statement names the program to be executed. These statements cause the operating system to search the private library to locate the program as an executable load module.

SYSPRINT defines printed output. As in the compiler step, such output may be directed to a tape, a direct access device, or to a printer.

SYSUT1 defines a utility data set used by the linkage editor. This data set may reside only in a direct access device class. The following is a typical SYSUT1 DD statement:

```
//SYSUT1  DD  DSNAME=&SYSUT1,UNIT=SYSDA,
//            SPACE=(1024,(200,20)),
//            SEP=SYSLMOD
```

In this example, DSNAME=&SYSUT1 specifies a temporary data set. UNIT=SYSDA specifies that the data set resides in a direct access device class. SPACE allocates space to the data set. SEP=SYSLMOD specifies that this data set is not to use the same channel as the data set defined in the SYSLMOD DD statement.

Data Sets That Must Be Defined by the Programmer

There are two kinds of input to the linkage editor: primary and secondary. If any secondary input is to be submitted, the programmer must define it; cataloged procedures do not supply DD statements for secondary input.

Primary Input

Primary input is the data set defined in the SYSLIN DD statement. Normally, this data set consists of the output from a previous compilation job step, but primary input may also be linkage editor control statements (discussed under "Linkage Editor Control Statements"). Primary input may reside on tape, direct access, or cards.

Table I-9. Linkage Editor Data Sets

ddname	Function	Device Type	Applicable Device Class	Defined in Cataloged Procedures Calling the Linkage Editor
SYSLIN	Primary input data, normally output of the compiler	Direct access Magnetic tape Card reader	SYSDA SYSSQ input stream (defined as DD * or DD DATA)	Yes
SYSLIB	Automatic call library (SYS1.FORTLIB)	Direct access	SYSDA	Yes
SYSLMOD	Link edit output (load module input)	Direct access	SYSDA	Yes
SYSPRINT	Writing listings, messages	Printer Magnetic tape Direct access	A SYSSQ	Yes
SYSUT1	Work data set	Direct access	SYSDA	Yes
user-defined	Additional libraries and object modules	Direct access Magnetic tape	SYSDA SYSSQ	No

Secondary Input

Secondary input consists of modules that are not part of the primary input data set but are to be included in the load module. The linkage editor uses secondary input to resolve external references between the primary input and other programs which it calls. Secondary input may be in the following forms:

1. Object modules specified by the user. These modules may be either sequential data sets or members of a library.

2. Load modules specified by the user. These modules must be members of a library. They may contain linkage editor control statements.

3. The automatic call library, SYS1.FORTLIB. This is the data set defined in the SYSLIB DD statement. SYS1.FORTLIB contains the FORTRAN library subprograms as its members. The linkage editor uses this library if unresolved references remain after other input has been processed.

The user defines secondary input in a linkage editor control statement and a DD statement.

Linkage Editor Control Statements

Linkage editor control statements specify an operation and one or more operands.

The first column of a control statement must be left blank. The operation field begins in column 2 and specifies the name of the operation to be performed. The operand field must be separated from the operation field by at least one blank. The operand field specifies one or more operands separated by commas. No embedded blanks may appear in the field. Linkage editor control statements may be placed before, between, or after either modules or other control statements in primary or secondary input data sets.

The INCLUDE and LIBRARY control statements specify secondary input.

INCLUDE Statement: The INCLUDE statement specifies additional programs to be included as part of the load module. Its format is:

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

Each ddname indicates the name of a DD statement specifying a library or a sequential data set, and each member-name is the name of the member to be included. When sequential data sets (not members) are specified, member-name is omitted.

The following is an example of the INCLUDE control statement and its corresponding DD statement:

```
//LIB1 DD DSNAME=MYLIB,DISP=OLD
//SYSLIN DD *
 INCLUDE   LIB1(PROG1)
```

LIBRARY Statement: The LIBRARY statement specifies additional libraries to be searched for object modules to be included in the load module.

The LIBRARY statement differs from the INCLUDE statement in that libraries specified in the LIBRARY statement are not searched until all other references (except those reserved for the automatic call library) are completed by the linkage editor. A module specified by an INCLUDE statement is included immediately.

The format of the LIBRARY statement is:

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

Each ddname indicates the name of a DD statement specifying a library, and each member-name is the name of a member of the library.

The following is an example of the LIBRARY control statement and its corresponding DD statement:

```
//LIB2 DD DSNAME=ADDLIB,DISP=OLD
//SYSLIN DD *
 LIBRARY LIB2(ADD1,ADD2)
```

Figure I-11 illustrates the use of linkage editor control statements. STEP1, STEP2, and STEP3 are compile job steps. STEP1 compiles a main program, MAIN, and places the object module in a sequential data set called &&GOFILE. STEP2 and STEP3 compile subprograms SUB1 and SUB2 and place the object modules in separate sequential data sets.

STEP4 is the link edit job step. It uses the &&GOFILE data set as primary input. The compiled subprograms are used as secondary input through the INCLUDE statements and the DD statements named DD1 and DD2. An additional data set, defined in the LIBRARY statement and in the DD statement named ADDLIB, is to be used if external references are not resolved among the three object modules. Note that the INCLUDE and LIBRARY statements are entered through the input job stream with a DD * statement.

After link edit processing, the load module CALC is stored as a member of the load module library PROGLIB. CALC contains the main program, the two subprograms, and, possibly, routines from the user library MYLIB and the system library SYS1.FORTLIB.

IDENTIFY Statement: For a description of the IDENTIFY statement, see the OS/VS linkage editor and loader publication listed in the Preface.

ORDERING AND PAGE-ALIGNING PROGRAM UNITS UNDER OS/VS

Under the VS control programs, linkage editor control statements may be used to specify the sequence of FORTRAN program units in the output load module and to specify their alignment on page boundaries. Such ordering and alignment can be used to effect a lower paging rate and thus make more efficient use of real storage.

ORDER Statement: The ORDER statement indicates the sequence in which program units are to appear in the output load module. The program units appear in the sequence in which they are specified on the ORDER statement. When multiple ORDER statements are used, their sequence further determines the sequence of program units in the output load module; those named on the first statement appear first, and so forth.

The format of the ORDER statement is:

Operation	Operand
ORDER	name[(P)] ,name [(P)] ...

where:

name
: is the name of a FORTRAN main program, subprogram, or COMMON block

(P)
: indicates that the starting address of the program of the program unit is to be on a page boundary within the load module. The program units are aligned on 4K page boundaries unless the ALIGN2 attribute is specified on the EXEC statement. (Page boundary alignment in the executing module can only occur when the operating system supervisor includes support for fetch on a page boundary. This support is available only with VS2.)

An ORDER statement may be placed before, between, or after object modules or other control statements.

PAGE Statement: The PAGE statement, like the (P) operand of the ORDER statement, aligns a program unit on a 4K page boundary in the output load module. If the ALIGN2 attribute is specified on the EXEC statement for the linkage editor job step, use of the PAGE statement aligns the specified program units on 2K page boundaries within the load module. (As with the (P) operand of the ORDER statement, page boundary alignment in the executing module can only occur when the operating system supervisor includes support for fetch on a page boundary. This support is available only with VS2.)

The format of the PAGE statement is:

Operation	Operand
PAGE	name [,name]...

where:

name
: is the name of a FORTRAN main program, subprogram or COMMON block.

The PAGE statement may be placed before, between, or after object modules or other control statement.

In the example shown in Figure I-10.1., the program units RAREUSE and MAINRT are aligned on 2K page boundaries by PAGE and ORDER control statements used with the

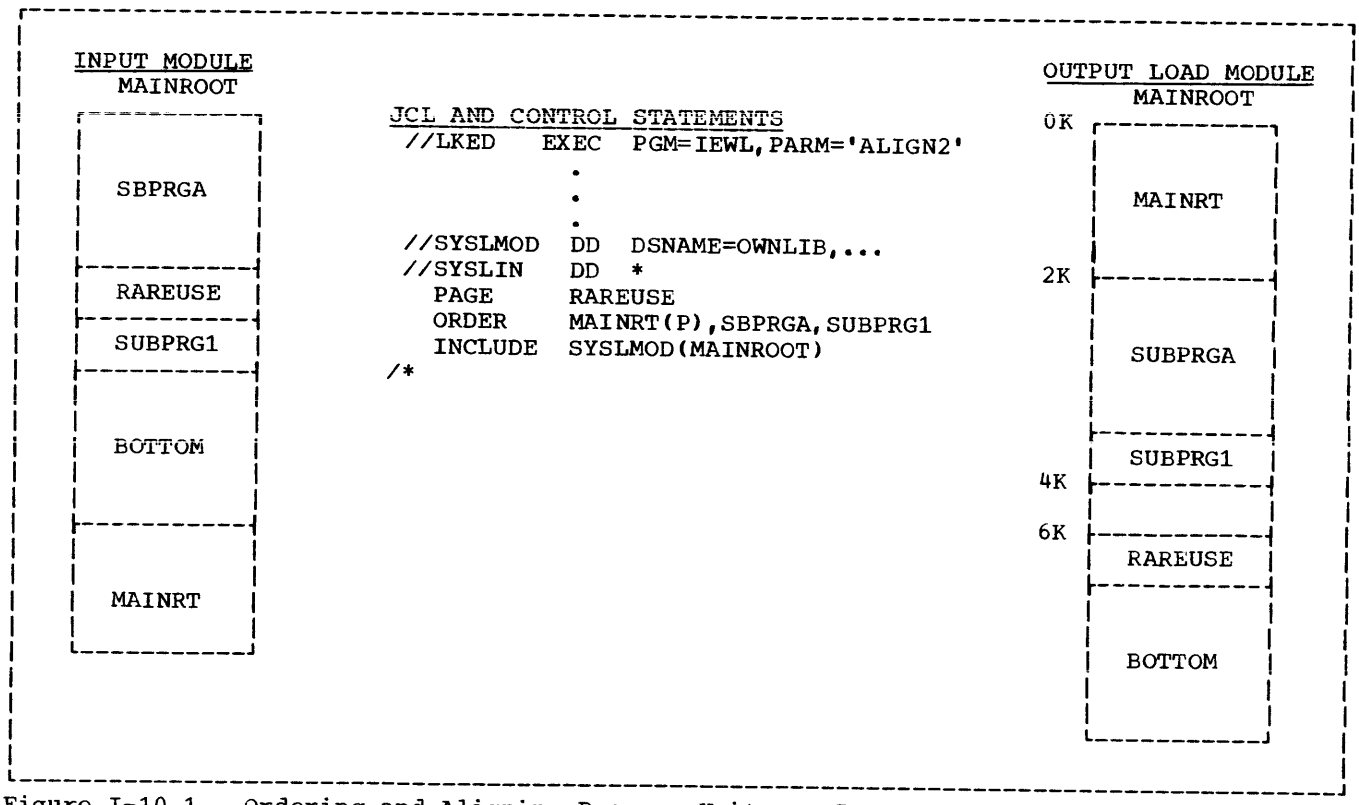


Figure I-10.1. Ordering and Aligning Program Units on Page Boundaries

ALIGN2 attribute. Program units SBPRGA and SUBPRG1 are sequenced by the ORDER control statement. Assume that each program unit is 2K in length except for SUBPPG1 and RAREUSE.

The linkage editor places the program units MAINRT and RAREUSE on 2K page boundaries because ALIGN2 is specified in the EXEC statement. Program units MAINRT, SBPRGA, and SUBPRG1 are sequenced as specified in the ORDER statement. RAREUSE, while placed on a 2K page boundary, appears after the program units specified in the ORDER statement because it was not included. The program unit BOTTOM comes after RAREUSE because it appeared after RAREUSE in the input module.

SYSTEM LOADER

The loader combines into one job step the functions of link editing and load module execution. The loader combines the object module with other modules to form one executable load module. It places the load module directly into main storage and then executes it. By placing the load module directly into main storage, the loader eliminates the need for writing and then reading the SYSLMOD data set. The EXEC statement identifies the loader by either of its names, IEWLDRGO or LOADER, in the PGM parameter, i.e.,

```
//EXEC PGM=LOADER
```

The EXEC statement may also specify other parameters and loader options.

LOADER OPTIONS

Loader options increase the flexibility of the loader. At the time the operating system is generated, each installation chooses its set of default options. At execution time, the programmer may specify other options through the EXEC statement PARM parameter. Options available are illustrated in Figure I-12. Options may be coded in any order.

MAP
NOMAP
: indicates whether a map of the loaded program is to be produced, listing external names and absolute storage addresses.

LET
NOLET
: indicates whether the loader is to mark the load module executable even

though abnormal conditions, which could cause execution to fail, have been detected.

CALL
NCAL

indicates whether the loader is to call a system library to resolve external references. If the library is to be called, the SYSLIB DD statement must be submitted.

SIZE=nnnnn

SIZE indicates the amount of storage, in bytes, that is to be allocated to loader processing. The size of the load module must be included in this number. If the option is not specified, the default size, 100K, is assumed.

```
          //JOBX    JOB
          //STEP1   EXEC      PGM=IFEAAB,PARM='NAME(MAIN),LOAD'
                                  .
                                  .
                                  .
 Step 1   //SYSLIN  DD        DSNAME=&&GOFILE,DISP=(,PASS),UNIT=SYSSQ
          //SYSIN   DD        *
                   ┌──────────────────────┐
                   │Source module for MAIN│
                   └──────────────────────┘
          /*
          //STEP2   EXEC      PGM=IFEAAB,PARM='NAME(SUB1),LOAD'
                                  .
                                  .
                                  .
 Step 2   //SYSLIN  DD        DSNAME=&&SUBPROG1,DISP=(,PASS),UNIT=SYSSQ
          //SYSIN   DD        *
                   ┌──────────────────────┐
                   │Source module for SUB1│
                   └──────────────────────┘
          /*
          //STEP3   EXEC      PGM=IFEAAB,PARM='NAME(SUB2),LOAD'
                                  .
                                  .
                                  .
 Step 3   //SYSLIN  DD        DSNAME=&&SUBPROG2,DISP=(,PASS),UNIT=SYSSQ
          //SYSIN   DD        *
                   ┌──────────────────────┐
                   │Source module for SUB2│
                   └──────────────────────┘
          /*
          //STEP4   EXEC      PGM=IEWL
                                  .
                                  .
                                  .
          //SYSLIB  DD        DSNAME=SYS1.FORTLIB,DISP=SHR
          //SYSLMOD DD        DSNAME=PROGLIB(CALC),UNIT=SYSDA
          //ADDLIB  DD        DSNAME=MYLIB,DISP=OLD
 Step 4   //DD1     DD        DSNAME=*.STEP2.SYSLIN,DISP=OLD
          //DD2     DD        DSNAME=*.STEP3.SYSLIN,DISP=OLD
          //SYSLIN  DD        DSNAME=*.STEP1.SYSLIN,DISP=OLD
          //        DD        *
           INCLUDE  DD1
           INCLUDE  DD2
           LIBRARY  ADDLIB(X,Y,Z)
          /*
          //
```

Figure I-11. Linkage Editor Processing

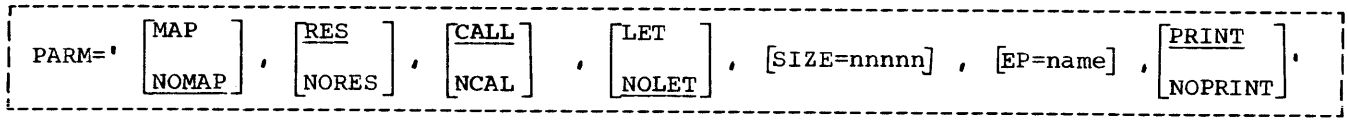


Figure I-12. Loader Options

EP=name
indicates the name that is to be the entry point of the program being loaded.

PRINT
NOPRINT
indicates whether loader messages are to be produced. Messages are produced in the data set defined by the SYSLOUT DD statement.

RES
NORES
indicates whether, in an MVT environment, the link pack area queue is to be searched to resolve external references. The link pack is an area of storage that contains a number of modules needed for job processing. If RES is specified, the link pack area queue is searched prior to any search of the SYSLIB data set.

LOADER DATA SETS

The loader normally uses six data sets; other data sets may be defined to describe libraries, loader output, and load module data sets.

Table I-10 lists the function, device types, and allowable device classes for each data set.

Data Sets Defined in Cataloged Procedures

Cataloged procedures calling the loader contain the DD statements SYSLIN, SYSLIB, SYSLOUT, FT05F001, FT06F001, and FT07F001.

SYSLIN defines the object module that is primary input to the loader. Normally this data set consists of the output from a previous compile job step, but it may also be an object module from a partitioned data set. Input may reside on a tape, a direct access device, or on cards. The following is the loader SYSLIN DD statement corresponding to the compiler SYSLIN DD statement illustrated in the section "Compilation."

```
//SYSLIN  DD  DSNAME=&&LOADSET,
//            DISP=(OLD,DELETE)
```

SYSLIB defines the system library, SYS1.FORTLIB, that is to be searched to resolve external references made by the input data set. The SYSLIB DD statement may be coded as follows:

```
//SYSLIB  DD  DSNAME=SYS1.FORTLIB,DISP=SHR
```

SYSLIB is not required if the NCAL loader option is specified.

SYSLOUT defines the output data set to store the loader map. A printed listing is obtained if SYSLOUT is allocated to a printer device class, as follows:

```
//SYSLOUT  DD  SYSOUT=A
```

SYSLOUT is not required if the NOMAP and NOPRINT options are specified.

FT05F001 defines the input data set to the load module. The function of this ddname is to associate the system input unit to FORTRAN READ statements having 5 as the data set reference number.

Since cataloged procedures may not contain DD * statements, FT05F001 defers data set definition to the SYSIN DD statement, which the programmer must supply. The following FT05F001 DD statement appears in cataloged procedures:

```
//FT05F001  DD  DDNAME=SYSIN
```

The programmer completes card input definition by supplying a SYSIN DD statement as follows:

```
//GO.SYSIN  DD  *
```

GO identifies card input with the GO step in cataloged procedures.

FT06F001 defines the system printer unit. The function of this ddname is to associate the system printer to FORTRAN WRITE statements having 6 as the data set reference number. It is defined as follows:

```
//FT06F001  DD  SYSOUT=A
```

FT07F001 defines the system card punch unit. The function of this ddname is to associate the card punch to FORTRAN WRITE statements having 7 as the data set reference number. It is defined as follows:

```
//FT07F001  DD  SYSOUT=B
```

Data Sets That Must be Defined by the Programmer

The programmer must define the SYSIN DD statement to complete the description of the input data set, and the SYSPRINT DD statement.

SYSPRINT defines system printed output, such as allocation and job control messages. Output may be directed to a tape, a direct access device, or to a printer. To direct output to a printer, SYSPRINT is coded:

```
//SYSPRINT  DD  SYSOUT=A
```

The programmer may also define other data sets as required by the load module execution function. See the section "Load Module Execution" for a discussion of these data sets.

Table I-10. Loader Data Sets.

ddname	Function	Device Type	Applicable Device Class	Defined in Cataloged Procedures Calling the Loader
SYSLIN	Input data to linkage function, normally output of the compiler	Direct access Magnetic tape Card reader	SYSDA SYSSQ Input stream (defined as DD *)	Yes
SYSLIB	Automatic call library (SYS1.FORTLIB)	Direct access	SYSDA	Yes
SYSLOUT	Writing listings	Printer Magnetic tape Direct access	A SYSSQ	Yes
SYSPRINT	Writing messages	Printer Magnetic tape Direct access	A SYSSQ	No
SYSIN	Input data to load module function	Card reader Magnetic tape Direct access	Input stream (defined as DD *) SYSSQ SYSDA	No
FT05F001	Primary input data to be processed by the load module	Card reader Magnetic tape Direct access	Input stream (defined as DD * or DD DATA) SYSSQ SYSDA	Yes
FT06F001	Printed output data	Printer	A	Yes
FT07F001	Punched output data	Card punch	B	Yes
FTnnFnnn*	User-defined data set	Unit record Magnetic tape Direct access	SYSSQ A,B SYSDA	No

*nn and nnn cannot be set to 0.

The load module execution job step executes a load module. The load module may be passed directly from a preceding link edit job step, it may be called from a library of programs, or it may form part of the loader job step. If passed from the link edit job step, the load module is called in the PGM parameter of the EXEC statement, i.e.,

```
// EXEC PGM=*.LKED.SYSLMOD
```

This statement defines the load module as consisting of the data set described in the SYSLMOD DD statement in the link edit (LKED) job step of the current job.

If the load module is called from a library, it is called by name, as any program, in the PGM parameter, i.e.,

```
// EXEC PGM=MATRIX
```

The library in which the load module resides must also be made available to the operating system via a DD statement. A load module may reside on the system library, SYS1.LINKLIB, or on a private library. A private library is defined in a JOBLIB or STEPLIB DD statement. For example, if the load module MATRIX is a member of a private library named MATH, the user may supply the following DD statement:

```
//JOBLIB DD DSNAME=MATH,DISP=(OLD,PASS)
```

The JOBLIB DD statement must appear after the JOB statement and before any other control statement. This placement ensures that the private library is kept available for all steps within the job.

If the load module is executed as part of the loader, it is not defined in an EXEC statement. The loader combines the link editing and load module execution into one job step.

LOAD MODULE DATA SETS

The load module execution job step may use many data sets. Table I-11 lists the function, device types, and allowable device classes for each data set.

Data Sets Defined in Cataloged Procedures

Cataloged procedures calling the execution job step contain the DD statements FT05F001, FT06F001, and FT07F001.

FT05F001 defines the input data set. The programmer codes 5 as the data set reference number in any FORTRAN READ statement that reads card input. Since cataloged procedures may not contain DD * statements, FT05F001 is coded to defer data set definition to the SYSIN DD statement, which the programmer must supply. The following FT05F001 DD statement appears in cataloged procedures:

```
//FT05F001 DD DDNAME=SYSIN
```

The programmer completes card input definition by supplying a SYSIN DD statement as follows:

```
//GO.SYSIN DD *
```

GO identifies card input with the GO step in cataloged procedures.

FT06F001 defines a printer data set. The programmer codes 6 as the data set reference number in any WRITE statement writing data to be printed. The following FT06F001 DD statement appears in cataloged procedures:

```
//FT06F001 DD SYSOUT=A
```

FT07F001 defines a card punch data set. The programmer codes 7 as the data set reference number in any WRITE statement writing data to be punched. The following FT07F001 DD statement appears in cataloged procedures:

```
//FT07F001 DD SYSOUT=B
```

Table I-11. Load Module Data Sets

ddname	Function	Device Type	Applicable Device Class	Defined in Cataloged Procedures Calling the Load Module
FT05F001	Input data to the load module	Card reader Magnetic tape Direct access	SYSSQ SYSDA	Yes
SYSIN	Input data to the load module	Card reader Magnetic tape Direct access	SYSSQ SYSDA	No
FT06F001	Printed output data	Printer	A	Yes
FT07F001	Punched output data	Card punch	B	Yes
FTnnFnnn*	User-defined sequential data set	Unit-record Magnetic tape Direct access	SYSSQ A,B	No
FTnnFnnn*	User-defined partitioned data set containing sequential members	Direct access	SYSSQ	No
FTnnFnnn*	User-defined Direct-access data set	direct access	SYSDA	No

*nn and nnn cannot be set to 0.

Data Sets That Must Be Defined by the Programmer

The SYSIN DD statement must be defined by the programmer to complete the description of the input data set begun by the DD statement FT05F001. (If no input data set is to be submitted, the SYSIN DD statement is omitted and the operating system ignores the FT05F001 DD statement.)

The FORTRAN programmer may define other data sets for use in the load module execution step. There are three types of data sets: sequential, partitioned, and direct-access.

Sequential Data Sets

A sequential data set may be coded either in EBCDIC (Extended Binary-Coded-Decimal Interchange Code) or in ASCII (American National Standard Code for Information Interchange). ASCII data sets may be processed by the IBM System/360 Operating System only if the option ASCII=INCRES or ASCII=INCTRAN has been specified at the time the operating system is generated.

An EBCDIC data set may reside on unit record devices, magnetic tape volumes or direct access volumes. Data sets defined on a tape or direct access device may be retained for use in later jobs; unit record data sets are temporary and exist for the current job only (although data from these data sets may be transmitted to permanent volumes during job processing). FT05F001, FT06F001, FT07F001, and SYSIN DD * all define sequential data sets.

Figure I-13 illustrates DD statements to define unit record data sets. Figure I-14 illustrates DD statements to create tape and direct access data sets. Figure I-15 illustrates DD statements to retrieve data sets.

An ASCII data set may reside only on magnetic tape volumes. Essentially, a programmer uses the same DD statement parameters to define an ASCII tape as he would an EBCDIC tape. The differences are that an ASCII tape must be identified either through the LABEL parameter (LABEL=AL) or through the DCB subparameter OPTCD (OPTCD=Q) and may be defined only on 9-track tape having a density of 800 bpi. Figure I-16 illustrates the corresponding DD statement for the one shown in Figure LM2 if an ASCII data set was being defined.

```
Example 1:  Data set in the input stream:

            //SYSIN  DD  *

Example 2:  Data set on the printer:

            //SYSPRINT  DD  SYSOUT=A

Example 3:  Data set on the card punch:

            //FT07F001  DD  SYSOUT=B
```

Figure I-13. Defining Unit Record Data Sets

```
Example 1:  Temporary Data set:

            //FT14F001  DD  DSNAME= &TEMP,UNIT=SYSSQ,SPACE=(TRK,(50))

Example 2:  Permanent data set on a tape volume:

            //FT36F001  DD  DSNAME=ANY,VOLUME=SER=7342,UNIT=2400,
            //              DISP=(NEW,KEEP,DELETE),LABEL=(,SL,EXPDT=69365)

Example 3:  Permanent data set on a direct access volume:

            //FT41F001  DD  DSNAME=SOME,DISP=(NEW,CATLG,DELETE),UNIT=2311,
            //              VOLUME=SER=AA69,SPACE=(300,(100,100)),
            //              DCB=(RECFM=VB,LRECL=304,BLKSIZE=612)
```

Figure I-14. Creating EBCDIC Sequential Data Sets on Tape or Direct Access Volumes

```
Example 1:  Retrieving an uncataloged data set (to be kept at the end of the job):

            //FT36F001  DD  DSNAME=ANY,VOLUME=SER=7342,UNIT=2400,DISP=OLD

Example 2:  Retrieving a cataloged data set (to be deleted at the end of the job):

            //FT41F001  DD  DSNAME=SOME,DISP=(OLD,DELETE)
```

Figure I-15. Retrieving Sequential Data Sets

```
Example 1:  Using the LABEL parameter:

            //FT36F001  DD  DSNAME=ANY,VOLUME=SER=7342,UNIT=2400,
            //              DISP=(NEW,KEEP,DELETE),LABEL=(AL,EXPDT=69365)

Example 2:  Using the DCB parameter OPTCD subparameter:

            //FT36F001  DD  DSNAME=ANY,VOLUME=SER=7342,UNIT=2400,
            //              DISP=(NEW,KEEP,DELETE),DCB=(OPTCD=Q)
```

Figure I-16. Creating an ASCII Tape Data Set

Partitioned Data Sets

A partitioned data set (PDS) may reside only on a direct access device; hence, any DD statement parameters defining unit record or magnetic tape data sets are not applicable.

A PDS consists of groups of sequential data which are called members of the data set. Partitioned data sets are used to contain libraries.

Figure I-17 illustrates DD statements to define partitioned data sets.

Figure I-18 illustrates DD statements to retrieve a member from a partitioned data set. The first example indicates that the member CASE2 is to be retrieved from the data set USERLIB, that the member is to be used for input operations (LABEL=(,,,IN)), and that the data set is to be retained at the end of the job (the absence of the second subparameter in DISP causes old data sets to be kept).

Like the first example, the second example retrieves a member for input operations, but the DELETE specification in the DISP parameter deletes the entire data set, including all members.

The following discussion describes how more than one member may be processed in the same job and how a single member may be deleted from a partitioned data set.

Retrieving More Than One Member

Two or more members of the same partitioned data set may be processed in one job in a sequential manner, i.e., the PDS must be closed for one member before attempting to read or write another member. Two or more members may be retrieved using either the READ statement END= option or the REWIND statement.

PDS Processing Using END=n Option: When the END=n option is executed and a subsequent READ or WRITE statement is issued with the same data set reference number, the FORTRAN sequence number is incremented by one. This allows another member of the PDS referenced by the same data set reference number to be processed.

The following FORTRAN program illustrates this method:

```
      INTEGER*4 X(20),Y(20)
10    READ (2,1,END=98) X
 1    FORMAT (20A4)
      GO TO 10
98    READ (2,1,END=99) Y
      GO TO 98
99    WRITE (6,2) X,Y
      STOP
      END
```

Execution of statement 10 results in processing the first PDS member which is referenced by the FORTRAN sequence number 001. Assume that this member has the name

```
Example 1:  New Partitioned data set with its first member:

            //FT20F001  DD  DSNAME=USERLIB(CASE1),DISP=(NEW,CATLG),UNIT=2311,
            //              SPACE=(TRK,(50,20,5)),VOLUME=SER=DA31

Example 2:  Adding a member to an existing data set:

            //FT20F002  DD  DSNAME=USERLIB(CASE2),DISP=OLD

Example 3:  Temporary data set (created and deleted in the same job):

            //FT21F001  DD  DSNAME=&TEMPLIB(MY),DISP=(NEW,PASS),UNIT=SYSDA,
            //              SPACE=(TRK,(20,5,1))
```

Figure I-17. Creating Partitioned Data Sets

```
Example 1:  Retrieving a member of a data set:

            //FT25F001  DD  DSNAME=USERLIB(CASE2),DISP=OLD,LABEL=(,,,IN)

Example 2:  Retrieving a member and deleting the data set at the end of the job:

            //FT26F001  DD  DSNAME=USERLIB(CASE1),DISP=(OLD,DELETE,KEEP),LABEL=(,,,IN)
```

Figure I-18. Retrieving Partitioned Data Sets

CASE1 and resides in the cataloged partitioned data set named USERLIB; the DD statement that must be supplied is:

```
//FT02F001 DD DSN=USERLIB(CASE1),
//            LABEL=(,,,IN),DISP=OLD
```

When the END=n option is executed in statement 10 and the next READ statement, statement 98, is encountered, the FORTRAN sequence number becomes 002. This closes the PDS for the first member. Another member may then be processed. If its name is CASE2, the DD statement that must be supplied is:

```
//FT02F002 DD DSN=USERLIB(CASE2),
//            LABEL=(,,,IN),DISP=OLD
```

PDS Processing Using REWIND: Execution of the REWIND statement closes a data set. Any subsequent READ or WRITE statement opens the data set again.

The following example illustrates the use of the REWIND statement in reading two members of the same PDS:

```
          INTEGER*4 X(20),Y(20)
          READ (2,1) X
          REWIND 2
          READ (3,1) Y
          WRITE (6,2) X,Y
1         FORMAT (20A4)
2         FORMAT (' ',20A4)
          STOP
          END
```

Execution of the first READ statement results in the processing of the first PDS member which is referenced by the FORTRAN sequence number 001. If the member has the name CASE1 and resides in the cataloged partitioned data set named USERLIB, the DD statement that must be supplied is:

```
//FT02F001 DD DSN=USERLIB(CASE1),
//            LABEL=(,,,IN),DISP=OLD
```

When the REWIND statement is executed, the PDS is closed for MEMBER1. The next READ statement reopens the data set for another PDS member. If the next member name is CASE4, the DD statement that must be supplied is:

```
//FT03F001 DD DSN=USERLIB(CASE4),
//            LABEL=(,,,IN),DISP=OLD
```

Deleting One Member

To delete a member while retaining the remainder of the data set, the programmer submits a separate job executing the operating system utility program IEHPROGM. (IEHPROGM is described in the appropriate utilities publication, as listed in the Preface.) Figure I-19 illustrates how a job may be submitted to delete a member only. The DD statement named DD2 defines the data set. The utility program statement SCRATCH releases the member named CASE1 from the partitioned data set named USERLIB.

The SCRATCH statement deletes only the directory entry that refers to the member; the space occupied by the member is released only if the entire data set is reorganized. To reorganize a partitioned data set, the programmer copies the members into a temporary data set, deletes and recreates the original data set, and copies the members back into it. The operating system utility programs contain facilities for copying members of partitioned data sets.

Direct-Access Data Sets

A direct-access data set may reside only on a direct access device; hence, any DD statement parameters defining unit record or magnetic tape data sets are not applicable.

A direct-access data set consists of a number of records that may be accessed individually; i.e., only the record that is needed is accessed regardless of its physical position within the data set. A direct-access data set requires a corresponding DEFINE FILE statement in the FORTRAN program. Figure I-20 illustrates a DD statement and the corresponding DEFINE FILE statement to create a direct-access data set. Note that the record characteristics described in the DEFINE FILE statement must agree with the space allocation requested in the DD statement (for example, in the example, both statements specify records, whose average length is 100 bytes, allocated in blocks of 50 records). Figure I-21 illustrates the statements to retrieve the direct-access data set.

Note that the data set created in Figure I-20 was to be cataloged; therefore, to retrieve it, only the DSNAME and DISP parameters are required.

DCB PARAMETER CONSIDERATIONS

The following DCB subparameters define record characteristics of a data set:

- RECFM, to specify the format of a record, for example, whether fixed-length, variable-length, or undefined-length, and whether records are blocked
- LRECL, to specify the size of a record or the maximum size of a variable-length record
- BLKSIZE, to specify the size of a record or a block of records and the length of the buffers required to transmit data between main storage and input/output devices

Table I-12 lists the DCB default values for load module data sets. Table I-13 summarizes the maximum allowable values for the BLKSIZE subparameter.

```
//SCRTCH    JOB
//          EXEC    PGM=IEHPROGM
//SYSPRINT  DD      SYSOUT=A
//DD2       DD      UNIT=2311,VOLUME=SER=DA31,DISP=OLD
//SYSIN     DD      *
          SCRATCH  DSNAME=USERLIB,VOL=2311=DA31,MEMBER=CASE1
/*
```

Figure I-19. Deleting a Member of a Partitioned Data Set

```
      DEFINE FILE 8(50,100,L,I2)

//FT08F001  DD  DSNAME=DADS,UNIT=SYSDA,VOLUME=SER=123478,
//              DISP=(NEW,CATLG,DELETE),SPACE=(100,(50,50)),
//              DCB=(RECFM=F,BLKSIZE=100)
```

Figure I-20. Creating a Direct-Access Data Set

```
      DEFINE FILE 8(50,100,L,I2)

//FT08F001  DD  DSNAME=DADS,DISP=OLD
```

Figure I-21. Retreiving a Direct-Access Data Set

Table I-12. DCB Default Values for Load Module Data Sets

	Sequential Data Sets					Direct-Access Data Sets		
ddname	RECFM[1]	LRECL[2]	BLKSIZE	DEN	BUFNO	RECFM	LRECL or BLKSIZE	BUFNO
FT03Fyyy	U	--	800	2	2	FA	The value specified as the maximum size of a record in the DEFINE FILE statement.	2
FT05Fyyy	F	80	80	-	2	F		2
FT06Fyyy	UA	132	133	-	2	F		2
FT07Fyyy	F	80	80	-	2	F		2
all others	U	--	800	2	2	F		2

[1]For records not under FORMAT control, the default is VS.
[2]For records not under FORMAT control, the default is 4 less than shown.

Table I-13. Maximum BLKSIZE Values

Device Type	BLKSIZE Value	
	Fixed-Length and Undefined-Length Records (minimum value is 1)	Variable-Length Records (minimum value is 9)
Card Reader	80	80
Card Punch	81	89
Printer		
120 characters	121	129
132 characters	133	141
144 characters	145	153
150 characters	151	159
Direct Access[1]		
2301	20483	20483
2302	4984	4984
2303	4892	4892
2305		
Mod I	14136	14136
Mod II	14660	14660
2311	3625	3625
2314	7294	7294
3330	13030	13030
Magnetic Tape[2]	32760	32760

[1]With track overflow, the maximum BLKSIZE value is 32760 for each device.
[2]The minimum value is 18.

DCB Considerations for Sequential EBCDIC Data Sets

FORTRAN records in an EBCDIC data set may be formatted or unformatted, i.e., they may or may not be defined in a FORMAT statement. List-directed I/O statements are considered formatted. Formatted records may be specified as fixed length, variable length, or of undefined length. Unformatted records may be specified only as variable length. If records are to be processed using asynchronous input/output, they may not be blocked.

FORMATTED RECORDS: Formatted records are specified as follows:

Fixed-Length Records: Unblocked fixed-length records are specified as RECFM=F; blocked records as RECFM=FB. For unblocked records, BLKSIZE specifies the record length, (e.g., BLKSIZE=80); the buffer length is the same as the record length. For blocked records, LRECL specifies the record length and BLKSIZE the block length, which must be a multiple of LRECL, (e.g., LRECL=80,BLKSIZE=400); the buffer length is the same as the block length.

Variable-Length Records: Unblocked variable-length records are specified as RECFM=V; blocked records as RECFM=VB.

For unblocked records, LRECL specifies the maximum length of any record in the data set, plus four additional bytes for the segment control word that precedes each record, and BLKSIZE specifies the buffer length, which is LRECL plus four bytes for the block control word that precedes each block, (e.g., LRECL=84,BLKSIZE=88); the block control word is required even though the record is unblocked. If the record is smaller than the size specified in LRECL, unused space is not written.

For blocked records, LRECL specifies the maximum record length plus four bytes for the segment control word, and BLKSIZE specifies the block length, a number equal to or larger than that specified in LRECL plus four bytes for the block control word, (e.g., LRECL=84,BLKSIZE=350). The block accommodates as many records as possible without exceeding the limit specified in

BLKSIZE. Unused space at the end of the block is not written.

If LRECL is omitted, its default value is set to BLKSIZE-4, resulting in having only one record written in any block.

Undefined-Length Records: Undefined-length records may be specified only as unblocked records, i.e., RECFM=U. BLKSIZE specifies the length of the buffer; this number must take into account the largest possible size record which may be encountered in the FORMAT statement; e.g., BLKSIZE=80 indicates that no record larger than 80 bytes will be encountered. Unused space is not written.

Figure I-22 illustrates the structure of formatted records in sequential data sets, using a FORTRAN record size of 80 characters. An example of a DCB parameter describing a block of ten variable-length records whose maximum size is 121 is:

```
DCB=(RECFM=VB,LRECL=125,BLKSIZE=1254)
```

UNFORMATTED RECORDS: Unformatted records are those not described by a FORMAT statement. The size of each record is determined by the input/output list of READ and WRITE statements. Unformatted records are always specified as variable and spanned. In addition, they may be blocked or unblocked. Unblocked records are specified as RECFM=VS, blocked records as RECFM=VBS. Blocked records reduce processing time substantially and are recommended whenever possible.

For unblocked records, BLKSIZE specifies the length of the buffer and is immaterial to the size of the logical record, i.e., it may be larger than, equal to, or smaller than the logical record. The first eight bytes of the block are reserved for the block and segment control words. For a record smaller than or equal to BLKSIZE(-8), one record per block is transmitted; unused space is not written. For a record larger than BLKSIZE-8, the record is transmitted over as many blocks as necessary to accommodate it; such a record is called a spanned record, since it spans more than one block.

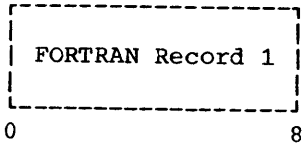
For blocked records, LRECL specifies the maximum record length, and BLKSIZE specifies a block length not necessarily a multiple of LRECL. The block accommodates as many records as possible. If necessary, the last record of a block may span to the next block.

Figure I-23 illustrates the structure of unformatted records in storage.

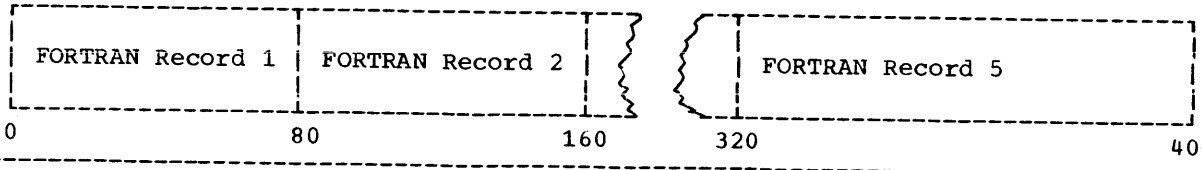
Note: The track overflow feature may be specified with any of the record formats. This feature permits more efficient utilization of track capacity by allowing records to be written when a block size exceeds a track size. The feature is requested by the letter T in the RECFM subparameter, e.g., RECFM=VBT.

FIXED-LENGTH RECORDS

1. Unblocked, e.g., DCB=(RECFM=F,BLKSIZE=80)

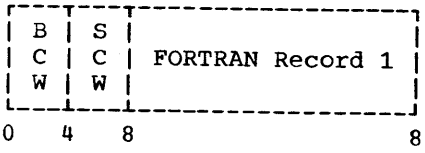


2. Blocked, e.g., DCB=(RECFM=FB,LRECL=80,BLKSIZE=400)

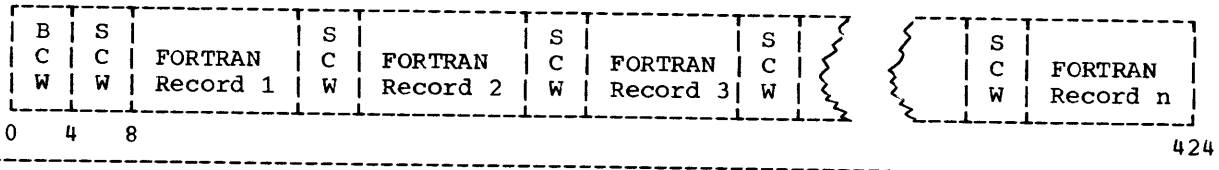


VARIABLE-LENGTH RECORDS

1. Unblocked, e.g., DCB=(RECFM=V,LRECL=84,BLKSIZE=88); record may be any size up to 80 bytes.

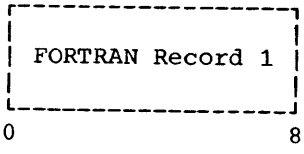


2. Blocked, e.g., DCB=(RECFM=VB,LRECL=84,BLKSIZE=424); a record may be any size up to 80 bytes, and the block will contain as many records as may be accommodated in 424 bytes.



UNDEFINED-LENGTH RECORDS

Unblocked only, e.g., DCB=(RECFM=U,BLKSIZE=80); a record may be any size up to 80 bytes.



Legend:
BCW=Block Control Word
SCW=Segment Control Word

Figure I-22. EBCDIC Sequential Data Sets--Structure of Formatted Records

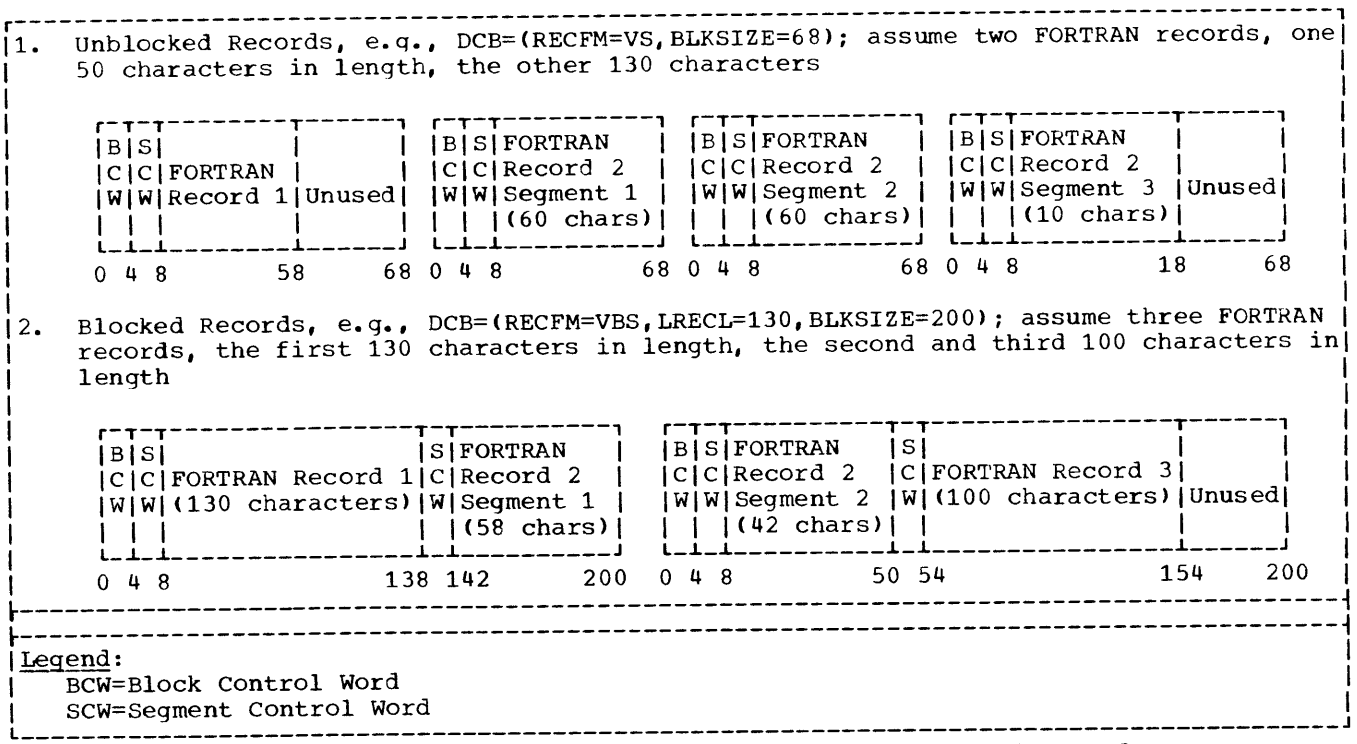


Figure I-23. EBCDIC Sequential Data Sets--Structure of Unformatted Records

DCB Considerations for ASCII Data Sets

ASCII data sets may have sequential organization only and may reside only on 9-track tape having a density of 800 bpi (the DCB subparameter DEN=2 must be implied or explicitly specified). If an ASCII data set has not been identified by means of the LABEL parameter (LABEL=AL), the programmer indicates the presence of an ASCII data set by specifying the DCB subparameter OPTCD=Q.

