HONEYWELL EDP

SOFTWARE MANUAL

SERIES 200

FORTRAN COMPILER D

GENERAL SYSTEM:	SERIES 200/OPERATING SYSTEM - MOD 1
SUBJECT:	The Honeywell Fortran Compiler D and its associated software. Also considered in this manual are the Fortran language elements used by the compiler and the modes of processing possible with Fortran Compiler D, including programming techniques and operating procedures.
SPECIAL INSTRUCTIONS:	This publication supersedes the Fortran Compiler D Reference Manual, File No. 123.1305.001D.1-027.

DATE: June 15, 1966

FILE NO. 123.1305.001D.2-027*

8768
5666
Printed in U. S. A.

*When ordering this publication please specify Title and Underscored portion of File Number.

FOREWORD

This manual describes the Honeywell Series 200 Fortran Compiler D and its associated software, the Diagnostic Preprocessor and the Screen conversion routine.

Hardware characteristics of the Series 200 are described in the Models 200/1200/2200 Programmers' Reference Manual, Order No. 139, and the Model 120 Programmers' Reference Manual, Order No. 141. Series 200 computers use a six-bit alphanumeric character as the basic data unit. Fortran D uses a minimum memory of 16,384 characters to compile and execute. The compiler can use up to 32,768 characters, and execution can use up to 262,144 characters. Fortran D can be run on any Series 200 computer having the required minimum configuration for the compiler. On a basic Model 120 (i.e., using the integrated peripheral control), there is a restriction that card reading and punching by the object program should not be interspersed, since a peripheral error may not be detected if these operations are interspersed.

Hardware requirements for Fortran D include advanced programming instructions, editing instructions, and six peripheral devices consisting of four Type 204B magnetic tape units, a card reader or an additional tape unit, and a printer or an additional tape unit. The Type 214 or 223 card reader or the Type 227 card reader/punch may be used. Any printer with at least 120 print positions may be used. Up to nine optional peripheral devices may be added to the configuration. These include extra tape units and a card punch. Where extra tape units are available, one may be used to create a stack of compiled user programs.

Either three- or four-character addressing can be specified for use with Fortran D.

The first six sections of this manual give a detailed outline of the Fortran language used by the compiler. Programmers familiar with Fortran will recognize that the compiler language incorporates many of the features of the proposed Fortran standard specified by the American Standards Association and published March 10, 1965, by the X3.4/3 Committee. For those who wish only to review the differences, Appendix B contains a comparison of the languages and a language summary. Programmers who are relatively unfamiliar with Fortran will wish to review the language sections.

Sections VII and VIII contain system information for programmers. Section VII describes the control cards used in job input decks, while Section VIII is a system summary of the compiler, the diagnostic preprocessor, and the Screen conversion routine. The compiler is designed to operate primarily in a load-and-go mode in which a job of one or several programs is loaded, compiled, and executed, then the next job is processed, etc. Section VIII explains this concept and other possible modes of processing, in which compilation is bypassed in a given program or job, or in which a series of jobs can be written onto a run tape and the multijob tape can be executed at a later run (go-later mode). Also described are the use of the diagnostic preprocessor to check source program errors before compilation and the use of Screen to facilitate conversion of source programs written in Fortran II to the language of this compiler.

Section IX covers operating instructions for setting up and running the system. Also included is information on initial conversion of the Honeywell-supplied symbolic program tape of the system to a run tape.

Section X is a collection of helpful programming tips and techniques. The section contains information on language and system limitations, memory restrictions, and time- and space-saving techniques, and summarizes ways in which the Fortran Compiler D system and language may differ from the system and language used previously. The programmer, as he becomes familiar with the compiler, may add to this section from his experience.

Section XI describes three- and four-character addressing modes.

The appendixes contain information which supplements that contained in the various sections of the manual. Of primary importance is Appendix G which contains all error printouts — from the diagnostic preprocessor, from the compiler and the run tape generator, from object time execution, and from Screen.

TABLE OF CONTENTS

		Page
Section I	Source Program Summary	1-1
	Program Definitions	1-1
	Source Program Formatting	1-2
	Fortran Character Set	1-2
	Honeywell Character Set	1-2
	Blank Characters	1-2
	Statements	1-3
	Formatting Fortran Statements	1-3
	Statement Characteristics	1-5
	Labeling Statements	1-6
	Syntax	1-7
	Operators	1-7
	Delimiters	1-8
	Names	1-8
	Data Names	1-8
	Constants	1-9
	Variables	1-9
	Arrays, Array Elements, and Subscripting	1-10
	Data Types	1-11
	Integer Data	1-12
	Real Data	1-13
	Logical Data	1-14
	Hollerith, Octal and Alphabetic Data	1-14
	Hollerith Data	1-14
	Octal Data	1-15
	Alphabetic Data	1-15
Section II	Arithmetic and Logical Expressions and Statements	2-1
	Arithmetic Expressions	2-1
	Definition and Evaluation	2-1
	Hierarchy of Arithmetic Operations	2-2
	Arithmetic Statements	2-4
	Logical Expressions	2-4
	Definition	2-4
	Logical Operators	2-5
	Relational Operators	2-6
	Hierarchy of Logical Operations	2-6
	Logical Statements	2-6
Section III	Control Statements	3-1
	Unconditional Go To	3-1
	Computed Go To	3-2
	Assigned Go To and Assign Statements	3-2
	Arithmetic IF Statement	3-3
	Logical IF Statement	3-3
	DO Statement	3-4
	Call Statement	3-7
	Return Statement	3-8
	Call Chain Statement	3-8

TABLE OF CONTENTS (cont)

			Page
Section	III (cont)	Continue Statement	3-8
		Pause Statement	3-8
		Stop Statement	3-9
		End Statement	3-9
Section	IV	Specification Statements	4-1
		Dimension Statement	4-1
		Common Statement	4-2
		Equivalence Statement	4-5
		Data-Type Statements	4-7
		External Statement	4-8
		Title Statement	4-9
		Data Initialization Statement	4-9
		Additional List Pairs	4-11
		Implied DO Loops	4-11
		Nested Pairs of Implied DO Loops	4-12
Section	V	Input/Output Statements	5-1
		Read Statement	5-1
		Write Statement	5-3
		Input/Output Lists	5-5
		Simple Lists	5-5
		Short-List Notation for Input/Output of Entire Arrays	5-7
		Lists with Implied DO Loops	5-7
		I/O Lists Used with Binary Tape Input or Output	5-11
		FORMAT Statement	5-11
		General Form of the FORMAT Statement	5-11
		Contents of the Field Specification	5-15
		Conversion Codes	5-16
		Conversion Field Width	5-16
		Decimal Position Indicator	5-18
		Basic Field Specification for Integer Conversion	5-18
		Input	5-18
		Output	5-19
		Input for Conversion of All Real Data	5-21
		Output Conversion to Fixed-Point Decimals (Fw.d)	5-24
		Output Conversion to Explicit Exponent (Ew.d)	5-25
		Generalized Field Specification, Gw.d	5-27
		Basic Field Specification for Octal Conversion	5-28
		Input	5-28
		Output	5-29
		Basic Field Specification for Logical Conversion	5-29
		Input	5-29
		Output	5-30
		Basic Field Specification for Alphabetic Conversion	5-31
		Input	5-31
		Output	5-31
		Field Specification for Hollerith Data	5-31

Page

Section V (cont)

Output 5-33
Input 5-35
Field Specification for Blank Conversion 5-36
Input 5-36
Output 5-37
Carriage Control for Printer Output 5-37
Field — Repetition Constant 5-39
Repetition of Groups of Field Specifications 5-40
Scale Factor 5-41
Input 5-42
Output 5-42
Multiple-Record Forms 5-45
Reading in FORMAT at Object Time 5-51
Alternate Creation of Variable Formats 5-53
End File Statement 5-55
Rewind Statement 5-56
Backspace Statement 5-56

Section VI

Procedures 6-1
Categories of Procedures 6-1
Subprograms 6-1
Naming and Typing Procedures 6-1
Functions 6-2
Statement Functions 6-2
Function Subprograms 6-4
Library Functions 6-7
Subroutines 6-9
Special Subroutines 6-12
Test Subroutines for Simulated Hardware and Hardware Features 6-12
I/O Condition Test Subroutines 6-13
I/O Subroutine REREAD 6-13
Dynamic Dumping Subroutines 6-14
Exit-to-Monitor Subroutine 6-14

Section VII

System Control Cards 7-1
Run-Level and Job-Level Control Cards 7-1
Control Cards for Standard Operation 7-1
Non-Standard Operation and Options 7-2
*JOBID Card 7-2
Job Name Option 7-3
Memory Size 7-3
Floating-Point Precision 7-4
Integer Precision 7-4
Peripheral Device Assignments 7-4
SAVE Option 7-5
PUNCH Option 7-5
Listing Options 7-5

TABLE OF CONTENTS (cont)

		Page
Section VII (cont)	Tape Input	7-6
	Sample *JOBID Card with Options	7-6
	*SCREEN Card	7-7
	*DIAG Card	7-7
	*GET Card	7-8
	*BINARY and END Cards	7-9
	*CHAIN Card	7-9
	*ALTER Card	7-9
	*DATA Card and 1EOFΔ Card	7-10
	*ENDATA Card	7-10
	*DUMP Card	7-11
	Comment Cards	7-11
	Card-Image Tape Input	7-11
Section VIII	System Description	8-1
	System Summary	8-1
	System Modules	8-1
	Run Options	8-1
	Standard Fortran Processing — Load-and-Go Operation	8-2
	Chaining a Load-and-Go Job	8-4
	Go-Later — Batched Job Processing	8-4
	Writing a Go-Later Tape	8-4
	Executing a Go-Later Tape	8-6
	System Options	8-7
	Stack Tape	8-7
	Punch Option	8-7
	Jobs Containing *GET and *BINARY Program Units	8-7
	Common Input Device	8-8
	Common Output and Common Punch Device Options	8-8
	Bypassing Execution	8-9
	Diagram of System Options	8-9
	Debugging Aids	8-10
	Source-Program Listing	8-10
	Relocatable Memory Map	8-10
	Object Memory Map	8-11
	Machine-Code (Pseudo-Easycoder) Listing	8-12
	Error Diagnostics	8-15
	Memory Dumps	8-16
	Diagnostic Preprocessing	8-16
	Preprocess-Only Option — *DIAG	8-16
	Preprocessing to Tape — *DIAG, T	8-17
	Diagnostics	8-18
	Screen Conversion	8-19
Section IX	Operating Procedures	9-1
	Fortran Run Options	9-1
	Standard Console Call	9-1
	Equipment Requirements	9-2

TABLE OF CONTENTS (cont)

		Page
Section IX (cont)	Tape Loader-Monitors	9-2
	Starting Procedure	9-2
	Terminating a Run	9-3
	Console Call Options	9-3
	Coded Halts During Fortran Runs	9-6
	Unprogrammed Halts and Looping	9-6
	Load-and-Go Run	9-8
	Load-and-Go Equipment	9-8
	Stack Tape (TF)	9-9
	Allocation of Work Tapes	9-9
	Input Tape (T5)	9-10
	Output to Tape	9-10
	Screen Run	9-10
	Writing a Go-Later Tape	9-11
	Starting and Terminating a Write Go-Later Tape Run	9-12
	Emergency Restart Option	9-12
	Card and Tape Input Option	9-12
	Executing a Go-Later Tape	9-12
	Starting a Go-Later Execution Run	9-13
	Creating a Compiler System Tape (CST)	9-14
Section X	General Programming Considerations	10-1
	Language Limitations	10-1
	Source Program Size Limitations	10-2
	Source Table	10-2
	Token Table	10-2
	IEFN Table	10-3
	FORMAT Table	10-3
	Size of Program String	10-3
	Compiler Characteristics and Limitations	10-3
	Tips for Saving Space and Time	10-4
	Chaining	10-5
	I/O Programming Tips	10-7
	Conversion Techniques	10-8
Section XI	Three-Character and Four-Character Address Modes	11-1
Appendix A	Octal-Decimal Conversion Procedure	A-1
Appendix B	Language Summary	B-1
	Comparison with ASA Proposed Fortran	B-1
	Additional Statements	B-1
	Terminology	B-1
	Additional Language Features	B-1
	Restrictions	B-2
	Change in Parameter	B-2
	Glossary	B-12
Appendix C	Internal Representation of Numbers	C-1
	Bit Representation	C-1

TABLE OF CONTENTS (cont)

		Page
Appendix C (cont)	Fixed-Point Numbers	C-1
	Floating-Point Numbers	C-1
	Storage of Other Data	C-2
	Accuracy of Calculations	C-2
	Formats of Data in Memory at Object Time	C-2
	Real Data	C-2
	Integer Data	C-3
	Octal Data	C-4
	Hollerith and Alphabetic Data	C-4
	Logical Data	C-5
	Constants	C-5
	Variables	C-5
Appendix D	Procedures and Routines Supplied with the Compiler	D-1
	Library Function Errors at Execution Time	D-4
	Floating-Point Packages	D-4
	Object I/O Modules	D-5
	Library Functions	D-6
	SIN	D-6
	COS	D-7
	EXP	D-8
	SQRT	D-10
	TANH	D-12
	ALOG	D-13
	ALOG10	D-15
	ATAN	D-15
	ATAN2	D-18
Appendix E	Easycoder Symbolic Program Units	E-1
	Regionalization	E-1
	Region-Defining DSA Statements	E-2
	System Processing	E-2
	Region 3 Address Interpretation	E-3
	Relocation of Address Fields	E-4
	Relocation of Fortran Program Units	E-5
	Programming Procedures	E-5
	Calling Sequences	E-5
	Use of Common Storage	E-6
	Use of Communication	E-6
	Restrictions	E-7
	Sample Program	E-7
	Notes on the Four-Character Address Mode	E-9
Appendix F	Tape and Memory Layouts	F-1
	Symbolic Program Tape	F-1
	Compiler System Tape	F-1
	Binary Run Tapes	F-1
	Object Tapes	F-1
	BCD Tapes	F-1

TABLE OF CONTENTS (cont)

		Page
Appendix F (cont)	Binary Tapes	F-1
	Binary Program Tapes	F-1
	Stack Tape	F-2
	Common Input Tape	F-2
	Common Output Tape	F-2
	Memory Map (Compilation Time)	F-2
	Memory Map (Execution Time)	F-2
Appendix G	Error Messages	G-1
	Preprocessor Error Messages	G-1
	Errors in Specification Statements	G-1
	Errors in Arithmetic and Logical Expressions	G-2
	Errors in Construction and Use of Subscripts	G-2
	Statement Label Errors	G-3
	FORMAT Statement Errors	G-3
	Errors in Construction of Program Constants	G-4
	Control and I/O Statement Errors	G-5
	Errors in IF or CALL Statements or the Use of Functions or Subroutines	G-5
	Keypunch and Other Miscellaneous Errors	G-6
	Compiler Error Messages	G-7
	Compiler Monitor Error Messages	G-12
	Run-Tape Generator Error Messages	G-13
	Execution-Time Error Messages	G-13
	Library Function Error Messages	G-13
	Input/Output Error Messages	G-14
	Screen Error Messages	G-14
Appendix H	Series 200 Character Codes	H-1

