File No. S360-25
Form C28-6813-0

44PS

IBM Systems Reference Library

IBM System/360 Model 44
Programming System
Guide to System Use for FORTRAN Programmers

IBM System/360 Model 44
Programming System
Guide to System Use for FORTRAN Programmers

This publication describes how to use the Model 44 Programming System to compile and execute programs written in the IBM System/360 FORTRAN IV language. A discussion of program optimization and of the restrictions of the Model 44 FORTRAN IV compiler is also included.

This publication is directed primarily at programmers who are familiar with the FORTRAN IV language. Previous knowledge of the Model 44 Programming System is not required.

PREFACE

The purpose of this publication is to provide programmers with the information required to process FORTRAN programs under control of the Model 44 Programming System. The three steps involved in processing a FORTRAN program are compilation, editing, and execution.

This publication is not intended to be an exhaustive discussion of the capabilities of the Model 44 Programming System; only those features that will be commonly used by FORTRAN programmers are presented. A more complete description of system capabilities can be found in the publication IBM System/360 Model 44 Programming System: Guide to System Use, Form C28-6812.

It is assumed that the reader is familiar with the FORTRAN language as described in the publication IBM System/360 FORTRAN IV Language, Form C28-6515. No previous knowledge of the Model 44 Programming System is required.

The organization of this publication is such that the new reader is familiarized with programming system concepts and learns of the facilities available to him before encountering procedural details. The detailed information also serves as a body of reference material for the programmer who is already familiar with system concepts. As an aid to the experienced programmer in his use of this publication, the content of each chapter is described briefly below:

1. "Introduction" describes the Model 44 Programming System and indicates the programming system components that are involved in FORTRAN processing.
2. "Job Definition" defines the terms "job" and "job step," describes several types of jobs, and shows how to structure a job via job control statements.
3. "Data Sets" provides information on transmission of data to and from external storage devices. It shows the correspondence between the data set reference numbers used in FORTRAN input/output statements and the symbolic unit names used by the programming system. Discussions of data set creation and data set maintenance are included.
4. "Job Processing" describes each of the processing steps -- compilation, editing, and execution -- in detail. The facilities available to the programmer are related to options in job control statements and linkage editor control statements.
5. "Control Statements" provides details on filling out job control statements and linkage editor control statements. Conventions of the job control language are described.
6. "System Output" describes and illustrates the listings, maps, error messages, and dumps produced by the system components.
7. "Programming Considerations" gives information useful when writing FORTRAN source programs. This includes techniques that enable the programmer to optimize compilation and execution speed. Restrictions of the FORTRAN IV compiler are also given.

Four appendixes provide examples of typical job decks; EBCDIC and BCDIC card codes for valid FORTRAN characters; linkage conventions for assembler language subprograms; and a list of diagnostic messages.

First Edition

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

Copies of this and other IBM publications can be obtained through IBM Branch Offices.

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

ILLUSTRATIONS

FIGURES

TABLES

The IBM System/360 Model 44 Programming System provides a means for compiling and executing programs written in the FORTRAN IV language. Under control of the programming system, a set of FORTRAN IV source statements is translated to form a module. In order to be executed, the module in turn must be processed to form a phase. The reasons for this will become clear later. For now it is sufficient to note that the course of the FORTRAN program through the programming system is from source statements to module to phase.

The Model 44 Programming System itself is essentially a collection of programs, some interrelated, others independent. The related programs include a supervisor, a set of system support programs, and two language processors. There are several independent or stand-alone programs. Not all of these component programs are involved in compiling and executing a FORTRAN program. Figure 1 shows the structure of the programming system and indicates those components that are of immediate interest to the FORTRAN programmer.

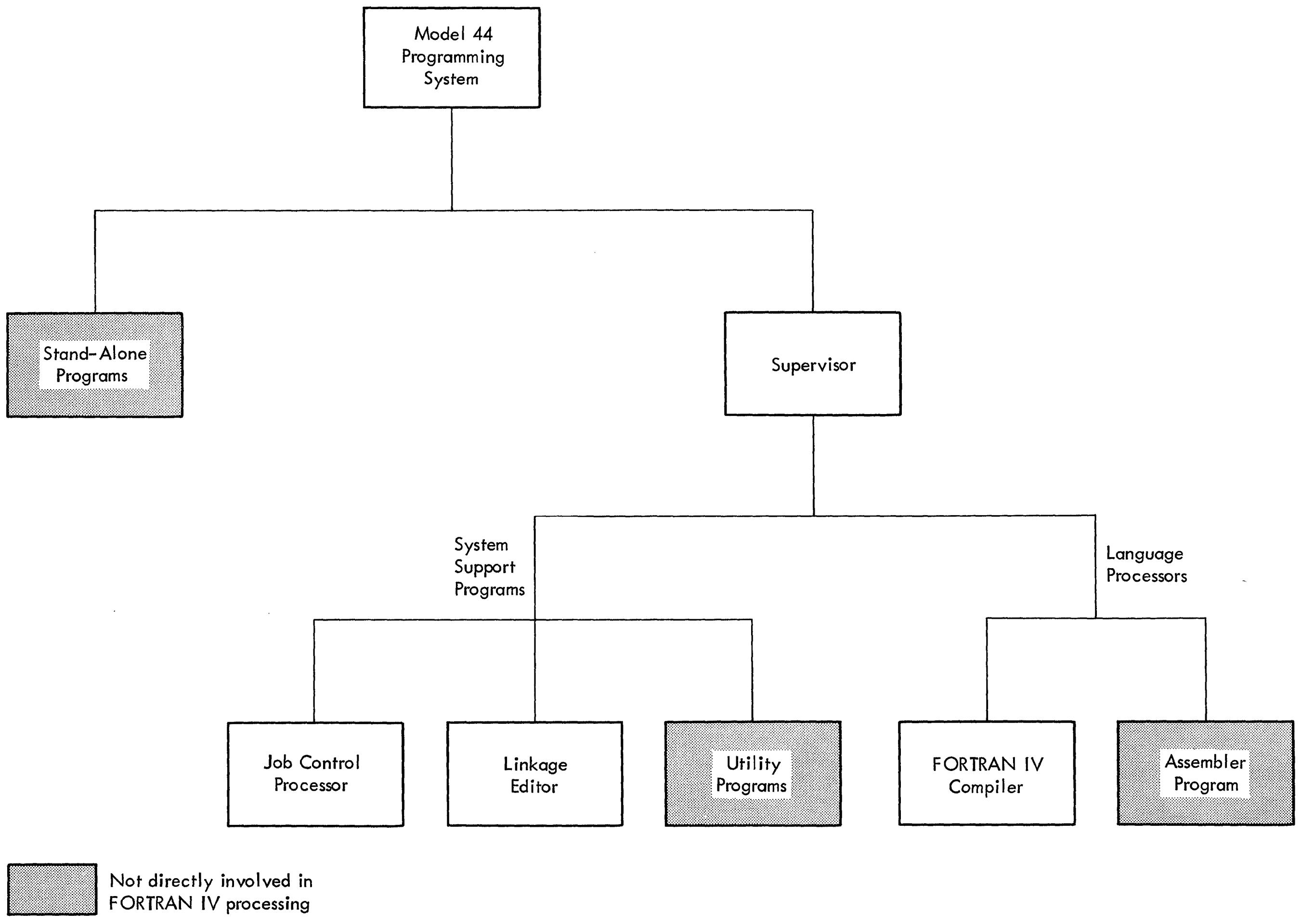


Figure 1. Programming System Structure

SUPERVISOR

The supervisor is the system control program. To say that a program operates under control of the programming system is to say that it operates under control of the supervisor. Accordingly, the stand-alone programs, although part of the programming system, do not operate under system control.

The main function of the supervisor is to provide the orderly and efficient flow of jobs through the programming system. (A job is some specified unit of work, such as the processing of a FORTRAN program.) The supervisor loads into the computer the phases that are to be executed. During execution of the program, control usually alternates between the supervisor and the processing program, as the supervisor, for example, handles all requests for input/output operations.

Detailed information about the supervisor's operation need not concern the FORTRAN programmer. Anyone interested in this material, however, can find it in the publication IBM System/360 Model 44 Programming System: Guide to System Use, Form C28-6812.

JOB CONTROL PROCESSOR

Among the system support programs is the job control processor. Its primary function is the processing of job control statements, which describe the jobs to be performed and specify the programmer's requirements for each job. Job control statements are written by the programmer, using the job control language. The use of job control statements and the rules for specifying them in job control language are discussed later.

LINKAGE EDITOR

The linkage editor, another system support program, processes modules and incorporates them into phases. A single module can be edited to form a single phase or several modules can be edited or linked together to form one executable phase. Moreover, a module to be processed by the linkage editor may be one that was just created (during the same job) or one that was created in a previous job and saved.

The use of the linkage editor to perform these functions is controlled by the programmer through job control statements. In addition, there are several linkage editor control statements. Information on their use is given later.

UTILITY PROGRAMS

The remaining system support programs are the utility programs. They are used primarily for initializing and maintaining external storage devices and for transmitting data between external storage devices. More information about external storage is given later. Since the utility programs are not directly involved in compiling and executing a FORTRAN program, they are not described in this publication. Details on their function and use can be found in IBM System/360 Model 44 Programming System: Guide to System Use, Form C28-6812.

FORTRAN IV COMPILER

The FORTRAN IV compiler is the system component that translates FORTRAN source statements and produces a module. As the statements are compiled, they are checked for errors by the compiler, which issues a diagnostic message for each error discovered. All of this is discussed more completely later.

ASSEMBLER PROGRAM

The other language processor is the assembler program, which, like the FORTRAN IV compiler, translates source statements to produce a module. Source statements processed by the assembler program, however, are written in assembler language. The assembler program, therefore, is parallel in function to the FORTRAN IV compiler and does not directly concern the FORTRAN programmer.

As will be shown later, it is possible, under control of the programming system, to combine modules produced by the FORTRAN IV compiler with modules produced by the assembler program to form one executable phase. In this case, certain conventions must be followed when the assembler language source programs are written. These conventions are explained in Appendix C. For those who are interested, the assembler language is described in the publication IBM System/360 Model 44 Programming System: Assembler Language, Form C28-6811, while the use of the assembler program is explained in IBM System/360 Model 44 Programming System: Guide to System Use, Form C28-6812.

PROGRAMMING SYSTEM OPERATION

The Model 44 Programming System is distributed to an installation as a deck of cards. Before it can be used, the system must be constructed. System construction is a process whereby the programming system is written onto an IBM 2315 Disk Cartridge, which is mounted on a single disk storage drive within the Model 44 processing unit. The disk cartridge containing the system is called the system residence volume or system residence disk. Once the system has been constructed, it can be tailored to meet the needs of the installation via a process known as system assembly.

The programming system is put into operation as a result of an operator-initiated procedure known as IPL (initial program load). At this time, the supervisor is loaded from the system residence disk into the main storage of the computer, where it remains for as long as the programming system is in operation.

The supervisor then loads the job control processor, which reads and interprets job control statements. One type of job control statement (the EXEC statement) is used to request the execution of a specific program. When an EXEC statement is encountered, the job control processor relays the name of the program to be executed to the supervisor and returns control to it. The supervisor then loads the requested program, overlaying the job control processor.

When the program finishes execution, control is returned to the supervisor, which again loads the job control processor, this time overlaying the program just executed. The job control processor continues reading and interpreting job control statements until another EXEC statement is encountered (in this case the above procedure is repeated) or until a STOP statement is encountered. The STOP statement terminates the operation of the programming system. Before the system can be used again, the operator must put it back into operation via either the IPL procedure or a restart procedure.

JOB DEFINITION

A job is a specified unit of work to be performed under control of the programming system. As was pointed out earlier, a typical job might be the processing of a FORTRAN program -- compiling source statements, editing the module thus produced to form a phase, and then executing the phase. Or a job might be the processing of a combined FORTRAN-assembler language source program -- compiling FORTRAN source statements, assembling the assembler language statements, editing the modules to produce a phase, and then executing the phase. Job definition -- the process of specifying the work to be done during a single job -- allows the programmer much flexibility. A job can include as many or as few job steps as the programmer desires.

JOB STEPS

A job step is exactly what the name implies -- one step in the processing of a job. Thus, in the first job mentioned above, one step is the compilation of source statements; another is the editing of a module; a third is the execution of a phase. The second job mentioned involves an additional job step: assembling source language statements. Each job step is associated with the execution of a program. A compilation requires the execution of the FORTRAN IV compiler. Similarly, an assembly implies the execution of the assembler program; an editing, the execution of the linkage editor. Finally, the execution of a phase is the execution of the problem program itself.

In contrast to job definition, the definition of a job step is fixed. Each job step involves the execution of a program, whether it be a program that is part of the Model 44 Programming System or a program that is written by the user.

Compilation Job Steps

The compilation of a FORTRAN program may necessitate more than one job step (more than one execution of the FORTRAN IV compiler). In many cases, a FORTRAN program actually consists of a main program and one or more subprograms, such as FUNCTION subprograms and SUBROUTINE subprograms written by the FORTRAN programmer. To compile such a program, a separate job step must be specified for the main program and for each of the subprograms. The FORTRAN IV compiler is executed several times in succession, once for the main program and once for each subprogram. Each execution of the compiler produces a module. The separate modules can then be combined into one phase by a single job step -- the execution of the linkage editor.

For a FORTRAN program that consists of a main program and two subprograms, compilation and execution requires five steps: compile (main program), compile (first subprogram), compile (second subprogram), edit (three modules), and execute (phase). Compilation and execution in three job steps -- compile, edit, and execute -- is applicable only when the FORTRAN source program is a single main program.

Multiple Phase Execution

The execution of a FORTRAN program has thus far been spoken of as the execution of a phase. It is possible, however, to organize a FORTRAN program so that it is executed as two or more phases. Such a program is called a multiphase program.

By definition, a phase is that portion of a program that is loaded into the computer by a single operation of the supervisor. (As was mentioned earlier, it is the programming system supervisor that loads phases for execution.) A FORTRAN program can be executed as a single phase as long as there is an area of main storage available to accommodate it. On the other hand, a program that is too large to be executed as a single phase must be structured as a multiphase program.

The number of phases in a FORTRAN program has no effect, however, on the number of job steps required to process that program. As will be seen, the linkage editor can produce one or more phases in a single job step. Similarly, both single-phase and multiphase programs require only one execution job step. Phase execution is the execution of all the phases that make up one FORTRAN program.

Detailed information on structuring multiphase programs, as well as information on using the facilities of the programming system to create multiple phases and execute them, can be found in a subsequent chapter, "Job Processing." For now, one need only be aware that the facility for creating and executing multiphase programs exists.

TYPES OF JOBS

The typical job falls into one of several categories. A brief description of these follows; a more complete discussion appears later, in the chapter "Job Processing."

Compile Only: This type of job involves only the execution of the FORTRAN IV compiler. It is useful when checking for errors in FORTRAN source statements. A compile-only job is also used to produce a module that is to be further processed in a subsequent job.

A compile-only job can consist of one job step or several successive compilation job steps.

Edit Only: This type of job involves only the execution of the linkage editor. It is used primarily to combine modules produced in previous compile-only jobs and to check that all cross-references between modules have been resolved. The programmer can specify that all modules be combined to form one phase; or he can specify that some modules form one phase and that others form one or more other phases. The phase output produced as the result of an edit-only job can be retained for execution in a subsequent job.

Compile and Edit: This type of job combines the functions of the compile-only and the edit-only jobs. It calls for the execution of both the FORTRAN IV compiler and the linkage editor. The job can include one or more compilations, resulting in one or more modules. The programmer can specify that the linkage editor process any or all of the modules just produced; in addition, he can specify that one or more previously produced modules be included in the linkage editor processing.

Execute Only: This type of job involves the execution of a phase (or multiple phases) produced in a previous job. Once a FORTRAN program has been compiled and edited successfully, it can be retained as one or more phases and executed whenever needed. This eliminates the need for recompiling and re-editing every time a FORTRAN program is to be executed.

Edit and Execute: This type of job combines the functions of the edit-only and the execute-only jobs. It calls for the execution of both the linkage editor and the resulting phase(s).

Compile, Edit, and Execute: This type of job combines the functions of the compile-and-edit and the execute-only jobs. It calls for the execution of the FORTRAN IV compiler, the linkage editor, and the problem program; that is, the FORTRAN program is to be completely processed.

When considering the definition of his job, the programmer should be aware of the following: if a job step is canceled during execution, the entire job is terminated; any remaining job steps are skipped. Thus, in a compile-edit-and-execute job, a failure in compilation precludes the editing of the module(s) and phase execution. Similarly, a failure in editing precludes phase execution.

For this reason, a job usually should (but need not) consist of related job steps only. For example, if two independent single-phase executions are included in one job, the failure of the first phase execution precludes the execution of the second phase. Defining each phase execution as a separate job would prevent this from happening. If successful execution of both phases can be guaranteed before the job is run, however, the programmer may prefer to include both executions in a single job.

JOB DEFINITION STATEMENTS

Once the programmer has decided what work is to be done within his job and how many job steps are required to perform the job, he can then define his job by writing job control statements. Since these statements are usually punched in cards, the set of job control statements is referred to as a job deck. In addition to job control statements, the job deck can include input data for a program that is executed during a job step. For example, input data for the FORTRAN IV compiler -- the FORTRAN source statements to be compiled -- can be placed in the job deck.

The inclusion of input data in the job deck depends upon the way the installation has assigned input/output devices. Job control statements are read from the unit named SYSRDR (system reader), which can be either a card reader or a magnetic tape unit. Input to the processing programs is read from the unit named SYSIPT (system input), which also can be either a card reader or a magnetic tape unit. The installation has the option of assigning either two separate devices for these units (one device for SYSRDR, a second device for SYSIPT) or one device to serve as both SYSRDR and SYSIPT. If two devices have been assigned, the job deck must consist of only job control statements; input data must be kept separate. If only one device has been assigned, input data must be included within the job deck.

There are four job control statements that can be used for job definition: the JOB statement, the EXEC statement, the end-of-job (/&) statement, and the end-of-data (/*) statement. The discussion of these job control statements in this chapter is limited to the function and use of each statement. The rules for writing each statement are given in a subsequent chapter, "Control Statements."

The JOB statement defines the start of a job. One JOB statement is required for every job; it must be the first statement in the job deck. If the programmer wishes to name his job, he may specify this name in the JOB statement. Also, any job accounting information required by the programmer's installation can be placed in this statement.

The EXEC statement requests the execution of a program. Therefore, one EXEC statement is required for each job step within a job. The EXEC statement indicates the program that is to be executed (for example, the FORTRAN IV compiler, the linkage editor). As in the JOB statement, the programmer may specify a name, in this case, for the job step, and also any accounting information required by the installation. As soon as the EXEC statement has been processed, the program indicated by the statement begins execution.

The end-of-job statement, also referred to as the /& -- slash ampersand -- statement, defines the end of a job. A /& statement must appear as the last statement in the job deck.

The end-of-data statement, also referred to as the /* -- slash asterisk -- statement, defines the end of a program's input data. When the data is included within the job deck (that is, SYSIPT and SYSRDR are the same device), it is placed immediately following the EXEC statement for the program that requires it. The /* statement immediately follows the input data. For example, FORTRAN source statements would be placed immediately after the EXEC statement for the FORTRAN IV compiler; a /* statement would follow the last FORTRAN source statement.

When input data is kept separate (that is, SYSIPT and SYSRDR are separate devices), the /* statement immediately follows each set of input data on SYSIPT. For example, if a job consists of two compilation job steps, an editing job step, and an execution job step, SYSIPT would contain the source statements for the first compilation followed by a /* statement, the source statements for the second compilation followed by a /* statement, any input data for the linkage editor followed by a /* statement, and perhaps some input data for the problem program followed by a /* statement.

A /* statement must always be used in an editing job step whether or not there is any input data for the linkage editor. When there is input data, the /* statement immediately follows the input data, whether it is in the job deck or on a separate SYSIPT. When there is no input data, the /* statement either immediately follows the EXEC statement for the linkage editor or appears in the appropriate place on a separate SYSIPT.

JOB DEFINITION EXAMPLES

The following are examples of "job decks" for the various types of jobs. Their purpose is to show the order of job definition statements within a job. No attempt is made to show the contents of each statement. In addition, the examples are limited to only the job definition statements and input data; no other job control statements are shown. (Examples of complete job decks, indicating the contents of all statements, are in Appendix A.)

Two compile-only jobs are shown below: a single compilation and a multiple compilation. For all other jobs, the reader can assume that only one set of source statements, one module, and/or one phase is involved. Input data is shown only for the sake of example; it is not always required in the job deck.

<u>Compile only (one compilation):</u>

```
JOB statement
EXEC statement (FORTRAN IV compiler)
Source language statements
/* statement
/& statement
```

<u>Compile only (three compilations):</u>

```
JOB statement
EXEC statement (FORTRAN IV compiler)
Source language statements
/* statement
EXEC statement (FORTRAN IV compiler)
Source language statements
/* statement
EXEC statement (FORTRAN IV compiler)
Source language statements
/* statement
/& statement
```

Edit only:

JOB statement
EXEC statement (linkage editor)
Module to be edited
/* statement
/& statement

Compile and edit:

JOB statement
EXEC statement (FORTRAN IV compiler)
Source language statements
/* statement
EXEC statement (linkage editor)
/* statement
/& statement

Execute only:

JOB statement
EXEC statement (phase)
Data used by problem program
/* statement
/& statement

Edit and execute:

JOB statement
EXEC statement (linkage editor)
Module to be edited
/* statement
EXEC statement[1]
Data used by problem program
/* statement
/& statement

Compile, edit, and execute:

JOB statement
EXEC statement (FORTRAN IV compiler)
Source language statements
/* statement
EXEC statement (linkage editor)
/* statement
EXEC statement[1]
Data used by problem program
/* statement
/& statement

OTHER JOB CONTROL STATEMENTS

The four job definition statements form the framework of the job deck. There are a number of other job control statements in the job control language. Not all of them must appear in the job deck; in fact, some FORTRAN programs can be processed without using any of these additional statements. The job control statements are grouped by category and summarized briefly in Table 1.

---

[1]In this case, the program to be executed need not be indicated; the system will assume that the phase just produced by the linkage editor is to be executed.

Table 1. Job Control Statements

STATEMENT	FUNCTION
JOB DEFINITION	
// JOB	Defines the start of a job.
// EXEC	Defines the start of a job step execution and indicates the program to be executed.
/&	Indicates the end of a job.
/*	Indicates the end of input data for a processing program.
SYMBOLIC UNIT ASSIGNMENT	
// ALLOC	Allocates space for a new data set.
// LABEL	Defines the characteristics of a data set.
// ACCESS	Permits access to an existing data set.
// RESET	Restores unit assignments to their status at the start of the job.
// LISTIO	Lists data set and device assignments.
DATA SET MAINTENANCE	
// DELETE	Deletes a data set from a volume or a member from a directoried data set.
// CONDENSE	Condenses a directoried data set.
// RENAME	Renames a data set or a member of a directoried data set.
// CATLG	Enters a data set name into the catalog.
// UNCATLG	Removes a data set name from the catalog.
MISCELLANEOUS	
// PAUSE	Allows pause for operator action.
* (comments)	Allows logging of comments on system log.
// REWIND	Rewinds a tape; repositions a data set on a direct access volume to its beginning.
// UNLOAD	Rewinds and unloads a tape.

The double slash preceding each statement name identifies the statement as a job control statement. Most of the statements are used for data management -- creating, manipulating, and keeping track of data sets (externally stored collections of data, from which data is read and into which data is written).

Information about using the remaining control statements is given in the chapters "Data Sets" and "Job Processing." Rules for writing these statements are in the chapter "Control Statements."

Almost all FORTRAN programs include input/output statements calling for data to be read from or written into data sets on external storage devices. Each data set is identified by a data set reference number within the FORTRAN source statement. When processing data under control of the Model 44 Programming System, the FORTRAN programmer can share system data sets -- data sets used by the programming system itself -- or he can use his own data sets, referred to hereinafter as private data sets.

The data set reference numbers acceptable to the Model 44 FORTRAN IV compiler range from 0 through 8. Within the Model 44 FORTRAN IV compiler, each data set reference number corresponds to a symbolic unit name, which in turn is associated with a particular data set. The correspondence between data set reference numbers and symbolic unit names is shown in Table 2.

Table 2. Data Set Reference Numbers and Symbolic Unit Names

0 - SYSLOG	3 - SYS003	6 - SYSOPT
1 - SYS001	4 - SYS004	7 - SYSPCH
2 - SYS002	5 - SYSIPT	8 - SYS000

The data set reference numbers 0, 1, and 5 through 8 refer to system units, symbolic units that are required for programming system operation. Each system unit has a predefined relationship with a system data set (that is, each system unit name will have been already associated with a system data set by the time the FORTRAN programmer's job is to be run).

The data set reference numbers 2, 3, and 4 refer to units for which a predefined relationship (also called a standard unit assignment) is not required. It is up to the programmer to determine whether or not a standard unit assignment for any of these units exists at his installation.

It is also the programmer's responsibility to determine whether the installation has modified the FORTRAN IV compiler and changed the relationships between data set reference numbers and symbolic unit names. The relationships shown in Table 2 reflect the FORTRAN IV compiler distributed as part of the Model 44 Programming System.

USING SYSTEM DATA SETS

To use a system data set, a programmer need only specify the appropriate data set reference number in his program. The FORTRAN IV compiler associates the number with the corresponding system unit. The relationship between the system units and the system data sets is predefined by standard unit assignments.

The system log (data set reference number 0) is located on the unit named SYSLOG. This unit is always the console printer-keyboard used for communication with the operator.

The system work data set (data set reference number 1) is located on the unit named SYS001. The data set contains intermediate data from any of the programming system components. (Intermediate data is data that is stored temporarily on an external medium by one part of a program to be read and processed by another part of that program.) Intermediate data for a FORTRAN program can be written into and read from the system work data set.