FORTRAN records in an ASCII data set must be formatted and unspanned and may be fixed-length, undefined-length, or variable-length specified as follows:

Fixed-Length Records: Like EBCDIC records, ASCII records may be blocked or unblocked. Unblocked records are defined as RECFM=F; blocked records as RECFM=FB.

For unblocked records, BLKSIZE specifies the buffer length, which is the record length plus the length of an optional block prefix. The block prefix is a field that, if present, precedes each unblocked record or the first record in a block. BUFOFF specifies the size of the block prefix.

For blocked records, LRECL specifies the record length and BLKSIZE the buffer length, which is the data length (a multiple of LRECL) plus the length of the block prefix if present. BUFOFF specifies the size of the block prefix.

BUFOFF may be specified for input data sets only; the operating system does not use the information contained in the block prefix but skips the number of bytes specified before beginning record processing. If BUFOFF is specified for output data sets, abnormal termination may result.

The following example defines a block of five records, each 80 bytes long, with a block prefix of 20 bytes:

```
DCB=(RECFM=FB,LRECL=80,BLKSIZE=420,
     BUFOFF=20)
```

Undefined-Length Records: Records may be unblocked only, defined as RECFM=U. BLKSIZE specifies the buffer length, which is the size of the largest record that may be encountered in the FORMAT statement plus the size of the block prefix if present. BUFOFF specifies the size of the block prefix. It may be coded for input data sets only. The operating system skips the number of bytes specified before beginning record processing. BUFOFF specified for

output data sets may result in abnormal termination.

The following example specifies a buffer length of 200 bytes with no block prefix:

```
DCB=(RECFM=U,BLKSIZE=200)
```

Variable-Length Records: Records may be blocked or unblocked. Unblocked records are specified as RECFM=D; blocked records as RECFM=DB.

For unblocked records, LRECL specifies the maximum length of any record in the data set plus four bytes for the segment control word that precedes each record. BLKSIZE specifies the buffer length, which is the same size as specified in LRECL plus the size of the block prefix if present. BUFOFF specifies the size of the block prefix.

For blocked records, LRECL specifies the maximum record length plus four bytes for the segment control word. BLKSIZE specifies the buffer length, which is the block length plus the size of the block prefix if present. BUFOFF specifies the size of the block prefix.

BUFOFF may be coded as BUFOFF=L or BUFOFF=n. BUFOFF=L indicates that the block prefix is four bytes long and is to be used in calculating the length of the block. BUFOFF=n indicates the size of the block prefix (where n is a number between 1 and 99); the operating system skips the number of bytes specified before beginning record processing. (Note that BUFOFF=4 is not equivalent to BUFOFF=L.) BUFOFF=L may be specified for both input and output data sets. BUFOFF=n may be specified for input data sets only.

The following example defines a block of up to 10 maximum length records (defined as 100 bytes) with the block prefix to be used to calculate the block length:

```
DCB=(RECFM=DB,LRECL=104,BLKSIZE=1044,
     BUFOFF=L)
```

Notes:

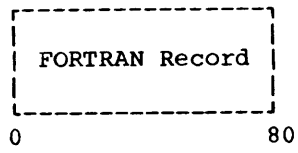
1. ASCII data sets may specify only ASA carriage control characters in the RECFM subparameter (for example, RECFM=FBA); machine control characters (coded M in the RECFM subparameter) are not available for ASCII data sets.

2. For efficient use of storage when writing blocked records, the programmer should always specify LRECL. If LRECL is omitted, its default value is set equal to the value of BLKSIZE less eight bytes. In output operations, the operating system checks to see if there is room in a block for a record of length LRECL; if there is not, the current block is written and a new one is started. The default convention thus results in having only one record written in any block.

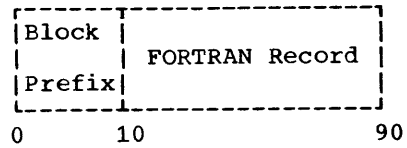
Figure I-24 illlustrates the structure of records in ASCII data sets.

FIXED-LENGTH RECORDS

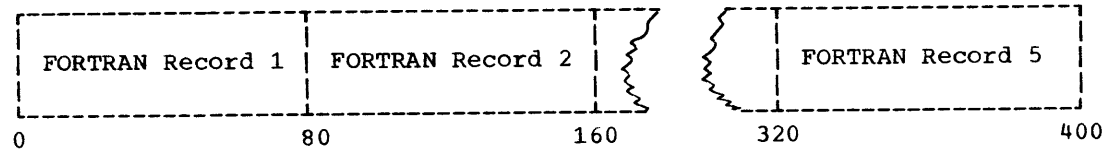
1. Unblocked, with no block prefix, e.g., DCB=(RECFM=F,BLKSIZE=80).



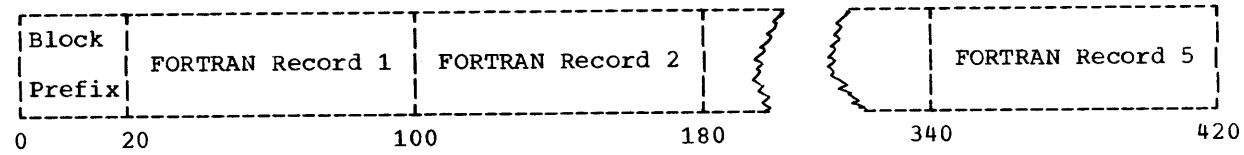
2. Unblocked, with block prefix (input data sets only), e.g., DCB=(RECFM=F,BLKSIZE=90,BUFOFF=10).



3. Blocked, with no prefix, e.g., DCB=(RECFM=F,LRECL=80,BLKSIZE=400).

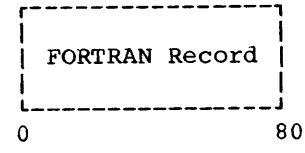


4. Blocked, with block prefix (input data sets only), e.g., DCB=(RECFM=FB,LRECL=80,BLKSIZE=420,BUFOFF=20).



UNDEFINED-LENGTH RECORDS (Unblocked only)

1. With no block prefix, e.g., DCB=(RECFM=U,BLKSIZE=80); a record may be any size up to 80 bytes.



2. With block prefix (input data sets only), e.g., DCB(RECFM=U,BLKSIZE=100,BUFOFF=20).

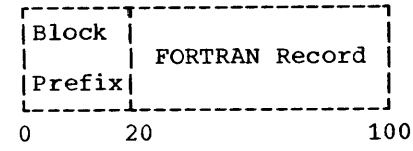
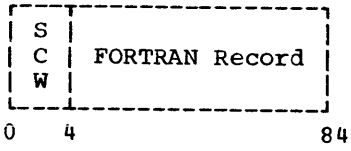


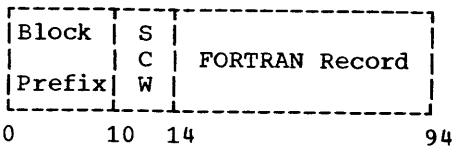
Figure I-24. ASCII Data Sets -- Structure of Records (Part 1 of 2)

VARIABLE-LENGTH RECORDS

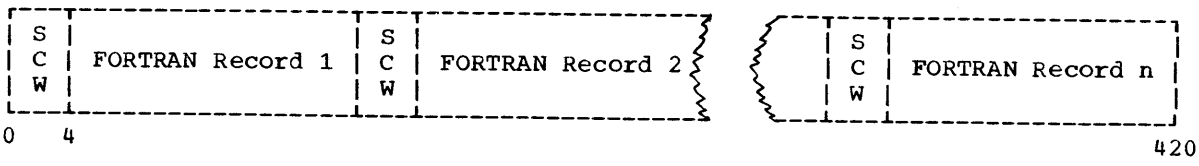
1. Unblocked, no block prefix, e.g., DCB=(RECFM=D,LRECL=84,BLKSIZE=84); a record may be any size up to 80 bytes.



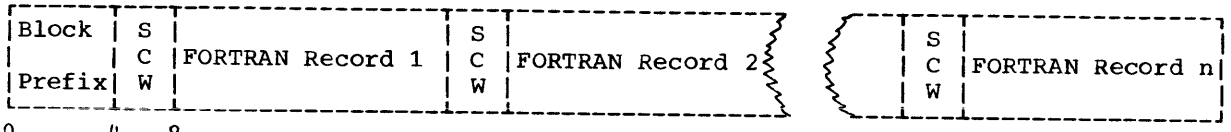
2. Unblocked, with block prefix, e.g., DCB=(RECFM=D,LRECL=84,BLKSIZE=94,BUFOFF=10); a record may be any size up to 80 bytes; a block prefix is not used to determine block length and may be specified only for an input data set.



3. Blocked, no block prefix, e.g., DCB=(RECFM=DB,LRECL=84,BLKSIZE=420); a record may be any size up to 80 bytes, and the block will contain as many records as may be accommodated in 420 bytes.



4. Blocked, with block prefix, e.g., DCB=(RECFM=DB,LRECL=84,BLKSIZE=424,BUFOFF=L); block prefix is used to determine block length.



Legend:
SCW=Segment Control Word

Figure I-24. ASCII Data Sets -- Structure of Records (Part 2 of 2)

DCB Considerations for Direct-access Data Sets

FORTRAN records may be formatted or unformatted, but must be fixed and unblocked only. The DEFINE FILE statement specifies the record length and buffer length for a direct-access data set. This provides the default value for BLKSIZE.

FORMATTED RECORDS: Record format is specified as RECFM=F (fixed). Record length is specified by BLKSIZE, e.g., BLKSIZE=80.

UNFORMATTED RECORDS: Record format is specified as RECFM=F, BLKSIZE specifies a pseudo block size, which, as for sequential data sets, may specify a length different from the record length. Unlike sequential data sets, no bytes are reserved for block or segment control words. For a record smaller than or equal to BLKSIZE, one record per block is transmitted; unused space is left blank. For a record larger than BLKSIZE, the record is transmitted over as many blocks as are required to accommodate it.

Figure I-25 illustrates the structure of records in a direct-access data set.

Notes:

1. Track-overflow, denoted by the letter T in the RECFM subparameter, may be specified to permit more efficient use of track capacity.

2. If a direct-access data set is to be processed by non-FORTRAN programs, DSORG=DA must be specified, i.e.,

   DCB=(RECFM=F,BLKSIZE=80,DSORG=DA)

   This specification causes the creation of a label indicating a direct-access data set. (If the data set is to be processed only by FORTRAN programs, the default specification, DSORG=PS, may be used.)

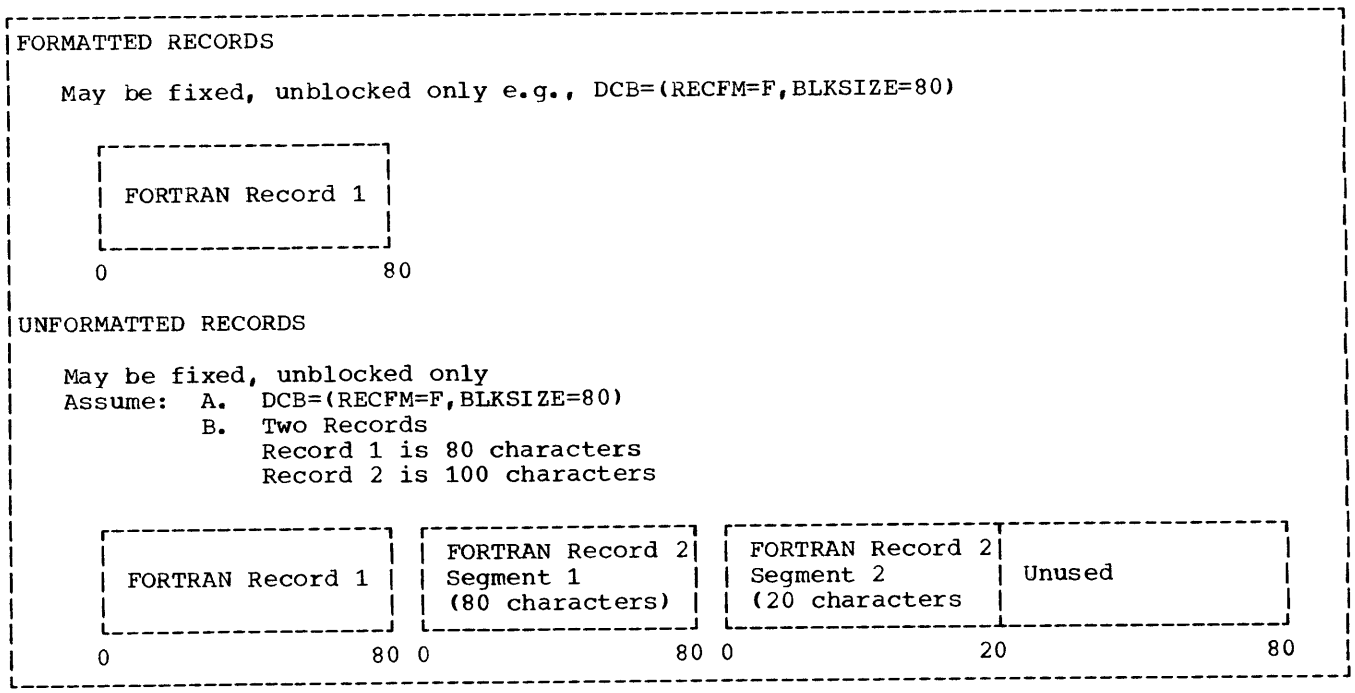


Figure I-25. Direct-Access Data Sets--Structure of Records

Cataloged procedures provide pre-coded DD and EXEC statements for the most common functions, such as compile, or compile and execute. Cataloged procedures may be easily modified to accommodate special situations.

Cataloged procedures are sets of EXEC and DD statements that are placed in the procedure library, SYS1.PROCLIB. Each procedure is retrieved by specifying its name in an EXEC statement; for example, to retrieve the procedure named FORTXC, the user submits the following EXEC statement:

```
// EXEC FORTXC
```

This statement retrieves the cataloged procedure named and places the statements from the procedure into the job stream.

CATALOGED PROCEDURE RESTRICTIONS

Cataloged procedures may consist only of EXEC statements, certain DD statements, and, optionally, PROC statements. PROC statements are used to assign default values to parameters. The PROC statement is discussed further in "Symbolic Parameters and the PROC Statement" later in this chapter.

The following statements may not form part of a cataloged procedure:

1. The JOB statement
2. The JOBLIB DD statement
3. A DD statement containing * or DATA in the operand field
4. The delimiter statement
5. The null statement
6. An EXEC statement that calls another cataloged procedure; that is, only the form PGM=progname is valid

FORTRAN PROCESSING CATALOGED PROCEDURES

IBM supplies seven procedures for use with the FORTRAN IV (H Extended) Compiler. They are:

1. FORTXC, to compile, illustrated in Figure I-26.
2. FORTXCL, to compile and link edit, illustrated in Figure I-27.
3. FORTXLG, to link edit and execute (load module execution), illustrated in Figure I-28.
4. FORTXCLG, to compile, link edit, and execute, illustrated in Figure I-29.
5. FORTXG, to execute, illustrated in Figure I-30.
6. FORTXCG, to compile and load, illustrated in Figure I-31.
7. FORTXL, to load, illustrated in Figure I-32.

The first job control statement in each cataloged procedure is the PROC statement, which assigns default values to symbolic parameters. The statements identified by the characters //* following the PROC statement consist of comments only, giving more information about the symbolic parameters. A full discussion of the PROC statement and symbolic parameters may be found in the appropriate job control language reference publication, as listed in the Preface. The following discussion briefly describes their use in FORTRAN cataloged procedures.

```
MEMBER NAME  FORTXC
//FORTXC PROC  FXPGM=IFEAAB,FXREGN=256K,FXPDECK=DECK,                  +39450000
//             FXPOLST=NOLIST,FXPOPT=0,FXLNSPC='3200,(25,6)'            39500000
//*                                                                     39550000
//*            PARAMETER   DEFAULT-VALUE      USAGE                     39600000
//*                                                                     39650000
//*              FXPGM     IFEAAB             COMPILER NAME             39700000
//*              FXREGN    256K               FORT-STEP REGION          39750000
//*              FXPDECK   DECK               COMPILER DECK OPTION      39800000
//*              FXPOLST   NOLIST             COMPILER LIST OPTION      39850000
//*              FXPOPT    0                  COMPILER OPTIMIZATION     39900000
//*              FXLNSPC   3200,(25,6)       FORT.SYSLIN SPACE          39950000
//*                                                                     40000000
//FORT   EXEC  PGM=&FXPGM,REGION=&FXREGN,COND=(4,LT),                  +40050000
//             PARM='&FXPDECK,&FXPOLST,OPT(&FXPOPT)'                    40100000
//SYSPRINT     DD SYSOUT=A,DCB=BLKSIZE=3429                            40150000
//SYSUT1       DD UNIT=SYSSQ,SPACE=(3465,(3,3)),DCB=BLKSIZE=3465       40200000
//SYSUT2       DD UNIT=SYSSQ,SPACE=(2048,(10,10))                      40250000
//SYSPUNCH     DD SYSOUT=B,DCB=BLKSIZE=3440                            40300000
//SYSLIN       DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSSQ,           +40350000
//             SPACE=(&FXLNSPC),DCB=BLKSIZE=3200                        40400000
```

Figure I-26. Cataloged Procedure FORTXC

```
MEMBER NAME  FORTXCL
//FORTXCL PROC FXPGM=IFEAAB,FXREGN=256K,FXPDECK=NODECK,FXPOLST=NOLIST, +41350000
//             FXPNAME=MAIN,FXPOPT=0,PGMLB='&&GOSET'                    41400000
//*                                                                     41450000
//*            PARAMETER   DEFAULT-VALUE      USAGE                     41500000
//*                                                                     41550000
//*              FXPGM     IFEAAB             COMPILER NAME             41600000
//*              FXREGN    256K               FORT-STEP REGION          41650000
//*              FXPDECK   NODECK             COMPILER DECK OPTION      41700000
//*              FXPOLST   NOLIST             COMPILER LIST OPTION      41750000
//*              FXPNAME   MAIN               COMPILER NAME OPTION      41800000
//*              FXPOPT    0                  COMPILER OPTIMIZATION     41850000
//*              PGMLB     &&GOSET            LKED.SYSLMOD DSNAME       41900000
//*                                                                     41950000
//FORT   EXEC  PGM=&FXPGM,REGION=&FXREGN,COND=(4,LT),                  +42000000
//             PARM='&FXPDECK,&FXPOLST,NAME(&FXPNAME),OPT(&FXPOPT)'     42050000
//SYSPRINT     DD SYSOUT=A,DCB=BLKSIZE=3429                            42100000
//SYSUT1       DD UNIT=SYSSQ,SPACE=(3465,(3,3)),DCB=BLKSIZE=3465       42150000
//SYSUT2       DD UNIT=SYSSQ,SPACE=(2048,(10,10))                      42200000
//SYSPUNCH     DD SYSOUT=B,DCB=BLKSIZE=3440                            42250000
//SYSLIN       DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSSQ,           +42300000
//             SPACE=(3200,(25,6)),DCB=BLKSIZE=3200                    42350000
//LKED   EXEC  PGM=IEWL,REGION=96K,COND=(4,LT),                       +42400000
//             PARM='LET,LIST,MAP,XREF'                                42450000
//SYSPRINT     DD SYSOUT=A                                             42500000
//SYSLIB       DD DSN=SYS1.FORTLIB,DISP=SHR                            42550000
//SYSUT1       DD UNIT=SYSDA,SPACE=(1024,(200,20))                     42600000
//SYSLMOD      DD DSN=&PGMLB.(&FXPNAME),UNIT=SYSDA,                   +42650000
//             DISP=(NEW,PASS),SPACE=(TRK,(10,10,1),RLSE)              42700000
//SYSLIN       DD DSN=&&LOADSET,DISP=(OLD,DELETE)                      42750000
//             DD DDNAME=SYSIN                                         42800000
```

Figure I-27. Cataloged Procedure FORTXCL

```
MEMBER NAME  FORTXLG
//FORTXLG PROC LKLNDD='DDNAME=SYSIN',GOPGM=MAIN,GOREGN=100K,          +44850000
//             GOF5DD='DDNAME=SYSIN',GOF6DD='SYSOUT=A',               +44900000
//             GOF7DD='SYSOUT=B'                                       44950000
//*                                                                    45000000
//*                                                                    45050000
//*            PARAMETER  DEFAULT-VALUE     USAGE                     45100000
//*              LKLNDD   DDNAME=SYSIN      LKED.SYSLIN OPERAND       45150000
//*              GOPGM    MAIN              OBJECT PROGRAM NAME       45200000
//*              GOREGN   100K              GO-STEP REGION            45250000
//*              GOF5DD   DDNAME=SYSIN      GO.FT05F001 OPERAND       45300000
//*              GOF6DD   SYSOUT=A          GO.FT06F001 OPERAND       45350000
//*              GOF7DD   SYSOUT=B          GO.FT07F001 OPERAND       45400000
//*                                                                    45450000
//LKED   EXEC  PGM=IEWL,REGION=96K,COND=(4,LT),                       +45500000
//             PARM='LET,LIST,MAP,XREF'                                45550000
//SYSPRINT     DD SYSOUT=A                                            45600000
//SYSLIB       DD DSN=SYS1.FORTLIB,DISP=SHR                           45650000
//SYSUT1       DD UNIT=SYSDA,SPACE=(1024,(200,20))                    45700000
//SYSLMOD      DD DSN=&&GOSET(&GOPGM),DISP=(,PASS),UNIT=SYSDA,        +45750000
//             SPACE=(TRK,(10,10,1),RLSE)                              45800000
//SYSLIN       DD &LKLNDD                                             45850000
//GO     EXEC  PGM=*.LKED.SYSLMOD,REGION=&GOREGN,COND=(4,LT)          45900000
//FT05F001     DD &GOF5DD                                             45950000
//FT06F001     DD &GOF6DD                                             46000000
//FT07F001     DD &GOF7DD                                             46050000
```

Figure I-28. Cataloged Procedure FORTXLG

```
MEMBER NAME  FORTXCLG
//FORTXCLG PROC FXPGM=IFEAAB,FXREGN=256K,FXPDECK=NODECK,                +42900000
//              FXPOLST=NOLIST,FXPOPT=0,GOREGN=100K,                    +42950000
//              GOF5DD='DDNAME=SYSIN',GOF6DD='SYSOUT=A',                +43000000
//              GOF7DD='SYSOUT=B'                                        43050000
//*                                                                      43100000
//*             PARAMETER   DEFAULT-VALUE      USAGE                    43150000
//*                                                                      43200000
//*               FXPGM     IFEAAB             COMPILER NAME            43250000
//*               FXREGN    256K               FORT-STEP REGION         43300000
//*               FXPDECK   NODECK             COMPILER DECK OPTION     43350000
//*               FXPOLST   NOLIST             COMPILER LIST OPTION     43400000
//*               FXPOPT    0                  COMPILER OPTIMIZATION    43450000
//*               GOREGN    100K               GO-STEP REGION           43500000
//*               GOF5DD    DDNAME=SYSIN       GO.FT05F001 OPERAND      43550000
//*               GOF6DD    SYSOUT=A           GO.FT06F001 OPERAND      43600000
//*               GOF7DD    SYSOUT=B           GO.FT07F001 OPERAND      43650000
//*                                                                      43700000
//FORT    EXEC  PGM=&FXPGM,REGION=&FXREGN,COND=(4,LT),                  +43750000
//              PARM='&FXPDECK,&FXPOLST,OPT(&FXPOPT)'                    43800000
//SYSPRINT      DD SYSOUT=A,DCB=BLKSIZE=3429                             43850000
//SYSUT1        DD UNIT=SYSSQ,SPACE=(3465,(3,3)),DCB=BLKSIZE=3465        43900000
//SYSUT2        DD UNIT=SYSSQ,SPACE=(5048,(10,10))                       43950000
//SYSPUNCH      DD SYSOUT=B,DCB=BLKSIZE=3440                             44000000
//SYSLIN        DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSSQ,            +44050000
//              SPACE=(3200,(25,6)),DCB=BLKSIZE=3200                     44100000
//LKED    EXEC  PGM=IEWL,REGION=96K,COND=(4,LT),                        +44150000
//              PARM='LET,LIST,MAP,XREF'                                 44200000
//SYSPRINT      DD SYSOUT=A                                              44250000
//SYSLIB        DD DSN=SYS1.FORTLIB,DISP=SHR                             44300000
//SYSUT1        DD UNIT=SYSDA,SPACE=(1024,(200,20))                      44350000
//SYSLMOD       DD DSN=&&GOSET(MAIN),DISP=(,PASS),UNIT=SYSDA,           +44400000
//              SPACE=(TRK,(10,10,1),RLSE)                               44450000
//SYSLIN        DD DSN=&&LOADSET,DISP=(OLD,DELETE)                       44500000
//              DD DDNAME=SYSIN                                          44550000
//GO      EXEC  PGM=*.LKED.SYSLMOD,REGION=&GOREGN,COND=(4,LT)           44600000
//FT05F001      DD &GOF5DD                                               44650000
//FT06F001      DD &GOF6DD                                               44700000
//FT07F001      DD &GOF7DD                                               44750000
```

Figure I-29. Cataloged Procedure FORTXCLG

```
MEMBER NAME  FORTXG
//FORTXG PROC  GOPGM=MAIN,GOREGN=100K,                                  +40500000
//             GOF5DD='DDNAME=SYSIN',GOF6DD='SYSOUT=A',                 +40550000
//             GOF7DD='SYSOUT=B'                                         40600000
//*                                                                      40650000
//*            PARAMETER   DEFAULT-VALUE      USAGE                      40700000
//*                                                                      40750000
//*              GOPGM     MAIN               PROGRAM NAME               40800000
//*              GOREGN    100K               GO-STEP REGION             40850000
//*              GOF5DD    DDNAME=SYSIN       GO.FT05F001 DD OPERAND     40900000
//*              GOF6DD    SYSOUT=A           GO.FT06F001 DD OPERAND     40950000
//*              GOF7DD    SYSOUT=B           GO.FT07F001 DD OPERAND     41000000
//*                                                                      41050000
//GO      EXEC  PGM=&GOPGM,REGION=&GOREGN,COND=(4,LT)                    41100000
//FT05F001      DD &GOF5DD                                               41150000
//FT06F001      DD &GOF6DD                                               41200000
//FT07F001      DD &GOF7DD                                               41250000
```

Figure I-30. Cataloged Procedure FORTXG

```
MEMBER NAME  FORTXCG
//FORTXCG PROC FXPGM=IFEAAB,FXREGN=256K,FXPDECK=NODECK,                 +46150000
//             FXPOLST=NOLIST,FXPOPT=0,GOF5DD='DDNAME=SYSIN',           +46200000
//             GOF6DD='SYSOUT=A',GOF7DD='SYSOUT=B',GOREGN=100K           46250000
//*                                                                      46300000
//*            PARAMETER  DEFAULT-VALUE     USAGE                      46350000
//*                                                                     46400000
//*              GOREGN   100K              GO-STEP REGION             46450000
//*              FXPGM    IFEAAB            COMPILER NAME              46500000
//*              FXREGN   256K              FORT-STEP REGION           46550000
//*              FXPDECK  NODECK            COMPILER DECK OPTION       46600000
//*              FXPOLST  NOLIST            COMPILER LIST OPTION       46650000
//*              FXPOPT   0                 COMPILER OPTIMIZATION      46700000
//*              GOF5DD   DDNAME=SYSIN      GO.FT05F001 OPERAND        46750000
//*              GOF6DD   SYSOUT=A          GO.FT06F001 OPERAND        46800000
//*              GOF7DD   SYSOUT=B          GO.FT07F001 OPERAND        46850000
//*                                                                     46900000
//FORT   EXEC  PGM=&FXPGM,REGION=&FXREGN,COND=(4,LT),                  +46950000
//             PARM='&FXPDECK,&FXPOLST,OPT(&FXPOPT)'                    47000000
//SYSPRINT     DD  SYSOUT=A,DCB=BLKSIZE=3429                           47050000
//SYSUT1       DD  UNIT=SYSSQ,SPACE=(3465,(3,3)),DCB=BLKSIZE=3465      47100000
//SYSUT2       DD  UNIT=SYSSQ,SPACE=(2048,(10,10))                     47150000
//SYSPUNCH     DD  SYSOUT=B,DCB=BLKSIZE=3440                           47200000
//SYSLIN       DD  DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSSQ,           +47250000
//             SPACE=(3200,(25,6)),DCB=BLKSIZE=3200                    47300000
//GO     EXEC  PGM=LOADER,COND=(4,LT),REGION=&GOREGN,                 +47350000
//             PARM='LET,NORES,EP=MAIN'                                47400000
//SYSLOUT      DD  SYSOUT=A                                            47450000
//SYSLIB       DD  DSN=SYS1.FORTLIB,DISP=SHR                           47500000
//SYSLIN       DD  DSN=&&LOADSET,DISP=(OLD,DELETE)                     47550000
//FT05F001     DD  &GOF5DD                                             47600000
//FT06F001     DD  &GOF6DD                                             47650000
//FT07F001     DD  &GOF7DD                                             47700000
```

Figure I-31. Cataloged Procedure FORTXCG

```
MEMBER NAME  FORTXL
//FORTXL   PROC GOF5DD='DDNAME=SYSIN',GOF6DD='SYSOUT=A',                +47800000
//             GOF7DD='SYSOUT=B',GOREGN=100K                             47850000
//*                                                                      47900000
//*            PARAMETER   DEFAULT-VALUE      USAGE                      47950000
//*                                                                      48000000
//*              GOF5DD    DDNAME=SYSIN      GO.FT05F001 OPERAND        48050000
//*              GOF6DD    SYSOUT=A          GO.FT06F001 OPERAND        48100000
//*              GOF7DD    SYSOUT=B          GO.FT07F001 OPERAND        48150000
//*              GOREGN    100K              GO-STEP REGION             48200000
//*                                                                      48250000
//GO       EXEC PGM=LOADER,COND=(4,LT),REGION=&GOREGN,                  +48300000
//             PARM='LET,NORES,EP=MAIN'                                  48350000
//SYSLOUT      DD SYSOUT=A                                               48400000
//SYSLIB       DD DSN=SYS1.FORTLIB,DISP=SHR                              48450000
//FT05F001     DD &GOF5DD                                                48500000
//FT06F001     DD &GOF6DD                                                48550000
//FT07F001     DD &GOF7DD                                                48600000
```

Figure I-32. Cataloged Procedure FORTXL

Symbolic Parameters and the PROC Statement

A symbolic parameter is a name preceded by an ampersand (&). It appears in the operand field of a statement and stands as a symbol for a parameter, a subparameter, or a value. For example, in the cataloged procedure FORTXC, the EXEC statement named FORT contains a number of symbolic parameters, such as &FXPGM and &FXREGN in the expressions PGM=&FXPGM and REGION=&FXREGN.

Symbolic parameters are used to make a cataloged procedure easily modified when it is called. The programmer may assign values to symbolic parameters when he calls a cataloged procedure, or he may permit the default value assigned by the PROC statement to be in effect.

Figure I-33 illustrates the format of the PROC statement. Since the statement deals only with the symbolic parameters, the identifying ampersand is unnecessary and is omitted. In the cataloged procedure FORTXC, the PROC statement assigns default values to &FXPGM and &FXREGN in the manner:

```
//FORTXC PROC  FXPGM=IFEAAB,FXREGN=228K,...
```

When the cataloged procedure is executed, these default values are assigned if the programmer does not override them. That is, the EXEC statement named FORT would appear as if it were coded:

```
//FORT  EXEC  PGM=IFEAAB,REGION=228K,...
```

Figure I-34 is another example illustrating how the PROC statement affects symbolic parameters of a cataloged procedure. For easier reference, the symbolic parameters in Figure I-34 are shown underscored.

Name	Operation	Operand
//[name]	PROC	symbolic-parameter=value[,...]

Figure I-33. PROC Statement Format

```
PROC Statement in cataloged procedure FORTXCLG:

//FORTXCLG  PROC    FXPGM=IFEAAB,FXREGN=228K,FXPDECK=NODECK,
//                  FXPOLST=NOLIST,FXPOPT=0,GOREGN=100K,
//                  GOF5DD='DDNAME=SYSIN',GOF6DD='SYSOUT=A',
//                  GOF7DD='SYSOUT=B'


Statements in FORTXCLG specifying symbolic parameters:

//FORT    EXEC      PGM=&FXPGM,REGION=&FXREGN,COND=(4,LT),
//                  PARM='&FXPDECK,&FXPOLST,OPTIMIZE(&FXPOPT)'
                    .
                    .
                    .
//GO      EXEC      PGM=*.LKED.SYSLMOD,REGION=&GOREGN,COND=(4,LT)
//FT05F001  DD      &GOF5DD
//FT06F001  DD      &GOF6DD
//FT07F001  DD      &GOF7DD


Substitution of default values at execution time:

//FORT    EXEC      PGM=IFEAAB,REGION=228K,COND=(4,LT),
//                  PARM='NODECK,NOLIST,OPTIMIZE(0)'
                    .
                    .
                    .
//GO      EXEC      PGM=*.LKED.SYSLMOD,REGION=100K,COND=(4,LT)
//FT05F001  DD      DDNAME=SYSIN
//FT06F001  DD      SYSOUT=A
//FT07F001  DD      SYSOUT=B
```

Figure I-34. Effect of PROC Statement in a Cataloged Procedure

COMPILING

The EXEC statement for the compilation step is named FORT and, through the PGM parameter, specifies the compiler as the program to be executed (PGM=IFEAAB). The DD statements describe data sets required by the compiler. SYSLIN describes the output of the compilation step, an object module stored as a temporary data set named &&LOADSET. (A double ampersand is assigned to avoid confusion with symbolic parameters, which are preceded by one ampersand.) The DISP parameter is coded (MOD,PASS); MOD permits more than one object module to be stored (if many source modules are submitted for compilation), and PASS permits the data set to be used in later job steps. The programmer specifies the source module data set in a SYSIN DD statement coded as follows:

```
//FORT.SYSIN  DD  * (or appropriate
                    parameters to define
                    the data set)
```

LINK EDITING

The EXEC statement for the link edit step is named LKED and specifies the linkage editor as the program to be executed (PGM=IEWL). In FORTXCL and FORTXCLG, the EXEC statement COND parameter indicates that the program is to be executed only if the FORT step has returned a code less than or equal to 4. (Each job step issues a return code indicating the results of processing, e.g., 0 for normal completion, 4 for minor errors detected, 8 for serious errors.) The DD statements describe required data sets. SYSLMOD describes the output of the link edit step, a load module named MAIN, which is stored as a member of a temporary library named &&GOSET. SYSLIN describes the input to the linkage editor. When the linkage editor is to be the first step executed, as in FORTXLG, SYSLIN indicates the object module defined by a SYSIN DD statement which the programmer must supply, as follows:

```
//LKED.SYSIN DD * (or appropriate
                    parameters)
```

EXECUTING THE LOAD MODULE

The EXEC statement for the go step is named GO, and specifies, as the program to be executed, the load module created in the link edit step (PGM=*.LKED.SYSLMOD). The COND parameter indicates that the program is to be executed only if previous steps returned a code less than or equal to 4. The DD statements describe required data sets. DD statement FT05F001 indicates that the input data set is to be defined by a SYSIN DD statement which the programmer must supply. FT06F001 defines a printer data set; FT07F001 a card punch data set.

The programmer specifies input to the load module by a SYSIN DD statement coded as follows:

```
//GO.SYSIN DD * (or appropriate parameters)
```

LOADING

The EXEC statement for the loader step is named GO, and specifies the loader as the program to be executed (PGM=LOADER). The DD statements describe required data sets. SYSLOUT describes printed output, such as a module map. The other data sets are the same ones as used by the linkage editor and the load module. Note that a SYSLMOD DD statement is not specified; the loader places the load module directly into storage for execution. When the loader is to be the first step executed, as in FORTXL, the object module must be defined in a SYSLIN DD statement (not SYSIN), supplied by the programmer, as follows:

```
//GO.SYSLIN  DD  *  (or appropriate
                    parameters)
```

Input to the load module is defined in a SYSIN DD statement, as follows:

```
//GO.SYSIN  DD  *  (or appropriate
                    parameters)
```

MODIFYING CATALOGED PROCEDURES

Except for the PGM parameter, any parameter in the PROC, EXEC, or DD statements may be modified. New parameters may be added; existing parameters may be overridden. Parameters not overridden continue to remain in effect.

When a cataloged procedure is modified, the changes apply only for the duration of the job.

Figure I-35 at the end of this chapter illustrates how a programmer may modify a cataloged procedure using some of the examples described below.

MODIFYING PROC STATEMENTS

The programmer modifies PROC statement parameters by specifying the changes in the EXEC statement that calls the procedure. When he changes a PROC statement parameter, the programmer is assigning a temporary value to a symbolic parameter, and this value is transferred to the appropriate parameter in the EXEC or DD statement in the cataloged procedure when it is executed.

For example, to change the region size of the compiler from 228K to 200K, and to change card punch output in the load module from output class B to output class C, the programmer may use the following statement (assume that changes are being made to FORTXCLG for this and for all examples in this chapter):

```
//  EXEC  FORTXCLG,FXREGN=200K,
//        GOF7DD='SYSOUT=C'
```

Note that the ampersand preceding a symbolic parameter is not coded; note also that a value containing a special character, as in SYSOUT=C, is enclosed in apostrophes.

Prior to being called, the appropriate statements in FORTXCLG appear as follows:

```
//FORT  EXEC  PGM=&FXPGM,REGION=&FXREGN,...
          .
          .
          .
//FT07F001  DD  &GOF7DD
```

When the cataloged procedure is called, the statements appear as though they were coded:

```
//FORT  EXEC  PGM=IFEAAB,REGION=200K,...
          .
          .
          .
//FT07F001  DD  SYSOUT=C
```

Note that a symbolic parameter not changed (PGM=&FXPGM) retains its default value.

An alternative method to change a value is by assigning the new value directly to the parameter itself, not the symbolic parameter associated with it. For example, the region size may be changed by specifying REGION=200K in place of FXREGN=200K. (Actually, in this example,

the region size for all job steps would be changed; to change the region size only for the compile job step, the appropriate job step name, FORT, must also appear in the parameter, i.e., REGION.FORT=200K.)

MODIFYING EXEC STATEMENTS

The programmer modifies EXEC statement parameters by specifying the changes in the EXEC statement that calls the procedure.

The following rules apply to EXEC statement modifications:

- Parameters are overridden in their entirety. If the programmer wishes to retain some options while changing others, he must respecify the options he wishes kept. (However, default options remain in effect if not overridden.)
- Parameters specified for individual job steps use the form:

  keyword.stepname=value

where:
keyword indicates the name of the parameter
stepname indicates the name of the procedure, for example, REGION.FORT=value

Parameters not specifying stepname are assumed to apply to all steps in the procedure; for example, REGION=value applies to the entire cataloged procedure.

- To make changes to more than one step, the programmer must specify all changes for an earlier step before those for later steps.
- Changes to symbolic parameters and EXEC statement parameters may be combined on the same card.

The programmer may make the following modifications:

1. Override existing parameters: For example, to modify the LKED step by raising the condition code from 4 to 8, he may use the statement:

```
//SOMENAME EXEC FORTXCLG,
//             COND.LKED=(8,LT)
```

2. Add new parameters: For example, to modify FORT by specifying the TIME parameter, he may use the statement:

```
//ANYNAME  EXEC  FORTXCLG,TIME.FORT=5
```

3. Change more than one parameter: For example, to modify FORT by changing the region from 228K to 200K and the PARM option NOLIST to LIST, he may use the statement:

```
//SOME EXEC FORTXCLG,
//          REGION.FORT=200K,
//          PARM.FORT=LIST
```

4. Change more than one step: For example, to modify FORT by specifying TIME and to modify LKED by raising the condition code from 4 to 8, he may use the statement:

```
//ANY  EXEC  FORTXCLG,TIME.FORT=5,
//           COND.LKED=(8,LT)
```

Note that the user may add a parameter while revising an existing one.

5. Combine changes to symbolic parameters and EXEC statement parameters: For example, to modify the symbolic parameter FXREGN, and to add the TIME parameter to the FORT EXEC statement, he may use the statement:

```
//ANY  EXEC  FORTXCLG,FXREGN=200K,
//           TIME.FORT=5
```

MODIFYING DD STATEMENTS

The programmer modifies DD statements by submitting new DD statements after the EXEC statement that calls the procedure. As with modifications to EXEC statements, the user may override or add parameters to DD statements in one or many steps. In addition, he may add entirely new DD statements to any step (whenever he supplies a SYSIN DD statement, the programmer is adding a new DD statement).

The following rules apply to DD statement modifications:

- Parameters are overriden in their entirety except for the DCB parameter where individual subparameters may be overridden
- Parameters are nullified by specifying a comma after the equal sign in the parameter, e.g., UNIT=,
- Parameters are overridden when mutually exclusive parameters are specified in their place, e.g., SPLIT overrides SPACE

- DD statements must indicate the related procedure step, using the form //stepname.ddname, e.g., //FORT.SYSIN

- To make changes in more than one step, the user must specify all changes for an earlier step before those for later steps

- To modify more than one DD statement in a job step, the programmer must specify the applicable DD statements in the same sequence as they appear in the cataloged procedure

The programmer may make the following modifications:

1. Override existing parameters. For example, to modify SYSLMOD so that the load module is stored in a private library rather than in the system library, the user may submit the statement:

```
//LKED.SYSLMOD DD DSNAME=PRIV(PROG),
//                DISP=(MOD,PASS)
```

   In this example the library PRIV is assumed to be an old library and is cataloged (that is, VOLUME and UNIT parameters need not be specified). Note that in subsequent uses of the library a JOBLIB DD statement, defining the private library, must also be submitted to make the library available to the system.

2. Add new parameters. For example, to store the load module in a new, uncataloged library, the programmer must specify the VOLUME, UNIT, and SPACE parameters. He may submit the statement:

```
//LKED.SYSLMOD DD DSNAME=MYLIB(FIRST),
//      DISP=(NEW,PASS),
//      VOLUME=SER=11234,
//      UNIT=SYSDA,
//      SPACE=(TRK,(50,10,2))
```

3. Add new DD statements. For example, to add new data sets having data set reference numbers 10 and 15 for processing in the go step, the user may submit the statements:

```
//GO.FT10F001  DD  DSNAME=DSET1,
//                 DISP=(NEW,DELETE),
//                 VOLUME=SER=T1132,
//                 UNIT=TAPE
//GO.FT15F001  DD  DSNAME=DSET2,
//                 DISP=(,DELETE),
//                 VOLUME=SER=DA45,
//                 UNIT=2311,
//                 SPACE=(TRK,(10,10))
```

   Note that the user may explicitly define a data set as new (DISP parameter for FT10F001) or may permit the system to assume a new data set by default (DISP in FT15F001).