LIST OF ILLUSTRATIONS

Figure 1-1.	The Fortran Coding Form	1-3
Figure 1-2.	Sequence of Program Statements	1-6
Figure 1-3.	One-Dimensional Array, Storage Sequence of Elements	1-10
Figure 1-4.	Two-Dimensional Array, Storage Sequence of Elements	1-10
Figure 3-1.	The DO Statement and Its Range	3-5
Figure 3-2.	Legal and Illegal Nesting of DO Loops	3-6
Figure 3-3.	Legal Transfers of Control	3-7
Figure 3-4.	Illegal Transfers of Control	3-7
Figure 4-1.	COMMON Statement for Three Labeled Blocks	4-4
Figure 4-2.	Communication Via Positional Correspondence	4-4
Figure 4-3.	Use of Dummy Array to Space Over Common Area	4-5
Figure 4-4.	Illegal Extension of Common Region	4-7

		Page
Figure 4-5.	Legal Extension of Common Region	4-7
Figure 4-6.	Right-Hand Portion of Array to be Initialized	4-13
Figure 5-1.	Example Highlighting Different Components of FORMAT Statement	5-14
Figure 5-2.	Data Fields and Field Widths	5-17
Figure 5-3.	Input of Integer Data	5-19
Figure 5-4.	Output of Integer Data	5-20
Figure 5-5.	Twenty-One Equivalent Ways of Keypunching an Exponent of Plus Two	5-21
Figure 5-6.	Input of Real Data	5-23
Figure 5-7.	Output of Real Data in Exponential Format	5-26
Figure 5-8.	Input of Octal Datum	5-29
Figure 5-9.	Input of Logical Data	5-30
Figure 5-10.	Input of Alphabetic Data	5-32
Figure 5-11.	Example of Output of Hollerith Data	5-34
Figure 5-12.	Use of Continuation Line with Hollerith Specification	5-35
Figure 5-13.	Sample Input Card	5-36
Figure 5-14.	Carriage Control in Single-Record FORMAT Statements	5-39
Figure 5-15.	Carriage Control in Multiple-Record FORMAT Statements	5-40
Figure 5-16.	Definition of a Unit Record	5-46
Figure 5-17.	Parenthesis Levels in a FORMAT Statement	5-48
Figure 5-18.	Rescanning a FORMAT Statement	5-49
Figure 5-19.	Handling Variations in Format at Object Time	5-54
Figure 6-1.	Categories of Procedures	6-1
Figure 7-1.	Input Deck for Standard Operation	7-1
Figure 8-1.	Standard Load-and-Go Flow Diagram	8-3
Figure 8-2.	Input Deck for Load-and-Go	8-3
Figure 8-3.	Job Divided into Two Chains of Program Units	8-4
Figure 8-4.	Sample Input Deck to Write Go-Later Tape	8-5
Figure 8-5.	Flow Diagram to Write Go-Later Multijob Tape	8-5
Figure 8-6.	Sample Input Deck to Execute Go-Later Tape	8-6
Figure 8-7.	Flow Diagram to Execute Go-Later Tape	8-6
Figure 8-8.	Job Containing Previously Compiled Program Units	8-8
Figure 8-9.	Load-and-Go Run With System Options	8-9
Figure 8-10.	Source-Program Listing	8-11
Figure 8-11.	Relocatable Memory Map	8-11
Figure 8-12.	Object Memory Map	8-12
Figure 8-13.	Pseudo-Easycoder Listing	8-13
Figure 8-14.	Input Deck for Diagnostic Preprocessing	8-17
Figure 8-15.	Flow Diagram for Diagnostic Preprocessing — Preprocess-Only Option	8-17
Figure 8-16.	Diagnostic Preprocessor Flow Diagram — Tape Option	8-18
Figure 8-17.	Diagnostic Preprocessor Listing	8-19
Figure 8-18.	Input Deck for Screen	8-20
Figure 8-19.	System Flow of Screen	8-20
Figure 8-20.	Screen Conversion of I/O Statements	8-21
Figure 8-21.	Screen Conversion of Library Function Names	8-21
Figure 8-22.	Listing of Card Input to Screen	8-23

LIST OF ILLUSTRATIONS (cont)

		Page
Figure 8-23.	Output Listing from Screen	8-24
Figure 9-1.	Operator Action in Unprogrammed Halt or Looping	9-7
Figure 9-2.	Minimum Equipment Configuration for Load-and-Go Operation	9-8
Figure 9-3.	Screen Equipment Configuration	9-11
Figure 9-4.	Minimum Equipment Configuration for Go-Later Execution	9-13
Figure 10-1.	Example of Chaining	10-6
Figure E-1.	Sample Regionalized Easycoder Program Unit	E-8
Figure F-1.	Compiler System Tape Organization	F-2
Figure F-2.	Organization of the Binary Run Tape	F-6
Figure F-3.	Organization of BCD Tapes	F-7
Figure F-4.	Organization of Binary Tapes	F-7
Figure F-5.	Memory Map (Compilation Time)	F-9
Figure F-6.	Memory Map (Execution Time)	F-10

LIST OF TABLES

Table 1-1.	Source Program Coding Format	1-4
Table 1-2.	Fortran Statement Categories	1-5
Table 1-3.	Relational Operators Defining Logical Relations	1-7
Table 1-4.	Logical Operators	1-7
Table 1-5.	Examples of Constants	1-9
Table 2-1.	Arithmetic Operation Symbols	2-1
Table 2-2.	Examples of Arithmetic Expressions	2-2
Table 2-3.	Logical Evaluation Using Logical Operators	2-5
Table 2-4.	Relational Operators	2-6
Table 3-1.	Control Statements	3-1
Table 4-1.	Specification Statements	4-1
Table 5-1.	I/O Statements	5-1
Table 5-2.	FORMAT Statement Summary	5-12
Table 5-3.	Contents of the Field Specification	5-15
Table 5-4.	Field Specification Formats	5-15
Table 5-5.	Conversion Codes	5-16
Table 5-6.	Scale Factor Shifting of Decimal Point	5-42
Table 5-7.	Effects of Scale Factor on Input Values (F Conversion)	5-43
Table 5-8.	Effects of Scale Factor on Output Values (F Conversion)	5-43
Table 5-9.	Effects of Scale Factor on Output Values (E Conversion)	5-44
Table 6-1.	Characteristics of Functions	6-2
Table 6-2.	Library Functions	6-7
Table 6-3.	Characteristics of the Subroutine Subprogram	6-11
Table 8-1.	System Modules	8-1
Table 9-1.	Console Call Options	9-4
Table 9-2.	Possible Halts During a Fortran Run	9-6
Table 9-3.	Equipment for Load-and-Go Operation	9-9

LIST OF TABLES (cont)

		Page
Table 9-4.	Equipment for Screen	9-11
Table 9-5.	Equipment to Execute Go-Later Jobs	9-13
Table 10-1.	Language Limitations	10-1
Table A-1.	Octal-Decimal Conversion Table	A-1
Table B-1.	Fortran Statement Summary	B-3
Table D-1.	Procedures and Execution Routines on the Compiler System Tape	D-1
Table D-2.	Library Error Conditions	D-4
Table E-1.	Determination of Address Lengths in Region 3	E-4

SECTION I
SOURCE PROGRAM SUMMARY

The purposes of this section are (1) to define program terms for Fortran Compiler D and to describe source program formatting, and (2) to define and describe the elements of syntax used to write the source programs.

PROGRAM DEFINITIONS

A main program is a sequence of properly formatted statements terminating with an END statement. It may optionally begin with a TITLE statement. A main program can be compiled independently, or it can be compiled in a chain of programs.

Subprograms are either functions or subroutines and begin with FUNCTION or SUBROUTINE statements. A subprogram is called by a main program or by another subprogram and must terminate with an END statement. A subprogram can be compiled independently.

Program (or program unit) is a general term that can refer to either a main program or a subprogram.

Fortran Compiler D uses an overlay technique called chaining to maximize the memory efficiency of object programs. Since every program in a job does not necessarily communicate with every other program, the entire job need not fit in memory at the same time. In fact, the job may be divided into independent segments which can each occupy all of memory at different times. Such a group of programs within a job is called a chain. Each chain is a separate memory load and is overlaid by the subsequent chain. A chain can call any other chain at any time. The common area of memory provides the necessary data communication between chains. A special control card, the Chain card, signals the beginning of the second and subsequent chains in the input deck. The name of a chain may be any digit or letter.

Chaining is the basic overlay technique used in Fortran Compiler D. A chain begins with a Chain card (*CHAIN, x), where x designates the particular chain. A chain is called with a CALL CHAINx statement, described in Section III. A chain terminates when either another Chain card or any control card that defines the end of the current job is encountered.

An executable program is a collection of statements, comment lines, and END statements that completely (except for input data values and their effects) describe a computing procedure.

A job is:

1. Compilation of one or more program units
2. Execution of an executable program
3. A combination of 1. and 2.

A job begins with a Job Identification card (*JOBID) and terminates when either another *JOBID or an *ENDRUN card is encountered.

A run consists of one or more consecutive jobs requiring no operator action. A run begins with a Console Call card and terminates with an *ENDRUN card.

SOURCE PROGRAM FORMATTING

Fortran Character Set

The characters used in Fortran statements are as follows:

Letters A through Z

Numbers 0 through 9

Ten special symbols:

=	Equals	(	Left parenthesis
+	Plus	)	Right parenthesis
-	Minus	,	Comma
*	Asterisk	.	Decimal point
/	Slash	Δ	Blank

Honeywell Character Set

The Honeywell character set consists of the Fortran character set and the following 17 special symbols, shown as they appear on the printer:

'	Apostrophe	C_r	Credit Sign
:	Colon	¢	Cents Sign
>	Greater Than	□	Box
&	Ampersand	■	Filled Box
;	Semicolon	≠	Not Equal
?	Question Mark	%	Percent Sign
#	Number Sign	!	Exclamation Point
"	Quotation Mark		
<	Less Than		
@	At Sign		

All characters of the Honeywell character set can be used in Hollerith constants. Appendix H shows the keypunching and machine codes for all Honeywell characters.

Blank Characters

Blank characters are used to improve the appearance and legibility of program statements.

Blanks are significant only in the following cases:

1. A blank in column 6 of a statement card indicates that the card contains the first line of the statement not a continuation line.
2. In Hollerith or alphabetic data, all blanks are literally reproduced.

Statements

FORMATTING FORTRAN STATEMENTS

The fundamental units of the Fortran program are statements. Statements are constructed from the basic syntactic elements of the Fortran language using the character set shown above. The basic Fortran syntactic elements - operators, delimiters, and names - are described later in Section I.

Statements are divided into lines, each line corresponding to a single punched card. The first line of a statement is called the initial line; any subsequent lines are called continuation lines. Each line consists of a string of 72 characters from the Honeywell character set. The character positions in a line correspond to the columns of a punched card and are numbered sequentially 1, 2, ..., 72 from left to right. Each statement begins on a new line and a new punched card; each initial line of a statement can be followed with up to nine continuation lines.

Shown in Figure 1-1 is the Fortran Coding Form indicating the coding fields. The body of the statement is written anywhere in columns 7 through 72. Column 7 of a continuation line thus follows column 72 of the preceding line. Columns 1 through 5 are reserved for a statement label when the programmer requires the label for cross-referencing with the program. Only the initial line of a statement can have a statement label. The label is a number, unique for the statement, that can be placed anywhere in columns 1 through 5. The terms statement label and statement number are used interchangeably throughout this manual.

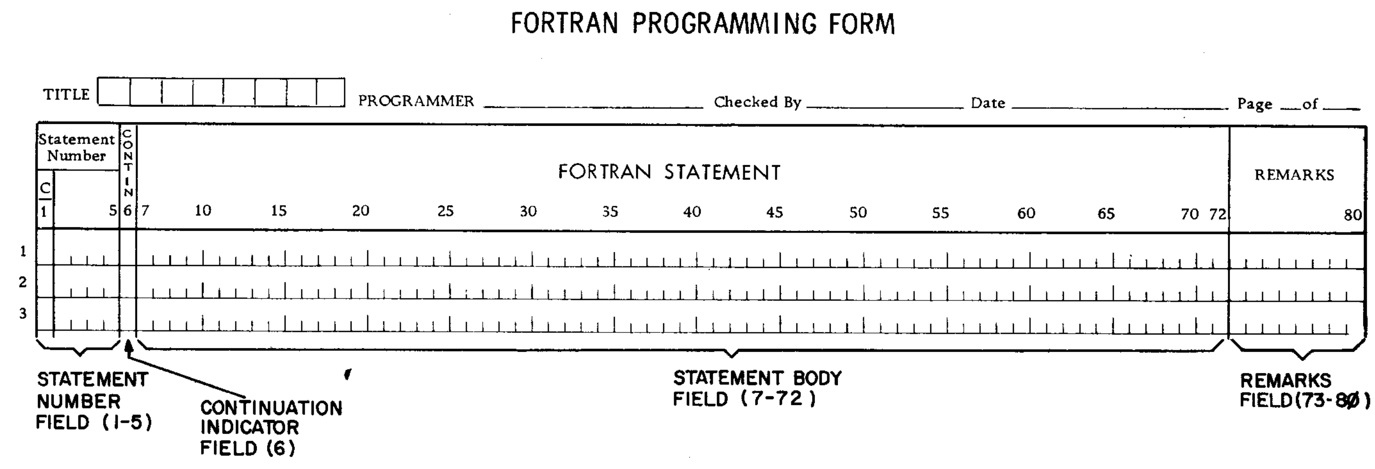


Figure 1-1. The Fortran Coding Form

The initial line of each statement must contain either a zero or a blank in column 6. Continuation lines of the statement can have any legal Fortran character other than zero or blank in column 6 to designate them as continuation cards. For example, continuation lines could be numbered 1 through 9.

The letter"C" in column 1 designates a line as a comment line. Comment lines are literally reproduced in the source program listing, but do not affect execution in any way; they are available solely for the convenience of the programmer in documenting the program. The body of the comment may appear anywhere in columns 2 through 80 of the comment card. Since columns 73 through 80 are ignored by the compiler but reproduced in the source program listing, these columns can be used for comments. When a comment requires more than one line, additional comment cards punched with a "C" in column 1 are used. (This is the only permissible method of indicating continuations of comments.) Comment lines may appear between lines of a statement.