The system input data set (data set reference number 5) is located on the unit named SYSIPT. The data set contains input to the processing programs, such as FORTRAN source statements and linkage editor control statements. Input data for a FORTRAN program can be placed on SYSIPT along with any other input data. If SYSIPT is assigned to the same device as SYSRDR, the input data should be placed in the job deck immediately after the EXEC statement that requests phase execution.

The system output data set (data set reference number 6) is located on the unit named SYSOPT. The data set contains system print output, such as a listing of FORTRAN source statements. Print output from a FORTRAN program can be written into the system output data set.

The system punch data set (data set reference number 7) is located on the unit named SYSPCH. The data set contains all of the system punch output. Punch output from a FORTRAN program can be written into the system punch data set.

The linkage editor input data set (data set reference number 8) is located on the unit named SYS000. The data set contains output from the compiler (or the assembler) that is to be used as input to the linkage editor. For example, a module that is produced by the compiler and intended for editing in a subsequent job step is written on SYS000. Later in the job, the linkage editor reads the module from SYS000. The FORTRAN programmer may use the linkage editor input data set provided that it is no longer needed during the job by the linkage editor (that is, there is no editing job step subsequent to the job step in which the FORTRAN programmer uses SYS000).

When using any system data set, the programmer should be aware of the installation device assignment for the unit on which the data set is located. For example, SYSIPT can be either a card reader or a magnetic tape unit. SYSPCH can be either a card punch or a magnetic tape unit. SYSOPT can be either a printer or a magnetic tape unit. SYS001 can be either a magnetic tape unit or an area of disk storage. Also, if SYS001 is an area of disk storage, the programmer should know how large an area the installation has reserved (or <u>allocated</u>) for SYS001 and, thus, determine whether it can accommodate the work data for his FORTRAN program.

If a programmer can satisfy his data requirements by using only system data sets, he need not concern himself with the details of using private data sets. It is also unlikely that he will have to use any of the job control statements intended for data management. Since the remainder of this chapter discusses the use and maintenance of private data sets, the programmer using only system data sets can skip to the next chapter.

<u>USING PRIVATE DATA SETS</u>

To use one of his own data sets, a programmer specifies any one of the data set reference numbers 2, 3, or 4 in his program. As with the system data sets, the FORTRAN IV compiler associates the number with a particular symbolic unit. Unless a standard unit assignment exists for this unit, the FORTRAN programmer must establish a relationship between the symbolic unit and his data set by using job control statements. Even when a standard unit assignment is in effect, the programmer can use job control statements to temporarily override the assignment and establish a new relationship.

In addition, the programmer must provide the system with whatever information it needs to be able to process the data set. The nature of the required information varies according to the type of data set.

One way of classifying a data set is according to the type of storage medium it occupies. This places a data set into one of three categories: unit record data sets, tape data sets, and direct access data sets.

UNIT RECORD DATA SETS

Unit record data sets include data sets on cards and data sets on the printed page. Card data sets can be further divided into input data sets, which contain data to be read, and output data sets, into which data is to be punched. Card data sets are processed either by a card reader (for input) or a card punch (for output). Printed data sets are processed by a printer.

It is unusual for private unit record data sets to be used since the type of data they contain can be accommodated by the system data sets. Furthermore, few installations will have card readers, card punches, or printers other than those used for system data sets. However, if the appropriate devices are available, the programmer is free to forego using system data sets.

For each private unit record data set that he uses, the programmer places an ACCESS statement in his job deck. In this statement he specifies the name of the data set and the symbolic unit name with which the data set is to be associated. He also indicates, in either of two ways, the device containing the data set. He can indicate a particular device by specifying the physical address of the device. Or he can indicate that a certain type of device is to be used by specifying a device type code. In this case, the system determines the particular device to be used and prints a message indicating its choice.

Details on writing the ACCESS statement for unit record data sets, including a list of the permissible device type codes and their meanings, can be found in the chapter "Control Statements."

TAPE DATA SETS

A tape data set is a data set on a reel of magnetic tape. A tape data set cannot extend beyond one reel of tape, nor can a reel of tape contain more than one data set.

Tape data sets fall into two categories: existing tape data sets and new tape data sets. An existing tape data set already contains data and has already been assigned to a particular tape volume (reel of tape). The programmer uses an existing tape data set either to read data from it or to add data to it.

A new tape data set is one that contains no data, nor has it been assigned to a tape volume. A new tape data set must be created by the programmer before data can be written into it. The programmer uses a new tape data set whenever he is writing an entirely new collection of data. This includes intermediate data, which is written by one part of a program and read by another part of that program.

When a data set is created, the programmer can request that the data set be placed into the system catalog. This means that the system will keep track of the data set and its location (the tape volume to which it is assigned). A data set in the system catalog is referred to as a cataloged data set.

Tape Labels

Each installation has the option of using tape labels to facilitate the use of tape data sets. Tape labels include a volume label, which identifies a particular reel of tape, and two data set labels, which provide information about the data set on the tape.

A volume label is written on the tape when the tape volume is initialized. (Volumes are initialized by a system utility program and the process usually is the responsibility of the installation. The system utility programs are discussed in the publication IBM System/360 Model 44 Programming System: Guide to System Use, Form C28-6812.) The volume label contains a volume serial number, consisting of from one through six characters, which serves to identify the tape volume.

The two data set labels are a header label and a trailer label. Both labels contain the name of the data set, its creation date, and its expiration date (the date the data set may be deleted). The header label may be written when the volume is initialized. Otherwise, it is written just before any data is written into the data set on the volume. The trailer label is written at the end of the data set.

A tape volume is considered labeled if the installation uses tape labels and if the tape has been initialized (that is, a volume label has been written on it). If the tape volume contains data that is to be read, it must also contain data set labels in order to be considered labeled.

Creating Tape Data Sets

The programmer must create any new tape data set that he wants to use. That is, he must allocate a tape volume to contain the data set -- either a particular tape volume or, as is more commonly the case, any fresh tape volume. A fresh tape volume is one that either contains no data set or contains an expired data set.

To create a tape data set, the programmer places an ALLOC statement in his job deck. In this statement, he specifies the name of the data set, the symbolic unit name with which the data set is to be associated, and a volume designation.

The volume designation identifies the device to be used, either through a device address or through a device type code. It may also include volume options, which vary according to the type of tape being used (that is, 7-track tape, 9-track tape). Finally, the volume designation indicates whether a fresh tape volume or a particular tape volume is to be used.

A fresh volume is requested by specifying the word FRESH in the volume designation. A particular tape volume is requested by specifying a volume identification (also referred to as the volid). If the tape is labeled, the volid is the volume serial number in the tape's volume label. If the tape is not labeled, the volid reflects whatever external identification is used by the installation.

The programmer can request that the data set be cataloged by specifying the CATLG parameter in the ALLOC statement. This causes the name of the data set, and an indication of its location, to be entered into the system catalog.

Details on writing the ALLOC statement for tape data sets, including lists of the permissible device type codes and volume options and their meanings, can be found in the chapter "Control Statements."

The system determines the device that is to be used, either the particular tape drive whose device address was specified or an available tape drive of the type specified. A message is printed instructing the operator to mount a tape volume on that unit, either a fresh tape volume or a tape volume with the specified volid. As soon as the tape volume is mounted, the operator gives a signal for the system to proceed.

If the tape volume is unlabeled, no further checking is done. If the tape volume is labeled, however, the system checks to see that it meets the specifications -- that is, whether the specified volid matches the volume serial number in the volume label or whether the volume is a fresh one (contains no header label or an unexpired label). If the tape volume does not meet the specifications, a message is printed, informing the operator of the discrepancy. The operator can then choose between continuing with the same tape volume or mounting another tape volume. If he mounts another volume, the checking procedure is repeated until an appropriate tape is found.

If the tape volume is labeled, the programmer must also include a LABEL statement immediately after the ALLOC statement in his job deck. In this statement, he must specify the expiration date of the data set unless the current date is to be used as the expiration date. The LABEL statement causes data set labels to be written (or their contents to be changed) when the first WRITE instruction is issued for that data set.

Details on writing the LABEL statement can be found in the chapter "Control Statements."

Using Existing Tape Data Sets

To use an existing tape data set, the programmer places an ACCESS statement in his job deck. In this statement he specifies the name of the data set, the symbolic unit name with which the data set is to be associated, and a volume designation. (The volume designation is not required for a cataloged data set because the system already has a record of this information.)

The volume designation identifies the device to be used, either through a device address or through a device type code. It may also include volume options, which vary according to the type of tape being used (that is, 7-track tape, 9-track tape). Finally, the volume designation specifies the volume identification (volid) of the tape containing the data set. The volid is required only if the tape is labeled; it may or may not be used for unlabeled tapes.

For a labeled tape, the volid is the volume serial number in the tape's volume label. For an unlabeled tape, the volid is whatever external identification is used by the installation.

If the programmer is adding data to an existing data set (rather than reading from it), he must also specify an EXT parameter in the ACCESS statement. This causes the tape volume to be positioned at the end of the existing data set.

Details on writing the ACCESS statement for tape data sets, including lists of the permissible device type codes and volume options and their meanings, can be found in the chapter "Control Statements."

The system determines the device that is to be used, either the particular tape drive whose device address was specified or an available tape drive of the type specified. A message is printed instructing the operator to mount the tape with the specified volid on that unit. If no volid was specified in the ACCESS statement (permitted for unlabeled tapes only), the message simply tells the operator to mount a tape volume. It is up to the programmer to make sure that the operator knows which volume is to be mounted.

As soon as the tape volume is mounted, the operator gives a signal for the system to proceed. If the tape volume is unlabeled, no further checking is done. If the tape volume is labeled, however, the system checks to see whether the specified volid matches the volume serial number in the volume label. If it does not match, a message is printed informing the operator of the discrepancy. The operator can then choose between continuing with the same tape volume or mounting another tape volume. If he mounts another volume, the checking procedure is repeated until an appropriate tape volume is found.

If the tape volume is labeled, the data set labels are checked when the first READ statement is issued for that data set. Checking a data set label includes comparing the data set name in the label with that specified in the ACCESS statement for the data set.

DIRECT ACCESS DATA SETS

A direct access data set resides on a <u>disk volume</u>, that is, a disk cartridge or a disk pack. A direct access data set may not extend beyond one disk volume; however, several direct access data sets may reside on a single volume. Each data set must reside on contiguous tracks and cylinders. The space on a volume occupied by a particular data set is called the <u>extent</u> of that data set.

Direct access data sets fall into two categories: existing direct access data sets and new direct access data sets. An existing direct access data set has already been assigned to a particular area of disk storage (its extent has already been defined). It may or may not contain any data.

A new direct access data set is one that contains no data, nor has its extent been defined. A new direct access data set must be created by the programmer before data can be written into it.

Disk Labels

All direct access volumes must be labeled. Disk labels include a volume label, which identifies a particular disk volume, and a volume table of contents (VTOC), which keeps track of the data sets on that volume. The VTOC is essentially a collection of labels, the first of which defines the VTOC. The VTOC also includes one label for each data set on the volume; each label contains such information as the data set name and the location of the data set on the volume. Finally, the VTOC contains one or more labels that manage space on the volume by keeping track of the extents of available space.

Disk labels are written on a direct access volume when the volume is initialized. Volumes are initialized by a system utility program and the process is usually the responsibility of the installation. (The system utility programs are discussed in the publication <u>IBM System/360 Model 44 Programming System: Guide to System Use</u>, Form C28-6812.)

Organization of Direct Access Data Sets

The programmer can organize a direct access data set in either of two ways. The first of these, called <u>sequential</u>, is the familiar structure in which records are placed in sequence. In the second organization, called <u>directoried</u>, each data set is organized into two parts, a <u>directory</u> and <u>members</u>.

A member of a directoried data set has the characteristics of a sequential data set; for example, it has a name, it is processed sequentially, and it can be associated with a symbolic unit name. However, a member is not a data set, but only part of one. Also, a member can have more than one name.

The directory keeps track of each member, its location in the data set, and its length. The directory contains at least one entry for each member. There are multiple entries for members with more than one name (one entry for each name). The system uses the directory to locate individual members when they are required.

Creating Direct Access Data Sets

The programmer must create any new direct access data sets that he wants to use. That is, he must <u>allocate</u> all or part of a disk volume for the data set. The programmer can request that space for the data set be allocated on a fresh disk volume (one that contains no data sets). Or he can request that space be allocated on a particular disk volume, either the volume having a specific volume serial number or the volume that already contains a specific data set whose location is known to the system. (The location of a data set is known to the system if it is one of the system data sets, if it is a cataloged data set, or if it is a data set for which an ALLOC or ACCESS statement was previously processed in the job.)

To create a direct access data set, the programmer places an ALLOC statement in his job deck. In this statement, he specifies the name of the data set and either of two types of volume designation.

The first type of volume designation is used when a programmer wants space allocated either on a fresh volume or on a particular volume identified by its volume serial number. It identifies the device to be used, either through a device address or through a device type code. In addition, it indicates the type of volume to be used. A particular volume is requested by specifying a volume identification (volid). The volid is the volume serial number in the disk's volume label. A fresh volume is requested by specifying the word FRESH in the volume designation.

The second type of volume designation is used when the programmer wants space allocated on a particular volume that already holds a specific data set. The programmer specifies the word SAME in the volume designation. He then identifies the data set either by specifying its name or by specifying the symbolic unit name with which it is currently associated.

Both types of volume designation allow the programmer to indicate whether or not write validity checking is to be performed for the data set. When write validity checking is performed, the system checks each block of data as it is written to see that it has been written correctly. Standard error recovery procedures are followed if an error is detected. The write checking procedure requires an additional disk revolution for each data block that is written.

The programmer must also indicate in the ALLOC statement the length of the data set. That is, he must specify the number of blocks that are to be allocated for the data set. The number of blocks is equal to the number of FORTRAN records in the data set.

The programmer can request that the data set be cataloged by specifying the CATLG parameter in the ALLOC statement. This causes the name of the data set, along with an indication of its location, to be placed into the system catalog.

Within a FORTRAN program, either sequential or direct access input/output statements can be used to transfer data to or from a direct access data set. If direct access statements (for example, the DEFINE FILE statement) have been used for the data set being created, the programmer must specify the FMT parameter in the ALLOC statement. This causes the system to prepare the disk area for direct access input/output operations.

If a directoried data set is being created, the length of the directory must also be specified in the ALLOC statement. The length of the directory is equal to the number of entries that are to be made in it, allowing one entry for each member name.

If a symbolic unit name is to be associated with the data set, the programmer can specify this name in the ALLOC statement. A symbolic unit name must be associated with a sequential data set before it can be used. For a directoried data set, a symbolic unit name is usually associated with each member of the data set, rather than with the entire data set.

The programmer must also include a LABEL statement in his job deck, immediately after the ALLOC statement. In the LABEL statement, he must specify the block length of the data set. The block length is the number of bytes in each FORTRAN record. This number cannot exceed 360 unless direct access input/output operations are to be performed on the data set. In this case, the block length specified for the data set in the LABEL statement should agree with the record length specified for the data set in the DEFINE FILE statement within the FORTRAN program.

The programmer can also specify the expiration date of the data set in the LABEL statement. The absence of this specification causes the system to assume that the current date is to be used, that is, that the data set is not to be retained after the date it is created.

Finally, the programmer can indicate whether or not write validity checking is to be performed for this data set. The specification given here can be overridden, however, by the write validity checking option in the ALLOC statement. In other words, the system acts in accordance with the specification in the ALLOC statement. If nothing is specified in the ALLOC statement, the system acts in accordance with the specification in the LABEL statement. If nothing is specified in either statement, no write validity checking is performed.

If the information to be given in the LABEL statement duplicates that given in the LABEL statement for another data set, the programmer need not repeat the information. This is true, however, only if the other data set is one for which an ALLOC or ACCESS statement was processed previously in the job. The programmer need only specify the word SAME in the LABEL statement and then identify the other data set. He can identify it either by specifying its name or by specifying the symbolic unit name with which it is currently associated.

Creating a Member of a Directoried Data Set

In addition to creating a directoried data set in the manner just described, the programmer must also create each member of the data set. Only one member can be created in a single job step. Whatever is written into the member during that job step determines the size of the member. Once the member is created, its size cannot be changed.

A member is given one or more unique names when it is created; the names are unique in that they may not duplicate any other member names in the data set. The number of names given to a member cannot be increased after the member has been created, although existing member names can be replaced by new names (this is explained in a later section, "Data Set Maintenance Statements").

A member of a directoried data set will be created only if there is space for it in the data set and if there is room in the directory for the entries required for that member.

To create a member, the programmer places an ACCESS statement in his job deck. In this statement, he specified the names to be given to the member, the name of the data set to which the member is to belong, and the symbolic unit name with which the member is to be associated.

The programmer must also indicate the location of the directoried data set to which the member is being added, unless its location is already known to the system. The location of the data set is indicated by a volume designation. The volume designation can be any of those used in the ALLOC statement to create a data set, with one exception. The ACCESS statement cannot indicate that the directoried data set resides on a fresh volume.

Finally, the programmer must specify the NEW parameter in the ACCESS statement to indicate that a new member is being created.

Using Existing Direct Access Data Sets

To use an existing direct access data set, the programmer places an ACCESS statement in his job deck. In this statement, he specifies the name of the data set, the symbolic unit name with which the data set is to be associated, and either of two types of volume designation. (The volume designation is not required for a cataloged data set because the system already has a record of this information.)

The first type of volume designation is used to request a volume through its volume serial number. It identifies the device to be used, either through a device address or through a device type code. It also specifies the volume identification (volid) of the disk containing the data set.

The second type of volume designation is used to request the same volume that contains another specific data set. The location of this other data set must be known to the system. The programmer specifies the word SAME in the volume designation. He then identifies the other data set, either by specifying its name or by specifying the symbolic unit name with which it is currently associated.

Both types of volume designation allow the programmer to indicate whether or not write validity checking is to be performed for the data set.

If the programmer is adding data to a sequential data set (rather than reading from it), he must also specify the EXT parameter in the ACCESS statement. This causes the disk volume to be positioned after the last item of data in the existing data set, rather than at the beginning of the data set. Adding data to a direct access data set does not affect the size of the data set. Additional data is limited to whatever amount can be contained in the extent that was defined for the data set at the time it was created.

Using Existing Members of a Directoried Data Set

A member of a directoried data set, once it has been created, cannot be enlarged; however, data within it can be manipulated freely or replaced. To use an existing member of a directoried data set, the programmer places an ACCESS statement in his job deck. In this statement, he specifies one name of the member, the name of the directoried data set to which the member belongs, and the symbolic unit name with which the member is to be associated.

The programmer must also indicate the location of the directoried data set to which the member belongs, unless its location is already known to the system. The location of the directoried data set is given by a volume designation. This can be either of the volume designations valid in the ACCESS statement for using an existing direct address data set (discussed in the previous section).

PLACING ALLOC AND ACCESS STATEMENTS IN THE JOB DECK

The ALLOC and ACCESS statements for data sets that are to be created or used during a job should be placed before the EXEC statement for the job step using the data sets. In most cases, this will be a phase execution job step. The programmer can place all of the ALLOC and ACCESS statements for a job in front of the first EXEC statement in the job deck. This means that the assignments made by the statements remain in effect throughout the entire job or until changed by a RESET statement (discussed in the next section, "Symbolic Unit Maintenance Statements").

SYMBOLIC UNIT MAINTENANCE STATEMENTS

Two job control statements, RESET and LISTIO, are used in conjunction with ALLOC and ACCESS statements that alter the assignments of system units.

The RESET statement is used to restore one or more symbolic units to their standard assignments. The statement is used when an assignment has been altered by an ALLOC or ACCESS statement in a previous job step. The RESET statement applies only to those units that were given standard assignments either when the system was constructed or when the operator performed an IPL procedure.

One RESET statement can be used to restore either all units with standard assignments or just one unit. If more than one unit is to be restored, but not all, a separate RESET statement is required for each. Rules for writing the RESET statement can be found in the chapter "Control Statements."

Regardless of whether RESET statements are used, all units are restored to their standard assignments at the end of the job.

The LISTIO statement is used to obtain a listing of current symbolic unit assignments. The listing, which is produced on SYSLST and on SYSLOG, includes the name of the symbolic unit, its current device address, the volume designation (volid) of the volume to which it is assigned, and the name of the data set currently associated with the symbolic unit.

Three types of listing can be obtained. The programmer can request a listing for a single unit by specifying its symbolic unit name in the LISTIO statement. He can request a listing of all assignments made or altered by ALLOC or ACCESS statements during the current job by specifying the word PROG in the LISTIO statement. (This listing does not include units already restored to their standard assignments as a result of RESET statements.) Finally, the programmer can request a listing for all units that have assignments by omitting any specification from the LISTIO statement.

Rules for writing the LISTIO statement can be found in the chapter "Control Statements."

DATA SET MAINTENANCE STATEMENTS

There are five job control statements used for the maintenance of data sets: CATLG, UNCATLG, DELETE, CONDENSE, and RENAME. These statements are intended primarily for use with direct access data sets, although the CATLG and UNCATLG statements can be used for other data sets.

Each of the data set maintenance statements is discussed here with respect to its function and use. Rules for writing these statements can be found in the chapter "Control Statements."

The CATLG statement is used to make an entry for a data set in the system catalog. A cataloged data set can be referred to by name only, without any need for stating its location. Catalog entries are retained until specifically deleted by an UNCATLG statement or until the data set is deleted.

The name of the data set to be cataloged may not duplicate the name of a data set already in the catalog. Catalog entries can also be made through use of the CATLG specification in the ALLOC statement that creates a data set.

The UNCATLG statement is used to delete a data set entry from the system catalog. Removal of the catalog entry does not change the data set itself or the volume containing it. The data set entry in the volume table of contents is also unaffected.

The DELETE statement is used to eliminate a data set or a member of a directoried data set. When a member has more than one entry in the directory (more than one member name), the DELETE statement can be used to remove one or more of the entries. The member continues to exist as long as it is represented by at least one entry in the directory.

When an entire data set is deleted, the system removes its entry from the volume table of contents (VTOC), updates one of the volume's space management labels to reflect the removal, and, if applicable, removes the entry for the data set from the system catalog.

The data set is not physically altered at this point. It cannot be referred to, however, and the system treats the space it occupies as vacant. The same applies to a member of a directoried data set when all its entries have been removed from the directory.

The space occupied by a deleted data set can be assigned to a new data set; the space occupied by a deleted member within a directoried data set, however, cannot be reassigned. The CONDENSE job control statement (described later) can be used to shift existing members toward the beginning of a directoried data set so that new members can be added at the end.

A separate DELETE statement is required for each data set that is to be deleted. Any number of the members of one directoried data set can be deleted with a single DELETE statement.

Any data set cited in a DELETE statement must have been referred to in an ALLOC or ACCESS statement processed previously in the job.

The CONDENSE statement is used to shift the contents of a directoried data set in order to fill space occupied by deleted members and directory entries. This space is treated as though it were empty. Existing members and directory entries are shifted toward the beginning of the data set to fill the space. The total size of the data set is not changed. Also, there is no change in the order in which the remaining members and entries appear.

After the data set has been condensed, all available space is at the end of the data set and at the end of the directory. New members may be added and new entries may be made in the directory.

Any data set cited in a CONDENSE statement must have been referred to in an ALLOC or ACCESS statement processed previously in the job.

The RENAME statement is used to change the name of a data set or the name of a member of a directoried data set. When a data set is renamed, the name is changed in the VTOC and, if applicable, in the system catalog. The name of a member is changed in the directory of the data set to which it belongs. Other names of that member, if any, are not affected.

The new name may not duplicate an existing name in the system catalog, volume table of contents, or data set directory. System data sets should not be renamed.

Any data set cited in a RENAME statement must have been referred to in an ALLOC or ACCESS statement processed previously in the job.

This chapter describes in greater detail the three types of job steps involved in processing a FORTRAN program. It describes the options available to the programmer for each process and refers to specifications in job control statements and linkage editor control statements. Once the reader has become familiar with the information presented here, he should be able to write control statements merely by referring to the next chapter, "Control Statements."

COMPILATION

Compilation is the execution of the FORTRAN IV compiler. The programmer requests compilation by placing in the job deck an EXEC statement that contains the program name FORTRAN (the name of the FORTRAN IV compiler). This is the EXEC FORTRAN statement.

Input to the compiler is a set of FORTRAN source statements, constituting either a main program or a subprogram. Source statements punched in either card code, Extended Binary-Coded-Decimal Interchange Code (EBCDIC) or Binary-Coded-Decimal Interchange Code (BCDIC), are acceptable. (Appendix B shows the EBCDIC and BCDIC card codes for each of the 49 characters that are valid in FORTRAN source statements.)

If any source statements are in the BCDIC card code, the programmer must specify BCD as a compiler option in the EXEC FORTRAN statement. Otherwise, the FORTRAN IV compiler assumes that all source statements for the compilation are punched in EBCDIC and, therefore, treats any BCD characters as invalid.

The FORTRAN source statements are read from SYSIPT. The job deck is read from SYSRDR. If SYSIPT and SYSRDR are assigned to the same unit, the FORTRAN source statements should be placed after the EXEC FORTRAN statement in the job deck.

Output from the FORTRAN IV compiler includes a source listing, a list of the source statements exactly as they appeared in the input deck. The source listing is produced on SYSOPT. Any errors in the source statements are indicated in the source listing and appropriate error messages are written. (The format of the source listing is discussed and illustrated in the chapter "System Output.") In addition, the module produced by the compiler is written on SYS000, the linkage editor input unit.