Figure I-35 illustrates a deck setup for FORTXCLG modified as follows:

- A SYSIN DD statement defines the source module

- A SYSIN DD statement defines input data to the load module

- Job steps are modified as shown in the EXEC statement discussion in "Modifying EXEC Statements", example 4 (TIME in FORT, COND in LKED)

- The SYSLMOD DD statement is modified as shown in the DD statement discussion above, example 1

- Additional data sets to the load module are defined as shown in the DD statement discussion, example 3

Note that all changes to a job step appearing earlier in the job processing sequence must be made before changes for later job steps.

```
//TEST  JOB  ACCT3,J.SMITH,MSGLEVEL=1
//JOBLIB  DD  DSNAME=PRIV,DISP=(MOD,PASS)
//ANY  EXEC  FORTXCLG,TIME.FORT=5,COND.LKED=(8,LT)
//FORT.SYSIN  DD  *

  +--------------+
  |Source module|
  +--------------+

/*
//LKED.SYSLMOD  DD  DSNAME=PRIV(PROG),DISP=(MOD,PASS)
//GO.FT10F001   DD  DSNAME=DSET1,DISP=(NEW,DELETE),VOLUME=SER=T1132,
//                  UNIT=TAPE
//GO.FT15F001   DD  DSNAME=DSET2,DISP=(,DELETE),VOLUME=SER=DA45,
//                  UNIT=2311,SPACE=(TRK,(10,10))
//GO.SYSIN  DD  *

  +------------------+
  |Load module input|
  +------------------+

/*
```

Figure I-35. Submitting Modifications to a Cataloged Procedure

PART II -- JOB OUTPUT

Part II describes job step output for the FORTRAN program depicted in Figures II-1 and II-2. Figure II-1 shows a program as coded. Figure II-2 shows the program as keypunched; keypunch errors have been introduced on purpose to provide instances of system diagnostic action.

IBM FORTRAN Coding Form

PROGRAM SAMPLE PROGRAM DATE 7/70 GRAPHIC PUNCH PAGE 1 OF 1

PROGRAMMER PUNCHING INSTRUCTIONS CARD ELECTRO NUMBER

STATEMENT NUMBER	CONT	FORTRAN STATEMENT	IDENTIFICATION SEQUENCE
C		PRIME NUMBER PROBLEM	
	100	WRITE (6,8)	
	8	FORMAT (52H FOLLOWING IS A LIST OF PRIME NUMBERS FROM 1 TO 1000/	
		119X,1H1/19X,1H2/19X,1H3)	
	101	I=5	
	3	A=I	
	102	A=SQRT(A)	
	103	J=A	
	104	DO 1 K=3,J,2	
	105	L=I/K	
	106	IF(L*K-I)1,2,4	
	1	CONTINUE	
	107	WRITE (6,5)I	
	5	FORMAT (I20)	
	2	I=I+2	
	108	IF(1000-I)7,4,3	
	4	WRITE (6,9)	
	9	FORMAT (14H PROGRAM ERROR)	
	7	WRITE (6,6)	
	6	FORMAT (31H THIS IS THE END OF THE PROGRAM)	
	109	STOP	
		END	

*A standard card form. IBM electro 888157, is available for punching statements from this form

Figure II-1. Sample Program as Coded

```
C       PRIME NUMBER PROBLEM
   100  WRITE (6,8)
     8  FORMAT (52H FOLLOWING IS A LIST OF PRIME NUMBERS FROM 1 TO 1000/
       119X,1H1/19X,1H2/19X,1H3)
   101  I=5
     3  A=I
   102  A=SQRT(A)
   103  J=A
   104  DO I K=3,J,2
   105  L=I/K
   106  IF (L*K-I)1,2,4
        CONTINUE
     1  CONTINUE
   107  WRITE (3,5)I
     5  FORMAT (I2O)
     2  I=I+2
   108  IF (1000-I)7,4,3
     4  WRITE (6,9)
     9  FORMAT (14H PROGRAM ERROR)
     7  WRITE (6,6)
     6  FORMAT (31H THIS IS THE END OF THE PROGRAM)
   109  STOP
        END
```

Figure II-2. Sample Program as Keypunched

Each compilation produces the following:

- Informative messages letting the programmer know the status of the compilation
- Any diagnostic messages generated during the compilation
- Output as determined by the options selected by the programmer, either explicitly or by default

The programmer may request the compiler options he wishes to exercise through the PARM parameter of the EXEC statement, or he may permit default options to govern compiler output.

Figure II-3 shows the sequence in which output from compiler options is printed.

In addition to the options shown in Figure II-3, the programmer may request the DECK option, which produces no listing, but generates a card deck of the object module.

COMPILER OUTPUT WITH DEFAULT OPTIONS

Default options cause the compiler to produce informative messages, diagnostic messages, compiler statistics, and a listing of the source module. Figure II-4 shows a printout for the FORTRAN program illustrated in Figure II-2.

Informative Messages

The first line of a compilation output listing (Figure II-4, A) states the release level number of the compiler, the compiler's name, and the date of the run shown in the format

year.date/hour.minute.second.

Date information in Figure II-4 is shown as 70.002/12.04.4, indicating the year 1970, the second day of the year, and the time of the day the job was completed, 12:04.4 (based on a 24-hour clock).

Option	Produces
--	Informative messages
SOURCE*	Source module listing
XREF	Cross-reference listing
LIST	Object module listing: Part I: Entry code, constants, external address constants
FORMAT	Edited source module listing
LIST	Object module listing: Part II: Executable instructions, internal address constants
MAP	Source module map
--	Diagnostic messages
--	Compiler statistics

*Specified as default option in cataloged procedures.

Figure II-3. Compiler Printed Output Format

```
(A)→ LEVEL   1 ( MAR 71 )          MAIN          OS/360   FORTRAN H EXTENDED          DATE 71.057/09.59.01     PAGE    1

(B) REQUESTED OPTIONS: NODECK,NOLIST,OPT=0

    OPTIONS IN EFFECT: NAME(  MAIN),NOOPTIMIZE,LINECOUNT(60),SIZE(MAX),AUTODBL(NONE),
                       SOURCE,EBCDIC,NOLIST,NODECK,OBJECT,NOMAP,NOFORMAT,NOGOSTMT,NOXREF,NOALC,NOANSF,FLAG(I)

                     C     PRIME NUMBER PROBLEM
    ISN 0002           100 WRITE (6,8)
    ISN 0003             8 FORMAT (52H FOLLOWING IS A LIST OF PRIME NUMBERS FROM 1 TO 1000/
                          119X,1H1/19X,1H2/19X,1H3)
    ISN 0004           101 I=5
    ISN 0005             3 A=I
    ISN 0006           102 A=SQRT(A)
    ISN 0007           103 J=A
    ISN 0008           104 DO 1 K=3,J,2
    ISN 0009           105 L=I/K
    ISN 0010           106 IF(L*K-I)1,2,4
(C) ISN 0011               CONTINUE
    ISN 0012             1 CONTINUE
    ISN 0013           107 WRITE(3,5)I
    ISN 0014             5 FORMAT (I20)
    ISN 0015             2 I=I+2
    ISN 0016           108 IF(1000-I)7,4,3
    ISN 0017             4 WRITE (6,9)
    ISN 0018             9 FORMAT (14H PROGRAM ERROR)
    ISN 0019             7 WRITE (6,6)
    ISN 0020             6 FORMAT (31H THIS IS THE END OF THE PROGRAM)
    ISN 0021           109 STOP
    ISN 0022               END

(D) NUMBER   LEVEL                    FORTRAN H EXTENDED ERROR MESSAGES

    IFE2241   8(E)      ISN    0011    THE STATEMENT AFTER AN ARITHMETIC IF, GO TO, OR RETURN HAS NO LABEL.
    IFE2081   4(W)      ISN    0011    THE CONTINUE STATEMENT DOES NOT HAVE A STATEMENT NUMBER.

(E) *OPTIONS IN EFFECT*NAME(  MAIN),NOOPTIMIZE,LINECOUNT(60),SIZE(MAX),AUTODBL(NONE),

    *OPTIONS IN EFFECT*SOURCE,EBCDIC,NOLIST,NODECK,OBJECT,NOMAP,NOFORMAT,NOGOSTMT,NOXREF,NOALC,NOANSF,FLAG(I)

    *STATISTICS*     SOURCE STATEMENTS =      21, PROGRAM SIZE =      706, SUBPROGRAM NAME =  MAIN

    *STATISTICS*    2 DIAGNOSTICS GENERATED, HIGHEST SEVERITY CODE IS  8

    ****** END OF COMPILATION ******                                    1C5K BYTES OF CORE NOT USED
```

Figure II-4. Compiler Output from Default Options

The second line (and a third line, if needed) shows the compiler options current at the time of job submission (Figure II-4, B).

At the end of the listing, compiler statistics are printed (Figure II-4, E) listing all current options, the number of statements in the source module, the source module size in decimal, and the number and severity of diagnostic messages.

The last entry of a compilation is the informative message:

******END OF COMPILATION******

Diagnostic Messages

Compiler diagnostic messages are assigned severity codes as follows:

Severity Code	Meaning
0	Indicates an informational message; messages with a 0 severity code act as notes to the programmer
4	Is a warning message code; usually, a minor error, which does not violate the syntax of the FORTRAN IV language was detected
8	Is an error message code; usually, an error which violates FORTRAN IV syntax was detected. The compiler attempts to make a corrective assumption.
12	Is a serious error message code; an error which violates FORTRAN IV syntax and for which the compiler could make no corrective assumption was detected
16	Is an abnormal termination message code; an error which prevents the compiler from continuing program processing was detected

Messages with severity code 4 permit the compiled object module to be passed to the link edit step. Severity levels higher than level 4 prevent link edit processing, unless the programmer has increased the permissible condition code in the COND parameter of the compilation EXEC statement.

Diagnostic messages in Figure II-4 are marked D. Messages display the call letters IFE identifying the FORTRAN IV (H Extended) compiler, an internal statement number developed from the original source statement, the severity code, and explanatory text. In Figure II-4, two diagnostic messages were generated by statement 11, an extraneous CONTINUE statement.

Source Listing

The source listing in Figure II-4 is marked C. Source listing statements are identical to the original statements submitted in the FORTRAN program, except for the addition of internal sequence numbers (ISN).

COMPILER OUTPUT WITH PROGRAMMER-SPECIFIED OPTIONS

The compiler always produces informative and diagnostic messages. The programmer may suppress the source listing by specifying NOSOURCE and may also choose to generate other forms of output.

Figures II-5 and II-6 show additional output for the program illustrated in Figure II-2. Figure II-5 illustrates the following:

1. A cross-reference listing
2. An object module listing
3. An edited source module listing
4. A source module map

Figure II-6 shows the deck setup of an object card deck.

Cross-Reference Listing

The programmer requests a cross-reference listing by specifying the compiler option XREF and a DD statement named SYSUT2. The cross-reference listing in Figure II-5 is marked A.

A cross-reference listing shows the symbols and statement labels in the source module together with the internal statement numbers in which they appear. Symbols (which define variables) are listed by name, according to length, in alphabetic order, beginning with names one character long. Statement labels are listed in ascending order and display each internal sequence number (ISN) in which they are referenced.

```
LEVEL  1 ( MAR 71 )              MAIN          OS/360  FORTRAN H EXTENDED          DATE 71.062/21.06.44        PAGE   2

                          *****F O R T R A N    C R O S S    R E F E R E N C E    L I S T I N G*****
SYMBOL  INTERNAL STATEMENT NUMBERS
A       0005  0006  0006  0007
I       0004  0005  0009  0010  0012  0014  0014  0015
J       0007  0008
K       0008  0009  0010
L       0009  0010
SQRT    0006
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                          *****F O R T R A N    C R O S S    R E F E R E N C E    L I S T I N G*****
 LABEL   DEFINED   REFERENCES
    1    0011      0008  0010
    2    0014      0010
    3    0005      0015
A   4    0016      0010  0015
    5    0013      0012
    6    0019      0018
    7    0018      0015
    8    0003      0002
    9    0017      0016
  100    0002
  101    0004
  102    0006
  103    0007
  104    0008
  105    0009
  106    0010
  107    0012
  108    0015
  109    0020
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

                   1        2                3            4                         5

                000000  47 F0 F 00C          MAIN     BC     15,12(0,15)
                000004  07                            DC     XL1'07'
                000005  D4C1C9D5404040                DC     CL7'MAIN   '
                00000C  90 EC D 00C                   STM    14,12,12(13)
                000010  98 23 F 020                   LM     2,3,32(15)
                000014  50 30 D 008                   ST     3,8(13)
                000018  50 D0 3 004                   ST     13,4(0,3)
                00001C  07 F2                         BCR    15,2
TEMPORARY FOR FIX/FLOAT
                000100  00000000                      DC     XL4'00000000'
                000104  00000000                      DC     XL4'00000000'
                000108  4E000000                      DC     XL4'4E000000'
                00010C  00000000                      DC     XL4'00000000'
CONSTANTS
                000110  4F080000                      DC     XL4'4F080000'
                000114  00000000                      DC     XL4'00000000'
                000118  4E000000                      DC     XL4'4E000000'
                00011C  80000000                      DC     XL4'80000000'
                000120  00000002                      DC     XL4'00000002'
                000124  00000003                      DC     XL4'00000003'
                000128  00000005                      DC     XL4'00000005'
                00012C  000003E8                      DC     XL4'000003E8'
ADCONS FOR VARIABLES AND CONSTANTS
ADCONS FOR EXTERNAL REFERENCES
                000148  00000000                      DC     XL4'00000000'                    SQRT
                00014C  00000000                      DC     XL4'00000000'                  IBCOM#
                000178  58 F0 D 09C           100     L      15, 156( 0,13)                 IBCOM#
B               00017C  45 E0 F 004                   BAL    14,   4( 0,15)
                000180  00000006                      DC     XL4'00000006'                       6
                000184  00000028                      DC     XL4'00000028'
                000188  45 E0 F 010                   BAL    14,  16( 0,15)
                00018C  58 00 D 078           101     L       0, 120( 0,13)                      5
                000190  50 00 D 084                   ST      0, 132( 0,13)                      I
                000194  58 00 D 084             3     L       0, 132( 0,13)                      I
                000198  50 00 D 05C                   ST      0,  92( 0,13)
                00019C  97 80 D 05C                   XI      92(13),128
                0001A0  68 00 D 058                   LD      0,  88( 0,13)
                0001A4  6B 00 D 068                   SD      0, 104( 0,13)       4E00000080000000
                0001A8  70 00 D 080                   STE     0, 128( 0,13)                      A
                0001AC  41 10 D 04C           102     LA      1,  76( 0,13)
                0001B0  58 F0 D 098                   L      15, 152( 0,13)                   SQRT
                0001B4  05 EF                         BALR   14,15
                0001B6  70 00 D 0A0                   STE     0, 160( 0,13)                   .S00
                0001BA  78 00 D 0A0                   LE      0, 160( 0,13)                   .S00
                0001BE  70 00 D 080                   STE     0, 128( 0,13)                      A
                0001C2  2B 00                 103     SDR     0, 0
                0001C4  78 00 D 080                   LE      0, 128( 0,13)                      A
                0001C8  6A 00 D 060                   AD      0,  96( 0,13)       4F08000000000000
                0001CC  60 00 D 050                   STD     0,  80( 0,13)
                0001D0  58 00 D 054                   L       0,  84( 0,13)
                0001D4  50 00 D 088                   ST      0, 136( 0,13)                      J
                0001D8  58 00 D 074           104     L       0, 116( 0,13)                      3
                0001DC  50 00 D 08C                   ST      0, 140( 0,13)                      K
                0001E0  58 00 D 084           105     L       0, 132( 0,13)                      I
                0001E4  8E 00 0 020                   SRDA    0,  32
                0001E8  5D 00 D 08C                   D       0, 140( 0,13)                      K
                0001EC  50 10 D 090                   ST      1, 144( 0,13)                      L
                0001F0  58 10 D 08C           106     L       1, 140( 0,13)                      K
                0001F4  5C 00 D 090                   M       0, 144( 0,13)                      L
```

Figure II-5. Compiler Output from Programmer-Specified Options (Part 1 of 3)

```
                0001F8  5B 10 D 084                 S      1, 132( 0,13)                  I
                0001FC  58 50 D 0BC                 L      5, 188( 0,13)                  2
                000200  07 95                       BCR    9, 5
                000202  58 50 D 0C0                 L      5, 192( 0,13)                  4
                000206  07 25                       BCR    2, 5
                000208  58 00 D 08C           1     L      0, 140( 0,13)                  K
                00020C  5A 00 D 070                 A      0, 112( 0,13)                  2
                000210  50 00 D 08C                 ST     0, 140( 0,13)                  K
                000214  59 00 D 088                 C      0, 136( 0,13)                  J
                000218  58 50 D 0B0                 L      5, 176( 0,13)                 105
                00021C  07 D5                       BCR    13, 5
                00021E  58 F0 D 09C           107   L      15, 156( 0,13)               IBCOM#
                000222  18 00                       LR     0, 0
                000224  45 E0 F 004                 BAL    14,   4( 0,15)
                000228  00000003                    DC     XL4'00000003'                  3
                00022C  00000072                    DC     XL4'00000072'
                000230  45 E0 F 008                 BAL    14,   8( 0,15)
                000234  0450D084                    DC     XL4'0450D084'                  I
                000238  45 E0 F 010                 BAL    14,  16( 0,15)
                00023C  58 00 D 084           2     L      0, 132( 0,13)                  I
                000240  5A 00 D 070                 A      0, 112( 0,13)                  2
                000244  50 00 D 084                 ST     0, 132( 0,13)                  I
                000248  58 00 D 07C           108   L      0, 124( 0,13)                1000
                00024C  5B 00 D 084                 S      0, 132( 0,13)                  I
                000250  58 50 D 0C4                 L      5, 196( 0,13)                  7
                000254  07 45                       BCR    4, 5
                000256  58 50 D 0C0                 L      5, 192( 0,13)                  4
                00025A  07 95                       BCR    9, 5
                00025C  58 50 D 0AC                 L      5, 172( 0,13)                  3
                000260  07 25                       BCR    2, 5
                000262  58 F0 D 09C             4   L      15, 156( 0,13)              IBCOM#
                000266  18 00                       LR     0, 0
                000268  45 E0 F 004                 BAL    14,   4( 0,15)
                00026C  00000006                    DC     XL4'00000006'                  6
                000270  00000076                    DC     XL4'00000076'
                000274  45 E0 F 010                 BAL    14,  16( 0,15)
                000278  58 F0 D 09C             7   L      15, 156( 0,13)              IBCOM#
                00027C  45 E0 F 004                 BAL    14,   4( 0,15)
(B)             000280  00000006                    DC     XL4'00000006'                  6
                000284  00000088                    DC     XL4'00000088'
                000288  45 E0 F 010                 BAL    14,  16( 0,15)
                00028C  58 F0 D 09C           109   L      15, 156( 0,13)              IBCOM#
                000290  45 E0 F 034                 BAL    14,  52( 0,15)
                000294  05                          DC     XL1'05'
                000295  40                          DC     XL1'40'
                000296  40                          DC     XL1'40'
                000297  40                          DC     XL1'40'
                000298  40                          DC     XL1'40'
                000299  F0                          DC     XL1'F0'
ADDRESS OF EPILOGUE
                00029A  58 F0 D 09C                 L      15, 156( 0,13)
                00029E  45 E0 F 034                 BAL    14,  52( 0,15)              IBCOM#
                0002A2  0540                        DC     XL2'0540'
                0002A4  404040F0                    DC     XL4'404040F0'
ADDRESS OF PROLOGUE
                0002AA  58 F0 3 09C                 L      15, 156( 0, 3)
                0002AE  45 E0 F 040                 BAL    14,  64( 0,15)              IBCOM#
                0002B2  18 D3                       LR     13, 3
                0002B4  58 F0 D 0A8                 L      15, 168( 0,13)
                0002B8  07 FF                       BCR    15,15
ADCON FOR PROLOGUE
                0002C0  000002AA                    DC     XL4'000002AA'
ADCON FOR SAVE AREA
                000024  000000B0                    DC     XL4'000000B0'
ADCON FOR EPILOGUE
                0000B0  0000029A                    DC     XL4'0000029A'
ADCONS FOR PARAMETER LISTS
                0000FC  80000130                    DC     XL4'80000130'
TEMPORARIES AND GENERATED CONSTANTS
                000150  00000000                    DC     XL4'00000000'
                000154  00000000                    DC     XL4'00000000'
ADCONS FOR B BLOCK LABELS
                000158  00000178                    DC     XL4'00000178'
                00015C  00000194                    DC     XL4'00000194'
                000160  000001E0                    DC     XL4'000001E0'
                000164  00000208                    DC     XL4'00000208'
                000168  0000021E                    DC     XL4'0000021E'
                00016C  0000023C                    DC     XL4'0000023C'
                000170  00000262                    DC     XL4'00000262'
                000174  00000278                    DC     XL4'00000278'
```

Figure II-5. Compiler Output from Programmer-Specified Options (Part 2 of 3)

```
                                        / STRUCTURED SOURCE LISTING /
                          C     PRIME NUMBER PROBLEM
(000    ISN 0002        100 WRITE (6,8)
        ISN 0003          8 FORMAT (52H FOLLOWING IS A LIST OF PRIME NUMBERS FROM 1 TO 1000/
                           119X,1H1/19X,1H2/19X,1H3)
        ISN 0004        101 I=5
        ISN 0005          3 A=I
(002    ISN 0006        102 A=SQRT(A)
        ISN 0007        103 J=A
        ISN 0008        104 DO 1 K=3,J,2
        ISN 0009        105 L=I/K
(001    ISN 0010        106 IF(L*K-I)1,2,4
        ISN 0011          1 CONTINUE
        ISN 0012        107     WRITE (3,5)I
        ISN 0013          5     FORMAT (I20)
 001)                             C
        ISN 0014          2     I=I+2
        ISN 0015        108     IF(1000-I)7,4,3
        ISN 0016          4     WRITE (6,9)
        ISN 0017          9     FORMAT (14H PROGRAM ERROR)
 002)                             C
        ISN 0018          7     WRITE (6,6)
        ISN 0019          6     FORMAT (31H THIS IS THE END OF THE PROGRAM)
        ISN 0020        109 STOP
        ISN 0021            END

                                                  /     MAIN /     SIZE OF PROGRAM 0002BA HEXADECIMAL BYTES

      NAME     TAG    TYPE  ADD.        NAME     TAG    TYPE  ADD.        NAME     TAG    TYPE  ADD.        NAME     TAG    TYPE  ADD.
         A SFA        R*4   000130           I SF         I*4   000134           J SF         I*4   000138           K SF         I*4   000130
         L S          I*4   000140      SQRT  F  XF      R*4   000000    IBCOM#  F  XF              000000

SOURCE STATEMENT LABELS

  LABEL  ISN   ADDR          LABEL  ISN   ADDR          LABEL  ISN   ADDR          LABEL  ISN   ADDR
    100    2  000178 NR        101    4  00018C NR        3    5  000194           102    6  0001AC NR
    103    7  0001C2 NR        104    8  0001D8 NR      105    9  0001E0           106   10  0001F0 NR
      1   11  000208           107   12  00021E           2   14  00023C           108   15  000248 NR
      4   16  000262             7   18  000278         109   20  00028C NR

COMPILER GENERATED LABELS

  LABEL  ISN   ADDR          LABEL  ISN   ADDR          LABEL  ISN   ADDR          LABEL  ISN   ADDR
 100000    1  000178

FORMAT STATEMENT LABELS

  LABEL  ISN   ADDR          LABEL  ISN   ADDR          LABEL  ISN   ADDR          LABEL  ISN   ADDR
      8    3  000028             5   13  000072         9   17  000076             6   19  000088
```

Figure II-5. Compiler Output from Programmer-Specified Options (Part 3 of 3)

Object Module Listing

The programmer requests an object module listing by specifying the option LIST. The object module listing in Figure II-5 is marked B.

An object module listing is in an assembler language format showing each assembler language instruction. The listing is arranged as follows:

- The first column, labeled 1, indicates the relative address of the instruction in hexadecimal format.
- The columns labeled 2 indicate the storage representation of the instruction in hexadecimal format.
- The column labeled 3 indicates statement numbers, which may be either those appearing in the source program or those generated by the compiler (compiler-generated numbers contain six digits).

- The columns labeled 4 indicate the pseudo-assembler language format for each statement.
- The column labeled 5 indicates any significant items referred to by the instruction, such as entry points of subprograms or other statement numbers.

Edited Source Module Listing

The programmer requests an edited source module listing by specifying the options FORMAT and OPTIMIZE(2) and by including a DD statement named SYSUT1. The edited source module listing in Figure II-5 is marked C.

This listing is independent of the usual source listing; it indicates the loop structure and logical continuity of the source program.

Each loop is assigned a unique 3-digit number. Entrance to the loop is indicated by a left parenthesis followed by a 3-digit number; exit from the loop is indicated by the 3-digit number followed by a right parenthesis on a separate line before the next non-comment line.

Indentions are used to show dominance relationships among executable source statements. Statement A dominates statement B if A is the last statement common to all logical paths from which B receives control. Statement A is called a dominator, statement B is called a dominee. By this definition, a statement can have only one dominator, but a dominator may have several dominees. For example, a computed GO TO statement is the last statement through which control passes before reaching three other statements. The GO TO statement is a dominator with three dominees.

A dominee is indented from its dominator unless it is either the only dominee or the last dominee of that dominator. The line of sight between a dominator and its dominee(s) may be obscured by intervening statements. This is a dominance discontinuity and is indicated by C--- on a separate line above the dominee.

Comments and non-executable statements are not involved in dominance relationships; their presence never causes a dominance discontinuity. Comments are aligned with the last preceding non-comment line; nonexecutable statements are aligned either with the last preceding executable statement or with the first one following.

The edited source module listing in Figure II-5 shows two loops; loop 001 is nested within loop 002.

Source Module Map

The programmer requests a storage map by specifying the option MAP. The source module map in Figure II-5 is marked D.

A map indicates the use made of each number within a program.

The first line of a map gives the name and size of the program. The size is noted in hexadecimal format.

The column headed NAME indicates the name of each variable, statement function, subprogram, or implicit function.

The column headed TAG indicates use codes for each name and variable. Use codes are the following:

1. The letter S for a variable appearing to the left of an equal sign; i.e., a variable whose value was stored during some operation
2. The letter F for a variable appearing to the right of an equal sign; i.e., a variable whose value was manipulated during some operation
3. The letter A for a variable used as an argument in a parameter list
4. The letter C for a variable in a COMMON block
5. The letter E for a variable in an EQUIVALENCE block
6. The letters ASF for an arithmetic statement function
7. The letters XF for an external function
8. The letters XR for an external reference to an array, variable, or subprogram that is referenced by name.
9. The letter D for a promoted (doubled) variable
10. The letter P for a padded variable
11. An asterisk (*) for a promoted library function

Note that the combination code ASF should not be confused with the individual codes A, S, and F. The individual codes print

out in the order SFA while the arithmetic function code always prints as ASF.

The column headed TYPE indicates the type and length of each variable.

The column headed ADD indicates the relative address assigned to a name. (Functions, arithmetic statement function, subroutines, and external references have a relative address of 00000.) For non-referenced variables, this column contains the letters NR instead of a relative address.

The MAP option also produces a table of statement numbers known as a label map (marked E in Figure II-5). The label map shows each statement number from the source statement, from compiler generated labels, and from FORMAT statements, the relative address assigned to each statement, and the symbol NR for each statement that is not referenced.

If the source module contains COMMON or EQUIVALENCE statements, the MAP option also produces a storage map for each COMMON and EQUIVALENCE block. Because the sample program in Figure II-1 contained no COMMON or EQUIVALENCE statement, no such storage map is illustrated in Figure II-5. Figure II-7 illustrates a section of another program and the resulting storage map. The map for a COMMON block contains the same kind of information as for the main program. Any variable equivalenced to a variable in COMMON is listed along with its displacement (offset) from the beginning of the block.

Object Module Deck

The programmer requests an object module deck by specifying the compiler option DECK. Figure II-6 shows a typical layout of an object deck.

The object deck may be used as input to the linkage editor in a later job. The deck is a copy of the object module which consists of dictionaries, text, and an end-of-module indicator. Object modules are described in greater detail in the appropriate linkage editor and loader publication, as listed in the Preface.

The object deck consists of four types of cards identified by the characters ESD, RLD, TXT, or END in columns 2 through 4. Column 1 of each card contains a 12-2-9 punch. Columns 73 through 80 contain the first four characters of the program name followed by a four-digit sequence number. The remainder of the card contains program information.

ESD CARD: ESD cards describe the entries of the External Symbol Dictionary, which contains one entry for each external symbol defined or referred to within a module. For example, if program MAIN calls subprogram SUBA, the symbol SUBA will appear as an entry in the External Symbol Dictionaries of both the program MAIN and the subprogram SUBA. The linkage editor matches the entries in the dictionary to the entries in the dictionaries of other included subprograms and, when necessary, to the automatic call library.

ESD cards are divided into four types, identified by the digits 0, 1, 2, or 5 in column 25 of the first entry in the card, column 41 if a second entry, and column 57 if a third entry (there can be 1, 2, or 3 external-symbol entries in a card).

ESD Type	Contents
0	Name of the program or subprogram and indicates the beginning of the module.
1	Entry point name appearing in an ENTRY statement of a subprogram.
2	Name of a subprogram referred to by the source module through CALL statements, EXTERNAL statements, and explicit and implicit function references (Some usages of FORTRAN are of such complexity, that a function subprogram is called in place of in-line coding. Such calls are called implicit function references).
5	Information about a COMMON block.

TXT CARD: TXT cards contain the constants and variables used by the programmer in his source module, any constants and variables generated by the compiler, coded information for FORMAT statements, and the machine instructions generated by the compiler from the source module.

RLD CARD: RLD cards describe entries in the Relocation Dictionary, which contains one entry for each address that must be resolved before a module can be executed. The Relocation Dictionary contains information that enables absolute storage addresses to be established when a module is loaded into main storage for execution. These addresses cannot be determined earlier because the starting address of a module is not known until the module is

loaded. The linkage editor consolidates RLD entries in the input modules into a single relocation dictionary when it creates a load module.

RLD cards contain the storage address of subprograms called by ESD type 2 cards.

END CARD: The END card indicates:

1. The end of the object module to the linkage editor,
2. The relative location of the main entry point, and
3. The length (in bytes) of the object module.

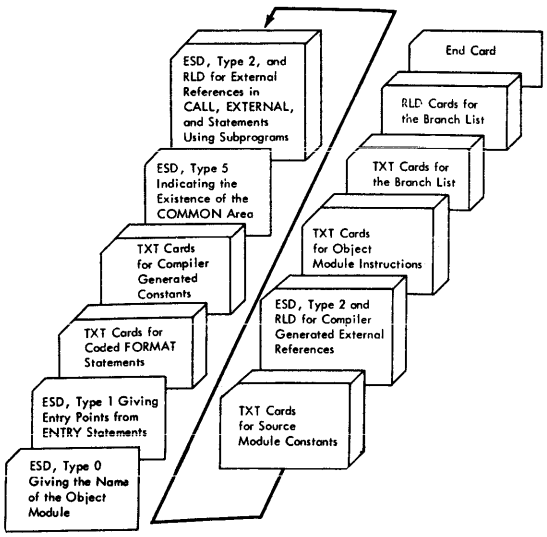


Figure II-6. Object Module Deck Structure

```
ISN 0002          SUBROUTINE POSVEC ( P, T)
                  C
                  C*************    THIS SUBROUTINE COMPUTES AND PRINTS       ******
                  C
                  C************* EIGENVALUES, CUMULATIVE PROPORTIONS OF TOTAL ******
                  C
                  C************* VARIANCES AND POSITIVE EIGENVECTORS.         ******
                  C
ISN 0003          INTEGER  P, T
ISN 0004          COMMON / SHARE1 / EIGVEC(80,80)
ISN 0005          COMMON / SHARE4 / CUMPRO(80)
ISN 0006          COMMON / VALUE / EIGVAL(80)
ISN 0007          DIMENSION POSVEX(80,80), POSVAL(80)
ISN 0008          EQUIVALENCE (EIGVEC(1,1), POSVEX(1,1))
ISN 0009          EQUIVALENCE (EIGVAL(1), POSVAL(1))

ISN 0010          CALL EIGEN ( P )
                             .
                             .
                             .
```

```
                                                  /  POSVEC /    SIZE OF PROGRAM 000478 HEXADECIMAL BYTES

  NAME    TAG    TYPE  ADD.       NAME     TAG    TYPE ADD.       NAME     TAG    TYPE  ADD.       NAME     TAG    TYPE  ADD.
     I SF        I*4   000180          J SF         I*4  0001B4         P SFA       I*4   0001B8          T SF        I*4   0001BC
 EIGEN SF XF     R*4   000000    CUMPRO SF     C    R*4  000000    EIGVAL  F  CE   R*4   000000    EIGVEC      CE   R*4   000000
IBCOM#  F XF     R*4   000000    IBERH#       XF    I*4    N.R.    POSVAL  F  CE   R*4   000000    POSVEC           R*4   000100
POSVEX  F CE     R*4   000000

                                      ***** COMMON INFORMATION   *****

   NAME OF COMMON BLOCK  *SHARE1*   SIZE OF BLOCK    006400 HEXADECIMAL BYTES

 VAR. NAME  TYPE  REL. ADDR.   VAR. NAME  TYPE  REL. ADDR.   VAR. NAME  TYPE  REL. ADDR.   VAR. NAME  TYPE  REL.ADDR.
   EIGVEC   R*4     000000

 EQUIVALENCED VARIABLES WITHIN THIS COMMON BLOCK
   VARIABLE  OFFSET             VARIABLE  OFFSET             VARIABLE  OFFSET             VARIABLE  OFFSET
   POSVEX    000000

   NAME OF COMMON BLOCK  *SHARE4*   SIZE OF BLOCK    000140 HEXADECIMAL BYTES

 VAR. NAME  TYPE  REL. ADDR.   VAR. NAME  TYPE  REL. ADDR.   VAR. NAME TYPE  REL. ADDR.   VAR. NAME  TYPE  REL. ADDR.
   CUMPRO   R*4     000000

   NAME OF COMMON BLOCK  * VALUE*   SIZE OF BLOCK    000140 HEXADECIMAL BYTES

 VAR. NAME  TYPE  REL. ADDR.   VAR. NAME  TYPE  REL. ADDR.   VAR. NAME TYPE  REL. ADDR.   VAR. NAME  TYPE  REL. ADDR.
   EIGVAL   R*4     000000

 EQUIVALENCED VARIABLES WITHIN THIS COMMON BLOCK
   VARIABLE  OFFSET             VARIABLE  OFFSET             VARIABLE  OFFSET             VARIABLE  OFFSET
   POSVAL    000000
```

Figure II-7. Source Statements and Storage Map for COMMON/EQUIVALENCE Blocks

LINKAGE EDITOR AND LOADER OUTPUT

As with compiler output, output from the link edit step depends upon the options in effect at the time of job submission.

LINKAGE EDITOR OUTPUT

The cataloged procedures calling the linkage editor specify the LET, LIST, and XREF options; the programmer may override or add to these options through the PARM parameter of the EXEC statement.

LINKAGE EDITOR OUTPUT WITH PROCEDURE-SPECIFIED OPTIONS

For the FORTRAN program illustrated in Figure II-2, linkage editor output is shown in Figure II-8. Options current at the time of job submission are always listed (Figure II-8, A) followed by any printed output.

The linkage editor combines a number of modules into one load module. The LET option marks the load module executable even though certain error conditions may have been detected. LET does not result in any printed output.

LIST causes linkage editor control statements associated with the job step to be printed. No linkage editor control statements are shown in Figure II-8. See the section "Linkage Editor Overlay Feature" for an example.

Cross-Reference Table

Unlike the compiler MAP and XREF options, which are mutually independent, the linkage editor XREF option produces both a module map and a cross-reference listing. If the programmer wants the only module map printed, he specifies MAP in place of XREF. The module map shows the name of each program unit, the relative location of its beginning point, its length, and any entry names to the program unit. The module map in Figure II-8 is labeled B.

A program unit (main program, subprogram, or COMMON block) is termed a control section in the link edit step. A control section may be the object module produced from the original source module, or it may be a module called by the linkage editor to perform such functions as input/output operations, interface with the operating system, or mathematical operations required by the program. Control sections called by the linkage editor are identified on the module map by the character * after the control section name (for example, IHOECOMH* in Figure II-8).

In Figure II-8, the control section MAIN begins at relative location 00 and has a length expressed in hexadecimal format of 278 (equal to 632 bytes). The remaining control sections are those called from the FORTRAN library by the linkage editor. Control section IHOECOMH begins at location 278 and has entry points named IBCOM, FDIOCS, and INTSWTCH. IBCOM is also the beginning point of the control section since it too begins at location 278; FDIOCS begins at location 334, and INSWTCH at location 1196. Control section IHOCOMH2 contains only one entry point, SEQDASD, beginning at a location within the section.

Following the map in Figure II-8 is the cross-reference table (labeled C) which lists the location of each reference within the load module, the symbol to which the reference refers, and the name of the control section in which the symbol is defined.

If an overlay structure has been specified, a separate map and cross-reference listing is produced for each segment of the structure. See the section "Linkage Editor Overlay Feature" for an example.

The total length of the load module is listed at the bottom of the cross-reference table.

The message (Figure II-8, D) is not part of the module map; it is a disposition message issued by the linkage editor stating that MAIN has been added to a load module library.

```
A  F88-LEVEL LINKAGE EDITOR OPTIONS SPECIFIED XREF,LET,LIST
          VARIABLE OPTIONS USED -  SIZE=(92160,8192)-          DEFAULT OPTION(S) USED

                                CROSS REFERENCE TABLE

CONTROL SECTION                                ENTRY
  NAME     ORIGIN  LENGTH                      NAME   LOCATION     NAME   LOCATION     NAME   LOCATION     NAME   LOCATION

MAIN          00     278
IHECCOMH*    278     CC4
                                               IBCOM#      2A4     FDICCS#     360     INTSWTCH    1028
IHOCOMH2*   1040     975
                                               SEQDASC    13AA
IHOSSQRT*   1988     168
                                               SQRT       1988     IH$SQRT    1988
IHOFCVTH*   1B20     AC7
                                               ADCON#     1B20     FCVACUTP   1BCA     FCVLCUTP   1C5A     FCVZOUTP   1CB6
                                               FCVIOUTP   215A     FCVEOUTP   224C     FCVCCUTP   224C     INT6SWCH   24A8
IHOEFNTH*   2528     7C8
B                                              ARITH#     2528     ADJSWTCH   2A88
IHOEFIOS*   2CF0    10FC
                                               FIOCS#     2CF0     FIOCSBEP   2CF6
IHOFIOS2*   3DE0     5AC
IHOUOPT *   4390     318
IHOFCONI*   46A8     2FD
                                               FQCONI#    46A8
IHOFCONO*   49A8     558
                                               FQCONO#    49A8
IHOERRM *   4F00     5EC
                                               ERRMON     4F00     IHOERRE    4F18
IHOUATBL*   54FC     208
IHOFTEN *   56F8     198
                                               FTEN#      56F8
IHOETRCH*   589C     2A6
                                               IHOTRCH    5890     ERRTRA     5898

LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION          LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION

     150          SQRT          IHOSSQRT                    154          IBCOM#        IHCECOMH
     360          SEQDASC       IHOCOMH2                    3B8          IHCCOMH2      IHCCOMH2
     F34          ADCON#        IHOFCVTH                    F2C          FIOCS#        IHCEFICS
     F38          ARITH#        IHOEFNTH                    F54          ADJSWTCH      IHCEFNTH
     F08          IHOUOPT       IHOUCPT                     F3C          FCVECUTP      IHOFCVTH
     F40          FCVLOUTP      IHOFCVTH                    F44          FCVIOUTP      IHOFCVTH
     F48          FCVCOUTP      IHOFCVTH                    F4C          FCVAOUTP      IHOFCVTH
     F50          FCVZOUTP      IHOFCVTH                   102C          IHOASYNC      $UNRESOLVED(W)
     E8C          IHOERRM       IHOERRM                     EE0          IHOERRE       IHCERRM
     F10          IHCCOMH2      IHOCCMH2                    EE4          IHOCOMH2      IHOCOMH2
     EE8          IHOCOMH2      IHOCCMH2                    EEC          IHOCCMH2      IHOCCMH2
     EF0          IHCCOMH2      IHOCOMH2                   117C          IHOECOMH      IHCECCMH
    1220          IHCECOMH      IHOECCMH                   1815          IHOECOMH      IHOECOMH
    1825          IHCECOMH      IHOECCMH                   1835          IHCECCMH      IHOECCMH
C   1A9C          IBCCM#        IHOECCMH                   1AC4          IHOERRM       IHCERRM
    2318          IBCOM#        IHOECCMH                   2314          IHOERRM       IHOERRM
    2368          FQCONO#       IHOFCONO                   236C          FQCONI#       IHCFCONI
    2AE4          IBCOM#        IHCECOMH                   2AE8          INTSWTCH      IHCECCMH
    2A84          INT6SWCH      IHOFCVTH                   2A80          IHOUOPT       IHCUCPT
    2AF0          ADCON#        IHOFCVTH                   2AEC          FIOCS#        IHCEFICS
    2BF0          IHOERRM       IHCERRM                    2E58          IHOASYNC      $UNRESOLVED(W)
    2E50          IHOERRM       IHCERRM                    2E54          IHOFIOS2      IHOFIOS2
    3B7C          IHOUATBL      IHOUATBL                   3B7C          IBCOM#        IHCECCMH
    3B91          IHOFIOS2      IHOFICS2                   3BA8          IHOFIOS2      IHOFICS2
    3DD9          IHOFIOS2      IHOFICS2                   493C          IHCQCONI      $UNRESOLVED(W)
    4938          FQTEN#        $UNRESOLVED(W)             4934          FTEN#         IHCFTEN
    4DA4          FTEN#         IHOFTEN                    4DAC          IHOQCONO      $UNRESOLVED(W)
    4DA8          FQTEN#        $UNRESOLVED(W)             54DC          IHCUCPT       IHCUOPT
    54E0          IBCOM#        IHCECCMH                   54E4          IHOTRCH       IHCETRCH
    54E8          FIOCSBEP      IHOEFIOS                   5A2C          LDFIC#        $UNRESOLVED(W)
    5A18          IBCOM#        IHOECOMH                   5A1C          ADCON#        IHCFCVTH
    5A24          FIOCSBEP      IHOEFICS
ENTRY ADDRESS      00
TOTAL LENGTH     5B38

D  ****MAIN      DOES NOT EXIST BUT HAS BEEN ADDED TO DATA SET
```

Figure II-8. Linkage Editor Output From Procedure-Specified Options

LOADER OUTPUT

For the FORTRAN program illustrated in Figure II-2, loader output is shown in Figure II-9. Options current at the time of job submission are always listed (Figure II-9, A) followed by any printed output.

MAP causes a load module map to be produced (Figure II-9, B). The map produced by the loader is somewhat different from that produced by the linkage editor. Like the link edit map, the loader map shows the name and the location of each program unit's beginning point. Unlike the link edit map, this map shows the absolute address rather than a relative address. The loader map also has a different format; it is designed horizontally, i.e., it is meant to be read across from line to line rather than down by column.

Each control section is mapped with three entries: its name, its type, and its beginning address. In Figure II-9, MAIN is identified as a type SD program (Section Definition, signifying a control section), and begins at absolute address 90008. IHOECOMH is also a type SD program and begins at absolute address 902C8; it contains IBCOM, FDIOCS, and INTSWTCH, all identified as type LR (Label Reference, signifying entry points within a control section). Sections called by the loader are identified by the character * after the section name.

PRINT produces a message indicating the length of the program and its absolute entry point (Figure II-9, C).