The contents of a line of the source program are summarized in Table 1-1, below.

Table 1-1. Source Program Coding Format

Coding Sheet Column	Contents	Use
1-5	Statement Label	Used only with the initial line of a statement. A statement label is assigned by the programmer when the statement is cross-referenced in the program.
1	C	Used only to designate comment lines.
2-80	Comment	Comments are included for purposes of programmer documentation. They are nonexecutable.
6	Continuation field indicator (a Fortran character other than 0 or blank)	Indicates continuation of a statement. The initial line of a statement has a zero or blank in column 6. All subsequent lines of the statement must be indicated in column 6. Any Fortran character other than 0 or blank may be used.
7-72	Statement	Body of the statement, either executable or nonexecutable, and one of five general categories of statements. Statements are described in detail in sections following.

STATEMENT CHARACTERISTICS

Statements are divided into five categories according to their purpose and are characterized as either executable or nonexecutable. Table 1-2 identifies the five statement categories and indicates the section in this manual in which each statement is described.

Table 1-2. Fortran Statement Categories

Type of Statement	Purpose	Section	Possible Statements
Arithmetic or Logical	Specifies a numerical or a logical computation.	II	a = b
Control	Governs the flow of program execution: iteration, sequencing changes, etc.	III	ASSIGN CALL CALL CHAIN CONTINUE DO END GO TO (Unconditional, Computed, or Assigned) IF (Arithmetic or Logical) PAUSE RETURN STOP
Procedure	Enables the programmer to define and use subprograms.	VI	FUNCTION SUBROUTINE Statement Functions
Input/Output	Transfers data from or to a peripheral device; manipulates a peripheral device.	V	BACKSPACE END FILE FORMAT READ REWIND WRITE
Specification	Indicates necessary or desired information about the object program: memory requirements, data typing, etc.	IV	COMMON DATA (Initialization) Data-Type REAL INTEGER LOGICAL DIMENSION EQUIVALENCE EXTERNAL TITLE

Executable statements are those to which control may be transferred during the course of a program. The following are executable statements:

1. All arithmetic and logical statements.
2. All input/output statements except FORMAT.
3. All control statements.

All other statements are nonexecutable. In a source program, statements are sequenced so that nonexecutable statements, except FORMAT statements, precede the executable statements. Figure 1-2 shows the sequence of source program input statements.

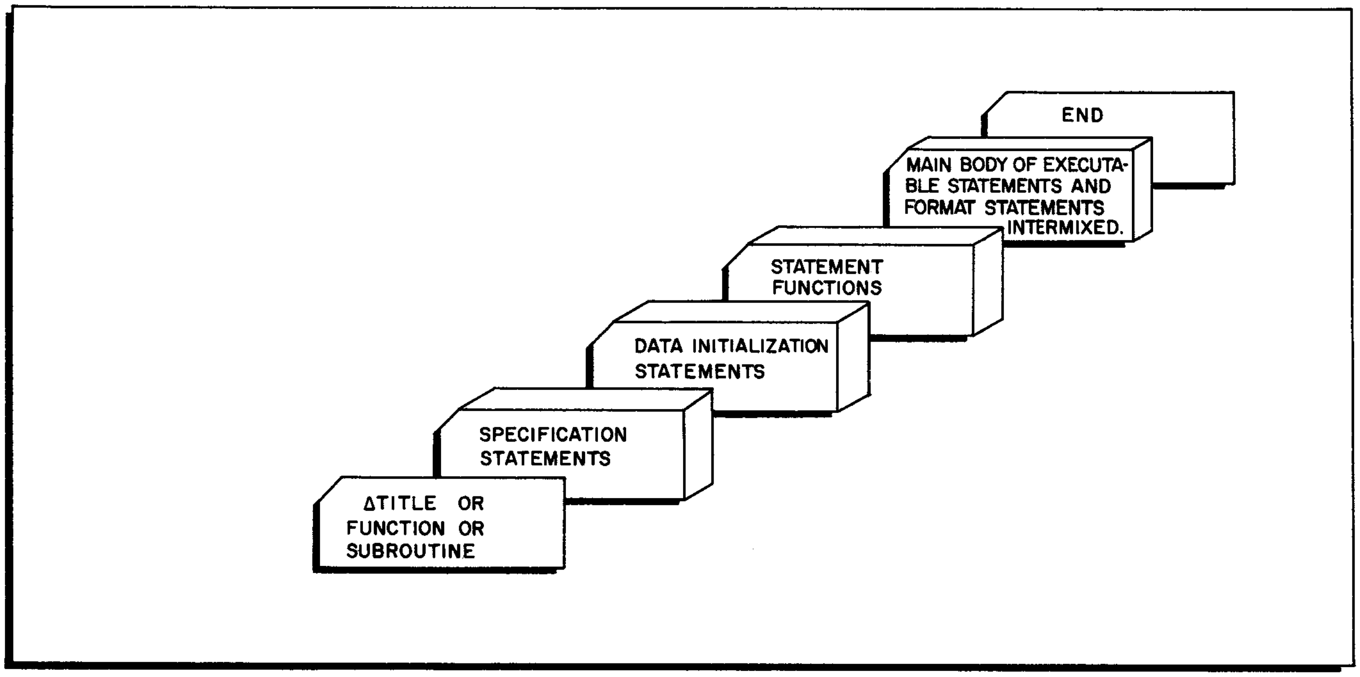

ΔTITLE OR FUNCTION OR SUBROUTINE
SPECIFICATION STATEMENTS
DATA INITIALIZATION STATEMENTS
STATEMENT FUNCTIONS
MAIN BODY OF EXECUTABLE STATEMENTS AND FORMAT STATEMENTS INTERMIXED.
END

Figure 1-2. Sequence of Program Statements

Statement operators are listed in capital letters under "Possible Statements" in Table 1-2. Statement operators begin all Fortran statements except arithmetic and logical statements and statement functions. The latter three types of statements are often called assignment statements. Statement operators cannot be divided between lines of a Fortran coding sheet.

LABELING STATEMENTS

The programmer labels with a number the initial lines of statements that he wishes to cross-reference in the program. A programmer has the option to label some statements; however, certain statements must be labeled and others cannot be labeled.

Every statement referenced by another statement must have a statement label. This includes all FORMAT statements and the last executable statement in the range of a DO loop. The first and last statements of a program cannot have statement numbers. This includes all TITLE, END, FUNCTION, and SUBROUTINE statements. Nonexecutable statements may have statement labels, but these will be ignored by the compiler in all cases except FORMAT statements. Only the initial line of a statement can be labeled.

The statement label assigned must be unique within that program and must be composed of numeric characters only. The numeric label can be placed anywhere in columns 1 through 5 of the coding form. However, to insure that no two labels in a program are the same, the programmer should remember that leading, embedded, and trailing blanks are ignored. Leading zeros are also ignored; embedded and trailing zeros are significant. Statement numbers can be assigned in any order, since their values do not imply sequencing. The permissible range of statement labels is 1 through 99999.

SYNTAX

The basic elements of the syntax of Fortran statements are operators, delimiters, and names.

Operators

Operators consist of statement operators, logical operators, and relational operators. Operators specify action to be taken on named elements.

Statement operators begin all statements except arithmetic and logical assignment statements and statement functions. Statement operators are listed under the column titled "Possible Statements," in capital letters in Table 1-2.

Relational and logical operators are written between named elements. The six relational and three logical operators are given in Tables 1-3 and 1-4.

Table 1-3. Relational Operators Defining Logical Relations

Relational Operator	Equivalent Mathematical Notation	Definition
.EQ.	$=$	Equal to
.GE.	$\geq$	Greater than or equal to
.GT.	$>$	Greater than
.LE.	$\leq$	Less than or equal to
.LT.	$<$	Less than
.NE.	$\neq$	Not equal to
The value of a logical relation is .TRUE. if satisfied, .FALSE. if not satisfied.		

Table 1-4. Logical Operators

Logical Operator	Equivalent Logical Notation	Definition
.NOT.	$\overline{}$ (Overscore)	Logical negation
.AND.	$\cap$	Logical AND
.OR.	$\cup$	Inclusive OR
The value of a logical expression resulting from use of a logical operator is either .TRUE. or .FALSE. Determination of the evaluation of logical expressions is given in Table 2-3.		

Delimiters

Delimiters are used to separate other statement elements and consist of the following eight symbolic Fortran characters:

+) - / , (= *

Names

Names identify or reference data or procedures. A data name identifies a constant, variable, array, or array element. These data categories are defined and illustrated later in Section I. A procedure name identifies a function or subroutine; procedure categories are defined and illustrated in Section VI.

A name is said to reference a datum if the current value of the datum will be made available during the execution of the statement containing the reference. A name may identify without referencing a datum. A name is said to reference a procedure if the actions specified by the procedure will be made available during execution of the statement containing the reference.

Data names can be connected by certain delimiters to form arithmetic expressions or by logical or relational operators to form logical expressions. Rules for forming logical and arithmetic expressions are given in Section II.

Associated with data and certain procedure names are data types. For Fortran Compiler D the allowable data types are integer, real, and logical. Data types are defined and illustrated later in Section I.

Certain names are predefined as names of procedures supplied by Honeywell with the compiler. The predefined procedures are library functions and are described in detail in Section VI.

The names of statement operators are reserved in this compiler. Statement operators can be used as variable or procedure names only in accordance with the rules given in Section X.

DATA NAMES

Names are used to identify or reference data that are classified as one of the following:

1. constants;
2. variables;
3. arrays; or
4. array elements.

Constants

A constant is a specific numerical value or a string of literal characters. It cannot vary during the computing process. A numerical constant can be signed or unsigned. The name of the constant is the same as the value of the constant. Thus in Table 1-5 below, the constant named 23 has a numerical value of 23. Additional examples of constants are given in the table. A data type is shown associated with each constant; data types are discussed later in Section I.

Table 1-5. Examples of Constants

Constant	Data Type
23	Integer (fixed point)
0	
-456	
+1275	
-71.42	Real (floating point)
8.06	
12.	
12.0E2	
12.0E-2	
3.E5	
.TRUE.	Logical (only two possible values as shown)
.FALSE.	

Variables

A variable, as defined in Fortran Compiler D, identifies a datum that can be altered during the computing process and is not subscripted. Subscripted variables are called array elements in this manual and are discussed below in the paragraph on "Arrays, Array Elements, and Subscripting."

A variable must consist of 1 to 6 alphanumeric characters, the first of which must be alphabetic.

In the statement

```
X = 12.7
```

X is a variable and 12.7 is a constant. It is possible that X may never be defined as anything but 12.7, but it is still a variable because it identifies a datum that could be altered during execution of the object program.

Data types are associated with variables. These data types and the rules governing them are discussed later in Section I with additional examples of variables.

Arrays, Array Elements, and Subscripting

An array is an ordered set of data of either one or two dimensions which may be referenced and/or altered during the computing process. A one-dimensional array corresponds to a vector (see Figure 1-3), and a two-dimensional array corresponds to a matrix (see Figure 1-4). Each member of an array is called an array element. A name is assigned to each of the elements, as well as to the entire array. Therefore, any single element may be referenced by name, or the entire ordered set may be referenced through use of the array name. An array name must consist of 1 to 6 alphanumeric characters, the first of which must be alphabetic.

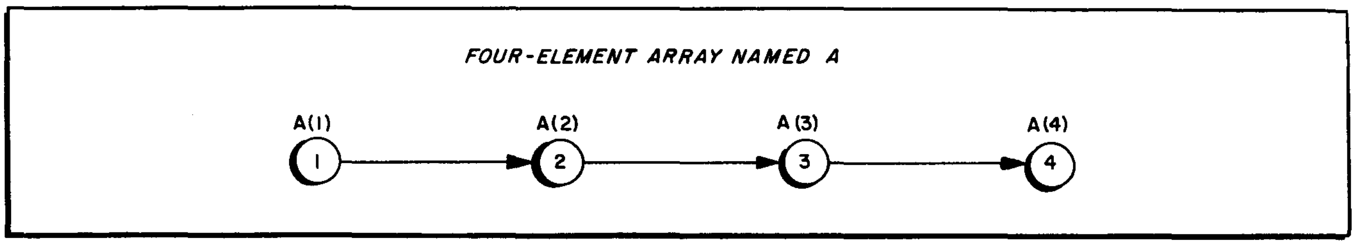


Figure 1-3. One-Dimensional Array, Storage Sequence of Elements

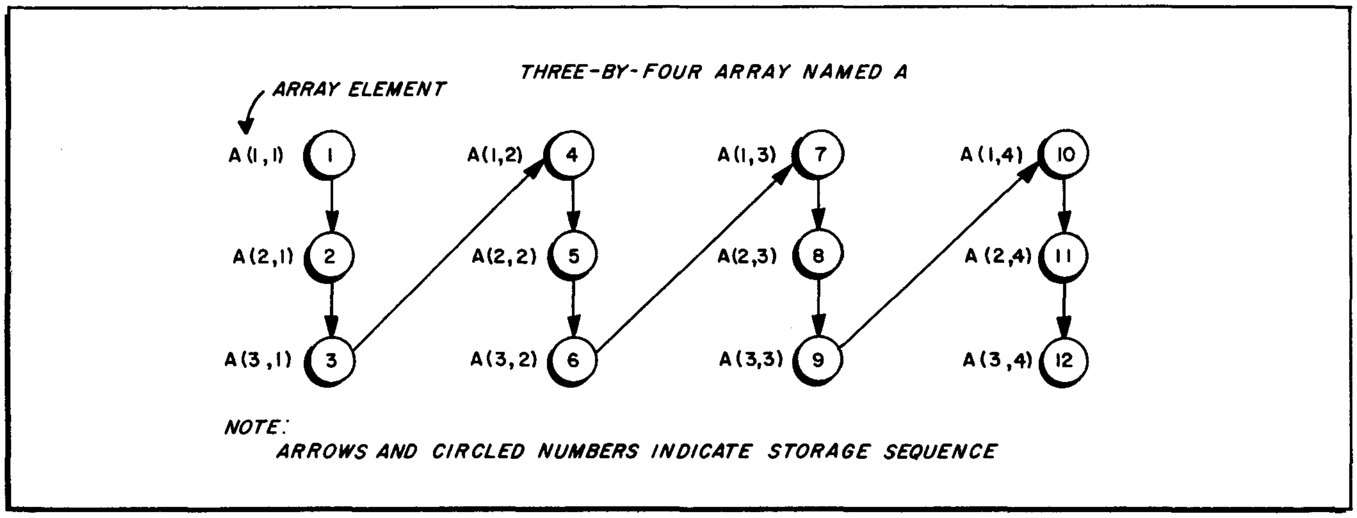


Figure 1-4. Two-Dimensional Array, Storage Sequence of Elements

The name of an element is formed by appending a qualifier, called a subscript, to the array name. The subscript indicates which element in an array is being referenced. When this array element notation is used, the number of subscripts must equal the dimension of the array. When an array is two-dimensional the two subscript expressions of each array element are separated by a comma. For example, in Figure 1-4, the element in the second row and third column of array A has the name A(2, 3). Array elements of two-dimensional arrays are stored sequentially in memory by columns as shown in Figure 1-4. Thus the first (leftmost) subscript expression of a two-dimensional array varies most rapidly and its last subscript expression varies least rapidly. An array element of a two-dimensional array can be identified not only by its double-subscripted name but also by its storage sequence. For example, in Figure 1-4, the element in the third row and third column can be called either A(3, 3) or A(9).

