File Number S360-25
Form C24-5038-0

DOS
TOS

IBM Systems Reference Library

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

This publication describes the procedures for compiling and executing FORTRAN IV programs under control of the Disk Operating System or Tape Operating System. Its purpose is to guide the programmer with examples and techniques of the FORTRAN IV language. It also exposes the user to the components of the Control Program and facilities of the System/360 Disk and Tape Operating Systems.

Douglas Albert

PREFACE

This publication is intended for FORTRAN programmers and for systems programmers who require information about the FORTRAN compiler that is contained in the Disk Operating System or the Tape Operating System. The presentation of information in this publication assumes that the reader is thoroughly familiar with the FORTRAN language as described in the following publication:

IBM System/360 Disk and Tape Operating Systems FORTRAN IV Specifications, Form C24-5014

The following publications listed contain information a programmer may require under certain circumstances. When such information is required, the text refers to the proper publication and also to this Preface, which contains the full title and form number of the required publication.

IBM System/360 Tape Operating System, Supervisor and Input/Output Macros, Form C24-3432

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

IBM System/360 Tape Operating System, System Control and System Service Programs, Form C24-3431.

IBM System/360 Tape Operating System, Operating Guide, Form C24-5021.

IBM System/360 Disk Operating System, Supervisor and Input/Output Macros Form C24-3429

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

IBM System/360 Disk Operating System, System Control and System Service Programs Form C24-3428

IBM System/360 Disk Operating System, Operating Guide, Form C24-5022

IBM System/360 Disk and Tape Operating Systems Specifications, Utility Programs Form C24-3465

IBM System/360 Bibliography, Form A22-6822.

SIMPLIFIED TERMINOLOGY

The titles of some IBM programming systems have been simplified as shown below:

former:	IBM System/360 Basic Operating System (8K Disk)
new:	IBM System/360 Basic Operating System
former:	IBM System/360 Basic Operating System (16K Tape)
new:	IBM System/360 Tape Operating System
former:	IBM System/360 Basic Operating System (16K Disk)
new:	IBM System/360 Disk Operating System

First Edition, May 1966

Significant changes and additions to the specifications contained in this publication will be reported in subsequent revisions or Technical Newsletters.

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

A form is provided at the back of this publication for readers' comments. If the form has been removed, comments may be addressed to IBM Corporation, Programming Publications, Endicott, New York 13760.

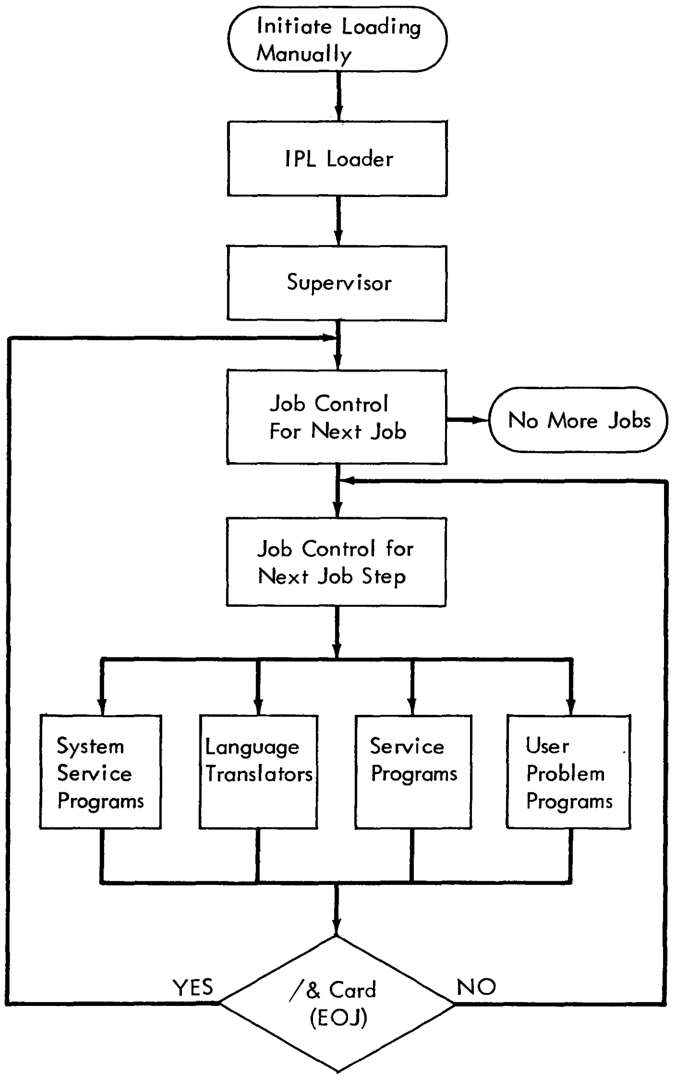


Figure 1. DOS or TOS System Flow

This publication describes the compilation and execution of FORTRAN programs under control of the Disk Operating System (DOS) or Tape Operating System (TOS). This introduction is intended for those readers who are not familiar with operating systems. Readers with a knowledge of an operating system may skip this section. No attempt is made to describe all of the features and uses of the Disk or Tape Operating System. Only that information required by a FORTRAN programmer is included.

An operating system is used to control the operation of a computing system. There are many reasons for using an operating system. Some of these are:

1. To keep the computer busy.

2. To reduce the chance for operator errors.

3. To handle unusual conditions (for example: division by zero).

The Disk or Tape Operating System keeps the computer busy by initiating the compilation and/or execution of a program as soon as the processing of the previous program has been completed. In addition to saving the time that the computer would be idle while waiting for the operator to initiate the next operation, this procedure eliminates the chance of the operator making an error while initiating the next operation. The DOS or TOS handles unusual conditions either by producing a message noting the error or by correcting it (for example: rereading a tape record if a read error has occurred).

The Disk or Tape Operating System, which is diagrammed in Figure 1, consists of a group of processing programs, together with the Control Program needed to maintain their continuous operation, and system service programs. The processing programs include language translators, service programs, and any problem programs written by the user. The Control Program consists of three components (see Control Program) that prepare and control the execution of all processing programs and problem programs within the DOS or TOS. The system service programs consist of two components (see System Service Programs) that are used to generate the system and to create, edit, and maintain the libraries within the system.

STRUCTURE OF DOS OR TOS

Figure 2 shows the structure of the Disk or Tape Operating System. Each component of the system is described separately below.

CONTROL PROGRAM

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

The control program contained in the DOS or TOS provides automatic transition from

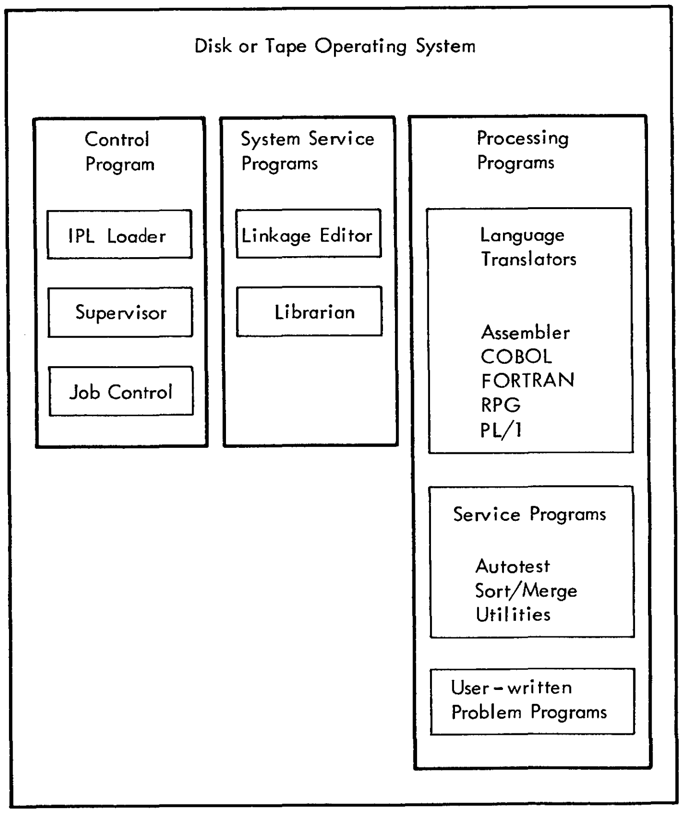


Figure 2. Structure of the Disk or Tape Operating System

program phase to program phase within a processing program, and from processing program to processing program within the system. Once the system has been initialized, job after job can be entered into the system for processing.

The three components of the Control Program are:

1. The Initial Program Loading Loader (IPL Loader)

2. The Supervisor

3. Job Control.

Each component of the Control Program is described in the following sections.

IPL Loader

Operation of the Disk or Tape Operating System is initiated through an initial program loading procedure. The IPL Loader is loaded into main storage from tape or disk storage simply by selecting the address of the unit in the load unit switches on the System/360 console and pressing the load key. The loader than reads the nucleus of the Supervisor from the resident unit into the lower portion of main storage. After sucessfully reading the Supervisor nucleus into main storage, the IPL Loader performs certain initializing and housekeeping functions before control is transferred to the Supervisor, which uses the System Loader to issue a call for Job Control.

Supervisor

The Supervisor is the component of the Control Program that operates with the problem programs. It consists of:

1. Permanent routines that are loaded into main storage during the IPL process and remain there throughout system operation until main storage is cleared.

2. Transient routines that remain on the system tape or disk unit until needed and are then retrieved and loaded into a common transient area.

During the execution of a processing program, control alternates between the processing program and the Supervisor. Figure 3 shows the major division of main storage and the position of the Supervisor.

Job Control

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

SYSTEM SERVICE PROGRAMS

The system service programs provide the functions of:

1. Generating the system.

2. Creating and maintaining the library sections.

3. Editing programs on tape before execution.

The system service programs are the:

1. Linkage Editor

2. Librarian

Linkage Editor

All programs to be linkage edited (put into executable form) are written onto symbolic unit SYS000, which is used as the input unit for the Linkage Editor. The output of the Linkage Editor is also SYS000.

Librarian

This is actually a group of programs used for maintaining DOS or TOS libraries and for providing printed and punched output from the libraries. The libraries are:

1. The core image library

2. The relocatable library

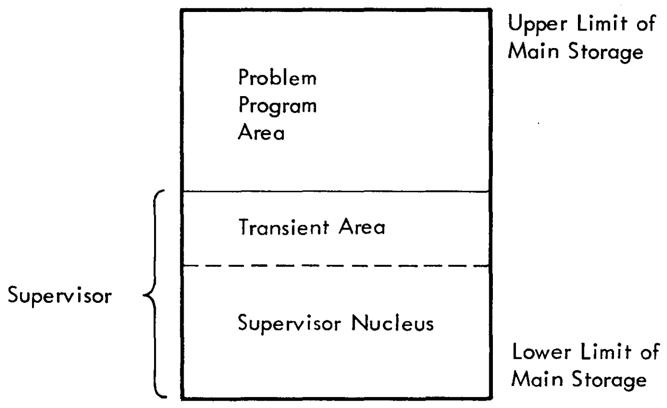


Figure 3. Major Divisions of Main Storage

CORE IMAGE LIBRARY: All permanent programs in the system (IBM-supplied and user programs) are loaded from this library by the Program Fetch routine of the Supervisor. The core image library is required for each resident system. The core image library contains any number of programs each of which is edited to run with the Supervisor. Each program is made up of one or more separate phases, which are executable, nonrelocatable sections of a program in core image form. Associated with each phase is a header record that contains a complete description of the phase.

RELOCATABLE LIBRARY: This library is used for the storage of object modules that are the output of a language translator(s) and can be used for subsequent linkage with other program modules. A module also can be a complete program. The relocatable library is not required for operating a system.

The purpose of the relocatable library is to allow the user to retain frequently used routines and combine them with other modules without requiring recompilation. The routines from the relocatable library are edited onto SYS000 by the Linkage Editor.

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

LIBRARIAN FUNCTIONS: Maintenance and service are the major functions performed for the libraries by the librarian programs. Maintenance includes the addition, deletion, or copying of items in the library. Service includes the translation of information in a library to printed or punched form. Information in a library directory and in header records can also be displayed.

PROCESSING PROGRAMS

The three types of processing programs in the Disk or Tape Operating System are:

1. Language translators

2. Service programs

3. Programs written by the user

Each type of processing program is described in the following sections.

Language Translators

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

The language translators available for the DOS or TOS are: Assembler, COBOL, FORTRAN, RPG (Report Program Generator), and PL/1.

Service Programs

These programs perform such independent operations as sorting and tape to printer. Refer to the publication IBM System/360 Bibliography, listed in the preface, for publications describing service programs.

Programs Written by the User

A user may insert programs he writes into the Relocatable Library so they can be linkage edited with the other programs whenever required. Which of his programs to place in the library will be determined by the user. The choice depends upon conditions existing at his installation and upon the program. For example, a program that calculates stresses in beams might be placed in the library because it could be used in programs for many types of designs, (bridges, buildings, etc.). On the contrary, a program that produces a listing of prime numbers probably would not be placed into the library because it might be run only once and have no use in other programs.

USE OF THE DISK AND TAPE OPERATING SYSTEMS FOR FORTRAN

The processing of programs under control of the Disk or Tape Operating System is specified in control cards, which specify whether a program is to be compiled, linkage edited, and/or executed. Each form of processing is described briefly in the following sections. Details of the control cards and complete information for processing programs, together with examples of each form, appear in the section Processing Jobs.

COMPILING WITH THE DOS OR TOS

Compilation is the process by which a group of source language statements is converted by a language translator into a group of machine language instructions. Ⓐ of Figure 4 shows the compilation of FORTRAN statements. The group of statements that form the input to the compilation is called a source module. The output produced by the compilation is called an object module, which is in a form that is acceptable as input by the Linkage Editor.

A group of FORTRAN statements is called a FORTRAN source module, which is processed by the FORTRAN compiler.

The object module produced by the FORTRAN compiler is a block of relocatable machine instructions assigned to contiguous main-storage locations. An object module consists of an associated control dictionary and one or two control sections. The control dictionary contains information needed by the Linkage Editor to handle cross references between different object modules. The control section contains the actual instructions and data fields that perform the operations required by the FORTRAN source language statements.

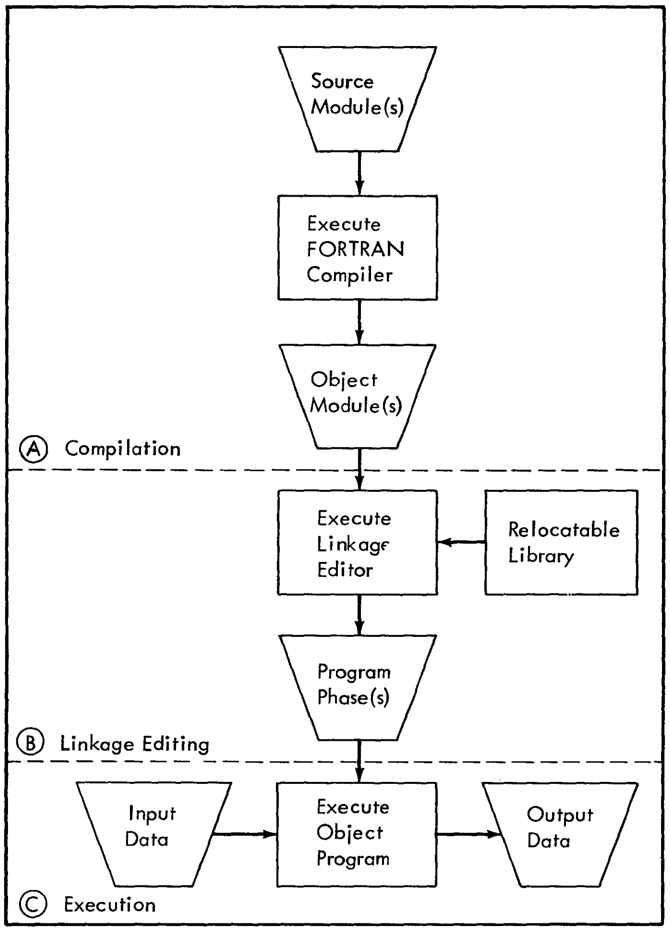


Figure 4. Processing of FORTRAN Programs

LINKAGE EDITING WITH THE DOS OR TOS

An object module produced during compilation (see Compiling with the DOS or TOS) cannot be executed without further processing. To convert it into executable form, and object module must be processed by the Linkage Editor. Ⓑ of Figure 4 shows the linkage editing of an object module. The output produced by the Linkage Editor is called a program phase, which is in executable, nonrelocatable, core image form. A program phase is a program or a portion of a program that is loaded into storage (by the Program Fetch) when a LOAD or FETCH macro instruction is executed.

As shown in Ⓑ of Figure 4 the Linkage Editor can obtain input from the library, which contains items in object module form.

EXECUTING A PROGRAM WITH THE DOS OR TOS

To execute a program, the DOS or TOS obtains a program phase, loads it into main storage, and executes the machine language instructions contained in the phase. If the program consists of more than one phase, the preceding procedure is repeated for each phase until the end of the program is reached. Execution takes place under control of the Supervisor and Job Control. Ⓒ of Figure 4 shows the execution of an object program.

CONTROL CARDS USED WITH THE DOS OR TOS

Through the use of control cards, the programmer indicates what action he wants the Disk or Tape Operating System to take. Control cards may be combined with source modules, programs, and/or input data depending upon the action the user desires.

The formats of the control cards are given in the following sections. No attempt has been made to describe all of the control cards that are recognized by the DOS or TOS. Only those control cards that are most useful to FORTRAN programmers are described. Readers interrested in other control cards are referred to the publication System Control and System Service Programs mentioned in the Preface.

JOB Card

The beginning of every job is specified by a JOB card. A job consists of all modules and control cards between a JOB card and the next /& card (the end-of-job card). A job may be executed as one or more job steps depending upon the number of EXEC cards within the job. Job steps are explained in EXEC Card.

The format of the JOB card is as follows:

```
// JOB jobname [comments]
```

The // characters must be in columns 1 and 2. They must be followed by at least one blank column. The letters JOB must appear after the blank column(s), and they must be followed by at least one blank column. A symbolic name of from one through eight characters must follow the blank column(s). This is the symbolic name of the job.

If the programmer desires to place comments in the JOB card, he must leave at least one blank column following the symbolic name. Any comment must end before column 72.

EXAMPLE: The following is a sample of a JOB card:

```
// JOB PRIMENOS DEPT908 SMITH
```

This card specifies that the name of the job is to be PRIMENOS. The optional comments indicates that the job is for department 908 and was written by Smith.

If the system is equipped with the timer feature (must be requested at system generation time), the time of day prints on SYSLST (if present) and SYSLOG. Refer to the IBM publication Supervisor and I/O Macros referred to in the Preface for the method of setting in the time of day.

EXEC Card

The beginning of every job step is specified by an EXEC card, which causes the execution of a program. A program step consists of control cards that apply to the execution of a program. All control cards for a paricular job step must precede the EXEC card for that job step.

The format of the EXEC card is as follows

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

The // characters must be in columns 1 and 2. They must be followed by at least one blank column. The letters EXEC must appear after the blank column(s), and they must be followed by at least one blank column. If the program to be executed was processed by the Linkage Editor in the preceding job step, the remainder of the EXEC card must be blank. If the program to be executed is in the core image library, the name of the program (from one through eight characters) must be placed in the EXEC card.

EXAMPLES: The following are samples of the various forms of EXEC cards.

1. To compile a FORTRAN source module:

```
// EXEC FORTRAN
```

2. To linkage edit an object module:

```
// EXEC LNKEDT
```

3. To execute a program in the core image library:

```
// EXEC PRIMENOS
```

4. To execute a program immediately after it has been linkage edited:

```
// EXEC LNKEDT

// EXEC
```

DATA FOR EXECUTED PROGRAMS: The data required by the program step to be executed can be placed following the EXEC card. For example, the cards for compiling a FORTRAN source program can be arranged as follows:

```
// EXEC FORTRAN

          READ (3,8)K

        8 FORMAT (I20)

       75 L = K**2
          .
          .
          .
          END

/*
```

Similarly, the data for an object program to be executed can be placed after the EXEC card as follows:

```
// EXEC STRESS

212 415 555      (Card 1)

782 425 227      (Card 2)

897 435 383      (Card 3)

/*
```

Note that the end of the data must be indicated by the use of an end-of-data card (see /* Card). If the program step does not use any input data or if the input data is read from a different input unit, no end-of-data card is required.

/* Card

The data for a program that is executed must be followed by a /* card unless it is the last card of a job. In this case a /& card must be used and the /* card is not required. For example, a FORTRAN source module must be followed by a /* card. Illustrations of the use of /* cards are given in Data for Executed Programs.

The format of the /* card is as follows:

```
/*
```

The / character must be in column 1 and the * character must be in column 2. Information in any other columns is ignored.

/& Card

The last card of every job must be a /& card. All modules and control cards between this card and the preceding JOB card constitute a job.

The format of the /& card is as follows:

```
/&
```

The / character must be in column 1 and the & character must be in column 2. Information in any other columns is ignored. When this card is read by the system, an end-of-job message in the form of EOJ JOBNAME is written on SYSLST and SYSLOG. If the timer feature is present the time of day is printed with this message.

OPTION Card

When the operating system for an installation is generated, certain options regarding listing, and punching are selected as standard. To use other options temporarily, OPTION cards can be included in a job. An option selected by an OPTION card remains in effect until the Disk or Tape Operating System detects either an OPTION card that changes the option or a JOB card. When a JOB card is detected, all options are reset to the standard options. If no OPTION cards are included in a job, the standard options are used.

The format of the OPTION card is as follows:

```
// OPTION option [,option,...]
```

The // characters must be in columns 1 and 2. They must be followed by at least one blank column. The letters OPTION must appear after the blank column(s), and they must be followed by at least one blank column. One or more of the following options listed must follow the blank column(s). If more than one option is specified, they must be separated by commas.