The other loader options do not produce any printed output. LET marks the load module executable even though certain error conditions may have been detected. CALL permits the loader to search system libraries to resolve external references. RES permits the loader to search the MVT link pack area queue to resolve external references. SIZE indicates the amount of storage allocated to the loader step.

```
                                        OS/36C LOADER

A ->OPTIONS USED - PRINT,MAP,LET,CALL,RES,SIZE=65536

   NAME   TYPE  ADDR      NAME   TYPE  ADDR      NAME   TYPE  ADDR      NAME   TYPE  ADDR      NAME   TYPE  ADDR
B
  MAIN      SD   82808   IHOECOMH* SD   82AC8   IBCOM#   * LR   82AF4   FDICCS# * LR   82BB0   INTSWTCH* LR   83878
  IHOCOMH2* SD   8389C   SEQDASD * LR   83BFA   IHOSSQRT* SD   842C8   IH$SQRT * LR   84208   SQRT    * LR   842C8
  IHOERRM * SD   8437C   ERRMON  * LR   84370   IHOERRE * LR   84388   IHOUOPT * SD   8496C   IHOEFNTH* SD   84C78
  ARITH#  * LR   84C78   ADJSWTCH* LR   851D8   IHOEFIOS* SD   8544C   FIOCS#  * LR   85440   FICCSBEP* LR   85446
  IHOFIOS2* SD   8653C   IHOFCVTH* SD   86AE0   ADCON#  * LR   86AE0   FCVAOUTP* LR   86B8A   FCVLOUTP* LR   86C1A
  FCVZOUTP* LR   86D76   FCVIOUTP* LR   8711A   FCVCOUTP* LR   8720C   FCVEOUTP* LR   8720C   INT6SWCH* LR   87468
  IHOFCONI* SD   874E8   FQCONI# * LR   874E8   IHOFCONO* SD   877E8   FQCONO# * LR   877E8   IHOUATBL* SD   87D4C
  IHOETRCH* SD   87F48   IHOTRCH * LR   87F48   ERRTRA  * LR   87F5C   IHOFTEN * SD   881FC   FTEN#   * LR   881FC

C  TOTAL LENGTH     5B8C
   ENTRY ADDRESS   828C8
```

Figure II-9. Loader Output

Load module output consists of messages and program output.

MESSAGES

Load module messages are generated in three forms: error code diagnostics, program interrupt messages, and operator messages. Error code diagnostics indicate input/output errors or misuse of FORTRAN library functions. Program interrupt messages indicate violations of system restrictions. Operator messages indicate interrupts caused by execution of STOP n or PAUSE statements.

ERROR CODE DIAGNOSTIC MESSAGES

An error code diagnostic generates a message followed by a traceback map written in the error message data set (usually FT06F001). The traceback map provides a guide to the programmer in determining the cause of the error and shows the path of calls made between routines in the program.

Figure II-10 shows an example of an error code message and its related traceback map, generated for the FORTRAN program illustrated in Figure II-2. The message, IHO219I, indicates that a call was made to the FIOCS routine, which processes input/output requests for a FORTRAN sequential data set, and that the operation could not be completed because of a missing DD statement. The traceback map lists the names of called routines, internal sequence numbers within routines, and contents of registers, as follows:

ROUTINE lists the names of all routines entered during processing. Names are shown with the latest routine called at the top and the earliest routine called at the bottom of the listing except when the earliest name shown is IBCOM. Then, the error could have occurred in one of the subroutines called by IBCOM. In this example, IBCOM, the FORTRAN input/output subroutine, was the last routine entered, called by MAIN, the main program. (IBCOM then called its subroutine FIOCS in which the error occurred.)

The entry CALLED FROM ISN lists the FORTRAN program's internal sequence number (ISN) that called the routine, except when calls were to IBCOM. ISNs are available to the traceback routine only if the compiler option GOSTMT was specified.

The entry REG. 14 lists the absolute storage location of the instruction calling ROUTINE.

The entry REG. 15 lists the absolute location of ROUTINE's entry point.

The entry REG. 0 lists the results of function subprogram operations, when applicable. (In this example, the contents of register 0 are meaningless.)

The entry REG. 1 lists the address of any argument list passed to ROUTINE.

The message ENTRY POINT=01087730 shows the entry point of the earliest routine entered. In the example, the number is identical to the number shown in register 15 for MAIN. The numbers do not necessarily have to agree; for example, if MAIN had several entry points, the number in the ENTRY POINT message might show a different entry point.

In Figure II-10, the control program executed its own routine to recover from the error, and displays the following message:

STANDARD FIXUP TAKEN, EXECUTION CONTINUING

If the data processing installation uses its own error recovery routine, the word USER would replace STANDARD.

After the fixup, execution continues. In the example, additional instructions calling for data set FT03F001 eventually cause execution to terminate. Message IHO900I explains that the number of errors exceeded the number permitted by the control program. A summary of errors is printed at the end of the listing to assist the programmer in determining how many times an error was encountered.

```
FOLLOWING IS A LIST OF PRIME NUMBERS FROM 1 TO 1000
                  1
                  2
                  3

IH0219I FIOCS - MISSING DD CARD OR DCB ERROR FOR ASCII TAPE FOR FT03F001

TRACEBACK  ROUTINE  CALLED FROM ISN   REG.  14   REG.  15   REG.   0   REG.   1
           IBCOM                      00045680   0004576C   00000005   FFFFFFFE
              MAIN                    0000F98C   01045480   FFFFFF2E   0005D7F8

ENTRY POINT=  01045480

STANDARD FIXUP TAKEN , EXECUTION CONTINUING

IH0219I FIOCS - MISSING DD CARD OR DCB ERROR FOR ASCII TAPE FOR FT03F001

TRACEBACK  ROUTINE  CALLED FROM ISN   REG.  14   REG.  15   REG.   0   REG.   1
           IBCOM                      00045680   0004576C   00000005   FFFFFFFF
              MAIN                    0000F98C   01045480   FFFFFF2E   0005D7F8

ENTRY POINT=  01045480

STANDARD FIXUP TAKEN , EXECUTION CONTINUING

IH0219I FIOCS - MISSING DD CARD OR DCB ERROR FOR ASCII TAPE FOR FT03F001

TRACEBACK  ROUTINE  CALLED FROM ISN   REG.  14   REG.  15   REG.   0   REG.   1
           IBCOM                      00045680   0004576C   00000005   FFFFFFFE
              MAIN                    0000F98C   01045480   FFFFFF2E   0005D7F8

ENTRY POINT=  01045480

STANDARD FIXUP TAKEN , EXECUTION CONTINUING

IH0900I EXECUTION TERMINATING DUE TO ERROR COUNT FOR ERROR NUMBER  219

IH0219I FIOCS - MISSING DD CARD OR DCB ERROR FOR ASCII TAPE FOR FT03F001

TRACEBACK  ROUTINE  CALLED FROM ISN   REG.  14   REG.  15   REG.   0   REG.   1
           IBCOM                      00045680   0004576C   00000007   FFFFFFFE
              MAIN                    0000F98C   01045480   FFFFFF2E   0005D7F8

ENTRY POINT=  01045480

SUMMARY OF ERRORS FOR THIS JOB   ERROR NUMBER   NUMBER OF ERRORS
                                     219              10
```

Figure II-10. Load Module Output with Traceback Map

Using the Traceback Map

In Figure II-10, the messages generated during traceback processing indicate that:

1. The error results from a missing DD statement for FT03F001; hence, an input/output operation is involved in the error, and, therefore, a FORTRAN input/output statement is involved.
2. The FORTRAN statement is encountered many times rather than once, since multiple occurrences of the same traceback map result.

From the foregoing description, the programmer can locate and correct either the DD statement or the FORTRAN statement containing 3 as its data set reference number. For the small program illustrated, the error may be easily located; if,

however, the source program is long and contains many input/output statements, the task of locating the error may be formidable. The traceback map further simplifies error location by pinpointing the exact FORTRAN statement involved.

If the GOSTMT compiler option is specified, the traceback map lists the internal sequence number (ISN) calling each routine. From the ISN the source module statement can be located. The example illustrated in Figure II-10 cannot take advantage of the GOSTMT option, however, because calls to IBCOM do not generate ISNs.

In addition, the programmer can take advantage of the LIST compiler option. If the option is specified, an assembler language translation of the source module is printed. The traceback map, used in the following manner, locates the last assembler language instruction executed:

1. For the topmost routine listed under the heading REG.14, subtract the 6 low-order digits in the number shown under ENTRY POINT. This produces the relative location of the instruction in the listing. In Figure II-10, location 087730 subtracted from 087960 yields 230.

2. Find the location in the pseudo-assembler listing. A portion of the pseudo-assembler listing illustrated in Figure II-5 has been reproduced in Figure II-11. Location 230 contains a BAL (Branch and Link) instruction; this is the instruction that would have been executed next if the error had not occurred.

3. Using the location as a beginning point, scan upward in the column that identifies statement numbers to locate the nearest number occurring before the instruction; this will be the statement number of the FORTRAN statement involved in the error. In Figure II-11, the statement number is 107.

4. Investigate the statement in the source module deck. Figure II-12 illustrates statement number 107 as it was coded and as it was keypunched. The statement had a keypunch error, designating 3 in place of 6 as the data set reference number.

5. If the statement had been correctly specified, the programmer would investigate the corresponding DD statement for accuracy. (In this example, this step is, of course, unnecessary since the programmer intended no DD statement named FT03F001.)

PROGRAM INTERRUPT MESSAGES

Program interrupt messages provide a guide to the programmer in determining the cause of the error; it indicates what system restriction was violated. Program interrupt messages are written in the object error message data set (usually on FT06F001).

Figure II-13 shows the format of a program interrupt message with and without the extended error handling facility. The extended error handling facility is discussed in detail in Part III in this publication.

The meaning of characters enclosed in braces is as follows:

A or Alignment
: indicates that the boundary alignment routine has been executed. Boundary alignment is performed to properly align items in storage. It is generally employed to ensure correct alignment of variables in a COMMON block or EQUIVALENCE group. Boundary alignment is performed if the option BOUNDRY=ALIGN is included at program installation time; otherwise, boundary violations cause the job to terminate. When boundary alignment is performed, message IHO210I is generated, for a maximum of ten performances. After ten performances, the message is suppressed but alignment violations continue to be corrected.

P
: indicates that the interruption was precise, i.e., the PSW (program status word) shown in message IHO210I is the one related to the interruption. In some computer models, such as the IBM System/360 Model 91, instructions may be executed in a non-sequential order such that if an interruption occurs, the printed PSW may not be the one causing the interruption. Such interruptions are called imprecise interruptions.

O or operation
: indicates that extended precision simulation has taken place. The simulator is a routine that forms part of the supervisor. Some models of the System/360 are equipped with hardware to accomplish arithmetic operations involving extended precision add, subtract, and multiply floating-point

```
0C01F8  5B 10 D 084                  S      1, 132( 0,13)                  I
0001FC  58 50 D 0BC                  L      5, 188( 0,13)                  2
000200  07 95                       BCR    9, 5
000202  58 50 D 0C0                  L      5, 192( 0,13)                  4
0C0206  07 25                        BCR    2, 5
000208  58 00 D 08C              1   L      0, 140( 0,13)                  K
00020C  5A 00 D 070                  A      0, 112( 0,13)                  2
000210  50 00 D 08C                  ST     0, 140( 0,13)                  K
000214  59 00 D 088                  C      0, 136( 0,13)                  J
000218  58 50 D 0B0                  L      5, 176( 0,13)                105
0C021C  07 D5                        BCR    13, 5
00021E  58 F0 D 09C            107   L      15, 156( 0,13)             IBCOM#
000222  18 0C                       LR     0, 0
0C0224  45 E0 F 004                  BAL    14,   4( 0,15)
000228  00000003                    DC     XL4'000C0C03'                  3
0C022C  0C000072                     DC     XL4'00000072'
000230  45 E0 F 008                  BAL    14,   8( 0,15)
0C0234  04500084                     DC     XL4'04500084'                  I
0C0238  45 E0 F 010                  BAL    14,  16( 0,15)
00023C  58 00 D 084              2   L      0, 132( 0,13)                  I
0C0240  5A 00 D 070                  A      0, 112( 0,13)                  2
0C0244  50 00 D 084                  ST     0, 132( 0,13)                  I
000248  58 00 D 07C            108   L      0, 124( 0,13)               1000
00024C  5B 00 D 084                  S      0, 132( 0,13)                  I
000250  58 50 D 0C4                  L      5, 196( 0,13)                  7
000254  07 45                       BCR    4, 5
000256  58 50 D 0C0                  L      5, 192( 0,13)                  4
00025A  07 95                        BCR    9, 5
00025C  58 50 D 0AC                  L      5, 172( 0,13)                  3
0C0260  07 25                        BCR    2, 5
000262  58 F0 D 09C              4   L      15, 156( 0,13)             IBCOM#
000266  18 00                       LR     0, 0
000268  45 E0 F 004                  BAL    14,   4( 0,15)
00026C  00000006                     DC     XL4'00000006'                  6
000270  00000076                     DC     XL4'00000076'
000274  45 E0 F 010                  BAL    14,  16( 0,15)
0C0278  58 F0 D 09C              7   L      15, 156( 0,13)             IBCOM#
00027C  45 E0 F 004                  BAL    14,   4( 0,15)
```

Figure II-11. Partial Object Code Listing

instructions. On those models which are not thus equipped, the simulator performs these operations. The simulator is always required for extended precision divide instructions.

```
Statement 107 as coded:

    107 WRITE (6,5)I


Statement 107 as keypunched:

    107 WRITE (3,5)I
```

Figure II-12. Comparison of FORTRAN Statement as Coded and as Keypunched

Exception codes themselves appear in the eighth position of the PSW and indicate the reason for the interruption. Their meanings are as follows:

Code	Meaning
1	indicates an operation exception, i.e., the operation was not one that could be defined by the operating system.
4	indicates a protection exception, i.e., an illegal reference was made to an area of storage protected by a key.
5	indicates an addressing exception, i.e., a reference was made to a storage location outside the range of storage available to the job.
6	indicates a specification exception, i.e., a unit of information does not begin on its proper boundary.

Code	Meaning
7	indicates a data exception, i.e., the arithmetic sign or the digits in a number are incorrect for the operation being performed.
9	indicates a fixed-point-divide exception, i.e., an attempt was made to divide by zero.
C	indicates an exponent-overflow exception, i.e., a floating-point arithmetic operation produced a positive number too large to be contained in a register (the largest number that may be contained is 16^{63} or approximately 7.2×10^{75}). Exponent-overflow generates the additional message: REGISTER CONTAINED number where: number is the floating-point number in hexadecimal format. (When extended-precision is in use, the message prints out the contents of two registers.) If extended error handling is specified, a standard fixup is taken and execution continues; otherwise, job termination results.
D	indicates an exponent-underflow exception, i.e., a floating-point arithmetic operation generated a negative number too large to be contained in a register (larger than 16^{-65} or approximately 5.4×10^{-79}). Exponent-underflow also generates the message: REGISTER CONTAINED number (When extended-precision is in use, the message prints out the contents of two registers.) If extended error handling is specified, a standard fixup is taken and execution continues; otherwise, job termination results.
F	indicates a floating-point-divide exception, i.e., an attempt was made to divide by zero in a floating-point operation. Floating-point divide also generates the message: REGISTER CONTAINED number (When extended-precision is in use, the message prints out the contents of two registers.) If extended error handling is specified, a standard fixup is taken and execution continues; otherwise, job termination results.

Note: Operation, protection, addressing, and data exceptions (codes 1, 4, 5, and 7) ordinarily cause abnormal termination without any corresponding message. Protection and addressing exceptions (codes 4 and 5) generate message IHO210I only if a specification exception (code 6) or an operation exception (code 1) has also been detected. A data exception (code 7) generates message IHO210I only if a specification exception has also been detected. When message IHO210I is generated for codes 4, 5, or 7, the job will terminate. The completion code in the dump indicates that job termination is due to a specification or operation exception; however, the error message indicates the true exception that caused the termination.

Requesting a Dump

Under MFT and VS1, program interrupts causing abnormal termination produce a dump called an indicative dump which displays the completion code and the contents of registers and system control fields.

To display the contents of main storage as well, the programmer must request an abnormal termination (ABEND) dump by including a SYSABEND DD statement in the appropriate job step. The following example shows how the statement may be specified for IBM-supplied cataloged procedures:

```
//GO.SYSABEND  DD  SYSOUT=A
```

Information on interpreting indicative and ABEND dumps is found in the appropriate debugging guide, as listed in the Preface.

To specify a dump under MVT and VS2, the programmer should include SYSUDUMP or SYSABEND DD statements.

OPERATOR MESSAGES

Operator messages are generated when a PAUSE or STOP n statement is executed. Operator messages are written on the system device specified for operator communication, usually the console. The message provides a guide to the programmer in determining how far his FORTRAN program has executed.

1. Without Extended Error Handling Facility

IHO210I PROGRAM INTERRUPT ({A P O}) OLD PSW IS xxxxxxx {1 4 5 6 7 9 C D F} xxxxxxxx

2. With Extended Error Handling Facility:

IHO210I IBCOM-PROGRAM INTERRUPT {ALIGNMENT OPERATION} OLD PSW IS xxxxxxx {1 4 5 6 7 9 C D F} xxxxxxxx

Figure II-13. Program Interrupt Message Format

Figure II-14 shows the form that the operator message may take. The meaning of lowercase characters in the figure is as follows:

Character	Meaning
yy	message identification number assigned by the system.
n	string of 1 through 5 decimal digits specified by the programmer in the statement. For the STOP statement, this number is placed in register 15.
'message'	literal constant specified by the programmer.
0	printed when a PAUSE statement containing no characters is executed. (Nothing is printed for a similar STOP statement.)

A PAUSE message causes program execution to halt pending operator response. To resume program execution, the operator issues the command:

```
REPLY yy,'z'
```

where yy is the message identification number and z is any letter or number.

A STOP message causes program termination.

PROGRAM OUTPUT

Program structure dictates the form that program output will take. Generally, output that is to be used in future jobs is directed to tape or direct access volumes for storage; the programmer defines such volumes on appropriate DD statements. Output that is to be visible at job end is directed to a unit record device; when using cataloged procedures, the programmer may conveniently direct such output by assigning the appropriate data set reference number in his WRITE statements, or he may define his own DD statements.

Figure II-15 shows the program output resulting from correct execution of the FORTRAN program illustrated in Figure II-1 The first four lines are generated as a result of the WRITE statement labeled 100 and the FORMAT statement labeled 8. All the remaining lines except the last result from WRITE statement 107 and FORMAT statement 5; the last line results from WRITE statement 7 and FORMAT statement 6.

Operator message caused by a PAUSE statement:

yy IHO001A PAUSE {n 'message' 0}

Operator message caused by a STOP n statement:

IHO002I STOP n

Figure II-14. Operator Message Format

```
FOLLOWING IS A LIST OF PRIME NUMBERS FROM 1 TO 1000
   1
   2
   3
   5
   7
  11
  13
  17
  19
  23
  29
  31
  37
  41
  43
  47
  53
  59
  61
  67
  71
  73
  79
  83
  89
  97
 101
 103
 107
 109
 113
 127
 131
 137
 139
 149
 151
 157
 163
 167
 173
 179
 181
 191
 193
 197
 199
 211
 223
 227
 229
 233
 239
 241
 251
 257
 263
 269
 271
 277
 281
 283
 293
 307
 311
 313
 317
 331
 337
 347
 349
 353
 359
 367
 373
 379
 383
 389
 397
 401
 409
 419
 421
 431
 433
 439
 443
 449
 457
 461
 463
 467
 479
 487
 491
 499
 503
 509
 521
 523
 541
 547
 557
 563
 569
 571
 577
 587
 593
 599
 601
 607
 613
 617
 619
 631
 641
 643
 647
 653
 659
 661
 673
 677
 683
 691
 701
 709
 719
 727
 733
 739
 743
 751
 757
 761
 769
 773
 787
 797
 809
 811
 821
 823
 827
 829
 839
 853
 857
 859
 863
 877
 881
 883
 887
 907
 911
 919
 929
 937
 941
 947
 953
 967
 971
 977
 983
 991
 997
THIS IS THE END OF THE PROGRAM
```

Figure II-15. Program Output

PART III -- PROGRAMMING TECHNIQUES

The programmer can significantly influence the efficiency of a job by the manner in which he uses FORTRAN statements and the facilities of the operating system.

This chapter discusses the following:

- FORTRAN implementation considerations
- Job control language considerations
- FORTRAN library considerations
- System considerations

FORTRAN IMPLEMENTATION

This topic describes how the programmer can use FORTRAN statements and compiler facilities to implement an efficient program. Topics are arranged in alphabetic order.

ARRAY CONSIDERATIONS

Wherever possible, arrays should be specified as one-dimensional rather than as multi-dimensional. References to higher dimensioned arrays are slower than references to lower dimensioned arrays.

ARITHMETIC IF STATEMENT

A test depending on the zero value of a real floating-point number is not recommended. Many real numbers must be represented approximately (although to a high degree of accuracy) in the internal hexadecimal code of the computer. Slight errors resulting from computation with these numbers may prevent the anticipated true zero condition from being obtained.

A fixed-point overflow condition in an arithmetic IF statement causes the middle branch (i.e., equal zero) to be taken.

ASYNCHRONOUS INPUT/OUTPUT PROGRAMMING CONSIDERATIONS

WAIT Statement: The WAIT statement need not appear in the same program unit as the corresponding READ or WRITE statement. However, the WAIT statement must identify the same list items as indicated in the list of the corresponding READ or WRITE statement. If OPTIMIZE(2) is requested, the WAIT statement list must be specified in order to ensure correct results.

REWIND, ENDFILE, and BACKSPACE Statements: These statements should not be issued for any data set on which a request is outstanding, i.e., for any data set expecting a WAIT statement.

Job Control Language, System, and Library Considerations: The DD statement DCB parameter affects asynchronous input/output processing as follows:

1. BLKSIZE specifies the block length. If it is not specified, for direct access devices the optimum block length for the particular device is assumed; for magnetic tape the block length defaults to 10K bytes.
2. BUFNO specifies the number of buffers (1, 2 or 3). If it is not specified, two buffers are assumed.
3. RECFM specifies the record format and may be specifed as either VS or VST. If RECFM is not specified, VS is assumed.

The following system restrictions affect asynchronous input/output processing:

1. If track overflow is specified (RECFM=VST), chained scheduling and backspacing are not permitted.
2. The asynchronous input/output facility does not check the characteristics of the data it moves (e.g., whether padded, promoted, REAL*16, etc.). Specifying correct data types is the programmer's responsibility.
3. A data set accessed by asynchronous input/output may not be accessed by any other type of input/output facility unless the data set is rewound before input/output facilities are changed.

4. Asynchronous input/output is not supported for unit record equipment, or for direct-access data sets.

The following library considerations affect asynchronous input/output processing:

1. Asynchronous input/output operations are assigned to a task having a higher priority than the main program, thereby permitting the operations to occur asynchronously with computation.

2. Each logical record is assigned the format RECFM=VS. The programmer may specify track overflow by coding RECFM=VST but blocked records are not permitted. If blocked records are specified, the RECFM request is ignored and RECFM=VS is assigned.

BACKSPACE STATEMENT

The BACKSPACE statement may be used to extend a data set, i.e., write additional data. In a program containing DD statements with more than one FORTRAN sequence number, the execution of an ENDFILE followed by the execution of a BACKSPACE does not cause the sequence number to increment; the latest data set remains available.

The following restrictions govern the use of the BACKSPACE statement:

1. It may not be used for any data set specifying track overflow (e.g., RECFM=FT).

2. It should not be used for any data set executing list-directed input/output statements (e.g., WRITE(10,*)A) because it would cause uncertain record placement.

3. When asynchronous input/output processing is specified, it should not be executed for any data set on which a request is outstanding, i.e., for any data set expecting a WAIT statement.

COMMON AND EQUIVALENCE STATEMENTS USED TOGETHER

Increased efficiency occurs when input/output operations are performed on data stored in contiguous storage locations. Normally, however, in input operations data may be stored in locations not necessarily contiguous; on output, data may be gathered from diverse storage locations. To make input/output operations more efficient, the following technique is suggested:

An I/O list comprising many items in a READ or WRITE statement should be defined in a COMMON statement to allocate contiguous storage space. An EQUIVALENCE statement should be defined which contains one item, equal in size to all the items in the I/O list, thus permitting the same space to be referred to by one name. Finally, an input/output statement calling the one name permits all items to be treated as one unit. An example follows:

```
COMMON/LISTA/A(4),B(4),C,D,E,F,G(8)
REAL*4 A(4),B(4),G(8),LISTB(20)
EQUIVALENCE (A(1),LISTB(1))
WRITE(6) LISTB
```

Note that the variable and the equivalenced array must be of the same type.

COMMON STATEMENT

Use of COMMON to contain variables passed among calling and called programs can result in time and storage savings. Consider the following example:

```
DIMENSION E(20),I(15)
READ (10) A,B,C
CALL SUBA (A,B,C,D,E,F,I)
 .
 .
 .
END


SUBROUTINE SUBA (P,Q,R,S,T,U,J)
DIMENSION T(20),J(15)
 .
 .
 .
RETURN
END
```

The compiler must assign storage to both main program and subprogram variables and must issue instructions required to transfer the variables from one program to another. Greater efficiency would result by specifying a COMMON block as follows:

```
COMMON A,B,C,D,E(20),F,I(15)
READ (10) A,B,C
CALL SUBA
 .
 .
 .
END

SUBROUTINE SUBA
COMMON P,Q,R,S,T(20),U,J(15)
 .
 .
 .
RETURN
END
```

Note, however, that the savings thus made may be partially or completely eliminated in a program compiled with OPTIMIZE(1) or OPTIMIZE(2) in which calls to user subprograms occur between two uses of a variable. If the variable is in COMMON, it will be stored before the call and must be loaded from storage after the call. If the variable is not in COMMON, it can be retained in a register and thus used more efficiently. This is true of relatively high-activity variables.

DATA INITIALIZATION STATEMENT -- SPECIFYING LITERALS

To initialize an array with literal data, the programmer should consider the following points:

1. He may initialize any element of an array by subscripting the array name. Only one element is initialized; if excess characters are specified, they are not placed, or spilled, into the next element. (Overflow from one element to the next is known as spill.) An array element partially filled is padded on the right with blanks. The following example illustrates how individual array elements may be initialized:

```
DIMENSION A(10)
DATA A(1),A(2),A(4),A(5)/'ABCD',
 'QRSTUVW','123',6666,/
 A(1)=ABCD
 A(2)=QRST
 A(3) not initialized (note that spill
      does not occur for a subscripted
      array name)
 A(4)=123
 A(5)=6666
 A(6) through A(10) not initialized.
```

2. He may initialize several consecutive elements of an array with a single constant by specifying the array name without a subscript. Spill occurs through as many elements as necessary to insert the constant (as long as the constant does not exceed the limits of the array). The following example illustrates how several array elements may be initialized with one constant:

```
DIMENSION ARRAY(9)
DATA ARRAY/'ABCDEFGHIJKLMNOPQRS
 TUVWXYZ'/
ARRAY(1)=ABCD
ARRAY(2)=EFGH
ARRAY(3)=IJKL
ARRAY(4)=MNOP
ARRAY(5)=QRST
ARRAY(6)=UVWX
ARRAY(7)=YZbb
ARRAY(8) and ARRAY(9) not initialized.
```

Note that spill normally begins only at the beginning of an array. To begin spill in the middle of an array, the programmer uses the EQUIVALENCE statement as in the following example:

```
DIMENSION ARRAYA(10),ARRAYB(5)
EQUIVALENCE (ARRAYA(6),ARRAYB(1))
DATA ARRAYB/'ABCDEFGHIJKLMNOPQRST'/
ARRAYA(1) through ARRAYA(5) not
     initialized
ARRAYA(6)=ABCD
ARRAYA(7)=EFGH
ARRAYA(8)=IJKL
ARRAYA(9)=MNOP
ARRAYA(10)=QRST
```

3. He may initialize individual variables after initializing an array. Each constant must be specified immediately following the variable which it is to initialize. The following example illustrates how literal data for arrays and variables may be specified together:

```
DIMENSION ARAY(5)
DATA ARAY/'ABCDEFGH'/,X/'4444'/
 ,Y/'5555'/
ARAY(1)=ABCD
ARAY(2)=EFGH
ARAY(3),ARAY(4), and ARAY(5) not
 initialized
X=4444
Y=5555
```

If each constant is not specified immediately after its associated array or variable name, spilled data may be

overlaid, as shown in the following example:

```
DIMENSION A(3)
DATA A,X/'ABCDEFGHIJKL',10.0/
A(1)=ABCD
A(2)=10.0
A(3)=IJKL
X is not initialized
```

In this example, the second element of the array is overlaid by the second initializing constant.

DIRECT-ACCESS INPUT/OUTPUT CONSIDERATIONS

Direct-access input/output provides the programmer with the ability to retrieve selected records from a data set, i.e., records can be retrieved without the necessity of reading all preceding records.

Direct-access input/output is suited to applications where large tables must be frequently searched during processing or where data sets are constantly updated.

Master and Detail Records: Records to be updated are called master records. Records containing information used to update master records are called detail records. In a direct-access data set, each master record should be constructed such that it contains a unique identification, or key, distinguishing it from all other master records. Each detail record should match the key of the appropriate master record. For example, astronomers assign numbers to identify stars; these star numbers could be used as the keys of master records, and detail records update the correct master record by matching the star number. Thus, detail records to update information for star number 383320 should contain a field specifying 383320 as the key of the master record.

A FORTRAN program indicates the record to be processed by its position in the data set. If star number 383320 is assigned to record position 383320 in the data set, the key can also be used to indicate the record position. Sometimes, however, arranging records in serial order is impractical (a data set might not contain 383320 positions). In such cases, the programmer can more conveniently arrange records in the data set by using a randomizing technique. For example, a randomizing technique to arrange star numbers might be to use the first four digits as the record position and to ignore the last two digits. Thus, star number 383320 is assigned to position 3833.

Another method might be to perform an algorithm on the number, such as squaring the number and truncating the first four and last four digits of the result. In this example, star number 383320 is assigned record position 3422. No general randomizing technique works best for all sets of identification numbers; the programmer should devise his own technique for each application.

Synonyms: Two problems arise when randomizing techniques are used: waste of space in a data set and duplication of record position numbers, called synonyms (e.g., by ignoring the last two digits, star numbers 383320, 383352, and 383396 randomize to the same number, 3833). The solution to the first problem should be developed within the randomizing technique itself. For example, if no star number begins with zero, the first thousand record positions are left blank (star number 123456 would be assigned to record position 1234). To eliminate this waste, a step might be added to this randomizing technique to subtract 1000 from the randomized numbers (star number 123456 would then be assigned to position 0234).

Chaining: The solution to the second problem is either to develop another randomizing technique that results in fewer synonyms, or to chain synonyms. Chaining is a method of arranging non-contiguous records in a chain such that each record contains a field specifying the location of the next record in the chain; the last record in the chain may contain 0 in the field to indicate the end of the chain. For example, see Figure III-1 which illustrates how star numbers 383320, 383396, and 383352, randomized to 3833, might be chained together.

Since only one record can be assigned to any one record position (if star number 383320 is assigned to record position 3833, star numbers 383396 and 383352 must be assigned elsewhere), space to accommodate synonyms must be allocated to a data set. For example, if no star number begins with zero, the programmer may choose to keep the first thousand positions of the data set available for synonyms. Thus, star numbers 383396 and 383352 are assigned somewhere in the first thousand record positions. To keep track of the exact location of any synonym, the programmer should create a record location counter (a dummy record that is initialized to the lowest record position available for synonyms, e.g., record position 1 in the example above). When the first synonym is encountered, it is inserted into record 1; address 1 is stored in the chaining field of the previous record having the same identification; and the location counter is

incremented by 1. The next synonym is allocated to the next record and the address of that record is stored in the chaining field of the previous record. The same procedure is followed for each succeeding synonym.

Creating a Direct-Access Data Set: To create a direct-access data set, the programmer initializes the volume by writing "skeleton records" using an installation-written routine. After the data set has been initialized, the programmer specifies DISP=OLD in the DD statement that defines the data set, and a FORTRAN load module can enter records into it. However, if a data set cannot be initialized prior to execution time, the programmer can initialize it at execution time by specifying DISP=NEW; the FORTRAN load module writes skeleton records into the volume as a series of blanks (hexadecimal 40).

Figure III-2 shows a block diagram describing the logic that can be used to write a direct-access data set for the first time. The block diagram does not show any attempt to write skeleton records.

For an example showing how a direct-access data set can be updated, see the section "Appendix A: Examples of Job Processing."

DEFINE FILE STATEMENT: The record description information in a DEFINE FILE statement must be consistent with space allocation specified in the SPACE parameter of the DD statement. The example below illustrates this relationship.

```
DEFINE FILE 8(1000,40,E,I)

//DD1     DD    SPACE=(40,(1000))...
```

Both the DEFINE FILE statement and the SPACE parameter describe 1000 records, each 40 bytes in length.

The DEFINE FILE statement may not be in a program unit that is overlaid; it may, however, be in a program unit not overlaid while the corresponding input/output operations are specified in an overlaid program unit. For example, the main program may specify the DEFINE FILE statement and a subprogram may perform input/output operations.

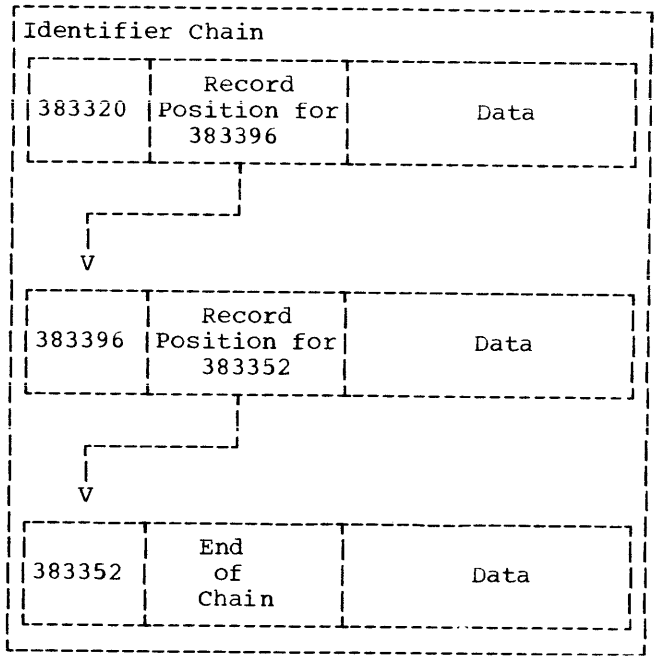


Figure III-1. Record Chaining

FIND STATEMENT: The FIND statement permits record retrieval to occur concurrently with computations of input/output operations, thus resulting in a decrease in execution time. The example below illustrates the use of the FIND statement. In this example, record 101 is retrieved while arithmetic operations are performed and is available for processing when the READ statement is reached.

```
    FIND (8'101)
 10 A=SQRT(X)
    .
    .
    .
    E=ALPHA+BETA*SIN(Y)
    WRITE (9)A,B,C,E
    READ (8'101)X,Y
```

EQUIVALENCE STATEMENT

To reduce compilation time for equivalence groups, the entries in the EQUIVALENCE statement should be specified in descending order according to displacement. Consider the following example:

```
EQUIVALENCE (VARA,ARAYA(3),ARAYB(5),
  ARAYC(10))
```

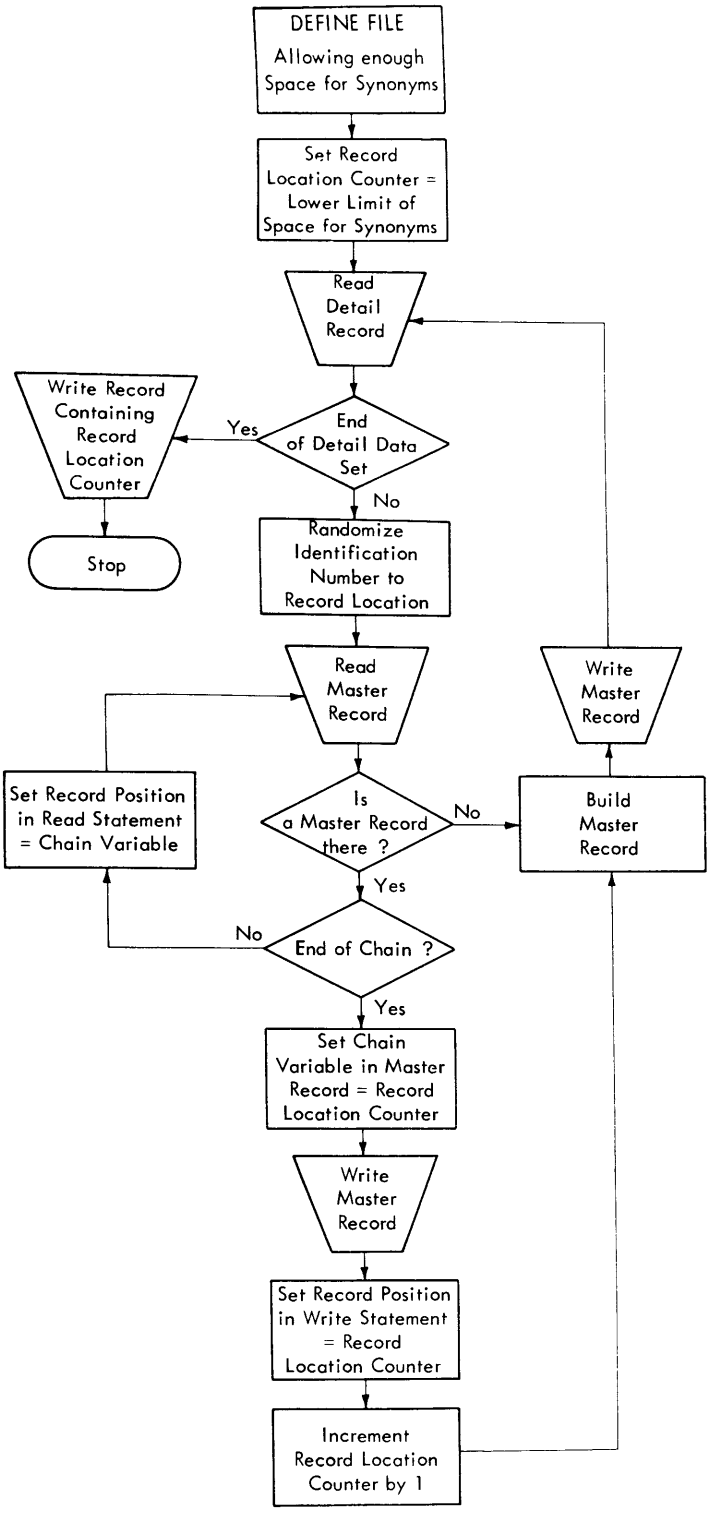


Figure III-2. Writing a Direct-Access Data Set for the First Time

This statement would be compiled faster by reversing the order, i.e.,

```
EQUIVALENCE (ARAYC(10),ARAYB(5),ARAYA(3)
  VARA)
```

To reduce compilation time and save internal table space, equivalence groups should be combined where possible. Consider the example:

```
EQUIVALENCE (ARRA(10,10),VAR1),
  (ARRB(5,5)VAR1)
```

This statement could be recoded more efficiently as:

```
EQUIVALENCE (ARRA(10,10),ARRB(5,5),VAR1)
```

EXTERNAL STATEMENT

By placing an ampersand before a function name in an EXTERNAL statement, the programmer "detaches" that name, i.e., declares it to be the name of a user-supplied function even though the name may be the same as a function or subroutine appearing in the FORTRAN library. If the function name following the ampersand is not the same as a library function, it is still considered detached; no diagnostic action is taken.

Also, by specifically typing a subprogram name, the programmer detaches the name from the library; for example, if SIN is typed as REAL*8, it is detached from the FORTRAN library.

GENERIC STATEMENT

The GENERIC statement requests the use of the Automatic Function Selection facility; i.e., the appearance of the generic name in a program causes the appropriate function name to be substituted according to the length and type of the arguments specified. For example, the generic name COS, specified with arguments of REAL*8, causes the function DCOS to be substituted.

In order to avoid conflict with specific references to functions, the function names substituted as a result of automatic function selection are aliases, not otherwise obtainable by the user. The aliases are formed by prefixing the characters IHO$$ to function names three characters in length, and IH$ to names four to six characters in length. Names six characters in length are reduced to five characters by deleting the next to last

character before prefixing the name with IH$. For example, the function DCOTAN substituted for COTAN would appear to have the name IH$DCOTN.

INPUT/OUTPUT STATEMENTS -- UNFORMATTED FORMS

The unformatted form of an input/output statement results in a faster data transfer rate into and out of storage. When operations are being performed on intermediate data sets, (those which are not intended to be printed), use of the unformatted forms increase program efficiency. In the example below, statement 11 is more efficient than statement 10:

```
      DIMENSION A(10),B(10),D(20)
      EQUIVALENCE (A(1),D(1))
   10 WRITE (20,9)A,B
    9 FORMAT (10E13.3/)
   11 WRITE (20) D
```

Note: A, B, and D must be the same type.

Unformatted input/output statements may not be used for ASCII data sets.

LIST-DIRECTED INPUT/OUTPUT

The buffer length (specified by BLKSIZE) must be large enough to contain the largest data item other than a complex item or literals in quotation marks; if it is not, an infinite loop results as the operating system attempts to get more space for the item.

LOGICAL IF STATEMENT

Use of the logical IF statement may result in more efficient compilation time. For example, statement 5 below is more efficient than statement 6:

```
    5 IF (A.GT.B) GOTO 20
    6 IF (A-B) 10,10,20
   10 CONTINUE
```

When a choice between logical operators can be made, the .OR. operator should be selected in place of the .AND. operator. For example, statement 7 below is more efficient than statement 8:

```
7 IF (A.LT.B .OR. C.EQ.D) GO TO 15
8 IF (.NOT.(A.GE.B .AND. C.NE.D)) GO TO 15
```

In statement 7, the compiler analyzes each set of comparisons separately; that is, the statement is compiled as though it were written:

```
IF (A.LT.B) GO TO 15
IF (C.EQ.D) GO TO 15
```

Thus, at execution time, if the first test is true, the remainder of the expression is not evaluated. Statement 8, however, must be evaluated in its entirety before a branch decision can be made.

The programmer can further improve the execution time of a logical IF statement by specifying as the first test the comparison that is most likely to be true. For example, in statement 7 above, if the test C.EQ.D is expected to be true more often than the test A.LT.B, the programmer should write the statement as follows:

```
IF (C.EQ.D .OR.  A.LT.B) GO TO 15
```

NAME HANDLING

The compiler places names used for variables, arrays, and subprograms into a table and searches the table whenever a reference is made to a name. The table is divided into six strings. Names that are one character long are placed into the first string; names two characters long are placed into the second string; and so on. For faster compiling, the programmer should allocate names as evenly as possible among the sizes.

OPTIMIZE COMPILER OPTION

The OPTIMIZE option permits compiler optimization techniques to improve execution time and to reduce the size of the object module.

OPTIMIZE(1) causes the entire program to be treated as a loop, with individual sections of coding, headed and terminated by labeled statements, treated as blocks. The object code is made more efficient by:

- Improving local register assignment. (Variables that are defined and used in a block are retained where possible in registers during the processing of the block. Time is saved because the number of load and store instructions are reduced.)

- Retaining the most active base addresses and variables in registers across the whole program. (Retention in registers saves time because the number of load instructions are reduced.)

- Improving branching by the use of assembler language RX format branch instructions in the object code. (An RX branch instruction saves a load instruction and reduces the number of required address constants.)

OPTIMIZE(2) performs object code optimization beyond that performed by OPTIMIZE(1) by:

- Assigning registers across a loop to the most active variables, constants, and base addresses within the loop.

- Moving outside the loop many computations which need not be within the loop.

- Recognizing and replacing redundant computations.

- Replacing where possible multiplication of induction variables by addition of those variables. (An induction variable is one that is only incremented by a constant or a variable whose value remains constant in the loop.)

- Using, where possible, the BXLE assembler instruction for loop termination. (The BXLE instruction is the fastest conditional branch; time and space are saved.)

Registers 0, 1, and 12-15 are required by the system. The remaining registers, 2-11, are available for use by optimization techniques.

PROGRAMMING CONSIDERATIONS WHEN USING OPTIMIZE(1) AND OPTIMIZE(2): Although these options can result in more efficient code, they place additional responsibilities on the programmer in coding his program with care.

Using COMMON Statements: Variables in COMMON are normally not stored on exit from a FORTRAN main program, unless an input/output statement or a subroutine call using them is issued.

Using Subprograms: If a programmer-defined subprogram is given the same name as a FORTRAN-supplied subprogram (e.g., SIN, ATAN), errors could be introduced during optimization. To avoid errors, the programmer should specify the subprogram name in an EXTERNAL statement (with an ampersand preceding the subprogram name).