Within a subscript, each subscript expression may be in one of the following formats:

Format	Example
k	2
v	I
v+k	I+2
v-k	I-2
c*v	2*I
c*v+k	2*I+1
c*v-k	2*I-1

Where:	c and k	are unsigned integer constants;
	v	is an integer variable; and
	+ - *	are arithmetic expression delimiters for plus, minus, and multiply (See Section II)

In arithmetic statements, the array name can be used to reference the first element of the array - for example, A for A(1).

The data type associated with every array element subscript and with each subscript expression is integer (fixed point), as described below. Arrays and array elements may have integer, real, or logical data types associated with them.

DATA TYPES

Associated with data names and certain procedure names are their characteristic data types. The data types for Fortran Compiler D are integer, real, and logical; they can be associated with constants, variables, arrays, array elements, and functions.

The name of a constant indicates its type. Data types associated with variables, arrays, and array elements and with functions can be indicated in one of two ways:

1. The programmer can explicitly supply the data type by using a data-type statement, described in Section IV. A data-type statement begins with REAL, INTEGER, or LOGICAL and assigns the specified data type to the variables, arrays, or functions that follow. All logical variables, arrays, and functions must be declared in data-type statements.

2. Real or integer types of data can be implied in the name. Any name that begins with one of the characters I, J, K, L, M, or N is an integer unless otherwise specified in a data-type statement. A variable that begins with any other character is real unless otherwise specified in a data-type statement.

Each of the data types is defined and illustrated in the paragraphs following.

Integer Data

An integer is an exact whole number which can be positive, negative, or zero. Unsigned numbers are presumed to be positive. No decimal point is expressed; therefore, integer data are often referred to as fixed-point. Any fractions resulting from operations on integer data are truncated without rounding before additional operations are performed (see the last example of truncation below).

Precision of the integer data can be specified by the programmer on the *JOBID card in the source program input deck. The range of precision is from 3 to 12 characters, but because integer data is stored internally in binary, the number of decimal digits that can be stored is from 5 to 20. When the programmer does not specify precision on the *JOBID card, a three-character (five-digit) precision is used. Appendix C gives a detailed summary of precision and internal storage of data.

Examples of integer constants are:

0	42	4157	+12428	17592186414
-0	-42	-4157	-12428	-17592186414

Examples of the results of truncation of fractions are:

3/4 is equal to 0;

5/2 is equal to 2;

(8/3) + (9/2) is equal to 6

An integer variable is normally implied in the data name. Any variable beginning with I, J, K, L, M, or N, unless otherwise specified in a data-type statement, is assumed to be an integer variable.

Examples of integer variables are:

INTEG	K22	NUMBER
J	L65A9	IVALUE

Note that the name of a variable is prefixed by one of the integer characters when an integer variable is desired. Honeywell-supplied functions are prefixed in this manner. For example,

ABS and LABS

represent the function for determining absolute value, the first for real arguments and the second for integer arguments. See Section VI for other functions.

Real Data

A real datum is a real, rational value that need not be a whole number. A real constant is written with a decimal point or a decimal exponent or both. Real data are also known as floating-point data.

Real data may have values that are positive, negative, or zero and are stored, in memory as decimal fractions (mantissas) and decimal exponents. If a sign is not given, the number is assumed to be positive. When writing a real constant with an exponent, the letter E precedes the exponent and identifies it. The mantissa can be written with a precision between 2 and 20 decimal digits. The programmer can specify mantissa precision on the *JOBID card in the card input deck. If not specified there, precision is given to seven digits. The number of digits specified in a constant is truncated, or trailing zeros are supplied to store the precision specified by the mantissa parameter. The range of the exponent is fixed at two decimal digits, i.e., $-99 \leq e \leq +99$. See Appendix C for detailed examples of storage of numbers.

Examples of real constants are:

12.	12.0 E2 (12.0 x 10^2 or 1200.)
.127	12.0 E-2 (12.0 x 10^{-2} or 0.12)
235.7450	3.E70
12544761.1234	2E1
123456789.01234567890	

When the character following the letter E is zero, the next digit (if there is one) is considered to be the exponent, e.g.,

12.0 E02 is equivalent to: 12.0 E2 or 12.0 x 10^2

A zero exponent is permissible, but a blank character appearing in an exponent is suppressed and ignored.

Unless otherwise specified in a data-type statement, any variable beginning with a character other than I, J, K, L, M, or N is real.

Examples of real variables are:

VARIBL	A126
FTRAN4	X

Logical Data

A logical datum can assume a truth value of either true or false. Logical constants are either .TRUE. or .FALSE.

Hollerith, Octal and Alphabetic Data

Three kinds of data can be manipulated in Series 200 Fortran but cannot be defined in a data-type statement. These are alphabetic, Hollerith, and ocatal data.

Alphabetic and octal data can be manipulated as variables associated with certain conversion elements in input/output lists. This is discussed in Section V under "FORMAT statement." An alphabetic datum can be placed in memory only through an input operation.

Hollerith and octal constants can be assigned to variables at loading time by use of a DATA statement. This process is described under "DATA Initialization Statement" on page 4-9.

Hollerith, octal, and alphabetic data are handled internally as fixed-point data and must be associated with integer data types.

HOLLERITH DATA

A Hollerith datum is a constant that carries symbolic information rather than a value that is available for mathematical computation. Any character of the Honeywell character set may appear in a Hollerith datum, including letters, digits, and special symbols. Blanks are valid and significant; they are not suppressed. A Hollerith datum is stored in an integer data field.

A Hollerith constant is written as follows:

```
nHxxxxxx
```

where: n is the number of characters (including blanks) in the constant
H indicates Hollerith
each x represents a Hollerith character

Examples of Hollerith constants follow:

```
4HDATA
71HTHIS ENTIRE SENTENCE IS A HOLLERITH CONSTANT, CONTAINING 71
CHARACTERS.
40HCAN SPECIAL SYMBOLS BE USED?  DIGITS TOO?
30HYES, E.G., * ( & : ) # % , ETC.
3H429
27HB L A N K S ARE SIGNIFICANT
```

A Hollerith constant can be indicated as the initial value of a variable or array element through use of the DATA statement (see page 4-9). The datum can then be employed in algebraic comparisons or as arguments of functions and subroutines.

Hollerith data can be manipulated by use of the conversion specification wH of the FORMAT statement. See Section V for discussion of the field specification wH.

OCTAL DATA

An octal datum consists of any combination of numbers 0 through 7. For programmers not familiar with octal numbering, a decimal-octal conversion table is included as Appendix A. An octal datum is associated with an integer data item.

An octal constant is written as follows:

```
nOxxxxxx
```

where: n is the number of x characters in the constant

O indicates octal

each x represents an octal digit.

Some examples of octal constants are:

```
 4O3777
10O4567012345
 7O0123456
```

An octal constant can be indicated as the initial value of a variable through use of the DATA statement (see page 4-9). The datum can then be employed in comparisons or as arguments of functions and subroutines. All comparisons, including octal, are made algebraically, not bit-by-bit. For example, in the octal constants compared below

$$770000 < 000001$$

because 770000 is interpreted as a negative, twos-complement number.

ALPHABETIC DATA

Since data cannot be declared as Hollerith in a specification statement, there are no variable character strings as such. As indicated in the discussion of Hollerith constants, however, a DATA initialization statement can be used to assign a Hollerith constant to an integer variable, which can then be employed in comparisons or as an argument of a function or subroutine.

Character strings can be placed in memory through input operations and stored in integer variable fields. When a character string is stored in this manner, it is called alphabetic data and can be manipulated as variables. An Aw field specification in a FORMAT statement is associated with an integer variable in an I/O list to read an alphabetic variable into or out of memory, as described in Section V.

SECTION II

ARITHMETIC AND LOGICAL EXPRESSIONS AND STATEMENTS

ARITHMETIC EXPRESSIONS

Definition and Evaluation

Arithmetic expressions consist of combinations of constants, variables, array elements, and/or function references, separated by the arithmetic operation symbols listed in Table 2-1.

Table 2-1. Arithmetic Operation Symbols

Operation Symbol	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

Arithmetic expressions are evaluated during execution of the object program. Evaluation consists of performing arithmetic on the constants, variables, array elements and/or functions in the expression. The single numeric value which results may be different each time the expression is evaluated since the value of any named element, except constants, may change between evaluations.

During evaluation, all values are invariant. The value of a variable or array element is the value last assigned before evaluation. The value of an array element is, of course, determined after evaluation of the subscript. The value of a function in an expression depends on the values of the constants, variables, and/or array elements that are its arguments, since the function is computed after evaluation of its arguments. Precedence rules for evaluation of arithmetic expressions are given later in this section.

The following rules govern arithmetic expressions:

1. An arithmetic expression must not contain logical data.
2. An arithmetic expression must not contain mixed data types, except that:
   a. A real datum can be raised to an integer exponent.
   b. In general, functions may have arguments of any type. (This does not hold for library functions. See Section VI.)
3. One arithmetic operation symbol cannot immediately follow another arithmetic operation symbol.

4. Parentheses may enclose any arithmetic expression.
5. A plus or minus sign may precede any arithmetic expression.
6. A negative mantissa cannot be raised to a real exponent.

Examples of Fortran arithmetic expressions are given in Table 2-1 with their mathematical representation and data names and types.

Table 2-2. Examples of Arithmetic Expressions

Fortran Expression	Mathematical Expression	Data Name and Type
2 + 3	$2 + 3$	Integer constants
I - J	$I - J$	Integer variables
A / B	$\frac{A}{B}$ or $A \div B$	Real variables
2. * X	$2X$ or $2(X)$ or $2 \cdot X$	Real variable
X ** 2	X^2	Real variable
Y (1) + Y (2) - Y(3)	$Y_1 + Y_2 - Y_3$	Real array elements
B**2 - 4. * A * C	$B^2 - 4AC$	Real variables
2. E4 * A * B ** (4+K)	$20,000AB^{4+K}$	Real variables
RFUN (X)	RFUN (x)	Real function reference

Hierarchy of Arithmetic Operations

During evaluation of an expression, arithmetic operations are performed one at a time, according to the following rules, and as described at the beginning of this section.

RULE 1. In the absence of parentheses specifying the exact order of evaluation, the priority of operations is:

1. Exponentiation;
2. Multiplication and division; and
3. Addition and subtraction.

Examples:

1. The expression A - B * C is evaluated as though it were written:

$$A - (B \cdot C)$$

That is, the product of B * C is evaluated first, then subtracted from A (because multiplication is at a higher level in the hierarchy of operations than subtraction).

2. The expression A + B/C**2 is evaluated as though it were written:

$$A + \frac{B}{C^2}$$

That is, C**2 is computed first; this value is divided into B, and the resulting quotient is added to A. (Exponentiation, followed by division, followed by addition.)

RULE 2. Where precedence is not otherwise indicated, evaluation of the expression proceeds from left to right.

Example:

In the following example, both division and multiplication are at the same level in the hierarchy of operations, and precedence is not otherwise indicated; therefore, evaluation proceeds from left to right:

A / B * C is interpreted as $\frac{A}{B} \cdot C$, not $\frac{A}{B \cdot C}$.

RULE 3. Parentheses may be used to alter the hierarchy of operations, since expressions in parentheses are always evaluated separately (regardless of the evaluation sequence otherwise implied by the hierarchy of operations).

Examples:

1. A / (B * C) is interpreted as $\frac{A}{B \cdot C}$. Without the parentheses, the interpretation would be as shown in the preceding example.
2. In the expression (A + B) * C, the use of parentheses overrides the normal hierarchy of operations. Therefore, the addition is performed before the multiplication, even though multiplication is at a higher level in the hierarchy of operations.

RULE 4. In a nest of parentheses (i.e., one pair of parentheses within another), the expression within the innermost pair of parentheses is evaluated first, then the one within the next innermost pair, etc. The expression within the outermost pair is evaluated last.

Examples:

1. In the expression:

   A * (B - (C / (D + E)))

   the sequence of evaluation is:

   a. D + E is computed.
   b. The sum of D + E is divided into C.
   c. The quotient of C/(D + E) is subtracted from B.
   d. The result is multiplied by A.

2. The expression 5.*(3.**APW+SQRT(A**2))/4.*(B**ABS(X)) is evaluated in the following sequence:

   a. A**2
   b. SQRT(A**2)
   c. 3.**APW
   d. 3.**APW+SQRT(A**2)
   e. ABS (X)
   f. B**ABS(X)
   g. 5.*(3.**APW+SQRT(A**2))
   h. 5.*(3.**APW+SQRT(A**2))/4.
   i. 5.*(3.**APW+SQRT(A**2))/4.*B**ABS(X)

ARITHMETIC STATEMENTS

An arithmetic statement has the form:

$\boxed{a = b}$

Where: a is a variable or array element of the real or integer type;
= means "is assigned the value"; and
b is a real or integer arithmetic expression.

The variable to the left of the equals sign determines the form of the result. If the variable on the left is integer and the expression on the right is real, the result is computed as a real value, truncated, and converted to an integer value. If the variable on the left is real and the expression on the right is integer, the result is computed as an integer value and converted to a real value.

Following are examples of arithmetic statements:

A = B	Store the value of B in A.
I = B	Truncate B to an integer and store in I.
A = I	Convert I to a real value and store in A.
A = 1.0	Store 1.0 in A.
JINDEX(1) = 2	Store 2 in integer array element JINDEX(1).
JINDEX = A	Convert real variable A to an integer value by truncating and store in JINDEX.
X = 12. *Y + (Z - 2.)**2	Using real arithmetic, subtract 2. from Z; square the result and add it to the product of 12. times Y; then store the final result in X.
N = N/M	Using integer arithmetic, divide N by M, and store the result of the division (i.e., quotient) in N.