The options apply to the use of the Disk or Tape Operating System for FORTRAN programs. A complete list of all options is given in the publication System Control and System Service Programs mentioned in the Preface. The options are listed in alphabetical order. They may appear in any order in an OPTION card.

Option	Purpose
CATAL	This option causes the output of the FORTRAN compiler to be written on symbolic unit SYS000. Following the execution of the linkage editor, if the librarian is executed, the program will be cataloged into the core-image library.
DECK	To cause the object program to be written on symbolic unit SYSPCH. However, if the LINK option is in effect, the DECK option is ignored.
ERRS	To cause erroneous source program statements, their associated error messages, and summary error messages to be written on symbolic unit SYSLST. Data storage summaries and correct source program statements are not listed. Note that if the LIST option is in effect, the ERRS option is ignored.
LINK	This option causes the output from the FORTRAN compiler (an object module) to be written on symbolic unit SYS000. Note that the Disk or Tape Operating System will ignore this option if the FORTRAN compiler detects errors in the FORTRAN source program.
LIST	To cause the soruce program listing, diagnostic messages, error summaries, and storage map to be written on symbolic unit SYSLST.

Option	Purpose
NODECK	To cause the DECK option to be ignored. However, if the LINK option is not specified, an object deck will always be punched.
NOERRS	To cause the ERRS option to be ignored. When the NOERRS and NOLIST options are used, nothing is written on symbolic unit SYSLST except the storage map unless the compilation is terminated as the result of source program errors, when an appropriate message is written
NOLINK	To cause the LINK option to be ignored.
NOLIST	To cause the LIST option to be ignored. FORTRAN compiler will supress the printing of all information except the compilation terminated message.

Note: These three cards indicate the meaningful combinations of listing output:

```
// OPTION LIST,LINK
// OPTION NOLIST,ERRS,LINK
// OPTION NOLIST,NOERRS,LINK
```

A programmer can build a FORTRAN program as one sequence of FORTRAN source language statements or as a combination of FORTRAN statements and subprograms. The subprograms may be compiled and stored in the relocatable library of the Disk or Tape Operating System.

FORTRAN LANGUAGE

The source language statements for writing FORTRAN programs are described in the publication IBM System/360 Disk and Tape Operating Systems, FORTRAN IV Specifications - see Preface of this publication. We assume the reader is thoroughly familiar with that publication.

Information that will help the FORTRAN programmer to write better FORTRAN programs is contained in the section Programming Suggestions.

PROGRAM FEATURES

Programs to be compiled by the FORTRAN compiler must be written so that they do not violate the conditions stated in Figure 5. If a problem requires a source program that would exceed the maximum conditions stated in Figure 5, we suggest the program be divided into one or more segments.

DATA FOR FORTRAN PROGRAMS

The data that can be processed by FORTRAN programmers must conform to the requirements stated in the FORTRAN language publication. Records are described in the explanation of the FORMAT statement, and storage areas are described by specification statements.

FORTRAN LIBRARY

The FORTRAN library, which is supplied with DOS or TOS FORTRAN as part of the Relocatable Library, contains subprograms that may be used by a programmer to perform frequently needed tests and computations. The Linkage Editor combines these subprograms with the output of the FORTRAN compiler to form the desired object program.

Item	Maximum Number
Size of data area in COMMON	65,532 bytes
Size of data not in COMMON (see Note)	65,532 bytes
Size of object program	65,532 bytes
Variables	500
Arrays	500
Variables and Arrays in COMMON	500
Names plus EQUIVALENCE lists in EQUIVALENCE statements	500
Statement numbers (including one additional statement number for each DO or implied DO in an input/output list)	500
Names in REAL statements	500
Names in INTEGER statements	500
Names in DOUBLE PRECISION statements	500
Unique real constants	500
Unique integer constants	500
Unique double-precision constants	500
References to unique subprogram entry point names (explicit and implicit)	500
Arithmetic statement function definitions	500
Dummy arguments for a subprogram	500
Total arguments to all subprograms and arithmetic statement functions	500
Nesting DO statements	25
Nesting function subprogram references	15

Note: The size of data not in COMMON includes the area for non-COMMON variables, arithmetic and address constants, and the constants and work areas created by the compiler.

Figure 5. Maximum Source Program Items

The library subroutines are divided into two groups, mathematical subprograms and service subprograms. A mathematical subprogram corresponds to a subprogram defined with a FUNCTION statement in a FORTRAN source program, and a service subprogram corresponds to a subprogram defined with a SUBROUTINE statement. Therefore, a mathmatical subroutine is called when one of its entry names appears in an arithmetic expression in a source statement, and a service subroutine is called when one of its entry names appears in a CALL statement.

MATHEMATICAL SUBPROGRAMS

The mathematical subprograms contained in the FORTRAN library relieves the FORTRAN programmer of the task of writing routines for calculating frequently used mathematical functions. These subprograms perform calculations for such items as sines, cosines, and logarithms. A mathematical subprogram can be called either explicitly or implicitly. Subprograms are called explicitly by using an entry name of a subprogram in a FORTRAN source program statement. They are called implicitly when exponentiation is used in a statement.

The most frequently needed information about mathematical subroutines is given in this section. Readers who are interested in performance statistics and the algorithms of these subprograms are referred to Appendix B.

Explicitly Called Subprograms

An explicitly called subprogram performs one or more mathematical functions. When a subprogram performs more than one function, each function is called by a unique entry name.

To explicitly call a subprogram, the FORTRAN programmer uses the appropriate entry name in a source language statement. The programmer must also supply one or more arguments, enclosed in parentheses, immediately following the entry name; if two or more arguments are supplied, they must be separated from each other by commas. The arguments must agree in type and number with the definition of the subprogram.

The following source statements illustrate the use of explicitly called subprograms:

```
        RESULT = SIN(ANGLE)
        ANS = STOCK + SQRT(AMNT)
```

In the first statement, the sine of the value of ANGLE is computed (using subprogram IJTSSCN) and that value becomes the value of RESULT.

In the second statement, the square root of the value of AMNT is computed (using subprogram IJTSSQT), that value is added to the value of STOCK, and the total becomes the value of ANS.

Figure 6 contains a list of all explicitly called mathematical subprograms. It shows the subprogram and entry names for each function. Figure 6 contains references to other figures that contain details about each entry name. The figures that contain the details (arranged alphabetically by entry name) are described in Summary of Mathematical Subprograms.

Function	Subprogram Name	Entry Name(s)	Reference to Summary
Arctangent	IJTLTAN	DATAN	Figure 9
	IJTSTAN	ATAN	
Common Logarithms	IJTLLOG	DLOG10	Figure 8
	IJTSLOG	ALOG10	
Cosine	IJTLSCN	DCOS	Figure 9
	IJTSSCN	COS	
Exponential	IJTLEXP	DEXP	Figure 8
	IJTEXPN	EXP	
Hyperbolic Tangent	IJTLTNH	DTANH	Figure 9
	IJTSTNH	TANH	
Maximum Value	IJTSMX0	AMAX0 MAX0	Figure 10
	IJTMAXD	DMAX1	
	IJTSMX1	AMAX1 MAX1	
Minimum Value	IJTSMX0	AMIN0 MIN0	Figure 10
	IJTMAXD	DMIN1	
	IJTSMX1	AMIN1 MIN1	
Modular Arithmetic	IJTMODI	AMOD	Figure 10
	IJTMODR	DMOD MOD	
Natural Logarithms	IJTLLOG	DLOG	Figure 8
	IJTSLOG	ALOG	
Sine	IJTLSCN	DSIN	Figure 9
	IJTSSCN	SIN	
Square Root	IJTLSQT	DSQT	Figure 8
	IJTSSQT	SQRT	
Truncation	IJTSINT	AINT	Figure 10
	IJTIFIX	IDINT INT	

Figure 6. Explicitly Called Mathematical Subprograms

Implicitly Called Subprograms

An implicitly called subprogram raises a number to a power (such as in performing exponentiation). A subprogram is called implicitly when two asterisks indicating exponentiation appear in a FORTRAN source language statement.

The following source statement illustrates the use of an implicitly called subprogram:

```
        ANS = BASE ** EXPON
```

If the values of BASE and EXPON are real, the subprogram IJTARXR is used to raise the value of BASE to the power specified by the value of EXPON; the result becomes the value of ANS. If the values of BASE and EXPON are not both real, a different subprogram will be called.

Figure 7 contains a list of all implicitly called mathematical subprograms. It shows the subprogram and entry name for each function. Figure 7 also contains a reference to Figure 11, which contains details about each entry name. Figure 11, containing entry names arranged alphabetically, is described in Summary of Mathematical Subprograms.

SERVICE SUBPROGRAMS

The service subroutines contained in the FORTRAN library are described in two groups: one group tests machine indicators and the other group performs utility services. Each service subprogram is called by using its entry name in a CALL statement.

Function	Subprogram Name	Entry Name	Reference to Summary
Raise an integer to an integer power	IJTAIXI	IJTAIXI	Figure 11
Raise a real number to an integer power	IJTARXI	IJTARXI	Figure 11
Raise a double precision number to an integer power	IJTADXI	IJTADXI	Figure 11
Raise a real number to a real power	IJTARXR	IJTARXR	Figure 11
Raise a double precision number to a double precision power	IJTADXD	IJTADXD	Figure 11

Figure 7. Implicitly Called Mathematical Subprograms

Machine Indicator Test Subprograms

A machine indicator test subprogram tests the status of a pseudo machine indicator, and may return a value to the program that called it. Each psuedo machine indicator, which occupies one storage location, is regarded as ON or OFF depending upon the contents of the indicator. If the indicator contains a zero, it is considered to be OFF. If it contains anything but zero, it is considered to be ON.

In the following descriptions of the psuedo machine test subprograms, i represents an integer expression and j represents an integer variable.

DIVIDE CHECK SUBPROGRAM (IJTDVCK). The entry name for this subprogram is DVCHK. The divide check subprogram tests for a divide check exception, returns a value of 1 or 2 to indicate the condition that exists, and then turns the divide check indicator off. The following source language statement shows the method of calling this subprogram:

```
        CALL DVCHK (j)
```

The value of j is set to 1 if the divide check indicator was ON, or to 2 if the indicator was OFF.

A description of the divide check exceptions is given in Execution Time Interruptions and Errors.

OVERFLOW INDICATOR SUBPROGRAM (IJTOVRF). The entry name for this subprogram is OVERFL. The overflow indicator subprogram tests for an exponent overflow or underflow exception, returns a value of 1, 2, or 3 to indicate the condition that exists, and then turns the overflow indicator off. The following source language statement shows the method of calling this subprogram:

```
        CALL OVERFL (j)
```

The value of j is set to 1 if a floating point overflow condition exists, to 2 if no overflow or underflow condition exists, or to 3 if a floating point underflow condition exists.

A description of exponent overflow and underflow exceptions is given in Execution Time Interruptions and Errors.

PSEUDO SENSE LIGHT SUBPROGRAM (IJTSLIT). The entry names for this subprogram are SLITE and SLITET. The pseudo sense light subprogram can be used to alter, test, and/or record the status of the pseudo sense lights. The action required determines which of the entry names must be used when calling this subprogram.

If either all pseudo sense lights are to be turned OFF or one pseudo sense light is to be turned ON, the subprogram should be called as follows:

```
        CALL SLITE (i)
```

If the value of i is 0, all four pseudo sense lights will be turned off. If the value of i is 1, 2, 3, or 4, the corresponding pseudo sense light will be turned on. If i assumes any value other than 0, 1, 2, 3, or 4, an error message is produced and execution of the program is discontinued.

If one pseudo sense light is to be tested and its status recorded, the subprogram should be called as follows:

```
        CALL SLITET (i, j)
```

The value of i must be 1, 2, 3, or 4 to specify the pseudo sense light to be tested. The value of j is set to 1 if the specified light was on, or to 2 if it was off. If i assumes any value other than 1, 2, 3, or 4, an error message is produced and execution of the program is discontinued.

Utility Subprograms

A utility subprogram can be used to end the execution of a program or to list the contents of specified storage locations.

END EXECUTION SUBPROGRAM (IJTFXIT). The entry name for this subprogram is EXIT. The end execution subprogram performs the same functions as the STOP statement, that is, execution is terminated and control is returned to the DOS or TOS supervisor. The following source language statement shows the method of calling this subprogram:

```
         CALL EXIT
```

STORAGE DUMP SUBPROGRAM (IJTFDMP). The entry names for this subprogram are DUMP and PDUMP. The storage dump subprogram can be used to list the contents of storage areas specified by limits placed in parentheses following the entry name. The format in which the data is listed can also be specified by the programmer. The entry name used depends on whether execution of the program is to end or continue after the storage dump has occurred.

If execution is not to continue after the storage dump has been made, the subprogram should be called as follows:

CALL DUMP $(a_1, b_1, f_1, \ldots, a_n, b_n, f_n,)$

The values of a and b specify the limits of storage area to be dumped, and f specifies the format in which the contents of the area are to be dumped. If only one word is to be dumped, a and b must be the same. The permitted values of f are:

Value of f	Format of Dumped Data
0	Hexadecimal
4	Integer
5	Real
6	Double Precision

If execution is to continue after the storage dump has been made, the subprogram should be called as follows:

CALL DUMP $(a_1, b_1, f_1, \ldots, a_n, b_n, f_n)$

The values of a, b, and f are the same as described for DUMP.

The following is an example of a statement that calls for a storage dump:

```
CALL DUMP (A, C, 5, D(1,1),D(5,5), 4, E, E, 0)
```

This example assumes that A, C, and E are real variables and that D is an array defined by the following statement:

```
        INTEGER D(5,5)
```

The example would cause the contents of locations from A through C to be dumped in real format, array D to be dumped in integer format, and variable E to be dumped in hexadecimal format. The format under which a variable is to be dumped must be either in hexadecimal or the same as the defined type of the variable. Otherwise a program check may occur.

SUMMARY OF MATHEMATICAL SUBPROGRAMS

Figures 8 through 11 contain a summary of information about the subprograms in the FORTRAN library. Information for each subprogram includes: the entry name, the program name, the mathematical definition, the arguments required, the type of value returned, the assembler requirements, and the error message. The following sections describe the information contained in each column (according to column headings used in Figures 8 through 11).

Entry Name

This column gives the entry name that the programmer must use to call the subprogram for a certain computation. Note that a subprogram may have more than one entry

Entry Name	Sub-program Name	Definition	Argument(s)			Function Value Returned	Assembler Requirements		Error Message
			No.	Type	Range		Registers	Save Area	
ALOG	IJTSLOG	$y = \log_e x$ or $y = \ln x$	1	R	$x>0$	R	0(2)	3D	IJT253I
ALOG10	IJTSLOG	$y = \log_{10} x$	1	R	$x>0$	R	0(2)	3D	IJT253I
DEXP	IJTLEXP	$y = e^x$	1	DP	$x<174.673$	DP	0(2)	5D	IJT262I
DLOG	IJTLLOG	$y = \log_e x$ or $y = \ln_e$	1	DP	$x>0$	DP	0(2)	5D	IJT263I
DLOG10	IJTLLOG	$y = \log_{10} x$	1	DP	$x>0$	DP	0(2)	5D	IJT263I
DSQRT	IJTLSQT	$y = \sqrt{x}$ or $y = x^{1/2}$	1	DP	$x>0$	DP	0(2,4)	3D	IJT261I
EXP	IJTEXPN	$y = e^x$	1	R	$x<174.673$	R	0	6D	IJT252I
SQRT	IJTSSQT	$y = \sqrt{x}$ or $y = x^{1/2}$	1	R	$x<0$	R	0(4)	3D	IJT251I

DP = Double Precision
R = Real

Figure 8. Summary of Logarithmic and Exponential Subprograms

Entry Name	Sub-program Name	Definition	No.	Argument(s)		Function Value Returned	Assembler Requirements		Error Message
				Type	Range		Registers	Save Area	
ATAN	IJTSTAN	$y = \arctan(x)$	1	R	Any real argument	RIR	0(2,4,6)	3D	None
COS	IJTSSCN	$y = \cos(x)$	1	RIR	$\lvert x\rvert < 2^{18}(\pi)$	R	0(2,4)	3D	IJT254I
DATAN	IJTLTAN	$y = \arctan(x)$	1	DP	Any double precision argument	DPIR	0(2,4,6)	3D	None
DCOS	IJTLSCN	$y = \cos(x)$	1	DPIR	$\lvert x\rvert < 2^{50}(\pi)$	DP	0(2,4)	3D	IJT264I
DSIN	IJTLSCN	$y = \sin(x)$	1	DPIR	$\lvert x\rvert < 2^{50}(\pi)$	DP	0(2,4)	3D	IJT264I
DTANH	IJTLTNH	$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	1	R	Any real argument	RIR	0(4)	3D	None
SIN	IJTSSCN	$y = \sin(x)$	1	RIR	$\lvert x\rvert < 2^{18}(\pi)$	R	0(2,4)	3D	IJT254I
TANH	IJTSTNH	$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	1	R	Any real argument	DPIR	0(4)	3D	None

DP	=	Double Precision
DPIR	=	Double Precision, expressed in radians
R	=	Real
RIR	=	Real, expressed in radians

Figure 9. Summary of Trigonometric Subprograms

Entry Name	Sub-program Name	Definition	Argument(s)			Function Value Returned	Assembler Requirements		Error Message
			No.	Type	Range		Registers	Save Area	
AINT	IJTSINT	y = (sign of x)(n) where n is the largest integer less than or equal to $\lvert x \rvert$	1	R	Any real argument	R	0	5D	None
AMAX0	IJTSMX0	$y = \max(x_1, \ldots, x_n)$	≥ 2	I	Any integer arguments	R	0	5D	None
AMAX1	IJTSMX1	$y = \max(x_1, \ldots, x_n)$	≥ 2	DP	Any real arguments	DP	0	5D	None
AMIN0	IJTSMX0	$y = \min(x_1, \ldots, x_n)$	≥ 2	I	Any integer arguments	R	0	5D	None
AMIN1	IJTSMX1	$y = \min(x_1, \ldots, x_n)$	≥ 2	R	Any real arguments	R	0	5D	None
AMOD	IJTMODI	$y = x_1$(module x_2) See Note	2	R	$x \neq 0$	R	0	5D	None
DMAX1	IJTMAXD	$y = \max(x_1, \ldots, x_n)$	≥ 2	DP	Any real arguments	DP	0	5D	None
DMIN1	IJTMAXD	$y = \min(x_1, \ldots, x_n)$	≥ 2	DP	Any real arguments	DP	0	5D	None
DMOD	IJTMODR	$y = x_1$ (module x_2) See Note	2	DP	$x \neq 0$	DP	0	5D	None
IDINT	IJTIFIX	y = (sign of x)(n) where n is the largest integer less than or equal to $\lvert x \rvert$	1	DP	Any double precision arguments	DP	0	5D	None
INT	IJTIFIX	y = (sign of x)(n) where n is the largest integer less than $\lvert x \rvert$	1	R	Any real arguments	I	0	5D	None
MAX0	IJTSMX0	$y = \max(x_1, \ldots, x_n)$	≥ 2	I	Any integer arguments	I	0	5D	None
MAX1	IJTSMX1	$y = \max(x_1, \ldots, x_n)$	≥ 2	R	Any real arguments	I	0	5D	None
MIN0	IJTSMX0	$y = \min(x_1, \ldots, x_n)$	≥ 2	I	Any integer arguments	I	0	5D	None
MIN1	IJTSMX1	$y = \min(x_1, \ldots, x_n)$	≥ 2	R	Any real arguments	I	0	5D	None
MOD	IJTMODR	$y = x_1$ (module x_2) See Note	2	I	$x \neq 0$	I	0	5D	None

DP = Double Precision
I = Integer
R = Real

Note: The expression x_1 (module x_2) is defined as $x_1 - \left[\frac{x_1}{x_2}\right](x_2)$, where the brackets indicate that an integer is used. The magnitude of integer does not exceed the magnitude of $\frac{x_1}{x_2}$ and the sign of the integer is the same as the sign of $\frac{x_1}{x_2}$

Figure 10. Summary of Miscellaneous Mathematical Subprograms

Entry Name	Sub-program Name	One Method of Implicit Function Reference	Argument(s)			Function Value Returned	Assembler Requirements		Error Message
			No.	Type	Range		Registers	Save Area	
IJTADXD	IJTADXD	y = r**p	2	r =DP p =DP		DP	0	9D	IJT245I
IJTADXI	IJTADXI	y = r**n	2	r =DP n=I		DP	0	9D	IJT243I
IJTAIXI	IJTAIXI	y = m**n	2	m=I n =I	R $\neq$0 and P = to any value or R = 0 and P>0	I	0	9D	IJT241I
IJTARXI	IJTARXI	y = r**n	2	r =R n=I		R	0	9D	IJT242I
IJTARXR	IJTARXR	y = r**p	2	r =R p=R		R	0	9D	IJT244I

DP =	Double Precision
I =	Integer
R =	Real

Figure 11. Summary of Implicitly Called Subprograms

name. The computation that is done depends upon the entry name used in the source program. For example, subprogram IJTSSCN has two entry names, SIN and COS. If a sine is to be computed, entry name SIN must be used. If a cosine is to be computed, entry name COS must be used.

Subprogram Name

This column gives the name of the subprogram called by the entry name in the preceding column. Note that there may be more than one subprogram associated with a specific mathematical function. For example, subprogram IJTSSQT or IJTLSQT can be used to compute the square root of a value. The type of the argument will probably determine which subprogram the programmer uses. It is possible to send a double precision argument to a routine that is programmed to receive a single precision argument but it is unsafe to send a real argument to a routine programmed to receive a double precision argument.

Definition

This column, which applies only to explicitly called subprograms, contains the mathematical equation that shows the computation done by the subprogram. For example, the equation for the square root subprogram is $y = \sqrt{x}$. An alternate equation is shown if there is another way of representing the computation. For example, $y = x^{1/2}$ is given as another equation for the square root subprogram.