The programmer can override the production of any of this output by specifying compiler options in the EXEC FORTRAN statement. The NOSOURCE option suppresses the production of a source listing, except for the indication of errors. The NOLINK option suppresses the writing of the module on SYS000. The programmer should specify NOLINK in a compile-only job or whenever the module is to be excluded from linkage editor processing during the same job.

If a module is produced on SYS000, the programmer should name this module by specifying a name for the job step in the EXEC FORTRAN statement. The module name is the same as the job step name.

The programmer can request output in two additional forms, again via options in the EXEC FORTRAN statement. The compiler will produce a _module deck_ (the module, written on SYSPCH) if the programmer specifies DECK in the EXEC statement. The module deck can be used in a subsequent job as input to the linkage editor.

A compiler storage map is written on SYSOPT if the programmer specifies MAP in the EXEC statement. This storage map includes a list of all the variables (both local and COMMON variables) that were defined in the source statements just compiled. (The contents of the compiler storage map are discussed and illustrated in the chapter "System Output.")

EDITING

Editing is the execution of the linkage editor. The programmer requests editing by placing in the job deck an EXEC statement that contains the program name LNKEDT (the name of the linkage editor). This is the EXEC LNKEDT statement.

Input to the linkage editor is a set of linkage editor control statements and one or more modules to be edited. These modules include either or both of the following:

1. Modules that were compiled previously in the job and placed at that time on the linkage editor input unit, SYS000.

2. Modules that were compiled in a previous job and saved as module decks. The module decks must be placed on SYSIPT along with the linkage editor control statements.

In addition, the linkage editor will process modules that are in the <u>module library.</u> The module library is a collection of frequently used subprograms, such as the FORTRAN-supplied library subprograms, in the form of modules. The module library is on the unit named SYSREL.

Many FORTRAN programs contain references to FORTRAN-supplied library subprograms. Some references are explicit: for example, the statement B = SQRT(A) contains an explicit reference to the square root library subprogram, which computes, in this case, the square root of A. Other references are implicit: for example, the statement C = D**5 contains an implicit reference to the exponential library subprogram, which computes, in this case, the value of D raised to the fifth power

When the linkage editor processes a module that makes use of a library subprogram, it automatically searches the module library for the requested subprogram module and processes it along with the module that requested it. It is possible to suppress this automatic linking facility by specifying NOAUTO as an option in the EXEC LNKEDT statement. In doing so, the programmer accepts responsibility for ensuring that all library subprograms required by a FORTRAN program are included in linkage editor processing.

Output from the linkage editor is one or more phases. A phase may be an entire program or it may be part of a multiphase program.

A phase produced by the linkage editor can be executed immediately after it is produced (that is, in the job step immediately following the linkage editor job step). Or it can be executed later, either in a subsequent job step of the same job or in a subsequent job. In either of the latter cases, the programmer must specify KEEP as an option in the EXEC LNKEDT statement in order to retain the phase output. Otherwise, the phase output is retained only for the duration of one job step after the linkage editor job step.

In addition to the phase, the linkage editor produces a phase map on SYSLST. The contents of the phase map are discussed and illustrated in the chapter "System Output." The programmer can suppress the production of a phase map by specifying the NOMAP option in the EXEC LNKEDT statement.

<u>Linkage Editor Control Statements</u>

Linkage editor control statements direct the execution of the linkage editor. Together with any module decks to be processed, they form the <u>linkage editor input deck</u>, which is read by the linkage editor from SYSIPT. If SYSIPT and SYSRDR are assigned to the same unit, the linkage editor input deck should be placed after the EXEC LNKEDT statement in the job deck.

There are three linkage editor control statements that may be used by the FORTRAN programmer: the MODULE statement, the PHASE statement, and the INCLUDE statement. The discussion of these statements in this chapter is limited to the function and use of each statement. The rules for writing each statement are given in a subsequent chapter, "Control Statements."

The MODULE statement is required whenever a module deck is included on SYSIPT in the linkage editor input deck. One MODULE statement must precede each module deck; each MODULE statement must specify a name for the module deck it precedes. The MODULE statements and their associated module decks must appear first in the linkage editor input deck; no other linkage editor control statements may precede them.

As soon as a MODULE statement has been processed, the module deck following it is copied onto the linkage editor input unit, SYS000. Thereafter, it is treated exactly as any modules already on that unit (that is, the modules placed there earlier by the FORTRAN IV compiler).

The PHASE statement is used to specify a name for the phase that is to be produced by the linkage editor and to indicate the origin of the phase, that is, the first main storage location that is to be occupied by the phase when it is loaded. For a single-phase program, the origin is specified as the letter S, which indicates the first main storage location available to a problem program.

The INCLUDE statement identifies a particular module for inclusion in a phase. There must be one INCLUDE statement for each module that is to be included (except for those subprogram modules in the module library that will be linked automatically); all of the INCLUDE statements for a particular phase must immediately follow the PHASE statement that names the phase. The order of the INCLUDE statements indicates the order in which modules are to be included in the phase.

Each INCLUDE statement must identify the module by name. For a module on SYS000 that was produced by the FORTRAN IV compiler earlier in the job, the module name is the name of the job step -- the step name -- specified in the EXEC FORTRAN statement. For a module on SYS000 that was copied by the linkage editor from a module deck on SYSIPT, the module name is the name specified in the MODULE statement.

The INCLUDE statement must also indicate the location of the module. If the module is on SYS000, the programmer must specify the letter L; if the module is in the module library, he must specify the letter R. An INCLUDE statement is required for modules in the module library if the modules have not been referred to in the source program or if the automatic linking facility has been suppressed.

The PHASE and INCLUDE statements can be omitted from the linkage editor input deck if all of the following conditions exist:

1. Only one phase is to be produced by the linkage editor.

2. All of the modules on SYS000, including any that are to be copied from module decks on SYSIPT, are to be included in the phase.

3. The modules are to be included in the phase in the order in which they appear on SYS000.

If the programmer omits the PHASE and INCLUDE statements, the linkage editor will generate these statements. The name of the phase will be the name of the first module included in the phase. The origin of the phase will be the first main storage location available to a problem program (equivalent to a specification of S).

Note that the programmer must omit both the PHASE and the INCLUDE statements if he wishes to use this feature. In other words, a PHASE statement in the linkage editor input deck must always be accompanied by a set of INCLUDE statements and vice versa.

PHASE EXECUTION

Phase execution is the execution of the problem program, for example, the program written by the FORTRAN programmer. If the program is a multiphase program, phase execution actually entails the execution of all the phases in the program.

The phase(s) to be executed must be in the phase library. The phase library is a collection of executable phases from which programs are loaded by the supervisor. A phase is written in the phase library by the linkage editor at the time the phase is produced. It is retained in the phase library if the programmer has so requested via the KEEP option in the EXEC LNKEDT statement.

The programmer requests the execution of a phase by placing in the job deck an EXEC statement that specifies the name of the phase. If the phase to be executed was produced in the immediately preceding job step, however, its name need not be specified in the EXEC statement.

The programmer can also request, via the EXEC statement, that the setting of the variable precision switch be checked. This switch, which is set manually by the operator, indicates the level of precision at which floating-point operations are performed. Precision may be 8, 10, 12, or 14 bits. In general, the highest precision provides greatest accuracy and the lowest precision provides greatest speed.

MULTIPHASE PROGRAMS

A FORTRAN program can be executed as a single phase as long as there is an area of main storage available to accommodate it. This area, known as the problem program area, must be large enough to contain the main program, all called subprograms (both library subprograms and those written by the user), and an area of common storage when applicable (whenever COMMON statements are used anywhere in the source program). When a program is too large to be executed as a single phase, it must be structured as a multiphase program.

A multiphase program may have either of two structures. The first of these is a complete phase overlay structure, permitted for a program of two or more phases. Only one phase of the program is in the problem program area at any given time, each phase completely replacing, or overlaying, the previous phase.

The other structure available for multiphase programs is known as root phase overlay and is used primarily for programs of three or more phases. One phase of the program is designated the root phase and, as such, remains in the problem program area throughout the execution of the entire program. The other phases in the program -- subordinate phases -- are loaded into the problem program area as they are needed. A subordinate phase may overlay any previously loaded subordinate phase, but, under ordinary circumstances, no subordinate phase should overlay the root phase. One or more subordinate phases can reside simultaneously in main storage with the root phase.

In order to choose the overlay structure best suited for his program, the programmer should examine the program for <u>subprogram structures</u>. A subprogram structure is a series of two or more subprograms, the first of which is called by the main program; the second subprogram is called by the first subprogram, the third is called by the second, and so on. For example, every FORTRAN main program contains a call to the library subprogram IBCOM; the IBCOM subprogram contains a call to the library subprogram FIOCS; in turn, FIOCS calls the library subprogram UNITAB. Thus, it can be said that every FORTRAN main program uses the subprogram structure consisting of IBCOM, FIOCS, and UNITAB. As a second example, consider the group of subprograms A, B, C, and D. Subprogram A contains a call to subprogram B, which, in turn, contains calls to subprograms C and D. In this example, two subprogram structures exist -- the first consisting of the subprograms A, B, and C, the other consisting of the subprograms A, B, and D.

The root phase overlay structure may be used whenever the problem program area is large enough to include the entire main program, the common area (when applicable), and the largest subprogram or subprogram structure used by the main program. Otherwise, the complete overlay structure must be used.

Allocation of COMMON by the Linkage Editor

For a multiphase program, the linkage editor allocates a common area equal in size to the largest common area required by any phase. The common area is present in main storage throughout the execution of the entire program. Parameters may be passed through the common area from one phase to another, making possible communication between phases.

Loading of Phases

When a multiphase program is to be executed, the first phase is loaded by the supervisor as a result of job control processing. The loading of subsequent phases, however, is controlled by the programmer. In doing so, the programmer makes use of a special library subprogram, BOAOVLY, provided expressly for multiphase programs. For each phase that is to be loaded, the programmer places in his source program a call to the BOAOVLY subprogram, which causes the appropriate phase to be loaded.

Since the calling statements differ, depending on the type of overlay structure being used, they are discussed in detail in the appropriate section, that is, "Complete Phase Overlay" or "Root Phase Overlay."

COMPLETE PHASE OVERLAY

The complete phase overlay structure requires that a FORTRAN main program be divided into two or more main programs, one for each phase of the multiphase program. Once the original main program has been divided by the programmer, each newly formed main program, together with the subprograms and subprogram structures it uses, is processed to form one phase of the new program.

For example, consider a FORTRAN main program that consists of 300 source statements and makes use of eight subprograms, named A through H. Assume that this main program can be divided into three parts of 100 statements each, so that all three parts make use of subprograms A, B, and C, only part 1 makes use of subprograms D and E, only part 2 makes use of subprograms F and G, and only part 3 makes use of subprogram H. The result is a three-phase program: the first phase includes part 1, as the main program, and subprograms A, B, C, D, and E; the second phase includes part 2, as the main program, and subprograms A,B,C,F, and G; the third phase includes part 3, as the main program, and subprograms A,B,C, and H.

Calling Statement for Complete Phase Overlay

To request that a new phase be loaded, the programmer must place the following CALL statement in his source program:

```
CALL LINK ('phasename')
```

This statement causes the phase whose name is specified to be loaded into the problem program area. In addition, control is given to the newly loaded phase, which then begins execution.

The phase name specified in the CALL statement must be the name of the phase as specified in a linkage editor PHASE statement.

Since the CALL LINK statement causes control to be transferred to a new phase, it should appear as the last executable statement in each phase except the last.

The following illustrates the CALL LINK statement:

```
    CALL LINK ('PHASEC')
```

This statement results in the loading of PHASEC by the supervisor and the transfer of control to PHASEC.

Linkage Editor Control Statements

Linkage editor control statements for a multiphase program using complete phase overlay are specified exactly as they would be for a single-phase program. The linkage editor input deck differs in that there must be one PHASE statement for each phase in the program. Each PHASE statement must specify a unique phase name; as in the case of a single-phase program, the origin of each phase should be specified by the letter S. A set of INCLUDE statements must follow each PHASE statement to indicate which modules are to be included in the phase.

The first PHASE statement in the linkage editor input deck identifies the phase that is to be loaded and executed first, unless the programmer explicitly specifies the name of another phase in the EXEC statement for phase execution. For example, with the following set of control statements, PHASEA would be executed first:

```
    //  EXEC       LNKEDT
        PHASE      PHASEA,S
        INCLUDE    MOD1,L
        INCLUDE    MOD2,L
        PHASE      PHASEB,S
        INCLUDE    MOD3,L
    /*
    //  EXEC
```

However, the last statement could have been written:

```
    //  EXEC       PHASEB
```

In this case, PHASEB would be loaded and executed first.

ROOT PHASE OVERLAY

The root phase overlay structure requires that the entire FORTRAN main program be included in a root phase, together with some of the subprograms it uses. The remaining subprograms are incorporated into two or more subordinate phases, so that the root phase and the largest subordinate phase can reside in the problem program area simultaneously.

The programmer can construct subordinate phases of several levels. A first-level subordinate phase is one that is loaded as the result of a call from the root phase; the origin of such a phase usually is the first available location following the root phase. A second-level subordinate phase is one that is loaded as the result of a call from a first-level phase; its origin usually is the first available location following the first-level phase. A third-level subordinate phase is one that is loaded as the result of a call from a second-level phase, and so on. When phases of several levels are used, the root phase and the largest subordinate phase structure -- a series of two or more levels of subordinate phases -- may not exceed the size of the problem program area.

Figure 2 gives an example of a root phase overlay structure in the problem program area. In this illustration, ROOT is the root phase; A, B, and C are first-level subordinate phases; AA and CC are second-level phases. Two subordinate phase structures exist. One consists of phases A and AA; the other is made up of phases C and CC.

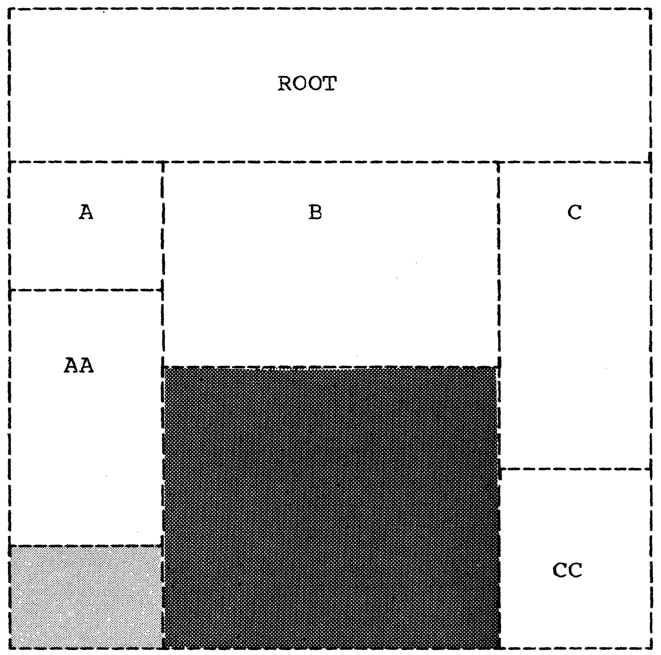


Figure 2. Root Phase Overlay Structure

The programmer is free to structure his subordinate phases in the way that best suits the needs of his program.

Calling Statement for Root Phase Overlay

To request that a new phase be loaded, the programmer must place the following CALL statement in his source program:

```
CALL LOAD ('phasename')
```

This statement causes the phase whose name is specified to be loaded into the problem program area. However, control returns to the next statement in the calling phase; it is not transferred to the newly loaded phase.

The phase name specified in the CALL statement must be the name of the phase as specified in a linkage editor PHASE statement.

After the requested phase has been loaded, the programmer can use any subprogram within it by means of a CALL statement addressing that subprogram. For example, consider a first-level subordinate phase ALPHA incorporating the subprograms BETA and GAMMA. The following sequence of statements in the root phase will cause phase ALPHA to be loaded and subprogram GAMMA to be executed:

```
CALL LOAD ('ALPHA')

CALL GAMMA (X,Y,Z)
```

Note that it is permissible to pass arguments (represented here by X, Y, and Z) from one phase to a subprogram in another phase. Once the called subprogram has been executed in the normal fashion, return is made to the calling phase (in the above example, from phase ALPHA to the root phase).

Linkage Editor Control Statements

There must be one PHASE statement in the linkage editor input deck for each phase of a multiphase program using root phase overlay. Each PHASE statement must specify a unique phase name. The origin of each phase is specified as follows:

1. The word ROOT is specified for the origin of the root phase. This causes the phase to be loaded at the first available location in the problem program area. The specification ROOT differs from the specification S in that it identifies the root phase to the linkage editor.

2. The character * (asterisk) can be specified to set the origin of a subordinate phase at the first location following the most recently processed phase. For example, assume that the first PHASE statement in the deck refers to the root phase; accordingly, its origin is specified by ROOT. Assume that the next PHASE statement refers to a first-level subordinate phase named ALPHA. The origin of ALPHA should be specified by * to cause it to be loaded into the area immediately following that occupied by the root phase. If the next PHASE statement refers to a second-level subordinate phase named BETA that is called by phase ALPHA, the origin of BETA should also be specified by * to cause it to follow phase ALPHA in storage.

3. The name of a phase currently in the phase library (this includes all phases previously created in this job step) can be specified to set the origin of the current phase equal to the origin of the phase whose name is specified. For example, consider again the linkage editor input deck discussed in point 2, above. Assume that the next PHASE statement (after the PHASE statement for BETA) refers to another first-level subordinate phase named GAMMA. Phase GAMMA should have the same origin as phase ALPHA, namely, the first available location following the root phase. This can be accomplished by specifying the phase name ALPHA as the origin in the PHASE statement for GAMMA.

If phase GAMMA calls a second-level subordinate phase, named DELTA, the PHASE statement for DELTA should be the next PHASE statement in the linkage editor input deck. Its origin should be specified by *, which loads DELTA at the first location following GAMMA. Note that the specification BETA, the name of the second-level phase called by ALPHA, should not be used. The origin of BETA follows ALPHA; the origin of DELTA should follow GAMMA. If GAMMA is longer than ALPHA, the specification BETA would cause DELTA to overlay part of GAMMA.

If phase GAMMA calls another second-level phase named ETA, its PHASE statement should be the next PHASE statement in the linkage editor input deck. The origin of ETA can be specified by DELTA, since ETA and DELTA are both second-level phases called by GAMMA and should have the same origin.

From the examples given thus far, it can be seen that phases should be processed in a given order. The root phase should be processed first, followed by a first-level subordinate phase, followed by a second-level phase, if any, and so on. If a program is to be structured as shown in Figure 3, the order in which these phases should be processed and the origin that should be specified for each is:

Phase	Origin
ROOTPH	ROOT
A	*
AA	*
AAA	*
AAB	AAA
AB	AA
B	A
BB	*
BC	BB
C	A or B
CC	*
D	A or B or C

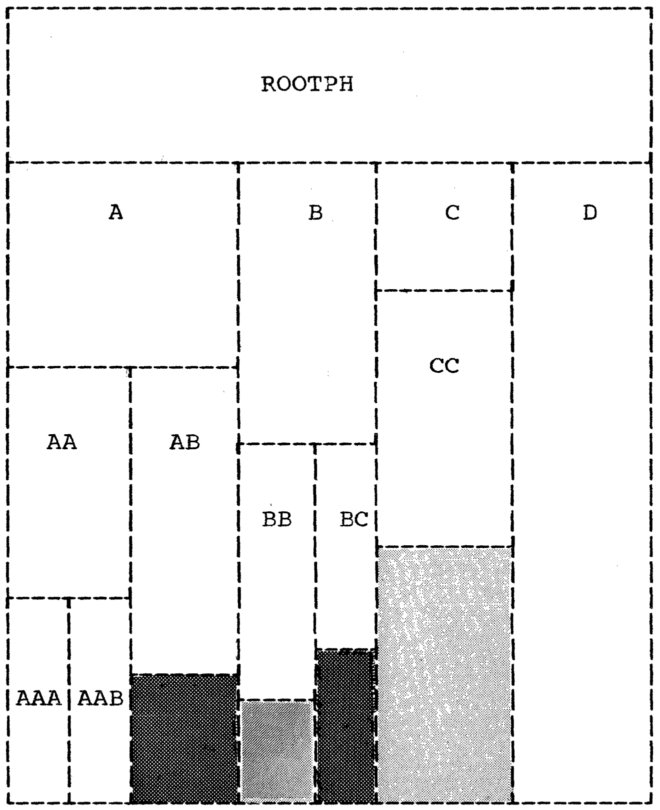


Figure 3. Order of Phases

LINKAGE EDITOR OPERATION

To the linkage editor each module it processes is a control section (CSECT). Each CSECT has a name -- the name of every CSECT that is a FORTRAN main program is MAIN44; the name of every subprogram CSECT is the subprogram name followed by an equal sign. For example, the CSECT name for the subprogram SUBPRO is SUBPRO=.

The linkage editor processes control sections according to the following rules:

1. If a CSECT name matches the name of another CSECT in the same phase or in the root phase, the new CSECT is not included in the current phase. For example, an attempt to include two main programs (both have the CSECT name MAIN44) in one phase causes the second main program to be ignored.

2. If a CSECT name matches the name of a CSECT in another phase (except the root phase), the new CSECT is included in the current phase but a warning message is issued. The message is numbered KA02I. (This does not hold true when the new CSECT is one automatically linked from the module library.) An example of this occurs when a complete overlay multiphase program is processed. Each phase contains a main program with CSECT name MAIN44. The linkage editor prints the KA02I message for each main program it processes other than the first. However, in these instances the warning message can be ignored.

Named COMMON and BLOCK DATA Areas

It has already been mentioned that the linkage editor allocates a common area equal in size to the largest common area in any phase. All references to COMMON are resolved to this area except for references to a named COMMON of the same name as a BLOCK DATA area. All references to such a named COMMON are resolved to the BLOCK DATA area, which is within a phase.

This causes no problem when the complete phase overlay structure is used. However, for the root phase overlay structure, the danger exists that a reference to named COMMON will be resolved to a BLOCK DATA area, even though the phase containing the BLOCK DATA is not in main storage. For this reason, a BLOCK DATA area of the same name as a named COMMON should appear only in the root phase.

The Model 44 Programming System provides two types of control statements that can be used by the FORTRAN programmer: job control statements and linkage editor control statements. This chapter gives the rules for writing these control statements and describes each statement with respect to format and content.

JOB CONTROL STATEMENTS

Job control statements are designed for an 80-column punched card format. Although certain restrictions must be observed, the statements are essentially free form. Information must start in column 1 and cannot extend beyond column 71. If the length of a statement exceeds 71 characters, it may be continued on additional cards, as discussed later in this section.

A statement may consist of from one through four fields. The order of the fields in the statement are: the identifier field, the name field, the operation field, and the operand field.

The identifier field occupies card columns 1 and 2. It contains a two-character combination that identifies the statement as a job control statement. The identifier combination for most job control statements is //. The exceptions are /& for the end-of-job statement, /* for the end-of-data statement, and *ƀ (asterisk followed by a blank) for the comments statement.

The name field begins in column 3 and may not extend beyond column 10. The name field is permitted in only the JOB, EXEC, ALLOC, and ACCESS statements. If the name field of a statement is not used, column 3 must contain a blank.

The operation field, which identifies the statement by name (JOB, EXEC, etc.), may start in any column after column 3. If the statement has a name field, the operation field must be separated from the name field by at least one blank.

The operand field follows the operation field, separated from it by at least one blank. The operand field usually consists of a series of specifications, separated from each other by commas or parentheses. Except where otherwise indicated, specifications should be punched in the order shown in the statement formats. In general, no blanks are permitted within the operand field. The exception to this rule occurs when a blank character is permitted <u>within a specification</u>. Otherwise, the first blank in an operand field causes any characters following the blank and preceding column 72 to be treated as comments.

Column 72 in each card is the continuation column. A nonblank character in this column indicates that the statement is continued on the next card. The first card of a statement must contain the identifier field, the name field (if used), the operation field, and at least one specification of the operand field. The statement can be interrupted only after a comma used to separate two specifications.

It is not necessary to fill up a card before continuing the statement on a new card. The final comma may appear in any column before column 71; in this case, at least one blank must follow the comma and then comments may appear through column 71. The continuation character is punched in column 72.

As many continuation cards as necessary may be used for a single statement. There must be a nonblank character in column 72 of each card except the last. Each card must contain the characters // in columns 1 and 2. The operand field of the statement must always resume in column 16. If column 16 of any continuation card is blank, the text on it and on any subsequent continuation cards for the statement is treated as comments.

Columns 73 through 80 of all cards are ignored by the system and may be used for any purpose.

Comments in Job Control Statements

There are several ways in which comments can appear in job control statements. All such comments are printed on SYSLST.

As was already shown, comments can appear in job control statements that have an operand field. They are written after the operand field (or a portion of an operand field that is continued on another card) and separated from it by at least one blank. Comments can also be written as a series of continuation cards, the first of which has a blank in column 16.

For statements in which an operand field is permitted but is not being used, the absence of the field must be indicated by a comma and at least one blank before the start of any comments.

Comments are also permitted in statements that do not have an operand field, such as the end-of-job (/&) statement, as long as the comments are preceded by at least one blank. Continuation cards may not be used, however, to extend these comments.

Comments statements may be placed anywhere in the job deck. Column 1 must contain an asterisk; column 2 must contain a blank; the remainder of the card, up to column 72, may contain any characters, including blanks. Comments statements are designed for communication with the operator; accordingly, they are written on the console printer-keyboard, SYSLOG, as well as being written on SYSLST.

Character Set

Statements may contain any of 39 alphameric characters recognized by the programming system. The term "alphameric characters" refers to both alphabetic and numeric characters.

Alphabetic characters are defined for the system as the 26 letters of the alphabet, A through Z, plus 3 special characters: $ # @.

The numeric characters are the digits 0 through 9.