LOGICAL EXPRESSIONS

Definition

When evaluated, a logical expression produces the value .TRUE. or .FALSE.. A logical expression may take any of the forms listed below:

1. A logical expression may be one of the following single named elements:
   a. A logical constant (i.e., .TRUE. or .FALSE.); or
   b. A logical variable; or
   c. A logical array element; or
   d. A reference to a logical function (i.e., a function that delivers a logical result).

2. A logical expression may be composed of any sequence of logical constants, logical variables, logical array elements, and/or references to logical functions, provided that each named element is separated from another by one of the logical operators defined below.

3. A logical expression may be composed of pairs of arithmetic expressions separated from one another by one of the relational operators defined below, provided that the type of the expression on each side of the relational operator is the same. Each pair is a logical expression that may be combined as any other logical expression described above.

Logical Operators

Logical operators are defined as follows:

Operator	Definition
.NOT.	Logical negation.
.AND.	Logical AND.
.OR.	Inclusive OR.

In combining the logical operators .NOT., .AND., and .OR. with logical expressions, the resulting logical expressions have values of either .TRUE. or .FALSE. as defined in Table 2-3. Letters a and b represent logical expressions.

Table 2-3. Logical Evaluation Using Logical Operators

Logical Expression	Value of a	Value of b	Value of Resulting Expression
.NOT.a	.TRUE.	---	.FALSE.
	.FALSE.	---	.TRUE.
a.AND.b	.TRUE.	.TRUE.	.TRUE.
	.TRUE.	.FALSE.	.FALSE.
	.FALSE.	.TRUE.	.FALSE.
	.FALSE.	.FALSE.	.FALSE.
a.OR.b	.TRUE.	.TRUE.	.TRUE.
	.TRUE.	.FALSE.	.TRUE.
	.FALSE.	.TRUE.	.TRUE.
	.FALSE.	.FALSE.	.FALSE.

In using logical operators the following rules must be observed:

1. Two logical operators must not be adjacent to each other unless the second operator is .NOT..

2. A period, as shown, is included at the beginning and end of each logical operator.

3. If the logical operator .NOT. is to apply to an expression following it that includes more than one named element, the required expression must be enclosed in parentheses. If the expression is not enclosed within parentheses, the .NOT. applies only to the first element of the expression.

Relational Operators

Logical relations are defined by using the relational operators given in Table 2-4.

Table 2-4. Relational Operators

Relational Operator	Equivalent Mathematical Notation	Definition
.EQ.	$=$	Equal to
.GE.	$\geq$	Greater than or equal to
.GT.	$>$	Greater than
.LE.	$\leq$	Less than or equal to
.LT.	$<$	Less than
.NE.	$\neq$	Not equal to
The value of a logical relation is .TRUE. if satisfied, .FALSE. if not satisfied.		

Hierarchy of Logical Operations

All arithmetic expressions are evaluated before logical expressions. The complete hierarchy of logical and arithmetic operations is as follows:

1. Arithmetic expressions are evaluated in the order given on page 2-2.
2. Any logical relations (.EQ., .GE., .GT., .LE., .LT., or .NE.) are evaluated from left to right in the expression.
3. .NOT.
4. .AND.
5. .OR.

As with arithmetic expressions, parentheses may be used to specify the hierarchy of logical expressions.

LOGICAL STATEMENTS

A logical statement has the form:

$\boxed{a = b}$

Where: a is a logical variable or logical array element;
= means "is assigned the value"; and
b is a logical expression.

The logical expression is evaluated and the previous value of the logical variable or array element on the left of the equals sign is assigned the truth value .TRUE. or .FALSE. For example:

	Logical Statement	Interpretation
1.	A = .FALSE.	Logical constant .FALSE. is stored in A.
2.	B = X.LE.5.	If X is less than or equal to 5., B has the value .TRUE.; otherwise, B has the value .FALSE.
3.	C = X.GT.5..OR.Y.LT.Z	Determine a value of .TRUE. or .FALSE. for the relation X.GT.5.. (It is .TRUE. if X is greater than 5., .FALSE. otherwise.) Determine a value of .TRUE. or .FALSE. for the relation Y.LT.Z. (It is .TRUE. if Y is less than Z, .FALSE. otherwise.) If the value of either relation is .TRUE., store .TRUE. in C; otherwise, store .FALSE. in C.
4.	A(1) = .NOT.(X.EQ.50./Y**2)	If X equals 50. divided by Y^2, store the value .FALSE. in logical array element A(1); otherwise, store the value .TRUE. in A(1).
5.	B = X.AND..NOT.Y	If Y is .FALSE. and X is .TRUE., store the value .TRUE. in B; otherwise, store .FALSE. in B.
6.	D = X.GT. (50.*Y + W/(X-2.))	The arithmetic expression is evaluated in the conventional manner (i.e., the expression in the innermost parentheses is evaluated first, then the expression in the outermost pair). If X is greater than the final result, the value .TRUE. is stored in D; otherwise, .FALSE. is stored in D.

The character "." may appear twice in succession in a logical statement under the following conditions:

1. When one period (or decimal point) is part of a constant and the other is part of a logical or relational operator. See example 3 above.
2. When a first logical operator is followed by .NOT.. Refer to example 5 above.

SECTION III
CONTROL STATEMENTS

Fourteen control statements govern the flow of control during execution of the program. Thus, control statements can be used to depart from the normal sequence of statements in the program, making it possible to bring in new sets of data or to carry out an iterative process. The fourteen control statements are listed in Table 3-1. All control statements are executable.

Table 3-1. Control Statements

Control Statement	Pages Where Defined
ASSIGN	3-2
CALL	3-7
CALL CHAIN	3-8
CONTINUE	3-8
DO	3-4 to 3-7
END	3-9
GO TO (unconditional)	3-1
GO TO (computed)	3-2
GO TO (assigned)	3-2
IF (arithmetic)	3-3
IF (logical)	3-3
PAUSE	3-8
RETURN	3-8
STOP	3-9

UNCONDITIONAL GO TO

An unconditional GO TO statement has the form:

```
GO TO n
```

Where: n is the statement label of an executable statement in the same program unit as the GO TO statement. The Unconditional GO TO statement transfers control directly to the statement labeled n.

Example: GO TO 17

When the statement labeled 17 has been executed, control is transferred in the normal sequence to the next executable statement following 17, unless the statement labeled 17 again changes the control sequence.

COMPUTED GO TO

A computed GO TO statement has the form:

```
GO TO (n₁, n₂,..., nₘ), i
```

Where: $n_1, n_2, \ldots, n_m$ are labels of statements in the same program unit as the GO TO statement, and i is the integer variable that must take on values in the range: $1 \leq i \leq 63$. Therefore, the maximum value of m is 63.

The computed GO TO statement transfers control to the statement whose label is ith from the left parenthesis.

Example: GO TO (20, 50, 75, 1066), I

Interpretations: If I=1, GO TO 20
If I=2, GO TO 50
If I=3, GO TO 75
If I=4, GO TO 1066

ASSIGNED GO TO AND ASSIGN STATEMENTS

An ASSIGN statement is used in conjunction with an assigned GO TO statement within the same program body. The ASSIGN and assigned GO TO statements are of the form:

```
ASSIGN n TO i
. . . . . . . .
. . . . . . . .
. . . . . . . .
GO TO i, (n₁, n₂ ,..., nₘ)
```

Where: n is a label of a statement in the same program unit as the ASSIGN statement.

i is an integer variable (the same in both statements).

$n_1, n_2, \ldots, n_m$ are all of the possible statement labels which n may assume. $m > 0$.

The ASSIGN statement assigns a statement label to the integer variable i. This label is one of the possible statement labels listed in the assigned GO TO statement.

The assigned GO TO statement then transfers control to the statement whose label has been assigned to the variable i. The list of statement labels within parentheses in the assigned GO TO statement should contain all of the possible statement labels which can be assigned to integer variable i. Commas are significant and must be punched before the opening parenthesis and between statement labels within the parentheses. The integer variable should not be used for computation until it is assigned a numerical value, thereby replacing the statement label value.

Example of assigned GO TO statement:

```
                    ASSIGN 375 TO J
         Same       ................
         Program    ................
                    ................
                    GO TO J, (210, 250, 375, 48, 56)
```

When the assigned GO TO statement is encountered, control is transferred to statement 375.

When using 4-character addressing, integer precision of 4 characters or more is required to execute an assigned GO TO statement.

ARITHMETIC IF STATEMENT

An arithmetic IF statement has the form:

$$\boxed{\text{IF(e) } n_1, n_2, n_3}$$

Where: e is an arithmetic expression of the integer or real type; n_1, n_2, n_3 are statement labels of executable statements in the same program unit as the IF statement.

The arithmetic IF statement transfers control to one of the three statements listed, depending on whether the arithmetic expression e is evaluated to be negative, zero, or positive. If the result of the evaluation is negative, control is transferred to the statement labeled n_1; if zero to n_2; and if positive to n_3. All three statement labels must be listed.

Example: IF (X**3 - 27.) 210, 425, 215

Branch to statement 210 if $X^3 < 27$, i.e., $(X^3 - 27.)$ is negative;

Branch to statement 425 if $X^3 = 27$, i.e., $(X^3 - 27.) = 0$;

Branch to statement 215 if $X^3 > 27$, i.e., $(X^3 - 27.)$ is positive.

LOGICAL IF STATEMENT

A logical IF statement has the form:

$$\boxed{\text{IF (e) s}}$$

Where: e is a logical expression.

s is any executable statement, except a DO statement or another logical IF statement.

Statement s is executed only if the value of the logical expression e is .TRUE. If the value of e is .FALSE., the logical IF statement is executed as if it were a CONTINUE statement.

Examples of logical IF statement:

1. IF (A.LE.B) GO TO 43

   If A is algebraically less than or equal to B, execute the statement labeled 43 next.

2. IF (A .AND. B) CALL BOTH

   If A is TRUE and if B is also TRUE, call subroutine BOTH.

3. IF (L) X = SIN(X)

   If the value of L is TRUE, replace X by SIN(X).

DO STATEMENT

A DO statement has one of the following forms:

```
        DO n i = m1, m2, m3
or:     DO n i = m1, m2
```

Where:	n	is the statement label of the last statement in the sequence of instructions to be executed repeatedly as a loop. That statement must be executable.
	i	is an integer variable.
	m_1	is the initial value of i; it may be an unsigned integer or integer variable.
	m_2	is the terminal value of i; it may be an unsigned integer or integer variable.
	m_3	is the amount by which i is to be incremented at the end of each pass through the loop; it may be an unsigned integer or integer variable. If m_3 is not stated, it is understood to be one.

A DO statement is placed at the beginning of a sequence of statements that are to be executed repeatedly as a loop. It defines the starting point of each complete repetition of the loop, the end point, and the number of times the loop is to be repeated. The starting point of the loop is the first executable statement after the DO statement; the termination point is that statement whose statement label (n) appears in the DO statement. All intervening instructions between the DO statement and terminal statement n (including the terminal statement) are executed in sequence each time that control passes through the loop. The parameters (m_1, m_2, m_3) of the control variable (i) control the number of repetitions of the loop. Both the initial value (m_1) of the control variable and the terminal value (m_2) are always specified in the DO statement. The amount by which the control variable is to be incremented (m_3) after each execution of the loop may or may not be specified in the DO statement. If m_3 is not explicitly defined, it is assumed to have a value of one. During execution of the DO statement, m_1, m_2, and m_3 must be greater than zero.

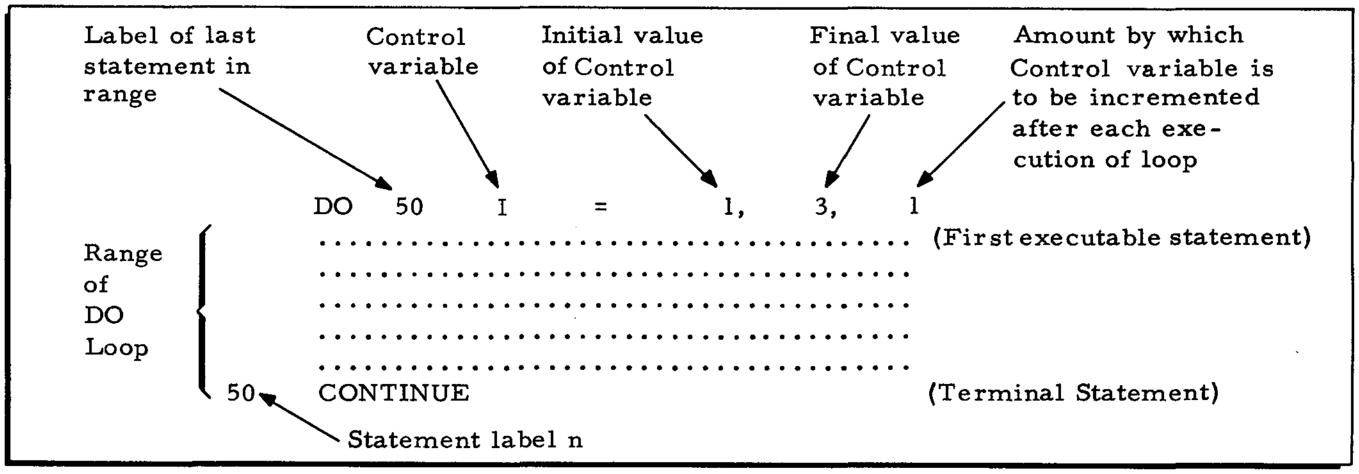
The action of a DO statement is as follows:

1. It initializes the control variable (i) with the value of the first (leftmost) DO parameter (m_1).
2. It executes the set of statements up to and including the terminal statement n.

3. After executing the terminal statement, it increments the control variable by the third DO parameter (m_3), or by a value of one if no third parameter is specified.

4. It compares the incremented value of the control variable with the value of the second DO parameter (m_2). If the incremented value is less than or equal to the terminal value, steps 2, 3, and 4 of this description are repeated; if the incremented value is greater than the terminal value, the DO loop is said to be satisfied, and control is transferred to the next statement in the program sequence following the terminal statement of the DO loop.

The range of a DO loop is the sequence of statements starting with the first executable statement after the DO statement and continuing up to (and including) statement number n, the terminal statement. The terminal statement must occur physically after the DO statement, not just logically after it. Furthermore, the terminal statement may not be any type of GO TO statement, or an arithmetic IF, DO, RETURN, or STOP statement. Figure 3-1 shows an example of a DO statement and its range.



```
       DO   50    I    =    1,   3,   1
       ...................................  (First executable statement)
       ...................................
       ...................................
       ...................................
       ...................................
   50  CONTINUE                             (Terminal Statement)
```

Label of last statement in range
Control variable
Initial value of Control variable
Final value of Control variable
Amount by which Control variable is to be incremented after each execution of loop
Range of DO Loop
Statement label n

Figure 3-1. The DO Statement and Its Range

Neither the control variable nor the DO parameters (m_1, m_2, m_3) may be altered by program statements during the execution of the loop. However, the value of the control variable is available throughout the range of the DO statement for use in computations, e.g., in subscripted expressions, and for referencing. Its value is that which was last assigned during execution of the loop. The value remains constant throughout the range until incremented at the end of the range. After the DO loop has been satisfied and control has been transferred out of the range of the DO statement, the control variable i is still available for use in any operation. Unless deliberately changed, it remains equal to the last incremented value, i.e., the value which exceeded m_2 and thus caused control to be transferred out of the DO loop.