One Method of Implicit Function Reference

This column appears only in Figure 11, in place of the Definition column (see the preceding Definition section). For each implicitly called subroutine, this column shows a source language statement in which the subprogram is called. Note that this is a sample statement and does not represent the only way by which the subprogram can be called.

Argument(s)

This column is divided as follows:

1. No. This column specifies the number of arguments that the programmer must supply to the subprogram.
2. Type. This column specifies the type of argument (such as: real, integer, or double precision) that the programmer

must supply. When more than one subprogram is available for the same calculation, the type of argument and type of result determines which subprogram must be used. For example, if the square root of a real value is to be computed, subprogram IJTSSQT must be used. If the argument is a double precision value, the subprogram IJTLSQT must be used if a double precision result is desired. If a real result is desired, IJTSSQT may be used.

3. Range. This column specifies the valid range of the argument(s). If an argument is not within this range, an error message (see Appendix C) is issued and the execution of the job is ended. For example, both square root subprograms treat a negative argument as an error. When this column specifies "any argument", it means that any argument of the specified type can be used, provided it is within the acceptable range specified by the FORTRAN language for that particular type. For example, if any integer argument is acceptable, the argument must not be greater than 2147483647 (that is, $2^{31}-1$).

   Note: It is physically impossible to exceed this value.

Function Value Returned

This column gives the type of function value (such as: real, integer, or double precision) that is produced by the execution of the subprogram.

Assembler Requirements

This column gives the information necessary to use the subprogram in an assembler language program and, therefore, is of concern to a FORTRAN programmer who wishes to use the assembly language. This column is divided as follows:

1. Registers. This column specifies the floating-point registers used by the subprogram to contain the function value produced and may be followed by one or more numbers in parentheses. The numbers within parentheses specify registers that are used for intermediate computations within the subprogram.

2. Save Area. This column specifies the minimum size of the save area. The letter D indicates a double word, (for example, 5D means five double words.

Error Message

This column gives the message that is issue when an argument is not within the valid range. The message is written in the data set associated with the system output. An explanation of all error messages is given in Appendix C.

USE OF ASSEMBLER LANGUAGE SUBPROGRAMS

This section provides information needed to prepare and use relocatable subprogram written in assembler language with a FORTRAN job.

CALLED AND CALLING PROGRAMS

Any subprogram that is referred to by another program is considered a called program. If this called subprogram refers to another subprogram then it is both a called and calling subprogram. In Figure 12, for example, if program A calls program B and program B calls program C then:

1. A is considered a calling program by B.

2. B is considered a called program by A.

3. B is considered a calling program by C.

4. C is considered a called program by B.

There are three basic FORTRAN job structures that can be formed using assembler-written subprograms in a FORTRAN job:

1. A FORTRAN program (or subprogram) calling an assembler-written subprogram.

2. An assembler-written subprogram calling a FORTRAN subprogram.

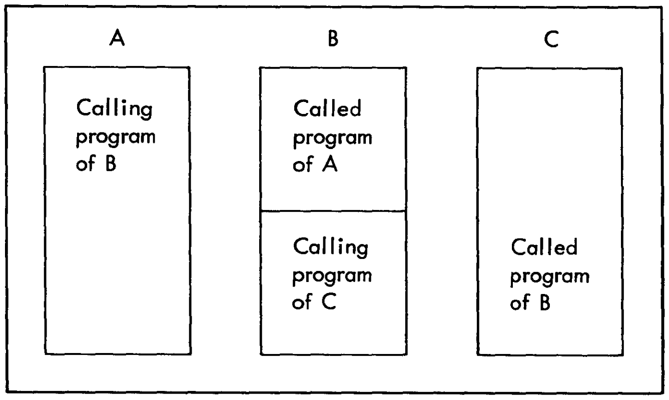


Figure 12. Called and Calling Programs

3. An assembler-written subprogram calling another assembler-written subprogram.

From these combinations, more complicated structures may be formed. For example, a FORTRAN program can call an assember-written subprogram which then could call another assembler-written subprogram.

The Disk and Tape Operating Systems FORTRAN IV has established certain conventions which must be considered when giving and returning control to assembler-written subprograms. These conventions, called linkage conventions, are described.

FORTRAN LINKAGE CONVENTIONS

When a FORTRAN subprogram calls another FORTRAN subprogram, certain save and return routines are generated in addition to a calling sequence that actually transfers control. Figure 13 shows a typical subprogram configuration generated by a calling subprogram.

Assembler-written subprograms need not be constructed with save and return routines exactly as the FORTRAN system generates them; However, there are basic conventions of the FORTRAN system to which the assembler programmer must adhere. These conventions include:

1. Utilizing the proper registers in establishing a linkage.

2. Reserving a parameter area in the calling program in which the called program may refer to the parameter list.

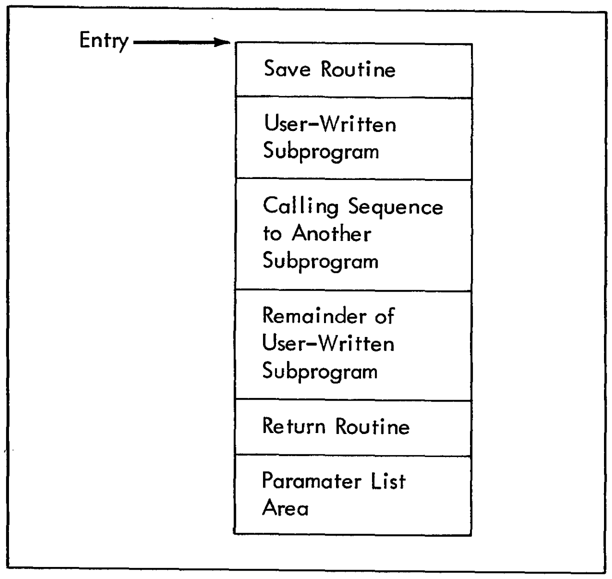

Entry
Save Routine
User-Written Subprogram
Calling Sequence to Another Subprogram
Remainder of User-Written Subprogram
Return Routine
Paramater List Area

Figure 13. Calling Subprograms Configuration

3. Reserving a save area in which the registers used in the linkage may be saved.

Register Use

The Disk and Tape Operating Systems FORTRAN IV has assigned roles to certain registers used in linkages. Figure 14 specifies the fucntion of each linkage register.

Register Number	Register Name	Contents
0	Result Register	Used for function subprograms only. The result is returned in general or floating-point register 0, depending on the type of function (e.g., integer, real, and double-precision). For subroutine subprograms, the result (s) is returned in the variables specified by the programmer.
1	Parameter List Register	Address of the parameter list passed to the called program.
13	Save Area Register	Address of the area reserved by the program being executed (called or calling) in which the contents of certain registers are stored.
14	Return Register	Address of the location in the calling program to which control should be returned after execution of the called program.
15	Entry Point Register	Address of the entry point in the called program.

Figure 14. Linkage Registers

Parameter Area

Every assembled subprogram that calls another subprogram must reserve an area of storage (parameter area) in which the parameter list used by the called subprogram is located. Each entry in the parameter area occupies four bytes at a fullword boundary.

The first byte (bits 0 through 7) of each entry in the parameter area contains zeros. However, bit 0 may contain 1 to indicate the last entry.

The last three bytes of each entry contain the 24-bit address of the argument.

A parameter may be one of the following:

1. The address of a variable.
2. The address of the first element in an array.
3. The address of the word containing the address of a subprogram.

Save Area

An assembled subprogram that calls another subprogram must reserve an area of storage (save area) in which certain registers (i.e., those used in the subprogram and those used in the linkage to the subprogram) are saved.

The maximum amount of storage reserved by the subprogram is 9 double words. Figure 15 shows the layout of the save area and the contents of each word.

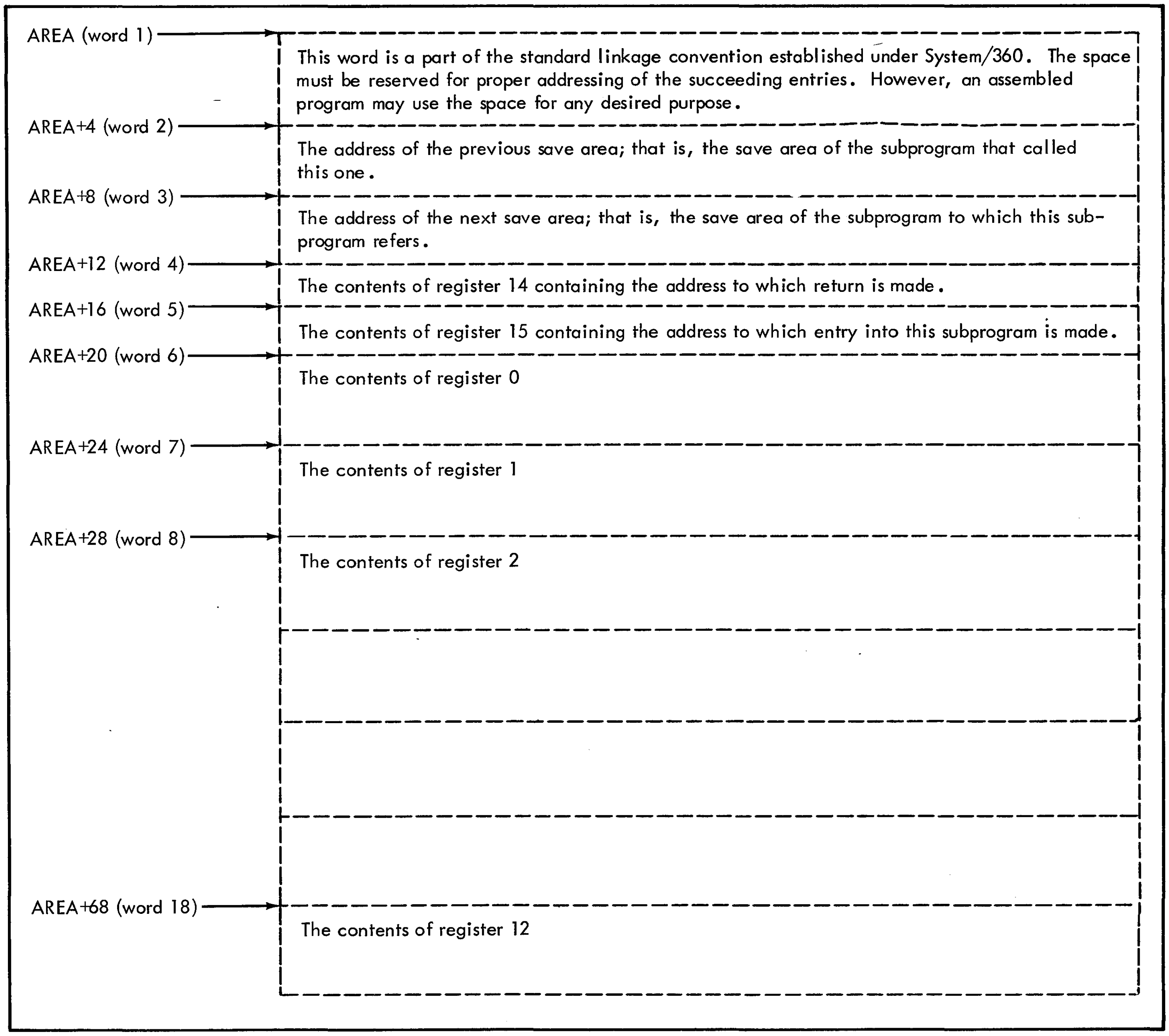
An assembled subprogram which does not call another subprogram need not establish a save area. However, if registers 13 or 14 are used by the subprogram, that subprogram should save their contents in a desired location and restore them before returning control to the calling program.

Sample Calling Subprogram Linkage

Figure 16 shows the linkage conventions used by an assembled subprogram that calls another subprogram.

The coding does not have to conform exactly to that shown in Figure 16. However, the linkage should include:

1. The calling sequence by which an assembled subprogram may reference another subprogram.
2. The save and return routines by which the appropriate save area is established and control is returned to the calling program.
3. The out-of-line parameter area by which an assembled subprogram may pass parameters. (An in-line parameter area may be used instead; see the section, In-line Parameter Area.)



AREA (word 1) → This word is a part of the standard linkage convention established under System/360. The space must be reserved for proper addressing of the succeeding entries. However, an assembled program may use the space for any desired purpose.

AREA+4 (word 2) → The address of the previous save area; that is, the save area of the subprogram that called this one.

AREA+8 (word 3) → The address of the next save area; that is, the save area of the subprogram to which this subprogram refers.

AREA+12 (word 4) → The contents of register 14 containing the address to which return is made.

AREA+16 (word 5) → The contents of register 15 containing the address to which entry into this subprogram is made.

AREA+20 (word 6) → The contents of register 0

AREA+24 (word 7) → The contents of register 1

AREA+28 (word 8) → The contents of register 2

AREA+68 (word 18) → The contents of register 12

Figure 15. Save Area Layout and Contents

```
deckname  START   0
          EXTRN   name

          USING   *,2
*   Save Routine
          STM     14,r1,12(13)      The contents of registers 14, 15 and 0 through r1
*                                   are stored in the save area of the calling program
*                                   (previous save area).  r1 is any number from 0
                                    through 12.
          LR      2,15              Loads the entry point into the base register.
          LR      r2,13             Loads register 13, which points to the save area
*                                   of the calling program, into any general register,
*                                   r2, except 0 and 13.
          LA      13,AREA           Loads the address of this program's save area into
*                                   register 13.
          ST      13,8(0,r2)        Stores the address of this program's save area into
*                                   word 3 of the save area of the calling program.
          ST      r2,4(0,13)        Stores the address of the previous save area (i.e.,
*                                   the same area of the calling program) into word 2
*                                   of this program's save area.
          BC      15,prob1
AREA      DS      9D                Reserves 9 double words for the save area.  This is
*                                   the last statement of the save routine.
prob1     User-written program statements
                  .
                  .
                  .
*  Calling Sequence
          LA      1,ARGLST          First statement in calling sequence.
          L       15,ADCON
          BALR    14,15
                  .
                  .
                  .
*         Remainder of user-written program statements
                  .
                  .
                  .
*  Return Routine
          L       13,AREA+4         First statement in return routine.  Loads the address
*                                   of the previous save area back into register 13.
          LM      2,r1,28(13)       The contents of registers 2 through r1 are restored
*                                   from the previous save area.
          L       14,12(13)         Loads the return address, which is in word 4 of the
*                                   calling program, into register 14.
          MVI     12(13),X'FF'      Sets flag FF in the save area of the calling program
*                                   to indicate that control has returned to the calling
*                                   program.
          BCR     15,14             Last statement in return routine.
ADCON     DC      A(name)           Contains the address of subprogram name.

*  Parameter Area
ARGLST    DC      AL4(arg1)         First statement in parameter area setup.
          DC      A14(arg2)
                  .
                  .
                  .
          DC      X'80'             First byte of last argument.
          DC      AL3(argn)         Last statement in parameter area setup.
```

Figure 16. Sample Linkage Routines Used with a Calling Subprogram

LOWEST LEVEL SUBPROGRAMS

If an assembled subprogram does not call any other program (that is, if it is at the lowest level), the programmer should omit the save routine, calling sequence, and parameter area shown in Figure 16. Figure 17 shows the appropriate linkage conventions used by an assembled subprogram at the lowest level.

IN-LINE PARAMETER AREA

The assembler programmer may establish an in-line parameter area instead of an out-of-line area. In this case, he may substitute the calling sequence and parameter area shown in Figure 18 for that shown in Figure 16.

Referencing COMMON

If an assembler-written subprogram is to share data in COMMON with a FORTRAN program, the assembler program must:

1. Define a blank common control section.

2. Load a general register with an address constant containing the address of the blank common control section.

   This is done as shown in Figure 19.

```
deckname        START        0
                USING        *,15
                STM          14,r,12(13)
                  •
                  •
                  •
      User - written program statements
                  •
                  •
                  •
                LM           2,r1,28(13)
                MVI          12(13),X'FF'
                BCR          15,14
```

Note: If registers 13 and/or 14 are used in the called subprogram, their contents should be saved and restored by the called subprogram.

Figure 17. Sample Linkage for Lowest Level Subprograms

```
                   .
                   .
                   .
                   LA          14,RETURN
                   L           15,ADCON
                   CNOP        2,4
                   BALR        1,15
                   DC          AL4(arg1)
                   DC          AL4(arg2)
                   .
                   .
                   .
                   DC          X'80'
                   DC          AL3(argn)
         RETURN    NOP         0
                   .
                   .
                   .
         ADCON     DC          A(prob1)
```

Figure 18. Sample of In-Line Parameter Area

```
deckname          START        0
                    •
                    •
                    •
                    L          4,COMADCON
                    •
                    •
                    •
                    •
                    •
COMADCON            DC         A(STARTCOM)
                    •
                    •
                    •
                    •
                    COM
STARTCOM            DS         D
```

Figure 19. Referencing COMMON

PROCESSING FORTRAN PROGRAMS

The three types of processing provided by the Disk or Tape Operating System are compilation, linkage editing, and execution. Each type of processing may be done separately in a job of its own or they may be done together in one job. Examples of combined processing are compiling and linkage editing, linkage editing and executing, or compiling, linkage editing, and executing. Each type of processing is described separately in this section and includes examples of how it can be combined with other types of processing.

COMPILATION OF A FORTRAN PROGRAM

The process of converting FORTRAN language statements, which make up a source module, into an object module is called compilation. This translation of FORTRAN statements into machine-language instructions, plus the preparation of information required by the Linkage Editor, is done by the FORTRAN compiler. A brief description of the cards contained in an object module is given in Linkage Editing a FORTRAN Program.

SOURCE PROGRAMS

The statements that may be used in source programs to be processed by the FORTRAN compiler are described in the language publication FORTRAN IV Specifications mentioned as a prerequisite in the Preface. Additional information regarding source programs appears in the section Building FORTRAN Programs. Program decks using statements as described in these publications are the source modules for the FORTRAN compiler.

An assortment of suggestions for improving program efficiency is given in the section Programming Suggestions. These suggestions do not constitute additions or changes in the FORTRAN language; they only point out refinements in the usage of source language statements.

CONTROL CARDS FOR COMPILATION

Compilation is specified by a control card that causes the FORTRAN compiler to be executed. Other control cards may be used to specify what is to be done with the results of the compilation. The examples given in the sections Compilation with Punched Output and Compilation for Linkage Editing illustrate combinations of control cards for various compilations.

Note that the formats of the control cards shown in the examples are given in detail in Control Cards.

Compilation with Punched Output

If the object module is to be punched into cards, the input to the compilation depends on the type of card punch used for output.

EXAMPLE 1: The following cards, which cause the object module to be punched, can be used when a separate unit is used for card punching. That is, it cannot be a card read punch that is also used for input to the compiler (see Example 2 for combined reading and punching):

```
// JOB PRIME

// OPTION  DECK,NOLINK

// EXEC FORTRAN

   source program

/&
```

EXAMPLE 2: The following cards can be used when output is to be punched using the same card read punch (for example, IBM 1442) from which the input is read:

```
// JOB PRIME

// OPTION  DECK,NOLINK

// EXEC FORTRAN

   source program

   blank cards (for object module)

/*

/&
```

Note that the blank cards into which the object module will be punched must be placed between the END statement of the source program (its last card) and the /* or /& card.

Compilation for Linkage Editing

An object module(s) can be linkage edited as part of the job in which the object module is produced. When this is done, no object module is punched into cards.

EXAMPLE 1: The following control cards cause one object module to be linkage edited immediately after compilation:

```
        // JOB PRIME

        // OPTION LINK

        // EXEC FORTRAN

           source program

        /*

        // EXEC LNKEDT

        /&
```

EXAMPLE 2: The following control cards cause two source modules to be compiled and then linkage edited together as part of the same job:

```
        // JOB TWOPROGS

        // OPTION LINK

        // EXEC FORTRAN

           source program A

        /*

        // EXEC FORTRAN

           source program B

        /*

        // EXEC LNKEDT

        /&
```

DIAGNOSTIC AIDS

The Disk and Tape Operating Systems FORTRAN IV compiler produces diagnostic aids that can be used by the programmer for locating errors in his program. These aids consist of messages and a storage map.

To give the programmer as much information about program errors as possible during each compilation, the compiler has been designed to continue compilation as long as possible in spite of source program errors. If an error is encountered during the processing of a source statement, the compiler flags the error and attempts to skip the portion of the statement that is in error and compile the remainder of that statement. This technique is particularly applicable to specification (nonexecutable) statements such as a DIMEMSION statement. Similarly, the compiler continues processing a program even when entire source statements must be ignored because of errors in them. Although any syntactic error in the source program will ultimately result in the termination of the compilation, the compiler continues until all source program statements have been inspected and an attempt has been made to allocate storage for all variables in the program. Therefore, the programmer receives complete diagnostic information for each compilation of his source program.

Whenever a source program error(s) causes compilation to be terminated, a message is printed on the listing before any object program cards are punched. (This message is printed even if no listing was requested with the compilation.)

Diagnostic Messages

Diagnostic messages are printed on the source program listing, which is produced by the compiler when requested in an OPTION card. There are two kinds of diagnostic messages: summary error messages and statement error messages. All of the messages are explained in Appendix C. The following sections describe the appearance of the messages in the source program listing.

SUMMARY ERROR MESSAGES: These are printed at the end of the source program listing; that is, following the listing of the source program cards and the statement error messages (if any). Summary error messages indicate errors that the compiler cannot associate with a specific source program statement. For example, undefined labels are listed following an appropriate summary error message (see Figure 20).

STATEMENT ERROR MESSAGES: These are printed following the source program statement that contains the error to which the message refers.