If the extended error handling facility is specified and a user-supplied subroutine uses program variables, there is no guarantee that correct values will be available.

If a subprogram is called at one entry point for the purpose of initializing arguments and at another entry point for computations, the latter call must include an argument list to ensure that the subprogram will receive current values for the arguments. This rule applies when the subprogram refers to the arguments by name (i.e., accesses them in their location in the calling routine rather than through local variables).

In the following example, the updated value of I will be correctly stored and transmitted to the subprogram. If the call to the subprogram did not include the argument list, I would be updated in a register but not in storage.

```
      CALL INIT(I)
      .
      .
      .
10    CALL COMP(I)

      I=I+1
      .
      .
      .
      GO TO 10

      SUBROUTINE INIT(/J/)
      .
      .
      .
      ENTRY COMP(/J/)
```

Because each COMMON block is an independent program unit, it is independently relocatable and thus requires a base address that specifies its beginning point in storage. Each base address must be stored into a register in order to be accessible. If many COMMON blocks are defined, the need to load base addresses slows down processing time. If multiple blocks can be combined into one block less than 4096 bytes in length (the maximum number that can be accommodated in a register) one base register can serve to address each variable.

Using the Assigned GO TO Statement: If the list of statement numbers is incomplete, errors that were not present in the unoptimized code may appear. The programmer should correct such GO TO statements.

PROGRAMMING CONSIDERATIONS WHEN USING OPTIMIZE(2): OPTIMIZE(2) evaluates expressions and eliminates common expressions. For example, if an expression occurs more than once such that the program path always passes through the first occurrence to reach a later occurrence with no change in the expression's value, the first value is saved and used instead. Consider the following example:

```
A=C+D
.
.
.
F=C+D+E
```

The common expression C+D is saved from its first evaluation occurring in A and is used in F.

Computational Reordering: Computational reordering performed by OPTIMIZE(2) may produce unexpected results. For example, a test of an argument of a FORTRAN library function may be executed after the call to the function. This is caused by the movement of the function call to the back target of the loop when the function argument is not changed within the loop. Consider the following example:

```
     DO 11 I=1,10
     DO 12 J=1,10
  9  IF (B(I).LT.0) GO TO 11
 12  C(J)=SQRT(B(I))
 11  CONTINUE
```

The optimization technique moves the library function call to before statement 9, causing the square root computation to occur before the test for zero. To avoid this situation, the program could be reconstructed in the following manner:

```
     DO 11 I=1,10
  9  IF (B(I).LT.0) GO TO 11
     DO 12 J=1,10
 12  C(J)=SQRT(B(I))
 11  CONTINUE
```

READ STATEMENT

The ERR parameter in the READ statement causes a branch to another statement if an input error is encountered. The READ statement encountering the error does not bring the data into working storage; the data remains in the buffer when the branch is taken. The next READ statement brings in the data. Thus, the programmer can direct the ERR parameter to an error processing routine that reads in the error and disposes of it prior to returning to normal processing. An example is:

```
  5  READ (4,100,ERR=200) A
100  FORMAT (I10)
     .
     .
     .
200  READ (4,100) AX
     GO TO 5
```

If the ERR parameter is not provided, an input error causes the program to terminate processing, unless the extended error handling feature is in effect. For a discussion of this feature, see the chapter "Extended Error Handling Facility."

RETURN STATEMENT

The RETURN statement issues the following codes in register 15:

Code	Meaning
0	A RETURN statement was executed in either a main program or a subprogram
4*i	A RETURN i statement was executed in a subprogram
16	A terminal error was detected during execution of a library subprogram

STOP STATEMENT

In the STOP n statement, any number specified larger than 4095 causes an overflow into a system return code field; the return code that will be issued is n modulo 4096, that is, the remainder after dividing n by 4096. However, the number specified by the programmer will be displayed.

USER-SUPPLIED SUBROUTINES

A user-supplied routine having the same name as a FORTRAN-supplied subroutine or function causes the linkage editor to issue the following warning message:

```
IEW024I   EXTERNAL SYMBOL PRINTED IS DOUBLY
          DEFINED -- ESD TYPE DEFINITIONS
          CONFLICT.
```

JOB CONTROL LANGUAGE CONSIDERATIONS

The following list describes how DD statement parameters can improve efficiency

of a program (see the section "Using Job Control Language" for examples in coding DD statements):

- The SEP parameter may assign data sets whose input/output operations occur at the same time to separate channels.
- The SEP subparameter in the UNIT parameter may assign data sets to separate direct access device arms. The SEP subparameter results in device optimization; the SEP parameter, described above, results in channel optimization.
- The DISP parameter may specify the CATLG option for frequently used data sets to make use of the system's cataloging capability.
- Subparameters in the DCB parameter may be used as follows:
  a. BUFNO may be specified as BUFNO=2 to provide double buffering, resulting in an input/output overlap advantage.
  b. BLKSIZE may be specified to provide a large buffer, resulting in fewer input/output requests.
  c. OPTCD=C may be specified to provide chained scheduling, which may result in decreased transfer time for input/output operations.

USING PRE-ALLOCATED DATA SETS

Installations operating under MVT or VS2 can provide pre-allocated data sets as an aid in reducing the time required by the system to allocate data sets used on a temporary basis.

Whenever a data set is defined in a DD statement, the system must search for allocation space and must build storage tables to describe data set characteristics; the more data sets defined, the greater the time required to perform these operations.

Pre-allocated data sets are allocated once, when the system is initiated and remain available for use by all jobs submitted to the system. Use of pre-allocated data sets avoids the necessity of having the system repeat the allocation process.

If his installation provides pre-allocated data sets, the FORTRAN programmer can use them by coding the parameter DSNAME=&ddname in a DD statement, replacing ddname with the name of a pre-allocated data set. (Pre-allocated data sets are defined in the cataloged procedures calling the MVT or VS2 initiator.) The programmer codes the other DD statement parameters that he normally would to define a new data set, i.e., UNIT, SPACE, DCB. An example is the following:

```
//MYNAME   DD   DSNAME=&DED1,UNIT=SYSSQ,
//               SPACE=(80,(100,10)),
//               DCB=(RECFM=F,BLKSIZE=80)
```

The following restrictions apply when using pre-allocated data sets:

- Data sets must be on direct access devices.
- Space is provided only for the duration of the job; if the programmer wishes to keep a data set after job completion, he should not use pre-allocated data sets.
- If the system cannot assign a pre-allocated data set, the programmer-coded DD statement is used to create a temporary data set.

Detailed information on pre-allocated data sets may be found in the publication IBM System/360 Operating System: System Programmer's Guide, Order No. GC28-6550, OS/VS1 Planning and Use Guide, Order No. GC24-5090, or OS/VS2 Planning Guide, Order No. GC28-0600, as appropriate.

FORTRAN LIBRARY CONSIDERATIONS

The uses of the utility subprograms DUMP and PDUMP, the sense light subprograms, and detaching library subprogram names, are discussed in the following paragraphs.

DUMP AND PDUMP SUBPROGRAMS

The DUMP and PDUMP subprograms write the contents of storage onto the system output data set; DUMP causes the program to terminate processing and PDUMP permits the program to continue processing.

In using either subprogram, the user specifies the following:

1. The variables delimiting the storage areas to be dumped. More than one area may be specified.

2. The format of the items to be dumped, coded as follows:

Code	Format
0	Hexadecimal
1	LOGICAL*1
2	LOGICAL*4
3	INTEGER*2
4	INTEGER*4
5	REAL*4
6	REAL*8
7	COMPLEX*8
8	COMPLEX*16
9	LITERAL
10	REAL*16
11	COMPLEX*32

The following examples illustrate how a user may specify storage areas to be dumped:

1. To dump a single variable, the user specifies the variable name as both the beginning and the ending point. For example, to dump the variable B in real format and to terminate processing, the user specifies the statement:

```
CALL DUMP (B,B,5)
```

2. To dump more than one variable individually, the user specifies each variable. For example, to dump variables A, B, and C in real format and to continue program processing, the user specifies:

```
CALL PDUMP (A,A,5,B,B,5,C,C,5)
```

3. To dump all of main storage between variables, the user specifies the first and last variable to be dumped. For example, to dump main storage between variables A and C in real format and to continue program processing, the user specifies:

```
CALL PDUMP (A,C,5)
```

4. To dump an array, the user specifies the first and last elements of the array. For example, to dump the array TABLE containing 20 elements in hexadecimal format and to terminate program execution, the user specifies:

```
CALL DUMP (TABLE(1),TABLE(20),0)
```

EXTENDED-PRECISION SUBROUTINES

The extended-precision subroutines are designed for use where maximum accuracy is required. Note, however, that each extended-precision subroutine increases the execution time of a program (approximately 3 to 9 times longer than the corresponding double precision subroutine).

SENSE LIGHT SUBPROGRAMS

If the programmer intends to use the SLITE, SLITET, DVCHK, or OVERFL subprograms, he should initialize the indicators to zero at the beginning of his program since the system does not automatically initialize them.

SYSTEM CONSIDERATIONS

Compilation and load module considerations are discussed in the following paragraphs.

COMPILATION CONSIDERATIONS

Included in this topic are discussions of compiler restrictions and storage requirements.

Compiler Storage Requirements

The compiler itself requires a minimum of 160K bytes of main storage. Approximately 200 to 300 source statements can be compiled when this amount of storage is available to the compiler.

The compiler's secondary storage requirement is 160 tracks on an IBM 2311 Disk Storage Unit.

The compiler itself requires a minimum of 160K bytes of main storage for compiler code and work areas. Approximately 200 to 300 source statements can be compiled when this amount of storage is available to the compiler.

The compiler code includes two tables whose sizes are determined at installation time. The adcon table NADCON handles address constants, parameters and

temporaries. If the table is exceeded the message

ADCON TABLE EXCEEDED

is issued and compilation is terminated. The backward connector table CMAJOR is used for certain optimization features. It receives backward connector information for each block in the source program, a block being the unit of instructions associated with a single user or compiler-generated label. If CMAJOR is too small to handle all of the blocks in the source program, the message

TABLE EXCEEDED OPTIMIZATION DOWNGRADED

is issued and the compilation is affected as follows: No branching optimization will be performed with either OPTIMIZE(1) or OPTIMIZE(2). With OPTIMIZE(2), no text optimization will be performed and register assignment will treat the whole program as one loop, as in OPTIMIZE(1). The result is longer and less efficient object code.

If either of these tables overflows without having been installed at the maximum size, it may be desirable to re-install the compiler with a larger size specified for the table in question (see the publication OS FORTRAN IV (H Extended) Compiler and Library (Mod II) Installation Reference Material, Order No. SC28-6861 for details). Note that such a procedure will slightly change the amount of main storage required for compiler code.

In the partition or region in which the compiler is running, any available space in excess of that required by the compiler code is used as a work area. In a multitasking environment, to limit the amount of storage for the compiler plus the work area, thereby making more storage available for other tasks, the FORTRAN programmer can reduce the amount of storage to be allocated through use of the SIZE option.

During compilation, if the unused work area is more than 10K bytes, the compiler prints the following informational message:

nnnnK BYTES OF CORE NOT USED

This message indicates how much smaller the specified SIZE value could be. This message may also be generated when SIZE has not been specified; in such a case, the message indicates how much smaller a region or partition size could be.

Note that the SIZE option indicates only the space required by the FORTRAN option and has no effect on space required by operating system facilities. The partition or region in which the compiler is running must be at least 10K bytes larger than the specified SIZE value to accommodate facilities required by the system, such as buffer allocation routines.

If the SIZE option is specified incorrectly, a compiler diagnostic message is produced and the SIZE parameter is ignored.

Figure III-3 illustrates a sample storage structure using the SIZE option and the REGION parameter.

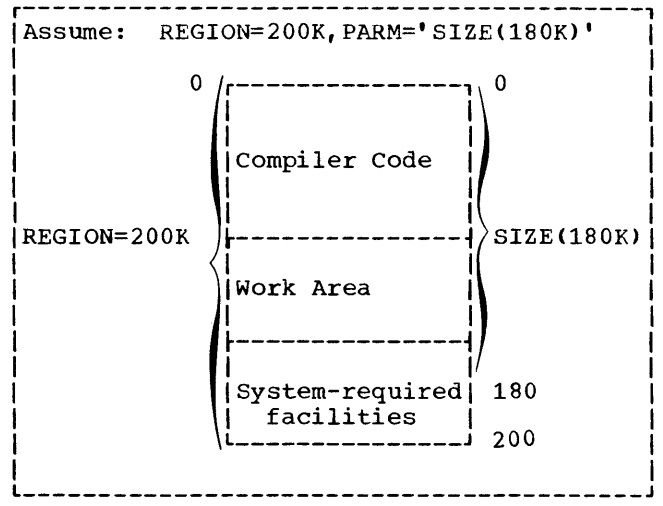


Figure III-3. Storage Structure Using SIZE and REGION

Compiler Restrictions

Compiler restrictions are the following:

1. For DO loops:
   - The maximum number of nested open DO statements is 25
   - The maximum number of implied DOs per input/output statement is 20
2. For FORMAT statements:
   - The maximum value for the repetition field (a) is 255.
   - The maximum value for the character specification field (w) is 255.
3. For statement functions:
   - The maximum number of arguments per function definition statement is 20.

- Within a function definition, the maximum number of nested references to other statement functions is 50.
- Within a function reference, the maximum number of nested references to other statement functions is 50.

4. For CALL statements:
   - The maximum number of arguments is 196; any argument containing a subscript is counted as two arguments.
5. For PAUSE statements:
   - The maximum number of characters permitted is 255.
6. For literal constants:
   - The maximum number of characters permitted is 255; this restriction applies to literal constants specified in list-directed input statements (statements with no corresponding FORMAT statement).

LOAD MODULE CONSIDERATIONS

Included in this topic are discussions of:

1. Load module restrictions
2. Boundary alignment considerations
3. Using names that the compiler recognizes as generic.

Load Module Restrictions

The following is a list of load module restrictions:

- The minimum record length for records on a magnetic tape volume is 18.
- A data set reference number cannot exceed the maximum data set reference number specified by the installation when the system is generated.

Boundary Alignment Considerations

Greater efficiency results if the programmer specifies proper boundary alignment for items in COMMON or EQUIVALENCE lists. If items are not properly aligned, further processing is dependent upon the BOUNDRY option specified at program installation time. If BOUNDARY=ALIGN was specified, boundary alignment is performed through execution of a subprogram from the SYS1.LINKLIB library; if BOUNDARY=NOALIGN was specified, the job terminates.

Using Names Recognized by the Compiler as Generic or an Alias

When automatic function selection has been requested, the H Extended compiler recognizes as generic the list of function names given in Appendix H of IBM System/360 and System/370 FORTRAN IV Language, Order No. GC28-6515-8, and subsequent revisions. Of this list, eight names are aliases; that is, they are common abbreviations for certain existing function names. In program units specifying GENERIC, they are also the generic names for the classes of function. For example, the name LOG, an alias, is recognized as generic for the family of natural logarithmic functions.

In any program unit in which GENERIC has been specified, user-supplied external procedures whose names coincide with a generic name will not be executed unless they are "detached;" that is, used in a conflicting Type statement or specified in an EXTERNAL statement and preceded by an ampersand; for example, REAL*8 LOG, or EXTERNAL &LOG. When a generic name has been so detached, it loses its generic status. Each member of that family of functions must then be referred to specifically within the program unit.

Aliases are recognized as substitute specific names even in program units not using the automatic function selection facility. Therefore, user-supplied external procedures whose names coincide with aliases for built-in functions must be detached as described above or else they will not be executed. For example, if a user-supplied function, MAX, is to be used, the name MAX must be detached. In references to the FORTRAN-supplied function within the program unit, the specific function name, MAX0, rather than the alias, MAX must be used. The aliases for built-in function names are: MAX for MAX0, MIN for MIN0, and IMAG for AIMAG.

The Automatic Precision Increase facility of the FORTRAN compiler automatically converts single precision floating point calculations to double precision and/or double precision to extended precision. It is designed to be used with programs originally written for earlier computers that offered greater precision than that available with System/360; the conversion facility may be used to convert programs where this extra precision may be of critical importance.

The facility is not meant to be used with new programs (those written for System/360 compilers). If such programs require operations with greater precision, they should be coded using the precision forms available in FORTRAN. Although the facility will convert new programs, the cost in programmer and compilation time and the increase in storage space makes its use inefficient.

No recoding of source programs is necessary to take advantage of the facility. Conversion is requested through an EXEC statement option at compilation time.

THE CONVERSION PROCESS

The conversion process comprises two functions: promotion and padding. Promotion is the process of converting items from one precision to a higher precision, for example, from single precision to double precision. The promotion function is described in greater detail below. Padding is the process of doubling the storage size of non-promoted items. Padding helps the user preserve the size relationships between promoted and non-promoted items sharing storage.

Promotion

The user may request either or both of the following conversions:

1. Single precision items to be promoted to double precision items, that is, REAL*4 to REAL*8 and COMPLEX*8 to COMPLEX*16.
2. Double precision items to be promoted to extended precision items, that is, REAL*8 to REAL*16 and COMPLEX*16 to COMPLEX*32.

Note that single precision items cannot be increased directly to extended precision items.

Promotion converts the following:

Constants: Single-precision real and complex constants are promoted to double precision. Double-precision real and complex constants are promoted to extended precision. Logical and integer constants are not affected.

Examples of promoted constants are:

Constant	Promoted Form of Constant
3.0	3.0D0
4.24E5	4.24D5
4.24D5	4.24Q5
(3.2,3.1416E0)	(3.2D0,3.1416D0)

Variables: REAL*4 and COMPLEX*8 variables are promoted to REAL*8 and COMPLEX*16, respectively. REAL*8 and COMPLEX*16 variables are promoted to REAL*16 and COMPLEX*32, respectively.

Examples of promoted variables are:

Variable	Promoted Form of Variable
REAL STAR, MOON,PLANET	REAL*8 STAR, MOON,PLANET
IMPLICIT REAL*8(S,T,U)	IMPLICIT REAL*16(S,T,U)
COMPLEX*8 A,B,C,D	COMPLEX*16 A,B,C,D

Functions: The correct FORTRAN-supplied functions are substituted when a program is converted. For example, a reference to SIN causes the DSIN function to be substituted if double precision calculation is to be performed; a reference to DINT causes QINT to be substituted if extended precision calculation is performed. Table III-1 lists FORTRAN-supplied built-in functions that are substituted. Table III-2 lists FORTRAN-supplied library functions that are substituted. Function values are promoted in the same manner as constants; that is, single precision values are promoted to double precision, double precision values are promoted to extended precision.

Previously compiled subprograms must be recompiled to be converted to the correct precision. For example, if a user-supplied subprogram accepts only single precision arguments and it is to be used with a program being converted to double precision, it must be recompiled using API to accept double precision arguments.

EXEC STATEMENT OPTIONS

The programmer requests the automatic precision increase facility through the PARM parameter in the EXEC statement calling the compiler. The PARM parameter specifies the AUTODBL subparameter to indicate the form that the conversion will take and the ALC subparameter to indicate whether storage alignment is to take place.

AUTODBL Subparameter

The AUTODBL subparameter takes one of the following forms:

AUTODBL(NONE)
: to indicate no conversion. This is the default condition.

AUTODBL(DBLPAD)
: to indicate promotion and padding of single and double precision items. REAL*4, REAL*8, COMPLEX*8 and COMPLEX*16 types are converted. Items of other types are padded if they share storage space with converted items.

AUTODBL(DBLPAD4)
: to indicate promotion of single precision items only, and padding of other items that share storage with promoted items.

AUTODBL(DBLPAD8)
: to indicate promotion of double precision items only, and padding of other items that share storage with promoted items.

The promotion and padding options, DBLPAD, DBLPAD4, and DBLPAD8, ensure that the storage-sharing relationship that existed prior to conversion is maintained. Note, however, that padding reduces the efficiency of input/output operations for padded arrays.

AUTODBL(DBL)
: to indicate promotion (but no padding) of both single and double precision items. Items of REAL*4 and COMPLEX*8 types are converted to REAL*8 and COMPLEX*16. Items of REAL*8 and COMPLEX*16 types are converted to REAL*16 and COMPLEX*32.

AUTODBL(DBL4)
: to indicate promotion of single precision items only.

AUTODBL(DBL8)
: to indicate promotion of double precision items only.

Note: If AUTODBL is specified, and an error in coding the parameter is detected, the compiler substitutes the option DBLPAD8 as a default.

For most programs, one of the above forms is sufficient. The following form offers greater flexibility to the user who wishes to tailor the conversion process to a particular program; however, it also increases the chance of error and should be used with care.

AUTODBL(abcde)
: indicates that the program is to be converted according to the value of abcde, a five-position field. Each position is coded with a numeric value that specifies how a particular conversion function is to be performed.

  The leftmost position (position a) describes the promotion function, that is, whether promotion is to occur and, if so, which items are to be promoted. The second position (position b) describes the padding function, that is, whether padding is to occur and, if so, the sections in the program (such as COMMON or argument lists) where padding is to take place. The third, fourth, and fifth positions describe whether padding is to occur for particular types (LOGICAL, INTEGER, and REAL, respectively) within the program sections specified in position b.

Table III-1. Built-In Functions -- Substitution of Single and Double Precision

Single Precision Function			Corresponding Double Precision Function			Corresponding Extended Precision Function		
Name	Argument Type	Function Value Type	Name	Argument Type	Function Value Type	Name	Argument Type	Function Value Type
AMOD	REAL*4	REAL*4	DMOD	REAL*8	REAL*8	QMOD	REAL*16	REAL*16
ABS	REAL*4	REAL*4	DABS	REAL*8	REAL*8	QABS	REAL*16	REAL*16
INT	REAL*4	INT*4	IDINT	REAL*8	INT*4	IQINT	REAL*16	INT*4
AINT	REAL*4	REAL*4	DINT	REAL*8	REAL*8	QINT	REAL*16	INT*
AMAX0[1]	INT*4	REAL*4						
AMAX1	REAL*4	REAL*4	DMAX1	REAL*8	REAL*8	QMAX1	REAL*16	REAL*16
MAX1[1]	REAL*4	INT*4						
AMIN0[1]	INT*4	REAL*4						
AMIN1	REAL*4	REAL*4	DMIN1	REAL*8	REAL*8	QMIN1	REAL*16	REAL*16
MIN1[1]	REAL*4	INT*4						
FLOAT	INT*4	REAL*4	DFLOAT	INT*4	REAL*8	QFLOAT	INT*4	REAL*16
IFIX	REAL*4	INT*4	IDINT	REAL*8	INT*4	IQINT	REAL*16	INT*4
HFIX[1]	REAL*4	INT*2						
SIGN	REAL*4	REAL*4	DSIGN	REAL*8	REAL*8	QSIGN	REAL*16	REAL*16
DIM	REAL*4	REAL*4	DDIM	REAL*8	REAL*8	QDIM	REAL*16	REAL*16
REAL	COMPLEX*8	REAL*4	DREAL	COMPLEX*16	REAL*8	QREAL	COMPLEX*32	REAL*16
AIMAG	COMPLEX*8	REAL*4	DIMAG	COMPLEX*16	REAL*8	QIMAG	COMPLEX*32	REAL*16
CMPLX	REAL*4	COMPLEX*8	DCMPLX	REAL*8	COMPLEX*16	QCMPLX	REAL*16	COMPLEX*32
CONJG	COMPLEX*8	COMPLEX*8	DCONJG	COMPLEX*16	COMPLEX*16	QCONJG	COMPLEX*32	COMPLEX*32

[1]The corresponding double precision function does not exist by name, but the single precision function is expanded as though the double precision function existed.

Table III-2. Library Functions -- Substitution of Single and Double Precision

Single Precision Function			Corresponding Double Precision Function			Corresponding Extended Precision Function		
Name	Argument Type	Function Value Type	Name	Argument Type	Function Value Type	Name	Argument Type	Function Value Type
EXP	REAL*4	REAL*4	DEXP	REAL*8	REAL*8	QEXP	REAL*16	REAL*16
CEXP	COMPLEX*8	COMPLEX*8	CDEXP	COMPLEX*16	COMPLEX*16	CQEXP	COMPLEX*32	COMPLEX*32
ALOG	REAL*4	REAL*4	DLOG	REAL*8	REAL*8	QLOG	REAL*16	REAL*16
CLOG	COMPLEX*8	COMPLEX*8	CDLOG	COMPLEX*16	COMPLEX*16	CQLOG	COMPLEX*32	COMPLEX*32
ALOG10	REAL*4	REAL*4	DLOG10	REAL*8	REAL*8	QLOG10	REAL*16	REAL*16
ARSIN	REAL*4	REAL*4	DARSIN	REAL*8	REAL*8	QARSIN	REAL*16	REAL*16
ARCOS	REAL*4	REAL*4	DARCOS	REAL*8	REAL*8	QARCOS	REAL*16	REAL*16
ATAN	REAL*4	REAL*4	DATAN	REAL*8	REAL*8	QATAN	REAL*16	REAL*16
ATAN2	REAL*4	REAL*4	DATAN2	REAL*8	REAL*8	QATAN2	REAL*16	REAL*16
SIN	REAL*4	REAL*4	DSIN	REAL*8	REAL*8	QSIN	REAL*16	REAL*16
CSIN	COMPLEX*8	COMPLEX*8	CDSIN	COMPLEX*16	COMPLEX*16	CQSIN	COMPLEX*32	COMPLEX*32
COS	REAL*4	REAL*4	DCOS	REAL*8	REAL*8	QCOS	REAL*16	REAL*16
CCOS	COMPLEX*8	COMPLEX*8	CDCOS	COMPLEX*16	COMPLEX*16	CQCOS	COMPLEX*32	COMPLEX*32
TAN	REAL*4	REAL*4	DTAN	REAL*8	REAL*8	QTAN	REAL*16	REAL*16
COTAN	REAL*4	REAL*4	DCOTAN	REAL*8	REAL*8	QCOTAN	REAL*16	REAL*16
SQRT	REAL*4	REAL*4	DSQRT	REAL*8	REAL*8	QSQRT	REAL*16	REAL*16
CSQRT	COMPLEX*8	COMPLEX*8	CDSQRT	COMPLEX*16	COMPLEX*16	CQSQRT	COMPLEX*32	COMPLEX*32
TANH	REAL*4	REAL*4	DTANH	REAL*8	REAL*8	QTANH	REAL*16	REAL*16
SINH	REAL*4	REAL*4	DSINH	REAL*8	REAL*8	QSINH	REAL*16	REAL*16
COSH	REAL*4	REAL*4	DCOSH	REAL*8	REAL*8	QCOSH	REAL*16	REAL*16
ERF	REAL*4	REAL*4	DERF	REAL*8	REAL*8	QERF	REAL*16	REAL*16
ERFC	REAL*4	REAL*4	DERFC	REAL*8	REAL*8	QERFC	REAL*16	REAL*16
GAMMA[1]	REAL*4	REAL*4	DGAMMA[1]	REAL*8	REAL*8			
ALGAMA[1]	REAL*4	REAL*4	DLGAMA[1]	REAL*8	REAL*8			
CABS	COMPLEX*8	REAL*4	CDABS	COMPLEX*16	REAL*8	CQABS	COMPLEX*32	REAL*16

[1]The extended precision equivalences of these functions do not exist. In promoting REAL*8 to REAL*16, the double precision function will be used.

All five positions must be coded; if a function is to be omitted, the corresponding position is coded with a zero. The values for each position are as follows:

- Position a, the promotion function:

Value	Meaning
0	No promotion
1	Promote REAL*4 and COMPLEX*8 items only
2	Promote REAL*8 and COMPLEX*16 items only
3	Promote all real and complex items

- Position b, the padding function:

Value	Meaning
0	No padding
1	Pad COMMON statement and argument list variables
2	Pad EQUIVALENCE statement variables equivalenced to promoted variables
3	Pad COMMON and EQUIVALENCE statement variables related to promoted variables and argument list variables
4	Pad EQUIVALENCE statement variables that do not relate to variables in COMMON statements
5	Pad variables everywhere

The code specified in this position determines in which areas of a program the padding requested by positions c to e is to take place.

- Position c, padding logical variables in program sections specified in position b:

Value	Meaning
0	Pad no logical variables
1	Pad LOGICAL*1 variables only
2	Pad LOGICAL*4 variables only
3	Pad all logical variables

- Position d, padding integer variables in program sections specified in position b:

Value	Meaning
0	Pad no integer variables
1	Pad INTEGER*2 variables only
2	Pad INTEGER*4 variables only
3	Pad all integer variables

- Position e, padding real and complex variables in program sections specified in position b:

Value	Meaning
0	Pad no real or complex variables
1	Pad REAL*4 and COMPLEX*8 variables
2	Pad REAL*8 and COMPLEX*16 variables
3	Pad REAL*4, REAL*8, COMPLEX*8, and COMPLEX*16 variables
4	Pad REAL*16 and COMPLEX*32 variables
5	Pad REAL*4, COMPLEX*8, REAL*16, and COMPLEX*32 variables
6	Pad REAL*8, REAL*16, COMPLEX*16, and COMPLEX*32 variables
7	Pad all real and complex variables

Note that promotion overrides padding. For example, if the first position specifies promotion to occur for single precision items, REAL*4 and COMPLEX*8 items are promoted regardless of the padding function specified in position e.

Coding Examples

The AUTODBL(abcde) settings that correspond to the mnemonic options are:

NONE(00000)

DBL(30000)

DBL4(10000)

DBL8(20000)

DBLPAD(33334)

DBLPAD4(13336)

DBLPAD8(23335)

The following examples illustrate other possible selections of the AUTODBL(abcde) format.

Example 1: AUTODBL(12330)

Promotion is performed and padding is performed for all EQUIVALENCE statements, logical variables, and integer variables.

Example 2: AUTODBL(01001)

No promotion is performed, but padding is performed for all REAL*4 and COMPLEX*8 variables in common blocks and argument lists. This code setting permits a program not requiring double precision accuracy to link with a subprogram compiled with the option AUTODBL(DBL).

Example 3: AUTODBL(01337)

No promotion is performed, but padding is performed for all integer, logical, real, and complex variables that are in COMMON or are used as subprogram arguments. This code setting permits a non-converted program to link with a program converted with the option AUTODBL(DBLPAD4).

ALC Subparameter

The ALC subparameter is used to specify storage alignment. It takes one of the forms:

ALC
NOALC
to indicate whether storage alignment is to take place. NOALC is the default value.

Ordinarily, to increase execution-time efficiency, COMMON statements are coded so that variables in COMMON blocks are aligned on proper boundaries: doubleword variables on doubleword boundaries, fullword variables on fullword boundaries, and halfword variables on halfword boundaries. When the conversion facility is used, these alignments may become altered. The ALC option restores alignment.

The ALC option should be used with care for it may cause previously matched COMMON blocks to become mismatched. For example, consider the two COMMON statements below where the variable INTER is to be shared:

Program 1	Program 2
REAL*8 R8	REAL*4 R4
COMMON/X/A,R8,INTER	COMMON/X/A,I,R4,INTER

With neither the AUTODBL nor the ALC option specified, both occurrences of the variable INTER will be at an offset of 12 bytes from the start of COMMON block X.

If ALC alone is used, INTER would be 16 bytes from the start of COMMON X in Program 1 since R8 would have been placed on a double word boundary. COMMON X in Program 2 would have been unaffected.

If AUTODBL(DBL) and ALC are specified, INTER would be 16 bytes from start of block X in Program 1 and 24 bytes from start in Program 2. (This is because of the promotion of REAL*4 to REAL*8 and subsequent alignment.)

It is recommended that ALC be used only when the COMMON variables are identical in type.

PROGRAMMING CONSIDERATIONS WITH API

This section provides a brief discussion of how use of the Automatic Precision Increase facility affects program processing.

Effect on COMMON or EQUIVALENCE Data Values

Promotion and padding operations preserve the storage sharing relationships that existed before conversion. However, in storage sharing items, data values are preserved only for the following:

1. Variables having the same length
2. Real and complex variables having the same precision

For example, the following items retain value sharing relationships:

LOGICAL*4 and INTEGER*4 (same length)

REAL*4 and COMPLEX*8 (same precision)

The following items do not retain value sharing relationships:

INTEGER*2 and INTEGER*4 (different lengths)

REAL*8 and COMPLEX*8 (different precision)

Effect on Literal Constants

Care should be exercised when specifying literal constants as data initialization values for promoted or padded variables, as subprogram arguments, or in NAMELIST input. For example, literals should be entered into arrays on an element by element basis rather than as one continuous string. Consider the following statements:

```
DIMENSION A(2),B(2)
DATA A/'ABCDEFGH'/,B(1)/'IJKL'/,B(2)
 /'MNOP'/
```

Array B will be initialized correctly but not array A, because padding takes place at the end of each element.

Effect on Programs Calling Subprograms

FORTRAN main programs and subprograms must be converted so that variables in COMMON retain the same relationship to guarantee correct linkage during execution. The recommended procedure is to compile all program units using AUTODBL(DBLPAD). If an option other than DBLPAD is selected, care must be taken if the COMMON variables in one program unit differ from those in another; COMMON variables that are not to be promoted should be padded.

Any non-FORTRAN external subprogram called by a converted program unit should be recoded to accept padded and promoted arguments.

Effect on FORTRAN Library Subprograms

1. If a call to a FORTRAN library subprogram contains promoted arguments, the next higher precision subprograms are substituted for the original ones. The extended symbol dictionary, used by the linkage editor to resolve references between program units, will contain the double and extended precision names for each single and double precision library program promoted.

2. If the programmer has supplied his own function for a FORTRAN-supplied function, but has neglected to detach the name through an EXTERNAL statement, the wrong function may be executed.

   Example: AUTODBL(DBL4)

```
      REAL*4 X,Y
    4 Y=SIN(X)
      STOP
      END
       .
       .
       .
      FUNCTION SIN(X)
       .
       .
       .
      RETURN
      END
```

   In this example, because the compiler cannot recognize SIN as a user-supplied function, it substitutes the name of the FORTRAN-supplied function DSIN in statement 3. However, the compiler does not change the function definition statement; the name remains SIN. At execution time the user-supplied function SIN is ignored and the FORTRAN-supplied function DSIN is executed in its place.

   The programmer can avoid this confusion either by making sure he detaches the name SIN, preceded by an ampersand, in an EXTERNAL statement or by changing the name of the function to DSIN.

Effect on CALL DUMP or CALL PDUMP Statements

If a CALL DUMP or CALL PDUMP statement requests a dump format of either REAL*4 or COMPLEX*8, output from a converted program is shown in single precision format. Each item is displayed as two single precision numbers rather than as one double precision number.

For variables that are promoted, the first number is approximately the value of the stored variable; the second number is meaningless.

For variables that are padded, the first number is exactly the value of the variable; the second number is meaningless.

Effect on Direct-Access Input/Output Processing

When a DEFINE FILE statement has been specified, any record exceeding the maximum specified record length causes record overflow to occur.

For converted programs, the programmer should check the record size coded in the statement to determine if it can handle the increased record lengths. If not sufficient, the size should be increased appropriately.

Effect on Asynchronous Input/Output Processing

Extreme care should be exercised in using the Automatic Precision Increase facility for programs containing asynchronous input/output statements.

The asynchronous input/output facility transmits the number of bytes as specified by the transmitting or receiving areas. These areas for any given data set must have the same characteristics regarding promotion and padding; e.g., both must be padded or both must be promoted.

Effect on Unformatted Input/Output Data Sets

Unformatted input/output data sets which have not been converted are not directly acceptable to converted programs if the I/O list contains promoted variables.

To make an unconverted data set accessible to the converted program, the programmer should code BFALN=F in the DCB parameter at load module execution time, causing data to be transmitted in its unconverted form. For example, assume that a program contains the statement:

```
WRITE (3)I,J,(ARRAY(N),N=1,J)
```

If the program is converted such that ARRAY contains promoted items, the programmer should code the following DD statement to write unconverted ARRAY records:

```
//FT03Fxxx   DD   DCB=(BFALN=F...)...
```

The BFALN parameter is not specified for:

- Programs and data sets having the same conversion characteristics,
- Formatted data sets, regardless of conversion characteristics; the FORMAT statement controls the correct transmission of data.

Effect on the Storage Map

The storage map produced by the MAP option contains the following codes:

Code	Meaning
D	Promoted variable
P	Padded variable
*	Promoted library function name

Overlay is a feature of linkage editor processing that allows the FORTRAN user to reduce the main storage requirements of his program by breaking it up into two or more segments that need not be in main storage at the same time. These segments can be assigned the same storage addresses and can be loaded at different times during execution of the program. The user specifies linkage editor control statements to indicate the relationship of segments within the overlay structure. FORTRAN programs run under VS seldom require overlay processing.

DESIGNING A PROGRAM FOR OVERLAY

Programs are placed in an overlay structure according to the size, frequency of use, and logical relationships between the program units that they comprise. The basic principle of overlay is illustrated by the simple example in Figure III-4. This figure shows a FORTRAN program consisting of a main program and two large subprograms named SUBA and SUBB. Normally, all three program units would be loaded into main storage at the same time and would remain there throughout execution of the entire program. However, if there is not enough main storage space available to accommodate all three program units at once, and if SUBA and SUBB do not have to be in main storage at the same time, the user could design an overlay structure in which the MAIN routine resides in main storage at all times, while subprograms SUBA and SUBB make use of the remaining space as they are needed.

Figure III-5 shows what happens at execution time to the program in Figure III-4. The MAIN routine is loaded and processing begins. When the MAIN routine calls SUBA, SUBA is loaded and processing continues until SUBB is called. SUBB then overlays SUBA in main storage and remains there until SUBA is called again. The main storage requirements of the program are thus reduced from the total number of bytes in all three program units to the total number of bytes in the MAIN program plus the larger of the two subprograms.

SEGMENTS

The relationships among the program units in the overlay program described in the preceding paragraphs can be graphically represented by an overlay "tree" structure as shown in Figure III-6. Each "branch" of the overlay tree consists of a separately loaded unit of the program to which the linkage editor assigns a number. Such overlay units, or segments, may contain one or more subprograms totaling 524,288 bytes (512K).

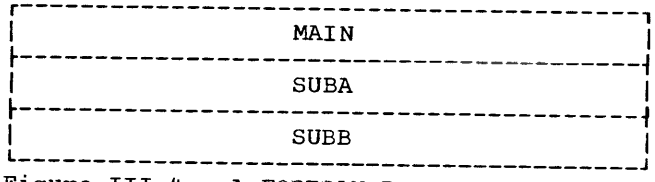


Figure III-4. A FORTRAN Program Containing Three Program Units

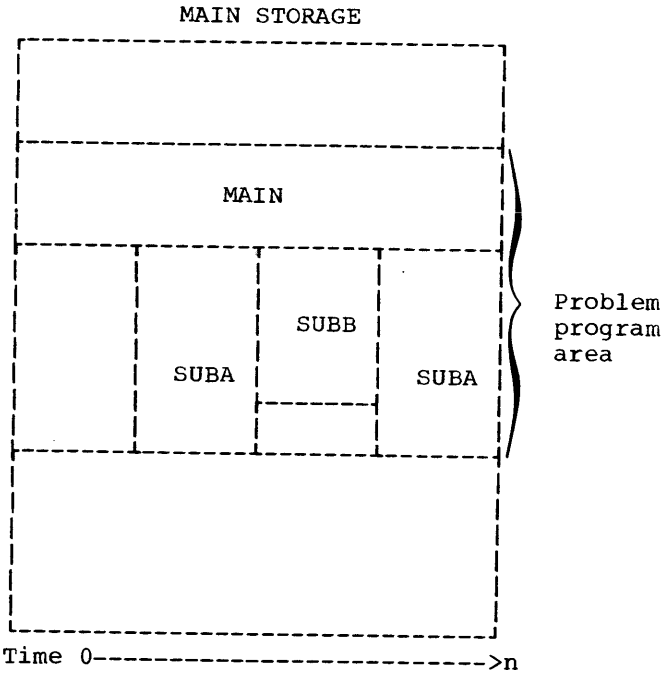


Figure III-5. Time/Storage Map of Program Described in Figure III-4

The first segment in any overlay program is called the root segment. The root segment remains in main storage at all

times during execution of the program. It must contain:

- The program unit which receives control at the start of processing. Usually this is the main routine in which processing begins at the entry point named MAIN.
- Any program units which should remain in main storage throughtout processing. For greater efficiency, subprograms that are frequently called should also be placed in the root segment if possible.
- Any program units containing DEFINE FILE statements.
- Certain information needed by the operating system to control the overlay operation. Like FORTRAN library subprograms, this information is automatically included in the root segment by the linkage editor.

PATHS

The relationships among the segments of an overlay program are expressed in terms of "paths." A path consists of a given segment and any segments between it and the root segment. The root segment is thus a part of every path, and when a given segment is in main storage, all segments in its path are also in main storage. The simple program in Figure III-6 is made up of only two paths as shown in Figure III-7.

The paths of an overlay program are determined by the dependencies between program units. A program unit is

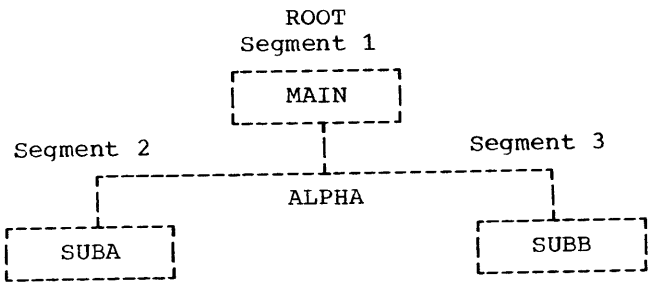


Figure III-6. Overlay Tree Structure of Program Described in Figure III-4

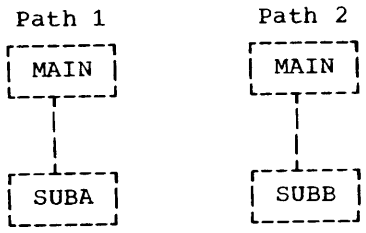


Figure III-7. Overlay Paths Implied by Tree Structure in Figure III-6

considered to be dependent on any other program unit which it calls or whose data it must process.

Figure III-8 shows a FORTRAN program in an overlay tree structure. The paths implied by that structure are illustrated in Figure III-9. The MAIN routine and subprograms SUB1 and SUB2 remain in main storage for the duration of execution time; they occupy the root segment. The segment containing subprograms SUB3 and SUB4 use the same area of main storage as the segment containing subprograms SUB11 and SUB12. Likewise, the main storage area used by the segment containing SUB5, SUB6, and SUB7 are used by the segment containing SUB8 and SUB9, as well as by the segment containing SUB10. Figure III-10 is a time/storage map of the program shown in Figures III-8 and III-9.

The structure in Figures III-8 and III-9 consists of segments numbered 1 through 6, with segment 1 being the root segment. Segments 2 and 6 have the same relative origin; that is, they will start at the same location when in main storage. This origin has been given the symbolic name ALPHA by the user (on an OVERLAY control card). The relative origin of segments 3, 4, and 5 has been given the symbolic name BETA.

The relative origin of the root segment, also called the relocatable origin, is assigned at 0. The relative origin of any segment other than the root segment is determined by adding the lengths of all segments in its path, including the root segment. When the program is loaded for execution, the first location of the root segment (the relocatable origin of the program) is assigned to an absolute storage address. All other origins are automatically increased by the number of that storage address, i.e., given an address relative to the address assigned to the root segment.

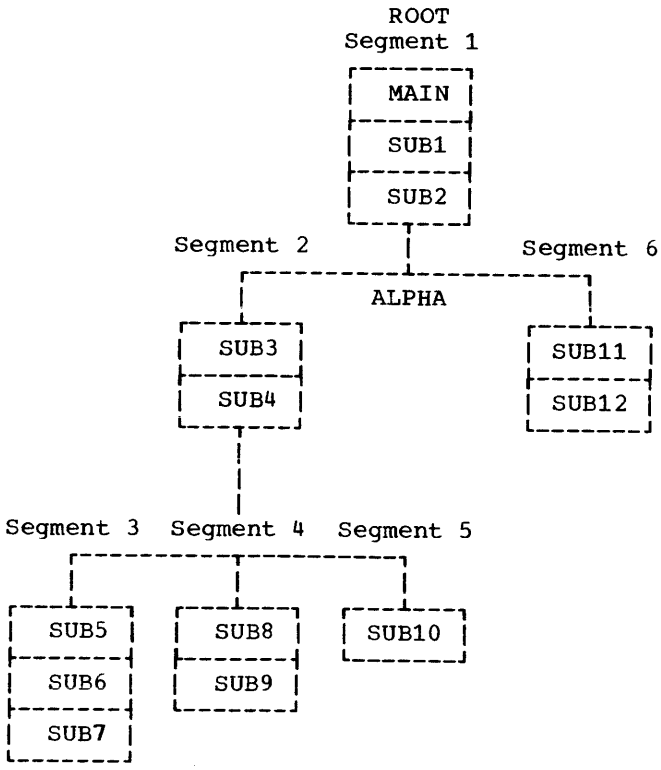


Figure III-8. Overlay Tree Structure Having Six Segments

COMMUNICATING BETWEEN SEGMENTS

Overlay segments can be related to one another either by being inclusive or exclusive. Inclusive segments are those which can be in main storage simultaneously; in other words, those which lie in the same path. Exclusive segments are those which lie in different paths. Thus, in the program shown in Figures III-8 and III-9, segments 2 and 5 are inclusive, while segments 2 and 6 are exclusive.