In addition to the 39 alphameric characters, the following characters may appear in job control statements, but only where specifically indicated in the statement formats:

asterisk	*
comma	,
equal sign	=
parentheses	( )
single quote	'
slash	/

All job control statements must be punched in the Extended Binary-Coded-Decimal Interchange Code (EBCDIC).

Statement Formats

The job control statements are presented in this chapter in alphabetic order. For each statement, the statement format appears first, showing the contents of the identifier, name, operation, and operand fields. Immediately following each statement format is a specifications table, which indicates for each specification in the statement format the reason for specifying it and how to specify it.

An attempt has been made to keep each statement format as simple as possible. For some statements, more complex specifications in the operand field are dealt with in additional tables, one for each of these more complex specifications. In all cases, the reader is directed to the appropriate table in the specifications table following the statement format.

The following notation is used in the statement formats:

1. All upper-case letters represent specifications that are to appear in the actual statement exactly as shown in the statement format. For example, JOB in the operation field of the JOB statement should be punched exactly as shown -- JOB

2. All lower-case letters represent generic terms that are to be replaced in the actual statement. For example, jobname is a generic term that should be replaced by the name that the programmer is giving his job.

3. Hyphens are used to join two or more words in order to form a single generic term. For example, data-length is one generic term.

4. Brackets are used to indicate that a specification is optional and is not always required in the statement. For example, [CATLG] means that the word CATLG may or may not appear in the statement, depending on the programmer's requirements.

5. Braces enclosing stacked items indicate that a choice of one item <u>must</u> be made by the programmer. For example:

$$\left\{\begin{matrix}2400\\1600\end{matrix}\right\}$$

means that either 2400 or 1600, but not both, <u>must</u> appear in the actual statement.

6. Brackets enclosing stacked items indicate that a choice of one item may, but need not, be made by the programmer. For example:

$$\begin{bmatrix}\text{DECK}\\ \text{NODECK}\end{bmatrix}$$

means that either DECK or NODECK, but not both, may appear in the actual statement, or the specification can be omitted entirely.

7. An underlined item represents the default option -- the choice that will be made by the programming system if the programmer omits a specification. For example:

$$\begin{bmatrix}\text{NOSOURCE}\\ \underline{\text{SOURCE}}\end{bmatrix}$$

means that either NOSOURCE or SOURCE, but not both, may appear in the actual statement, or the specification can be omitted entirely (in which case SOURCE is assumed by the programming system). In other words, specifying SOURCE produces the same result as omitting the specification entirely.

<u>Note:</u> The default options shown in this publication are those that exist in the distributed version of the Model 44 Programming System. However, these defaults can be altered by an installation during the system construction process or the system assembly process.

8. All punctuation marks shown in the statement formats other than hyphens, brackets, braces, and underlines are punched exactly as shown. For example, [,NOAUTO] means that the specification, if present in the statement, should consist of the seven characters ,NOAUTO so that the initial comma is included.

ACCESS Statement (Unit Record Data Sets)

Id	Name	Operation	Operand
//	SYSxxx	ACCESS	dsname, $\left\{\begin{array}{l}\text{type=}\\\text{devadr=}\end{array}\right\}$

Specification	Reason for Specifying	How to Specify
//	Required	As shown
SYSxxx	Required; associates the data set with a symbolic unit name	Any valid symbolic unit name
ACCESS	Required	As shown
dsname	Required; indicates the name of the data set	From one through eight alphameric characters, the first of which must be a letter
type=	To identify, through its device type code, the device to be used	One of the unit record device type codes (see next chart), followed by an equal sign
devadr=	To identify, through its device address, the device to be used	A three-character device address (supplied by the installation), followed by an equal sign

Unit Record Device Type Codes:

Code	Meaning
1442	IBM 1442-N1 Card Read-Punch
1442P	IBM 1442-N2 Card Punch
2520	IBM 2520 Card Read-Punch
2520P	IBM 2520-B2, B3 Card Punch
2501	IBM 2501 Card Reader
2540	IBM 2540 Card Read-Punch (Reader side)
2540P	IBM 2540 Card Read-Punch (Punch side)
1403	IBM 1403 Printer, Model 2, 3, or N1 (132 characters)
1403M7	IBM 1403 Printer, Model 7 (120 characters)
1443	IBM 1443 Printer, Model N1 (120 characters)
1443S	IBM 1443 Printer, Model N1 (144 characters) Special Feature

Note: Each code is specified as shown.

Example:

```
    //SYS004 ACCESS CARDDATA,1442=
```

This statement causes an IBM 1442-N1 Card Read Punch to be used for the data set named CARDDATA. The data set is associated with symbolic unit SYS004 (corresponding to data set reference number 4).

ACCESS Statement (Tape Data Sets)

Id	Name	Operation	Operand
//	SYSxxx	ACCESS	dsname,volume[,EXT]

Specification	Reason for Specifying	How to Specify
//	Required	As shown
SYSxxx	Required; associates the data set with a symbolic unit name	Any valid symbolic unit name
ACCESS	Required	As shown
dsname	Required; indicates the name of the data set	From one through eight alphameric characters, the first of which must be a letter; for labeled tapes, the data set name as contained in the data set label
volume	Required; identifies the device and volume to be used	The tape volume designation (see next chart)
EXT	Indicates that data is to be added to the data set	As shown

Tape Volume Designation:

```
{ type   }    [(options)]    { ='volid' }
{ devadr }                   { =        }
```

Specification	Reason for Specifying	How to Specify
type	To identify, through its device type code, the device to be used	One of the tape device type codes (see below)
devadr	To identify, through its device address, the device to be used	A three-character device address (supplied by the installation)
(options)	To specify tape options for tape units with device type codes (see below) 2400T7, 2400T7C, or 2400D	From one through four tape options (see next page), separated by commas; the list must be enclosed in parentheses
='volid'	To identify, through its volid, the tape volume to be used	From one through six characters (for labeled tapes, the volume serial number from the volume label), enclosed in single quotes; an equal sign must precede the first quote
=	To indicate that the tape has no volid; permitted for unlabeled tapes only	An equal sign

Tape Device Type Codes:

Code	Meaning
2400	IBM 2400 Magnetic Tape Unit with 9-track read/write head; 800 bpi only
2400H	IBM 2400 Magnetic Tape Unit with 9-track read/write head; 1600 bpi only
2400D	IBM 2400 Magnetic Tape Unit with 9-track read/write head; dual density
2400T7	IBM 2400 Magnetic Tape Unit with 7-track read/write head
2400T7C	IBM 2400 Magnetic Tape Unit with 7-track read/write head and the Convert Feature

Note: Each code is specified as shown.

Tape Options:

For tape units of device type code 2400T7: $\begin{bmatrix}200\\556\\\underline{800}\end{bmatrix} \begin{bmatrix}E\\\underline{O}\end{bmatrix} \begin{bmatrix}T\\\underline{NT}\end{bmatrix}$

For tape units of device type code 2400T7C: $\begin{bmatrix}200\\556\\\underline{800}\end{bmatrix} \begin{bmatrix}E\\\underline{O}\end{bmatrix} \begin{bmatrix}T\\\underline{NT}\end{bmatrix} \begin{bmatrix}NC\\\underline{C}\end{bmatrix}$

For tape units of device type code 2400D: $\begin{bmatrix}\underline{800}\\1600\end{bmatrix}$

Option	Meaning
200	To indicate a tape density of 200 bpi
556	To indicate a tape density of 556 bpi
800	Default option; indicates a tape density of 800 bpi
1600	To indicate a tape density of 1600 bpi
E	To indicate even parity; should not be specified unless NC is specified
O	Default option; indicates odd parity
T	To indicate that the translate feature is to be used; should not be specified unless NC is specified
NT	Default option; indicates that the translate feature is not to be used
NC	To indicate that the convert feature is not to be used; required if either E or T is specified
C	Default option; indicates that the convert feature is to be used

Note: Options may appear in the option list in any order; each option is specified as shown.

Example:

```
//SYS004 ACCESS TAPEDATA,2400D(1600)='T7063'
```

This statement causes an IBM 2400 Magnetic Tape Unit with 9-track read/write head and dual density to be used for the data set named TAPEDATA. The tape density is 1600 bytes per inch. The data set is located on the volume whose volid is T7063; the data set is associated with symbolic unit SYS004 (corresponding to data set reference number 4).

ACCESS Statement (Direct Access Data Sets)

Id	Name	Operation	Operand
//	[SYSxxx]	ACCESS	dsname[(member names)][,volume][,EXT][,NEW]

Specification	Reason for Specifying	How to Specify
//	Required	As shown
SYSxxx	Associates the data set, or a member of a directoried data set, with a symbolic unit name (a member is associated if member names are specified in the operand field); may be omitted if no data transmission is intended for the data set (for example, the data set is to be deleted, condensed, or renamed subsequently in the job)	Any valid symbolic unit name
ACCESS	Required	As shown
dsname	Required; indicates the name of the data set	The name of the data set, as contained in the VTOC of the volume on which it is located
(member names)	For directoried data sets only; required when an existing member is to be used or when a new member is to be created; indicates one name of an existing member or one or more names of a new member	One or more member names, separated by commas; the list must be enclosed in parentheses; each member name consists of from one through eight alphameric characters, the first of which must be a letter
volume	To indicate the location of the data set; may be omitted for system data sets, cataloged data sets, or data sets specified in a previous ACCESS or ALLOC statement within the job	One of the disk volume designations (see next chart)
EXT	Indicates that data is to be added to the data set; not permitted if member names are specified	As shown
NEW	For directoried data sets only; required when a data set member is to be created	As shown

Disk Volume Designations:

To identify a volume through its volid:

$$\left\{\begin{array}{l}\text{type}\\ \text{devadr}\end{array}\right\} \left[\begin{array}{l}\text{(WRCHK)}\\ \text{(NOWRCHK)}\end{array}\right] \text{='volid'}$$

To identify a volume through another data set it contains:

$$\text{SAME} \left[\begin{array}{l}\text{(WRCHK)}\\ \text{(NOWRCHK)}\end{array}\right] \left\{\begin{array}{l}\text{=dsname}\\ \text{=SYSxxx}\end{array}\right\}$$

Specification	Reason for Specifying	How to Specify
type	To identify, through its device type code, the device to be used	One of the direct access device type codes (see below)
devadr	To identify, through its device address, the device to be used	A three-character device address (supplied by the installation)
(WRCHK)	To indicate that write validity checking is to be performed for the data set	As shown, enclosed in parentheses
(NOWRCHK)	To indicate that write validity checking is not to be performed for the data set	As shown, enclosed in parentheses
='volid'	To identify the disk volume that contains the data set; required if type or devadr is specified	The volume serial number from the volume label, enclosed in single quotes; an equal sign must precede the first quote
SAME	Required when the volume is being identified through another data set it contains	As shown
=dsname	To identify the other data set by name	An equal sign followed by the name of the other data set
=SYSxxx	To identify the other data set through the symbolic unit name currently associated with it	An equal sign followed by the symbolic unit name associated with the data set

Direct Access Device Type Codes:

Code	Meaning
SDSD	Single Disk Storage Drive (2315 Disk Cartridge)
1316	IBM 1316 Disk Pack mounted on an IBM 2311 Disk Storage Drive

Note: Each code is specified as shown.

ALLOC Statement (Tape Data Sets)

Id	Name	Operation	Operand
//	SYSxxx	ALLOC	dsname,volume[,CATLG]

Specification	Reason for Specifying	How to Specify
//	Required	As shown
SYSxxx	Required; associates the data set with a symbolic unit name	Any valid symbolic unit name
ALLOC	Required	As shown
dsname	Required; indicates the name of the data set	From one through eight alphameric characters, the first of which must be a letter
volume	Required; identifies the device and volume to be used	The tape volume designation (see next chart)
CATLG	To enter the data set into the system catalog	As shown

Tape Volume Designation:

```
{type  }  [(options)]  {='volid'}
{devadr}               {=FRESH  }
```

Specification	Reason for Specifying	How to Specify
type	To identify, through its device type code, the device to be used	One of the tape device type codes (see below)
devadr	To identify, through its device address, the device to be used	A three-character device address (supplied by the installation)
(options)	To specify tape options for tape units with device type codes (see below) 2400T7, 2400T7C, or 2400D	From one through four tape options (see next page), separated by commas; the list must be enclosed in parentheses
='volid'	To identify, through its volid, the tape volume to be used	From one through six characters (for labeled tapes, the volume serial number from the volume label), enclosed in single quotes; an equal sign must precede the first quote
=FRESH	To indicate that a fresh tape volume is to be used	As shown, preceded by an equal sign

Tape Device Type Codes:

Code	Meaning
2400	IBM 2400 Magnetic Tape Unit with 9-track read/write head; 800 bpi only
2400H	IBM 2400 Magnetic Tape Unit with 9-track read/write head; 1600 bpi only
2400D	IBM 2400 Magnetic Tape Unit with 9-track read/write head; dual density
2400T7	IBM 2400 Magnetic Tape Unit with 7-track read/write head
2400T7C	IBM 2400 Magnetic Tape Unit with 7-track read/write head and the Convert Feature

Note: Each code is specified as shown.

Tape Options:

For tape units of device type code 2400T7: $\begin{bmatrix}200\\556\\\underline{800}\end{bmatrix} \begin{bmatrix}E\\\underline{O}\end{bmatrix} \begin{bmatrix}T\\\underline{NT}\end{bmatrix}$

For tape units of device type code 2400T7C: $\begin{bmatrix}200\\556\\\underline{800}\end{bmatrix} \begin{bmatrix}E\\\underline{O}\end{bmatrix} \begin{bmatrix}T\\\underline{NT}\end{bmatrix} \begin{bmatrix}NC\\\underline{C}\end{bmatrix}$

For tape units of device type code 2400D: $\begin{bmatrix}\underline{800}\\1600\end{bmatrix}$

Option	Meaning
200	To indicate a tape density of 200 bpi
556	To indicate a tape density of 556 bpi
800	Default option; indicates a tape density of 800 bpi
1600	To indicate a tape density of 1600 bpi
E	To indicate even parity; should not be specified unless NC is specified
O	Default option; indicates odd parity
T	To indicate that the translate feature is to be used; should not be specified unless NC is specified
NT	Default option; indicates that the translate feature is not to be used
NC	To indicate that the convert feature is not to be used; required if either E or T is specified
C	Default option; indicates that the convert feature is to be used

Note: Options may appear in the option list in any order; each option is specified as shown.

Example:

```
//SYS003 ALLOC NEWDATA,2400T7C(556)=FRESH
```

The statement causes an IBM 2400 Magnetic Tape Unit with a 7-track read/write head and the convert feature to be used for the data set named NEWDATA. The tape density is 556 bytes per inch; default options indicate odd parity, the nonuse of the translate feature, and the use of the convert feature. The data set is assigned to a fresh tape volume and associated with symbolic unit SYS003 (corresponding to data set reference number 3).

ALLOC Statement (Direct Access Data Sets)

Id	Name	Operation	Operand
//	[SYSxxx]	ALLOC	dsname[,volume],data length[,directory length][,CATLG][,FMT]

Specification	Reason for Specifying	How to Specify
//	Required	As shown
SYSxxx	To associate the data set with a symbolic unit name	Any valid symbolic unit name
ALLOC	Required	As shown
dsname	Required; indicates the name of the data set	From one through eight alphameric characters, the first of which must be a letter
volume	Identifies the device and/or volume on which space for the data set is to be allocated; required unless the data set is to be allocated on the sytem residence volume	One of the disk volume designations (see next chart)
data length	Required; indicates the number of blocks to be allocated for the data set	A decimal number from 1 through 65535
directory length	Required for a directoried data set only; indicates the number of entries in the directory, one for each member name	A decimal number from 1 through 65534
CATLG	To enter the data set into the system catalog	As shown
FMT	Required if FORTRAN direct access input/output operations are to be performed on the data set	

Disk Volume Designations:

To request a fresh volume or a volume having a particular volid:

$$\left\{\begin{array}{l}\text{type}\\ \text{devadr}\end{array}\right\}\left[\begin{array}{l}\text{(WRCHK)}\\ \text{(NOWRCHK)}\end{array}\right]\left\{\begin{array}{l}\text{=FRESH}\\ \text{='volid'}\end{array}\right\}$$

To request a volume that contains another particular data set:

$$\text{SAME}\left[\begin{array}{l}\text{(WRCHK)}\\ \text{(NOWRCHK)}\end{array}\right]\quad\left\{\begin{array}{l}\text{=dsname}\\ \text{=SYSxxx}\end{array}\right\}$$

Specification	Reason for Specifying	How to Specify
type	To identify, through its device type code, the device to be used	One of the direct access device type codes (see next chart)
devadr	To identify, through its device address, the device to be used	A three-character device address (supplied by the installation)
(WRCHK)	To indicate that write validity checking is to be performed for the data set	As shown, enclosed in parentheses
(NOWRCHK)	To indicate that write validity checking is not to be performed for the data set	As shown, enclosed in parentheses
=FRESH	To indicate that a fresh disk volume is to be used	As shown, preceded by an equal sign
='volid'	To identify, through its volid, the disk volume to be used	The volume serial number from the volume label, enclosed in single quotes; an equal sign must precede the first quote
SAME	Required when a volume containing another particular data set is to be used	As shown
=dsname	To identify the other data set by name	An equal sign followed by the name of the other data set
=SYSxxx	To identify the other data set through the symbolic unit name currently associated with it	An equal sign followed by the symbolic unit name associated with the data set

Direct Access Device Type Codes:

Code	Meaning
SDSD	Single Disk Storage Drive (2315 Disk Cartridge)
1316	IBM 1316 Disk Pack mounted on an IBM 2311 Disk Storage Drive

Note: Each code is specified as shown.

Example:

```
//SYS002 ALLOC DISKDATA,1316(NOWRCHK)='D0036',50
```

This statement causes 50 blocks of space to be allocated on an IBM 1316 Disk Pack for the data set named DISKDATA. The disk pack has the volume identification D0036. No write checking is performed for the data set, which is associated with symbolic unit SYS002 (corresponding to data set reference number 2).

(Note: This statement must be immediately followed by a LABEL statement.)

CATLG Statement

Id	Name	Operation	Operand
//		CATLG	dsname[,volume]

Specification	Reason for Specifying	How to Specify
//	Required	As shown
CATLG	Required	As shown
dsname	Required; indicates the name of the data set to be entered into the system catalog	From one through eight alphameric characters, the first of which must be a letter; may not duplicate any data set name already in the catalog
volume	Indicates the location of the data set to the system; may be omitted for a system data set or a data set specified in a previous ALLOC or ACCESS statement within the job	The cataloging volume designation (see below)

Cataloging Volume Designation:

type[(options)]='volid'

Specification	Reason for Specifying	How to Specify
type	To identify the device containing the data set by its device type code	Any of the unit record, tape, or direct access device type codes listed for the ACCESS statement
(options)	To specify tape options or the write checking options for direct access devices	From one through four options, separated by commas; the list must be enclosed in parentheses (see the ACCESS statement for permissible options)
='volid'	To identify, through its volid, the volume containing the data set	The volume serial number, enclosed in single quotes; an equal sign must precede the first quote

Example:

```
    // CATLG DISKDATA,1316(NOWRCHK)='D0036'
```

This statement causes an entry for the data set named DISKDATA to be placed in the system catalog. The data set is located on an IBM 1316 Disk Pack with volume identification D0036. No write checking is to be performed for the data set.

CONDENSE Statement

Id	Name	Operation	Operand
//		CONDENSE	dsname

Specification	Reason for Specifying	How to Specify
//	Required	As shown
CONDENSE	Required	As shown
dsname	Required; indicates the name of the directoried data set to be condensed	The name of the data set as contained in the VTOC of the volume on which it is located

Example:

```
// CONDENSE DRCTRYB
```

This statement causes the directoried data set named DRCTRYB to be condensed. After condensing, all space in the data set follows the data set; all space in the directory follows the last entry in the directory.

DELETE Statement

Id	Name	Operation	Operand
//		DELETE	dsname[(member names)]

Specification	Reason for Specifying	How to Specify
//	Required	As shown
DELETE	Required	As shown
dsname	Required; indicates the name of the data set that is to be deleted or from which one or more member names are to be deleted	The name of the data set as contained in the VTOC of the volume on which it is located
(member names)	For directoried data sets only; to delete one or more member names from a data set (deleting all the names of a particular member deletes the member)	One or more member names, separated by commas; the list must be enclosed in parentheses; each member name must appear exactly as specified in the ACCESS or RENAME statement that assigned the name to the member

Example:

```
     // DELETE DISKDATA
```

This statement causes the data set named DISKDATA to be deleted from the volume on which it is located. Its name is removed from the volume table of contents (VTOC) and from the system catalog, if applicable. (Note: This statement must be preceded in the job deck by an ALLOC or ACCESS statement that refers to DISKDATA.)

EXEC Statement (FORTRAN)

Id	Name	Operation	Operand
//	[stepname]	EXEC	FORTRAN[(parameter list)][,(VPSnn)][,accounting information]

Specification	Reason for Specifying	How to Specify
//	Required	As shown
stepname	To name the job step; required to name the module produced by the compiler, unless NOLINK is specified in the parameter list	From one through eight alphameric characters, the first of which must be a letter
EXEC	Required	As shown
FORTRAN	Required	As shown
(parameter list)	To specify compiler options	From one through five parameters (see next chart), separated by commas; the list must be enclosed in parentheses
(VPSnn)	To ensure that the variable precision switch is set to the value nn	One of the following, enclosed in parentheses: VPS14 VPS10 VPS12 VPS08
accounting information	To satisfy any installation requirement	From 1 through 16 alphameric characters, the first of which must be other than a left parenthesis or a blank

Parameters:

$$\begin{bmatrix}\text{DECK}\\ \underline{\text{NODECK}}\end{bmatrix} \begin{bmatrix}\text{NOSOURCE}\\ \underline{\text{SOURCE}}\end{bmatrix} \begin{bmatrix}\text{NOLINK}\\ \underline{\text{LINK}}\end{bmatrix} \begin{bmatrix}\text{BCD}\\ \underline{\text{EBCDIC}}\end{bmatrix} \begin{bmatrix}\text{MAP}\\ \underline{\text{NOMAP}}\end{bmatrix}$$

Parameter	Reason for Specifying
DECK	To produce a module deck on SYSPCH
NODECK	Default option -- no deck produced
NOSOURCE	To suppress production of a source listing on SYSOPT
SOURCE	Default option -- source listing produced on SYSOPT
NOLINK	To suppress the writing of the module on SYS000, the linkage editor input unit
LINK	Default option -- module written on SYS000
BCD	Required if any source statements are punched in BCDIC
EBCDIC	Default option -- source statements are punched in EBCDIC
MAP	To produce a compiler storage map on SYSLST
NOMAP	Default option -- no compiler storage map produced

Note: Parameters may appear in the parameter list in any order; each parameter is specified as shown.

EXEC Statement (LNKEDT)

Id	Name	Operation	Operand
//	[stepname]	EXEC	LNKEDT[(parameter list)][,accounting information]

Specification	Reason for Specifying	How to Specify
//	Required	As shown
stepname	To name the job step	From one through eight alphameric characters, the first of which must be a letter
EXEC	Required	As shown
LNKEDT	Required	As shown
(parameter list)	To specify linkage editor options	From one through three parameters (see below), separated by commas; the list must be enclosed in parentheses
accounting information	To satisfy any installation requirement	From 1 through 16 alphameric characters, the first of which must be other than a left parenthesis or a blank

Parameters:

$\begin{bmatrix}\text{KEEP}\\ \underline{\text{NOKEEP}}\end{bmatrix}\begin{bmatrix}\text{NOMAP}\\ \underline{\text{MAP}}\end{bmatrix}$ [NOAUTO]

Parameter	Reason for Specifying
KEEP	To retain the phase output produced by the linkage editor; required if phase execution is desired subsequent to the job step immediately following the linkage editor job step
NOKEEP	Default option -- phase output is discarded at the end of the job step immediately following the linkage editor job step
NOMAP	To suppress the production of a phase map on SYSLST
MAP	Default option -- phase map produced on SYSLST
NOAUTO	To suppress the automatic linking facility of the linkage editor during this job step

Note: Parameters may appear in the parameter list in any order; each parameter is specified as shown.