When a source program listing is requested, one FORTRAN source card is printed on each line. If the source card contains an error(s), the line below it is used to indicate the position of the error(s) within the statement. A dollar sign ($) is printed under each position when an error is detected by the compiler. Depending upon the type of the error, the dollar sign will appear under one of the following:

1. The character or location causing the error.

2. The location at which a character is missing.

3. The character preceding the one causing the error.

For examples of the placement of the dollar sign, refer to Sample Source Program Listing.

One or more lines that follow the line containing the dollar sign error markers are used for the printing of statement error messages. Up to six error messages are printed on each line. Each error message is preceded by a number separated from the message by a right parenthesis. The number specifies which dollar sign (counting from left to right) identifies the error stated in the error message.

Note: If a dollar sign appears following a comment card(s), the error refers to the preceding noncomment statement.

Sample Source Program with Storage Map

Figure 20 shows a sample of a source program listing with a corresponding storage map. This sample program has forced errors

```
          // JOB ERRSAMP
          // OPTION LIST,LINK
          // EXEC FORTRAN
Card
Number                                DISK OR TAPE OPERATING SYSTEM FORTRAN   360B-F0-409 19
  1                C      PRIME NUMBER PROBLEM
  2                       REAL Z(5) X(4)
                                   $
            01) COMMA
  3                   100 WRITE (3,8)
  4                     8 FORMAT (52H FOLLOWING IS A LIST OF PRIME NUMBERS FROM 1 TO 1000/
  5                      /19X,1H1/19X,1H2/19X 1H3)
  6                   1B1 I=5
                        $
            01) SYNTAX
  7                     3 A=I
  8                   102 A=SQRT(A)
  9                   103 J=A
 10                   104 DO 1 K=3,J,2
 11                   105 L=I12345 K12345
                                  $
            01) NAME LENGTH
 12                   106 IF(L*K-I)1,2,4
 13                       CONTINUE
                  $
            01) LABEL
 14                   107 WRITE (3,5)I
 15                       FORMAT (I20)
                                $
            01) LABEL
 16                     2 I=I+2
 17                   108 IF(1000-I)7,4
                                    $
            01) SYNTAX
 18                       WRITE (3,9)
 19                     9 FORMAT (14H PROGRAM ERROR)
 20                     7 WRITE (3,6)
 21                     6 FORMAT (31H THIS IS THE END OF THE PROGRAM
                                                          $
            01) SYNTAX
 22                   100 STOP
                        $
            01) DUP. LABEL
 23                       END
```

Ⓐ

```
          03/10/66            FORTMAIN                                              0002

                                             UNCLOSED DO LOOP TARGETS

  00001

                                             UNDEFINED LABELS

  00001       00004       00005

                                             SCALARS

  SYMBOL   LOCATION     SYMBOL   LOCATION     SYMBOL   LOCATION     SYMBOL   LOCATION     SYMBOL   LOCATION
  A          005C       I          0060       J          0064       K          0068       L          006C
  I12345     0070

                                              ARRAYS

  SYMBOL   LOCATION     SYMBOL   LOCATION     SYMBOL   LOCATION     SYMBOL   LOCATION     SYMBOL   LOCATION
  X          0074       Z          0084
                                                  COMPILATION TERMINATED
```

Ⓑ

Figure 20. Sample Source Program with Storage Map

to illustrate the appearance of diagnostic messages. The program used to create this sample listing is the same as the one shown in Appendix A, with errors inserted to produce some of the error messages listed in Appendix C.

The following list indicates the types of errors shown in Ⓐ of Figure 20.

Card Number	Error
2	There is a comma missing
6	Statement number is not numerical
11	An operator missing, causing two variables to be treated as a single name giving the NAMELENGTH message
13	Statement number is missing
15	FORMAT statement must be numbered
17	IF statements must have three branch points
21	Right parenthesis missing
22	Duplicate statement number

Ⓑ of Figure 20 shows a storage map for the above sample program. In the sample program, there is no statement numbered 1. Since statement number 1 is referred to in a DO statement, the first message under UNCLOSED DO LOOP TARGETS is produced. Statement number 1 is also referred to in an IF statement that caused the message under UNDEFINED LABLES to print out. Statements 4 and 5 are also undefined.

Diagnostic Storage Map

The compiler also prints a storage map as a disgnostic aid. The storage map indicates the storage locations assigned to each variable, the subroutines called within the program, and the locations assigned to each numbered source program statement. Figure 21 is a sample storage map of the program shown in Appendix A.

The numbers printed in the storage map are in decimal and hexadecimal form. The numbers showing locations are in hexadecimal form. The two numbers showing the amount of common and the amount of core are in decimal form. The symbol locations shown on the storage map are relative to the ADDRESS BASE TABLE number. The statement locations are relative to 0. To find the absolute location of a symbol, the base address of the program (assigned by linkage editor) is added to the relative address.

LINKAGE EDITING A FORTRAN PROGRAM

The process of preparing one or more object modules for execution is called linkage editing. To prepare a program phase, which is the output of a linkage editing operation, the Linkage Editor uses information contained in the control dictionaries that are a part of every object module.

```
          02/15/66             FORTMAIN                                                  0002

                                                       SCALARS

SYMBOL   LOCATION        SYMBOL    LOCATION       SYMBOL   LOCATION      SYMBOL    LOCATION      SYMBOL    LOCATION
I          006C          A           0070         J          0074        K           0078        L           007C
                                                       CALLED SUBROUTINES

IJTAPST     IJTACOM      IJTSSQT     SQRT

LABEL    LOCATION        LABEL     LOCATION       LABEL    LOCATION      LABEL     LOCATION      LABEL     LOCATION
00100      0078          00008       0088         00101      00D8        00003       00E0        00102       0100
00103      010E          00104       012C         00105      0134        00106       0144        00001       015E
00107      0172          00005       0190         00002      019A        00108       01A6        00004       018C
00009      01D0          00007       01E8         00006      01FC        00109       0226
         COMPILATION COMPLETE    AMOUNT OF COMMON 000000      AMOUNT OF CORE 000692     ADDRESS BASE TABLE     0200
```

Figure 21. Storage Map

Each object module consists of four groups of cards that are identified by a special character in column 1 and an identifier in columns 2 through 4. The contents of each group of cards and the order in which the groups must be read is as follows:

Identifier	Type of Card(s)
ESD	External Symbol Dictionary Cards
TXT	Text Cards (machine instructions)
RLD	Relocation Dictionary Card
END	End Card

The control dictionaries contain information that the Linkage Editor requires for producing a program phase. For a more detailed description of the cards produced by the Linkage Editor, refer to the publication System Control and System Service Programs mentioned in the Preface.

CONTROL CARDS FOR LINKAGE EDITING

An EXEC card specifying the execution of LNKEDT is the only card required to produce a program phase. However, other control cards used in the same job can determine what is linkage edited. The examples illustrate combinations of control cards for various linkage editing operations.

Details regarding the formats of the various control cards are given in Control Cards.

Linkage Editing One Object Module

The following cards cause the linkage editing of one object module into one program phase:

```
// JOB PRIME

// OPTION LINK

bINCLUDE

   object module

/*

// EXEC LNKEDT

/&
```

Linkage Editing Two Object Modules

The following cards cause two object modules to be linkage edited into one program phase:

```
// JOB AANDB

// OPTION LINK

bINCLUDE

   object module for A

   object module for B

/*

// EXEC LNKEDT

/&
```

Linkage Editing Multiple Object Modules

A job can specify that more than one object module be linkage edited to form one program phase.

EXAMPLE 1: The following cards cause too source modules to be compiled and linkage edited into one program phase:

```
// JOB AANDB

// OPTION LINK

// EXEC FORTRAN

   source program A

/*

// EXEC FORTRAN

   source program B

/*

// EXEC LNKEDT

/&
```

FORTRAN PROGRAM EXECUTING CONSIDERATIONS

Execution is the process of obtaining a program phase, loading it into main storage, and executing the machine language instructions contained in that program phase. The program phase to be executed is specified in a job control card. Execution of the program phase occurs under control of the Supervisor and Job Control.

JOB CONTROL CARDS FOR EXECUTION

The number of control cards required to execute a program depends upon:

1. The number of program phases in the program.

2. Whether execution is to be done as a separate job or as part of a job.

3. Whether or not data for the program is to be read from the same device as the job control cards.

Examples of combinations of control cards for program execution under various conditions are given. Details regarding the formats of job control cards appear in Control Cards.

Execution as a Separate Job

These examples assume that the FORTRAN source program has been previously compiled and linkage edited and placed in the core-image library.

EXAMPLE 1: The following cards cause the execution of a program name PRIME from SYSRES:

```
// JOB PNUMBERS

// EXEC PRIME

/&
```

EXAMPLE 2: The following cards cause the sequential execution of three programs within a single job:

```
// JOB REASONAN

// EXEC CREACT

// EXEC LREACT

// EXEC SUMSQS

/&
```

EXAMPLE 3: The following cards cause the execution of a job consisting of three programs using data contained among the job control cards for two of the programs:

```
// JOB RESNAN

// EXEC CAPAC

   data for CAPAC

/*

// EXEC INDUCT

   data for INDUCT

/*

// EXEC PATHAG

/&
```

The programs will be executed in the order CAPAC, INDUCT, and PATHAG.

Execution as Part of a Job

These examples assume that at least one of the program phases to be executed is compiled and linkage edited as part of the job.

EXAMPLE 1: The following cards cause the compilation, linkage editing, and execution of a source program:

```
// JOB PRIME

// OPTION LINK

// EXEC FORTRAN

   source program

/*

// EXEC LNKEDT

// EXEC

/&
```

EXAMPLE 2: The following cards cause the compilation, linkage editing, and execution of one program and the execution of another program (both use data contained among the job control cards):

```
        // JOB XEQPROGA

        // OPTION LINK

        // EXEC FORTRAN

           source program for PROGA

        /*

        // EXEC LNKEDT

        // EXEC

           data for PROGA

        /*

        // EXEC PROGB

           data for PROGB

        /&
```

EXAMPLE 3: The following cards cause action similar to that for Example 2, except that the program compiled as part of the job is to be executed second:

```
        // JOB XEQPROGB

        // OPTION LINK

        // EXEC FORTRAN

           source program for PROGB

        /*

        // EXEC LNKEDT

        // EXEC PROGA

           data for PROGA

        /*

        // EXEC

           data for PROGB

        /&
```

EXECUTION TIME INTERRUPTIONS AND ERRORS

The execution of an object program may be discontinued because an interruption caused by the computer has occurred, or an error in the argument of a subprogram has been detected.

Interruptions

An interruption occurs when the computer detects that either an invalid arithmetic operation has been attempted, or the result of an arithmetic operation cannot be fully contained in a floating-point register. These interruptions are due specifically to one of the following conditions:

1. Exponent overflow exception.

2. Exponent underflow exception.

3. Divide exceptions.

An exponent overflow exception is recognized when the result of a floating point addition, subtraction, multiplication, or division is equal to or greater than 16^{63} (approximately 7.2×10^{75}).

An exponent underflow exception is recognized when the result of a floating point addition, subtraction, multiplication, or division is less than than 16^{-63} (approximately 5.4×10^{-79}).

A division exception is recognized when an attempt to divide by zero is made.

When any of these interruptions occur, an indicator is set and a message is written in the data set used for system output (see message number 225). After the interruption has been handled, the execution of the object program continues from the point at which the interruption occurred.

Errors

The execution of the object program is terminated and control returned to the DOS or TOS supervisor whenever an argument cannot be handled by a mathematical subprogram. These subprograms do not check for errors in the argument (such as: wrong type, invalid characters, wrong length, etc.) and, therefore, a computation done with an erroneous argument produces an unpredicatable result. However, some mathematical functions require that the argument be within a specific range (see Figures 8 through 11). If an argument is not within the valid range, a message (see Appendix C) is written in the data set used for system output, and control is returned to the DOS or TOS supervisor. For example, an attempt to take the square root of a negative number is regarded as an error.

FORTRAN UNIT ASSIGNMENT

In a FORTRAN source program, input and output devices are referred to by a logical unit number. By using this form of reference, an object program compiled from a FORTRAN source program is not dependent upon the availability of any specific input/output device. Therefore, an available device can be assigned for use by the object program at execution time.

The assignment of available input/output devices is done by the computer operator at execution time. To make the assignment, the operator must know which symbolic units of the Disk or Tape Operating System are to be used. There is a fixed correspondence between these symbolic units and the logic unit numbers used in a FORTRAN source program. Figure 22 shows the DOS or TOS symbolic unit name for each FORTRAN logical unit. The FORTRAN programmer references input/output devices by logical unit numbers. However, the computer operator must assign the input/output devices to the DOS or TOS by their equivalent symbolic unit names.

For example, if a FORTRAN program reads input data from logical unit 5, the programmer might instruct the computer operator to "mount the input data tape on SYS002". To obey this instruction, the operator would mount a tape on an available magnetic tape unit and indicate to the DOS or TOS that the unit he selected is to be used as SYS002. The operator would then mount the tape containing the input data on the unit selected to by SYS002. During the execution of the FORTRAN object program, all references to logical unit 5 will refer to the tape unit selected by the computer operator. In the listing that shows the control cards used for a job, the units assigned by the operator using control cards will appear in // ASSGN cards located between the JOB card and the EXEC card.

There are several FORTRAN logical unit numbers that can be assigned only to specific types of units. These units (1, 2, 3,

FORTRAN Unit Number	DOS or TOS Symbolic Unit Name	Disk File NAME	Type of Device Permitted	Type of Operation Permitted	BACKSPACE, END FILE, and REWIND Permitted
1	SYSIPT	None	Card Reader or Tape Unit	Input only	No
2	SYSPCH	None	Card Punch or Tape Unit	Output only	No
3	SYSLST	None	Printer or Tape Unit	Output only	No
4	SYS001	IJSYS01	Tape Units or Punch Card Devices or Disk Storage Units	Input and Output	Yes for Tape Units; No for Punch Card Devices; Yes for Sequential Disk files; No for Direct Access Files
5	SYS002	IJSYS02			
6	SYS003	IJSYS03			
7	SYS004	IJSYS04			
8	SYS005	IJSYS05			
9	SYS006	IJSYS06			
10	SYS007	IJSYS07			
11	SYS008	IJSYS08			
12	SYS009	IJSYS09			
13	SYS010	IJSYS10			
14	SYS011	IJSYS11			
15	SYSLOG	None	Console Typewriter or Printer	Output only	No

Figure 22. Devices Assigned to FORTRAN Units

and 15) and their specified uses are shown in Figure 22. For example, an attempt to read input data from logical unit 3 will result in program termination.

The control operations BACKSPACE, END FILE, and REWIND apply to tape files and disk files; they have no meaning for files on other devices. However, as shown in Figure 22 these control operations are not permitted for FORTRAN logical units 1, 2, and 3. For example, if a FORTRAN program uses logical unit 3 for tape output, an attempt to backspace the tape will result in termination of the job.

Execution of Control Commands

Under certain conditions REWIND, BACKSPACE, and END FILE statements are not executed. Whether or not they are executed depends upon the execution of a READ or WRITE statement preceding them. Before a REWIND or BACKSPACE statement can be executed, a READ, WRITE, or END FILE statement for that file must have been executed. One of these statements must be executed first so that the Disk or Tape Operating System can determine the type of device being used for the file and thereby determine what statements are legal for that device.

If a READ, WRITE, or END FILE statement has not been executed for a file when a REWIND or a BACKSPACE statement for that file is encountered, the statement will be ignored; that is, no attempt to rewind or backspace the file will be made and no message will be produced if the operation is not valid for the logical unit.

Execution of END FILE

An END FILE statement will be ignored if it is preceded by the execution of a READ statement for that file. For tape files, if the END FILE statement follows a WRITE statement or is the first statement referring to the file, a tapemark will be written. For sequential disk files, the END FILE statement causes an end-of-file record to be written. An END FILE statement referring to a direct access file is not legal.

Execution of REWIND

Unless a REWIND statement is preceded by the execution of a READ, WRITE, or END FILE statement, it is ignored. If the statement preceding the REWIND statement was a READ or END FILE statement, the rewinding operation is done immediately. If a WRITE statement preceded the REWIND statement, a tapemark (for tape files) or an end-of-file record (for sequential disk files) is written before the rewinding operation occurs.

After the execution of a REWIND statement, the file is positioned at the beginning of the first data record in the file; that is, the next record read from the file will be the first data record in the file or the next record written will become the first data record in the file.

If a REWIND statement is illegal for the device (see Execution of Initial READ and WRITE and Figure 22), an error message is issued and the program is terminated.

Execution of BACKSPACE

Unless a BACKSPACE statement is preceded by the execution of a READ, WRITE, or END FILE statement, it is ignored. If the statement preceding the BACKSPACE statement was a WRITE statement, a tapemark (for the tape files) or an end-of-file record (for sequential disk files) is written and a backspace operation occurs. (Note that everything is positioned just as it was before these operations occurred. These steps are taken to ensure that the end of a sequential file will always be marked.) After these steps have been taken or if the statement preceding the BACKSPACE statement was a READ statement, the Disk or Tape Operating System checks the record count of the file. If the record count is zero (indicating that the next record to be read from or written into the file will be the first data record of the file), a warning message is issued. If the record count is not zero, the file is backspaced one logical record.

After the execution of any BACKSPACE statement, the file is positioned at the beginning of a logical record. Backspacing is done according to logical records to agree with READ and WRITE operation, which read or write logical records.

If a BACKSPACE statement is illegal for the device (see Execution of Initial READ and WRITE and Figure 22), an error message is issued and the program is terminated.

LENGTH OF LOGICAL RECORDS: The relation between the length of logical records and physical records depends upon whether or not reading and writing are done according to a FORMAT statement. If a FORMAT statement is used, logical records and physical records are the same length.

The following chart shows the maximum length of a physical record read or written by FORTRAN programs for various I/O devices. They are:

Device	Number of Bytes
Card Reader	80
Card Punch	80
Printer	Number of print positions
Magnetic Tape	255
Disk	
Sequential Access	255
Direct Access	As specified in the DA statement and must be less than 1726.

When a FORMAT statement is not used, a logical record may contain more than 255 bytes and, therefore, may comprise more than one physical record. Because a READ or WRITE statement processes logical records and can produce more than one physical record under these conditions, the execution of the BACKSPACE statement has been designed to backspace over a logical record.

PROGRAMMING FOR A CARD READ PUNCH

If the programmer intends to use the IBM 1442 Card Read Punch for both input and output, the programming technique given should be studied. If in any one source program a card read punch is to be used only for input or only for output, no special programming techniques need be followed. That is, programming is the same as for a separate card reader or card punch.

Special programming techniques are required when using a card read punch because the cards from which data is read and into which data is punched are fed from the same hopper. Figure 23 shows that the cards travel from the input hopper through the read station and then through the punch station to an output stacker.

Because a card passes through the punch station after it passes the read station, it is possible to punch output data into the same card from which the input data was read. The programming technique for doing this is described in Punching Output into Input Cards. However, the programmer may not want the output data punched into the cards that contain the input data (or perhaps there is too much output data to fit into the unused columns of the input data cards). To punch data into blank cards, refer to Punching Output into Blank Cards.

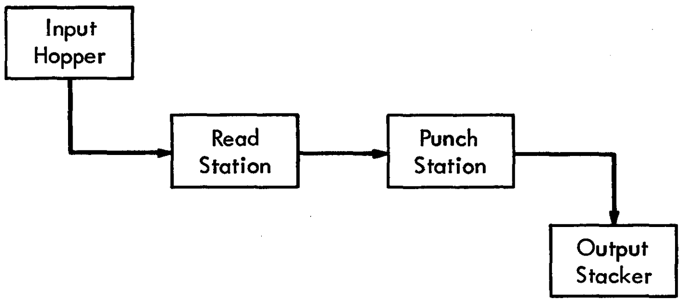

$$\begin{tikzcd}\boxed{\text{Input Hopper}} \arrow[r] & \boxed{\text{Read Station}} \arrow[r] & \boxed{\text{Punch Station}} \arrow[d] \\ & & \boxed{\text{Output Stacker}}\end{tikzcd}$$

Figure 23. Card Flow in a Card Read Punch

Note: The IBM 2520 Card Read Punch must be either an input or an output device.

Punching Output Into Input Cards

Input data is obtained by using a READ statement, which causes one card from the input hopper to pass through the read station. After being read, the card stops before the punch station. If a WRITE statement is the next statement that refers to the card read punch, the output data specified by the WRITE statement will be punched into the input card.

Assuming that logical unit 5 is a card read punch, the following sequence of statements will cause output data to be punched into cards from which input data was read:

```
        .
        .
        .
10  FORMAT  (I10,F10.3,I5,F15.6)

11  FORMAT  (T41,4I10)
        .
        .
        .
    READ (5,10) I,A,J,B,
        .
        . no statements that refer to
        . unit 5
    WRITE  (5,11) K,L,M,N
        .
        .
        .
```

These above statements cause the data in the first 40 columns of a card to be read as variables I, A, J, and B. Then the data from variables K, L, M, and N is punched into the last 40 columns of the same card.

Punching Output Into Blank Cards

If the output data is to be punched into separate cards, rather than into the input cards, blank cards must be inserted with the input cards. That is, each input card must be followed by a blank card. After input data has been read from an input card, that card must be moved past the punch station without punching anything in it. Moving the input card past the punch station will also place the blank card, which follows it, in position to receive the output data. The movement of the input card past the punch station is done using a dummy operation, which causes cards to move but does not read or write data. The dummy operation can be specified in the FORTRAN source program in either of the following ways:

1. By placing a READ statement with no input list between the READ statement that reads the input data and the WRITE statement.

2. By placing a / character (slash) at the end of the list of format codes in the FORMAT statement that controls the reading of the input data.