Inclusive References

An inclusive reference is a reference from a segment in main storage to a subprogram which will not overlay the calling segment. When a CALL is made from a program unit in one segment to a program unit in an inclusive segment, control may be returned to the calling segment by means of a RETURN statement.

When a CALL is issued to a subprogram which is higher (closer to the root segment) on the overlay tree, the called subprogram must return control to the calling segment by a RETURN statement before any exclusive overlay segments may be loaded.

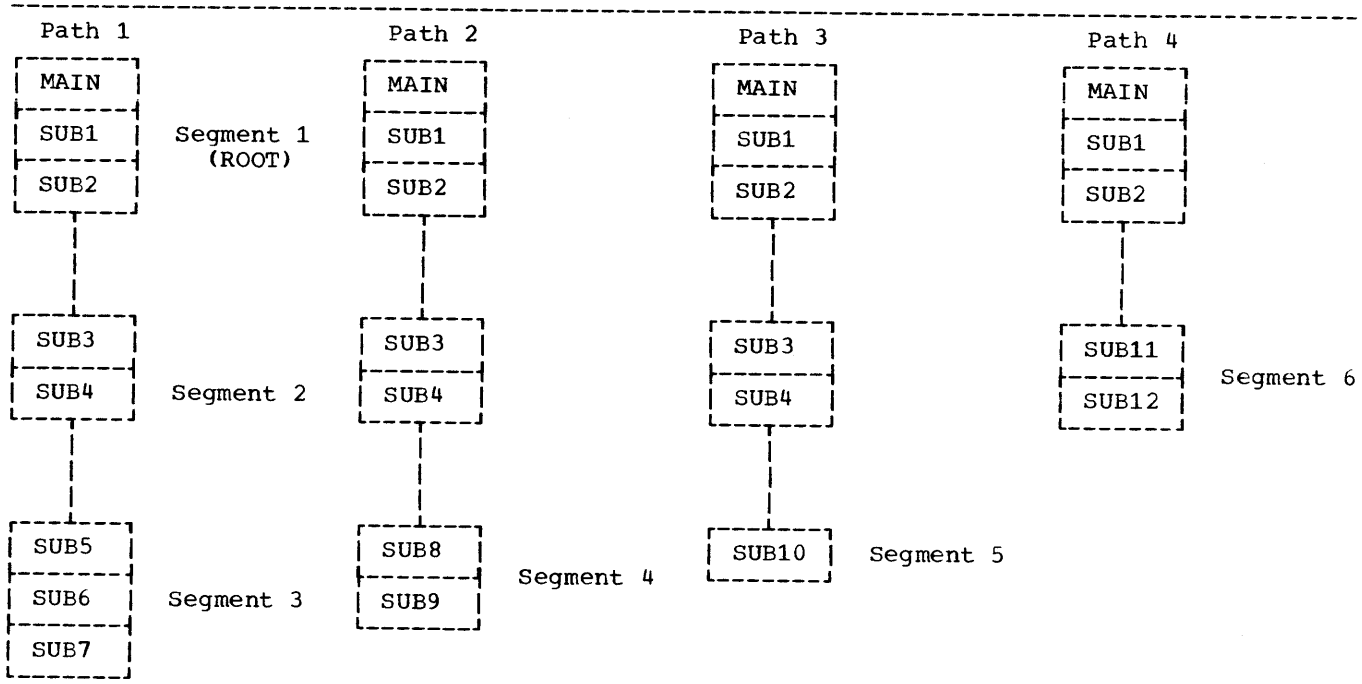


Figure III-9. Overlay Paths Implied by Tree Structure in Figure III-8

Exclusive References

An exclusive reference, one made in any segment to another segment which will overlay it, can be either valid or invalid.

An exclusive reference is considered valid only if there is a reference to the called routine in a segment common to both the segment to be loaded and the segment to be overlaid. Assume, for example, in Figure III-11 that the main program (common segment) contains a call to Segment A but not to Segment B. A reference in Segment B to a routine in Segment A is valid because of the inclusive reference between the common segment and Segment A. (A table in the common segment, supplied by the linkage editor, contains the address of Segment A. The overlay does not destroy this table.) An exclusive reference in Segment A to a routine in Segment B is invalid since the common segment contains no reference to Segment B.

Both valid and invalid exclusive references are considered errors by the linkage editor; however, the user can allow a program containing a valid exclusive reference to be executed. (See the discussion of XCAL and LET options later in this chapter.) Programs containing invalid exclusive references are never executable. For more detailed information on exclusive references, see the appropriate linkage editor and loader publication, as listed in the Preface.

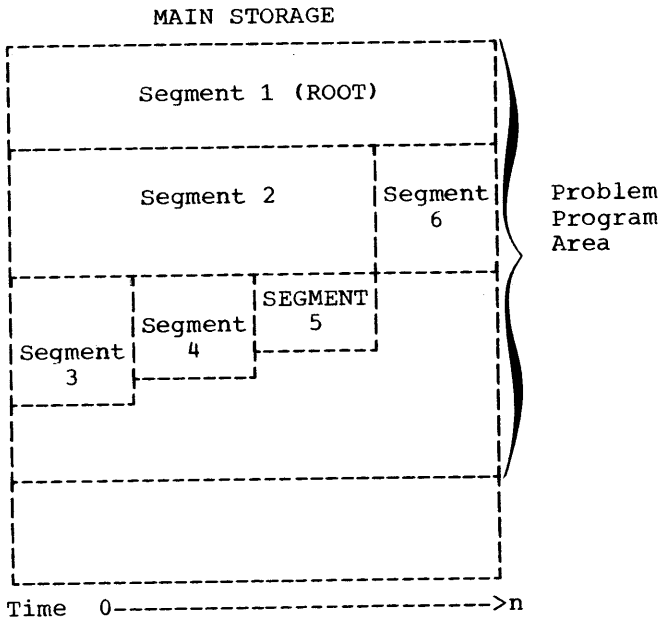


Figure III-10. Overlay Configuration of Program Described in Figure III-8

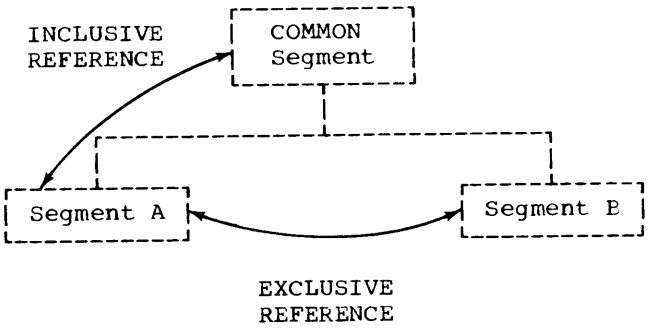


Figure III-11. Communication Between Overlay Segment

COMMON AREAS

The linkage editor treats all FORTRAN COMMON areas as separate subprograms. When modules containing COMMON areas are processed by the linkage editor, the COMMON areas are collected. That is, when two or more blank (unnamed) COMMON areas are encountered in the input to the linkage editor, only the largest of them is retained in the output module. (In the case of named COMMON areas, the question of different lengths does not arise since all named COMMON areas of the same name must be the same length.)

THE OVERLAY PROCESS

Overlay is initiated at execution time in response to a reference to a subprogram which is not already in main storage. The subprogram reference may be either a FUNCTION name or a CALL statement to a SUBROUTINE name. When the reference is executed, the overlay segment containing the required subprogram, as well as any segments in its path not currently in main storage, is loaded.

When a segment is loaded, it overlays any segment in storage with the same relative origin. It also overlays any segments that are lower (farther from the root segment) in the path of the overlaid segment. For example, if segments 1, 2, and 3 in Figures III-8 and III-9 are in main storage when the main program executes a call to subprogram SUB11, segment 6 is called into main storage and segments 2 and 3 will not be available for as long as segment 6 is in main storage.

Whenever a segment is loaded it contains a fresh copy of the program units that it comprises; any data values that may have

been established or altered during previous processing are returned to their initial values each time the segment is loaded. Thus, data values that are to be retained for longer than a single load phase should be placed in the root segment.

Overlay is not initiated when a return is made from a subprogram or when a segment in main storage executes a reference to a subprogram that is already in main storage.

In an overlay program, when blank or named COMMON areas are encountered by the linkage editor, they are collected as described above. Their ultimate location in the output module depends upon which linkage editor control statements are used in the building of the overlay structure (see the discussion of linkage editor control statements below). Overlay structures built without the use of INSERT statements (in which the program units for each segment are included between OVERLAY statements) produce an output module in which the linkage editor "promotes" the COMMON areas automatically. The promotion process places each COMMON area in the lowest segment on the overlay tree which will always be in main storage with any segment containing a reference to it.

Figures III-12 and III-13 show an overlay program as it appears before and after the automatic promotion of COMMON areas. The position of a promoted COMMON area within the segment to which it is promoted is unpredictable.

If INSERT statements are used to structure the overlay program, a blank COMMON area should appear physically in the input stream in the segment to which it belongs. A named COMMON area should either appear physically in the segment to which it belongs or should be placed there with an INSERT statement.

COMMON areas encountered in modules from automatic call libraries are automatically promoted to the root segment. If such COMMON areas are named, they may be positioned by the use of an INSERT statement.

Named COMMON areas in BLOCK DATA subprograms must be exactly as large as any identically named COMMON areas in FORTRAN programs that are to be link edited with the BLOCK DATA subprograms.

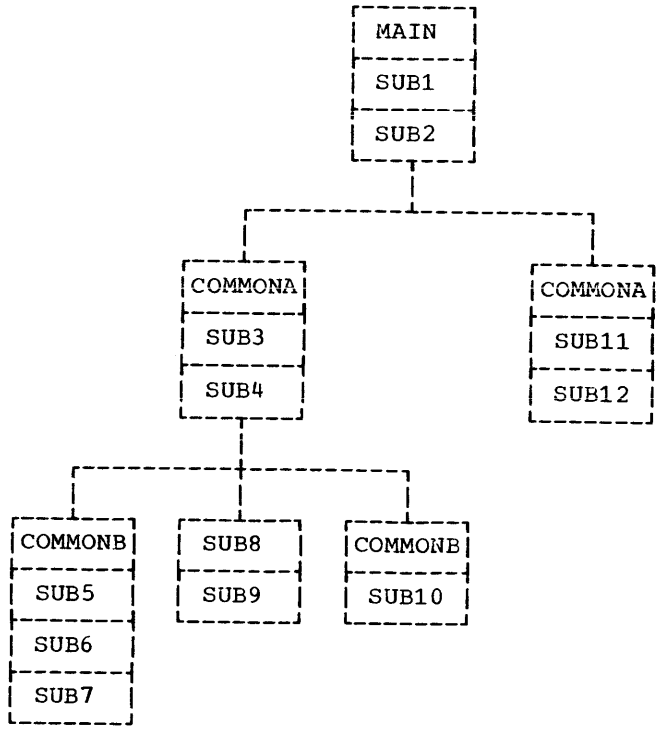


Figure III-12. Overlay Program Before Automatic Promotion of Common Areas

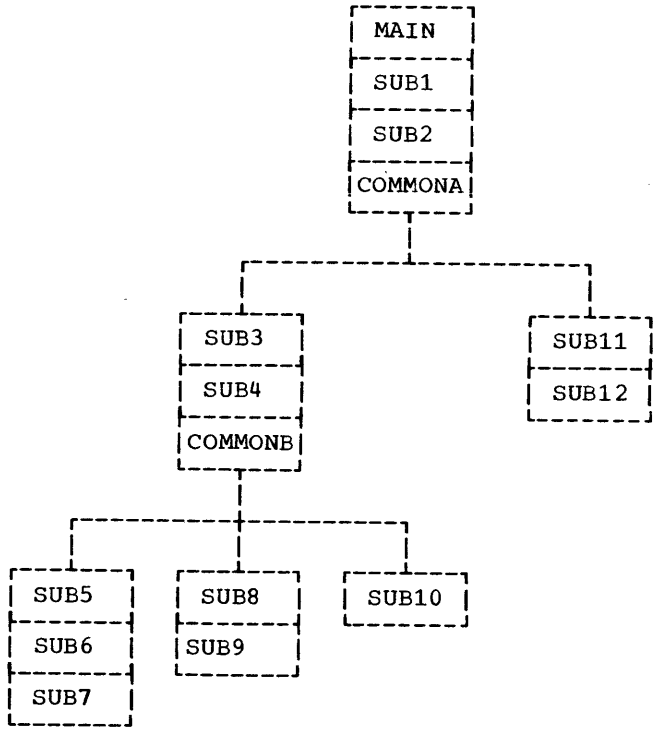


Figure III-13. Overlay Program After Automatic Promotion of Common Areas

CONSTRUCTION OF THE OVERLAY PROGRAM

The programmer communicates his overlay strategy to the operating system in two ways: through the use of special processing options which he specifies in the PARM parameter of the EXEC statement that calls the linkage editor, and through the use of linkage editor control statements. The general functions of these options and statements are described in the section "Linkage Editor and Loader." Those which are of particular interest to the programmer constructing an overlay program are discussed immediately below.

LINKAGE EDITOR CONTROL STATEMENTS

Once the programmer has designed an overlay tree structure for this program, he places the program in that structure by indicating to the linkage editor the relative positions of the segments that make up the tree. The control statements which accomplish this are placed in the input stream following the //SYSLIN DD statement, or after the //LKED.SYSLIN DD statement if a cataloged procedure is used.

The most important control statements for implementing an overlay program are the OVERLAY, INSERT, INCLUDE, and ENTRY statements. The OVERLAY statement indicates the beginning of an overlay segment. The INSERT statement is used to rearrange the sequence of object modules in the resulting load module(s). The INCLUDE statement is used to incorporate input from secondary sources into the load module. The ENTRY statement specified the first instruction to be executed.

OVERLAY Statement

The OVERLAY statement indicates the beginning of an overlay segment. Its format is:

Operation	Operand
OVERLAY	name

where name indicates the beginning of the segment, that is, the symbolic name of the relative origin. The following statement may be used to indicate overlay of segments 2 and 6 in Figure III-8:

```
OVERLAY ALPHA
```

OVERLAY statements are placed directly before the object decks of the first program unit of the new segment, before an INSERT statement specifying the program units to be placed in the segment, or before an INCLUDE statement specifying the program units to be placed in the segment. Assuming that object decks were available, the input deck to the linkage editor for the program in Figures III-8 and III-9 could be arranged as follows:

```
Object deck for     MAIN
                    SUB1
                    SUB2
-------------------------------------
OVERLAY ALPHA
Object deck for     SUB3
                    SUB4
-------------------------------------
OVERLAY BETA
Object deck for     SUB5
                    SUB6
                    SUB7
-------------------------------------
OVERLAY BETA
Object deck for     SUB8
                    SUB9
-------------------------------------
OVERLAY BETA
Object deck for     SUB10
-------------------------------------
OVERLAY ALPHA
Object deck for     SUB11
                    SUB12

ENTRY MAIN
```

INSERT Statement

There are many instances in which it is inconvenient or impossible for the user to position object decks physically in the input stream. Library routines, normally placed in the root segment, and routines compiled in an earlier step in the same job are examples of program units for which the object decks are not available for positioning at the time the job is set up.

The INSERT statement is used to position such control sections in an overlay structure. A control section, or CSECT, is the operating system designation for the smallest separately relocatable unit of a program. Examples of FORTRAN control sections are main programs, subprograms, and blank or named COMMON blocks.

The format of the INSERT statement is:

Operation	Operand
INSERT	csect-name[,csect-name...]

where csect-name indicates the name(s) of the control section(s) to be positioned.

The INSERT statement is placed in the input sequence directly following the OVERLAY statement that specifies the segment origin in which the control section is to be positioned. If the control section is to be positioned in the root segment, the INSERT statement is placed before the first OVERLAY statement.

Using INSERT statements and a FORTRAN source deck, the overlay structure specified in Figures III-8 and III-9 could be implemented as follows:

Input to the compiler:

```
FORTRAN source deck containing units
MAIN through SUB12
```

Input to the linkage editor:

```
ENTRY MAIN
INSERT MAIN,SUB1,SUB2
-----------------------------------
OVERLAY ALPHA
INSERT SUB3,SUB4
-----------------------------------
OVERLAY BETA
INSERT SUB5,SUB6,SUB7
-----------------------------------
OVERLAY BETA
INSERT SUB8,SUB9
-----------------------------------
OVERLAY BETA
INSERT SUB10
-----------------------------------
OVERLAY ALPHA
INSERT SUB11,SUB12
```

If INSERT statements are used more than once in the same program for a control section of the same name, the CSECT will be positioned in the segment specified by the first occurrence of the CSECT name in the input stream. Any additional INSERT statements referring to the CSECT will be ignored, and, at execution time, all references to the CSECT will resolve to the first one positioned. Thus, if a subprogram is required in more than one path, it must be either inserted in the root segment or renamed before being used with an INSERT statement.

INCLUDE Statement

The INCLUDE statement is described in the section "Linkage Editor and Loader." When used in an overlay program, the INCLUDE statement is generally placed in the input stream in the position where the material to be included is required.

It is possible to manipulate the control sections that were added by an INCLUDE statement through the use of the INSERT statement. Assume that the control sections of the overlay program from the previous examples resided in libraries as follows:

LIBA	
BOOK1	BOOK2
MAIN	SUB3
SUB1	SUB4
SUB2	

LIBB
SUB5
SUB6
SUB7
SUB8
SUB9
SUB10
SUB11
SUB12

where LIBA is a partitioned data set with members BOOK1 and BOOK2 and LIBB is a sequential data set.

Then the overlay structure could be implemented by the use of the following control statements:

```
ENTRY MAIN
-----------------------------------
INCLUDE LIBA(BOOK1)
INCLUDE LIBB
OVERLAY ALPHA
-----------------------------------
INCLUDE LIBA(BOOK2)
OVERLAY BETA
INSERT SUB5,SUB6,SUB7
OVERLAY BETA
INSERT SUB8,SUB9
OVERLAY BETA
INSERT SUB10
OVERLAY ALPHA
INSERT SUB11,SUB12
```

ENTRY Statement

The ENTRY statement specifies the first instruction of the program to be executed. Its format is:

Operation	Operand
ENTRY	external-name

where external-name indicates the name of an instruction in the root segment. Usually it will be the name MAIN.

The ENTRY statement may be placed before, between, or after the program units or other control statements in the input stream. An ENTRY statement is necessary in all overlay programs because, after linkage editor processing, the first part of the root segment contains special overlay control information rather than executable code. See the previous examples of overlay implementation for the use and placement of the ENTRY statement.

LINKAGE EDITOR OVERLAY OPTIONS

In addition to the necessary linkage editor control statements, the user implementing an overlay structure must provide certain information to the operating system by means of the PARM parameter of the EXEC statement that calls the linkage editor.

Linkage editor options are described in the section "Linkage Editor and Loader." Those options which are of special interest to the user of the overlay feature are discussed in the following paragraphs.

OVLY indicates that the load module produced will be an overlay structure, as directed by subsequent linkage editor control statements. OVLY must be specified for all overlay processing.

LIST indicates that linkage editor control statements are to be listed in card image format on the system output data set, SYSPRINT.

MAP indicates that the linkage editor is to produce a map of the output module. The map of the output module of an overlay structure shows the control sections grouped by segment. Within each segment, the control sections are listed in ascending order according to their assigned origins. The number of the segment in which each appears is also printed.

XREF indicates that the linkage editor is to produce a cross-reference table of the output module. The cross-reference table includes a module map and a list of all address constants that refer to other control sections. Since the cross-reference table includes a module map, XREF may be substituted for MAP.

XCAL indicates that a valid exclusive call is not to be considered an error, and that the load module is to be marked executable even though improper branches were made between control sections.

LET indicates that any exclusive call (valid or invalid) is accepted. The output module will be marked executable even though certain error or abnormal conditions were found during link editing. At execution time, a valid exclusive call may or may not be executed correctly. An invalid call will usually cause unpredictable results; the requested segment will not be loaded.

OVERLAY EXAMPLE

Assume that the program suggested by Figure III-8 consists of a main program and twelve subroutines. The sole function of the main program is to call the subroutines, and each subroutine prints the message "IN SUBx", where x is a number identifying the subroutine.

Figure III-14 illustrates the input and Figure III-15 the output of the program.

Figure III-14 shows the program in card deck form, as it might be submitted for execution. The EXEC statement specifies the cataloged procedure FORTXCLG, to process three job steps to compile, link edit, and execute. The EXEC statement also specifies the linkage editor options OVLY, XREF, and LIST.

Input to the compile step consists of the statements in the main program unit and in the subroutines.

Input to the link edit step consists of INSERT, OVERLAY, and ENTRY statements, which define the overlay structure of the program.

There is no input to the load module step; however, output from the step is directed to the printer, as defined by the DD statement //GO.FT06F001.

Output from the compile job step is shown in Figure III-15. Note that each program unit is compiled separately. The

FORTRAN IV (H Extended) compiler recognizes the FORTRAN END statement as the last statement in a unit.

Output from the link edit job step is shown in Figure III-16. The LIST option causes the linkage editor control statements to be listed (labeled A). The XREF option causes a cross reference table to be printed for each overlay segment. Each overlay segment consists of the FORTRAN program units specified in an INSERT statement together with any modules called by the linkage editor. For example, the following statement defines one overlay segment:

```
INSERT MAIN,SUB1,SUB2
```

The overlay segments are labeled B through G. The OVLY option causes no printed output.

Output from the load module execution job step is shown in Figure III-17. The messages generated by the subroutines are labeled H.

```
//     EXEC  FORTXCLG,PARM.LKED='OVLY,XREF,LIST'
//FORT.SYSIN DD *  INPUT TO COMPILE JOB STEP
         CALL SUB1
         CALL SUB2
         CALL SUB3
         CALL SUB4
         CALL SUB5
         CALL SUB6
         CALL SUB7
         CALL SUB8
         CALL SUB9
         CALL SUB10
         CALL SUB11
         CALL SUB12
         STOP
         END
         SUBROUTINE SUB1
         DIMENSION A(100,10)
   15    FORMAT (1H0,8HIN SUB1 )
         WRITE (6,15)
         RETURN
         END
         SUBROUTINE SUB2
         DIMENSION A(100,10)
   15    FORMAT (1H0,8HIN SUB2 )
         WRITE (6,15)
         RETURN
         END
         SUBROUTINE SUB3
         DIMENSION A(100,10)
   15    FORMAT (1H0,8HIN SUB3 )
         WRITE (6,15)
         RETURN
         END
         SUBROUTINE SUB4
         DIMENSION A(100,10)
   15    FORMAT (1H0,8HIN SUB4 )
         WRITE (6,15)
         RETURN
         END
         SUBROUTINE SUB5
         DIMENSION A(100,10)
   15    FORMAT (1H0,8HIN SUB5 )
         WRITE (6,15)
         RETURN
         END
         SUBROUTINE SUB6
         DIMENSION A(100,10)
   15    FORMAT (1H0,8HIN SUB6 )
         WRITE (6,15)
         RETURN
         END
         SUBROUTINE SUB7
         DIMENSION A(100,10)
   15    FORMAT (1H0,8HIN SUB7 )
         WRITE (6,15)
         RETURN
         END
         SUBROUTINE SUB8
         DIMENSION A(100,10)
   15    FORMAT (1H0,8HIN SUB 8 )
         WRITE (6,15)
         RETURN
         END
         SUBROUTINE SUB9
         DIMENSION A(100,10)
   15    FORMAT (1H0,8HIN SUB 9 )
         WRITE (6,15)
         RETURN
         END
         SUBROUTINE SUB10
         DIMENSION4A(100,10)
   15    FORMAT (1H0,8HIN SUB 10)
         WRITE (6,15)
         RETURN
         END
         SUBROUTINE SUB11
         DIMENSION A(100,10)
   15    FORMAT (1H0,8HIN SUB 11)
         WRITE (6,15)
         RETURN
         END
         SUBROUTINE SUB12
         DIMENSION A(100,10)
   15    FORMAT (1H0,8HIN SUB 12)
         WRITE (6,15)
         RETURN
         END
/*
//LKED.SYSIN  DD * INPUT TO LINK EDIT JOB STEP
 INSERT MAIN,SUB1,SUB2
 OVERLAY ALPHA
 INSERT SUB3,SUB4
 OVERLAY BETA
 INSERT SUB5,SUB6,SUB7
 OVERLAY BETA
 INSERT SUB8,SUB9
 OVERLAY BETA
 INSERT SUB10
 OVERLAY ALPHA
 INSERT SUB11,SUB12
 ENTRY MAIN
/*
//GO.FT06F001 DD  SYSOUT=A  OUTPUT FROM LOAD MODULE JOB STEP
//
```

Figure III-14. Linkage Editor Overlay Input

```
REQUESTED OPTIONS: NODECK,NOLIST,OPT=0

CPTIONS IN EFFECT: NAME(  MAIN),NOOPTIMIZE,LINECOUNT(60),SIZE(MAX),AUTODBL(NONE),
                   SOURCE,EBCDIC,NOLIST,NODECK,OBJECT,NOMAP,NOFORMAT,NOGOSTMT,NOXREF,NOALC,NOANSF,FLAG(I)

   ISN 0002            CALL SUB1
   ISN 0003            CALL SUB2
   ISN 0004            CALL SUB3
   ISN 0005            CALL SUB4
   ISN 0006            CALL SUB5
   ISN 0007            CALL SUB6
   ISN 0008            CALL SUB7
   ISN 0009            CALL SUB8
   ISN 0010            CALL SUB9
   ISN 0011            CALL SUB10
   ISN 0012            CALL SUB11
   ISN 0013            CALL SUB12
   ISN 0014            STOP
   ISN 0015            END

*OPTIONS IN EFFECT*NAME(  MAIN),NOOPTIMIZE,LINECOUNT(60),SIZE(MAX),AUTODBL(NONE),

*OPTIONS IN EFFECT*SOURCE,EBCDIC,NOLIST,NODECK,OBJECT,NOMAP,NOFORMAT,NOGOSTMT,NOXREF,NOALC,NOANSF,FLAG(I)

*STATISTICS*     SOURCE STATEMENTS =      14, PROGRAM SIZE =      318, SUBPROGRAM NAME =  MAIN

*STATISTICS*  NO  DIAGNOSTICS GENERATED

****** END OF COMPILATION ******                                    105K BYTES OF CORE NOT USED

REQUESTED OPTIONS: NODECK,NOLIST,OPT=0

CPTIONS IN EFFECT: NAME(  MAIN),NOOPTIMIZE,LINECOUNT(60),SIZE(MAX),AUTODBL(NONE),
                   SOURCE,EBCDIC,NOLIST,NODECK,OBJECT,NOMAP,NOFORMAT,NOGOSTMT,NOXREF,NOALC,NOANSF,FLAG(I)

   ISN 0002            SUBROUTINE SUB1
   ISN 0003            DIMENSION A(100,10)
   ISN 0004         15 FORMAT (1HO,8HIN SUB1 )
   ISN 0005            WRITE (6,15)
   ISN 0006            RETURN
   ISN 0007            END

*OPTIONS IN EFFECT*NAME(  MAIN),NOOPTIMIZE,LINECOUNT(60),SIZE(MAX),AUTODBL(NONE),

*OPTIONS IN EFFECT*SOURCE,EBCDIC,NOLIST,NODECK,OBJECT,NOMAP,NOFORMAT,NOGOSTMT,NOXREF,NOALC,NOANSF,FLAG(I)

*STATISTICS*     SOURCE STATEMENTS =       6, PROGRAM SIZE =      218, SUBPROGRAM NAME =  SUB1

*STATISTICS*  NO  DIAGNOSTICS GENERATED

****** END OF COMPILATION ******                                    105K BYTES OF CORE NOT USED

                            .
                            .
                            .
REQUESTED OPTIONS: NODECK,NOLIST,OPT=0

CPTIONS IN EFFECT: NAME(  MAIN),NOOPTIMIZE,LINECOUNT(60),SIZE(MAX),AUTODBL(NONE),
                   SOURCE,EBCDIC,NOLIST,NODECK,OBJECT,NOMAP,NOFORMAT,NOGOSTMT,NOXREF,NOALC,NOANSF,FLAG(I)

   ISN 0002            SUBROUTINE SUB12
   ISN 0003            DIMENSION A(100,10)
   ISN 0004         15 FORMAT (1HO,8HIN SUB12)
   ISN 0005            WRITE (6,15)
   ISN 0006            RETURN
   ISN 0007            END

*OPTIONS IN EFFECT*NAME(  MAIN),NOOPTIMIZE,LINECOUNT(60),SIZE(MAX),AUTODBL(NONE),

*OPTIONS IN EFFECT*SOURCE,EBCDIC,NOLIST,NODECK,OBJECT,NOMAP,NOFORMAT,NOGOSTMT,NOXREF,NOALC,NOANSF,FLAG(I)

*STATISTICS*     SOURCE STATEMENTS =       6, PROGRAM SIZE =      218, SUBPROGRAM NAME = SUB12

*STATISTICS*  NO  DIAGNOSTICS GENERATED

****** END OF COMPILATION ******                                    105K BYTES OF CORE NOT USED

*STATISTICS*  NO  DIAGNOSTICS THIS STEP
```

Figure III-15. Linkage Editor Overlay Output -- Compile Job Step

```
F88-LEVEL LINKAGE EDITOR OPTIONS SPECIFIED OVLY,XREF,LIST
          VARIABLE OPTIONS USED -  SIZE=(92160,8192)-          DEFAULT OPTION(S) USED
IEW0000     INSERT MAIN,SUB1,SUB2
IEW0000     OVERLAY ALPHA
IEW0000     INSERT SUB3,SUB4
IEW0000     OVERLAY BETA
IEW0000     INSERT SUB5,SUB6,SUB7
(A) IEW0000     OVERLAY BETA
IEW0000     INSERT SUB8,SUB9
IEW0000     OVERLAY BETA
IEW0000     INSERT SUB10
IEW0000     OVERLAY ALPHA
IEW0000     INSERT SUB11,SUB12
IEW0000     ENTRY MAIN
****MAIN      DOES NOT EXIST BUT HAS BEEN ADDED TO DATA SET
```

CROSS REFERENCE TABLE

CONTROL SECTION NAME	ORIGIN	LENGTH	SEG. NO.	ENTRY NAME	LOCATION	NAME	LOCATION	NAME	LOCATION	NAME	LOCATION
$SEGTAB	00	30	1								
MAIN	30	13E	1								
SUB1	170	DA	1								
SUB2	250	DA	1								
IHOECOMH*	330	DC4	1								
				IBCOM#	35C	FDIOCS#	418	INTSWTCH	10E0		
IHOCOMH2*	10F8	975	1								
				SEQDASD	1462						
IHOFCVTH*	1A70	A07	1								
				ADCON#	1A70	FCVAOUTP	1B1A	FCVLOUTP	1BAA	FCVZOUTP	1D06
				FCVIOUTP	20AA	FCVEOUTP	219C	FCVCOUTP	219C	INT6SWCH	23F8
IHOEFNTH*	2478	7C8	1								
				ARITH#	2478	ADJSWTCH	29D8				
IHOEFIOS*	2C40	10F0	1								
				FIOCS#	2C40	FIOCSBEP	2C46				
IHOFIOS2*	3D30	5AC	1								
IHOUOPT *	42E0	318	1								
IHOFCONI*	45F8	2FD	1								
				FQCONI#	45F8						
IHOFCONO*	48F8	558	1								
				FQCONO#	48F8						
IHOERRM *	4E50	5EC	1								
				ERRMON	4E50	IHOERRE	4E68				
IHOUATBL*	5440	208	1								
IHOFTEN *	5648	198	1								
				FTEN#	5648						
IHOETRCH*	57E0	2A6	1								
				IHOTRCH	57E0	ERRTRA	57E8				
$ENTAB	5A88	84	1								

(B)

LOCATION	REFERS TO SYMBOL	IN CONTROL SECTION	SEG. NO.	LOCATION	REFERS TO SYMBOL	IN CONTROL SECTION	SEG. NO.
A8	SUB1	SUB1	1	AC	SUB2	SUB2	1
B0	SUB3	SUB3	2	B4	SUB4	SUB4	2
B8	SUB5	SUB5	3	BC	SUB6	SUB6	3
C0	SUB7	SUB7	3	C4	SUB8	SUB8	4
C8	SUB9	SUB9	4	CC	SUB10	SUB10	5
D0	SUB11	SUB11	6	D4	SUB12	SUB12	6
D8	IBCOM#	IHOECOMH	1	208	IBCOM#	IHOECOMH	1
2E8	IBCOM#	IHOECOMH	1	418	SEQDASD	IHOCOMH2	1
470	IHOCOMH2	IHOCOMH2	1	FEC	ADCON#	IHOFCVTH	1
FE4	FIOCS#	IHOEFIOS	1	FF0	ARITH#	IHOEFNTH	1
100C	ADJSWTCH	IHOEFNTH	1	FC0	IHOUOPT	IHOUOPT	1
FF4	FCVEOUTP	IHOFCVTH	1	FF8	FCVLOUTP	IHOFCVTH	1
FFC	FCVIOUTP	IHOFCVTH	1	1000	FCVCOUTP	IHOFCVTH	1
1004	FCVAOUTP	IHOFCVTH	1	1008	FCVZOUTP	IHOFCVTH	1
10E4	IHOASYNC	$UNRESOLVED(W)		F44	IHOERRM	IHOERRM	1
F98	IHOERRE	IHOERRM	1	FC8	IHOCOMH2	IHOCOMH2	1
F9C	IHOCOMH2	IHOCOMH2	1	FA0	IHOCOMH2	IHOCOMH2	1
FA4	IHOCOMH2	IHOCOMH2	1	FA8	IHOCOMH2	IHOCOMH2	1
1228	IHOECOMH	IHOECOMH	1	12D8	IHOECOMH	IHOECOMH	1
18CD	IHOECOMH	IHOECOMH	1	18DD	IHOECOMH	IHOECOMH	1
18ED	IHOECOMH	IHOECOMH	1	2268	IBCOM#	IHOECOMH	1
2264	IHOERRM	IHOERRM	1	22B8	FQCONO#	IHOFCONO	1
22BC	FQCONI#	IHOFCONI	1	2A34	IBCOM#	IHOECOMH	1
2A38	INTSWTCH	IHOECOMH	1	29D4	INT6SWCH	IHOFCVTH	1
29D0	IHOUOPT	IHOUOPT	1	2A40	ADCON#	IHOFCVTH	1
2A3C	FIOCS#	IHOEFIOS	1	2B40	IHOERRM	IHOERRM	1
2DA8	IHOASYNC	$UNRESOLVED(W)		2DA0	IHOERRM	IHOERRM	1
2DA4	IHOFIOS2	IHOFIOS2	1	3AC0	IHOUATBL	IHOUATBL	1
3ACC	IBCOM#	IHOECOMH	1	3AE1	IHOFIOS2	IHOFIOS2	1
3AF8	IHOFIOS2	IHOFIOS2	1	3D29	IHOFIOS2	IHOFIOS2	1
4884	FTEN#	IHOFTEN	1	4CF4	FTEN#	IHOFTEN	1
542C	IHOUOPT	IHOUOPT	1	5430	IBCOM#	IHOECOMH	1
5434	IHOTRCH	IHOETRCH	1	5438	FIOCSBEP	IHOEFIOS	1
5968	IBCOM#	IHOECOMH	1	596C	ADCON#	IHOFCVTH	1
5974	FIOCSBEP	IHOEFIOS	1				

Figure III-16. Link Edit Overlay Output -- Link Edit Job Step

```
CONTROL SECTION                          ENTRY
  NAME     ORIGIN  LENGTH  SEG. NO.      NAME   LOCATION     NAME   LOCATION     NAME   LOCATION     NAME   LOCATION
SUB3         5B10      DA      2
SUB4         5BF0      DA      2

LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  SEG. NO.   LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  SEG. NO.
   5BA8           IBCOM#          IHOECOMH          1        5C88           IBCOM#          IHOECOMH          1
```

C

```
CONTROL SECTION                          ENTRY
  NAME     ORIGIN  LENGTH  SEG. NO.      NAME   LOCATION     NAME   LOCATION     NAME   LOCATION     NAME   LOCATION
SUB5         5CD0      DA      3
SUB6         5DB0      DA      3
SUB7         5E90      DA      3

LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  SEG. NO.   LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  SEG. NO.
   5D68           IBCOM#          IHOECOMH          1        5E48           IBCOM#          IHOECOMH          1
   5F28           IBCOM#          IHOECOMH          1
```

D

```
CONTROL SECTION                          ENTRY
  NAME     ORIGIN  LENGTH  SEG. NO.      NAME   LOCATION     NAME   LOCATION     NAME   LOCATION     NAME   LOCATION
SUB8         5CD0      DA      4
SUB9         5DB0      DA      4

LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  SEG. NO.   LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  SEG. NO.
   5D68           IBCOM#          IHOECOMH          1        5E48           IBCOM#          IHOECOMH          1
```

E

```
CONTROL SECTION                          ENTRY
  NAME     ORIGIN  LENGTH  SEG. NO.      NAME   LOCATION     NAME   LOCATION     NAME   LOCATION     NAME   LOCATION
SUB10        5CD0      DA      5

LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  SEG. NO.   LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  SEG. NO.
   5D68           IBCOM#          IHOECOMH          1
```

F

```
CONTROL SECTION                          ENTRY
  NAME     ORIGIN  LENGTH  SEG. NO.      NAME   LOCATION     NAME   LOCATION     NAME   LOCATION     NAME   LOCATION
SUB11        5B10      DA      6
SUB12        5BFC      DA      6

LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  SEG. NO.   LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  SEG. NO.
   5BA8           IBCOM#          IHOECOMH          1        5C88           IBCOM#          IHOECOMH          1

ENTRY ADDRESS      30
TOTAL LENGTH     5F70
```

G

Figure III-16. Link Edit Overlay Output -- Link Edit Job Step (Part 2 of 2)

```
IN SUB1
IN SUB2
IN SUB3
IN SUB4
IN SUB5
IN SUB6
IN SUB7
IN SUB8
IN SUB9
IN SUB10
IN SUB11
IN SUB12
```

Figure III-17. Linkage Editor Overlay Output -- Load Module Execution Job Step

EXTENDED ERROR HANDLING FACILITY

The extended error handling facility provides the FORTRAN programmer with greater control over load module errors. This facility is specified at program installation time through the parameter OPTERR=INCLUDE in the FORTLIB macro instruction.

When a program error occurs, the user is given:

- Messages more informative than those issued with standard diagnostic facilities
- The ability to continue execution after the error
- Either standard FORTRAN corrective action with continued execution or, optionally, user-specified corrective action

When an error occurs with extended error handling in effect, a short message text is printed along with an error identification number. The data in error (or some other associated information) is printed as part of the message text. A summary error count, printed when a job is completed, informs the user how many times each error occurred. For a complete listing of compiler and library messages, see the publication OS FORTRAN IV (H Extended) Compiler and Library (Mod II) Messages, Order No. SC28-6865.

A traceback map, tracing the subroutine flow back to the main program, is printed after each error occurrence; execution then continues. (If the extended error handling facility is not specified, a traceback map is printed only for errors causing program termination and for error IHO218I if the ERR= option has been specified in a READ statement.)

For each error condition detected, the user has both dynamic and default control over:

- The number of times the error is allowed to occur before program termination
- The maximum number of times each message may be printed
- Whether or not the traceback map is to be printed with the message
- Whether or not a user-written error-exit routine is to be called

The action that takes place is governed by information stored in an area of main storage called the option table. (A permanent copy of the option table is maintained in the FORTRAN library.)

FUNCTIONAL CHARACTERISTICS

When an error is detected, the FORTRAN error monitor (ERRMON) receives control.

The error monitor prints the necessary diagnostic and informative messages and then takes one of the following actions:

- Terminates the job
- Returns control to the calling routine, which takes a standard corrective action and then continues execution
- Calls a user-written closed subroutine to correct the data in error and then returns to the routine that detected the error, which then continues execution

The actions of the error monitor are controlled by settings in the option table. The option table consists of a doubleword preface, followed by a doubleword entry for each error condition. (If the extended error handling facility is not specified, the option table is reduced to the preface alone.) IBM provides a standard set of 97 entries; the programmer can provide additional entries at program installation time. Table III-3 shows the values for each error condition. If an option recorded in a table entry does not apply to a particular error condition, it is shown as not applicable (NA).

Table III-3. Option Table Default Values

Error Code	Number of Errors Allowed	Number of Messages Allowed	Print Control	Modifiable Entry	Print Buffer Content	Traceback Allowed	Standard Corrective Action	User Exit
			Option Bits					
205	1	1	NA	No	No	No	No	No
206	10	5	NA	Yes	NA	Yes	Yes	No
207	10	5	NA	Yes	NA	Yes	Yes	No
208	Unlimited	5	NA	Yes	NA	Yes	Yes	No
209	10	5	NA	Yes	NA	Yes	Yes	No
210	Unlimited	10	NA	Yes	NA	Yes	Yes	No[1]
211	10	5	NA	Yes	NA	Yes	Yes[1]	No
212	10	5	No[2]	Yes	Yes	Yes	Yes	No
213	10	5	NA	Yes	NA	Yes	Yes	No
214	10	5	NA	Yes	NA	Yes	Yes	No
215	Unlimited	5	NA	Yes	Yes	Yes	Yes	No
216	10	5	NA	Yes	NA	Yes	Yes[3]	No
217	1[4]	1	NA	Yes	NA	Yes	Yes	No
218	10[5]	5	NA	Yes	Yes[5]	Yes	Yes	No
219	10[6]	5	NA	Yes	NA	Yes	Yes	No
220	10	5	NA	Yes	NA	Yes	Yes	No
221	10	5	NA	Yes	NA	Yes	Yes	No
222	10	5	NA	Yes	Yes	Yes	Yes	No
223	10	5	NA	Yes	Yes	Yes	Yes	No
224	10	5	NA	Yes	Yes	Yes	Yes	No
225	10	5	NA	Yes	Yes	Yes	Yes	No
226	10	5	NA	Yes	Yes	Yes	Yes	No
227	10	5	NA	Yes	NA	Yes	Yes	No
229	10	5	NA	Yes	NA	Yes	Yes	No
230	1	1	NA	No	NA	Yes	No	No
231	10	5	NA	Yes	NA	Yes	Yes	No
232	10	5	NA	Yes	NA	Yes	Yes	No
233-237	10	5	NA	Yes	NA	Yes	Yes	No
240	1	1	NA	No	NA	Yes	No	No
241-285	10	5	NA	Yes	NA	Yes	Yes	No
286	10	5	NA	Yes	NA	Yes	Yes	No
287	10	5	NA	Yes	NA	Yes	Yes	No
288	10	5	NA	Yes	NA	Yes	Yes	No
289-301	10	5	NA	Yes	NA	Yes	Yes	No

[1]No corrective action is taken except to continue execution. For boundary alignment, the corrective action is part of the support for misalignment.

[2]If a print control character is not supplied, the overflow line is not shifted to incorporate the print control character. Thus, if the device is tape, the data is intact, but if the device is a printer, the first character of the overflow line is not printed but is treated instead as the print control. Unless the user has planned the overflow, the first character would be random and thus the overflow print line control can be any of the possible ones. It is suggested that when the device is a printer, the option be changed to provide single spacing.

[3]Corrective action consists of return to execution for SLITE.

[4]It is not considered an error if the END parameter is present in a READ statement. No message or traceback is printed and the error count is not altered.

[5]For an input/output error, the buffer may have been partially filled or not filled at all when the error was detected. Thus, the buffer contents could be blank when printed. When an ERR parameter is specified in a READ statement, it is honored even though the error occurrence is greater than the amount allowed.