EXEC Statement (Phase)

Id	Name	Operation	Operand
//	[stepname]	EXEC	[phasename][,(VPSnn)][,accounting information]

Specification	Reason for Specifying	How to Specify
//	Required	As shown
stepname	To name the job step	From one through eight alphameric characters, the first of which must be a letter
EXEC	Required	As shown
phasename	To identify the phase that is to be executed; may be omitted if phase was produced by the linkage editor in the immediately preceding job step	The name of the phase, exactly as specified on the PHASE card used at the time the phase was created
(VPSnn)	To ensure that the variable precision switch is set to the value nn	One of the following, enclosed in parentheses: VPS14 VPS10 VPS12 VPS08
accounting information	To satisfy any installation requirement	From 1 through 16 alphameric characters, the first of which must be other than a left parenthesis or a blank

JOB Statement

Id	Name	Operation	Operand
//	[jobname]	JOB	{DUMP / NODUMP} [,accounting information]

Specification	Reason for Specifying	How to Specify
//	Required	As shown
jobname	To name the job	From one through eight alphameric characters, the first of which must be a letter
JOB	Required	As shown
DUMP	To produce a dump if the program terminates abnormally; the contents of main storage and of the general registers are written on SYSLST	As shown
NODUMP	Default option -- no dump produced	As shown
accounting information	To satisfy any installation requirement	From 1 through 16 alphameric characters, the first of which must be other than a left parenthesis or a blank

LABEL Statement

Id	Name	Operation	Operand
//		LABEL	$\left[\begin{array}{l}\text{label-information}\\ \text{SAME}\left\{\begin{array}{l}\text{=dsname}\\ \text{=SYSxxx}\end{array}\right\}\end{array}\right]$

Specification	Reason for Specifying	How to Specify
//	Required	As shown
LABEL	Required	As shown
label-information	To provide label information for a data set; required for direct access data sets unless SAME is specified	Label specifications (see next chart)
SAME	To indicate that the label information for a data set duplicates the information already given for another data set	As shown
=dsname	To identify the other data set by name	An equal sign followed by the name of the other data set
=SYSxxx	To identify the other data set through the symbolic unit name currently associated with it	An equal sign followed by the symbolic unit name associated with the other data set

Label Specifications:

```
[block-length][,expiration-date] [,WRCHK  ]
                                 [,NOWRCHK]
```

Specification	Reason for Specifying	How to Specify
block-length	Required for direct access data sets; indicates the number of bytes in a FORTRAN record	Either a decimal number from 1 through 360 or a number equal to the number specified for record length in a DEFINE FILE statement within the FORTRAN program
expiration-date	To specify the date on which the data set may be deleted; otherwise, the current date is used as the expiration date	The date in the form yyddd, where yy (two digits from 00 through 99) represents the year and ddd (three digits from 001 through 366) represents the day of the year
WRCHK	To indicate that write checking is to be performed on a direct access data set; can be overridden by a specification of NOWRCHK in an ALLOC or ACCESS statement for the data set	As shown
NOWRCHK	To indicate that write checking is not to be performed on a direct access data set; can be overridden by a specification of WRCHK in an ALLOC or ACCESS statement for the data set	As shown

LISTIO Statement

Id	Name	Operation	Operand
//		LISTIO	$\left[\begin{array}{l}\text{PROG}\\ \text{SYSxxx}\end{array}\right]$

Specification	Reason for Specifying	How to Specify
//	Required	As shown
LISTIO	Required	As shown
PROG	To limit the list of current unit assignments to only those assignments made or altered during the current job	As shown
SYSxxx	To request that the current assignment of a single symbolic unit be listed	The name of the symbolic unit whose current assignment is to be listed

RENAME Statement

Id	Name	Operation	Operand
//		RENAME	$\left\{\begin{array}{l}\text{old-dsname, new-dsname}\\\text{dsname (old-member-name, new-member-name)}\end{array}\right\}$

Specification	Reason for Specifying	How to Specify
//	Required	As shown
RENAME	Required	As shown
old-dsname	To indicate the data set whose name is to be changed	The name of the data set as it appears in the volume table of contents
new-dsname	To specify the new name for a data set whose name is to be changed	From one through eight alphameric characters, the first of which must be a letter
dsname	To indicate the name of a directoried data set containing a member whose name is to be changed	The name of the data set as it appears in the volume table of contents
old-member-name	To indicate the member name that is to be changed	The name of the member as it appears in the directory
new-member-name	To indicate the new name of the member whose name is to be changed	From one through eight alphameric characters, the first of which must be a letter

RESET Statement

Id	Name	Operation	Operand
//		RESET	[SYSxxx]

Specification	Reason for Specifying	How to Specify
//	Required	As shown
RESET	Required	As shown
SYSxxx	To indicate the unit whose assignment is to be restored; the absence of this specification causes all units with standard assignments to be restored	The symbolic unit name of any unit having a standard assignment

UNCATLG Statement

Id	Name	Operation	Operand
//		UNCATLG	dsname

Specification	Reason for Specifying	How to Specify
//	Required	As shown
UNCATLG	Required	As shown
dsname	Required; indicates the name of the data set to be removed from the system catalog	The name of the data set as it was entered into the system catalog

LINKAGE EDITOR CONTROL STATEMENTS

Linkage editor control statements consist of only two fields -- an operation field and an operand field. Both fields are required.

The operation field, which identifies the statement by name, may start in any column after column 1. The operand field follows the operation field, separated from it by at least one blank. The operand field consists of from one through three specifications, separated from each other by commas. Specifications must be punched in the order shown in the statement formats. No blanks are permitted within the operand field.

Linkage editor control statements may not be continued; all information must be punched in one card. Comments may be written in the statements; they must be separated from the last character of the operand field by at least one blank and must not extend beyond column 71.

Character Set

In addition to the 39 alphameric characters permitted in job control statements, linkage editor control statements allow the use of the characters comma and asterisk, but only where specifically indicated in the statement formats.

All linkage editor control statements must be punched in the Extended Binary-Coded-Decimal Interchange Code (EBCDIC).

Statement Formats

The linkage editor control statements are presented here in alphabetic order. For each statement, the statement format appears first, showing the contents of the operation and operand fields. Immediately following each statement format is a specifications table, which indicates for each statement format specification the reason for specifying it and how to specify it.

The notation used in these statement formats is the same as that used for the job control statements.

INCLUDE Statement

Operation	Operand
INCLUDE	module, $\left\{\begin{matrix} L \\ R \end{matrix}\right\}$

Specification	Reason for Specifying	How to Specify
INCLUDE	Required	As shown
module	Required to identify the module that is to be included in the phase	The name of the module as it appears in a MODULE statement or in the name field of an EXEC FORTRAN statement
L	To indicate that the module to be processed can be found on SYS000	As shown
R	To indicate that the module to be processed can be found in the module library	As shown

MODULE Statement

Operation	Operand
MODULE	name

Specification	Reason for Specifying	How to Specify
MODULE	Required	As shown
name	Required; indicates the name of the module	From one through eight alphameric characters, the first of which must be a letter

PHASE Statement

Operation	Operand
PHASE	phasename, { S | * | ROOT | phase } [,NOAUTO]

Specification	Reason for Specifying	How to Specify
PHASE	Required	As shown
phasename	Required to name the phase	From one through eight alphameric characters, the first of which must be alphabetic
S	To specify that the phase have its origin at the first available location in the problem program area	As shown
*	To specify that the phase have its origin at the first available location after the most recently processed phase in the job step; equivalent to the S specification if this is the first PHASE statement in the linkage editor input deck	As shown
ROOT	For multiphase programs only; identifies the phase as a root phase (its origin is the first available location in the problem program area	As shown
phase	To indicate that this phase is to have the same origin as another phase currently in the phase library	The name of the other phase as specified in the linkage editor PHASE statement that named it
NOAUTO	To suppress the automatic linking facility for this phase only	As shown

The components of the Model 44 Programming System produce aids that may be used to document and debug programs. This chapter describes the listings, maps, card decks, and error messages produced by these components.

COMPILER OUTPUT

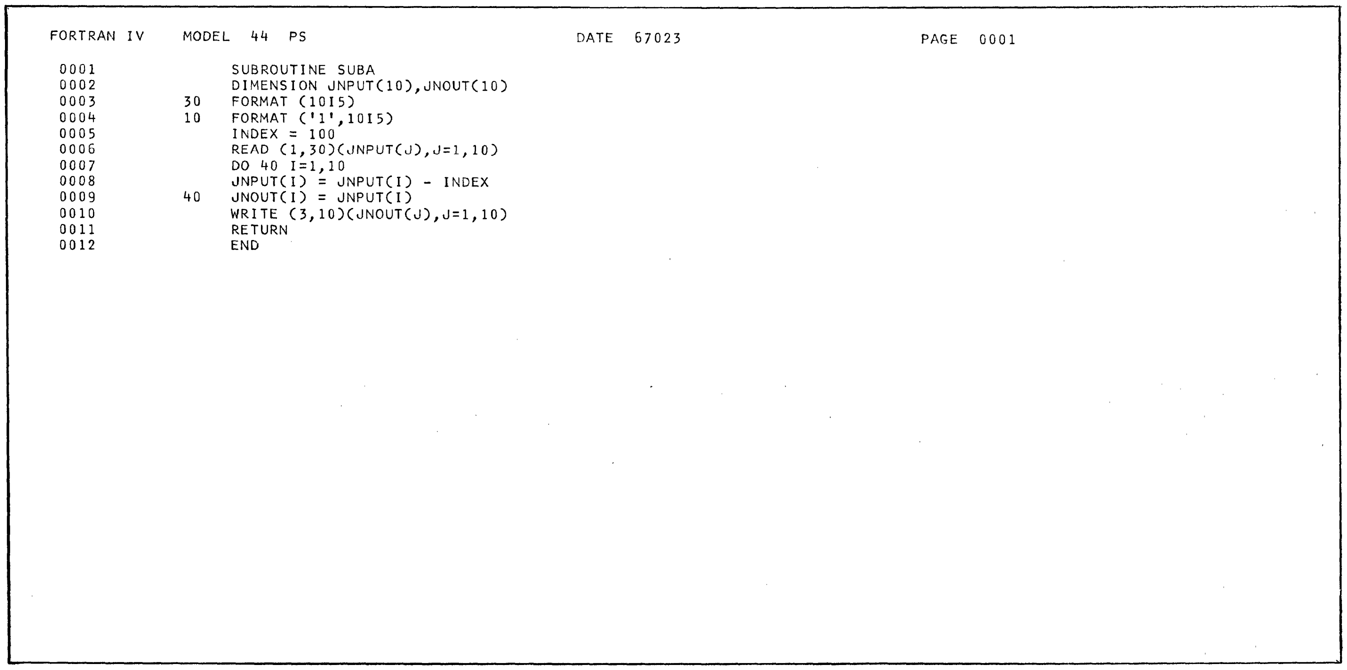
Output from the compiler includes a source listing, a compiler storage map, and/or a module deck, depending on options specified by the programmer in the EXEC statement for the FORTRAN compiler.

Source Listing

Unless the NOSOURCE option is specified, a source listing is written on the system output unit SYSOPT. An example of a source listing is shown in Figure 4.

Compiler Error/Warning Messages

The error/warning messages produced by the compiler are noted on the source listing. Figure 5 illustrates a source listing with error messages.



```
FORTRAN IV    MODEL  44  PS                           DATE  67023                        PAGE  0001

 0001                 SUBROUTINE SUBA
 0002                 DIMENSION JNPUT(10),JNOUT(10)
 0003          30     FORMAT (10I5)
 0004          10     FORMAT ('1',10I5)
 0005                 INDEX = 100
 0006                 READ (1,30)(JNPUT(J),J=1,10)
 0007                 DO 40 I=1,10
 0008                 JNPUT(I) = JNPUT(I) - INDEX
 0009          40     JNOUT(I) = JNPUT(I)
 0010                 WRITE (3,10)(JNOUT(J),J=1,10)
 0011                 RETURN
 0012                 END
```

Figure 4. Source Listing

```
FORTRAN IV    MODEL  44  PS                                DATE  67023                           PAGE  0001

 0001               DIMENSION A(10,10),B(10,10)
 0002               READ (1,5) E,F,G
 0003           1   GO TO 2
 0004               DO 10, I=1,10
             $          $
        01)  LABEL           02)  SYNTAX
 0005               DO 20 J=2,10
 0006          10   A(I,J) = B(I,J)*C(I,J)
 0007          20   CONTINUE
 0008               WRITE (3,6)  A,
                                                                                        $
        01)  SYNTAX
 0009           6   FORMAT (5F10.2
                                                                                        $
        01)  SYNTAX
 0010               END
```

Figure 5. Source Listing with Errors

Error information for a source statement containing errors appears on the listing lines immediately following that statement. For each error encountered, a dollar sign is printed beneath the active character preceding the one that was being inspected when the error was detected. The listing line that follows the printed statement contains only the dollar sign markers.

The next line of the listing describes the marked errors. The errors are numbered within the statement (counting from one for the first error marked); the number is followed by a right parenthesis, the error number, and the type of error. Four errors are described on each line, for as many lines as are required to list all the marked errors in the source statement.

For a description of error/warning messages, see Appendix D.

Storage Map

If the MAP option is specified, a compiler storage map is written on SYSOPT. The map is divided into several tables, classified as follows:

- COMMON variables
- EQUIVALENCE variables
- Scalar variables
- Array variables
- Subprograms called
- NAMELIST variables
- Statement labels

In the case of COMMON variables, a separate table is provided for each blank

```
FORTRAN IV    MODEL  44  PS                              DATE  67023                          PAGE  0002


                                            COMMON BLOCK /        / MAP SIZE  000010
SYMBOL     LOCATION      SYMBOL     LOCATION     SYMBOL     LOCATION       SYMBOL     LOCATION      SYMBOL     LOCATION
CM1        000000        CM2        000004       CM3        000008         CM4        00000C

                                            COMMON BLOCK / NCM1    / MAP SIZE  000008
SYMBOL     LOCATION      SYMBOL     LOCATION     SYMBOL     LOCATION       SYMBOL     LOCATION      SYMBOL     LOCATION
AA         000000        BB         000004

                                            COMMON BLOCK / NCM2    / MAP SIZE  000008
SYMBOL     LOCATION      SYMBOL     LOCATION     SYMBOL     LOCATION       SYMBOL     LOCATION      SYMBOL     LOCATION
CC         000000        DD         000004

                                            SCALAR MAP
SYMBOL     LOCATION      SYMBOL     LOCATION     SYMBOL     LOCATION       SYMBOL     LOCATION      SYMBOL     LOCATION
B          0000E4        C          0000E8       A          0000EC         D          0000F0        E          0000F4
I          0000F8        J          0000FC       L          000100         F          000104        K          000108

                                            ARRAY MAP
SYMBOL     LOCATION      SYMBOL     LOCATION     SYMBOL     LOCATION       SYMBOL     LOCATION      SYMBOL     LOCATION
ARRAY      00010C        LIST       000300

                                            SUBPROGRAMS CALLED
SYMBOL     LOCATION      SYMBOL     LOCATION     SYMBOL     LOCATION       SYMBOL     LOCATION      SYMBOL     LOCATION
FRXPR=     000328        IBCOM=     00032C       SIN        000330

                                            NAMELIST MAP
SYMBOL     LOCATION      SYMBOL     LOCATION     SYMBOL     LOCATION       SYMBOL     LOCATION      SYMBOL     LOCATION
XX         000340        YY         0003A4

                                            LABEL MAP
LABEL      LOCATION      LABEL      LOCATION     LABEL      LOCATION       LABEL      LOCATION      LABEL      LOCATION
    1      000454           10      00045A           7      000466             5      0004C0

   TOTAL MEMORY REQUIREMENTS 000514 BYTES

COMPILER HIGHEST SEVERITY CODE WAS  0
```

Figure 6. Compiler Storage Map

or named COMMON defined in the set of source statements. In all other cases, a separate table is produced for each classification, with the appropriate heading preceding the data. The variable names, statement labels or subprogram names are arranged across the page, five to a line. The relative location of each appears next to the name. If a particular classification of names is not used anywhere in the source program, the corresponding table does not appear in the storage map.

Figure 6 shows a sample compiler storage map.

Module Deck

If the DECK option is specified, a module deck is produced on the system punch unit, SYSPCH. This deck is made up of four types of cards -- TXT, RLD, ESD, and END. A functional description of these cards is given in the following paragraphs.

MODULE DECK CARDS: Every card in the module deck contains a 12-2-9 punch in column 1 and an identifier in columns 2 through 4. The identifier consists of the characters ESD, RLD, TXT, or END. The first four characters of the name of the program are placed in columns 73 through 76 with the sequence number of the card in columns 77-80.

ESD Card: Four types of ESD cards are generated as follows:

ESD, type 0
: contains the name of the program and indicates the beginning of the module. The name is the module name followed by an equal sign.

ESD, type 1
contains the entry point (where control is given to begin execution of the module). The entry point is the name in a SUBROUTINE or FUNCTION statement or the name MAIN44.

ESD, type 2
contains the names of subprograms referred to in the source module by CALL statements, EXTERNAL statements, explicit function references, and implicit function references.

ESD, type 5
contains information about each COMMON area.

The number 0, 1, 2, or 5 is placed in card column 25.

RLD Card: An RLD card is generated for external references indicated in the ESD, type 2 cards. To complete external references, the linkage editor matches the addresses in the RLD card with external symbols in the ESD card. When external references are resolved, the storage at the address indicated in the RLD card contains the address assigned to the subprogram indicated in the ESD, type 2 card. RLD cards are also generated for a branch list produced for statement numbers.

TXT Card: The TXT card contains the constants and variables used by the programmer in his source statements, any constants and variables generated by the compiler, coded information for FORMAT statements, and the machine instructions generated by the compiler from the set of source statements.

END Card: One END card is generated for each set of compiled source statements. This card indicates the end of the module to the linkage editor, the relative location of the main entry point, and the length (in bytes) of the module.

MODULE DECK STRUCTURE: Figure 7 shows the FORTRAN module deck structure. The cards are listed in the order in which they appear in the module deck.

LINKAGE EDITOR OUTPUT

The linkage editor produces a phase map unless the NOMAP option is specified. The linkage editor also produces diagnostic messages, which are listed in Appendix D.

Phase Map

The phase map is written on SYSLST. To the linkage editor, each program (main or subprogram) is a control section (CSECT).

Each control section name is written along with the origin and the length of the control section. The origin and length of a control section are written in hexadecimal numbers.

For each control section, any entry points and their locations are also written; any functions called from the module library are listed.

Figure 8 shows a sample phase map.

ESD, Type 0 Program Name of the Module
ESD, Type 1 Entry Points
ESD, Type 5 COMMON Area
ESD, Type 2 External References
TXT Cards for NAMELIST Tables
TXT Cards for Literal Constants
TXT Cards for FORMAT Statements
TXT Cards for Temporary Storage and Constants
TXT Cards for Module Code
TXT Cards for the BASE Table
TXT Cards for the BRANCH Table
TXT Cards for Subprogram Argument Lists
TXT Cards for Subprogram Addresses
TXT Cards for Address Constants
RLD Cards for the Module
END Card

Figure 7. Object Module Deck Structure

```
 67/000     PHASE      TRANSFER ADDR.   LOCORE    HICORE    BLOCK NO.   ESD TYPE    LABEL       LOADED     REL-FACTOR

COMMON                                                                  COMMON                  004200      0001A0

COMMON                                                                  COMMON      CTRL        0043A0      000004

ROOT        RTPHAS     0043A8           0043A8    007947      293       CSECT       MAIN44=     0043A8      0043A8
                                                                        * ENTRY     MAIN44      0043A8

                                                                        CSECT       BOAIBCOM    004B98      004B98
                                                                          ENTRY     IBCOM=      004B98
                                                                        * ENTRY     ADCON=      004C54
                                                                          ENTRY     FIRSTIM     005DD4

                                                                        CSECT       BOAFEXIT    007170      007170
                                                                          ENTRY     EXIT        007176

                                                                        CSECT       BOAOVLY     007190      007190
                                                                          ENTRY     LOAD        0071A8
                                                                        * ENTRY     LINK        007198

                                                                        CSECT       BOAFIOCS    007288      007288
                                                                          ENTRY     RCBORG=     007890
                                                                          ENTRY     BUFORG=     00788C
                                                                          ENTRY     FIOCS=      007288
                                                                        * ENTRY     VDIOCS=     007894
                                                                        * ENTRY     FIOCD=      0072C2

                                                                        CSECT       BOAUOPT     0078B8      0078B8
                                                                          ENTRY     USEROPT     0078B8

                                                                        CSECT       BOAUNITB    0078C0      0078C0
                                                                          ENTRY     UNITAB=     0078C0

            P1         007948           007948    0086C7      313       CSECT       SUB=        007948      007948
                                                                          ENTRY     SUB         007948

                                                                        CSECT       BOAFRXPI    008610      008610
                                                                          ENTRY     FRXPI=      008618

            P2         007948           007948    0086F7      318       CSECT       CFUNC=      007948      007948
                                                                          ENTRY     CFUNC       007948

                                                                        CSECT       BOAFRXPI    008640      008640
                                                                          ENTRY     FRXPI=      008648

LINKAGE EDITOR HIGHEST SEVERITY WAS  0
```

Figure 8. Phase Map

PHASE OUTPUT

At execution time, FORTRAN phase execution diagnostic messages are generated in three forms -- error code diagnostic messages, program interrupt messages, and operator messages. An error code indicates an input/output error or a misuse of a FORTRAN library function. A program interrupt message indicates a condition that is beyond the capacity of the programming system to correct. An operator message is generated when a STOP or PAUSE statement is executed.

Error Code Diagnostic Messages

When an error condition arises during execution of a FORTRAN program, a message is written on SYSOPT, as follows:

    OAxxxI

The error code is the number specified by the digits xxx. These error codes are described in Appendix D. If any errors are detected, execution of the job step is terminated and a condition code of 16 is returned to the programming system.

Program Interrupt Messages

A program interrupt message containing the old Program Status Word (PSW) is produced on SYSLST when one of the following exceptions occurs:

- Specification Exception (6)
- Fixed-Point Divide Exception (9)
- Exponent-Overflow Exception (C)
- Exponent-Underflow Exception (D)
- Floating-Point Divide Exception (F)

Operator intervention is not required for any of these interruptions, and execution is not terminated. Figure 9 shows the interruption message format.

The five characters in the PSW (i.e., 6, 9, C, D, or F) represent the code number (in hexadecimal) associated with the type of interruption.

Sample Storage Printouts

Figure 10 shows a sample printout for each dump format that can be specified in a call to DUMP or PDUMP. The printouts are given in the following order: hexadecimal, LOGICAL*1, LOGICAL*4, INTEGER*2, INTEGER*4, REAL*4, REAL*8, COMPLEX*8, COMPLEX*16, and literal.

Operator Messages

A message is transmitted to the operator when a STOP or PAUSE statement is executed. Operator messages are written on SYSLOG, the console printer. For a description of these messages, see Appendix D.

$$\text{0A210I PROGRAM INTERRUPT ( ) - OLD PSW IS xxxxxxx}\left\{\begin{matrix}6\\9\\C\\D\\F\end{matrix}\right\}\text{xxxxxxxx}$$

Figure 9. Program Interrupt Message

```
CALL PDUMP WITH HEXADECIMAL FORMAT SPECIFIED

00A3E0     485F5E10  00000000  485F5E10  10000000  42100000

006DC8     42B00000  00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
006DF8     C0000000  00000000  41200000  41566666  0000000C  41100000

CALL PDUMP WITH LOGICAL*1 FORMAT SPECIFIED

006E1E     T  F

CALL PDUMP WITH LOGICAL*4 FORMAT SPECIFIED

006E10     F  T

CALL PDUMP WITH INTEGER*2 FORMAT SPECIFIED

006E18        10

006E1A      -100

006E1C        10

CALL PDUMP WITH INTEGER*4 FORMAT SPECIFIED

006E20               1           2           3           4           5           6           7           8           9          10
006E48              11          12

CALL PDUMP WITH REAL*4 FORMAT SPECIFIED

006E00      0.20000000E 01   0.53999996E 01

CALL PDUMP WITH REAL*8 FORMAT SPECIFIED

006DC8      0.1759999999999999D 03

CALL PDUMP WITH COMPLEX*8 FORMAT SPECIFIED

006DD0                (3.0000000,4.0000000)            (4.0000000,8.0000000)

CALL PDUMP WITH COMPLEX*16 FORMAT SPECIFIED

006DE0              (0.9999999999999990,0.9999999999999990)          (-0.9999999999999990,-0.9999999999999990)

CALL PDUMP WITH LITERAL FORMAT SPECIFIED

006E5C     THIS ARRAY CONTAINS ALPHAMERIC DATA
```

Figure 10. Sample Storage Printouts

This section discusses program optimization and limitations of the compiler.

PROGRAM OPTIMIZATION

Facilities are available in the FORTRAN language that enable a programmer to optimize compilation and execution speed and to reduce the size of the phase.

Initialization

The programmer should initially set to zero all variables that are not initialized by arithmetic statements in his program. The value of a variable cannot be guaranteed until the programmer has given that variable a value by a replacement statement. For example, in the following subprogram:

```
      SUBROUTINE ALPHA(X,Y,Z)
      A=B+2.0
          .
          .
          .
```

the result A may contain any value, because B was not initialized. If the programmer expects B to be zero, he should initialize B as shown in the following statements:

```
      SUBROUTINE ALPHA(X,Y,Z)
      B=0.0
      A=B+2.0
          .
          .
          .
```

Whenever possible, for greater efficiency the DATA initialization statement should be used to define initial values.

Arithmetic Statements

When the programmer wants to calculate the square root, the square root library subprogram should be used instead of the exponential function. For example, the statement:

```
      HYPOT=SQRT(A*A+B*B)
```

is more accurate than the statement:

```
      HYPOT=(A*A+B*B)**0.5
```

because the SQRT function is more accurate than the exponential function.

The mixed mode arithmetic expression is provided as a convenience to the programmer. The number of instructions generated to perform conversions can be reduced, however, if the order of evaluation of expressions is kept in mind.

For example, in the expression:

```
      A = A + I + J
```

where A is real and I and J are integer, the evaluation is from left to right. Instructions are, therefore, produced to convert I to real before it is added to A, and additional instructions are included to convert J to real before it is added to the previous result. If the expression is written in either of the following ways:

```
      A = A + (I + J)
      A = I + J + A
```

one of the conversions is eliminated because I and J are added together first, and the result is converted to real before being added to A.

IF Statement

An arithmetic IF statement lists three statement numbers. One of the listed numbers should immediately follow the IF statement to eliminate unnecessary branching in the phase. For example, the coding represented by the following statements:

```
        IF (A-B)20,30,30
     30 A=0.0
          .
          .
          .
     20 B=0.0
          .
          .
          .
```

is more efficient than the coding represented by the statements:

```
      IF(A-B) 20,30,30
   10 X=2.+Y
       .
       .
       .
   30 A=0.0
       .
       .
       .
   20 B=0.0
```

DO Loop Considerations

Values for expressions that remain constant within a DO loop should be calculated before entry into the loop, instead of calculating the expression each time through the loop. For example, in the following statements:

```
      DO  10 I=1,100
      X(I)=2.4*(G+ALPHA)+Y(I)
   10 CONTINUE
```

the expression 2.0*(G+ALPHA) must be calculated each time the DO loop is executed. For greater efficiency, the following statements should be substituted:

```
      BETA=2.4*(G+ALPHA)
      DO 10 I=1,100
      X(I)=BETA+Y(I)
   10 CONTINUE
```

Because the expression 2.0*(G+ALPHA) is calculated only once, the execution time is decreased.