The following sequence of instructions uses an extra READ statement to move an input data card past the punch station:

```
         .
         .
         .
10  FORMAT  (I20,F20.3,I10,F11.7,E19.6)

11  FORMAT  (4I20)
         .
         .
         .
    READ (5,10) I,A,J,B,C

    READ (5,10)
         .
         .  no statements that
         .  refer to unit 5
    WRITE (5,11) K,L,M,N
         .
         .
         .
```

Assuming that logical unit 5 is a card read punch, the statements in this section cause the data in one 80-column card to be read as variables I, A, J, B, and C. Then the data from variables K, L, M and N is punched in the blank card that follows the input card.

The following sequence of instructions uses a / character in the FORMAT statement to move an input data card past the punch station:

```
         .
         .
         .
10  FORMAT  (I20,F20.3,I10,F11.7,E19.6,
11  FORMAT  (4I20)
         .
         .
         .
    READ (5,10) I,A,J,B,C
         .
         .  no statements that
         .  refer to unit 5
    WRITE (5,11) K,L,M,N
         .
         .
         .
```

<u>Read Only and Write Only</u>

There are no special programming requirements for using a card read punch for reading only or writing only. For example, two READ statements in sequence will read data from two cards in sequence. Similarly, two WRITE statements in sequence will punch data into two cards in sequence.

LABEL PROCESSING

In object programs produced by the FORTRAN compiler, both volume and file labels for tapes are processed by the Disk or Tape Operating System, and, therefore, are not made available to the FORTRAN programmer. READ and WRITE statements in a source program always refer to data records. For most programs, therefore the programmer need not be concerned about labels.

If it becomes necessary to do additional label processing, access to the label records must be made by using some other source language (for example, assembler language), not FORTRAN. Information regarding labels and label processing is described in the publications <u>Data Management Concepts</u> and <u>Supervisor and Input/Output Macros</u> mentioned in the <u>Preface</u>.

For disk files the user must provide the Disk Operating System with label information for the files he is using. The user must supply the following statements:

VOL Statement

The volume statement is used when checking or writing standard labels for a DASD or tape file. A VOL statement must be used for each file on a multifile volume. Its format is:

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

SYSxxx	Symbolic unit name.
filename	The filename that appears in the VOL statement is IJSYSdd where dd is the ten and unit positions of the Disk Operating System symbolic unit.

DLAB Statement

The DASD-label statement (completed in a continuation statement) contains file label information for DASD-label checking and creation. This statement must immediately follow the volume (VOL) statement. The DLAB statement and the continuation statement have the following format.

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

'label fields 1-3' — The first three fields of the Format 1 DASD file label are contained just as they appear in the label. This is a 51-byte character string, contained within single quotes and followed by a comma. The entire 51-byte field must be contained in the first of the two statements. Column 72 must contain a continuation character. The Format 1 label is shown in Appendix D. Fields 1-3 are:

File Name. 44-byte alphameric including file ID and, if used, generation number and version number of generation.

Format Identifier. 1-byte, EBCDIC 1.

File Serial Number. 6-byte alphameric, must be the same as the volume serial number in the volume label of the first or only pack of the file.

xxxx	Volume Sequence Number. This 4-digit EBCDIC number is the EBCDIC equivalent of the 2-byte binary volume sequence number in field 4 of the Format 1 label. This number must begin in column 16 of the continuation statement. Columns 1-15 are blank.
yyddd,yyddd	The File Creation Data, followed by the File Expiration Date. These two 5-digit numbers are the EBCDIC equivalent of the 3-byte discontinuous binary dates in fields 5 and 6 of the Format 1 label. yy is the year (00-99), and ddd is the day of the year (001-366).
'systemcode'	System Code is a 13-character string, within single quotes. For an output file, it is written in field 8 of the Format 1 label. It is ignored when used for an input file. This field is never used by the System/360 Disk Operating System label processing routines, but is essential in order for the files to be processable by Operating System.
NDS	Indicates that a single nonsequential DASD label block is to be composed. This is never used in FORTRAN programs.

XTENT Statement

The extent statement defines each area, or extent, of a DASD file. One XTENT statement must follow each DLAB statement. The XTENT statement has the following format.

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

type — Extent Type. 1 or 3 columns, containing:
  1 = data area (no split cylinder)
128 = data area (split

	cylinder). If type 128 is specified, the lower head is assumed to be $H_1H_2H_2$ from <u>lower</u>, and the upper head is assumed to be $H_1H_2H_2$ from <u>upper</u>.
sequence	<u>Extent Sequence Number</u> must always be 000.
lower	<u>Lower Limit of Extent.</u> 9 columns, containing the lowest address of the extent in the form $B_1C_1C_1C_2C_2C_2H_1H_2H_2$, where: B_1 = initially assigned cell number. 0 for 2311 C_1C_1 = Subcell number. 00 for 2311 $C_2C_2C_2$ = cylinder number. 000 to 199 for 2311 H_1 = head block position. 0 for 2311 H_2H_2 = head number. 00 to 09 for 2311
upper	<u>Upper Limit of Extent.</u> 9 columns containing the highest address of the extent, in the same form as the lower limit. If <u>Extent Type</u> equals 1 the $H_1H_2H_2$ must be 009.
'serial no.'	<u>Volume Serial Number.</u> This is a 6-byte alphameric character string, contained within single quotes. The number is the same as in the volume label (volume serial number) and the Format 1 label (file serial number).
SYSxxx	This is the symbolic address of the DASD drive.
B_2	Currently assigned cell number. 0 for 2311 This field is optional. If missing, Job Control assigns $B_1 = B_2$

The following example illustrates how the user can supply label information to the Disk Operating System for disk files.

```
//ƀJOBƀEXAMPLE

//ƀVOLƀSYS010,IJSYS10

//ƀDLABƀ'STRESSƀINTERMEDIATEƀ...ƀ1006351',ƀ...ƀC
                        (Col. 54)───┘    (Col. 72)───┘
          ƀ...ƀ0001,66185,66185,'16KƀDOSƀDISKƀ'
      (Col. 16)───┘
//ƀXTENTƀ1,0,000006000,000006009,'006351',SYS010

.

.

.

.

(Note:  ƀ is equal to a blank)
```

PROGRAMMING CONSIDERATIONS FOR DIRECT ACCESS FILES

When programming for direct access files, the programmer must consider the following:

1. The area defined by the extents must be preformatted. (Have the correct number of record areas of the correct size written on the disk.) This disk area can be preformatted by using the Clear Disk Utility program described in the <u>Utility Programs</u> publication listed in the <u>Preface</u>.
2. An unformatted WRITE statement may cause only one physical record to be written.
3. Any number of logical units may be assigned to a single physical unit.
4. Only one direct access file may be defined for a single logical unit in a single program.
5. A file written by a direct access WRITE may not be read by a sequential READ statement.

6. The associated variable will be updated to the value of the expression appearing after the quote mark in a FIND statement.

LIBRARY PROCEDURES

The DOS and TOS contain two libraries: relocatable and core image. As their names imply, relocatable object programs are stored in the relocatable library, and executable programs (core image format) are stored in the core image library.

CATALOGING A PROGRAM IN THE RELOCATABLE LIBRARY

To enter a program in the relocatable library, the source program must be compiled and and object program deck must be punched. To obtain an object program deck the programmer must supply an OPTION card (see OPTION card). Using the object program deck in card form as input, the programmer can enter the program into the relocatable library by following the procedure given under Librarian Functions: Relocatable Library in the publication System Control and System Service Programs mentioned in the Preface.

EXAMPLE:

```
// JOB MATINV1  Compile matrix invert
                subroutine

// OPTION DECK,NOLINK

// EXEC FORTRAN

   SUBROUTINE MATINV
        .
        .
        .
   END

/&

// JOB MATINV2  Catalog matrix invert
                subroutine

// EXEC MAINT

bCATALR MATINV
        .
        .
        .       object deck of MATINV

/&
```

A module which has been cataloged to the relocatable library may be included in a program either implicitly or explicitly.

The implicit inclusion is performed by referring to the module name in a FORTRAN program.

The explicit inclusion is performed by use of the linkage editor statement INCLUDE. The following example illustrates both methods.

EXAMPLE: PROG A is a FORTRAN main program. PROG B is a FORTRAN subroutine. PROG A references the name PROGB. Both PROGA and PROGB have been cataloged to the relocatable library. The following statements cause the linkage editing of a program, PROGA, and a subroutine, PROGB. The explicit reference to PROGA will cause both PROGA and PROGB to be included in the program.

```
// JOB EXAMPLE

// OPTION LINK

   INCLUDE PROGA

// EXEC LNKEDT

// EXEC

/&
```

CATALOGING A PROGRAM IN THE CORE IMAGE LIBRARY

To enter a program that has been compiled and linkage edited into the core image library, the programmer must supply an OPTION card specifying CATAL and a /& card following an EXEC card specifying LNKEDT.

EXAMPLE 1: The following cards cause two source modules to be compiled and linkage edited into a program. The program will be cataloged into the core image library.

```
// JOB EXAMPLE

// OPTION CATAL

   PHASE PROG,*

// EXEC FORTRAN

   source program A

/*

// EXEC FORTRAN

   source program B

/*

// EXEC LNKEDT

/&
```

EXAMPLE 2: The following cards cause PROGA to be linkage edited and cataloged into the core image library. PROGA has been previously cataloged to the relocatable library.

```
// JOB RTOC

// OPTION CATAL

   PHASE PROG,*

   INCLUDE PROGA

// EXEC LNKEDT

/&
```

To execute a program that has previously been cataloged to the core-image-library, the following statements must be provided:

```
// JOB XCIL

// EXEC PROG

/&
```

Various suggestions for improving the efficiency of FORTRAN programs are given in this section. Limitations of the compiler are also included.

PROGRAM OPTIMIZATION

By following the suggestions regarding the use of FORTRAN statements as given in this section, a programmer can optimize compilation and execution speed, and can reduce the size of the object module.

ARITHMETIC STATEMENTS

These suggestions apply to programs using exponentiation, square roots, and mixed-mode expressions.

Exponentiation or Multiplication

The use of multiplication instead of the exponential function is recommended when the exponent is a small integer. For example, the following two statements perform the same function:

```
      VOL = (4.*R*R*R)/3.
      VOL = (4.*R**3)/3.
```

The first statement is more efficient than the second because the exponential function requires a library subprogram. When multiplication is used, storage may be conserved and both compiler and linkage editor processing time may be decreased.

Square Root or Exponentiation

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 following two statements specify the same calculation:

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

The first statement is more accurate than the second because the SQRT function is faster and more accurate than the exponential function.

Mixed Mode Expressions

The mixed mode arithmetic expression is provided to reduce errors because of unintentional use of different modes in arithmetic statements. However, when mixed mode arithmetic statements are used, extra instructions are generated.

An in-line subprogram is generated to perform changes in the mode of a variable.

The statements A=A+1 and A=A+1.0 produce identical code. Thus, accuracy and speed are also identical.

IF STATEMENTS

An arithmetic IF statement contains three statement numbers to which branching can occur. One of those numbers should be the number of the statement immediately following the IF statement. This eliminates unnecessary branching within the program phase. For example, the following statements minimize branching:

```
      IF (A-B)20,30,30

30    A=0.
       .
       .
       .
20    B=0.
       .
       .
       .
```

This sequence of statements is more efficient than the following statement:

```
      IF(A-B) 20,30,30

10    X=2.+Y
       .
       .
       .
30    A=0.
       .
       .
       .
20    B=0.
```

DO LOOPS

The suggestions in this section apply to DO loops containing unchanging variables, unvarying subscripts, and subscript calculations.

Unchanging Variables in DO Loops

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 the expression 2.0*(G+ALPHA) must be calculated each time the DO loop is executed:

```
         DO 10 I=1,100

         X(I)=2.0*(G+ALPHA)+Y(I)

     10  CONTINUE
```

For greater efficiency, the following statements should be substituted:

```
         BETA=2.0*(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.

Unvarying Subscripts in DO Loops

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

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

By substituting the following statements, only one subscript calculation is made for Z(J), and execution time is decreased:

```
         B=Z(J)

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

Subscript Calculations in DO Loops

Extra subscript calculation within the range of a DO should be avoided. For example in the following statements two intricate subscript calculations are made each time statement 5 is executed:

```
         DO 10 I=1,10
             .
             .
             .
      5  X(3*I+4)=Y(3*I+4,3*I+4)+B
             .
             .
             .
     10  CONTINUE
```

The DO loop should be rewritten as shown in the following statements to reduce the subscript calculation to simpler terms and allow faster execution of the DO loop:

```
         DO 10 I=7,34,3
             .
             .
             .
      5  X(I)=Y(I,I)+B
             .
             .
             .
     10  CONTINUE
```

READ AND WRITE STATEMENTS

These suggestions apply to READ and WRITE statements that are used for arrays or that contain subscript calculations.

Reading and Writing Arrays

To read or write an array, an implied DO in a READ/WRITE statement should be used instead of a DO loop. For example, five 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 following statements, only one record containing ten values is written:

```
      10 FORMAT (5(F20.5,I10))

         WRITE(5,10)(A(I),J(I),I=1,5)
```

The use of an implied DO saves both program phase execution time and space on the input/output media.

If the entire array is to be read or written, the array should be used instead of an implied DO. The entire array is written by the following statements:

```
20 FORMAT (5F20.5)

25 WRITE (5,20) ((A(I,J),I=1,10),J=1,10)

   or

25 WRITE (5,20)A
```

The latter WRITE statement is preferred.

Subscript Calculations for READ or WRITE

Extra subscript calculation within the range of an implied DO should be avoided for the same reasons given in Subscript Calculations in DO Loops. For example, the following statements contain a complex subscript calculation:

```
      2 FORMAT('0',10F12.6)

      READ(1,2)A(3*I+1),I=1,10)
```

If the following statements are substituted, the subscript calculation is simplified and the program phase execution time is reduced:

```
      2 FORMAT('0',10F12,6)
            .
            .
            .

      READ(1,2)(A(I),I=4,31,3)
```

PROGRAM STRUCTURE

These suggestions concern variables for called programs, and the order of data in common.

Variables for Called Programs

If a large number of variables are to be passed among calling and called programs, some of the variables should be placed in the COMMON area. Consider the following sequence of statements:

```
   DIMENSION E(20),I(15)

   READ (10)A,B,C

   CALL EXAMPL (A,B,C,D,E,F,I)
         .
         .
         .

   END

   SUBROUTINE EXAMPL (X,Y,Z,P,Q,R,J)

   DIMENSION Q(20),J(15)
         .
         .
         .

   RETURN

   END
```

In the main program and subroutine EXAMPL, time and storage are wasted by allocating storage for variables in both the main program and subprogram and also by the subsequent instructions required to transfer variables from one program to another.

The two programs should be written using a COMMON area, as follows:

```
   COMMON A,B,C,D,E(20),F,I(15)

   READ (10)A,B,C

   CALL EXAMPL
         .
         .
         .

   END

   SUBROUTINE EXAMPL

   COMMON X,Y,Z,P,Q(20),R,J(15)
         .
         .
         .

   RETURN

   END
```

Storage is allocated for variables in COMMON only once, and fewer instructions are needed to cross-reference the variables between programs.

Order of Data in COMMON

The efficiency in referencing data in COMMON is affected by the order in which the data appears. For the most efficient usage of COMMON, variables and one dimensional arrays should be the first items of data in COMMON.

COMPILER RESTRICTIONS

Figure 24 shows the average number of source statements that can be handled by the FORTRAN compiler, and the size of the table area used by the compiler. The table area is used by the compiler to contain information concerning variables, arrays, subscripts, functions, data set reference numbers, statement numbers, etc.

Main Storage Available (in bytes)	Average Number of Source Statements	Table Area (in bytes)
10,000	200	1,500
24,000	2,000	15,000
50,000	6,000	32,380
204,800	6,000	32,380

Figure 24. Source Module Size Restriction

A listing of a FORTRAN program is shown. This listing was obtained by specifing LIST in an OPTION card. The listing shows a program that contains no errors detectable by the FORTRAN compiler.

```
                  DISK OR TAPE OPERATING SYSTEM FORTRAN   360B-F0-409      10

C      PRIME NUMBER PROBLEM                                          $4090001
  100 WRITE (3,8)                                                    $4090002
    8 FORMAT (52H FOLLOWING IS A LIST OF PRIME NUMBERS FROM 1 TO 1000/  $4090003
     /19X,1H1/19X,1H2/19X,1H3)                                       $4090004
  101 I=5                                                            $4090005
    3 A=1                                                            $4090006
  102 A=SQRT(A)                                                      $4090007
  103 J=A                                                            $4090008
  104 DO 1 K=3,J,2                                                   $4090009
  105 L=I/K                                                          $4090010
  106 IF(L*K-I)1,2,4                                                 $4090011
    1 CONTINUE                                                       $4090012
  107 WRITE (3,5)I                                                   $4090013
    5 FORMAT (I20)                                                   $4090014
    2 I=I+2                                                          $4090015
  108 IF(1000-I)7,4,3                                                $4090016
    4 WRITE (3,9)                                                    $4090017
    9 FORMAT (14H PROGRAM ERROR)                                     $4090018
    7 WRITE (3,6)                                                    $4090019
    6 FORMAT (31H THIS IS THE END OF THE PROGRAM)                    $4090020
  109 STOP                                                           $4090021
      END                                                            $4090022
```

```
        02/15/66            FORTMAIN                                                 0002


                                                     SCALARS

SYMBOL    LOCATION         SYMBOL    LOCATION        SYMBOL   LOCATION        SYMBOL    LOCATION        SYMBOL    LOCATION
I            006C          A            0070         J           0074         K                         L            007C

                                                     CALLED SUBROUTINES

IJTAPST     IJTACOM      IJTSSQT      SQRT


LABEL     LOCATION         LABEL     LOCATION        LABEL    LOCATION        LABEL     LOCATION        LABEL     LOCATION
00100        0078          00008        0088         00101       0008         00003        00E0         00102        0100
00103        010E          00104        012C         00105       0134         00106        0144         00001        015E
00107        0172          00005        0190         00002       019A         00108        01A6         00004        01BC
00009        01D0          00007        01E8         00006       01FC         00109        0226
          COMPILATION COMPLETE     AMOUNT OF COMMON 000000     AMOUNT OF CORE 000692     ADDRESS BASE TABLE      0200
```

This section contains information to help the programmer select the proper subprogram for solving his problem.

DEFINITION OF SYMBOLS

The symbols used throughout this appendix are:

Symbol	Explanation
$g(x)$	= The result given by the subprogram.
$f(x)$	= The correct extra precision result.

The symbols used in describing the effect of an argument error on the accuracy of the result given by the subprogram are:

Symbol	Explanation
ε	$= \left\lvert \frac{f(x) - g(x)}{f(x)} \right\rvert$ The relative error of the result given by the subprogram.
δ	= The relative error of the argument.
E	$= \lvert f(x) - g(x) \rvert$ The absolute error of the result given by the subprogram.
Δ	= The absolute error of the argument.

The symbols used in describing the accuracy of the result given by the subprogram are:

Symbol	Explanation
$M(E)$	$= \mathrm{Max}\, \lvert f(x) - g(x) \rvert$ The maximum absolute error produced during testing.
$M(\varepsilon)$	$= \mathrm{Max} \left\lvert \frac{f(x) - g(x)}{f(x)} \right\rvert$ The maximum relative error produced during testing.
$\sigma(E)$	$= \sqrt{\frac{1}{N} \sum_i \lvert f(x_i) - g(x_i) \rvert^2}$ The root-mean-square (standard deviation) absolute error.
$\sigma(\varepsilon)$	$= \sqrt{\frac{1}{N} \sum_i \left\lvert \frac{f(x_i) - g(x_i)}{f(x_i)} \right\rvert^2}$ The root-mean-square (standard deviation) relative error.

MATHEMATICAL SUBPROGRAM DESCRIPTIONS

To facilitate quick reference, the following description of the mathematical subprograms are arranged in the order in which they appear on the FORTRAN system tape supplied by IBM. The description of each mathematical subprogram includes the:

1. Purpose.
2. Permissible entry points.
3. Symbolic name of the subprogram.
4. Range.
5. Accuracy.
6. Considerations that should be noted in using the subprogram.
7. Method by which each subprogram is derived.
8. Calling sequence.

ACCURACY

Because the size of a machine word is limited, small errors may be generated by mathematical subroutines. In an elaborate computation, slight inaccuracies can accumulate and become large errors. Thus, in interpreting final results, the user should take into account any errors introduced during the various intermediate stages.

The accuracy of an answer produced by a subroutine is influenced by two factors:

1. The accuracy of the argument, and
2. The performance of the subroutine.

The Accuracy of the Argument

Most arguments contain errors. An error in a given argument may have accumulated over several steps prior to the use of the subroutine. Even data fresh from input conversion contain slight errors. Because decimal data cannot usually be exactly converted into the binary form required by the processing unit the conversion process is usually only approximate. Argument errors always influence the accuracy of answers. The effect of an argument error on the accuracy of an answer depends solely on the nature of the mathematical function involved and not on the particular coding by which that function is computed within a subroutine. In order to assist users in assessing the accumulation of errors, a guide on the propagational effect of argument errors is provided for each function. Wherever possible, this is expressed as a simple formula.

The Performance of the Subroutine

The performance statistics supplied in this appendix are based upon the assumption that arguments are perfect (that is, without errors, and therefore having no argument error propagation effect upon answers). Thus the only errors in answers are those introduced by the subroutines themselves.

For each subroutine, accuracy figures are given for one or more representative segments within the valid argument range(s). In each case the particular statistics given are those most meaningful to the function and range under consideration.

For example, the maximum relative error and standard deviation of the relative error of a set of answers are generally useful and revealing statistics, but useless for the range of a function where its value becomes 0, since the slightest error of the argument value can cause an unbounded fluctuation on the relative magnitude of the answer. Such is the case with sin(x) for x near Π, and in this range it is more appropriate to discuss absolute errors.