One DO loop may contain within it one or more DO loops, provided that each inner DO loop is completely contained within the range of the outer loop. The use of one DO loop within another is called nesting of DO loops. DO loops may be nested up to a depth of 10. Examples of permissible nesting are illustrated in Figure 3-2, using the bracket symbol to represent a DO loop. Note that two loops may end on the same terminal statement (example C in Figure 3-2). The ranges of two loops cannot overlap in the way shown in Figure 3-2 example E, i.e., the one that starts last must be satisfied first. DO loops are said to be completely nested when there is only one inner loop within the next outer loop. If two or more inner loops are within the same outer loop, as in example B of Figure 3-2, the entire nest is noncompletely nested.

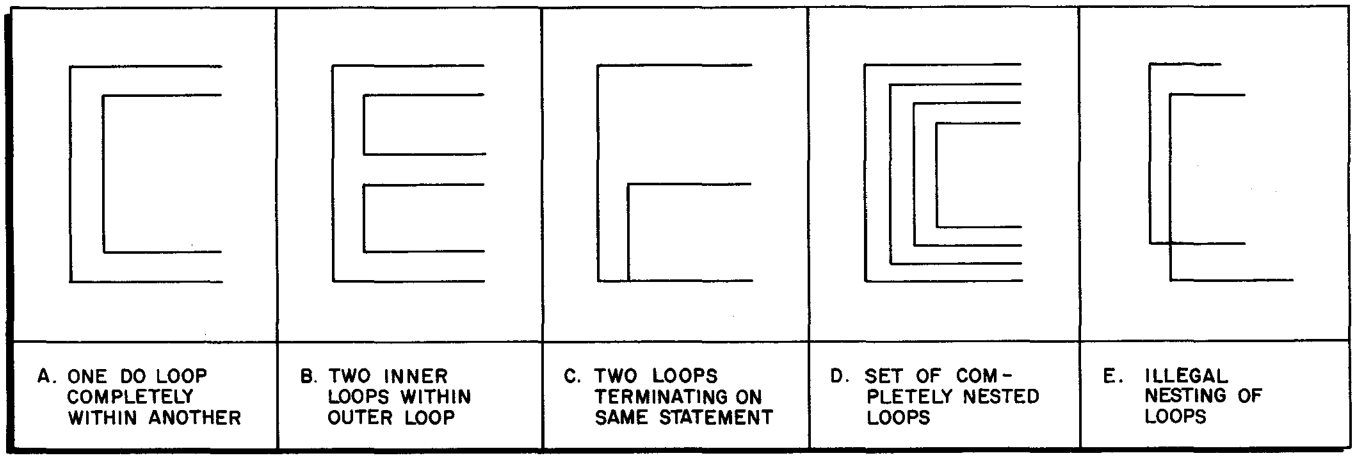

A. ONE DO LOOP COMPLETELY WITHIN ANOTHER
B. TWO INNER LOOPS WITHIN OUTER LOOP
C. TWO LOOPS TERMINATING ON SAME STATEMENT
D. SET OF COMPLETELY NESTED LOOPS
E. ILLEGAL NESTING OF LOOPS

Figure 3-2. Legal and Illegal Nesting of DO Loops

The statements in the first DO loop of Figure 3-2 are executed in sequence starting with the first executable statement and continuing up to the DO statement at the beginning of the inner loop. At this time, control is transferred to the inner loop, and the statements in this loop are executed repeatedly in sequence as many times as necessary until the loop is satisfied. Then control returns to the outer loop (specifically, to the first executable statement after the terminal statement of the inner DO loop). The balance of statements in the outer DO loop are executed until one complete iteration of the outer loop has been performed. At this point, the whole cycle repeats, exactly as described above until the outer DO loop is also satisfied. Then control is transferred to the next executable statement following the terminal statement of the outer DO loop.

It is permissible to transfer control freely within a DO loop; it is also permissible to transfer control out of the range of a DO loop or to another DO statement. Examples of legal transfers are shown graphically in Figure 3-3. It is not legal, however, to transfer into the range of a DO loop from outside its range. Figure 3-4 shows illegal transfers of control. Control may be transferred outside the range of a DO loop by calling a subroutine, function subprogram, library function, or statement function. Extended ranges whereby control can be passed out of and back to a DO loop through arithmetic IF and GO TO statements are not permitted.

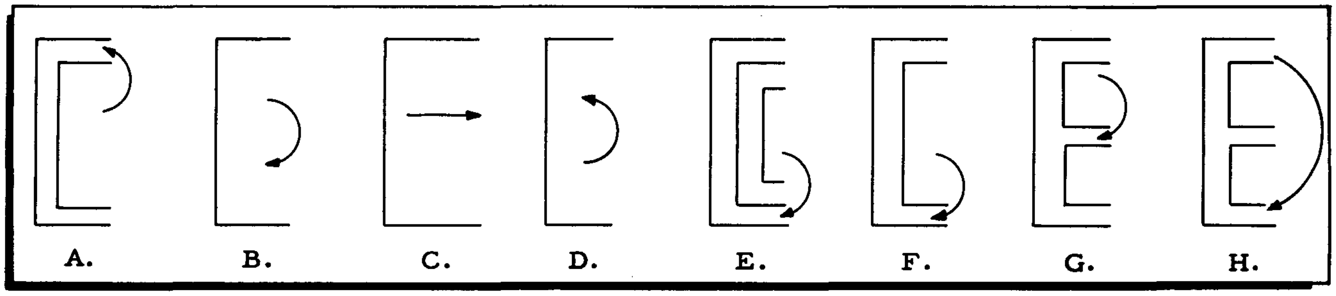


Figure 3-3. Legal Transfers of Control

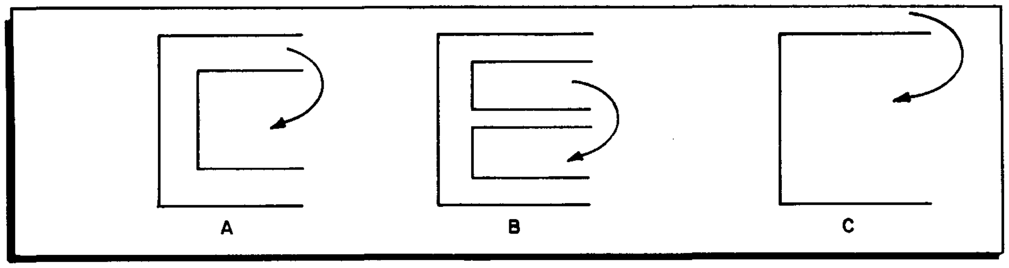


Figure 3-4. Illegal Transfers of Control

CALL STATEMENT

The CALL statement transfers control to a designated subroutine; it is the only mechanism available for transferring control to a subroutine. When the subroutine returns control to the calling program, the first executable statement following the CALL statement is the next statement to be executed. If the CALL statement is the terminal statement of a DO loop and the DO loop has not been satisfied, control returns to the first executable statement in the DO loop.

The CALL statement may be written with or without arguments.

The two general forms are:

1. CALL subnam (arg_1, arg_2, ..., arg_n)
2. CALL subnam

Where:	subnam	is the subroutine name.
	(arg_1, arg_2, ..., arg_n)	is the list of actual arguments which are to replace the dummy arguments in the SUBROUTINE statement at the time the subroutine is entered; i.e., these arguments are to be transmitted from the calling program to the called subroutine. The arguments must agree in number, type, and order with the SUBROUTINE statement arguments. (See Section VI.)

RETURN STATEMENT

The RETURN statement has the form:

```
RETURN
```

The return statement causes control to be transferred from a function subprogram or subroutine subprogram to the point in the using program at which it was relinquished. In the case of a subroutine, control is returned to the first executable statement following the CALL statement that gave control to the subroutine. If the CALL statement is the terminal statement of a DO loop and the DO loop has not been satisfied, control returns to the first executable statement in the DO loop. In the case of a function subprogram, control returns to the statement in which the function is imbedded. Since a RETURN marks the end of logical flow of the subprogram, there may be more than one RETURN in a subprogram.

CALL CHAIN STATEMENT

A CALL CHAIN statement has the form:

```
CALL CHAINx
```

Where: x is the character that identifies the chain. The programmer can use any digit or letter as identifier.

The programmer divides the input deck into groups of related programs called chains. Each chain is an independent memory load that includes all the routines (library and execution package) required by programs in the chain. In the input deck a `*CHAIN, x` card identifies each chain. The programmer can then use the CALL CHAINx statement to call any chain at any time. The called chain will overlay the chain currently in memory. Control is transferred to the first main program of the chain called. See the discussion of the COMMON statement (Section IV) for additional information on chaining.

CONTINUE STATEMENT

A CONTINUE statement has the form:

```
CONTINUE
```

This is a dummy statement which does not alter the sequence of program instructions. It is usually assigned a statement label and used to reference a point in the program. An example is the termination of a DO loop.

PAUSE STATEMENT

A PAUSE statement has the form:

```
PAUSE
  or
PAUSE n
```

Where: n is an identification constant of six or less octal digits, the first of which must be less than or equal to 3 for a 16K machine.

The PAUSE statement causes a halt in the execution of the program. The statement is usually included to allow time for operator action. The identification constant n, when included, indicates the particular PAUSE statement which caused the halt. The identification constant, if present, is stored in the machine's A- and B-address registers, which can be interrogated from the console. (See Section IX for the console display of the identification constant.) The program will continue execution with the next statement upon resumption of the run.

STOP STATEMENT

A STOP statement has the form:

```
STOP
 or
STOP n
```

Where: n is an identification constant of six or less octal digits, the first of which must be less than or equal to 3 for a 16K machine.

The STOP statement causes final termination of the object program. When no identification constant is present, an automatic exit to the monitor occurs. The identification constant n, when included, indicates the particular STOP statement which caused termination and is stored in the machine's A- and B-address registers, which can be interrogated from the console. When a STOP n is encountered, a halt occurs and n is displayed in the A- and B-address registers. (See Section IX for console display of the identification constant.) Upon resumption of the program, an exit to the monitor occurs.

END STATEMENT

An END statement has the form:

```
END
```

The END statement must be the final statement of a main program or of a subprogram. When control passes to an END statement in a main program, it is executed as a STOP. In a subprogram it is executed as a RETURN. The END statement may not have a statement label. The statement operator, END, may appear anywhere in columns 7 through 72.

SECTION IV
SPECIFICATION STATEMENTS

The programmer uses specification statements to declare or specify certain information about the object program that the compiler cannot obtain in any other way. There are nine specification statements, listed in Table 4-1. Specification statements are nonexecutable.

Table 4-1. Specification Statements

Statement	
COMMON	
DATA Initialization	
DIMENSION	
EQUIVALENCE	
EXTERNAL	
INTEGER	Data-Type
LOGICAL	Data-Type
REAL	Data-Type
TITLE	

DIMENSION STATEMENT

The DIMENSION statement is used to specify to the compiler how much memory will be required for arrays used in the program. A DIMENSION statement indicates the names of the arrays, the number of dimensions of each array named, and the maximum size of each dimension.

A DIMENSION statement has the form:

$\boxed{\text{DIMENSION } v_1 (i_{11}, i_{12}), v_2(i_{21}, i_{22}) \ldots v_n(i_{n1}, i_{n2})}$

Where: $v_1, v_2, \ldots v_n$ are names of arrays.

i_{n1} is an unsigned integer constant (greater than zero), representing the maximum number of rows in array v_n.

i_{n2} is an unsigned integer constant (greater than zero), representing the maximum number of columns in two-dimensional array v_n.

(i_{n2} is not written when the array is one-dimensional, nor is the comma preceding it.)

As indicated by the format, a DIMENSION statement can be used to declare all arrays in a given program. Each array is separated from the previous array by a comma. The integer constants following the name of the array give the maximum number of array elements in the array dimension. The first dimension following the name applies to rows and the second dimension, when present, represents columns. For example:

```
        DIMENSION ARRAY (3,2), MATRIX (4,5), B (10)
```

indicates three arrays, the first of which has three rows and two columns; the second array has four rows and five columns; and the third array is one-dimensional with 10 elements.

During execution of the object program, if a reference to an array element contains a subscript which assumes a value larger than the maximum specified in the DIMENSION statement, or if it assumes a zero or negative value, the computational results will be erroneous.

Although information about the names, sizes, and dimensions of arrays is most usually given in DIMENSION statements, it is sometimes convenient to use COMMON and data-type statements for this purpose.

COMMON STATEMENT

Use of the COMMON statement permits different programs to share common memory areas that are never overlaid. The COMMON statement can be used to set up a nonexclusive common block, called an unlabeled common block, or to set up exclusive, or labeled, common blocks.

An unlabeled common block provides data storage that is common to all programs within a job. The smallest amount of common storage used in any of the chains of the job is the amount which is reserved for the job. For example, in a job consisting of three chains, where chain 1 specifies 100 memory cells in unlabeled common storage chain 2 specifies 200, and chain 3 specifies 300, only the first 100 cells are preserved after execution of any one of the chains.

A labeled common block provides data storage common to programs within a chain only. Labeled common storage is released after executing the chain.

A COMMON statement has the form:

COMMON $v_1, v_2, \ldots, v_n$

or

COMMON /LBL1/ $v_1, v_2, \ldots, v_n$/LBL2/ $v_1', v_2', \ldots, v_n'$

Where: LBL1, LBL2, etc.	are names of common blocks within the common region. Block names are enclosed between two slashes. Block names may be one to six alphabetic or numeric characters, the first of which must be alphabetic. When no block name is given, the unlabeled common region is designated.
$v_1, v_2, \ldots, v_n$	are names of single variables or arrays that are to be assigned in the order listed to the common block named in the preceding label or to the unlabeled common region when no label is given. If an array is named, it may be followed by its dimensioning information in parentheses.

A single COMMON statement can be used to define any number of labeled common blocks and a single unlabeled common block. Refer to Figure 4-1 for an example of the definition of three labeled blocks. When defining an unlabeled common block in the same COMMON statement with labeled common blocks, the programmer omits the label designation between the slashes. The unlabeled common block can also be defined in a separate COMMON statement as shown in the first format above.