Any subscripts that remain constant within the range of a DO should not be used in the DO loop. For example, in the following statements:

```
      DO  10 I=1,50
       .
       .
       .
      X(I)=Y(I)+Z(J)
       .
       .
       .
   10 CONTINUE
```

a subscript calculation for Z(J) is performed each time the DO loop is executed, even though Z(J) remains constant for each execution of the loop. By substituting the following statements:

```
      B=Z(J)
      DO 10 I=1,50
       .
       .
       .
      X(I)=Y(I)+B
       .
       .
       .
   10 CONTINUE
```

only one subscript calculation is made for Z(J) and execution time is decreased.

READ/WRITE Statements

To read or write an array, an implied DO in a READ/WRITE statement should be used instead of a DO loop. For example, 5 records, each containing two values, are written by the following statements:

```
   10 FORMAT (F20.5,I10)
      DO 15 I=1,5
   15 WRITE(5,10)A(I),J(I)
```

In the statements:

```
   10 FORMAT (5(F20.5,I10))
      WRITE(5,10)(A(I),J(I),I=1,5)
```

only one record containing 10 values is written. The use of an implied DO saves phase execution time and space on the volume.

Extra subscript calculation within the range of an implied DO should be avoided. This is the same consideration shown in regard to the DO loop. For example, if the statements:

```
    2 FORMAT('0',10F12.6)
       .
       .
       .
      READ(1,2)(A(I),I=4,31,3)
```

are substituted for the statements:

```
    2 FORMAT('0',10F12.6)
      READ(1,2)(A(3*I+1),I=1,10)
```

the intricacy of the subscript calculation is reduced and the phase execution time is reduced.

Boundary Alignment of Variables in COMMON Blocks and EQUIVALENCE Groups

The Model 44 Programming System will adjust improper boundary alignments resulting from the ordering of variables in a COMMON block or in an EQUIVALENCE group. However, considerable efficiency is lost during program execution if the order of the variables is such that they are not located on proper boundaries. A complex variable of length 16 or a real or complex variable of length 8 should be located on a double-word boundary; a real, integer, or logical variable of length 4 should be located on a fullword boundary; an integer variable of length 2 should be located on a halfword boundary. (Information on avoiding improper alignment of variables and the resulting loss in efficiency can be found in the discussions of COMMON blocks and EQUIVALENCE groups in the publication _IBM System/360 FORTRAN IV Language_, Form C28-6515.)

If a variable is located on an improper boundary, each machine-instruction reference to the variable requires that:

1. The specification exception resulting from this reference be processed.

2. The boundary adjustment routine be invoked to simulate the execution of the instruction containing the reference in order to circumvent the boundary violation.

The use of the boundary adjustment routine is an installation option; that is, at the time the system is assembled, an installation can indicate whether or not the routine is to be invoked.

An installation can also modify the system to request that a boundary adjustment message be printed. The message indicates that a boundary adjustment is to take place. It is printed once for each boundary alignment error, up to a maximum of _n_ errors. The value of _n_ is determined by the installation. Boundary adjustment takes place, however, whether or not the boundary adjustment message is printed.

The format of the message is:

```
OA210I PROGRAM INTERRUPT (A) OLD PSW
IS xxxxxxxxxxxxxxxx
```

The A in parentheses identifies boundary adjustment as the cause of the message.

The boundary adjustment routine is invoked whenever a boundary violation occurs in either a FORTRAN main program or subprogram. The routine is also available to assembler language subprograms that operate in a FORTRAN environment (see Appendix C).

When, for some reason, the boundary adjustment routine cannot be loaded from the phase library, the diagnostic message OA219I is printed. The loading of the boundary adjustment routine is dependent upon the amount of space available in the problem program area. The first location available to the boundary adjustment routine is the one immediately following the highest location thus far occupied by any phase of the user's program. This is not necessarily the highest location occupied by the phase in which the boundary alignment error occurs.

FUNCTION Subprograms

The function variables for the principal entry and for each alternate entry to a FUNCTION subprogram are made equivalent. As a result, the value returned for a function is the value of the last function variable set before the RETURN statement causing the return, regardless of the entry point used. For example:

```
FUNCTION SIN(X)
ENTRY COS(X)
SIN = X-X**3/6+X**5/120
COS = SQRT(1.0-SIN**2)
RETURN
END
```

always returns the cosine value, since the variables SIN and COS occupy the same space in storage. In order to produce the desired result, the FUNCTION subprogram should be coded:

```
FUNCTION SIN(X)
X = X-PI/2.0
ENTRY COS(X)
COS = 1-X**2/2+X**4/24
RETURN
END
```

In this case, a change in the argument determines that the value in COS actually is the sine of the angle X when the SIN entry to the function is used.

References to FUNCTION Subprograms

The convention for linkage to FUNCTION subprograms requires that all registers containing active partial results from an expression be saved before branching to the FUNCTION subprogram. As a result, more efficient codes can be produced by placing FUNCTION references so that they are evaluated before the rest of the expression in which they appear is evaluated.

For example, in the statement:

```
A = B * C + D * E * FN(G)
```

the partial results B * C and D * E must both be stored in temporary locations before a call is made to the FUNCTION subprogram FN. If the statement is rewritten as follows:

```
A = FN(G) * D * E + B * C
```

the unnecessary STORE instructions are eliminated because no partial results exist when FN is called.

Use of DUMP and PDUMP

The storage locations assigned to variables in a FORTRAN program are listed in the compiler storage map. Whenever possible, the programmer should refer to the storage map before using the DUMP or PDUMP subroutines. Otherwise, the following conventions should be observed when using the DUMP or PDUMP subroutines to insure that the appropriate areas of storage are dumped.

In the following examples, A is a variable in COMMON, B is a real number, and the array TABLE is dimensioned as:

```
DIMENSION TABLE(20)
```

If an array and a variable are to be dumped at the same time, a separate set of arguments should be used for the array and for the variable. The specification of limits for the array should be from the first element in the array to the last element. For example, the following statement could be used to dump TABLE and B in hexadecimal format, and to terminate execution after the dump is taken:

```
CALL DUMP (TABLE(1),TABLE(20),0,B,B,0)
```

If an area in COMMON is to be dumped at the same time as an area of storage not in COMMON, the arguments for the area in COMMON should be given separately. For example, the following statement could be used to dump the variables A and B in real format without terminating execution:

```
CALL PDUMP (A,A,5,B,B,5)
```

If variables not in COMMON are to be dumped, the programs should list each variable separately in the argument list. For example, if R, P, Q are defined implicitly in the program, the statement:

```
CALL PDUMP(R,R,5,P,P,5,Q,Q,5)
```

should be used to dump the three variables in storage. If, however, the statement:

```
CALL PDUMP(R,Q,5)
```

is used, all main storage between R and Q is dumped.

If an array and a variable are passed as arguments to a subroutine, the arguments in the call to DUMP or PDUMP in the subroutine should specify the parameters used in the definition of the subroutine. For example, if the subroutine SUBI is defined as:

```
SUBROUTINE SUBI(X,Y)
DIMENSION X(10)
```

and the call to SUBI within the source program is:

```
DIMENSION A(10)
      .
      .
      .
CALL SUBI(A,B)
```

then the following statement in the subroutine should be used to dump the variables in hexadecimal format without terminating execution:

```
CALL PDUMP (X(1),X(10),0,Y,Y,0)
```

If the statement

```
CALL PDUMP (X(1),Y,0)
```

Table 3. Compiler Restrictions

ITEM	MAXIMUM NUMBER
Unique variable names	8000
Unique array names	3000
Variables and arrays in COMMON	8000
Names in EQUIVALENCE statements plus number of EQUIVALENCE lists	5000
Statement numbers, including one additional statement number for each DO, Logical IF, and implied DO in an input/output list	16000
Names in Explicit Specification statements	8000
Unique real constants	16000
Unique integer constants	16000
Unique double-precision real constants	8000
Unique complex constants	8000
Unique double-precision complex constants	4000
References to unique subprogram entry point names (explicit and implicit)	8000
Statement function definitions	8000
Nested statement function definitions	15
Dummy arguments for a subprogram	8000
Total arguments to all subprograms and statement functions	16000
Nested DO statements	3000
Nested FUNCTION subprogram references	20

is used, all storage between A(1) and Y is dumped, as the result of the method of transmitting arguments.

Block Length

A block of data written by the FORTRAN IV compiler is never less than 360 bytes long. Even though the LABEL job control statement permits a block length specification smaller than 360, the size of the buffer from which records are written is always at least 360 bytes. While writing his source program, the FORTRAN programmer should try to format his records so that optimum use is made of the 360-byte buffer, thereby conserving space on external storage media.

COMPILER RESTRICTIONS

Table 3 is a list of the limitations imposed on the source program by the FORTRAN compiler.

This appendix illustrates a number of job decks, representing several types of jobs, that could be used with the Model 44 Programming System. For each example, it is assumed that SYSIPT and SYSRDR are assigned to the same device; however, the portions of the job deck read by SYSIPT (that is, all input data) are indicated so that they can easily be removed in the event SYSIPT and SYSRDR are assigned to separate devices.

Compile only (one compilation):

Figure 11 shows a job that consists of one job step -- a FORTRAN compilation. A job name and accounting information are provided in the JOB statement. The comma in the operand field is required by the absence of the DUMP or NODUMP specifications (indicating that NODUMP is to be assumed).

The EXEC statement indicates that the job step is to be unnamed, that a module deck and a compiler map are to be produced, and that a module is not to be written on SYS000. By default, a source listing is produced and it is assumed that source statements are to be punched in EBCDIC.

```
//JOBONE    JOB     ,PGNO3410
//          EXEC    FORTRAN(DECK,NOLINK,MAP)____
                 .                                 |
                 .                                 |
                 .                                 |
    FORTRAN source statements                      }SYSIPT
                 .                                 |
                 .                                 |
                 .                                 |
/*                  (end of data)_________________|
/&                  (end of job)
```

Figure 11. Sample of Compile Only (One Compilation)

Compile only (three compilations):

Figure 12 shows a job that consists of three job steps -- three FORTRAN compilations involving one main program and two subprograms. The job steps are named MAINPRO, SUBA, and SUBB. In each job step, a module deck, a compiler map, and a source listing are produced and no module is written on SYS000. The EXEC statement for job step SUBA indicates that the source statements following it are in BCD; for the other two job steps, EBCDIC is assumed. The EXEC statements for MAINPRO and SUBB illustrate that compiler options may be specified in any order.

```
//JOBTWO   JOB    ,PGNO3411
//MAINPRO  EXEC   FORTRAN(DECK,NOLINK,MAP)
              .
              .
              .
    FORTRAN source statements (main program)       SYSIPT
              .
              .
              .
/*                (end of data)
//SUBA     EXEC   FORTRAN(DECK,NOLINK,MAP,BCD)
              .
              .
              .
    FORTRAN source statements (subprogram)         SYSIPT
              .
              .
              .
/*                (end of data)
//SUBB     EXEC   FORTRAN(DECK,MAP,NOLINK)
              .
              .
              .
    FORTRAN source statements (subprogram)         SYSIPT
              .
              .
              .
/*                (end of data)
/&                (end of job)
```

Figure 12. Sample of Compile Only (Three Compilations)

Edit only:

Figure 13 shows a job that consists of one job step -- the editing of three module decks. The EXEC statement indicates that the job step is unnamed and that the phase output produced by the linkage editor is to be retained in the phase library for use in subsequent jobs. By default, a phase map is produced on SYSLST.

The modules to be edited are named MAIN, SUBONE, and SUBTWO and will be copied in that order onto SYS000 by the linkage editor. A single phase, named ALPHA, is to be produced; its origin is to be the first available location in the problem program area. The INCLUDE statements indicate that phase ALPHA is to be composed of modules MAIN, SUBONE, and SUBTWO, in that order, and that each module will be found on SYS000. (Note that the PHASE and INCLUDE statements could be omitted from the job deck; the only difference in the results obtained is that phase ALPHA would instead be named MAIN, the name of the first module to be included in the phase.)

The three module decks to be edited here could well be the three decks produced in the previous example of three compilations. Although the job steps in that example are named MAINPRO, SUBA, and SUBB, these names are not carried over with the module decks into another job. In order to be edited, the modules must be named again in MODULE statements. Of course, the names used for the compilation job steps could be repeated in the MODULE statements or, as is the case here, entirely new names could be used.

```
//JOBTHREE JOB     ,PGNO3412
//         EXEC    LNKEDT(KEEP)──────────┐
           MODULE  MAIN                   │
             .                            │
             .                            │
             .                            │
           Module deck (main program)     │
             .                            │
             .                            │
             .                            │
           MODULE  SUBONE                 │
             .                            │
             .                            │
             .                            │
           Module deck (subprogram)       │
             .                            │
             .                            │
             .                            > SYSIPT
           MODULE  SUBTWO                 │
             .                            │
             .                            │
             .                            │
           Module deck (subprogram)       │
             .                            │
             .                            │
             .                            │
           PHASE   ALPHA,S                │
           INCLUDE MAIN,L                 │
           INCLUDE SUBONE,L               │
           INCLUDE SUBTWO,L               │
/*                 (end of data)──────────┘
/&                 (end of job)
```
Figure 13. Sample of Edit Only

Compile and edit:

Figure 14 shows a job that consists of two job steps -- a FORTRAN compilation and the editing of the resulting module and a module deck produced in a previous job. The compilation job step is named MAINPRO; output from the compiler is to include a source listing, a compiler map, and a module on SYS000. The name of the module on SYS000 will be the job step name, MAINPRO. No module deck is produced and source statements are assumed to be in EBCDIC.

The editing job step is unnamed; phase output from the job step is to be retained in the phase library; a phase map is to be produced. The module deck, which will be copied onto SYS000 by the linkage editor, is named SUBPROG. One phase, BETA, is to be produced and is to include the modules MAINPRO and SUBPROG in that order; both modules will be found on SYS000. The PHASE and INCLUDE statements could be left out of this job deck without affecting the results in any way other than phase BETA being named MAINPRO instead.

```
//JOBFOUR  JOB     ,PGNO3413
//MAINPRO  EXEC    FORTRAN(MAP)________________
               .                               |
               .                               |
               .                               |
    FORTRAN source statements                  > SYSIPT
               .                               |
               .                               |
               .                               |
/*                 (end of data)_______________|
//           EXEC    LNKEDT(KEEP)______________
             MODULE  SUBPROG                   |
               .                               |
               .                               |
               .                               |
             Module deck                       |
               .                               > SYSIPT
               .                               |
               .                               |
             PHASE   BETA,S                    |
             INCLUDE MAINPRO,L                 |
             INCLUDE SUBPROG,L                 |
/*                   (end of data)_____________|
/&                   (end of job)
```

Figure 14. Sample of Compile and Edit

Execute only:

Figure 15 shows a job that consists of one job step -- the execution of the phase, BETA, produced in the previously illustrated compile-and-edit job. The JOB statement now indicates that a dump is to be produced if the job terminates abnormally.

Before the phase is executed, two data sets required by it are associated with symbolic unit names. The ACCESS statement associates the data set named INPUT with symbolic unit SYS004 (which corresponds to data set reference number 4). The device to be used for this data set is an IBM 2400 Magnetic Tape Unit with a 9-track read/write head and a recording density of 800 bytes per inch; the data set itself is located on the tape whose volid is T645.

The ALLOC statement associates the data set named MASTER with symbolic unit SYS002 (which corresponds to data set reference number 2). In addition, 20 blocks of space are allocated for the data set on a fresh disk volume, which must be an IBM 1316 Disk Pack mounted on an IBM 2311 Disk Storage Drive. Finally, the data set MASTER is to be entered into the system catalog. The LABEL statement, which is required after the ALLOC statement shown, indicates a FORTRAN record length of 360 bytes and an expiration date of January 1, 1968.

```
//JOBFIVE   JOB     DUMP,PGNO3414
//SYS004    ACCESS  INPUT,2400='T645'
//SYS002    ALLOC   MASTER,1316=FRESH,20,CATLG
//          LABEL   360,68001
//          EXEC    BETA
/&                  (end of job)
```

Figure 15. Sample of Execute Only

Edit and execute:

Figure 16 shows a job that consists of two job steps -- the editing of two module decks and the execution of the resulting phase. The editing job step is unnamed and no phase map is to be produced. Also, the phase output can be discarded at the end of the next job step (in this case, immediately after the phase is executed).

The modules to be edited are named PAYMAIN and PAYSUB and will be copied in that order onto SYS000 by the linkage editor. The absence of PHASE and INCLUDE statements causes the linkage editor to generate the following statements:

```
PHASE   PAYMAIN,*
INCLUDE PAYMAIN,L
INCLUDE PAYSUB,L
```

The result is that a single phase named PAYMAIN is produced and the two modules on SYS000 (namely, PAYMAIN and PAYSUB) are included in the phase in that order. The origin of the phase is the first available location in the problem program area.

The presence of input data after the phase execution EXEC statement indicates that the data set reference number 5 (corresponding to SYSIPT) is cited in the source program.

```
//JOBSIX    JOB     DUMP,PGNO3415
            EXEC    LNKEDT(NOMAP)  ---------------┐
            MODULE  PAYMAIN                       |
             .                                    |
             .                                    |
             .                                    |
            Module deck                           |
             .                                    |
             .                                    |
             .                                    |
            MODULE  PAYSUB                        >  SYSIPT
             .                                    |
             .                                    |
             .                                    |
            Module deck                           |
             .                                    |
             .                                    |
             .                                    |
/*                  (end of data)  ---------------┘
//          EXEC  --------------------------------┐
             .                                    |
             .                                    |
             .                                    |
     Input data to FORTRAN program                >  SYSIPT
             .                                    |
             .                                    |
             .                                    |
/*                  (end of data)                 |
/&                  (end of job)  ----------------┘
```

Figure 16. Sample of Edit and Execute

Compile, edit, and execute:

Figure 17 shows a job that consists of four job steps -- two FORTRAN compilations involving a subprogram and a main program, the editing of the two resulting modules, and the execution of the resulting phase. The compilation job steps are named SUBPROG and MAIN. In each job step, a source listing, a compiler map, and a module on SYS000 are to be produced, a module deck is not to be produced, and the source statements are punched in EBCDIC. (Note that in the EXEC statement for job step SUBPROG, all compiler options are specified, while in the EXEC statement for job step MAIN, the default options are omitted.)

The editing job step is unnamed; phase output is to be retained; a phase map is to be produced. A single phase, named GAMMA, is to be produced; its origin is to be the first available location in the problem program area. The phase is to include two modules, MAIN and SUBPROG, in that order; the source of each module is SYS000. (Note that the omission of the PHASE and INCLUDE statements from this job deck would cause a change not only in the phase name, but also in the order in which the modules are included in the phase.)

Before the phase is executed, one data set required by it is associated with a symbolic unit name. This is the data set MASTER (cataloged in the execute-only example), which is again associated with symbolic unit SYS002. No further information is required in the ACCESS statement because MASTER is a cataloged data set. The presence of input data after the phase execution EXEC statement indicates that data set reference number 5 (corresponding to SYSIPT) is cited in the source program.

```
//JOBSEVEN JOB      DUMP,PGNO3416
//SUBPROG  EXEC     FORTRAN(NODECK,SOURCE,LINK,MAP,EBCDIC)
               .
               .
               .
    FORTRAN source statements (subprogram)          SYSIPT
               .
               .
               .
/*                  (end of data)
//MAIN     EXEC     FORTRAN(MAP)
               .
               .
               .
    FORTRAN source statements (main program)        SYSIPT
               .
               .
               .
/*                  (end of data)
//         EXEC     LNKEDT(KEEP)
           PHASE    GAMMA,S
           INCLUDE  MAIN,L                          SYSIPT
           INCLUDE  SUBPROG,L
/*                  (end of data)
//SYS002   ACCESS   MASTER
//         EXEC
               .
               .
               .
    Input data to FORTRAN program                   SYSIPT
               .
               .
               .
/*                  (end of data)
/&                  (end of job)
```

Figure 17. Sample of Compile, Edit, and Execute

APPENDIX B: EBCDIC AND BCDIC CARD CODES

Character	EBCDIC	BCDIC
(blank)		
+	12-8-6	12
-	11	
/	0-1	
=	8-6	3-8
. (period)	12-3-8	
)	11-5-8	12-4-8
*	11-4-8	
, (comma)	0-3-8	
(	12-5-8	0-4-8
' (apostrophe)	5-8	4-8
&	12	
0	0	
1	1	
2	2	
3	3	
4	4	
5	5	
6	6	
7	7	
8	8	
9	9	
A	12-1	
B	12-2	
C	12-3	
D	12-4	
E	12-5	
F	12-6	
G	12-7	
H	12-8	
I	12-9	
J	11-1	
K	11-2	
L	11-3	
M	11-4	
N	11-5	
O	11-6	
P	11-7	
Q	11-8	
R	11-9	
S	0-2	
T	0-3	
U	0-4	
V	0-5	
W	0-6	
X	0-7	
Y	0-8	
Z	0-9	
$	11-3-8	

This appendix provides a list of the 49 characters valid in a FORTRAN source program (except in literal data where any valid card code is acceptable). The EBCDIC punch combination for each character is shown. A BCDIC punch combination is shown only when it differs from the EBCDIC punch combination. Only five characters

+ = ( ) '

have different punch combinations; in all other cases, the EBCDIC and BCDIC combinations are the same.

Note: If the source program is punched entirely in EBCDIC (that is, the EBCDIC option is in effect), statement numbers passed as arguments must be coded as &n (where n represents the statement number).

However, if BCD characters appear in the source program (that is, the BCD option is in effect), statement numbers passed as arguments must be coded as $n and the character $ must not be used as an alphabetic character elsewhere in the source program.

A FORTRAN programmer can use assembler language subprograms with his FORTRAN program. This section describes the linkage conventions that must be used by the assembler language subprogram to communicate with the FORTRAN program.

SUBROUTINE REFERENCES

The FORTRAN programmer can refer to a subprogram in two ways: by a CALL statement or by a function reference within an arithmetic expression. For each subprogram reference, the compiler generates:

1. An 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.

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. For the last entry, the first (leftmost) 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 the called subprogram, 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 18 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.

FORTRAN programs save floating-point registers before calling a subprogram. The subprogram does not have to save and restore them.

```
AREA------->┌──────────────────────────────────────────────────────────────────────┐
(word 1)    │This word is part of the standard linkage convention used by the programming│
            │system.   An assembler language subprogram can use the word 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; that is, the return address.  When a subprogram│
            │returns control, the first byte of this word is set to ones.                 │
AREA+16---->├──────────────────────────────────────────────────────────────────────┤
(word 5)    │The contents of register 15; that is, the entry address.                     │
AREA+20---->├──────────────────────────────────────────────────────────────────────┤
(word 6)    │The contents of register 0.                                                  │
AREA+24---->├──────────────────────────────────────────────────────────────────────┤
(word 7)    │The contents of register 1.                                                  │
            ├─────────────────────────────────────────────────────────────────────        │
   .        │               .                                                             │
   .        │               .                                                             │
   .        │               .                                                             │
AREA+68---->├──────────────────────────────────────────────────────────────────────┤
(word 18)   │The contents of register 12.                                                 │
            └──────────────────────────────────────────────────────────────────────┘
```

Figure 18. Save Area

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. A branch is made to the address in general register 15 and the return address is saved in general register 14. Table 4 illustrates the use of the linkage registers.

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. Instructions to save any general registers used by the subprogram in the save area reserved by the calling program.
3. Instructions to restore the "saved" registers before returning control to the calling program.
4. An instruction that sets to ones the first byte in the fourth word of the save area, indicating that control is returned to the calling program.
5. An instruction that returns control to the calling program.

Figure 19 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 for transferring arguments from the calling program and returning the arguments to the calling program.

Sharing Data in COMMON

With Model 44 FORTRAN, general register 4 contains the address of the COMMON area. If the size of the COMMON area exceeds 4095 bytes, additional registers (e.g., register 5, 6, and 7) are assigned consecutively.

Table 4. 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. (For subroutine subprograms, the result is returned by the subprogram in a variable passed to the subprogram by the programmer's CALL statement.)
1	Argument List Register	Address of the argument list passed to the called subprogram.
13	Save Area Register	Address of the area reserved by the calling program in which the contents of certain registers are stored by the called program.
14	Return Register	Address of the location in the calling program to which control is returned after execution of the called program.
15	Entry Point Register	Address of the entry point in the subprogram.

Higher Level Assembly 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.

Figure 20 shows the linkage conventions for an assembler subprogram that calls another assembler subprogram.