METHOD

Some of the formulas are widely known. Others not so widely known are derived from more common formulas. In such cases, the steps leading from the common formula to the computational formula are sketched with enough detail so the derivation may be reconstructed by anyone with a basic understanding of mathematics and with access to the common texts on numerical analysis.

Any of the common numerical analysis texts may be used as a reference. One such text is:

Hildebrand, F.B., - Introduction to Numerical Analysis, McGraw-Hill New York, N.Y., 1956.

Background information for algorithms involving continued fractions may be found in:

Wall, H.S., - Analytic Theory of Continued Fractions, Van Nostrand, Princeton, N.J., 1948.

ALOG Subprogram

PURPOSE: To compute the natural (ALOG) or the common (ALOG10) logarithm of a real number.

ENTRY POINTS: ALOG and ALOG10

MODULE NAME: IJTSLOG

RANGE: $0<x$

ACCURACY: The accuracy of the ALOG subprogram is shown in Figures 25 and 26.

CONSIDERATIONS: Checking is done at object time to ensure that the argument is within the valid argument range; if it is not, an error message is printed and execution is terminated.

METHOD:

1. Write $x = (16^p)(m)$, $\frac{1}{16} \leq m < 1$

ARGUMENT RANGE	σ(E) ROOT-MEAN-SQUARE Absolute ERROR	M (E) MAXIMUM Absolute ERROR
ALOG		
$0.5 \leq x \leq 1.5$	8.62×10^{-8}	3.46×10^{-7}
ALOG10		
$0.5 \leq x \leq 1.5$	4.78×10^{-8}	1.64×10^{-7}
These statistics are based on a uniformly distributed argument sample.		

Figure 25. ALOG Subprogram, Absolute Error

ARGUMENT RANGE	$\sigma(\epsilon)$ ROOT-MEAN-SQUARE Relative ERROR	M (ϵ) MAXIMUM Relative ERROR
ALOG		
x outside (0.5, 1.5)	1.20×10^{-7}	8.32×10^{-7}
ALOG10		
x outside (0.5, 1.5)	2.17×10^{-7}	1.05×10^{-6}
These statistics are based on an exponentially distributed argument sample.		

Figure 26. ALOG Subprogram, Relative Error

2. Define 2 constants a, b (a=base point, 2^{-b}=a) as follows:

   If $\frac{1}{16} \le m < \frac{1}{8}$, $a = \frac{1}{16}$ and $b = 4$

   If $\frac{1}{8} \le m < \frac{1}{2}$, $a = \frac{1}{4}$ and $b = 2$

   If $\frac{1}{2} \le m < 1$, $a = 1$ and $b = 0$.

   Write $z = \frac{m-a}{m+a}$ Then $m = (a)\left(\frac{1+z}{1-z}\right)$,

   and $|z| \le \frac{1}{3}$

3. Now $x = (2^{4p-b})\left(\frac{1+z}{1-z}\right)$ Hence $\log_e x$ =

   $(4p-b) \log_e 2 + \log_e \left(\frac{1+z}{1-z}\right)$

4. $\text{Log}_e \left(\frac{1+z}{1-z}\right)$ is computed using the

   Chebyshev interpolation polynomial of degree 4 in z^2 for the range $0 \le z^2 \le \frac{1}{9}$

   The maximum relative error of this

   approximation is $2^{-27.8}$

5. $\text{Log}_{10} x = (\log_{10} e)\ (\log_e x$

The effect of an argument error is $E \sim \delta$. In particular, if δ is the minimal round-off error of the argument, say $(6)(10^{-8})$ then $E \sim (6)(10^{-}\ )$. This means that if the argument is close to 1, the relative error can be very large, since the function value there is very small.

CALLING SEQUENCE: Out-of-line.

SQRT Subprogram

PURPOSE: To compute the square root of a real number.

ENTRY POINT: SQRT

MODULE NAME: IJTSSQT

RANGE: $0 \le x$

ACCURACY: The accuracy of the SQRT subprogram is shown in Figure 27.

CONSIDERATIONS: Checking is done at object time to ensure that the argument is within the valid argument range. If it is not, an error message is printed and execution is terminated.

METHOD:

1. If x=0, $\sqrt{x} = 0$. Otherwise write x = $(16^{2p})(m)$ where p is an integer and $\frac{1}{256}$

   $\le m < 1$. Then $\sqrt{x} = (16^{p})(\sqrt{m})$ and p and $\sqrt{m}$ are the exponent and the mantissa of the answer respectively.

2. For the 1st approximation of $\sqrt{m}$, take one of the following two hyperbolic approximations of the form a + b/(c + m):

   a. For $\frac{1}{16} \le m < 1$, the values a=

      1.80713, b = -1.57727, c= 0.954182 minimize the maximal relative error ε_0 over the range while making the exact fit at m=1. The exact fitting at m=1 minimizes the computational loss of the last hexadecimal digit for the values of m slightly less than 1. $\varepsilon_0 < 2^{-5.44}$

ARGUMENT RANGE	$\sigma(\epsilon)$ ROOT-MEAN-SQUARE Relative ERROR	M (ϵ) MAXIMUM Relative ERROR
The full range	1.68×10^{-7}	8.70×10^{-7}
These statistics are based on an exponentially distributed argument sample.		

Figure 27. SQRT Subprogram, Relative Error

b. For $\frac{1}{256} \le m < \frac{1}{16}$, the values a = 0.428795, b=-0.0214398, c=0.0548470 minimize $(m^{1/8})(\varepsilon_0)$ over the range where ε_0 denotes the relative error. $\varepsilon_0 < (2^{-6.5})(m^{-1/8})$

3. Apply Newtown-Raphson iteration, $y_{n+1} = \frac{1}{2}\left(yn + \frac{x}{yn}\right)$ twice to the 1st approximation y_0 For $\frac{1}{16} \le m < 1$, the final relative error is theoretically less than $2^{-24.7}$ For $\frac{1}{256} \le m < \frac{1}{16}$, the final absolute error is theoretically less than 2^{-29}.

The effect of an argument error is $\varepsilon \sim 1/2\,\delta$

CALLING SEQUENCE: Out-of-line.

ATAN Subprogram

PURPOSE: To compute the principal value (in radians) of the arctangent of a real number.

ENTRY POINT: ATAN

SYMBOLIC NAME: IJTSTAN

RANGE: Any size real argument is acceptable to this subprogram.

ACCURACY: The accuracy of the ATAN subprogram is shown in Figure 28.

ARGUMENT RANGE	$\sigma(\epsilon)$ ROOT-MEAN-SQUARE Relative ERROR	$M(\epsilon)$ MAXIMUM Relative ERROR
The full range	4.54×10^{-7}	9.75×10^{-7}
These statistics are based on tangents of uniformly distributed numbers between $-\frac{\Pi}{2}$ and $\frac{\Pi}{2}$.		

Figure 28. ATAN Subprogram, Relative Error

METHOD:

1. Reduce computation of ATAN(x) to the case $0 \le x \le 1$ by using Atan(-x) = -Atan(x), Atan$\left(\frac{1}{|x|}\right) = \frac{\Pi}{2} -$ Atan$|x|$.

2. Reduce further to the case $|x| \le \tan 15^\circ = 0.26795$ by Atan(x) = 30° + Atan $\frac{\sqrt{3}x-1}{x+\sqrt{3}}$, $\left|\frac{\sqrt{3}x-1}{x+\sqrt{3}}\right| \le \tan 15^\circ$ if $\tan 15^\circ < x \le 1$.
Here compute $\sqrt{3}x-1$ as $(\sqrt{3}-1)\,x-1+x$ to avoid the loss of significant digits.

3. For $|x| \le \tan 15^\circ$, use the approximation formula:

$$\frac{\text{Atan }(x)}{x} \cong 0.60310579 - 0.05160454x^2 + \frac{0.55913709}{x^2+1.4087812} \quad (*)$$

This formula can be obtained by transforming the continued fraction:

$$\frac{\text{Atan }(x)}{x} = 1 - \frac{1}{3}\; x^2 + \frac{\frac{1}{5}x^2}{\frac{5}{7}+x^2 \quad -w}$$

, after substituting $\left(-\frac{75}{77}x^{-2} + \frac{3375}{77}\right)(10^{-4})$ for $w = \frac{(4)(5)}{(7)(7)(9)} \Big/ \left(\frac{43}{(7)(11)} + x^{-2} + \ldots\right)$

The original continued fraction can be obtained by transforming the Taylor series into a continued fraction form. The relative error of the formula (*) is less than $2^{-27.1}$.

The effect of an argument error is $E \sim \Delta/(1+x^2)$. For small x, $\varepsilon \sim \delta$; and as x becomes large, the effect of δ on ε diminishes.

CALLING SEQUENCE: Out-of-line.

TANH Subprogram

PURPOSE: To compute the hyperbolic tangent of a real number.

ENTRY POINT: TANH

MODULE NAME: IJTSTNH

RANGE: Any size real argument is acceptable to this subprogram.

ACCURACY: The accuracy of the TANH subprogram is shown in Figure 29.

CONSIDERATIONS: The subprogram EXP is used by this subprogram.

METHOD:

1. For $|x| \leq 2^{-12}$, give $\tanh x \cong x$.

2. For $2^{-12} < |x| < 0.54931$, use the following fractional approximation:

$$\frac{\tanh x}{x} = 1 - \frac{x^2+35.1535}{x^2+45.1842+\frac{105.4605}{x^2}}$$

This formula can be obtained by transforming the continued fraction:

$$\frac{1}{1} + \frac{x^2}{3} + \frac{x^2}{5} + \frac{x^2}{7} + w$$

with an approximate value 0.0307 for w.

The relative error of this approximation is less than 2^{-27}.

3. For $0.54931 \leq x < 9.011$, use $\tanh x = 1 - 2/(e^{2x}+1)$.

4. For $9.011 \leq x$, use $\tanh x \cong 1$.

5. For $x \leq -0.54931$, use $\tanh x = -\tanh(-x)$.

6. The exponential subroutine is used in the case 3 above.

The effect of an argument error is $E \sim (1-\tanh^2 x)\Delta, \sim 2\Delta/\sinh 2x$. Thus, for small x, $\varepsilon \sim \delta$, and as x gets larger, the effect of δ on ε diminishes.

CALLING SEQUENCE: Out-of-line.

EXP Suprogram

PURPOSE: To compute the value of "e" raised to the power of a real argument.

ARGUMENT RANGE	$\sigma(\epsilon)$ ROOT-MEAN-SQUARE Relative ERROR	M (ϵ) MAXIMUM Relative ERROR
$\|x\| \leq 0.54931$	1.66×10^{-7}	8.12×10^{-7}
$0.54931<\|x\|\leq 5$	7.53×10^{-8}	5.74×10^{-7}
These statistics are based on a uniformly distributed argument sample.		

Figure 29. TANH Subprogram, Relative Error

ENTRY POINT: EXP

MODULE NAME: IJTEXPN

RANGE: $x<174.673$

ACCURACY: The accuracy of the EXP subprogram is shown in Figure 30.

CONSIDERATIONS: Checking is done at object time to ensure that the argument is within the valid argument range. If it is not, an error message is printed and execution is terminated.

METHOD:

1. If $x < -180.218$, give 0 as the answer.

2. If $|x| < 2^{-28}$, give 1 as the answer.

3. Otherwise, divide x by $\log_e 2$ and write $y = x/\log_e 2 = 4a-b-d$, where a, b are integers, $0 \leq b \leq 3$, $0 \leq d < 1$. Then $e^x = 2^y = (16^a)(2^{-b})(2^{-d})$

4. Compute 2^{-d} by the following fractional approximation:

$$2^{-d} \cong 1 - \frac{2d}{0.034657359d^2+d+9.9545958-\frac{617.97227}{d^2+87.417497}}$$

This formula can be obtained by transforming the well known continued fraction:

$$e \cong \frac{1}{1} - \frac{z}{1} + \frac{z}{2} - \frac{z}{3} + \frac{z}{2} - \frac{z}{5} + \frac{z}{2} - \frac{z}{7} + \frac{z}{2}$$

The maximum relative error of this approximation is 2^{-29}.

5. Multiply 2^{-d} by 2^{-b} by a right shift, and give the hexadecimal exponent of a to obtain 2^y.

ARGUMENT RANGE	$\sigma(\epsilon)$ ROOT-MEAN-SQUARE Relative ERROR	M (ϵ) MAXIMUM Relative ERROR
$\|x\|\leq 1$	1.28×10^{-7}	4.65×10^{-7}
$\|x\|\leq 170$	1.17×10^{-7}	4.69×10^{-7}
These statistics are based on a uniformly distributed argument sample.		

Figure 30. EXP Subprogram, Relative Error

6. Computations are carried out in fixed point to insure accuracy.

The effect of an argument error is $\varepsilon \sim \Delta$. Since $\Delta = (\delta)(x)$, for the larger value of x, even the round-off error of the argument causes a substantial relative error in the answer.

CALLING SEQUENCE: Out-of-line.

COS Subprogram

PURPOSE: To compute the trigonometric sine (SIN) or the trigonometric cosine (COS) of a real argument representing an angle (in radians).

ENTRY POINTS: COS and SIN

MODULE NAME: IJTSSCN

RANGE: $|x| < (2^{18})(\Pi)$

ACCURACY: The accuracy of the COS subprogram is shown in Figures 31 and 32.

CONSIDERATIONS: Checking is done at object time to ensure that the argument is within the valid argument range. If it is not, an error message is printed and execution is terminated.

METHOD:

1. Define $z = \left(\frac{4}{\Pi}\right)(|x|)$ Separate z into the integer part q and the fraction part r $z = q + r$ $|x| = \left(\frac{\Pi}{4}\right) q + \frac{\Pi}{4} r$ A

   long form multiplication is used to obtain accuracy.

2. If cosine is desired, add 2 to q. If sine is desired and if x is negative, add 4 to q.

ARGUMENT RANGE	$\sigma(\epsilon)$ ROOT-MEAN-SQUARE Relative ERROR	M (ϵ) MAXIMUM Relative ERROR
SIN	2.02×10^{-7}	1.59×10^{-6}
$\|x\| \le \frac{\Pi}{2}$	2.02×10^{-7}	1.59×10^{-6}
These statistics are based on a uniformly distributed argument sample.		

Figure 31. COS Subprogram, Relative Error

This reduces the general case to the computation of sin(x) for $x \ge 0$, since

$\cos(\pm x) = \sin\left(\frac{\Pi}{2} + x,\right)$

$\sin(-x) = \sin(\Pi + x)$.

3. Let $q_0 \equiv q \bmod 8$.
   Then for $q_0 = 0$, $\sin(x) = \sin\left(\frac{\Pi}{4} r\right)$

   for $q_0 = 1$, $\sin(x) = \cos\left(\frac{\Pi}{4}\right)(1-r)$

   for $q_0 = 2$, $\sin(x) = \cos\left(\frac{\Pi}{4} r\right)$

   for $q_0 = 3$, $\sin(x) = \sin\left(\frac{\Pi}{4}\right)(1-r)$

   for $q_0 = 4$, $\sin(x) = -\sin\left(\frac{\Pi}{4} r\right)$

   for $q_0 = 5$, $\sin(x) = -\cos\left(\frac{\Pi}{4}\right)(1-r)$

   for $q_0 = 6$, $\sin(x) = -\cos\left(\frac{\Pi}{4} r\right)$

   for $q_0 = 7$, $\sin(x) = -\sin\left(\frac{\Pi}{4}\right)(1-r)$

These formulas reduce the case to the computation of $\left(\sin \frac{\Pi}{4} r_1\right)$ or $\cos\left(\frac{\Pi}{4} r_1\right)$

ARGUMENT RANGE	σ(E) ROOT-MEAN-SQUARE Absolute ERROR	M (E) MAXIMUM Absolute ERROR
SIN		
$\|x\| \le \frac{\Pi}{2}$	5.55×10^{-8}	1.31×10^{-7}
$\frac{\Pi}{2} < \|x\| \le 10$	5.53×10^{-8}	1.41×10^{-7}
$10 < \|x\| \le 100$	5.61×10^{-8}	1.46×10^{-7}
COS		
$0 \le x \le \Pi$	5.48×10^{-8}	1.47×10^{-7}
$-10 \le x < 0, \Pi < x \le 10$	5.67×10^{-8}	1.42×10^{-7}
$10 < \|x\| \le 100$	5.61×10^{-8}	1.35×10^{-7}
These statistics are based on a uniformly distributed argument sample.		

Figure 32. COS Subprogram, Absolute Error

where $r_1 = r$ or $1-r \quad 0 \leq r \leq 1$

4. $\sin\left(\frac{\Pi}{4}r_1\right)$ and $\cos\left(\frac{\Pi}{4}r_1\right)$ are computed

   using the Chebyshev interpolation polynomials of degree 3 in r^2 for the respective functions. The maximum relative error of the sine polynomial is $2^{-28.1}$ and that of the cosine polynomial is $2^{-24.6}$.

The effect of an argument error is $E \sim \Delta$. As the argument gets larger, Δ grows, and since the function value is periodically dimishing, no consistent relative error control can be maintained outside the principal range:

$$\left(-\frac{\Pi}{2}, \frac{\Pi}{2}\right)$$ Similar observation

holds true for cosine as well.

CALLING SEQUENCE: Out-of-line.

DLOG Subprogram

PURPOSE: To compute the natural (DLOG) or the common (DLOG10) logarithm of a double-precision number.

ENTRY POINTS: DLOG and DLOG10

MODULE NAME: IJTLLOG

RANGE: $0<x$

ACCURACY: The accuracy of the DLOG Subprogram is shown in Figures 33 and 34.

CONSIDERATIONS: Checking is done at object time to ensure that the argument is within the valid argument range. If it is not, an error message is printed and execution is terminated.

ARGUMENT RANGE	σ(E) ROOT-MEAN-SQUARE Absolute ERROR	M (E) MAXIMUM Absolute ERROR
DLOG		
$0.5 \leq x \leq 1.5$	7.29×10^{-17}	1.85×10^{-16}
DLOG10		
$0.5 \leq x \leq 1.5$	3.09×10^{-17}	8.23×10^{-17}
These statistics are based on a uniformly distributed argument sample.		

Figure 33. DLOG Subprogram, Absolute Error

METHOD:

1. Write $x = (16^p)(2^{-q})(m)$ where p is the exponent, $0 \leq q \leq 3$ and $1/2 \leq m < 1$.

2. Define 2 constants a, b (a = base point, $2^{-b} = a$) as follows:

   If $\frac{1}{2} \leq m < \frac{1}{\sqrt{2}}$ $\quad a = \frac{1}{2}$, $b = 1$

   If $\frac{1}{\sqrt{2}} \leq m < 1$, $\quad a = 1$, $b = 0$

   Obtain $z = \left(\frac{m-a}{m+a}\right)$ Then $m = (a)\left(\frac{1+z}{1-z}\right)$ and

   $|z| < 0.1716$

3. Now $x = 2^{4-q-b}\left(\frac{1+z}{1-z}\right)$

   Hence $\log_e x = (4p-q-b)\log_e 2 + \log_e \left(\frac{1+z}{1-z}\right)$

4. $\log_e\left(\frac{1+z}{1-z}\right)$ is computed by using the

   Chebyshev interpolation polynomial of degree 7 in z^2 for the range $0 \leq z^2 \leq 0.02944$. The maximum relative error of this polynomial is $2^{-59.6}$.

5. If the common logarithm is wanted, use $\log_{10} x = (\log_{10} e)(\log_e x)$

The effect of an argument error is $E \sim \delta$. This means that if the argument is close to 1, the relative error can be very large, since the function value there is very small.

ARGUMENT RANGE	$\sigma(\epsilon)$ ROOT-MEAN-SQUARE Relative ERROR	M (ϵ) MAXIMUM Relative ERROR
DLOG		
x outside (0.5,1.5)	5.46×10^{-17}	3.31×10^{-16}
DLOG10		
x outside (0.5,1.5)	9.96×10^{-17}	6.14×10^{-16}
These statistics are based on an exponentially distributed argument sample.		

Figure 34. DLOG Subprogram, Relative Error

CALLING SEQUENCE: Out-of-line.

DSQRT Subprogram

PURPOSE: To compute the square root of a double-precision number.

ENTRY POINT: DSQRT

MODULE NAME: IJTLSQT

RANGE: $0 \leq x$.

ACCURACY: The accuracy of the DSQRT Subprogram is shown in Figure 35.

CONSIDERATIONS: Checking is done at object time to ensure that the argument is within the valid argument range. If it is not, an error message is printed and execution is terminated.

METHOD:

1. If $x = 0$, the answer is 0.
2. Write $x = 16^{2p-q}$ (m), where 2p-q is the exponent, q = 0 or 1, and m is the mantissa, $\frac{1}{16} \leq m < 1$ Then $x = (16^{p})(2^{-2q})(\sqrt{m})$
3. Construct the 1st approximation of $\sqrt{x}$ as follows: $y_0 = (2^{-2q})(16^{p})\left(\frac{2}{9} + \frac{8}{9}m\right)$

   Multiplication of 2^{-2} for the odd characteristic case is accomplished by the use of the halve instruction. The maximum relative error of this approximation is $\frac{1}{9}$
4. Apply Newton-Raphson iteration, $Y_{n+1} = \frac{1}{2}\left(yn + \frac{x}{yn}\right)$, 4 times to y_0 as follows:

   a. Twice in the short form, and then

   b. Twice in the long form.

   The last step is performed as $Y_4 = y_3 + \frac{1}{2}\left(\frac{x}{Y_3} - y_3\right)$ to minimize the computational truncation error. The maximum relative error of the final result is theoretically $2^{-65.7}$

The effect of an argument error is $\varepsilon \sim 1/2\delta$

CALLING SEQUENCE: Out-of-line.

DATAN Subprogram

PURPOSE: To compute the principal value (in radians) of the arctangent of a double-precision number.

ENTRY POINT: DATAN

MODULE NAME: IJTLTAN

RANGE: Any size double-precision argument is acceptable to this subprogram.