For example, COMMON statement 1 below is the equivalent of COMMON statements 2 and 3.

```
1.    COMMON /LABEL1/U, V(2,3), VARIBL/ /A, B, C, D(3,4), E
2.    COMMON A, B, C, D(3,4), E
3.    COMMON /LABEL1/U, V(2,3), VARIBL
```

In each case, an unlabeled common block is defined that contains variables A, B, C, and E, and array D. A single labeled common block is defined that contains variables U and VARIBL and array V.

Variables and arrays are stored in each common block in the order in which they appear in the COMMON statement. Equivalent variables in different programs must have exact positional correspondence in the COMMON statements.

For example, program A in Figure 4-2 has three variables X, Y, and Z, which are equivalent to three variables in program B, CAT, RAT, and BAT, respectively. In the figure, a labeled common block called LABEL1 is constructed to provide the required equivalence. Since X is equivalent to CAT, the two variables are assigned corresponding positions in their respective COMMON statements. Similarly, the remaining variables are equated by positional correspondence.

$$\text{COMMON } /\overbrace{\text{LABEL1}}^{\substack{\text{BLOCK \#1}\\ \text{LABEL}}}/\underbrace{\text{A, B, C, D(3, 4)}}_{\substack{\text{Variables and Ar-}\\ \text{ray in Block \#1}}}\ /\overbrace{\text{LABEL2}}^{\substack{\text{BLOCK \#2}\\ \text{LABEL}}}/\underbrace{\text{X, Y, Z}}_{\substack{\text{Variables}\\ \text{in}\\ \text{Block \#2}}}\ /\overbrace{\text{LABEL 3}}^{\substack{\text{BLOCK \#3}\\ \text{LABEL}}}/\underbrace{\text{U, V(2, 3), VARIBL}}_{\substack{\text{Variables and Array}\\ \text{in Block \#3}}}$$

Figure 4-1. COMMON Statement for Three Labeled Blocks

Program A:
COMMON /LABEL1/X, Y, Z

Program B:
COMMON /LABEL1/CAT, RAT, BAT

Memory Assignment

1	2	3
X	Y	Z
CAT	RAT	BAT

Common block "LABEL1"

Figure 4-2. Communication Via Positional Correspondence

The size of a common block in a program (or subprogram) is the sum of the storage locations required for all the elements declared (through COMMON and EQUIVALENCE statements) to be in that block. The sizes of identically labeled common blocks must be the same in programs which are to be executed together; i. e. , they must have the same total number of variables and/or array elements. The sizes of unlabeled common blocks in programs which are to be executed together need not be the same.

An array may be dimensioned in a COMMON statement by referencing the highest subscript of the array. For example, the statement

```
        COMMON A(3, 4)
```

dimensions array A as a three-by-four array and places it in the common region as shown below:

Memory Assignment

1	2	3	4	5	6	7	8	9	10	11	12
A(1, 1)	A(2, 1)	A(3, 1)	A(1, 2)	A(2, 2)	A(3, 2)	A(1, 3)	A(2, 3)	A(3, 3)	A(1, 4)	A(2, 4)	A(3, 4)

There are two ways to equate common elements which have different positions within a block. Consider the statement in program 1:

```
COMMON A, B(2, 2), C, D, E
```

If four variables in program 2 are equivalent to A, C, D, and E, program 1 may be reconstructed to include labeled common blocks as follows:

```
COMMON /LABELA/A, C, D, E/ /B(2, 2)
```

Program 2 would contain the statement:

```
COMMON/LABELA/ALPHA, BETA, GAMMA, PSI
```

As an alternative to changing program 1 to include a labeled block, program 2 may introduce a dummy array to "space over" the uninteresting portion of the common block. Spacing over array B is illustrated in Figure 4-3.

Program 1:

```
COMMON A, B(2, 2), C, D, E,
```

Program 2:

```
COMMON ALPHA, DUMMY (4), BETA, GAMMA, PSI
```

Memory Assignment:	1	2	3	4	5	6	7	8
	A	B(1, 1)	B(2, 1)	B(1, 2)	B(2, 2)	C	D	E
	ALPHA	DUMMY(1)	DUMMY(2)	DUMMY(3)	DUMMY(4)	BETA	GAMMA	PSI

Figure 4-3. Use of Dummy Array to Space Over Common Area

EQUIVALENCE STATEMENT

The EQUIVALENCE statement permits two or more variables or array elements to share the same memory location. The statement makes it possible to conserve required memory space or to establish two or more names for the same variable.

An EQUIVALENCE statement has the form:

$$\boxed{\text{EQUIVALENCE } (v_{1a}, v_{1b}, v_{1c}, \ldots), (v_{2a}, v_{2b}, v_{2c}, \ldots), \ldots}$$

Where:	
$v_{1a}, v_{1b}, v_{1c}, \ldots$	are the names of variables or array elements that are to share the same memory location.
$v_{2a}, v_{2b}, v_{2c}, \ldots$	are the names of variables or array elements that are to share another memory location.

All of the variables enclosed within a set of parentheses are assigned to the same location; hence, they are called an equivalence set. The maximum number of unrelated equivalence sets is 64. There may be any number of variables within one set of parentheses, and any number of sets of parentheses in a single EQUIVALENCE statement. Variable names must be separated by commas, as must sets of parentheses.

An EQUIVALENCE statement may relate single variables to each other, entire arrays to each other, elements of an array to single variables, or vice versa. Array elements may appear in EQUIVALENCE statements. An element of a two-dimensional array may be expressed in an EQUIVALENCE statement in either of two ways.

1. It may be expressed exactly as in a DIMENSION statement; i. e. , when the element is part of a two-dimensional array, it may be written (within parentheses) as two integer constants separated by a comma.

   Example:

   Element A (2, 3) of the two-dimensional array A (3, 3) may be equivalenced to variable C(7) of the one-dimensional array C(10) as follows:

   ```
   EQUIVALENCE (A(2, 3), C(7) )
   ```

2. It may be expressed as the equivalent single-dimensioned subscript that shows the order in which the element is stored. Refer to Figure 1-4. The circled numbers indicate single-dimensioned subscripts of the two-dimensional array shown.

   Example:

   Element A (2, 3) of the two-dimensional array A (3, 3) may be equivalenced to variable C(7) of the one-dimensional array C(10) by single-subscript method as follows:

   ```
   EQUIVALENCE (A(8), C(7) )
   ```

When one element of one array is equivalenced to an element of another array, the remainder of the array elements are automatically equivalenced. The declared equivalence of the two elements determines the positional correspondence of the rest of the elements.

Example:

```
DIMENSION A(7), B(3, 3)
EQUIVALENCE (A(3), B(2, 1))
```

Memory Assignment	1	2	3	4	5	6	7	8	9	10
	A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)			
		B(1, 1)	B(2, 1)	B(3, 1)	B(1, 2)	B(2, 2)	B(3, 2)	B(1, 3)	B(2, 3)	B(3, 3)

It is permissible for a variable or an element of an array to appear in both a COMMON statement and an EQUIVALENCE statement, provided that the common region is not extended in such a way that one of the following three rules is violated.

1. Identically labeled common blocks in programs that are to be run together must be of the same size. (The unlabeled common region is excluded from this restriction.)
2. The common region may be extended only in the direction away from the origin (i. e., to the right in the diagrams shown in this manual). An illegal extension of common region is shown in Figure 4-4. Figure 4-5 shows a legal extension of the common region.
3. Elements in the common region may not be equivalenced to other elements in the common region.

```
COMMON A, B, C,
DIMENSION D(3)
EQUIVALENCE (B, D(3))
```

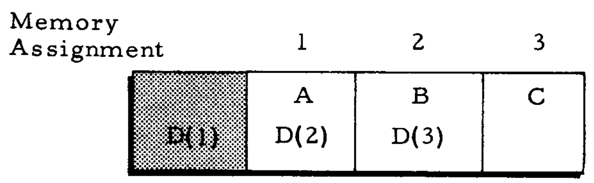


Figure 4-4. Illegal Extension of Common Region

```
COMMON A, B, C
DIMENSION D(3)
EQUIVALENCE (B, D(1))
```

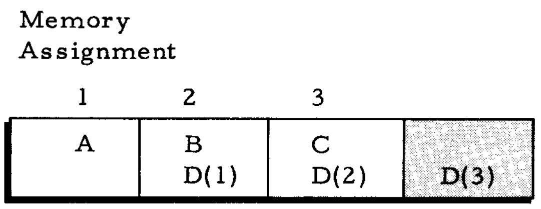


Figure 4-5. Legal Extension of Common Region

A variable or array element that is equivalenced to an element in the common region is itself treated as though it were assigned to the common region, even if it does not appear in a COMMON statement.

DATA-TYPE STATEMENTS

There are three statements used to declare explicitly the data types of variables, arrays, or functions. These are of the form:

REAL	$v_1, v_2, v_3, \ldots$
INTEGER	$v_1, v_2, v_3, \ldots$
LOGICAL	$v_1, v_2, v_3, \ldots$

Where: $v_1, v_2, v_3 \ldots$ are the names of variables, arrays, or functions.

As described in Section I, real and integer variables, arrays, and functions need not be explicitly declared in a data-type statement. If the first letter of the variable name begins with I, J, K, L, M, or N an integer type is implied; all other first letters imply real data. A data-type statement may be used to override the implied data type and must be used to declare logical type data.

Once declared, data types remain constant throughout the program and cannot be changed. Therefore, it is illegal for the same name to appear in two different data-type statements. Individual elements of an array assume the same data type that is associated with the array. Arrays can be dimensioned in a data-type statement, exactly as in a DIMENSION or COMMON statement.

EXTERNAL STATEMENT

Appearance of a name in an EXTERNAL statement indicates to the compiler that the name is that of an external procedure (function or subroutine). An EXTERNAL statement has the form:

```
EXTERNAL a, b, c,......
```

Where: a, b, c, are the names of functions and/or subroutines that appear in the call argument lists to function or subroutine subprograms.

An EXTERNAL statement is used for function or subroutine names when the name appears in the argument list of a CALL statement or a function reference argument list and has not previously been declared as an external procedure by its use in a CALL statement or function call. Thus, any subroutine, A, to be used as a call argument, must appear in an EXTERNAL statement if it does not appear in a CALL statement as:

CALL A (Refer to Section III).

Any function subprogram, B, to be used as a call argument, must appear in an EXTERNAL statement if it does not appear in a function call in an arithmetic or logical expression such as:

D = B(E, F, G) (Refer to Section VI).

The name of a function or subroutine may appear in both an EXTERNAL statement and a data-type statement. The name of a compiler-supplied function may not appear in an external statement. (Refer to Section VI for compiler-supplied functions.)

In the example below, the function subprogram FSUB is to be included in the argument list to function subprogram ABLE. Thus, the name FSUB must be declared in an EXTERNAL statement in the program which calls subprogram ABLE to distinguish it from the names of variables in the argument list, such as R and T.

Excerpt from Main Program

```
   EXTERNAL FSUB
   ....
   ....
 4 Y = R + ABLE (FSUB, R, T)
   ....
```

Function Subprograms ABLE and FSUB

```
    FUNCTION ABLE (XFUN, A, B)
 20 ABLE = XFUN (A) + B**2
    ....
    RETURN
    ....
    END

    FUNCTION FSUB (X)
    ....
    ....
 30 FSUB = X**3 + 25.
    ....
    ....
    ....
    ....
    END
```

At statement 4 in the main program, subprogram ABLE is called. The real arguments passed to ABLE are the subprogram name FSUB and the numeric values of R and T. These real arguments are substituted for their respective dummy arguments, ZFUN, A, and B. When execution of statement 20 begins, the dummy subprogram name, XFUN, is encountered. This calls the real subprogram FSUB. FSUB is executed, and when the END statement is encountered, control returns to statement 20 of subprogram ABLE. Execution of subprogram ABLE continues until either the RETURN or the END statement is encountered, at which time control returns to statement 4 of the main program.

TITLE STATEMENT

A TITLE statement has the form:

```
ΔTITLE prgnam
```

The TITLE statement is an optional statement for naming a main program. Column 1 is left blank; the word TITLE appears in columns 2 through 6; and the program name is taken to be the six characters that appear in columns 7 through 12. If no TITLE statement is used and there is no FUNCTION or SUBROUTINE statement, the compiler automatically considers the program to be a main program and assigns a name to it.

DATA INITIALIZATION STATEMENT

When an object program is loaded for execution, initial values can be assigned to variables and/or array elements by means of a DATA initialization statement. The process of assigning starting values is called initialization. No variable or array element can be initialized in a DATA initialization statement if it has been assigned to a common or labeled common block in a COMMON statement.

A DATA initialization statement has the form:

$$\boxed{\text{DATA } k_1/d_1/,\ k_2/d_2/, \ldots\ldots\ldots\ldots\ k_n/d_n/}$$

Where: Each k is a list containing names of variables and array elements which are to be assigned initial values. No dummy variables are permitted.

Each d is a list of constants, signed or unsigned. The values of these constants are assigned to the variables of the preceding variable list when the program is loaded.

An unsigned integer constant, j, and an asterisk may precede any constant in the constant list. That constant is then repeated j times.

The variable list and the constant list that follows must have a one-to-one correspondence, since the value of the first constant will be assigned to the first variable, the value of the second constant will be assigned to the second variable, etc. As long as this one-to-one correspondence is maintained, the lists may have any over-all length up to the capacity of nine continuation lines.

Hollerith and octal constants, as well as integer constants, must have corresponding integer variables, either beginning with I, J, K, L, M, or N or previously typed in an INTEGER statement. Variables corresponding to logical constants must have been previously defined in a LOGICAL statement.

The constant list is bounded by slashes. Commas separate individual entries in each of the lists. The DATA initialization statement must follow all other specification statements in the source program and must precede the first executable statement as shown on page 1-6.

Example:

$$\text{DATA } \underbrace{\text{A, I, J, K /}}_{\text{Variables}}\ \underbrace{\text{29. 3, 5, 3H5KC, 403777/}}_{\text{Constants}}$$

At loading time the following data assignments are made:

Constant	Variable to Which Constant is Assigned
29.3	A
5	I
5KC	J
3777	K

If a number of variables are to be initialized to the same value, the appropriate constant in the list may be preceded by a repetition constant and an asterisk as shown in the example below.