[6]The count field does not necessarily mean that up to 10 missing DD cards will be detected in a single debugging run, since a single WRITE performed in a loop could cause 10 occurrences of the message for the same missing DD card.

The field that is defined as the user-exit address also serves as a means of specifying standard corrective action. When the table entry contains an address, the user exit is specified; when it contains the integer 1, standard correction is specified. It is not possible at program installation time to create an option table entry with the user-exit address specified. The user exit must be specified by altering the option table at execution time. To specify that no corrective action, either standard or user-written, is to be taken, the table entry must specify that only one error is to be allowed before termination of execution.

To make changes to the option table dynamically at load module execution time, the programmer calls one of four subroutines; these subroutines are summarized below.

SUBPROGRAMS FOR THE EXTENDED ERROR HANDLING FACILITY

IBM provides four subroutines for use in extended error handling: ERRSAV, ERRSTR, ERRSET, and ERRTRA. These subroutines allow access to the option table to alter it dynamically.

Certain option table entries may be protected against alteration when the option table is set up. If a request is made by means of CALL ERRSTR or CALL ERRSET to alter such an entry, the request is ignored. See Table III-3 for those IBM-supplied option table entries that cannot be altered.

Changes made dynamically are in effect for the duration of the program that made the change. Only the current copy of the option table in main storage is affected; the copy in the FORTRAN library remains unchanged. All passed parameters, unless otherwise indicated, are 4-byte (fullword) integers.

ERRSAV Subroutine

The CALL ERRSAV statement is used to modify an entry temporarily. The statement causes an option table entry to be copied into an 8-byte storage area accessible to the FORTRAN programmer. CALL ERRSAV has the format:

```
CALL ERRSAV (ierno, tabent)
```

where:

ierno
: is the error number to be referenced in the option table. Should any number not within the range of the option table be used, an error message will be printed.

tabent
: is the name of an 8-byte storage area where the option table entry is to be stored.

An example of CALL ERRSAV is:

```
CALL ERRSAV (213,ALTERX)
```

The example states that error number 213 is to be stored in the area named ALTERX.

ERRSTR Subroutine

To store an entry in the option table, the following statement is used:

```
CALL ERRSTR (ierno,tabent)
```

where:

ierno
: is the error number for which the entry is to be stored in the option table. Should any number not within the range of the option table be used, an error message will be printed.

tabent
: is the name of an 8-byte storage area containing the table entry data.

An example of CALL ERRSTR is:

```
CALL ERRSTR (213,ALTERX)
```

The example states that error number 213. Stored in ALTERX, is to be restored to the option table.

Table III-4. Corrective Action After Error Occurrence (Part 1 of 2)

Error Code	Parameters Passed to User[1]	Standard Corrective Action	User-Supplied Corrective Action
205	A,B,D	Program Termination	See Note 2
206	A,B,I	I=low order part of number for input too large.	User may alter I (See note 3).
211	A,B,C	Treat format field containing C as end of FORMAT statement.	(a) If compiled FORMAT statement, put hexadecimal equivalent of character in C (see Note 1). (b) If variable format, move EBCDIC character into C (see Note 1).
212	A,B,D	Input: Ignore remainder of I/O list. Output: Continue by starting new output record. Supply carriage control character if required by Option Table.	See Note 2.
213	A,B,D	Ignore remainder of I/O list.	See Note 2.
214	A,B,D	Input: Ignore remainder of I/O list. Ignore input/output request if for ASCII tape. Output: If unformatted write initially requested, change record format to VS. If formatted write initally requested, ignore input/output request.	If user correction is requested, the remainder of the I/O list is ignored.
215	A,B,E	Substitute zero for the invalid character.	The character placed in E will be substituted for the invalid character; input/output operations may not be performed (see Note 1).
217	A,B,D	Increment FORTRAN sequence number and read next file.	See Note 2.
218[2]	A,B,D,F	Ignore remainder of I/O list.	See Note 2.
219-224	A,B,D	Ignore remainder of I/O list.	See Note 2.
225	A,B,E	Substitute zero for the invalid character.	The character placed in E will be substituted for the invalid character (see Note 1).
226	A,B,R	R=0 for input number too small; R=16^{63}-1 for input number too large.	User may alter R.
227	A,B,D	Ignore remainder of I/O list.	See Note 2.
229	A,B,D	Ignore remainder of I/O list.	See Note 2.
231	A,B,D	Ignore remainder of I/O list.	See Note 2.

Table III-4. Corrective Action After Error Occurrence (Part 2 of 2)

Error Code	Parameters Passed to Use[1]	Standard Corrective Action	User Supplied Corrective Action
232	A,B,D,G	Ignore remainder of I/O list.	See Note 2.
233	A,B,D	Change record number to list maximum allowed (32,000).	See Note 2.
234-236	A,B,D	Ignore remainder of I/O list.	See Note 2.
237	A,B,D,F	Ignore remainder of I/O list.	See Note 2.
238	A,B,D	Ignore remainder of I/O list.	See Note 2.
286	A,B,D	Ignore Request	See Note 2.
287	A,B,D	Ignore Request	See Note 2.
288	A,B,D	Implied Wait	See Note 2.

[1]

Parameter Code	Meaning
A	Address of return code field (INTEGER*4)
B	Address of error number (INTEGER*4)
C	Address of invalid format character (LOGICAL*1)
D	Address of data set reference number (INTEGER*4)
E	Address of invalid character (LOGICAL*1)
F	Address of DECB
G	Address of record number requested (INTEGER*4)
I	Result after conversion (INTEGER*4)
R	Result after conversion (REAL*4)

[2]If error condition 218 (input/output error detected) occurs while error messages are being written on the object error data set, the message is written on the console and the job is terminated.

If no DD card has been supplied for the object error data set, error message IHO219I is written out on the console and the job is terminated.

Notes:

1. Alternatively, the user can set the return code to 0, thus requesting a standard corrective action.
2. If the error was not caused during asynchronous input/output processing, the user exit-routine cannot perform any asynchronous I/O operation and, in addition, may not perform any REWIND, BACKSPACE, or ENDFILE operation. If the error was caused during asynchronous input/output processing, the user cannot perform any input/output operation. On return to the library, the remainder of the input/output request will be ignored.
3. The user exit routine may supply an alternative answer for the setting of the result register. The routine should always set an INTEGER*4 variable and the FORTRAN library will load fullword or halfword depending on the length of the argument causing the error.

ERRSET Subroutine

The CALL ERRSET statement permits the user to change up to five different options. It consists of six parameters. The last four parameters are optional, but each omitted parameter must have its place noted by a comma and a zero if succeeding parameters are specified. (Omitted parameters at the end of the list require no place notation.) CALL ERRSET has the format:

```
CALL ERRSET (ierno,inoal,inomes,
   itrace,iusadr,irange)
```

where:

ierno
: is the error number to be referenced in the option table. Should any

number not within the range of the option table be used, an error message will be printed. (Note that if ierno is specified as 212, there is a special relationship between the ierno and irange parameters. See the explanation for irange.)

inoal
is an integer specifying the number of errors permitted before execution is terminated. If inoal is specified as either zero or a negative number, the specification is ignored, and the number-of-errors option is not altered. If a value of more than 255 is specified, an unlimited number of errors is permitted.

inomes
is an integer indicating the number of messages to be printed. A negative value specified for inomes causes all messages to be suppressed; a specification of zero indicates that the number-of-messages option is not to be altered. If a value greater than 255 is specified, an unlimited number of error messages is permitted.

itrace
is an integer whose value may be 0, 1, or 2. A specification of 0 indicates the option is not to be changed; a specification of 1 requests that no traceback be printed after an error occurrence; a specification of 2 requests the printing of a traceback after each error occurrence. (If a value other than 1 or 2 is specified, the option remains unchanged.)

iusadr
specifies one of the following:

1. The value 1, as a 4-byte integer, indicating that the option table is to be set to show no user-exit routine (i.e., standard corrective action is to be used when continuing execution).

2. The name of a closed subroutine that is to be executed after the occurrence of the error identified by ierno. The name must appear in an EXTERNAL statement in the source program, and the routine to which control is to be passed must be available at link editing time.

3. The value 0, indicating that the table entry is not to be altered.

irange
serves a double function. It specifies one of the following:

1. An error number higher than that specified in ierno. This number indicates that the options specified for the other parameters are to be applied to the entire range of error conditions encompassed by ierno and irange. (If irange specifies a number lower than ierno, the parameter is ignored, unless ierno specifies the number 212.)

2. A print control character if ierno specified 212. The value 1 is specified to provide single spacing for an overflow line (standard fixup for WRITE statements). If a value other than 1 is specified, no print control is provided.

The default value 0 is assumed if the parameter is omitted (i.e., no print control is provided, and the values specified for all parameters apply only to the error condition number in ierno).

Examples of CALL ERRSET are:

Example 1:

```
CALL ERRSET (310,20,5,0,MYERR,320)
```

This example specifies the following:

1. Error condition 310 (ierno)
2. The error condition may occur up to 20 times (inoal)
3. The corresponding error message may be printed up to 5 times (inomes)
4. The default for traceback information is to remain in force (itrace)
5. The user-written routine MYERR is to be executed after each error occurrence (iusadr)
6. The same options are to apply to all error conditions from 310 to 320 (irange)

Example 2:

```
CALL ERRSET (212,10,5,2,1,1)
```

This example specifies:

1. Error condition 212
2. The condition may occur up to 10 times

3. The corresponding message may be printed up to 5 times

4. Traceback information is to be displayed after each error occurrence

5. Standard corrective action is to be executed after an error

6. Print control is to be employed

   For illustration purposes, this example explicitly specifies all default options except in requesting print control.

Example 3:

```
CALL ERRSET (212,0,0,0,0,1)
```

This example illustrates an alternate method of specifying exactly the same options as the second example. It states that no changes are to be made to default settings except in requesting print control.

ERRTRA Subroutine

The CALL ERRTRA statement permits the user to dynamically request a traceback and continued execution. It has the format:

```
CALL ERRTRA
```

The call has no parameters.

USER-SUPPLIED ERROR HANDLING

The user has the ability of calling, in his own program, the FORTRAN error monitor (ERRMON) routine, the same routine used by FORTRAN itself when it detects an error. ERRMON examines the option table for the appropriate error number and its associated entry and takes the actions specified. If a user-exit address has been specified, ERRMON transfers control to the user-written routine indicated by that address. Thus, the user has the option of handling errors in one of two ways: (1) simply by calling ERRMON without supplying a user-written exit routine; or (2) by calling ERRMON and providing a user-written exit routine.

In either case, certain planning is required at the installation level. For example, error numbers must be assigned to error conditions to be detected by the user, and additional option table entries must be made available for these conditions. The routine that uses the error monitor for error service should have the status of an installation general-purpose function similar to the IBM-supplied mathematical functions. The number of installation error conditions must be known when the FORTRAN library is created at program installation time, so that entries will be provided in the option table by the ADDNTRY parameter of the FORTLIB macro instruction. The error numbers chosen for user subprograms are restricted in range. IBM-designated error conditions have reserved error codes from 000 to 301. Error codes for installation-designated error situations must be assigned in the range 302 to 899. The error code is used by FORTRAN to find the proper entry in the option table.

To call the ERRMON routine, the following statement is used:

```
CALL ERRMON (imes,iretcd,ierno
   [,data1,data2,...] )
```

where:

imes
: is the name of an array aligned on a fullword boundary, that contains, in EBCDIC characters, the text of the message to be printed. The number of the error condition should be included as part of the text, because the error monitor prints only the text passed to it. The first item of the array contains an integer whose value is the length of the message. Thus, the first four bytes of the array will not be printed. If the message length is greater than the length of the buffer, it will be printed on two or more lines of printed output.

iretcd
: is an integer variable made available to the error monitor for the setting of a return code. The following codes can be set:

  0 -- The option table or user-exit routine indicates that standard correction is required.

  1 -- The option table indicates that a user exit to a corrective routine has been executed. The function is to be reevaluated using arguments supplied in the parameters data1,data2 . . . . For input/output type errors, the value 1 indicates that standard correction is not wanted.

ierno
is the error condition number in the option table. Should any number not within the range of the option table be specified, an error message will be printed.

data1,data2 . . .
are variable names in an error-detecting routine for the passing of arguments found to be in error. One variable must be specified for each argument. Upon return to the error-detecting routine, results obtained from corrective action are in these variables. Because the content of the variables can be altered, the locations in which they are placed should be used only in the CALL statement to the error monitor; otherwise, the user of the function may have literals or variables destroyed.

Since data1 and data2 are the parameters which the error monitor will pass to a user-written routine to correct the detected error, care must be taken to make sure that these parameters agree in type and number in the call to ERRMON and in a user-written corrective routine, if one exists.

Note: If optimization has been requested, current values of variables may not be correct.

An example of CALL ERRMON is:

```
CALL ERRMON(MYMSG,ICODE,315,D1,D2)
```

The example states that the message to be printed is contained in an array named MYMSG, the field named ICODE is to contain the return code, the error condition number to be investigated is 315, and arguments to be passed to the user-written routine are contained in fields named D1 and D2.

Figure III-18 illustrates the use of the CALL ERRSET and CALL ERRMON statements in a program utilizing a user-supplied subprogram to handle divide-by-zero situations. The CALL ERRSET and CALL ERRMON statements are highlighted for easier reference.

User-Supplied Exit Routine

When a user-exit address is supplied in the option table entry for a given error number, the error monitor calls the specified subroutine for corrective action. The subroutine may be user-written and is called by assembler-language code equivalent to the following statement:

```
CALL x (iretcd,ierno,data1,data2...)
```

where:

x
is the name of the subroutine whose address was placed into the option table by the iusadr parameter of the CALL ERRSET statement. (Interpretations of the other parameters -- iretcd, ierno, data1, data2 -- are the same as those for the CALL ERRMON statement.)

If an input/output error is detected (i.e., an error for codes 211-237, 286-288), and the error originally was caused by a FORTRAN input/output statement, subroutine x must not execute any FORTRAN input/output statements, but may issue an asynchronous input/output statement. If the original error was caused by an asynchronous input/output statement, subroutine x may issue a FORTRAN input/output statement but must not issue an asynchronous input/output statement. Similarly, if errors for codes 216 or 241-301 occur, the subroutine x must not call the library routine that detected the error or any routine which uses that library routine. For example, a statement such as

```
R = A ** B
```

cannot be used in the exit routine for error 252, because the FORTRAN library subroutine FRXPR# uses the function subprogram EXP, which detects error 252.

Note: Although a user-written corrective routine may change the setting of the return code (iretcd), such a change is subject to the following restrictions:

1. If iretcd is set to 0, then data1 and data2 must not be altered by the corrective routine, since standard corrective action is requested. If data1 and data2 are altered when iretcd is set to 0, the operations that follow will have unpredictable results.

2. Only the values 0 and 1 are valid for iretcd. A user-exit routine must ensure that one of these values is used if it changes the return code setting. A value other than 0 or 1 will cause unpredictable results.

```
//SAMPLE JOB     1,SAMPLE,MSGLEVEL=1
//STEP1  EXEC    FORTXCLG
//FORT.SYSIN DD *
C       MAIN PROGRAM THAT USES THE SUBROUTINE DIVIDE
        COMMON E
        EXTERNAL FIXDIV
C       SET UP OPTION TABLE WITH ADDRESS OF USER EXIT

        CALL ERRSET(302,30,5,1,FIXDIV)

        E=0
C       GET VALUES TO CALL DIVIDE WITH
        READ(5,9) A,B
        IF(B) 1,2,1
2       E=1.0
1       CALL DIVIDE(A,B,C)
        WRITE(6,10)C
9       FORMAT(2E20.8)
10      FORMAT('1',2E20.8)
        STOP
        END
        SUBROUTINE DIVIDE(A,B,C)
C       ROUTINE TO PERFORM THE CALCULATION C=A/B
C       IF B=0 THEN USE ERROR MESSAGE FACILITY TO SERVICE ERROR
C       PROVIDE MESSAGE TO BE PRINTED
        DIMENSION MES(4)
        DATA MES(1)/12/,MES(2)/' DIV'/,MES(3)/'302I'/,MES(4)/' B=0'/
        DATA RMAX/Z7FFFFFFF/
C       MESSAGE TO BE PRINTED IS
C       DIV302I B=0
C       ERROR CODE 302 IS AVAILABLE AND ASSIGNED TO THIS ROUTINE
C       STEP1 SAVE A,B IN LOCAL STORAGE
        D=A
        E=B
C       STEP2   TEST FOR ERROR CONDITION
100     IF(E) 1,2,1
C       NORMAL CASE -- COMPUTE FUNCTION
1       C=D/E
        RETURN
C       STEP3  ERROR DETECTED CALL ERROR MONITOR

2       CALL ERRMON(MES,IRETCD,302,D,E)

C       STEP 4   BE READY TO ACCEPT A RETURN FROM THE ERROR MONITOR
        IF(IRETCD) 5,6,5
C       IF IRETCD=0 STANDARD RESULT IS DESIRED
C       STANDARD RESULT WILL BE C=LARGEST NUMBER IF D IS NOT ZERO
C       CR C=0 IF E=0 AND D=0
6       IF(D) 7,8,7
C       C=0.0
        GO TO 9
7       C=RMAX
9       RETURN
C       USER FIX UP INDICATED. RECOMPUTE WITH NEW VALUE PLACED IN E
5       GO TO 100
        END
        SUBROUTINE FIXDIV(IRETCD,INO,A,B)
C       THIS IS A USER EXIT TO SERVE THE SUBROUTINE DIVIDE
C       THE PARAMETERS IN THE CALL MATCH THOSE USE IN THE CALL TO
C       ERRMON MADE BY SUBROUTINE DIVIDE
C     STEP1  IS ALTERNATE VALUE FOR B AVAILABLE -- MAIN PROGRAM
C       HAS SUPPLIED A NEW VALUE IN E.  IF E=0 NO NEW VALUE IS AVAILABLE
        COMMON E
        IF(E) 1,2,1
C       NEW VALUE AVAILABLE TAKE USER CORRECTION EXIT
1       B=E
        RETURN
C       NEW VALUE NOT AVAILABLE USE STANDARD FIX UP
2       IRETCD=0
        RETURN
        END
/*
//GO.SYSIN  DD *
             0.1E00        0.0E00
/*
```

Figure III-18. Sample Program Using Extended Error Handling Facility

The user-written exit routine can be written in FORTRAN or in assembler language. In either case, it must be able to accept the call to it as shown above. The user-exit routine must be a closed subroutine that returns control to the caller.

If the user-written exit routine is written in assembler language, the end of the parameter list can be checked. The high-order byte of the last parameter will have the hexadecimal value 80. If the routine is written in FORTRAN, the parameter list must match in length the parameter list passed in the CALL statement issued to the error monitor.

When the extended error handling facility encounters a condition or a request that requires user notification, an informative message is printed.

The error monitor is not recursive; if it has already been called for an error, it cannot be re-entered if the user-written corrective routine causes any of the error conditions that are listed in the option table. Boundary misalignment is therefore not allowed in a user-exit routine.

Actions the user may take if he wishes to correct an error are described in Tables III-4, III-5, and III-6.

OPTION TABLE CONSIDERATIONS

Figures III-19 and III-20 describe the fields of the option table and list the default values for the contents of these fields.

When a user-written exit subroutine is to be executed for a given error condition, the programmer must enter the address of the routine into the option table entry associated with that error condition.

Addresses for user-exit subroutines cannot be entered into option table entries during program installation. An installation may, however, construct an option table containing user-exit addresses and place that option table into the FORTRAN library. (Each address must be specified as a V-type address constant.) Use of this procedure, though, results in the inclusion, in the load module, of all such user-exit subroutines by the linkage editor.

If the user-exit address is not specified in advance through the use of V-type address constants, the programmer must issue a CALL ERRSET statement at execution time to insert an address into the option table that was created during program installation.

The programmer should be warned that altering an option table entry to allow "unlimited" error occurrence (specifying a number greater than 255) may cause a program to loop indefinitely.

CONSIDERATIONS FOR THE LIBRARY WITHOUT EXTENDED ERROR HANDLING FACILITY

When the extended error handling facility is not chosen, execution terminates after the first occurrence of an error, unless it is one caused by boundary misalignment, divide check, exponent underflow, or exponent overflow. The messages for errors 205, 215, 216, 218, 221-230, 228, and 238-301 are the same as those with the extended error handling facility. The other error messages are of the form "IHOxxxI" with no text.

Without the extended error handling facility, ERRMON becomes an entry point to the traceback routine. User programs that call the error monitor do not have to be altered. The error message will be printed with a traceback map and execution will terminate.

Note, that if the facility is not selected, the ERRTRA, ERRSET, ERRSAV, and ERRSTR subprograms are assumed to be user supplied if they are called in a FORTRAN program.

Field Contents	Field Length in Bytes	Default	Field Description
Number of entries	4	97	Number of entries in the Option Table.
Boundary alignment	1	40 (hexa-decimal)	Bit 1 indicates whether boundary alignment was chosen at program installation time. Bits 0 and 2 through 7 are reserved for future use. Bit 1: 0 indicates NOALIGN 1 indicates ALIGN
Extended error handling	1	FF (hexa-decimal)	Indicates whether extended error handling was chosen at program installation time. FF indicates that extended error handling was excluded. 00 indicates that extended error handling was included.
Alignment count	1	10	Maximum number of boundary alignment messages allowed when extended error handling is not chosen.
Reserved	1	0	Reserved for future use.

Figure III-19. Option Table Preface

Field Contents	Field Length in bytes	Default[1]	Field Description
Number of error occurrences allowed	1	10[2]	Number of times this error condition should be allowed to occur. When the value of the error count field (below) equals this value, job processing is terminated. Number may range from 0 to 255. A value of 0 means an unlimited number of occurrences.[3]
Number messages to print	1	5[4]	Number of times the corresponding error message is to be printed before message printing is suppressed. A value of 0 means no message is to be printed.
Error count	1	0	The number of times this error has occurred. A value of 0 indicates that no occurrences have been encountered.
Option bits	1	42 (hexa-decimal)	Eight option bits defined as follows (the default setting is underscored): Bit 0: 0 (underscored) — No control character supplied for overflow lines. Bit 0: 1 — Control character supplied to provide single spacing for overflow lines. Bit 1: 0 — Table entry cannot be modified.[5] Bit 1: 1 (underscored) — Table entry can be modified. Bit 2: 0 (underscored) — Fewer than 256 error have occurred. Bit 2: 1 — More than 256 errors have occurred. (Add 256 to error count field above to determine the number.) Bit 3[6]: 0 (underscored) — Do not print buffer contents with error message. Bit 3[6]: 1 — Print buffer contents. Bit 4: 0 (underscored) — Reserved. Bit 5: 0 (underscored) — Print messages default number of times only. Bit 5: 1 — Unlimited printing requested; print for every occurrence of error. Bit 6: 0 — Do not print traceback map. Bit 6: 1 (underscored) — Print traceback map. Bit 7: 0 (underscored) — Reserved.
User exit	4	1	Indicates where a user corrective routine is located. A value of 1 indicates that no user-written routine is available. A value other than 1 specifies the address of the user-written routine.

[1]The default values shown apply to all error numbers (including additional user entries) unless excepted by a footnote.
[2]Errors 208, 210, and 215 are set as unlimited, and errors 205, 217, 230, and 240 are set to 1.
[3]An unlimited number of errors may cause the FORTRAN job to loop indefinitely until the operator intervenes.
[4]Error 210 is set to 10, and errors 205, 217, 230, and 240 are set to 1.
[5]The entry for errors 205, 230, and 240 cannot be modified.
[6]The entry is set to 0 except for errors 212, 215, 218, 221, 222, 223, 224, and 225.

Figure III-20. Option Table Entry

Table III-5. Corrective Action After Mathematical Subroutine Error Occurrence (Part 1 of 4)

Error Code	FORTRAN Reference (See Note 1)	Invalid Argument Range	Options: Standard Corrective Action (See Notes 2 and 3)	Options: User-Supplied Corrective Action (See Note 4)
216	CALL SLITE (I)	I>4	The call is treated as a NO OP (an instruction that requests no action be performed)	I
216	CALL SLITET (I,J)	I>4	J=2	I
241	K=I**J	I=0, J≤0	K=0	I,J
242	Y=X**I	X=0, I≤0	If I=0, Y=1 If I<0, Y=•	X,I
243	DA=D**I	D=0, I≤0	If I=0, Y=1 If I<0, Y=•	D,I
244	XA=X**Y	X=0, Y≤0	XA=0	X,Y
245	DA=D**DB	D=0, DB≤0	DA=0	D,DB
246	CA=C**I	C=0+0i, I≤0	If I=0, C=1+0 If I<0, C=•+0i	C,I
247	CDA=CD*I	C=0+0i, I≤0	If I=0, C=1+0 If I<0, C=•+0i	CD,I
251	Y=SQRT (X)	X<0	$Y=\|X\|^{1/2}$	X
252	Y=EXP (X)	X>174.673	Y=•	X
253	Y=ALOG (X)	X=0	Y=-•	X
		X<0	Y=log|X|	X
	Y=ALOG10 (X)	X=0	Y=-•	X
		X<0	$Y=\log_{10}\|X\|$	X
254	Y=COS (X) Y=SIN (X)	$\|X\|\geq 2^{18}*\pi$	$Y=\sqrt{2/2}$	X
255	Y=ATAN2 (X,XA)	X=0, XA=0	Y=0	X,XA

Notes:

1. The variable types are as follows:

Variable	Type
I,J	INTEGER*4
X,XA,Y	REAL*4
D,DA,DB	REAL*8
C,CA	COMPLEX*8
Z,X_1,X_2	Complex variables to be given the length of the functioned argument when they appear.
CD	COMPLEX*16

2. The largest number that can be represented in floating point is indicated by the symbol •.
3. The value e=approximately 2.7183.
4. The user-supplied answer is obtained by recomputation of the function using the value set by the user routine for the parameters listed.

Table III-5. Corrective Action After Mathematical Subroutine Error Occurrence (Part 2 of 4)

Error Code	FORTRAN Reference (See Note 1)	Invalid Argument Range	Options: Standard Corrective Action (See Notes 2 and 3)	Options: User-Supplied Corrective Action (See Note 4)
256	Y=SINH (X) Y=COSH (X)	|X|≤175.366	Y=(SIN X)• Y=•	X
257	Y=ARSIN (X) Y=ARCOS (X)	|X|>1	If X>1.0, ARSIN(X)=$\frac{\pi}{2}$ If X<-1.0, ARSIN(X)=-$\frac{\pi}{2}$ If X>1.0, ARCOS=0 If X<-1.0, ARCOS=π	X
258	Y=TAN (X) Y=COTAN (X)	|X|≥$(2^{18})*\pi$	Y=1	X
259	Y=TAN (X)	X is too close to an odd multiple of $\frac{\pi}{2}$	Y=•	X
	Y=COTAN (X)	X is too close to a multiple of π	Y=•	X
261	DA=DSQRT (D)	D<0	DA=$\|D\|^{1/2}$	D
262	DA=DEXP (D)	D>174.673	DA=•	D
263	DA=DLOG (D)	D=0 D<0	DA=-• DA=log|X|	D
	DA=DLOG10 (D)	D=0 D<0	DA=-• DA=$\log_{10}$ |X|	D
264	DA=DSIN (D) DA=DCOS (D)	|D|≥$2^{50}*\pi$	DA=$\sqrt{2}/2$	D
265	DA=DATAN2(D,DB)	D=0, DB=0	DA=0	D, DB
266	DA=DSINH (D) DA=DCOSH (D)	|D|≥175.366	DA=(SIN X)• DA=•	D
267	DA=DARSIN (D) DA=DARCOS (D)	|D|>1	If X>1.0, DARSIN(X)=$\frac{\pi}{2}$ If X<-1.0, DARSIN(X)=-$\frac{\pi}{2}$ If X>1.0, DARCOS=0 If X<-1.0, DARCOS=π	D

Notes:

1. The variable types are as follows:

Variable	Type
I, J	INTEGER*4
X, XA, Y	REAL*4
D, DA, DB	REAL*8
C, CA	COMPLEX*8
Z, X_1, X_2	Complex variables to be given the length of the functioned argument when they appear.
CD	COMPLEX*16

2. The largest number that can be represented in floating point is indicated by the symbol •.
3. The value e=approximately 2.7183
4. The user-supplied answer is obtained by recomputation of the function using the value set by the user routine for the parameters listed.

Table III-5. Corrective Action After Mathematical Subroutine Error Occurrence (Part 3 of 4)

Error Code	FORTRAN Reference (See Note 1)	Invalid Argument Range	Options: Standard Corrective Action (See Notes 2 and 3)	Options: User-Supplied Corrective Action (See Note 4)
268	DA=DTAN (D) DA=DCOTAN (D)	$\|X\| \geq 2^{50} * \pi$	DA=1	D
269	DA=DTAN (D)	D is too close to an odd multiple of $\frac{\pi}{2}$	DA=•	D
	DA=DCOTAN (D)	D is too close to a multiple of π	DA=•	D
For errors 271 through 275, $C=X_1+iX_2$				
271	Z=CEXP (C)	$X_1 > 174.673$	Z=*(COS X_2+ SIN X_2)	C
272	Z=CEXP (C)	$\|X_2\| \geq 2^{18} * \pi$	$Z=e^{X_1}+0*i$	C
273	Z=CLOG (C)	C=0+0i	Z=-•+0i	C
274	Z=CSIN (C) Z=CCOS (C)	$\|X_1\| \leq 2^{18} * \pi$	Z=0+SINH(X_2)*i Z=COSH(X_2)+0*i	C
275	Z=CSIN (C)	$X_2 > 174.673$	$Z=\frac{•}{2}$(SIN X_1+iCOS X_1)	C
	Z=CCOS (C)		$Z=\frac{•}{2}$(COS X_1-iSIN X_1)	C
	Z=CSIN (C)	$X_2 < -174.673$	$Z=\frac{•}{2}$(SIN X_1-iCOS X_1)	C
	Z=CCOS (C)		$Z=\frac{•}{2}$(COS X_1+iSIN X_1)	C

Notes:

1. The variable types are as follows:

Variable	Type
I,J	INTEGER*4
X,XA,Y	REAL*4
D,DA,DB	REAL*8
C,CA	COMPLEX*8
Z,X_1,X_2	Complex variables to be given the length of the functioned argument when they appear.
CD	COMPLEX*16

2. The largest number that can be represented in floating point is indicated by the symbol •.
3. The value e=approximately 2.7183
4. The user-supplied answer is obtained by recomputation of the function using the value set by the user routine for the parameters listed.

Table III-5. Corrective Action After Mathematical Subroutine Error Occurrence (Part 4 of 4)

Error Code	FORTRAN Reference (See Note 1)	Invalid Argument Range	Options: Standard Corrective Action (See Notes 2 and 3)	Options: User-Supplied Corrective Action (See Note 4)
For errors 281 through 285, CD=X_1+iX_2				
281	Z=CDEXP (CD)	X_1>174.673	Z=*(COS X_2+iSIN X_2)	CD
282	Z=CDEXP (CD)	|X_2|≥2^{50}*π	Z=e^{X_1}+0*i	CD
283	Z=CDLOG (CD)	CD=0+0i	Z=-•+0i	CD
284	Z=CDSIN (CD) Z=CDCOS (CD)	|X_1|≥2^{50}*π	Z= 0+SINH(X_2)*i Z= COSH(X_2)+0*i	CD
284	Z=CDSIN (CD) Z=CDCOS (CD)	|X_1|≥2^{50}*π	Z=0+0i	CD
285	Z=CDSIN (CD)	X_2>174.673	Z=$\frac{•}{2}$(SIN X_1+iCOS X_1)	CD
	Z=CDCOS (CD)		Z=$\frac{•}{2}$(COS X_1-iSIN X_1)	CD
	Z=CDSIN (CD)	X_2.-174.673	Z=$\frac{•}{2}$(SIN X_1-iCOS X_1)	CD
	Z=CDCOS (CD)		Z=$\frac{•}{2}$(COS X_1+iSIN X_1)	CD
290	Y=GAMMA (X)	X≤2^{-252} or X≥57.5744	Y=•	X
291	Y=ALGAMA (X)	X≤0 or X≥4.2937*10^{73}	Y=•	X
300	DA=DGAMMA (D)	D≤2^{-252} or D≥57.5774	DA=•	D
301	DA=DLGAMA (D)	D≤0 or D≥4.2937*10^{73}	DA=•	D

Notes:

1. The variable types are as follows:

Variable	Type
I,J	INTEGER*4
X,XA,Y	REAL*4
D,DA,DB	REAL*8
C,CA	COMPLEX*8
Z,X_1,X_2	Complex variables to be given the length of the functioned argument when they appear.
CD	COMPLEX*16

2. The largest number that can be represented in floating point is indicated by the symbol •.
3. The value e= approximately 2.7183
4. The user-supplied answer is obtained by recomputation of the function using the value set by the user routine for the parameters listed.

Table III-6. Corrective Action After Program Interrupt Occurrence

Program Interrupt Messages			Options	
Error Code	Parameters Passed to User Exit (Note 1)	Reason for Interrupt (Note 2)	Standard Corrective Action	User-Supplied Corrective Action
207	D,I	Exponent overflow (Interrupt Code 12)	Result register set to the largest possible floating point number. The sign of the result register is not altered.	User may alter D. (Note 3)
208	D,I	Exponent underflow (Interrupt Code 13)	The result register is set to zero.	User may alter D. (Note 3)
209	None	Divide check, integer divide (interrupt code 9), decimal divide (Interrupt Code 11), floating point Code 11), floating point divide (interrupt code 15). See Note 4.	For floating point divide, where n/0 and n=0, result register is set to 0; where n≠0, result register set to largest possible floating point number. No standard fixup for other interrupts.	See Note 5.
210	None	Specification interrupt (interrupt Code 6) occurs for boundary misalignment. Operation exception (interrupt code 1) occurs for operation interrupt. Other interrupts occur during boundary alignment adjustment or extended precision floating point simulation. They will be shown with this error code and the PSW portion of the message will identify the interrupt.	No special corrective action other than correcting boundary misalignments.	See Note 5.

Notes:

1. The variable types and meaning are as follows:

Variable	Type	Meaning
D	REAL*8	This variable contains the contents of the result register after the interrupt.
I	INTEGER*4	The variable contains the "exponent" as an integer value for the number in D. It may be used to determine the amount of the underflow or overflow. The value in I is not the true exponent, but what was left in the exponent field of a floating point number after the interrupt.

2. A program interrupt asynchronously. Interrupts are described in the appropriate principles of operation publication, as listed in the Preface.
3. The user exit routine may supply an alternate answer for the setting of the result register. This is accomplished by placing a value for D in the user-exit routine. Although the interrupt may be caused by a long or short floating-point operation, the user-exit routine need not be concerned with this. The user-exit routine should always set a REAL*8 variable and the FORTRAN library will load a short or long data item depending upon the floating-point operation that caused the interrupt.
4. For floating-point divide check, the contents of the result register is shown in the message.
5. The user-exit routine does not have the ability to change result registers after a fixed-point divide check. The boundary alignment adjustments are informative messages, and there is nothing to alter before execution continues.

APPENDIXES

The following examples show several methods of processing load modules.

Example 1: Submitting a Job Consisting of One Job Step

Problem Statement: A previously created data set, SCIENCE.MATH.MATRICES, contains a set of 80 matrices. Each matrix is an array containing REAL*4 variables. The size of the matrices varies from 2x2 to 25x25; the average size is 10x10. The matrices are inverted by a load module MATINV in the library MATPROGS. Each inverted matrix is written (assume FORMAT control) as a single record on the data set SCIENCE.MATH.INVMATRS. The first variable in each record denotes the size of the matrix.

The input/output flow for the example is shown in Figure A-1. The job control statements used to define this job are shown in Figure A-2.

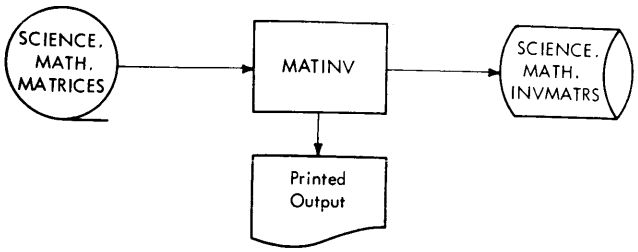


Figure A-1. Input/Output Flow for Example 1

Explanation: The JOB statement identifies the programmer as JOHN SMITH and supplies the account number 537. Control statements and control statement error messages are written in the SYSOUT data set.

The JOBLIB DD statement indicates that the private library MATPROGS is to be concatenated with the system library.

The EXEC statement indicates that the load module MATINV is the program to be executed.

DD statement FT08F001 identifies the input data set, SCIENCE.MATH.MATRICES. (Data set reference number 8 is used to read the input data set.) Assume that this data set has been previously created and cataloged; therefore no information other than the data set name and disposition has to be supplied.

DD statement FT10F001 identifies the printed output. (Data set reference number 10 is used for printed output.)

DD statement FT04F001 defines the output data set. (Data set reference number 4 is used to write the data set containing the inverted matrices.) Because the data set is to be created and cataloged in this job step, a complete data set specification is supplied. The DSNAME parameter indicates that the data set is named SCIENCE.MATH.INVMATRS. The DISP parameter indicates that the data set is new and is to be cataloged. SPACE indicates that space is to be reserved for 80 records, 408 characters long (80 matrices of average size); when space is exhausted, space for 9 more records is allocated. The space is contiguous; any unused space is released, and allocation is to begin and end on cylinder boundaries.

DCB indicates that records are variable-length (because the size of matrices vary). The record length is specified as 2504, the maximum size of a variable-length record. The maximum size of a record in this data set is the maximum number of elements (625) in any matrix multiplied by the number of bytes (4) allocated for an element, plus 4 for the segment control word (SCW). The buffer length is specified as 2508 (the 4 extra bytes are for the block control word (BCW) that contains the length of the block).

SEP indicates that read and write operations are to take place on different channels.

Sample Coding Form

```
1-10      11-20     21-30     31-40     41-50     51-60     61-70     71-80
//INVERT JOB 537,JOHNSMITH,MSGLEVEL=1
//JOBLIB DD DSNAME=MATPROGS,DISP=OLD
//INVERT EXEC PGM=MATINV
//FT08F001 DD DSNAME=SCIENCE.MATH.MATRICES,DISP=OLD
//FT10F001 DD SYSOUT=A
//FT04F001 DD DSNAME=SCIENCE.MATH.INVMATRS,                              1
//            DISP=(NEW,CATLG),UNIT=DACLASS,VOLUME=SER=1089W,            2
//            SPACE=(408,(80,9),RLSE,CONTIG,ROUND),SEP=FT08F001,         3
//            DCB=(RECFM=VB,LRECL=2504,BLKSIZE=2508
```

Figure A-2. Job Control Statements for Example 1

Example 2: Submitting a Job Consisting of Multiple Job Steps

Problem Statement: Raw data gathered from a rocket test firing is to be converted in a report and graphs describing the success of the test. The data set RAWDATA contains the information gathered from the test firing.

Job step 1 executes the load module PROGRD. PROGRD compares the raw data in RAWDATA against forecasted results contained in the data set PROJDATA, and generates the data set &REFDATA containing refined data to be passed to the second job step.

Job step 2 executes the load module ANALYZ. ANALYZ processes &REFDATA against the data set PARAMS which contains parameters used in developing values, and generates the data set &VALUES containing results to be passed to the third job step.

Job step 3 executes the load module REPORT. REPORT prints the values in &VALUES as a series of reports and graphs.

Figure A-3 shows the I/O flow for this example. Figure A-4 shows the job control statements for the job. The load modules PROGRD, ANALYZ, and REPORT are contained in the private library FIRING.

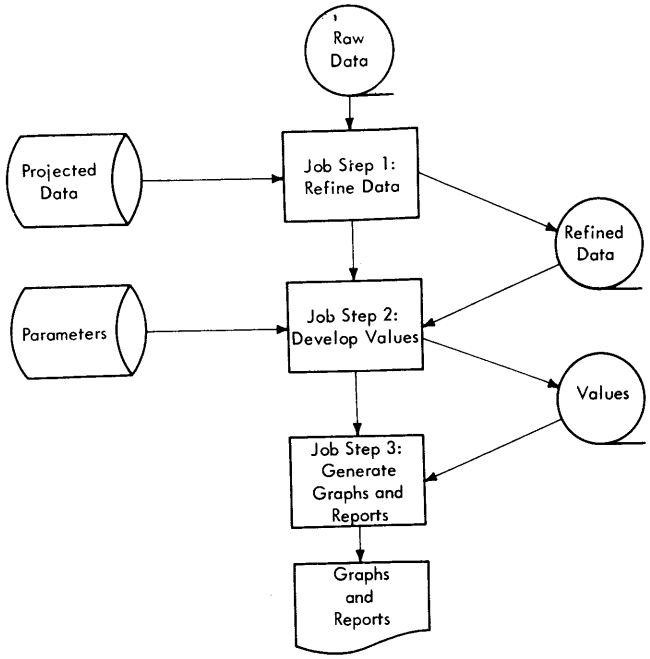


Figure A-3. Input Flow for Example 2

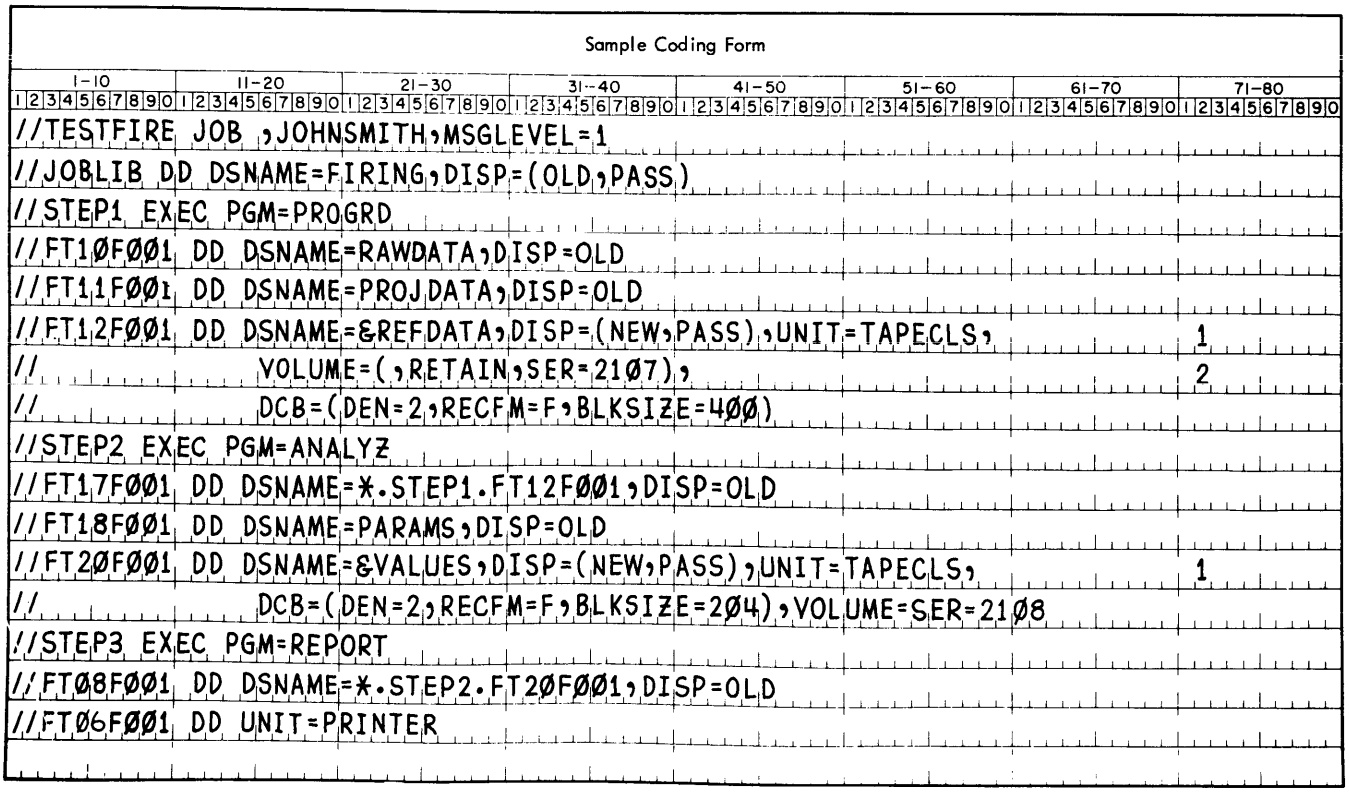


Figure A-4. Job Control Statements for Example 2

Explanation of Job Control Statements: In Figure A-4, the JOB statement indicates the programmer's name, JOHN SMITH, and specifies that control statements and control statement error messages are to be written in the SYSOUT data set. Because the first positional parameter indicating accounting information is omitted, its absence is noted by a comma.