ACCURACY: The accuracy of the DATAN Subprogram is shown in Figure 36.

METHOD:

1. Reduce the general case to the case $0 \leq x \leq 1$ by using atan(-x) = - atan(x), atan $\frac{1}{|x|} = \frac{\Pi}{2}$ -atan $|x|$
2. Reduce further to the case $|x| \leq \tan 15^\circ$ by $\text{atan}(x) = 30^\circ + \text{atan}\left(\frac{\sqrt{3}x-1}{x+\sqrt{3}}\right)$

   Extra care is taken to avoid a loss of significant digits in computing $\sqrt{3}$ x -1.

ARGUMENT RANGE	$\sigma(\epsilon)$ ROOT-MEAN-SQUARE Relative ERROR	M (ϵ) MAXIMUM Relative ERROR
The full range	2.17×10^{-17}	1.08×10^{-16}
These statistics are based on an exponentially distributed argument sample.		

Figure 35. DSQRT Subprogram, Relative Error

ARGUMENT RANGE	$\sigma(\epsilon)$ ROOT-MEAN-SQUARE Relative ERROR	M (ϵ) MAXIMUM Relative ERROR
The full range	6.64×10^{-17}	2.08×10^{-16}
These statistics are based on tangents of uniformly distributed numbers between $-\frac{\Pi}{2}$ and $\frac{\Pi}{2}$.		

Figure 36. DATAN Subprogram, Relative Error

3. For the basic range $|x| \leq \tan 15^{\circ}$, use a continued fraction of the form:

$$\frac{\text{atan } x}{x} = 1 + \frac{a_1 x^2}{b_1 + x^2} + \frac{a_2}{b_2 + x^2} + \frac{a_3}{b_3 + x^2} + \frac{a_4}{b_4 + x^2} \quad (*)$$

The coefficients were derived by transforming the continued fraction:

$$\frac{\text{atan } x}{x} = 1 + \frac{-\frac{1}{3}}{\frac{3}{5} + x^{-2}} - \frac{\frac{(3)(4)}{(25)(7)}}{\frac{23}{(5)(9)} + x^{-2}} - \frac{\frac{(16)(25)}{(7)(81)(11)}}{\frac{59}{(9)(13)} + x^{-2}} - \frac{\frac{(4)(3)(49)}{(5)(11)(169)}}{\frac{(3)(37)}{(13)(17)} + x^{-2} - w}$$

Here take the approximation $\frac{2}{(5)(11)(13)(17)}$ $(-x^{-2} + 40)$ for the value of w where the true w is:

$$w(x) = \frac{\frac{(64)(27)}{(5)(289)(19)}}{\frac{179}{(3)(7)(17)} + x^{-2}} \text{ further terms}$$

The relative error of the formula (*) is less than $2^{-57.9}$.

The effect of an argument error is $E \sim \Delta/(1+x^2)$. For small $x, \varepsilon \sim \delta$; and as x becomes large, the effect of δ on ε diminishes.

CALLING SEQUENCE: Out-of-line.

DTANH Subprogram

PURPOSE: To compute the hyperbolic tangent of a double-precision number.

ENTRY POINT: DTANH

MODULE NAME: IJTLTNH

RANGE: Any size double precision argument is acceptable to this subprogram.

ACCURACY: The accuracy of the DTANH Subprogram is shown in Figure 37.

CONSIDERATIONS: The subprogram DEXP is used by this subprogram.

METHOD:

1. For $|x| < 0.54931$, use the following fractional approximation: $\frac{\tanh(x)}{x} =$

$$1 - \frac{a_1 x^2 + a_2 x^4 + a_3 x^6 + x^8}{b_0 + b_1 x^2 + b_2 x^4 + b_3 x^6 + x^8} \quad (*)$$

where

a_1	= 676440.765	b_0	= 2029322.295
a_2	= 45092.124	b_1	= 947005.55
a_3	= 594.459	b_2	= 52028.55
		b_3	= 630.476

This formula was obtained by transforming the continued fraction,

$$\frac{\tanh(x)}{x} = \frac{1}{1} + \frac{x^2}{3} + \frac{x^2}{5} + \cdots + \frac{x^2}{15 + w}$$

with an approximate value 0.017 for

$$w = \frac{x^2}{17} + \frac{x^2}{19} + \cdots$$

The maximum relative error of the formula (*) is $2^{-64.5}$.

2. For $0.54931 \leq x \leq 20.101$, use

$$\tanh(x) = 1 - \frac{2}{e^{2x} + 1}$$

3. For $20.101 \leq x$, use $\tanh(x) \cong 1$.

4. For $x \leq -0.54931$, $\tanh(x) = -\tanh(-x)$.

5. The long form exponential function is used in the case 2 above.

ARGUMENT RANGE	$\sigma(\epsilon)$ ROOT-MEAN-SQUARE Relative ERROR	M (ϵ) MAXIMUM Relative ERROR
$\lvert x\rvert \leq 0.54931$	4.45×10^{-17}	2.00×10^{-16}
$0.54931 < \lvert x\rvert \leq 5$	2.54×10^{-17}	1.99×10^{-16}
These statistics are based on a uniformly distributed argument sample.		

Figure 37. DTANH Subprogram, Relative Error

The effect of an argument error is $E \sim (1-\tanh^2 x)\Delta$, $\varepsilon \sim 2\Delta/\sinh 2x$. Thus, for small x, $\varepsilon \sim \delta$, and as x gets larger, the effect of δ on ε diminishes.

CALLING SEQUENCE: Out-of-line.

DCOS Subprogram

PURPOSE: To compute the sine (DSIN) or cosine (DCOS) of a double-precision argument representing an angle (in radians).

ENTRY POINTS: DCOS and DSIN

MODULE NAME: IJTLSCN

RANGE: $|x| < 2^{50} (\Pi)$

ACCURACY: The accuracy of the DCOS Subprogram is shown in Figures 38 and 39.

CONSIDERATIONS: Checking is done at object time to ensure that the argument is within the valid argument range. If it is not, an error message is printed and execution is terminated.

METHOD:

1. Divide $|x|$ by $\frac{\Pi}{4}$ and decompose the quotient into the integer part and fraction part.

   $y = |x| \left(\frac{4}{\Pi}\right) = q + r$, q integer, $0 \le r < 1$.

2. If cosine entry, add 2 to q. If sine entry with negative argument add 4 to q. Let $q_0 \equiv q \bmod 8$.

   $\cos(x) = \sin\left(|x| + \frac{\Pi}{2}\right)$,

   $\sin(-x) = \sin(|x| + \Pi)$.

ARGUMENT RANGE	$\sigma(\epsilon)$ ROOT-MEAN-SQUARE Relative ERROR	M (ϵ) MAXIMUM Relative ERROR
DSIN		
$\|x\| \le \frac{\Pi}{2}$	4.85×10^{-17}	4.08×10^{-16}
These statistics are based on a uniformly distributed argument sample.		

Figure 38. DCOS Subprogram, Relative Error

3. Now the answer is

   $\sin \frac{\Pi}{4}(q_0 + r) \qquad 0 \le q_0 \le 7$

Compute	$\sin \frac{\Pi}{4} r$	if $q_0 = 0$ or 4
	$\cos \frac{\Pi}{4}(1-r)$	if $q_0 = 1$ or 5
	$\cos \frac{\Pi}{4} r$	if $q_0 = 2$ or 6
	$\sin \frac{\Pi}{4}(1-r)$	if $q_0 = 3$ or 7

   $\frac{1}{r_0} \sin \frac{\Pi}{4} r_0$, where r_0 is r or $1-r$, is computed by the use of the Chebyshev interpolation polynomial of degree 6 in $r_0{}^2$ in the range $0 \le r_0{}^2 \le 1$. The maximum relative error of this polynomial is 2^{-58}.

   $\cos \frac{\Pi}{4} r_0$ is computed by using the Chebyshev interpolation polynomial of degree 7 in $r_0{}^2$ in the range $0 \le r_0{}^2 \le 1$. The maximum relative error of this polynominal is $2^{-64.3}$.

4. If $q \le 4$, give negative sign to the result.

ARGUMENT RANGE	σ(E) ROOT-MEAN-SQUARE Absolute ERROR	M (E) MAXIMUM Absolute ERROR
DSIN		
$\|x\| \le \frac{\Pi}{2}$	2.17×10^{-17}	9.10×10^{-17}
$\frac{\Pi}{2} < \|x\| \le 10$	6.35×10^{-17}	1.64×10^{-16}
$10 < \|x\| \le 100$	1.03×10^{-15}	2.69×10^{-15}
DCOS		
$0 \le x \le \Pi$	6.40×10^{-17}	1.79×10^{-16}
$-10 \le x < 0, \Pi < x \le 10$	5.93×10^{-17}	1.76×10^{-16}
$10 < \|x\| \le 100$	1.01×10^{-15}	2.65×10^{-15}
These statistics are based on a uniformly distributed argument sample.		

Figure 39. DCOS Subprogram, Absolute Error

The effect of an argument error is $E \sim \Delta$. As the argument gets larger, Δ grows, and since the function value is periodically dimishing, no consistent relative error control can be maintained

outside the principal range $\left(-\frac{\Pi}{2},\frac{\Pi}{2}\right)$.

This holds for both sine and cosine.

CALLING SEQUENCE: Out-of-line.

DEXP Subprogram

PURPOSE: To compute the value of "e" raised to the power of a double-precision argument.

ENTRY POINT: DEXP

MODULE NAME: IJTLEXP

RANGE: $x<174.67309$

ACCURACY: The accuracy of the DEXP subprogram is shown in Figure 40.

CONSIDERATIONS: Checking is done at object time to ensure that the argument is within the valid argument range. If it is not, an error message is printed and execution is terminated.

METHOD:

1. If $x< -180.2183$, give 0 as the answer.

2. Divide x by $\log_e 2$ and decompose the quotient as:

   $y = x / \log_e 2 = 4a - b - \frac{c}{16} - d$

   where a, b, and c are integers,

   $0 \le b \le 3,\ 0 \le c \le 15$ and $0 \le d < \frac{1}{16}$

ARGUMENT RANGE	$\sigma(\epsilon)$ ROOT-MEAN-SQUARE Relative ERROR	M (ϵ) MAXIMUM Relative ERROR
$\|x\| \le 1$	7.49×10^{-17}	2.27×10^{-16}
$1<\|x\|\le 20$	8.69×10^{-16}	2.31×10^{-15}
$20<\|x\|\le 170$	9.33×10^{-16}	2.33×10^{-15}
These statistics are based on a uniformly distributed argument sample.		

Figure 40. DEXP Subprogram, Relative Error

   Then $e^x = 2^y = (16^a)\ (2^{-b})(2^{-c/16})(2^{-d})$

3. Compute 2^{-d} by using the Chebyshev interpolation polynominal of degree 6 over the range $0 \le d < \frac{1}{16}$ The maximum relative error of this polynominal is 2^{-57}.

4. If $c > 0$, multiply 2^{-d} by $2^{-c/16}$. The 15 constants $2^{-c/16}$, $1 \le c \le 15$ are included in the subroutine.

5. If $b > 0$, halve the result b - times.

6. Multiply by 16^a by adding a to the characteristic of the result.

The effect of an argument error is $\varepsilon \sim \Delta$. Since $\Delta = (\varepsilon)(x)$, for the larger value of x, even the round-off error of the argument causes substantial relative error in the result.

CALLING SEQUENCE: Out-of-line.

MOD Subprogram

PURPOSE: To compute the result of the first integer argument modulo the second integer argument. Modulo is a mathematical operator which yields the remainder function of division. Thus, 9 modulo $6 \equiv 3$.

ENTRY POINTS: MOD

MODULE NAME: IJTMODI

RANGE: Any arguments, except as noted under considerations, are acceptable to this subprogram.

ACCURACY: No error is produced by the computation in this subprogram.

CONSIDERATIONS: Checking is done at object time to ensure that the second argument is not zero. If it is zero, the first argument is given as the result.

METHOD: MOD (x,y)=x-(x/y)*y

The result is calculated according to the above formula. If this result is negative, the absolute value of modulus y is added to it, giving a nonnegative value less than y.

CALLING SEQUENCE: Out-of-line.

AMOD Subprogram

PURPOSE: To compute the result of the first real-number argument (AMOD) or double-precision argument (DMOD) modulo the second argument. Modulo is a mathematical operator which yields the remainder function of division. Thus 3≡39 modulo 6.

ENTRY POINTS: AMOD and DMOD.

MODULE NAME: IJTMODR

RANGE: Any arguments, except as noted under considerations, are acceptable to this subprogram.

ACCURACY: No error is produced by the computation in this subprogram.

CONSIDERATIONS: Checking is done at object time to ensure that the second argument is not zero. If it is zero the first argument is given as the result.

METHOD: AMOD or DMOD (x,y) = x - (x/y) * y

The result is calculated according to this formula. If this result is negative, the absolute value of modulus y is added to it. giving a nonnegative value less than y.

CALLING SEQUENCE: Out-of-line.

MAX0 Subprogram

PURPOSE: To select the maximum (AMAX0 or MAX0) or the minimum (AMIN0 or MIN0) integer value from a list of integer numbers, with the option of converting the result (AMAX0 or AMIN0) to a real number or giving an integer result (MAX0 or MIN0).

ENTRY POINTS: MAX0, AMAX0, MIN0, and AMIN0

MODULE NAME: IJTSMX0

RANGE: Any size argument is acceptable to this subprogram.

ACCURACY: No error is produced by the computation in this subprogram.

CONSIDERATIONS: None.

METHOD: Starting with the second argument and proceeding to the end of the argument list, each argument is algebraically compared to the maximum (AMAX0 and MAX0) or to the minimum (AMIN0 and MIN0) of the previous arguments to yield a new maximum or minimum. For AMAX0 or AMIN0 the result is then converted to a real number.

CALLING SEQUENCE: Out-of-line.

MAX1 Subprogram

PURPOSE: To select the maximum (AMAX1 or MAX1) or the minimum (AMIN1 or MIN1) real value from a list of real numbers, with the option of converting the result (MAX1 or MIN1) to an integer number or giving a real-number result (AMAX1 or AMIN1).

ENTRY POINTS: MAX1, AMAX1, MIN1, and AMIN1

MODULE NAME: IJTSMX1

RANGE: Any size argument is acceptable to this subprogram.

ACCURACY: No error is produced by the computation in this subprogram.

CONSIDERATIONS: None.

METHOD: Starting with the second argument and proceeding to the end of the argument list, each argument is algebraically compared to the maximum (AMAX1 and MAX1) or to the minimum (AMIN1 and MIN1) of the previous arguments to yield a new maximum or minimum. For MAX1 or MIN1 the result is then converted to an integer number.

CALLING SEQUENCE: Out-of-line.

DMAX1 Subprogram

PURPOSE: To select the maximum (DMAX1) or the minimum (DMIN1) double-precision value from a list of double-precision numbers.

ENTRY POINTS: DMAX1 and DMIN1

MODULE NAME: IJTMAXD

RANGE: Any size argument is acceptable to this subprogram.

ACCURACY: No error is produced by the computation in this subprogram.

CONSIDERATIONS: None.

METHOD: Starting with the second argument and proceeding to the end of the argument list, each argument is algebraically compared to the maximum (DMAX1) or to the minimum (DMIN1) of the previous arguments to yield a new maximum or minimum.

CALLING SEQUENCE: Out-of-line.

IFIX Subprogram

PURPOSE: To convert a single real number to an integer number (IFIX) and to truncate the fractional portion of the mantissa of a real number and convert the modified real value to an integer number.

ENTRY POINTS: IFIX, INT and IDINT.

MODULE NAME: IJTIFIX

RANGE: $|x| \leq (2**31)$

ACCURACY: There is a loss of precision when absolute x is greater than 2**24.

CONSIDERATIONS:

1. The library subprogram IFIX is used to convert the modified value.

2. If the value exceeds the valid argument range, the result is given as zero.

METHOD: An unnormalized add of floating point zero is used to shift the fractional part of the absolute value of the argument out of the register. The exponent is discarded and the sign of the argument is transferred to the result.

CALLING SEQUENCE: In-line.

AINT Subprogram

PURPOSE: To truncate the fractional portion of the mantissa of a real number.

ENTRY POINT: AINT

MODULE NAME: IJTSINT

RANGE: Any size argument is acceptable to this subprogram.

ACCURACY: No error is produced by the computation in this subprogram.

CONSIDERATIONS: If the absolute value of the argument is less than one the result is zero.

METHOD: AINT (x) = sign of x times the largest integer $\geq |x|$.

The fractional portion of the argument is deleted.

CALLING SEQUENCE: Out-of-line.

SERVICE SUBPROGRAM DESCRIPTIONS

The FORTRAN service subprograms are all out-of-line subprograms. The description of each service subprogram includes:

1. Purpose.

2. Permissible entry points.

3. Symbolic name of subprogram.

4. Format.

5. Storage requirements.

6. Considerations that should be noted in using the subprogram.

7. Usage.

In the following descriptions , i represents an integer expression and j represents an integer variable.

EXIT Subprogram

PURPOSE: To terminate the execution of a program and returns control to the system director.

ENTRY POINT: EXIT

MODULE NAME: IJTFXIT

FORMAT: CALL EXIT

CONSIDERATIONS: This subprogram performs the same function as the STOP statement. A program written in assembler language may use the EXIT subprogram. A program written in FORTRAN may use either the STOP statement or the EXIT subprogram.

USAGE: Control is given to a routine which performs internal services and returns control to the Disk or Tape Operating System.

DUMP Subprogram

PURPOSE: To dump the indicated limits of storage in the specified format with (DUMP) or without (PDUMP) program termination.

ENTRY POINTS: DUMP and PDUMP

MODULE NAME: IJTFDMP

FORMAT: CALL DUMP or CALL PDUMP $(A_1, B_1, F_1, \ldots, A_n, B_n, F_n)$

Where A and B are variable data names indicating the limits of storage to be dumped, either A or B may represent the upper or lower limits of storage to be dumped, and F is an integer that indicates the format of the dump.

CONSIDERATIONS: None.

USAGE: Each set of A, B, and F parameters are treated separately. If no parameters are given, all storage is dumped in hexadecimal. The A and B parameters are examined to determine which indicates the lower limit and which indicates the upper limit of storage to be dumped.

The value of F is used to determine the format of the dump as follows:

0 specifies a hexadecimal dump format
4 specifies an integer dump format
5 specifies a real dump format
6 specifies a double-precision dump format.

If F is not specified, the dump is in hexadecimal format. The appropriate conversion routine is then called to format the records to be dumped and these records are written by the input/output routine.

SLITE Subprogram

PURPOSE: To turn sense lights on or off (SLITE), and test whether a sense light is on or off (SLITET).

ENTRY POINTS: SLITE and SLITET

MODULE NAME: IJTSLIT

SLITE SUBPROGRAM: This subprogram turns sense lights on or off.

FORMAT: SLITE (i) where i is 0, 1, 2, 3, or 4.

CONSIDERATIONS: If the value of i is not 0, 1, 2, 3, or 4 a message is printed and execution is terminated.

USAGE: The value of i is tested and the corresponding sense light turned on. If the value of i is zero, all sense lights are turned off. Each sense light occupies one byte of storage. When the byte contains zeros it is off. When it does not contain zeros, it is on.

SLITET SUBPROGRAM: This subprogram tests whether a sense light is on or off.

FORMAT: SLITET (i,j), where i is 1, 2, 3, or 4; j is set to 1 or 2.

CONSIDERATIONS: If the value of i is not 1, 2, 3, or 4 a message is printed and execution is terminated.

USAGE: The value of i is tested and the corresponding sense light is examined. j is then set to 1 if the sense light was on or 2 if the sense light was off. The sense light that was tested is then turned off. Each sense light occupies one byte of storage. When the byte contains zeros it is off; when it does not contain zeros it is on.

OVERFL Subprogram

PURPOSE: To test for exponent overflow or underflow.

ENTRY POINT: OVERFL

MODULE NAME: IJTOVRF

FORMAT: OVERFL (j) Where j is set to 1, 2, or 3.

CONSIDERATIONS: None.

USAGE: The status of the overflow indicator is examined. j is set to 1 if a floating-point overflow condition exists (that is, if the result of an arithmetic operation is greater than 16^{63}). j is set to 2 if no overflow condition exists. j is set to 3 if a floating-point underflow condition exists (that is, if the result of an arithmetic operation is not zero but less than 16^{-63}). After j is set, the machine is left in a no overflow condition.

DVCHK Subprogram

PURPOSE: To test for divide check interruptions.

ENTRY POINT: DVCHK

MODULE NAME: IJTOVCK

FORMAT: DVCHK (j) where j is set to 1 or 2.

CONSIDERATIONS: None.

USAGE: The status of the divide check indicator is examined. j is set to 1 if the indicator is on, or 2 if the indicator is off. After j is set, the indicator is turned off. The divide check indicator occupies one byte of storage. When the byte contains zeros, it is off. When it contains any other bit pattern, it is on.

APPENDIX C: MESSAGES

The diagnostic messages produced during the compilation and execution of a FORTRAN program have been divided into two groups, unnumbered and numbered. The unnumbered messages are listed first in alphabetical order. The numbered messages are in numerical order.

UNNUMBERED MESSAGES

These messages, which can appear in the source program listing, indicate errors in the source program. Some of them print following the statement containing the error. These are called statement error messages. The other messages print at the end of the program listing. These are called summary error messages and pertain to errors that the compiler cannot attribute to one particular statement.

In the listing of these errors, the word following Explanation indicates the type of error. The word Statement identifies a statement error message. Summary identifies a summary error message.

ALLOCATION

Explanation: Statement. The storage allocation indicated by the preceding source program statement cannot be done. Either the name of a variable has been used improperly, or there is an inconsistancy between the present usage of the name of a variable and some previous usage of that same name. The following are examples of this type of statement error:

1. A name in a COMMON statement is a dummy variable or has been listed in a previous COMMON statement.

2. A variable in an EQUIVALENCE statement is followed by more than three subscripts.

3. A name in an explicit specification statement has already been defined in an explicit specification.

ARRAY ERRORS

Explanation: Summary. The array names listed following this message have been difined as requiring more than 32,768 bytes.