Name	Oper.	Operand	Comments
deckname	START	0	
	ENTRY	name	NAME THE ENTRY POINT FOR THIS SUBPROGRAM
	DC	CLm'name'	m MUST BE AN ODD INTEGER TO INSURE THAT THE PROGRAM
	DC	X'm+1'	STARTS ON A HALF-WORD BOUNDARY. THE NAME MAY BE
*			PADDED WITH BLANKS.
	USING	*,15	
name	ST	14,12(13)	THE CONTENTS OF REGISTERS 14, 15, AND 0 THROUGH R ARE
	ST	15,16(13)	STORED IN THE SAVE AREA OF THE CALLING PROGRAM. R IS ANY
	ST	0,20(13)	NUMBER FROM 0 THROUGH 12 AND D IS THE APPROPRIATE
		.	DISPLACEMENT
		.	
		.	
	ST	R,D(13)	
	user	written source statements	
		.	
		.	
		.	
	L	2,28(13)	THE CONTENTS OF REGISTERS 2 THROUGH R ARE RESTORED.
		.	
		.	
		.	
	L	R,D(13)	
	MVI	12(13),X'FF'	INDICATE CONTROL RETURNED TO CALLING PROGRAM
	BCR	15,14	RETURN TO CALLING PROGRAM

Figure 19. Lowest Level Assembler Subprogram

```
deckname START  0
         ENTRY  name1           ENTRY NAME FOR THIS SUBPROGRAM
         EXTRN  name2           ENTRY NAME THE CALLED SUBPROGRAM
         DC     CLm'name1'
         DC     X'm+1'
         USING  *,15
*  SAVE ROUTINE
name1    ST     14,12(13)
         ST     15,16(13)
         ST     0,20(13)
                .
                .
                .
         ST     R,D(13)
         LR     r2,13           LOADS REGISTER 13, WHICH POINTS TO THE SAVE AREA OF THE
*                               CALLING PROGRAM, INTO ANY GENERAL REGISTER, R2, EXCEPT 0
*                               AND 13.
         LA     13,AREA         LOADS THE ADDRESS OF THIS PROGRAM'S SAVE AREA INTO
*                               REGISTER 13.
         ST     13,8(0,r2)      STORES THE ADDRESS OF THIS PROGRAM'S SAVE AREA INTO THE
*                               CALLING PROGRAM'S SAVE AREA
         ST     r2,4(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
*  user-written program statements
prob1
                .
                .
                .
*  CALLING SEQUENCE
         LR     12,15           SAVE BASE REGISTER FOR THIS PROGRAM
         LA     1,ARGLIST       LOAD ADDRESS OF ARGUMENT LIST
         L      15,ADCON
         BALR   14,15
         LR     15,12           RESTORE BASE REGISTER FOR THIS PROGRAM
*  more user-written program statements
                .
                .
                .
*  RETURN ROUTINE
         L      13,AREA+4       LOADS THE ADDRESS OF THE PREVIOUS SAVE AREA BACK INTO
*                               REGISTER 13
         L      2,28(13)
                .
                .
                .
         L      R,D(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 20. Higher Level Assembler Subprogram

In-Line Argument List

The assembler programmer can establish an in-line argument list instead of an out-of-line list. In this case, he may substitute the calling-sequence and argument list shown in Figure 21 for that shown in Figure 20.

Name	Oper.	Operand
ADCON	DC	A(prob$_1$)
	.	
	.	
	.	
	LA	14,RETURN
	L	15,ADCON
	CNOP	2,4
	BALR	1,15
	DC	AL4(arg$_1$)
	.	
	.	
	.	
	DC	X'80'
	DC	AL3(arg$_n$)
RETURN	BC	0,X'isn'

Figure 21. In-Line Argument List

GETTING 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 for A
00000000	address for B
10000000	address for C

For purposes of discussion, assume A is a double-precision (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 double-precision variable A to location VAR in the subprogram.

```
L      q,0(1)
L      r,0(q)
ST     VAR
L      r,4(q)
ST     VAR+4
```

where q and r are any general registers.

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.

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 5 shows the general subscript format for arrays of 1, 2, and 3 dimensions.

Table 5. Dimension and Subscript Format

Array A	Subscript Format
A(D1)	A(C1*V1+J1)
A(D1,D2)	A(C1*V1+J1,C2*V2+J2)
A(D1,D2,D3)	A(C1*V1+J1,C2*V2+J2,C3*V3+J3)
D1, D2, and D3 are integer constants used in the DIMENSION statement. C1, C2, C3, J1, J2, and J3 are integer constants. V1, V2, and V3 are integer 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 between the variable and 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=(C1*V1+J1-1)*L
DISPLC=(C1*V1+J1-1)*L+(C2*V2+J2-1)*D1*L
DISPLC=(C1*V1+J1-1)*L+(C2*V2+J2-1)*D1*L
     +(C3*V3+J3-1)*D2*D1*L
```

where L is the length of each variable in the 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 variables in a three-dimensional array, the displacement (DISPLC) is calculated to be 28. The following instructions can be used to move the variable:

```
LA      q,8(1)
LA      r,DISPLC
L       s,0(q,r)
ST      s,ARVAR
```

where q, r, and s are general registers.

This appendix contains a detailed description of the diagnostic messages produced during operation of the Model 44 Programming System. Messages are discussed in the following order:

- Supervisor messages
- Job control messages
- Compiler messages
- Linkage editor messages
- Phase execution messages

SUPERVISOR MESSAGES

Supervisor messages may appear at any time during execution. They are written by the supervisor.

FA0CI ERR LDING MESS WRTR

Explanation: An input/output error occurred while the system was loading its message writer routine. The job is canceled.

FA0DI cuu NOT OPERATIONAL

Explanation: cuu is the physical address of an input/output device. An input/output operation was requested for a data set on a device that is not operational. The job is canceled.

FA0EI cuu SNSE UN CHK

Explanation: cuu is the physical address of an input/output device. A unit check interruption occurred in response to a sense operation on a device. The job is canceled.

FA0FI cuu I/O PROG CHK

Explanation: cuu is the physical address of an input/output device. A program check occurred during execution of an input/output operation. This may be the result of a zero count in a data transmission request or an invalid data address. The job is canceled.

FA10I xxxx CAN'T BE LOADED

Explanation: xxxx is the name of a system routine. The routine is needed by a system program, but it cannot be found or it cannot be loaded because of an input/output error on SYSAB1. The job is canceled.

FB0BI OPRTR CNCLED

Explanation: A job has been canceled by the operator.

FB11I CNCL IN CNCL RTN

Explanation: A CANCEL was requested by the operator while the system was executing the CANCEL routine.

GA06I PD LST FULL
LAST SVC PSW xxxxxxxxxxxxxxxx

Explanation: The x's are replaced by the new program status word for the last supervisor interruption. Too many supervisor calls have been issued in too short a time. The job is canceled.

GA07I ILLEG CODE - SVC x

Explanation: x is an invalid code that was used in a supervisor call. The job is canceled.

GA08I xxxxxxxx CAN'T BE FTCHD

Explanation: xxxxxxxx was used as the name of a phase. The system cannot find any phase with this name in the phase library. The job is canceled.

PROG CHK INT CODE x
HA02I IN USER PROG CHK RTN

Explanation: A program check developed during execution of a user's program check interruption routine. x is the interruption code. The job is canceled.

PROG CHK INT CODE x
HA03I USER RTN NOT APPLICABLE

Explanation: x is the program check interruption code. On interrupt codes 1 through 5, no user program check routine is entered. The job is canceled.

PROG CHK INT CODE x
HA04I NO USER RTN SPECIFIED

Explanation: x is the program check interruption code. There is no user program check routine specified to handle this type of program check. The job is canceled.

PROG CHK INT CODE x
HA05I PSW - xxxxxxxxxxxxxxxx In sPVSR STATE

Explanation: x is the program check code and the other x's are replaced by a program status word. A program check occurred in the supervisor state. The PSW is the last problem program PSW. The job is canceled.

JA0AI JOB CANCELLED

Explanation: A job has been canceled. Another message usually appears giving the reason for the cancellation.

JOB CONTROL MESSAGES

Messages written by the job control processor are distinguished by the initial characters IA.

These messages are written on SYSLST. In the following listing, they are grouped by type. Each group shares a common text message, but the identification code differs to indicate the source of the error condition.

The messages are as follows:

IAxxx STMNT FMT ERR

Messages IA01I through IA09I indicate an error in the text of a job control statement. The xxx portion identifies the problem area more specifically, as follows:

IA01I - Identification field. The first two columns do not contain the proper characters for a job control statement. The job is canceled.

IA02I - Name field. An invalid name has been specified. It may not be appropriate for the statement, as when something other than a symbolic unit name is specified in the name field of an ALLOC or ACCESS statement. The job is canceled.

IA03I - Operation field. The system does not recognize the operation specified. The job is canceled.

IA04I - Operand field. A required parameter is missing. The job is canceled.

IA05I - Operand delimiter. An improper character has been used as a delimiter. The job is canceled.

IA06I - Field size or count. A parameter in the operand field is too long, or specifies an unacceptable size, or there are too many characters within a pair of parentheses. The job is canceled.

IA07I - Operand field. The operand field contains a parameter that cannot be recognized or that should not be used in this statement. The job is canceled.

IA08I - Continuation error. The first two columns of a continuation statement do not contain the // characters, information starts before column 16, or a continuation statement is required but column 72 is not punched. The job is canceled.

IA09I - VPS field. The VPS field of an EXEC statement contains an invalid entry, or a VPS setting has been specified for a system that is not equipped with this facility. The job continues, but the parameter is ignored.

IAxxx STMNT SEQ ERR

Messages IA11I through IA17I indicate improper use of a job control statement. The xxx portion identifies the specific problem, as follows:

IA11I - A LABEL statement was misused. For a unit record data set or a tape data set, the LABEL statement did not follow an ACCESS or ALLOC statement. For a direct access data set, it did not follow an ALLOC statement. Otherwise, it appeared in an invalid place in the job deck. The job is canceled.

IA12I - An ALLOC statement for a direct access data set was not followed by a LABEL statement. The job is canceled.

IA13I - The system read a // statement that was not a JOB statement and was not preceded by a JOB statement. The job is canceled.

IA14I - The phase name field of an EXEC statement is blank and the job step does not immediately follow a successful linkage editor job step, or the linkage editor reported an error severity level greater than 4. The job is canceled.

IA15I - A DELETE, CONDENSE, or RENAME statement refers to a data set that was not cited in an ACCESS or ALLOC statement previously in the job. The statement is ignored.

IA16I - A data set or symbolic unit referred to in the SAME=parameter field of a LABEL statement was not defined previously in the job nor is it a system data set. The job is canceled.

IA17I - An invalid statement appears among the job control statements or an EXEC statement is missing. The job is canceled.

IAxxx VOL REQ ERR

Messages IA21I through IA28I apply to volumes requested in ALLOC or ACCESS statements.

IA21I - The system has no record of the volume or device referred to. The job is canceled.

IA22I - A request for a particular type of device cannot be satisfied. Not enough devices of this type are available. The job is canceled.

IA23I - The volume field of an ACCESS or ALLOC statement contains an entry that cannot be resolved. The job is canceled.

IA25I - An attempt has been made to remove the system residence volume. The job is canceled.

IA26I - A statement has requested assignment of a device that is not operational. The job is canceled.

IA27I - The volume field of an ACCESS or ALLOC statement specifies the address of a device that was assigned to another data set previously in the same job step. The job is canceled.

IA28I - A job control maintenance statement has been detected for a data set on a volume that is not mounted. The statement is ignored.

IAxxx DSNAME ERR xxxxxxxx

Messages IA31I through IA38I apply to the names of data sets and members. The name causing the condition is printed with the message.

IA31I - The required data set cannot be found in the volume specified. The job is canceled.

IA32I - The required member cannot be found in the data set specified. The job is canceled.

IA33I - The data set named cannot be found in the system catalog. The action requested for the data set is not performed.

IA34I - The name specified for a data set duplicates the name of a data set that is already on the same volume. The job is canceled.

IA35I - The name of a member in a directoried data set duplicates another name already in the directory. The job is canceled.

IA36I - A data set name duplicates another name in the system catalog. The job is canceled.

IA37I - The block length requested for the data set is too large for the device. The job is canceled.

IA38I - An attempt has been made to close a new member of a directoried data set, but the member was never written.

IA41I INSUFF SP xxxxxx

<u>Explanation:</u> xxxxxx is a volume identification number. This message indicates there is not enough room on a disk volume to permit a requested operation. The job is canceled.

IA42I INSUFF SP xxxxxx

Explanation: xxxxxx is the volume identification number of a disk volume whose volume table of contents is full. No new data sets can be added to the volume until some of those already on it are deleted or, if there is vacant space on the disk, the volume table of contents is enlarged through reinitialization. The job is canceled.

IA43I INSUFF SP xxxxxxxx

Explanation: xxxxxxxx is the name of a directoried data set whose directory is full. No new members can be added until some directory entries are deleted. The job is canceled.

IA44I INSUFF SP xxxxxxxx

Explanation: xxxxxxxx is the name of a directoried data set in which there is not enough room to add another member, or it is the name of a data set of any type in which there is not enough room to write another block of data. The job is canceled.

IA45I INSUFF SP CATLG

Explanation: There is not enough space in the system catalog to add another entry. The job is canceled.

IA46I INSUFF SP JOBTABLE

Explanation: The job control processor's working space is full. The job is canceled. Either the size of the job must be reduced or the size of the system's SDSUAS data set must be increased before the next run.

IA47I INSUFF SP FCB

Explanation: The system does not have enough space in main storage to construct a file control block for the symbolic unit cited in an ALLOC or ACCESS statement. The symbolic unit number may exceed the number that can be handled at the installation. The job is canceled.

IA50I ABN EOJ

Explanation: The job did not include a /& (end-of-job) statement. The job is canceled.

IA55I hhmmss

Explanation: This message, appearing after a JOB statement, gives the time that the execution of the job started, expressed in hours, minutes and seconds.

IA58I CUU RW RR RN PW PR PN

IA59I xxx xx xx xx xx xx xx

Explanation: These messages report the number of input/output errors detected during the job. The count is listed in columns by device. The CUU column is the device address; RW is the number of recovered writing errors; RR, recovered reading errors; RN, the number of recovered nondata transmit errors; PW, permanent writing errors; PR, permanent reading errors; and PN the number of permanent nondata transmit errors.

IA61I NEW NAME NOT CAT

Explanation: A renamed data set cannot be cataloged. The name has been changed, as specified in a RENAME statement, but the new name cannot be entered in the system catalog.

IA62I SYSERR

Explanation: An unrecoverable system error has occurred. The operator must reinitiate the initial program loading procedure.

IA70I DA FMT ERR xxxxxx

Explanation: xxxxxx is the volume identification number of a volume whose volume label is unreadable or in an improper format. The volume cannot be used by the system until it is initialized via a system utility program. The job is canceled.

IA71I DA FMT ERR xxxxxx

Explanation: xxxxxx is the volume identification number of a volume whose volume label has been changed during the job. The job is canceled.

IA72I DA FMT ERR xxxxxx

Explanation: xxxxxx is the volume identification number of a volume whose volume table of contents is not in the proper format. The volume cannot be used until it is initialized via a system utility program. The job is canceled.

IA73I DA FMT ERR xxxxxxxx

Explanation: xxxxxxxx is the name of a sequential data set for which a directoried data set request has been made. The job is canceled.

IA75I DISK I/O ERR

Explanation: The system's standard error recovery procedure has failed. The system is unable to write on a disk volume during an ACCESS or ALLOC operation, either in handling the volume table of contents or a data set. The job is canceled.

IA76I DISK I/O ERR

Explanation: The system's standard error recovery procedure failed while attempting to recover an input/output error during a DELETE operation. The job is canceled.

IA79I NO CATLG

Explanation: A cataloging request has been made but cannot be executed because the system does not have a catalog.

IA82I JC INIT DONE

Explanation: The system has just completed an initial program loading procedure.

IA86I CAUTION JOB TBL FULL

Explanation: The job control processor's working space is full. This is only a warning message. Any additional job control statement will overlay a previous entry. If this happens, some references to data sets or symbolic units mentioned in previous statements may not be acceptable, and some symbolic unit assignments may not be made. The size of the job should be reduced, or the size of system data set SDSUAS should be increased.

IA88I SYSxxx cuu dsname volid

Explanation: SYSxxx is a symbolic unit name, cuu is the unit's physical address, dsname is the data set associated with the unit, and volid identifies the volume containing the data set. This format is used by the system in responding to a LISTIO request.

IA89I M cuu volid

Explanation: M is the abbreviation for Mount, cuu is a device address, and volid is a volume identification number. A new volume has just been assigned to a disk device. The operator can mount the volume to prepare for the IA90A message.

IA90A M ALL REQ DISKS

Explanation: This message instructs the operator to mount all disk volumes requested in preceding IA89I messages. When this is done, he signals the system to continue processing.

IA91D VOL xxxxxx UNREADABLE

Explanation: xxxxxx represents a volume identification number. This message appears after an IA90A message. It indicates that the system is unable to read the volume label of a disk that has been mounted. The operator can mount another volume, instruct the system to ignore the volume but continue processing, or cancel the job.

IA92I JCT OFLOW

Explanation: A LABEL statement uses the SAME parameter, but the reference cannot be resolved because the job control processor's working space was filled earlier in the program. This message follows an IA86I message. The job is canceled.

IA93I OPEN ERR SYSxxx

Explanation: SYSxxx identifies a system unit. An error was detected while the job control processor was opening a data set on the specified system unit. The data set is not opened, but processing continues.

IA94I CLOSE ERR SYSxxx

Explanation: SYSxxx identifies a system unit. An error was detected while the job control processor was closing a data set on the specified system unit. The data set is not closed, but processing continues.

COMPILER MESSAGES

This section contains an alphabetic list of the error/warning messages produced by the FORTRAN IV compiler. An explanation of each message, including its condition code setting, is given. Serious error messages have a condition code setting of 16 or 12; warning messages have a condition code setting of 4 or 0.

ALLOCATION

Explanation: The storage assignment specified by a source statement cannot be performed because the use of a variable name is either improper or in conflict with some prior use of that name. For example, a name listed in a COMMON block has been listed in another COMMON block; or a variable listed in an EQUIVALENCE statement is followed by more than seven subscripts. (Condition code -- 12)

BLOCK DATA PROGRAM ERRORS

Explanation: This message is produced if variables in the source statements have been specified within a BLOCK DATA subprogram but have not also been defined as COMMON. A list of these variables follows the message. (Condition code -- 0)

COMMA

Explanation: A comma required in a statement does not appear. (Condition code -- 0)

COMMON BLOCK/ /ERRORS

Explanation: This message pertains to errors that exist in the definitions of EQUIVALENCE sets that refer to the COMMON area. The message is produced when there is a contradiction in the allocation specified, when there is an attempt to extend the beginning of the COMMON area, or if the assignment of COMMON storage results in an attempt to allocate a variable at a location that does not fall on the appropriate boundary. The name of the COMMON block in error appears between the two slashes. A list of the variables that could not be allocated because of the errors follows the message. (Condition code -- 4)

CONVERT

Explanation: In a DATA statement or in an Explicit Specification statement containing data values, the mode of a constant is different from the mode of the variable with which the constant is associated. The constant is converted to the correct mode by the compiler; this message is simply a notification to the programmer that the conversion is performed. (Condition code -- 0)

DUMMY DIMENSION ERRORS

Explanation: If variables specified as dummy array dimensions are not in COMMON and are not global dummy variables, this message is produced. A list of the dummy variables that are in error follows the message. (Condition code -- 12)

DUPLICATE LABEL

Explanation: The label appearing in the label field of a statement is already defined (has appeared in the label field of a previous statement). (Condition code -- 12)

DUPLICATE STEP NAME IN SYSPSD 'name'

Explanation: A module to be written on SYS000 by the compiler has the same step name (as specified in the EXEC FORTRAN statement) as a module already on SYS000. The name in question appears in the message between the single quotes. Compilation continues. (Condition code -- 16)

EQUIVALENCE ALLOCATION ERRORS

Explanation: This message is produced when there is a conflict between two EQUIVALENCE groups, or if there is an incompatible boundary alignment in an EQUIVALENCE group. A list of the variables that could not be allocated according to source statement specifications follows the message. (Condition code -- 4)

EQUIVALENCE DEFINITION ERRORS

<u>Explanation:</u> This message denotes an error in an EQUIVALENCE group when an array element is outside the array. A list of the errors follows the message. (Condition code -- 4)

EXIT ROLL FULL, COMPILATION TERMINATED

<u>Explanation:</u> This message is produced when the EXIT roll (an internal table used by the compiler) has exceeded the amount of main storage assigned for it. Compilation is terminated. (Condition code -- 16)

FUNCTION ENTRIES UNDEFINED

<u>Explanation:</u> The program being compiled is a FUNCTION subprogram, but there is no scalar with the same name as the FUNCTION nor is there a definition for each ENTRY. A list of the undefined names follows the message. (Condition code -- 0)

ID CONFLICT

<u>Explanation:</u> The name of a variable or subprogram has been used in conflict with the type that was defined for it in a previous statement. For example, the name listed in a CALL statement is the name of a variable, not a subprogram; or a single name appears more than once in the dummy list of a statement function; or a name listed in an EXTERNAL statement has already been defined in another context. (Condition code -- 12)

ILLEGAL LABEL

<u>Explanation:</u> Invalid use of a statement label has occurred; for example, an attempt has been made to branch to the label of a FORMAT statement. (Condition code -- 12)

ILLEGAL STA.

<u>Explanation:</u> The context in which a statement has been used is invalid. For example, the statement "s" in a Logical IF statement (the result of the true condition) is a Specification statement, a DO statement, etc.; or an ENTRY statement appears in a main program. (Condition code -- 12)

ILLEGAL STA. WRN.

<u>Explanation:</u> A RETURN statement appears in a main program; or a RETURN <u>i</u> statement appears in a FUNCTION subprogram. (Condition code -- 0)

ILLEGAL TYPE

<u>Explanation:</u> The variable in an Assigned GO TO statement is not an integer variable; or, in an assignment statement, the variable on the left side of the equal sign is of logical type and the expression on the right side is not. (Condition code -- 12)

I/O ERROR SYS001

<u>Explanation:</u> An unrecoverable output error has occurred on SYS001. Compilation is terminated. (Condition code -- 16)

I/O ERROR SYSPSD ON INPUT

<u>Explanation:</u> An unrecoverable input error has occurred on SYSPSD. Compilation continues. (Condition code -- 16)

I/O ERROR SYSPSD ON OUTPUT

<u>Explanation:</u> An unrecoverable output error has occurred on SYSPSD. Compilation continues. (Condition code -- 16)

LABEL

<u>Explanation:</u> A statement that should be labeled is not. For example, a FORMAT statement or a statement following a GO TO statement is not labeled. (Condition code -- 0)

NAME LENGTH

<u>Explanation:</u> The name of a variable, COMMON block, NAMELIST, or subprogram exceeds six characters in length; or two variable names appear in an expression without a separating operation symbol. (Condition code -- 4)

NO CORE AVAILABLE

<u>Explanation:</u> This message is produced when the program being compiled exhausts the supply of main storage available to the compiler. (Condition code -- 16)

NO END CARD

<u>Explanation:</u> The set of source statements does not contain an END statement. (Condition code -- 0)

NUMBER ARG

<u>Explanation:</u> A reference to a library subprogram specifies an incorrect number of arguments. (Condition code -- 4)

OPEN ERROR POLISH FILE, COMPILATION TERMINATED

<u>Explanation:</u> An error has occurred during an attempt to open SYS001, which is used by the compiler to accommodate its strings of Polish notation. Compilation is terminated. (Condition code -- 16)

ORDER

<u>Explanation:</u> Source statements are used in an improper sequence. For example, an IMPLICIT statement appears as other than the first statement in a main program or the second statement in a subprogram; or an ENTRY statement appears within a DO loop. (Condition code -- 12)

POLISH FILE READ ERROR, COMPILATION TERMINATED

<u>Explanation:</u> An unrecoverable input error has occurred on SYS001. Compilation is terminated. (Condition code -- 16)

POLISH READ END OF FILE

<u>Explanation:</u> An end-of-file mark was erroneously read on SYS001 by the compiler as it was reading Polish notation. Compilation is terminated. (Condition code -- 16)

SIZE

<u>Explanation:</u> A number used in a source statement does not conform to the values allowed for its use. For example, a label used in a statement exceeds the maximum value for a statement label; or the size specification in an Explicit Specification statement is not one of the acceptable values; or an integer constant is too large. (Condition code -- 12)

SUBSCRIPT

<u>Explanation:</u> The number of subscripts used in an array reference is either too large or too small for the array. (Condition code -- 12)

SYNTAX

<u>Explanation:</u> A statement or part of a statement does not conform to the FORTRAN IV syntax. For example, a statement cannot be identified; or a nondigit appears in the label field; or fewer than three labels follow the expression in an Arithmetic IF statement; or a constant that begins with a decimal point does not have a digit as its second character. (Condition code -- 12)

SYSIPT I/O ERROR, CARD DELETED

<u>Explanation:</u> An input/output error occurred while the compiler was reading a card from SYSIPT. The card being read is ignored. Compilation continues. (Condition code -- 16)

SYSIPT I/O ERROR, CARD DELETED COMPILATION TERMINATED

<u>Explanation:</u> An input/output error occurred while the compiler was reading a card from SYSIPT. Either the card being read was a comments card that preceded all source language statements or this was the tenth input/output error on SYSIPT. Compilation is terminated. (Condition code -- 16)

UNCLOSED DO LOOPS

<u>Explanation:</u> This message is produced if one or more DO loops are initiated, but their terminal statements do not exist. A list of the undefined labels that appeared in the DO statements

follows the message. (Condition code -- 12)

UNDEFINED LABELS

Explanation: Labels used in the set of source statements are not defined. A list of the undefined labels follows the message. (Condition code -- 12)

UNDIMENSIONED

Explanation: The use of a variable name indicates an array (that is, subscripts follow the name), but the variable has not been dimensioned. (Condition code -- 12)

UNRECOVERABLE ERROR LINK EDIT, OUTPUT DELETED

Explanation: The LINK option was specified or assumed in the EXEC FORTRAN statement, but an unrecoverable output error has occurred on SYS000. The writing of the module on SYS000 is terminated. Compilation continues. (Condition code -- 16)

UNRECOVERABLE PUNCH ERROR, OUTPUT DELETED

Explanation: The DECK option was specified in the EXEC FORTRAN statement, but an unrecoverable error has occurred on SYSPCH. The punching of the requested deck is terminated. (Condition code -- 0)

WORK ROLL FULL, COMPILATION TERMINATED

Explanation: This message is produced when the WORK roll (an internal table used by the compiler) has exceeded the amount of main storage assigned for it. Compilation is terminated. (Condition code -- 16)

LINKAGE EDITOR MESSAGES

Linkage editor error messages are written on SYSLST during the linkage editing job step. These messages apply to the ESD, TXT, REP, RLD, and END statements produced by the language processors and to the linkage editor control statements.