The JOBLIB DD statement indicates that the private library FIRING is to be concatenated with the system library.

The EXEC statement STEP1 indicates that the load module PROGRD is to be executed.

DD statement FT10F001 and FT11F001 identify the data sets containing raw data (RAWDATA) and the forecasted results (PROJDATA) respectively.

DD statement FT12F001 defines the temporary data set &&REFDATA (data set reference number 12 is used to write &&REFDATA). DISP indicates that the data set is new and is to be passed to the next job step. UNIT indicates that the data set is to be written on one of the units contained in the device class TAPECLS. VOLUME indicates that the volume whose serial number is 2107 is to be used to contain the data set. DCB indicates that the volume is written in high density and that records are fixed-length with FORMAT control and a buffer length of 400.

The EXEC statement STEP2 indicates that the load module ANALYZ is to be executed.

DD statement FT17F001 identifies the data set containing refined data. It specifies the data set as the one defined on DD statement FT12F001 appearing in job step STEP1. DISP indicates that the data set is to be deleted after execution of this job step.

DD statement FT18F001 identifies a previously created and cataloged data set PARAMS.

DD statement FT20F001 defines the temporary data set &VALUES containing the

values to be printed. DISP indicates that the data set is new and is to be passed to the next job step. VOLUME and UNIT indicate that the data set is to be written on volume 2108 using a device in the device class TAPECLS. DCB indicates high density and fixed-length records (written under FORMAT control) with a buffer length of 204.

The EXEC statement STEP3 indicates that the load module REPORT is to be executed.

DD statement FT08F001 identifies the data set containing the values to be printed.

DD statement FT06F001 indicates that the device class PRINTER is to be used to print the reports (data set reference number 6 is used to write the data set).

Example 3: Updating a Direct-Access Data Set

A data set has been created that contains master records for an index of stars. Each star is identified by a unique 6-digit star identification number. Each star is assigned a record position in the data set by truncating the last two digits in the star identification number. Because synonyms arise, records are chained.

Problem Statement: Figure A-5 shows a block diagram illustrating the logic for this problem.

A card data set read from the input stream is used to update the star master data set. Each detail record in this data set contains:

1. The star identification field of the star master record that the detail record is to update.
2. Six variables that are to be used to update the star master record.

The following conventions are observed in processing this data set:

1. The star master record that contains the record location counter pointing to space reserved for chained records is assigned to record location 1.
2. A zero in the chain variable indicates that the end of a chain has been reached.

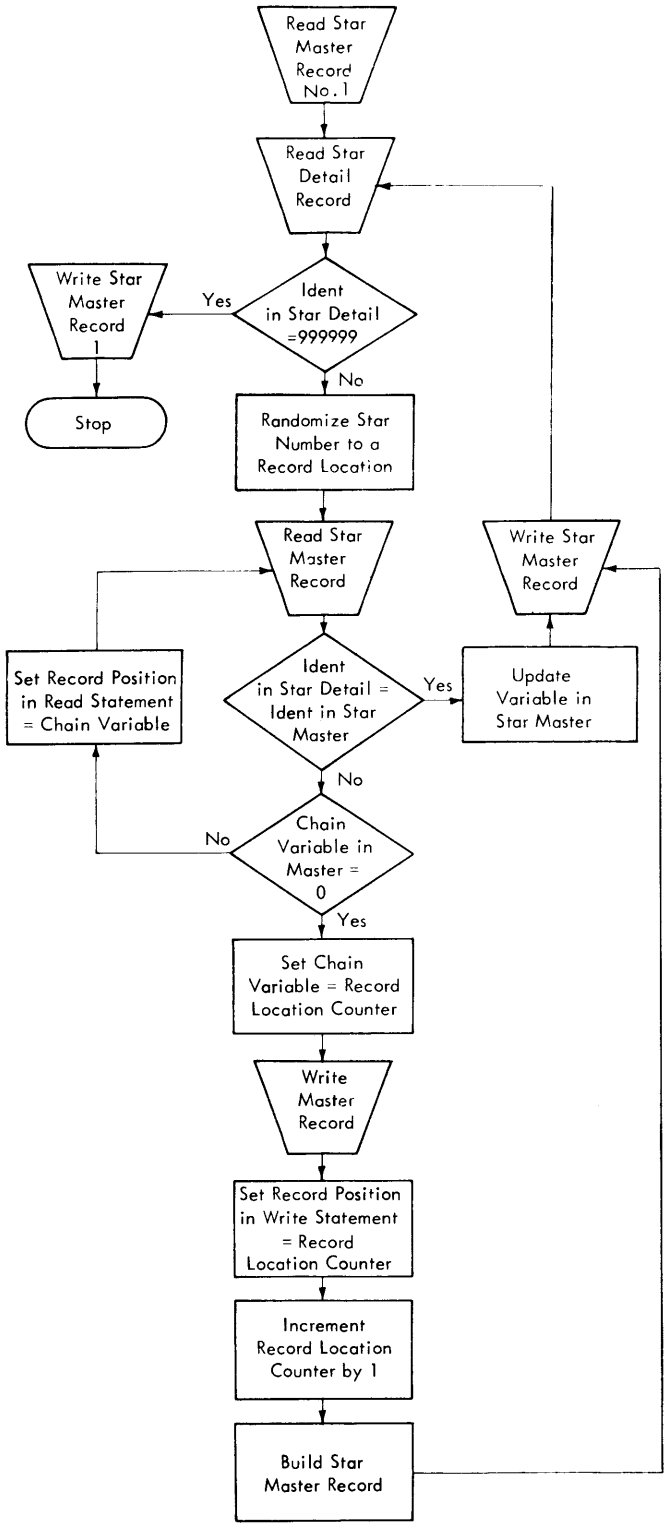


Figure A-5. Block Diagram for Example 3

3. The first variable in each star master record is the star identification field; the second variable in each star master is the chain variable pointing to the next record in the chain.

4. Each record contains six other variables that contain information about that star.

When a detail record is read, its identification field is randomized and the appropriate star master record is read. If the correct star master record is found, the record is to be updated. If a star master is not found, then a star master record is to be created for that star.

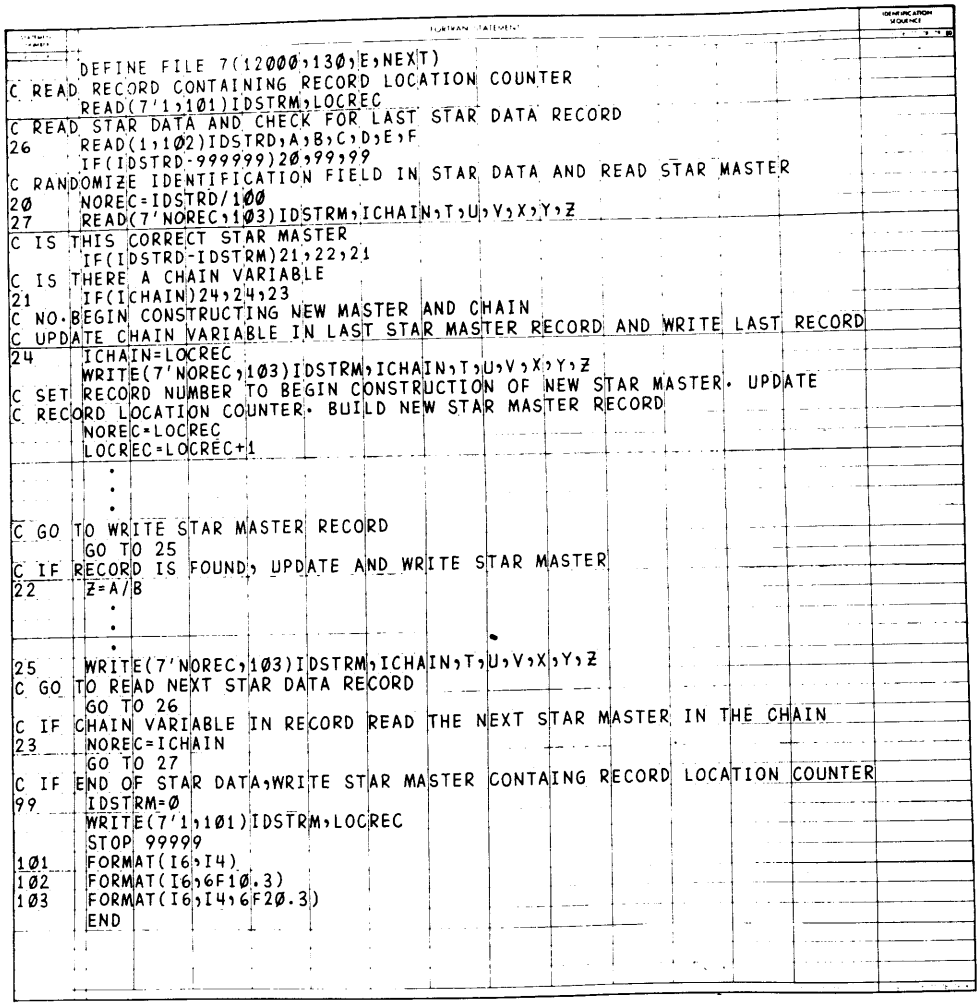
The last record in the star detail data set contains star identification number 999999 to indicate the end of the data set.

Explanation: Figure A-5 is similar to the diagram shown in Figure III-2 except Figure A-5 includes blocks that describe updating variables in master records already present in the data set. (Figure III-2 includes blocks describing certain operations that must be performed when a direct access data set is created.) Also, Figure A-5 is adapted to Example 3, while Figure III-2 is more general. Figure A-6 shows the FORTRAN coding for this program.

The star master record that contains the record counter is read, placing the record location counter in LOCREC. Whenever a detail record is read, the identification variable is checked to determine if the end of the detail data set has been reached. The star detail records contain the variables A, B, C, D, E, and F.

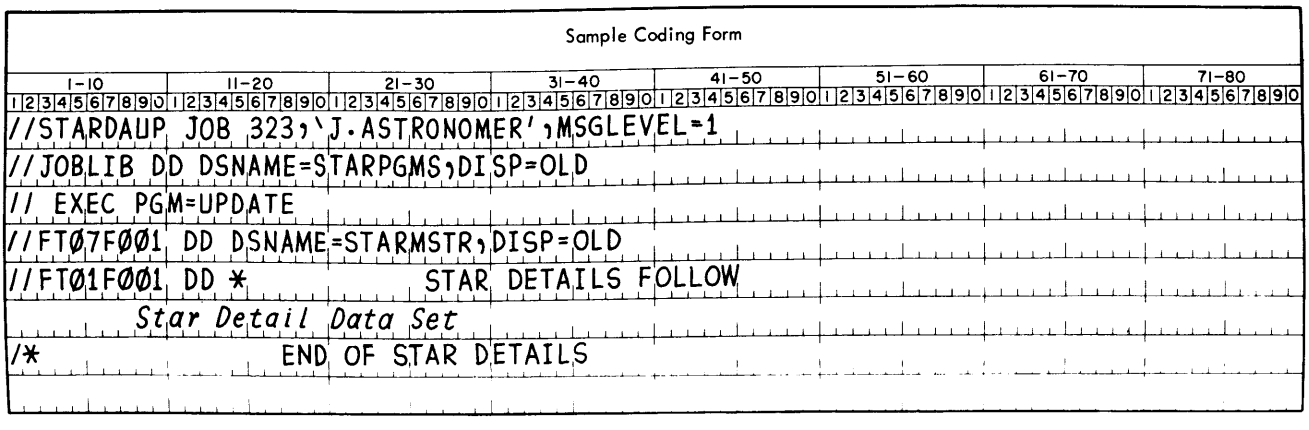
The identification number in the detail record is randomized and the result is placed in the variable NOREC, which is used to read a master record. The master record contains the star identification number (IDSTRM), a chain record location (ICHAIN), and six variables (T, U, V, X, Y, and Z) which are to be updated by the variables in the star detail records. IDSTRM and IDSTRD are compared to see if the correct star master is found. If it is not found, then the variables containing the chain record numbers are followed until the correct star master is found or a new star master is created.

Job Control Statements: The program shown in Figure A-6 is compiled and link edited, placing the load module in the data set STARPGMS and assigning the load module the name UPDATE. The data set that contains the star master records was cataloged and assigned the name STARMSTR when it was created. Figure A-7 shows the job control statements needed to execute the module UPDATE.



```
      DEFINE FILE 7(12000,130,E,NEXT)
C READ RECORD CONTAINING RECORD LOCATION COUNTER
      READ(7'1,101)IDSTRM,LOCREC
C READ STAR DATA AND CHECK FOR LAST STAR DATA RECORD
26    READ(1,102)IDSTRD,A,B,C,D,E,F
      IF(IDSTRD-999999)20,99,99
C RANDOMIZE IDENTIFICATION FIELD IN STAR DATA AND READ STAR MASTER
20    NOREC=IDSTRD/100
27    READ(7'NOREC,103)IDSTRM,ICHAIN,T,U,V,X,Y,Z
C IS THIS CORRECT STAR MASTER
      IF(IDSTRD-IDSTRM)21,22,21
C IS THERE A CHAIN VARIABLE
21    IF(ICHAIN)24,24,23
C NO. BEGIN CONSTRUCTING NEW MASTER AND CHAIN
C UPDATE CHAIN VARIABLE IN LAST STAR MASTER RECORD AND WRITE LAST RECORD
24    ICHAIN=LOCREC
      WRITE(7'NOREC,103)IDSTRM,ICHAIN,T,U,V,X,Y,Z
C SET RECORD NUMBER TO BEGIN CONSTRUCTION OF NEW STAR MASTER. UPDATE
C RECORD LOCATION COUNTER. BUILD NEW STAR MASTER RECORD
      NOREC=LOCREC
      LOCREC=LOCREC+1
        .
        .
        .
C GO TO WRITE STAR MASTER RECORD
      GO TO 25
C IF RECORD IS FOUND, UPDATE AND WRITE STAR MASTER
22    Z=A/B
        .
        .
        .
25    WRITE(7'NOREC,103)IDSTRM,ICHAIN,T,U,V,X,Y,Z
C GO TO READ NEXT STAR DATA RECORD
      GO TO 26
C IF CHAIN VARIABLE IN RECORD READ THE NEXT STAR MASTER IN THE CHAIN
23    NOREC=ICHAIN
      GO TO 27
C IF END OF STAR DATA,WRITE STAR MASTER CONTAING RECORD LOCATION COUNTER
99    IDSTRM=0
      WRITE(7'1,101)IDSTRM,LOCREC
      STOP 99999
101   FORMAT(I6,I4)
102   FORMAT(I6,6F10.3)
103   FORMAT(I6,I4,6F20.3)
      END
```

Figure A-6. FORTRAN Coding for Example 3



Sample Coding Form

```
1-10      11-20     21-30     31-40     41-50     51-60     61-70     71-80
1234567890123456789012345678901234567890123456789012345678901234567890
//STARDAUP JOB 323,'J.ASTRONOMER',MSGLEVEL=1
//JOBLIB DD DSNAME=STARPGMS,DISP=OLD
// EXEC PGM=UPDATE
//FT07F001 DD DSNAME=STARMSTR,DISP=OLD
//FT01F001 DD *              STAR DETAILS FOLLOW
        Star Detail Data Set
/*                 END OF STAR DETAILS
```

Figure A-7. Job Control Statements for Example 3

A FORTRAN programmer can use assembler language subprograms with his FORTRAN main program. This section describes the linkage conventions that must be used by the assembler language subprogram to communicate with the FORTRAN main program. To understand this appendix, the reader should be familiar with the appropriate assembler language publication and assembler programmer's guide, as listed in the Preface.

SUBROUTINE REFERENCES

The FORTRAN programmer can refer to a subprogram in two ways: by a CALL statement or a function reference within an arithmetic expression. For example, the statements:

```
CALL MYSUB(X,Y,Z)
I=J+K+MYFUNC(L,M,N)
```

refer to a subroutine subprogram MYSUB and a function subrpogram MYFUNC, respectively.

For subprogram reference, the compiler generates:

1. A contiguous argument list; the addresses of the arguments are placed in this list to make the arguments accessible to the subprogram.
2. A save area in which the subprogram can save information related to the calling program.
3. A calling sequence to pass control to the subprogram, using linkage conventions (Table A-1 illustrates the use of registers during linkage).

Argument List

The argument list contains addresses of variables, arrays, and subprogram names used as arguments. Each entry in the argument list is four bytes and is aligned on a fullword boundary. The last three bytes of each entry contain the 24-bit address of an argument. The first byte of each entry contains zeros, unless it is the last entry in the argument list. If this is the last entry, the sign bit in the entry is set to 1.

The address of the argument list is placed in general register 1 by the calling program.

Save Area

The calling program contains a save area in which the subprogram places information, such as the entry point for this program, an address to which the subprogram returns, general register contents, and addresses of save areas used by programs other than the subprogram. The amount of storage reserved by the calling program is 18 words. Figure A-8 shows the layout of the save area and the contents of each word. The address of the save area is placed in general register 13.

The called subprogram does not have to save and restore floating-point registers.

Calling Sequence

A calling sequence is generated to transfer control to the subprogram. The address of the save area in the calling program is placed in general register 13. The address of the argument list is placed in general register 1, and the entry address is placed in general register 15. If there is no argument list, then general register 1 will contain zero. A branch is made to the address in register 15 and the return address is saved in general register 14.

Table A-1. Linkage Registers

Register Number	Register Name	Function
0	Result Register	Used for function subprograms only. The result is returned in general or floating-point register 0. An extended precision result is returned in floating-point registers 0 and 2. A complex result is returned in floating-point registers 0 (real part) and 2 (imaginary part). Note: For subroutine subprograms, the result(s) is returned in a variable(s) passed by the programmer.
1	Argument List Register	Address of the argument list passed to the called subprogram.
2	Result Register	See Function of Register 0.
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. Note: Register 15 is also used as a condition code register, a RETURN code register, and a STOP code register. The particular values that can be contained in the register are: 16 - a terminal error was detected during execution of a subprogram (an IHOxxxI message is generated) 4*i - a RETURN i statement was executed n - a STOP n statement was executed 0 - a RETURN or a STOP statement was executed

AREA (word 1)	This word is used by a FORTRAN-compiled routine to store its epilogue address and may not be used by the assembler language subprogram for any purpose.
AREA+4 (word 2)	If the program that calls the assembler language subprogram is itself a subprogram, this word contains the address of the save area of the calling program; otherwise, this word is not used.
AREA+8 (word 3)	The address of the save area of the called subprogram.
AREA+12 (word 4)	The contents of register 14 (the return address). When the subprogram returns control, the first byte of this location is set to ones.
AREA+16 (word 5)	The contents of register 15 (the entry address).
AREA+20 (word 6)	The contents of register 0.
AREA+24 (word 7)	The contents of register 1.
.	.
.	.
AREA+68 (word 18)	The contents of register 12.

Figure A-8. Save Area Layout and Word Contents

CODING THE ASSEMBLER LANGUAGE SUBPROGRAM

Two types of assembler language subprograms are possible: the first type (lowest level) assembler subprogram does not call another subprogram; the second type (higher level) subprogram does call another subprogram.

Coding a Lowest Level Assembler Language Subprogram

For the lowest level assembler language subprogram, the linkage instructions must include:

1. An assembler instruction that names an entry point for the subprogram.
2. An instruction(s) to save any general registers used by the subprogram in the save area reserved by the calling program. (The contents of linkage registers 0 and 1 need not be saved.)
3. An instruction(s) to restore the "saved" registers before returning control to the calling program.
4. An instruction that sets the first byte in the fourth word of the save area to ones, indicating that control is returned to the calling program.
5. An instruction that returns control to the calling program.

Figure A-9 shows the linkage conventions for an assembler language subprogram that does not call another subprogram. In addition to these conventions, the assembler program must provide a method to transfer arguments from the calling program and return the arguments to the calling program.

Higher Level Assembler Language Subprogram

A higher level assembler subprogram must include the same linkage instructions as the lowest level subprogram, but because the higher level subprogram calls another subprogram, it must simulate a FORTRAN subprogram reference statement and include:

1. A save area and additional instructions to insert entries into its save area.
2. A calling sequence and a parameter list for the subprogram that the higher level subprogram calls.
3. An assembler instruction that indicates an external reference to the subprogram called by the higher level subprogram.
4. Additional instructions in the return routine to retrieve entries in the save area.

Note: If an assembler language main program calls a FORTRAN subprogram, the following instructions must be included in the assembler language program before the FORTRAN subprogram is called:

```
L 15,=V(IBCOM#)
BAL 14,64(15)
```

```
Name      Oper.   Operand        Comments
deckname  START   0
          BC      15,m+1+4(15)   BRANCH AROUND CONSTANTS IN CALLING SEQUENCE
          DC      X'm'           m MUST BE AN ODD INTEGER TO INSURE THAT THE PROGRAM
          DC      CLm'name'      STARTS ON A HALFWORD BOUNDARY.  THE NAME CAN BE PADDED
*                                WITH BLANKS.
          STM     14,R,12(13)    THE CONTENTS OF REGISTERS 14, 15, AND 0 THROUGH R ARE
*                                STORED IN THE SAVE AREA OF THE CALLING PROGRAM. R IS ANY
*                                NUMBER FROM 2 THROUGH 12.
          BALR    B,0            ESTABLISH BASE REGISTER (2≤B≤12)
          USING   *,B
                  (user-written source statements)
                    .
                    .
                    .
          LM      2,R,28(13)     RESTORE REGISTERS
          MVI     12(13),X'FF'   INDICATE CONTROL RETURNED TO CALLING PROGRAM
          BCR     15,14          RETURN TO CALLING PROGRAM
```

Figure A-9. Linkage Conventions for Lowest Level Subprogram

These instructions cause initialization of return coding, interruption exceptions and opening of the error message data set. If this is not done and the FORTRAN subprogram terminates either with a STOP statement or because of an execution-time error, the data sets opened by FORTRAN are not closed and the result of the termination cannot be predicted. Register 13 must contain the address of the save area that contains the registers to be restored upon termination of the FORTRAN subprogram. If control is to return to the assembler language subprogram, then register 13 contains the address of its save area. If control is to return to the operating system, then register 13 contains the address of its save area.

Figure A-10 shows the linkage conventions for an assembler subprogram that calls another assembler subprogram.

```
|Name     |Oper. |Operand       Comments
|---------|------|-------------------------------------------------------------------
|deckname |START |0
|         |EXTRN |name2         NAME OF THE SUBPROGRAM CALLED BY THIS SUBPROGRAM
|         |BC    |15,m+1+4(15)
|         |DC    |X'm'
|         |DC    |CLm'name1'
|*        |      |SAVE ROUTINE
|         |STM   |14,R,12(13)   THE CONTENTS OF REGISTERS 14, 15, AND 0 THROUGH R ARE
|*        |      |              STORED IN THE SAVE AREA OF THE CALLING PROGRAM. R IS
|*        |      |              ANY NUMBER FROM 2 THROUGH 12.
|         |BALR  |B,0           ESTABLISH BASE REGISTER
|         |USING |*,B
|         |LR    |Q,13          LOADS REGISTER 13, WHICH POINTS TO THE SAVE AREA OF THE
|*        |      |              CALLING PROGRAM, INTO ANY GENERAL REGISTER, Q, EXCEPT 0,
|*        |      |              11, 13, AND 15.
|         |LA    |13,AREA       LOADS THE ADDRESS OF THIS PROGRAM'S SAVE AREA INTO
|*        |      |              REGISTER 13.
|         |ST    |13,8(0,Q)     STORES THE ADDRESS OF THIS PROGRAM'S SAVE AREA INTO THE
|*        |      |              CALLING PROGRAM'S SAVE AREA
|         |ST    |Q,4(0,13)     STORES THE ADDRESS OF THE PREVIOUS SAVE AREA (THE SAVE
|*        |      |              AREA OF THE CALLING PROGRAM) INTO WORD 2 OF THIS PRO-
|*        |      |              GRAM'S SAVE AREA
|         |BC    |15,prob1
|AREA     |DS    |18F           RESERVES 18 WORDS FOR THE SAVE AREA
|*        |      |END OF SAVE ROUTINE
|prob1    |      |(user-written program statements)
|         |      |
|*        |      |CALLING SEQUENCE
|         |LA    |1,ARGLIST     LOAD ADDRESS OF ARGUMENT LIST
|         |L     |15,ADCON
|         |BALR  |14,15
|         |      |(more user-written program statements)
|*        |      |RETURN ROUTINE
|         |L     |13,AREA+4     LOADS THE ADDRESS OF THE PREVIOUS SAVE AREA BACK INTO
|*        |      |              REGISTER 13
|         |LM    |2,R,28(13)
|         |L     |14,12(13)     LOADS THE RETURN ADDRESS INTO REGISTER 14.
|         |MVI   |12(13),X'FF'
|         |BCR   |15,14         RETURN TO CALLING PROGRAM
|*        |      |END OF RETURN ROUTINE
|ADCON    |DC    |A(name2)
|*        |      |ARGUMENT LIST
|ARGLIST  |DC    |AL4(arg1)     ADDRESS OF FIRST ARGUMENT
|         | .    |
|         | .    |
|         | .    |
|         |DC    |X'80'         INDICATE LAST ARGUMENT IN ARGUMENT LIST
|         |DC    |AL3(argn)     ADDRESS OF LAST ARGUMENT
```

Figure A-10. Linkage Conventions for Higher Level Subprogram

In-Line Argument List

The assembler programmer may establish an in-line argument list instead of out-of-line list. In this case, he may substitute the calling sequence and argument list shown in Figure A-11 for that shown in Figure A-10.

```
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 A-11. In-Line Argument List

Sharing Data in COMMON

Both named and blank COMMON in a FORTRAN IV program can be referred to by an assembler language subprogram. To refer to named COMMON, the V-type address constant

name DC V(name of COMMON)

is used.

If a FORTRAN program has a blank COMMON area and blank COMMON is also defined (by the COM instruction) in an assembler language subprogram, only one blank COMMON area is generated for the output load module. Data in this blank COMMON is accessible to both programs.

To refer to blank COMMON, the following linkage may be specified:

```
     COM
name DS 0F
     .                    .
     .                    .
     .                    .
-------------->   cname   CSECT
                          .
     L   11,=A(name)      .
     USING name,11        .
```

RETRIEVING ARGUMENTS FROM THE ARGUMENT LIST

The argument list contains addresses for the arguments passed to a subprogram. The order of these addresses is the same as the order specified for the arguments in the calling statement in the main program. The address for the argument list is placed in register 1. For example, when the statement:

```
CALL MYSUB(A,B,C)
```

is compiled, the following argument list is generated.

00000000	address of A
00000000	address of B
10000000	address of C

For purposes of discussion, A is a REAL*8 variable, B is a subprogram name, and C is an array.

The address of a variable in the calling program is placed in the argument list. The following instructions in an assembler language subprogram can be used to move the REAL*8 variable A to location VAR in the subprogram.

```
L      Q,0(1)
MVC    VAR(8),0(Q)
```

where:
Q is any general register except 0.

For a subprogram reference, an address of a storage location is placed in the argument list. The address at this storage location is the entry point to the subprogram. The following instructions can be used to enter subprogram B from the subprogram to which B is passed as an argument.

```
L      Q,4(1)
L      15,0(Q)
BALR   14,15
```

where:
Q is any general register except 0.

For an array, the address of the first variable in the array is placed in the argument list. An array [for example, a three-dimensional array C(3,2,2)] appears in this format in main storage.

```
 C(1,1,1) C(2,1,1) C(3,1,1) C(1,2,1)   --┐
┌---------------------------------------┘
└-C(2,2,1) C(3,2,1) C(1,1,2) C(2,1,2)   --┐
┌---------------------------------------┘
└-C(3,1,2) C(1,2,2) C(2,2,2) C(3,2,2)
```

Table A-2 shows the general subscript format for arrays of 1, 2, and 3 dimensions.

Table A-2. Dimension and Subscript Format

Array A	Subscript Format
A(D1)	A(S1)
A(D1,D2)	A(S1,S2)
A(D1,D2,D3)	A(S1,S2,S3)
D1, D2, and D3 are integer constants used in the DIMENSION statement. S1, S2, and S3 are subscripts used with subscripted variables.	

The address of the first variable in the array is placed in the argument list. To retrieve any other variables in the array, the displacement of the variable, that is, the distance of a variable from the first variable in the array, must be calculated. The formulas for computing the displacement (DISPLC) of a variable for one, two, and three dimensional arrays are:

```
DISPLC=(S1-1)*L
DISPLC=(S1-1)*L+(S2-1)*D1*L
DISPLC=(S1-1)*L+(S2-1)*D1*L+(S3-1)*D2*D1*L
```

where:
L is the length of each variable in this array.

For example, the variable C(2,1,2) in the main program is to be moved to a location ARVAR in the subprogram. Using the formula for displacement of integer variables in a three-dimensional array, the displacement (DISP) is calcualted to be 28. The following instructions can be used to move the variable,

```
L     Q,8(1)
L     R,DISP
L     S,0(Q,R)
ST    S,ARVAR
```

where:
Q and R are any general registers except 0.
S is any general register. Q and R cannot be general register 0.

Example: An assembler language subprogram is to be named ADDARR, and a real variable, an array, and an integer variable are to be passed as arguments to the subprogram. The statement:

```
CALL ADDARR (X,Y,J)
```

is used to call the subprogram. Figure A-12 shows the linkage used in the assembler subprogram.

RETURN i in an Assembler Language Subprogram

When a statement number is an argument in a CALL to an assembler language subprogram, the subprogram cannot access the statement number argument.

To accomplish the same thing as the FORTRAN statement RETURN *i* (used in FORTRAN subprograms to return to a point other than that immediately following the CALL), the assembler subprogram must place 4**i* in register 15 before returning to the calling program.

For example, when the statement:

```
CALL SUB (A,B,&10,&20)
```

is used to call an assembler language subprogram, the following instructions would cause the subprogram to return to the proper point in the calling program:

```
      .
      .
      .
LA 15,4 (to return to 10)

BCR 15,14
      .
      .
      .
LA 15,8 (to return to 20)

BCR 15,14
```

OBJECT-TIME REPRESENTATION OF FORTRAN VARIABLES

The programmer who uses FORTRAN in connection with assembler language may need to know how the various FORTRAN data types appear in the computer. The following examples illustrate the object-time representation of FORTRAN variables as they appear in System/360 and System/370.

```
Name     Operation Operand
ADDARR   START     0
B        EQU       8
         BC        15,12(15)
         DC        X'7'
         DC        CL7'ADDARR'
         STM       14,12,12(13)
         BALR      B,0
         USING     *,B
         L         2,8(1)          MOVE 3RD ARGUMENT TO LOCATION CALLED
         MVC       INDEX(4),0(2)   INDEX IN ASSEMBLER LANGUAGE SUBPROGRAM.
         L         3,0(1)          MOVE 1ST ARGUMENT TO LOCATION CALLED VAR
         MVC       VAR(4),0(3)     IN ASSEMBLER LANGUAGE SUBPROGRAM.
         L         4,4(1)          LOAD ADDRESS OF ARRAY INTO REGISTER 4.
                   (user-written statements)
                     .
                     .
                     .
         LM        2,12,28(13)
         MVI       12(13),X'FF'
         BCR       15,14
         DS        0F
INDEX    DS        1F
VAR      DS        1F
```

Figure A-12. Assembler Subprogram Example

INTEGER Type

INTEGER values are treated as fixed-point operands by the compiler, and are governed by the principles of System/360 and System/370 fixed-point arithmetic. INTEGER values are converted into either fullword (32-bit) or halfword (16-bit) signed integers.

```
Example:    INTEGER*2 ITEM/76/,VALUE
            INTEGER*4 F,F64/100/
            F = 15
            VALUE =-2
```

The values of the variables ITEM, VALUE, F, F64 appear in storage as follows:

```
     <-----2 Bytes------>                    <-----2 Bytes------>
     ┌─┬───────┬────────┐                   ┌─┬───────┬────────┐
ITEM │0│0000000│01001100│              VALUE│1│1111111│11111110│
     └─┴───────┴────────┘                   └─┴───────┴────────┘
      S 1              15                    S 1              15

     <--------------4 Bytes--------------->
     ┌─┬───────┬────────┬────────┬────────┐
F    │0│0000000│00000000│00000000│00001111│
     └─┴───────┴────────┴────────┴────────┘
      S 1                                31

     <--------------4 Bytes--------------->
     ┌─┬───────┬────────┬────────┬────────┐
F64  │0│0000000│00000000│00000000│01100100│
     └─┴───────┴────────┴────────┴────────┘
      S 1                                31
```

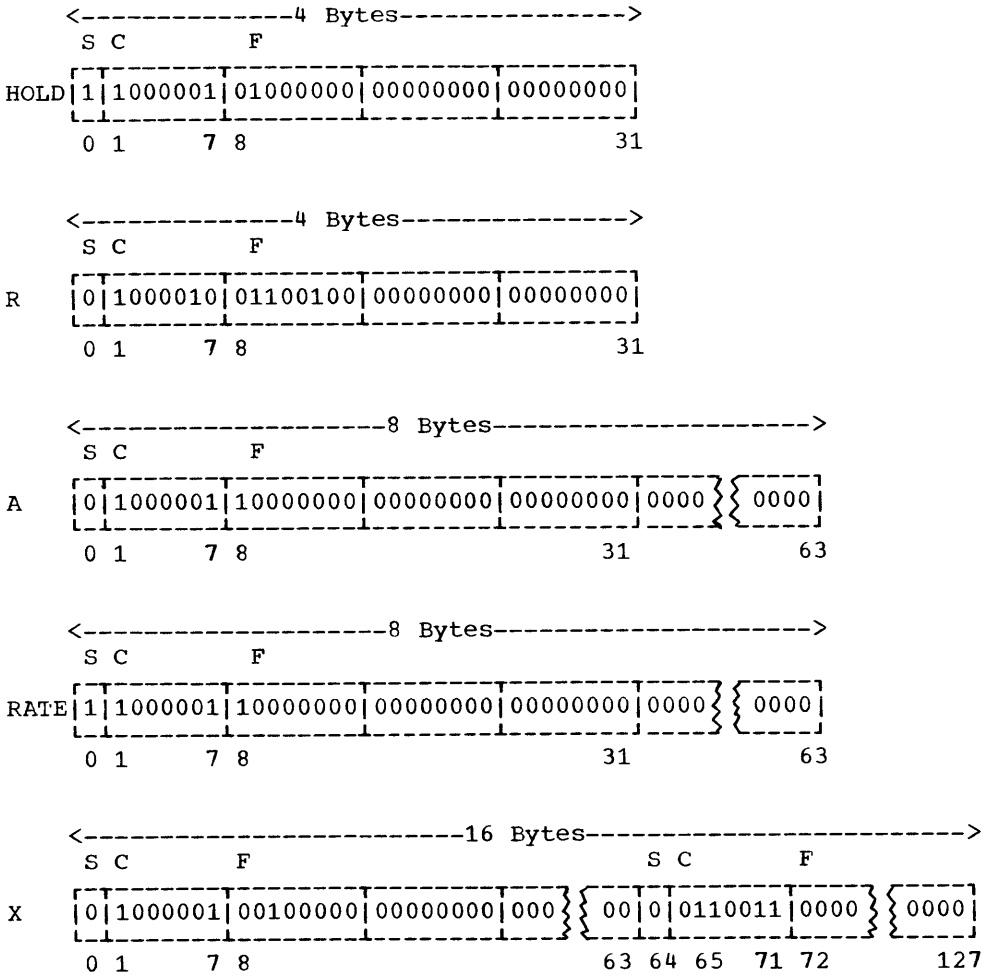
where S in bit position 0 represents the sign bit. All negative numbers are represented in two's complement notation with a one in the sign-bit position. (For a more complete explanation of fixed-point arithmetic, refer to the appropriate principles of operation publication, as listed in the Preface.)

REAL Type

All REAL variables are converted into short (32 bit), long (64 bit), or extended (128 bit) floating-point numbers by the compiler. The length of the numbers is determined by FORTRAN IV specification conventions.

Example:
```
REAL*4 HOLD,R/100./
REAL*8 A,RATE/-8./
REAL*16 X
HOLD = -4.
A = 8.0D0
X = 2.0Q0
```

The values of the variables appear in storage as follows:



Legend:

S (sign bit) occupies bit position 0.

C (characteristic), or exponent, occupies bit positions 1 through 7.

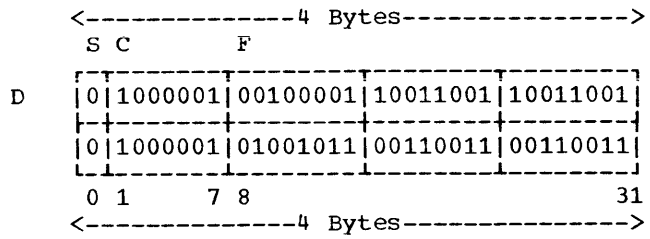
F (fraction) occupies either bit positions 8 through 31 for a short, floating-point number; bit positions 8 through 63 for a long, floating-point number; or bit positions 8 through 63, 72 through 127 for an extended-precision floating-point number (bit positions 64-71 represent a sign + characteristic having a value 14 less than the data represented in bits 0 through 7).

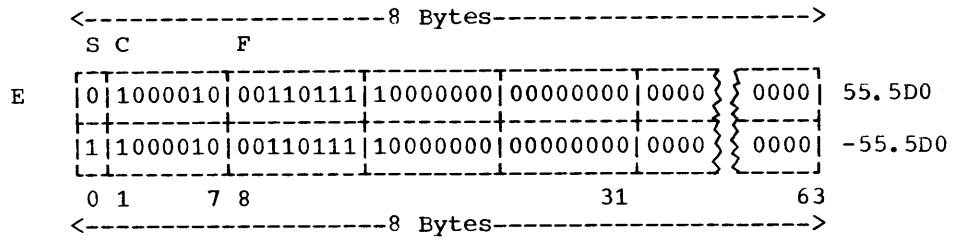
COMPLEX Type

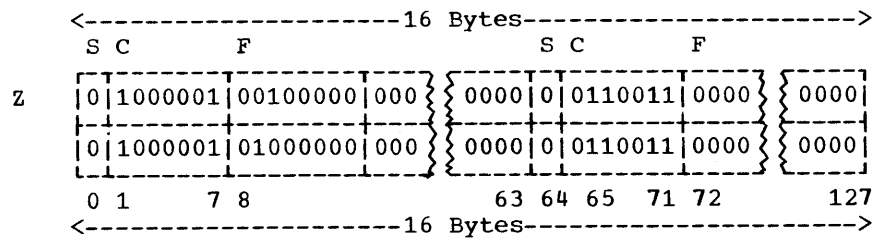
A COMPLEX variable has two parts (real and imaginary) and is treated internally as a pair of REAL numbers. The COMPLEX parts are converted into two short, long, or extended floating-point numbers.

Example:
```
COMPLEX D/(2.1,4.7)/,E*16,Z*32
E = (55,5D0-55.5D0)
Z=(2.0Q0,4.0Q0)
```

The values of the variables D, E, and Z appear in storage as follows:







LOGICAL Type

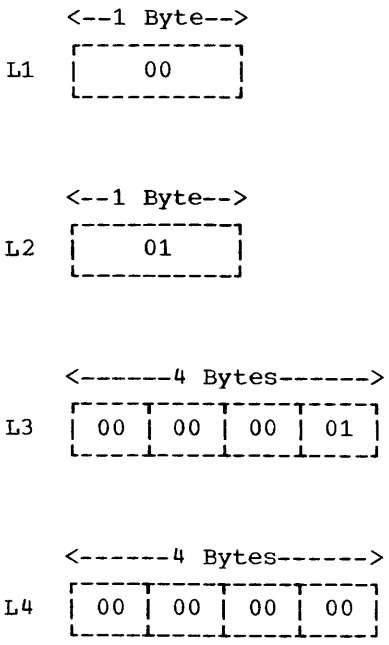
FORTRAN IV LOGICAL variables may specify only 2 values:

.TRUE. or .FALSE.

These logical values are assigned numerical values of '1' and '0', for .TRUE. and .FALSE., respectively.

Example:
```
LOGICAL*1 L1,L2/.TRUE./
LOGICAL*4 L3,L4/.FALSE./
L1 = .FALSE.
L3 = .TRUE.
```

The variables L1, L2, L3, and L4 are assigned the following values (using hexadecimal notation):



The DUMP or PDUMP subroutine can also be used as an additional tool for understanding the object-time representation of FORTRAN data. Refer to the "Use of DUMP and PDUMP" section in the "Programming Considerations" chapter of this publication or consult the FORTRAN IV Library Subprograms publication.

The UNIT parameter of the DD statement can identify an input or output unit by its actual address, its type number, or its group name. Type numbers, automatically established at system generation, correspond to units entered into system configurations. Type numbers and corresponding units are listed here for the reader's convenience.

Tape Units

Type	Description
2400	2400 series 9-Track Magnetic Tape Drive that can be allocated to a data set written or to be written in 800 bpi density
2400-1	2400 series Magnetic Tape Drive with 7-Track Compatibility and without Data Conversion
2400-2	2400 series Magnetic Tape Drive with 7-Track Compatibility and Data Conversion
2400-3	2400 series 9-Track Magnetic Tape Drive that can be allocated to a data set written or to be written in 1600 bpi density
2400-4	2400 series 9-Track Magnetic Tape Drive having an 800 and 1600 bpi capability

Direct Access Units

Type	Description
2301	2301 Drum Storage Unit
2302	2302 Disk Storage Drive
2303	2303 Drum Storage Unit**
2311	Any 2311 Disk Storage Drive**
2314	2314 Storage Facility
2321	Any bin mounted on a 2321 data cell drive**
3330	3330 Disk Storage Facility*
2305	2305 Drum Storage Unit*

Unit Record Equipment

Type	Description
1052	1052 Printer-Keyboard
1403	1403 Printer or 1404 Printer (continuous form only)
1442	1442 Card Read Punch
1443	any 1443 Printer
2501	2501 Card Reader
2520	2520 Card Read Punch
2540	2540 Card Read Punch (read feed)
2540-2	2540 Card Read Punch (punch feed)
2671	2671 Paper Tape Reader
3211	3211 Printer*

Graphic Units

Type	Description
1053	1053 Model 4 Printer
2250-1	2250 Display Unit, Model 1
2250-3	2250 Display Unit, Model 3
2260-1	2260 Model 1 Display Station (Local Attachment)
2260-2	2260 Model 2 Display Station (Local Attachment)
2280	2280 Film Recorder
2282	2282 Film Recorder-Scanner

---

*Used with System/370 models only.
**Used with System/360 Models only.

APPENDIX D: AMERICAN NATIONAL STANDARD CARRIAGE CONTROL CHARACTERS

The following are the carriage control characters supported by the FORTRAN IV language.

Code	Interpretation
blank	Space one line before printing
0	Space two lines before printing
+	Suppress space before printing
1	Skip to first line of a new page

(Where more than one page reference is given, the major reference appears first.)

READER'S COMMENTS

TITLE: IBM OS FORTRAN IV (H Extended) Compiler Programmer's Guide

ORDER NO. SC28-6852-1

Your comments assist us in improving the usefulness of our publications; they are an important part of the input used in preparing updates to the publications. All comments and suggestions become the property of IBM.

Please do not use this form for technical questions about the system or for requests for additional publications; this only delays the response. Instead, direct your inquiries or requests to your IBM representative or to the IBM Branch Office serving your locality.

Corrections or clarifications needed:

Page	*Comment*

Please include your name and address in the space below if you wish a reply.

Thank you for your cooperation. No postage necessary if mailed in the U.S.A.

cut along this line

fold fold

FIRST CLASS
PERMIT NO. 33504
NEW YORK, N.Y.

BUSINESS REPLY MAIL

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .

IBM CORPORATION
1271 Avenue of the Americas
New York, New York 10020

Attention: PUBLICATIONS

fold fold

IBM

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
[U.S.A. only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]

IBM OS FORTRAN IV (H Ext) Compiler P. G. Printed in U.S.A. SC28-6852-1

IBM OS FORTRAN IV (H Ext) Compiler P. G. Printed in U.S.A. SC28-6852-1

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)