COMMA

Explanation: Statement. A comma is missing from the preceding source program statement. This error message can occur for a DEFINE FILE statement, an EQUIVALENCE statement, a DIMENSION statement, a computed GO TO statement, or any explicit specification statement.

COMMON ALLOCATION ERRORS

Explanation: Summary. An error in the allocation of the COMMON storage area has been detected. This message is followed by a list of the variables for which storage could not be allocated because of the error. The following are examples of this type of error:

1. There is a contradiction between COMMON and EQUIVALENCE statements. For example, an EQUIVALENCE statement sets (A,B(6), C(2) and (B(8), C(1)), where A is a variable in a COMMON statement.

2. An attempt to extend the beginning of the COMMON area has been made. For example, COMMON A,B,C and EQUIVALENCE (A,F(10)).

3. An attempt has been made to allocate a double-precision variable to a location that is not on a double word boundary with the COMMON area (Note that the COMMON area begins on a double word boundary). This error can be produced by either a COMMON or an EQUIVALENCE statement.

COMPILATION TERMINATED

Explanation: Summary. Source program errors caused the compiler to stop compilling before any cards were punched. This message is printed regardless of the print option selected by the programmer. Whenever this message is printed, it is the last one on the listing.

COMPILATION TERMINATED, DATA OVERFLOW

Explanation: Summary. The program requires more than 65,532 bytes of storage for the data area(s), excluding the COMMON area. When this message is printed, some cards of the object program have already been punched.

COMPILATION TERMINATED, PROGRAM OVERFLOW

Explanation: Summary. The program requires more than 65,532 bytes of storage for object program instructions. When this message is printed, some cards of the object program have already been punched.

DATA OVERFLOW

Explanation: Summary. Too much space has been allocated for variables (as opposed to object program instructions). This message indicates one of the following conditions:

1. More than 65,532 bytes of storage have been allocated for COMMON.

2. More than 65,532 bytes of storage have been allocated for variables not in COMMON.

Note that this message will be printed twice if both of the above conditions occur during the compilation.

DUP. LABEL

Explanation: Statement. The label of the preceding source program statement has been used as the label of a statement earlier in the program.

FUNCTION NAME NOT REFERENCED

Explanation: Summary. The source program being compiled is a FUNCTION subprogram, but no unsubscripted variable with the same name as the function has been set.

ID CONFLICT

Explanation: Statement. The name of a variable or subprogram is used improperly. That is, the type of the variable is incorrect for the present usage as determined either by some previous source program statement or by a previous part of the present statement. The following are examples of this type of statement error:

1. The name of a SUBROUTINE subprogram appears as a part of the arithmetic expression.

2. The DO statement controls for an input/output list are either not scalar (nonsubscripted) variables or not integers.

3. The same name appears more than once in the dummy list of an arithmetic function definition statement.

4. A name listed in an EXTERNAL statement has already been defined by the program as a variable or an array.

ILLEGAL LBL

Explanation: Statement. A defined label is used illegally in the preceding source program statement. Either the statement specifies a branch to a FORMAT statement or the statement does not use the label of a FORMAT statement where one is required.

LABEL

Explanation: Statement. A label has been omitted from the preceding IF, GO TO, RETURN, or STOP statement. This is the only statement error detected by the compiler that does not cause compilation to be terminated.

NAME LENGTH

Explanation: Statement. A name in the preceding source program statement is not usable as is. This message indicates one of the following conditions:

1. The name of a variable or subprogram consists of more than six characters.

2. Two variable names appear in an arithmetic expression without a separate operation symbol.

3. The name of a variable is followed by a number without an intervening symbol to separate them.

NO CORE

Explanation: Statement. There is not enough main storage to compile the statement or the program. If this error is caused by a single statement, compilation may be possible if the programmer rewrites that statement to form two or more statements. If this error occurs throughout the source program, the programmer must reduce either the number of variables or the number of statements in the source program.

NO MORE CORE d

Explanation: Summary. The compiler does not have enough main storage to complete the compilation of the source program. The value of d indicates the section of the table area that is too small.

Value of d	Meaning
1	The dynamic table area is completely full.
2	The source program exceeds one of the limits given under Program Features if the error is not covered by another message.
3-4	One of the two fixed sized tables has been exceeded. This is caused by source statements that are too long.

NON-COMMON EQUIVALENCE ERRORS

Explanation: Summary. An error in allocation caused by the arrangement of EQUIVALENCE sets that do not refer to variables in the COMMON area has been detected. This message is followed by a list of the variables for which storage could not be allocated because of the error. This message indicates one of the following conditions:

1. There is a conflict between two EQUIVALENCE sets. For example, (A,B(6),C(3)) and (B(8),C(1)).

2. An attempt has been made to allocate a double precision variable to a location that is not on a double word boundary (Note that the first double precision variable in the EQUIVALENCE statement is forced to a double word boundary).

ORDER

Explanation: Statement. The preceding source language statement is used in an incorrect sequence. This message indicates one of the following conditions:

1. Either a FUNCTION or a SUBROUTINE statement appears after the first statement of the program.

2. A specification statement appears after the first active statement of the program.

The statement that caused this error message is not processed. Therefore, the information contained in the statement is not available to the compiler. For example, if this type of error occurs in a DIMENSION statement, the array definitions in the statement are ignored.

SIZE

Explanation: Statement. A number, including a label used in a statement, is outside the legal range of values for its type. This error message may indicate that the decimal point has been omitted from a floating point constant.

SUBSCRIPT

Explanation: Statement. An incorrect number of subscripts has been on an array variable in the preceding source program statement.

SYNTAX

Explanation: Statement. The preceding source program statement or part of it does not conform to the rules of FORTRAN or the statement cannot be identified at all. The following are some additional conditions that cause this message to be printed:

1. The label contains a character that is not a digit.

2. The variable in a DO statement is not followed by an equal sign.

3. Extraneous information follows a complete statement on a card.

4. There are less than three labels following the arithmetic expression in an IF statement.

5. The left and right parentheses in an arithmetic expression do not match.

6. The dimensions of a dimensional variable are not integers.

7. A constant that begins with a decimal point does not have a digit as its second character.

UNCLOSED DO LOOP TARGETS

Explanation: Summary. Labels referred to in DO statements have not been defined (that is, used as labels) in the source program. This message is followed by a list of the labels that were used in DO statements but not defined.

UNDEFINED LABELS

Explanation: Summary. The source program contains undefined labels. This message is followed by a list of the undefined labels.

UNDIMENSIONED

Explanation: Statement. A variable name that is not a defined array variable is followed by a left parenthesis. If the variable name is on the left-hand side of an assignment statement, the message may indicate that an arithmetic statement function definition is misplaced.

NUMBERED MESSAGES

These messages indicate errors that occur during the execution of the object program. Each message consists of a message number that is written on SYSLST when the error is detected. The messages appear in the form IJTnnnI, where nnn is the message number.

In the listing of these messages, the word following Explanation indicates the source of the error. If the source is a subroutine, the word is the symbolic name of a subprogram. Note that all errors, except those marked with an asterisk, cause the job to terminate.

Message Number

212 Explanation: Data. An input or output record for which a FORMAT statement has been specified is more than 255 bytes long.

213 Explanation: Data. The data read from a logical record for which no FORMAT statement has been specified does not fill the input list.

214 Explanation: Program. The output list is too long. The writing of a logical record for which no FORMAT statement has been specified produces a logical record containing more than 255 physical records.

215 Explanation: Program. An attempt has been made to execute direct access statements while operating under control of Tape Operating System.

216 Explanation: Program. An attempt has been made to read from a device that can be used for output only.

217 Explanation: Program. An attempt has been made to write on a device that can be used for input only.

218 Explanation: Program. The number of a FORTRAN logical unit is not between 1 and 15, inclusive.

219 Explanation: Data. An end of file has been read on a disk, tape or card reader.

220 Explanation: Data. The physical end of the tape has been encountered.

221 Explanation: Program. An END FILE, REWIND, or BACKSPACE operation refers to a device on which these operations cannot be done.

222* Explanation: Data. An attempt has been made to backspace a file that has a record count of zero.

223 Explanation: Data. An input or output record for which a FORMAT statement has been specified contains an illegal character.

224 Explanation: Program. The number of a reference sense light is not 1, 2, 3, or 4.

225* Explanation: Data. An arithmetic program interruption has occurred. This message is followed by the old PSW, which has the following format:

```
xxxxxxxixxxxxxxxx
```

The values of x do not apply to this message. The value of i indicates the cause of the arithmetic program interrupt.

Value of i	Cause of Interrupt
9	Fixed-point divide exception
B	Decimal divide exception
C	Exponent overflow exception
D	Exponent underflow exception
F	Floating-point divide exception

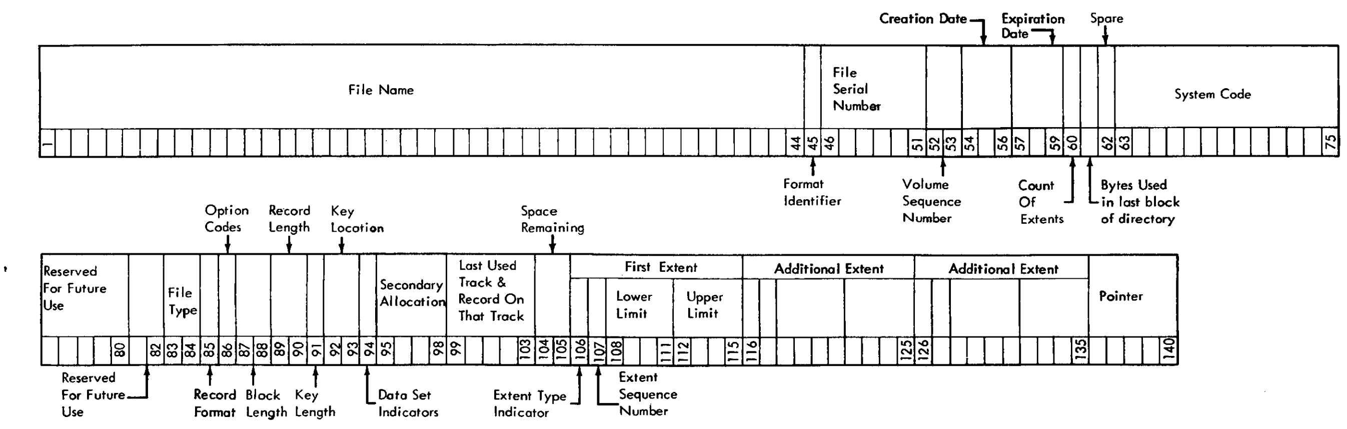
227 Explanation: Program. The unit number, record size, or number of records in a define file statement are equal to zero or negative.

228 Explanation: Program. The record length is greater than 1726.

229 Explanation: Program. A find or a direct read/write statement without a define file statement.

Message Number	
230	Explanation: Program. A find or a direct read/write statement addressed a device that is not a disk.
231	Explanation: Program. A direct access record number in a find or read/write statement is less, or equal to 0, or greater than the maximum number of records in the file.
232	Explanation: Program. An attempt to position the disk pointer below cylinder MIN, or Tract MIN, or Record 1.
233	Explanation: Program. The end of the extent on disk has been reached using a sequential Write.
234	Explanation: Program. An unformatted direct write statement is writing more than one physical record.
235	Explanation: Program. The direct I/O record size is too large to fit in the unused main storage.
236	Explanation: Program. There is not enough main storage to assign a buffer area.
237	Explanation: Program. No record found. An attempt to read a record on disk that did not exist or write a record on disk that could not be performed.
241	Explanation: Data. While attempting to raise an integer base to an integer power, the routine has found the base to be equal to zero and the exponent to be negative or zero.
242	Explanation: Data. While attempting to raise a real base to an integer power, the routine has found the base to be equal to zero and the exponent to be negative or zero.
243	Explanation: Data. While attempting to raise a double precision base to an integer power, the routine has found the base to be equal to zero and the exponent to be negative or zero.
244	Explanation: Data. While attempting to raise a real base to a real power, the routine has found the base to be equal to zero and the exponent to be negative or zero.
245	Explanation: Data. While attempting to raise a double precision base to a real power, the routine has found the base to be equal to zero and the exponent to be negative or zero.
251	Explanation: Data. The argument for the single precision square root function is negative.
252	Explanation: Data. The argument for the single precision exponential function is too large; that is, greater than 174.673.
253	Explanation: Data. The argument for a single precision logarithmic function is zero or negative. This message may also be given to implicit calls to the exponential function.
254	Explanation: Data. The absolute value of the argument for the single precision sine or cosine function is too large, that is, equal to or greater than Π *2**18.
261	Explanation: Data. The argument for the double precision square root function is negative.
262	Explanation: Data. The argument for the double precision exponential function is too large; that is, greater than 174.67309.
263	Explanation: Data. The argument for the double precision logarithmic function is zero or negative. This message may also be given to implicit calls to the exponential function.
264	Explanation: Data. The absolute value of the argument for the double precision sine or cosine function is too large; that is, equal to or greater than Π *2**50.

Note: There are other messages that may occur during the execution of a FORTRAN object program. However, those messages are produced by other components of the Disk or Tape Operating System, rather than by FORTRAN. For additional messages, refer to the Tape or Disk Operating Guide listed in the Preface.



Format 1: This format is common to all data files on disk.

FIELD	NAME AND LENGTH	DESCRIPTION
1.	FILE NAME 44 bytes, alphameric EBCDIC	This field serves as the key portion of the file label. It can consist of three sections: 1. File ID is an alphameric assigned by the user and identifies the file. Can be 1-35 bytes if generation and version numbers are used, or 1-44 bytes if they are not used. 2. Generation Number. If used, this field is separated from File ID by a period. It has the format Gnnnn, where G identifies the field as the generation number and nnnn (in decimal) identifies the generation of the file. 3. Version Number of Generation. If used, this section immediately follows the generation number and has the format Vnn, where V identifies the field as the version of generation number and nn (in decimal) identifies the version of generation of the file. Note: Disk or Tape Operating System compares the entire field against the file name given in the DLAB card. The generation and version numbers are treated differently by System/360 Operating System.

The remaining fields comprise the DATA portion of the file label:

FIELD	NAME AND LENGTH	DESCRIPTION
2.	FORMAT IDENTIFIER 1 byte, EBCDIC numeric	1 = Format 1
3.	FILE SERIAL NUMBER 6 bytes, alphameric EBCDIC	Uniquely identifies a file/volume relationship. It is identical to the Volume Serial Number of the first or only volume of a multi-volume file.
4	VOLUME SEQUENCE NUMBER 2 bytes, binary	Indicates the order of a volume relative to the first volume on which the data file resides.
5	CREATION DATE 3 bytes, discontinuous binary	Indicates the year and the day of the year the file was created. It is of the form YDD, where Y signifies the year (0-99) and DD the day of the year (1-366).
6	EXPIRATION DATE 3 bytes, discontinuous binary	Indicates the year and the day of the year the file may be deleted. The form of this field is identical to that of Field 5.
7A	EXTENT COUNT 1 byte, binary	Contains a count of the number of extents for this file on this volume. If user labels are used, the count includes the user label track as a separate extent. This field is maintained by the Disk or Tape Operating System programs.
7	BYTES USED IN LAST BLOCK OF DIRECTORY 1 byte, binary	Used by System/360 Operating System only for partioned (library structure) data sets. Not used by Disk or Tape Operating System.
7C	SPARE 1 byte	Reserved for future use.
8	SYSTEM CODE 13 bytes	Uniquely identifies the programming system.
9	RESERVED 7 bytes	This field is reserved for future use.
10	FILE TYPE 2 bytes	The contents of this field uniquely identify the type of data file: Hex 4000 = Consecutive organization Hex 2000 = Direct-access organization Hex 8000 = Indexed-sequential organization Hex 0200 = Library organization Hex 0000 = Organization not defined in the file label.
11	RECORD FORMAT 1 byte	The contents of this field indicate the type of records contained in the file:

Bit Position	Content	Meaning
0 and 1	01	Variable-length records
	10	Fixed-length records
	11	Undefined format
2	0	No track overflow
	1	File is organized using track overflow (System/360 Operating System only)
3	0	Unblocked records
	1	Blocked records

FIELD	NAME AND LENGTH	DESCRIPTION

Bit Position	Content	Meaning
4	0	No truncated records
	1	Truncated records in file
5 and 6	01	Control character ASA code
	10	Control Character machine code
	00	Control Character not stated
7	0	Records have no keys
	1	Records are written with keys.

FIELD	NAME AND LENGTH	DESCRIPTION
12	OPTION CODES 1 byte	Bits within this field are used to indicate various options used in building the file. BIT 0 = If on, indicates data file was created using Write Validity Check. 1-7 = unused
13.	BLOCK LENGTH 2 bytes, binary	indicates the block length for fixed length records or maximum block size for variable length blocks.
14.	RECORD LENGTH 2 bytes, binary	indicates the record length for fixed length records or the maximum record length for variable length records.
15.	KEY LENGTH 1 byte, binary	indicates the length of the key portion of the data records in the file.
16.	KEY LOCATION 2 bytes, binary	indicates the high order position of the data record.
17.	DATA SET INDICATORS 1 byte	Bits within this field are used to indicate the following: BIT 0 If on, indicates that this is the last volume on which this file normally resides. This bit is used by the Disk or Tape Operating System DTFSR routine only. None of the other bits in this byte are used by Disk or Tape Operating System. 1 If on, indicates that the data set described by this file must remain in the same absolute location on the direct access device. 2 If on, indicates that Block Length must always be a multiple of 8 bytes. 3 If on, indicates that this data file is security protected; a password must be provided in order to access it. 4-7 Spare. Reserved for future use.
18.	SECONDARY ALLOCATION 4 bytes, binary	indicates the amount of storage to be requested for this data file at End of Extent. This field is used by System/360 Operating System only. It is not used by Disk or Tape Operaing System routines. The first byte of this field is an indication of the type of allocation request. Hex code "C2" (EBCDIC "B") indicates bytes, hex code "E3" (EBCDIC "T") indicates tracks, and hex code "C3" (EBCDIC "C") indicates cylinders. The next three bytes of this field is a binary number indicating how many bytes, tracks or cylinders are requested.
19.	LAST USED TRACK AND RECORD ON THAT TRACK 5 bytes discontinuous binary	indicates the last occupied track in a consecutive file organization data file. This field has the format CCHHR. It is all binary zeros if the last track in a consecutive data file is not on this volume or if it is not consecutive organization.
20	AMOUNT OF SPACE REMAINING ON LAST TRACK USED 2 bytes, binary	A count of the number of bytes of available space remaining on the last track used by this data file on this volume.
21.	EXTENT TYPE INDICATOR 1 byte	indicates the type of extent with which the following fields are associated: HEX CODE 00 Next three fields do not indicate any extent. 01 Prime area (Indexed Sequential); or Consecutive area, etc., (i.e., the extent containing the user's data records.) 02 Overflow area of an Indexed Sequential file. 04 Cylinder index or master index area of an Indexed Sequential file. 40 User label track area 80 Shared cylinder indicator.
22.	EXTENT SEQUENCE NUMBER 1 byte, binary	indicates the extent sequence in a multi-extent file.
23.	LOWER LIMIT 4 bytes, discontinuous binary	the cylinder and the track address specifying the starting point (lower limit) of this extent component. This field has the format CCHH.
24	UPPER LIMIT 4 bytes	the cylinder and the track address specifying the ending point (upper limit) of this extent component. This field has the format CCHH.
25-28	ADDITIONAL EXTENT 10 bytes	These fields have the same format as the fields 21-24 above.
29-32	ADDITIONAL EXTENT 10 bytes	These fields have the same format as fields 21-24 above.
33	POINTER TO NEXT FILE LABEL WITHIN THIS LABEL SET 5 bytes, discontinuous binary	the disk address (format CCHHR) of a continuation label if needed to further describe the file. If field 9 indicates Indexed Sequential organization, this field will point to a Format 2 file label within this label set. Otherwise, it points to a Format 3 file label, and then only if the file contains more than three extent segments. This field contains all binary zeros if no additional file label is pointed to.

READER'S COMMENT FORM

IBM System/360 C24-5038-0
Disk and Tape Operating Systems
FORTRAN IV Programmer's Guide

- Your comments, accompanied by answers to the following questions, help us produce better publications for your use. If your answer to a question is "No" or requires qualification, please explain in the space provided below. All comments will be handled on a non-confidential basis. Copies of this and other IBM publications can be obtained through IBM Branch Offices.

	Yes	No
● Does this publication meet your needs?	☐	☐
● Did you find the material:		
Easy to read and understand?	☐	☐
Organized for convenient use?	☐	☐
Complete?	☐	☐
Well illustrated?	☐	☐
Written for your technical level?	☐	☐

- What is your occupation? ____________________
- How do you use this publication?

As an introduction to the subject?	☐	As an instructor in a class?	☐
For advanced knowledge of the subject?	☐	As a student in a class?	☐
For information about operating procedures?	☐	As a reference manual?	☐

Other ____________________

- Please give specific page and line references with your comments when appropriate. If you wish a reply, be sure to include your name and address.

COMMENTS:

- Thank you for your cooperation. No postage necessary if mailed in the U. S. A.

C24-5038-0

Fold Fold

FIRST CLASS
PERMIT NO. 170
ENDICOTT, N. Y.

BUSINESS REPLY MAIL
NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .

IBM Corporation
P. O. Box 6
Endicott, N. Y. 13760

Attention: Programming Publications, Dept. 157

Fold Fold

Cut Along Line

IBM S/360 Printed in U.S.A. C24-5038-0

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]

Additional Comments:

C24-5038-0

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]