In most cases, an error message is accompanied by a listing of the statement containing or causing the error.

Some of the statements reproduced in an error listing do not correspond exactly to the actual input statement. This is because the linkage editor does some processing of the statements in the statement input area, and some fields have been altered by the time an error is detected. This applies mainly to the byte count, length, and type fields of the ESD statement. In no case, however, should there be any problem identifying the statement.

For TXT and RLD cards, only the first 36 columns of the variable field are printed. For a REP card error, other than a sequence error, the error code is printed immediately after the REP card listing. The notation FOR REP CARD is printed next to the error code.

Error messages fall into three categories:

1. Warning Messages. These are produced to call a programmer's attention to a condition that may or may not represent an error. They do not affect continuation of the job step.

2. Severe Errors. These messages are written when the linkage editor detects errors that would prohibit successful execution of the program. Linkage editing continues, but its output is flagged so that it will not be accepted for execution.

3. Termination Messages. These messages are written when conditions develop that require immediate termination of the job. Most of these are not the fault of the program, but represent an inability of the system to continue functioning properly.

Most of these messages are written in the format KAxxI, where KA identifies a linkage editor error message, xx represents a numeric code identifying a particular message, and I means the message is for information. A few messages include written text, as discussed in the following list of numeric codes and their corresponding messages.

The last line of any linkage editor listing contains the message LINKAGE EDITOR HIGHEST SEVERITY WAS xx, where xx indicates the severity level, as follows:

0 indicates no significant errors and execution of the job may continue.

4 indicates that one or more warning messages have been printed, but execution may continue.

12 indicates that the program contains errors that prevent its execution. The phase or phases being edited are not entered in the phase library.

16 indicates that a termination condition exists, and editing has not been completed. No phases have been entered in the phase library. The job is canceled.

Warning Messages, Severity Level 4

The following messages are designed solely to call a programmer's attention to an unusual condition.

Error Code	Condition
KA01I	A COMMON control section has the same name as a regular control section, but their lengths differ. Space has been reserved for the longer.
KA02I	Two or more control sections in different phases have the same name.
KA03I	The previous control section had a length of 0. If this condition is not intentional, it could have been caused by an error of the language processor.
KA04I	An END card that should indicate the length of a control section does not. The length of the last or only control section in the external symbol dictionary is 0. This does not represent an actual error if the control section contains only instructions to the language processor that do not require any main storage space.
KA05I	A control section name in a CSECT list in an INCLUDE statement is duplicated.

Severe Error Messages, Severity Level 12

The following messages document errors that prohibit execution of the program. Linkage editing continues.

Error Code	Condition
KA11I	The type field of an ESD statement contains an invalid entry. This usually represents a language processor error.
KA12I	A COMMON control section has the same name as an entry point.
KA13I	A Label Definition type entry in an ESD statement does not point to a Section Definition or Private Code type entry. This usually represents a language processor error.
KA14I	An origin for a control section that should be aligned on a double word boundary is not so aligned. This usually represents a language processor error.
KA15I	An ESD statement indicates that a private code section is named. A private code section cannot be named. This usually represents a language processor error.
KA16I	An SD, LD, or ER type entry with a blank name field is invalid. This usually represents a language processor error.
KA18I	An entry point name improperly duplicates another entry point or control section name.
KA19I	Two or more ESD statements in the same input module have the same identification number. This usually represents a language processor error.
KA35I	System unit SYS000 or SYSREL contains a statement that is either invalid or out of sequence. Module cards must be in the order ESD, TXT, RLD, REP, and END.
KA36I	A MODULE statement was not followed by a statement with the 12-2-9 loader identification punch in its first column.
KA37I	The linkage editor has read beyond the last block of an input module. The input deck is out of sequence, or an END card is missing.
KA38I	A statement on SYSIPT is invalid or out of sequence.

Error Code	Condition
KA39I	A job control statement other than the /* (end-of-data) statement has been read. The /* statement is the only job control statement that should be read by the linkage editor.
KA40I	A hexadecimal field in a PHASE or REP card contains an invalid character.
KA41I	A module contains an ESD identification number of 0 or greater than 255. Except for REP cards, this usually represents a language processor error.
KA42I	A TXT, REP, RLD, or END statement contains an ESD identification number that is not in the module's external symbol dictionary. Except for a REP card, it may represent a language processor error. For a TXT or REP card, it also may mean that the ESD number does not point to a control section. This message is written only for the first TXT or REP card containing the error even though the following cards may contain the same erroneous number.
KA43I	The operand field of a control statement extends beyond column 71; the variable field of a REP card extends beyond column 71; or the last field in a REP card contains a number of characters that is not divisible by four.
KA44I	An entry point in the external symbol dictionary has an ESD number that should point to a control section, but the control section that it points to is not in the external symbol dictionary. This may represent the loss of cards or a language processor error. This error is detected when an END card is processed, so the message is listed with the END card.
KA45I	The CSECT name list of an INCLUDE statement contains one or more control section names that are not in the module. This error code is printed with the END card since the error cannot be detected earlier. In some cases, this message is given because the control section in the external symbol dictionary was not processed as the result of another error condition, usually made by a language processor. In this case, the ESD card for the control section has been printed with another error code. If a phase map has been produced, the control sections specified in the INCLUDE statement that were actually included in the phase are listed.
KA46I	An RLD statement contains a position pointer to an ESD number in the ESD dictionary that is not of the SD or PC type. This usually represents a language processor error.
KA47I	An entry in the operand field of a linkage editor control statement contains too many characters.
KA48I	A required entry is missing from the operand field of a linkage editor control statement.
KA49I	A linkage editor control statement contains an invalid delimiter, or a required delimiter is missing.
KA50I	A decimal field in a PHASE statement contains a non-decimal character.
KA51I	The third specification in the operand field of a PHASE statement is invalid. Only NOAUTO can be specified in this field.
KA52I	A name in a PHASE or MODULE statement contains an invalid character.
KA53I	Two or more phases in the program have the same name.
KA54I	A PHASE statement with an * or S origin also has a phase qualifier. This is permitted only when a control section or entry point is specified as the origin.
KA55I	A symbol specified in a PHASE statement for the origin of the phase was not defined previously.
KA56I	A PHASE statement specifies a negative origin.

Error Code	Condition
KA57I	The END statement for the previous phase contains an invalid entry in its transfer address field.
KA58I	The previous phase contained no text. This may occur when the linkage editor is unable to find the modules named in an INCLUDE statement.
KA59I	The entry point specified in an ENTRY statement is not the name of a properly defined entry point or control section.
KA60I	A TXT or REP statement contains a load address outside the limits of the current phase. This usually represents a language processor error, when it is in a TXT statement.
KA61I	The program calls for a phase size greater than 368,640 bytes.
KA62I	The control section name field of an INCLUDE statement contains the names of more than five control sections.
KA63I	A specification other than R or L appears as the second operand of an INCLUDE statement.
KA64I	A module named in an INCLUDE statement cannot be found in the place indicated by the R or L specification.
KA65I	The linkage editor has read a job control statement for the next job step and is unable to save it in the user communication region. When the linkage editor reads a job control statement at the end of the job step, it attempts to save it for the job control processor. This message is written when the attempt to store it in the user communication region results in an error return.
KA66I	A PHASE statement identifies a phase as ROOT but also specifies a phase qualifier or relocation factor.

Termination Messages, Severity Level 16

The following messages indicate error conditions so severe that the linkage editor cancels the job immediately. In addition to the error code, the message LINKAGE EDITOR CANNOT CONTINUE is printed.

Error Code	Condition
KA80I	End of extent was detected during a write operation. The output data set is not large enough.
KA81I	A permanent transmission error was detected during an input/output operation.
KA82I	An input/output operation terminated without transmitting any data.
KA83I	An input/output operation terminated because of an invalid command.
KA84I	An input/output operation terminated with an incorrect length condition.
KA86I	There is no room left in the SYSPSD directory to list the module specified in a MODULE statement; or an invalid end-of-extent condition was detected reading the last block of the directory. This means a module cannot be inserted with other modules on SYS000 for inclusion in a program.
KA87I	An invalid end-of-extent condition was detected while reading SYSAB2 or the directories on SYSPSD or SYSREL.
KA88I	No phase can be created because there are no entries in the SYSPSD directory. This message also appears when the name field of an entry contains blanks. The EXEC statement name field was blank when the module was assembled or compiled.
KA90I	The linkage editor's control dictionary and linkage table are full. The program probably contains too many control sections and entry points. A maximum of 2047 control dictionary entries is permitted. If there is no ROOT phase, the maximum is 2048.

Error Code	Condition
KA91I	The program specifies a phase name that duplicates the name of a phase already resident in the phase library.
KA92I	There is not enough room in the phase library directory for all the phases in this program.
KA93I	The system is unable to open the SDS000 or SDS001 data sets. The volumes containing these data sets may not be mounted, symbolic unit SYS000 or SYS001 may have been reassigned, or an error condition may have developed during opening.
KA94I	The system is unable to close SYS000 or SYS001. This usually indicates that a system error condition developed during the job step.
KA95I	SYS001 is assigned to a 7-track tape without the convert feature on; or SYS000 and SYS001 are assigned to the same data set.

Text Messages

The following messages are written by the linkage editor. In some cases, as indicated, the phase output is flagged so that it cannot be executed, but linkage editing is not interrupted.

xxxx ILLEGAL OPTION FOR LINKAGE EDITOR

Explanation: This message appears when the EXEC LNKEDT statement contains an invalid parameter. The incorrect parameter is written at the start of the message.

xxxx UNRESOLVED ADDRESS CONSTANTS

Explanation: This message appears when a control section contains an address constant for an external symbol in another module, and the linkage editor is unable to supply an address. The number of such unresolved external references in the program is written at the start of the message. If MAP is specified, a list of unresolved symbols is written. The phase output is flagged so it cannot be executed.

xxxx ADDRESS CONSTANTS OUTSIDE LIMITS OF PHASE

Explanation: This message is written when the program contains address constants referring to points outside the phase that contains the address constant. This condition usually represents a language processor error. The phase output is flagged so it cannot be executed.

The following messages are written only if the MAP option has been specified in the EXEC LNKEDT statement. They are warning messages and do not prevent linkage editing or execution.

ROOT PHASE OVERLAID BY ANOTHER PHASE

Explanation: The program specifies a phase origin that would overlay all or part of a phase that has been designated a root phase. The phase that causes the overlay condition is marked by the word OVERROOT in the listing.

POSSIBLE INVALID ENTRY POINT DUPLICATION IN INPUT

Explanation: The input contains possible duplication of entry point names. This may occur when control sections from a single module are being split among different phases, in which case the message can be ignored. When this message appears, one or more entry points in the input have been ignored. The phase map shows whether an entry point for a certain control section is missing. If it is, any reference to the entry point has probably been resolved to the wrong location.

PHASE EXECUTION DIAGNOSTIC MESSAGES

During phase execution, three types of diagnostic messages are produced:

- Execution error messages.
- Program interrupt messages.
- Operator messages.

Execution Error Messages

In the following text, the error codes are given with an explanation describing the type of error. Preceding the explanation, an abbreviated name is given indicating the origin of the error. For any phase execution error, a condition code of 16 is generated and the job is terminated.

The abbreviated name for the origin of the error is:

IBC - BOAFCOMH routine (performs interruption, conversion, and error procedures).

FIOCS - BOAFIOSH routine (performs input/output operations for FORTRAN phase execution).

NAMEL - BOANAMEL routine (performs the processing of NAMELIST specifications).

DIOCS - BOADIOCSE routine (performs direct-access input/output operations for FORTRAN phase execution).

LIB - FORTRAN-supplied library. In the explanation of the messages, the module name is given followed by the entry point name(s) enclosed in parentheses.

OA200I

Explanation: FIOCS -- An attempt was made to read from a data set for which input operations are not allowed.

OA201I

Explanation: FIOCS -- An attempt was made to write into a data set for which output operations are not allowed.

OA202I

Explanation: FIOCS -- A READ or WRITE operation was attempted on a data set whose most recent operation resulted from an ENDFILE statement.

OA203I

Explanation: FIOCS -- An attempt was made to rewind, backspace, or write an end-of-file mark on one of the system units SYSOPT, SYSPCH, or SYSIPT.

OA204I

Explanation: FIOCS -- An attempt was made to rewind, backspace, or write an end-of-file mark on a data set described by a DEFINE FILE statement.

OA205I

Explanation: FIOCS -- A data set reference number greater than 15 has been used in an input/output statement.

OA206I

Explanation: FIOCS -- An attempt was made to open a data set, but the data set could not be found. This message appears when a data set reference number not valid for the installation has been used in an input/output statement.

OA207I

Explanation: FIOCS -- A label error was detected when a data set was opened.

OA208I

Explanation: FIOCS -- An input/output request has been made that is invalid for a data set.

OA209I

Explanation: IBC -- There is insufficient main storage to allocate one request control block and one 360-byte buffer.

OA211I

Explanation: IBC -- An invalid character has been detected in a FORMAT statement.

OA212I

Explanation: IBC -- An attempt has been made to read or write, under FORMAT control, a record that exceeds the buffer length (360 bytes).

OA213I

Explanation: IBC -- The input list in an input/output statement without a FORMAT specification is larger than the logical record.

OA215I

Explanation: IBC -- An invalid character exists for the decimal input corresponding to an I, E, F, or D format code.

OA216I

Explanation: IBC -- An invalid sense-light number was detected in the argument list in a call to the SLITE or SLITET subprogram.

OA217I

Explanation: IBC -- An end-of-data condition was sensed during a READ operation or an end-of-extent condition was detected during a WRITE operation.

OA218I

Explanation: IBC -- A permanent input/output error has been encountered.

OA219I

Explanation: IBC -- A boundary error has occurred but the boundarv alignment routine could not be found in the phase library.

OA220I

Explanation: IBC -- A boundary error has occurred but there is not enough space in main storage for the boundary alignment routine to be loaded.

OA221I

Explanation: NAMEL -- An input variable name exceeds eight characters.

OA222I

Explanation: NAMEL -- An input variable name is not in the NAMELIST dictionary, or an array is specified with an insufficient amount of data.

OA223I

Explanation: NAMEL -- An input variable name or a subscript has no delimiter.

OA224I

Explanation: NAMEL -- A subscript is encountered after an undimensioned input name.

OA225I

Explanation: IBC -- An invalid character is encountered on input for the Z format code.

OA231I

Explanation: DIOCS -- Direct-access input/output statements are used for a sequential data set.

OA232I

Explanation: DIOCS -- The relative position of a record is not a positive integer, or the relative position exceeds the number of records in the data set.

OA233I

Explanation: DIOCS -- The record length specified in the DEFINE FILE statement exceeds the physical limitation of available main storage.

OA234I

Explanation: DIOCS -- Direct access input/output statements have been used for one of the system units SYSIPT, SYSPCH, or SYSOPT.

OA235I

Explanation: DIOCS -- A data set referred to in a direct access input/output statement was not previously described in a DEFINE FILE statement.

OA236I

Explanation: DIOCS -- A data set reference number used in a DEFINE FILE statement has no corresponding symbolic unit.

OA241I

Explanation: LIB -- For an exponentiation operation (i**j) in the subprogram BOAFIXPI (FIXPI#) where i and j represent integer variables or integer constants, the value of i is zero and the value of j is less than or equal to zero.

OA242I

Explanation: LIB -- For an exponentiation operation (r**j) in the subprogram BOAFRXPI (FRXPI#), where r represents a real*4 variable or integer constant, the value of r is zero and the value of j is less than or equal to zero.

OA243I

Explanation: LIB -- For an exponentiation operation (d**j) in the subprogram BOAFDXPI (FDXPI#), where d represents a real*8 variable or real*8 constant and j represents an integer variable or integer constant, the value of d is zero and the value of j is less than or equal to zero.

OA244I

Explanation: LIB -- For an exponentiation operation (r**s) in the subprogram BOAFRXPR (FRXPR#), where r and s represent real*4 variables or real*4 constants, the value of r is zero and the value of s is less than or equal to zero.

OA245I

Explanation: LIB -- For an exponentiation operation (d**p) in the subprogram BOAFDXPD (FDXPD#), where d and p represent real*8 variables or real*8 constants, the value of d is zero and the value of p is less than or equal to zero.

OA246I

Explanation: LIB -- For an exponentiation operation (z**j) in the subprogram BOAFCXPI (FXCPI#), where z represents a complex*8 variable or integer constant, the value of z is zero and the value of j is less than or equal to zero.

OA247I

Explanation: LIB -- For an exponentiation operation (z**j) in the subprogram BOAFCDXI (FCDXI#), where z represents a complex*16 variable or complex*16 constant and j represents an integer variable or integer constant, the value of z is zero and the value of j is less than or equal to zero.

OA251I

Explanation: LIB -- In the subprogram BOASSQRT (SQRT), the value of the argument is less than zero.

OA252I

Explanation: LIB -- In the subprogram BOASEXP (EXP), the value of the argument is greater than 174.673.

OA253I

Explanation: LIB -- In the subprogram BOASLOG (ALOG and ALOG10), the value of the argument is less than or equal to zero. Because this subprogram is called by an exponential subprogram, this message also indicates that an attempt has been made to raise a negative base to a real power.

OA254I

Explanation: LIB -- In the subprogram BOASSCN (SIN and COS), the absolute value of an argument is greater than or equal to $2^{18}\pi$.
($2^{18}\pi$ = .82354966406249996D+06)

OA255I

Explanation: LIB -- In the subprogram BOASATN2, when entry name ATAN2 is used, the value of both arguments is zero.

OA256I

Explanation: LIB -- In the subprogram BOASSCNH (SINH or COSH), the value of the argument is greater than or equal to 174.673.

OA257I

Explanation: LIB -- In the subrpogram BOASASCN (ARCSIN or ARCOS), the absolute value of the argument is greater than one.

OA258I

Explanation: LIB -- In the subprogram BOASTNCT (TAN or COTAN), the absolute value of the argument is greater than or equal to $2^{18}\pi$.
($2^{18}\pi$ = .82354966406249996D+06)

OA259I

Explanation: LIB -- In the subprogram BOASTNCT (TAN or COTAN), the value of the argument is too close to one of the singularities ($\pi/2$, $3\pi/2$, ... for the tangent; π, 2π,... for the cotangent).

OA261I

Explanation: LIB -- In the subprogram BOALSQRT (DSQRT), the value of the argument is less than zero.

OA262I

Explanation: LIB -- In the subprogram BOALEXP (DEXP), the value of the argument is greater than 174.673.

OA263I

Explanation: LIB -- In the subprogram BOALLOG (DLOG and DLOG10), the value of the argument is less than or equal to zero. Because the subprogram is called by an exponential subprogram, this message also indicates that an attempt has been made to raise a negative base to a real power.

OA264I

Explanation: LIB -- In the subprogram BOALSCN (DSIN and DCOS), the absolute value of the argument is greater than or equal to $2^{50}\pi$.
($2^{50}\pi$ = .35371188737802239D+16)

OA265I

Explanation: LIB -- In the subprogram BOALATN2, when entry name DATAN2 is used, the value of both arguments is zero.

OA266I

Explanation: LIB -- In the subprogram BOALSCNH (DSINH or DCOSH), the absolute value of the argument is greater than or equal to 174.673.

OA267I

Explanation: LIB -- In the subprogram BOALASCN (DARSIN or DARCOS), the absolute value of the argument is greater than one.

OA268I

Explanation: LIB -- In the subprogram BOALTNCT (DTAN or DCOTAN), the absolute value of the argument is greater than or equal to $2^{50}\pi$.
($2^{50}\pi$ = .35371188737802239D+16)

OA269I

Explanation: LIB -- In the subprogram IHCLTNCT (DTAN or DCOTAN), the value of the argument is too close to one of the singularities ($\pi/2$, $3\pi/2$, ... for the tangent; π, 2π,... for the cotangent).

OA271I

Explanation: LIB -- In the subprogram BOACSEXP (CEXP), the value of the real part of the argument is greater than 174.673.

OA272I

Explanation: LIB -- In the subprogram BOACSEXP (CEXP), the absolute value of the imaginary part of the argument is greater than or equal to $2^{18}\pi$.
($2^{18}\pi$ = .82354966406249996D+06)

OA273I

Explanation: LIB -- In the subprogram BOACSLOG (CLOG), the value of both the real and imaginary parts of the argument is zero.

OA274I

Explanation: LIB -- In the subprogram BOACSSCN (CSIN or CCOS), the absolute value of the real part of the argument is greater than or equal to $2^{18}\pi$.
($2^{18}\pi$ = .82354966406249996D+06)

OA275I

Explanation: LIB -- In the subprogram BOACSSCN (CSIN or CCOS), the absolute value of the imaginary part of the argument is greater than 174.673.

OA281I

Explanation: LIB -- In the subprogram BOACLEXP (CDEXP), the value of the real part of the argument is greater than 174.673.

OA282I

Explanation: LIB -- In the subprogram BOACLEXP (CDEXP), the absolute value of the imaginary part of the argument is greater than or equal to $2^{50}\pi$.
($2^{50}\pi$ = .35371188737802239D+16)

OA283I

Explanation: LIB -- In the subprogram BOACLLOG (CDLOG), the value of both the real and imaginary parts of the argument is zero.

OA284I

Explanation: LIB -- In the subprogram BOACLSCN (CDSIN or CDCOS), the absolute value of the real part of the argument is greater than or equal to $2^{50}\pi$.
($2^{50}\pi$ = .35371188737802239D+16)

OA285I

Explanation: LIB -- In the subprogram BOACLSCN (CDSIN or CDCOS), the absolute value of the imaginary part of the argument is greater than 174.673.

OA290I

Explanation: LIB -- In the subprogram BOASGAMA (GAMMA), the value of the argument is outside the valid range. (Valid range: $2^{-252}<x<57.5744$)

OA291I

Explanation: LIB -- In the subprogram BOASGAMA (ALGAMA), the value of the argument is outside the valid range. (Valid range: $0<x<4.2937x10^{73}$)

OA300I

Explanation: LIB -- In the subprogram BOALGAMA (DGAMMA), the value of the argument is outside the valid range. (Valid range: $2^{-252}<x<57.5744$)

OA301I

Explanation: LIB -- In the subprogram BOALGAMA (DLGAMA), the value of the argument is outside the valid range. (Valid range: $0<x<4.2937\times10^{73}$)

Program Interrupt Messages

The following text describes program interrupt messages. The format of these messages is described in "System Output."

Specification Exception: The specification exception, assigned code number 6, is recognized whenever a data address does not specify an integral boundary for that unit of information. A specification error would occur, for example, during the execution of the following program segment:

```
DOUBLE-PRECISION D,E
COMMON A,B,C
EQUIVALENCE (B,D)
D = 3.0D02
```

Fixed-Point-Divide Exception: The fixed-point-divide exception, assigned code number 9, is recognized whenever division of a fixed-point number by zero is attempted. A fixed-point-divide exception would occur during execution of the following statements:

```
J = 0
I = 7
K = I/J
```

Exponent-Overflow Exception: The exponent-overflow exception, assigned code number C, is recognized whenever the result of a floating-point addition, subtraction, multiplication, or division is greater than or equal to 16^{63} (approximately 7.2×10^{75}). For example, an exponent-overflow would occur during execution of the statement:

```
A = 1.0E+75 + 7.2E+75
```

Exponent-Underflow Exception: The exponent-underflow exception, assigned code number D, is recognized whenever the result of a floating-point addition, subtraction, multiplication, or division is less than 16^{-65} (approximately 5.4×10^{-79}). An exponent-underflow exception would occur during execution of the statement:

```
A = 3.2E-40*5.4E-50
```

Floating-Point-Divide Exception: The floating-point-divide exception, assigned code number F, is recognized when division of a floating-point number by zero is attempted. A floating-point-divide exception would occur during execution of the following statements:

```
B = 0.0
A = 1.0
C = A/B
```

Operator Messages

Operator messages for STOP and PAUSE are generated during phase execution.

The message for a PAUSE can be one of the forms:

```
PAUSE n
PAUSE 'message'
PAUSE 0
```

where:

n	is the 1- through 5-digit unsigned integer constant specified in a PAUSE source statement
message	is the literal constant specified in a PAUSE source statement
0	is printed when a PAUSE statement that does not specify an integer or literal constant is executed

Explanation: The programmer should give instructions that indicate the action to be taken by the operator when the PAUSE is encountered.

User Response: To resume execution, the operator presses the EOB key on the console keyboard.

The message for a STOP statement can be one of the forms:

```
STOP n
STOP 0
```

where:

n	is the 1- through 5-digit unsigned integer constant specified in a STOP source statement
0	is printed when a STOP statement that does not specify an integer constant is executed

User Response: None

C28-6813-0

Printed in U.S.A. C28-6813-0

IBM®

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]

READER'S COMMENTS

Title:	IBM System/360 Model 44 Programming System Guide to System Use for FORTRAN Programmers	Form:	C28-6813-0

Your comments assist us in improving the usefulness of our publications; they are a major part of the input used for technical newsletters and revisions.

Please do not use this form for technical questions about the system; it only delays the response. Instead, direct your technical questions to your local IBM representative.

Corrections or clarifications needed:

<u>Page</u> <u>Comment</u>

If you wish a reply, please include your name and address below:

C28-6813-0

fold fold

FIRST CLASS
PERMIT NO. 33504
NEW YORK, N.Y.

BUSINESS REPLY MAIL

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .

IBM CORPORATION

1271 AVENUE OF THE AMERICAS
NEW YORK, N.Y. 10020

Attention: PUBLICATIONS

fold fold

IBM®

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]