Example:

DATA A, B, C/1.0, 1.0, 1.0/

may be written equivalently as: DATA A, B, C /3 *1.0/

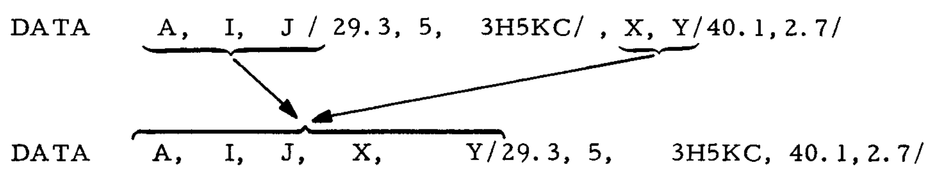
Repetition Constant　　Multiplication Symbol

Additional List Pairs

The DATA initialization statement may be expanded to accommodate more than one pair of lists, simply by adding a comma after the previous list of constants, then adding the next list of variables, followed by its associated list of constants and a terminating slash.

Any number of paired lists may be added in this manner (up to the capacity of 9 continuation cards). The effect is the same as if one pair of long lists were written. (This option is convenient when adding variables and initialization values in a subsequent version of the program.)

Example:



```
        DATA    A, I, J / 29.3, 5,  3H5KC/ , X, Y/40.1,2.7/
is equivalent to:
        DATA    A, I, J,  X,    Y/29.3, 5,  3H5KC, 40.1,2.7/
```

Implied DO Loops

The DATA list can include not only variables and names of array elements but also implied DO loops. Implied DO loops are described in detail in relation to input/output statements in Section V. Briefly, an implied DO loop provides a shortened notation, similar to a DO loop, for describing repetitive variables. In the DATA statement it is particularly useful for initializing all or some of the elements of an array.

The general form of the implied DO loop is:

$$\boxed{(v_1, v_2, \ldots, v_n, i=m_1, m_2, m_3)}$$

Where: Each v is a variable or array element.

i is an integer variable that controls the implied DO loop and can be used as the subscript of array elements if present.

Each m is a parameter of the implied DO loop. They must be unsigned integer constants.

m_1 is the initial value of i.

m_2 is the terminal value of i.

m_3 is the value by which i is incremented at each iteration. If not explicitly stated, m_3 is understood to be 1.

Example 1: A one-dimensional array is named ARRAY. Each equivalent expression represents the first 10 elements of ARRAY.

(ARRAY(I), I = 1, 10)

is the implied DO loop equivalent of

ARRAY(1), ARRAY(2), ARRAY(3), ARRAY(4), ARRAY(5),
ARRAY(6), ARRAY(7), ARRAY(8), ARRAY(9), ARRAY(10)

Example 2: A two-dimensional array is named B. Each equivalent expression represents six elements from alternate columns of the second row of array B.

(B(2,I), I = 1, 11, 2)

is the implied DO loop equivalent of

B(2,1), B(2,3), B(2,5), B(2,7), B(2,9), B(2,11)

An example of the use of a single implied DO loop to initialize an entire array is shown below. All 20 elements of the array named C are to be initialized with the value 0.0. At the same time, variable A is to be assigned a value of 5.0, and variable B a value of 3.0.

$$\text{DATA}\quad \underbrace{\text{A, B,}}_{\text{Variables}}\quad \underbrace{\text{(C(I), I = 1,20)}}_{\text{Implied DO Loop}}\ /\ \text{5.0, 3.0, } \underset{\substack{\uparrow\\ \text{Repetition Constant}}}{20}\ \underset{\substack{\uparrow\\ \text{Multiplication Symbol}}}{*}\ \underset{\substack{\nwarrow\\ \text{Initialization Value for Each Array Element}}}{0.0}\ /$$

When the list contains a single implied DO loop, as in the above example, the parameters of the DO loop must be integer constants. When the list contains a nested set of implied DO loops (refer to next paragraph), the parameters of the outermost loop must be integer constants, but the parameters of an inner loop may be: (1) integer constants and/or (2) integer variables which appear as control variables in an outer implied DO loop of the same nest. Only integer variables may be used as subscripts of an implied DO loop appearing in a DATA statement.

When initializing an entire array, as above, it is also permissible to use an array name alone in a DATA statement list to represent all elements of an array when the array has been declared and dimensioned in a DIMENSION, COMMON, or data-type statement. The DATA statement above could then have been written as:

```
DIMENSION C(20)
DATA A, B, C / 5.0, 3.0, 20 * 0.0 /
```

Implied DO loops are thus most useful when applied only to a portion of an array.

Nested Pairs of Implied DO Loops

Implied DO loops can be nested to a depth of two. The general form of a nested pair of implied DO loops is shown below.

$$\boxed{\underbrace{\underbrace{((v_1,\ v_2,\ \ldots,\ v_k,\ i{=}m_1,\ m_2,\ m_3),}_{\text{inner loop}}\ \ j{=}n_1,\ n_2,\ n_3)}_{\text{outer loop}}}$$

Where: v_1, v_2, ..., v_k is a list of variables or array elements

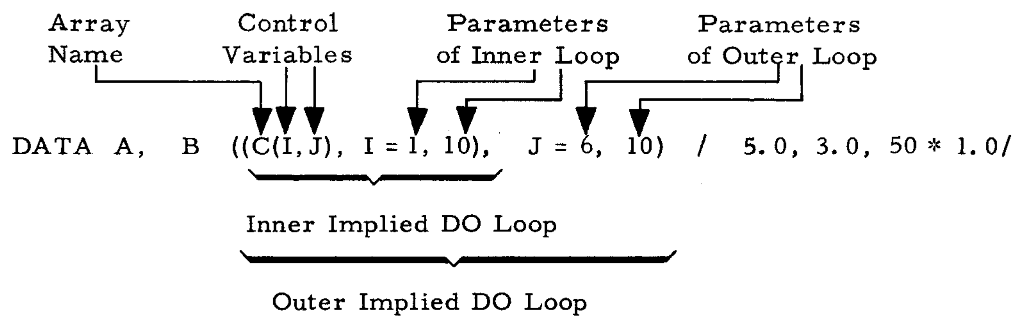
i is the control variable of the inner DO loop

j is the control variable of the outer DO loop

m_1, m_2, m_3 are respectively the initial, terminal, and incremental values of i.

n_1, n_2, n_3 are respectively the initial, terminal, and incremental values of j.

An example of a nested set of two implied DO loops is given below. It is desired to initialize with a value of 1.0 one-half of the elements of a 10 x 10 array named C. The following statement will initialize the elements as shown in Figure 4-6.



```
DATA A, B ((C(I,J), I = 1, 10), J = 6, 10) / 5.0, 3.0, 50 * 1.0/
```

1, 6	1, 7	1, 8	1, 9	1, 10
2, 6	2, 7	2, 8	2, 9	2, 10
3, 6	3, 7	3, 8	3, 9	3, 10
4, 6	4, 7	4, 8	4, 9	4, 10
5, 6	5, 7	5, 8	5, 9	5, 10
6, 6	6, 7	6, 8	6, 9	6, 10
7, 6	7, 7	7, 8	7, 9	7, 10
8, 6	8, 7	8, 8	8, 9	8, 10
9, 6	9, 7	9, 8	9, 9	9, 10
10, 6	10, 7	10, 8	10, 9	10, 10

Figure 4-6. Right-hand Portion of Array to be Initialized

SECTION V
INPUT/OUTPUT STATEMENTS

Input/output (I/O) statements are the programmer's tools for directing the flow of information between peripheral devices and the central processor so that the data can be precisely understood by both man and machine. Fortran Compiler D accomplishes all actual data transfer with a single input statement and a single output statement — READ and WRITE, respectively. Associated with I/O statements are a FORMAT statement and three I/O device manipulation statements. The FORMAT statement specifies the physical arrangement of data on peripheral input or output media, indicates the type of input or output conversion required between machine language and external data, and specifies editing information. The three I/O device manipulation statements are BACKSPACE, END FILE, and REWIND.

Every READ statement involving transfer of data that is not in binary form and every WRITE statement involving transfer of data that must be converted into other than binary form must be accompanied by a FORMAT statement. The transfer of data in binary form means that the data is passed and stored just as it appears in memory. No editing or conversion is applied. FORMAT statements are nonexecutable but are interspersed with the executable program statements. The FORMAT statement is discussed in detail beginning on page 5-11. All other I/O statements are executable. Page references for I/O statements are given in Table 5-1. For layouts of BCD and binary tapes, see Appendix F.

Table 5-1. I/O Statements

I/O Statement	Page Reference
BACKSPACE	5-56
ENDFILE	5-55
FORMAT	5-11
READ	5-1
REWIND	5-55
WRITE	5-3

READ STATEMENT

The READ statement has the form:

```
     READ (i,n) list
or:  READ (i) list
```

Where: n is either

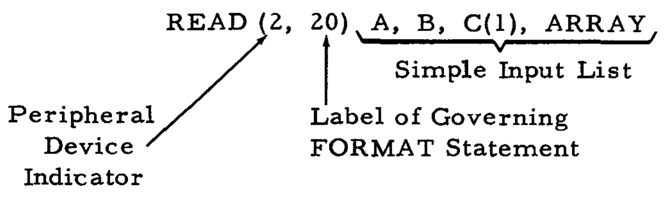
1. the statement label of a FORMAT statement which describes how the incoming data are arranged and the type of conversion required, or
2. the name of an array in which the necessary format information is stored.

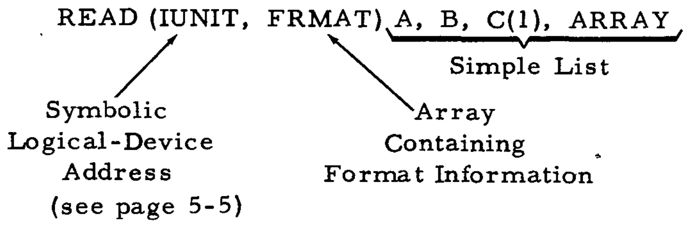
i is a code identifying the input device (a magnetic tape unit or card reader). It may be written as either an unsigned integer constant ($1 \leq i \leq 15$) or an integer variable.

list is a correctly sequenced list of the names of variables, arrays, and/or array elements that are to receive input values at execution time. Successive names must be separated by a comma. Since the list sequence indicates the order (from left to right) in which the names will receive input values, the list sequence must correspond to that of the input data. The list may be empty.

Under the first form — READ (i, n) list — successive records of "formatted" information (sometimes called binary-coded-decimal information) are read from the designated peripheral unit under control of FORMAT statement n until the entire input list is satisfied.[1] If the peripheral device is indicated by an integer variable, the value of the variable must be set to the appropriate unit number prior to execution of the READ statement. The value of this variable may be changed during execution of the program.

Examples:


READ (2, 20) A, B, C(1), ARRAY
Simple Input List
Peripheral Device Indicator
Label of Governing FORMAT Statement


READ (IUNIT, FRMAT) A, B, C(1), ARRAY
Simple List
Symbolic Logical-Device Address (see page 5-5)
Array Containing Format Information

In the second form — READ (i) list — no FORMAT statement is designated because the input data are automatically understood to be in binary form whenever this version of the input statement is used. All values read into memory by a single execution of the statement come from one logical record.

[1] For convenience throughout Section V, FORMAT statement label n has been set equal to 20 in all examples. This number is purely arbitrary. Also throughout the section, the card reader is assigned unit number 2, the printer is assigned unit number 3 and the card punch is assigned unit number 5. These assignments follow Honeywell conventional practice; customer installations may make other unit assignments.

Example:

$$\text{READ }\underset{\text{Peripheral Device Indicator}}{(2)}\ \underbrace{\text{A, B, C(1), ARRAY}}_{\text{Simple Input List}}$$

An input or output list may contain a simple list of names, an implied DO loop, or a combination of the two. Integer variables in an input/output list may be used in subscript expressions elsewhere in the list, and the input value will be the value used in the subscript expression. If an integer variable in the list is a parameter of an implied DO loop, it must appear prior to and external to the range of the implied DO loop.

An error in a READ statement detected during execution of a job, such as an attempt to read from the printer, will cause job termination and printout of an error message. (See Appendix G.)

A WRITE or END FILE statement cannot be directly followed by a READ statement that references the same device.

WRITE STATEMENT

A WRITE statement has the form:

```
WRITE (i,n) list, or
WRITE (i) list
```

Where: n is either

1. the statement label of a FORMAT statement which describes how the outgoing data are to be arranged on the output medium and the type of conversion required, or

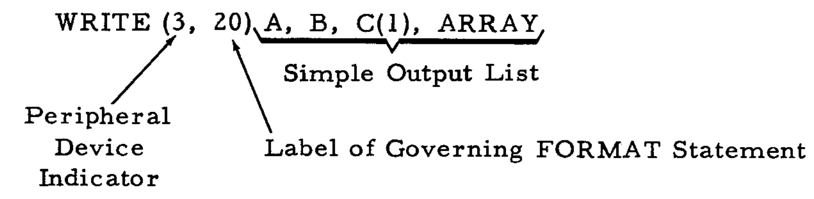
2. the name of an array in which the necessary format information is stored.

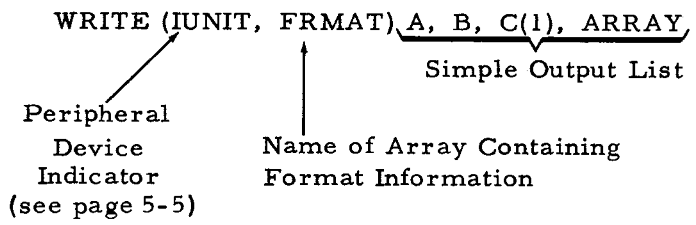
i is a code identifying the output device. It may be written as either an unsigned integer constant ($1 \leq i \leq 15$) or an integer variable which must have a value at execution time.

list is a correctly sequenced list of the names of variables, arrays, and/or array elements which are to transmit their associated values at execution time. Successive names must be separated by a comma. The list sequence must correspond to the desired sequence of the output data.

In the form—WRITE (i, n) list—the output device code, i, addresses a printer, card punch, tape unit, or other output device. Each execution initiates printing of a new line, writing of a new tape record of formatted information, or punching of a new card (as the case may be) and causes a value to be transmitted from memory to the external medium for each named element in the list. Values are transmitted in the order given in the list, converted to external form (under control of FORMAT statement n or an equivalent format array), and placed in sequential data fields of the printed line, tape record, or punched card in the same order as they are transmitted. The number of columns allotted to each value is specified in the FORMAT statement. Blank spaces and titular information may be interspersed between values, when the FORMAT statement so specifies.

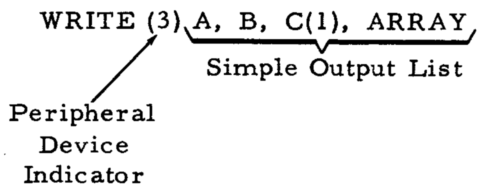
Examples:


WRITE (3, 20) A, B, C(1), ARRAY
Simple Output List
Peripheral Device Indicator
Label of Governing FORMAT Statement


WRITE (IUNIT, FRMAT) A, B, C(1), ARRAY
Simple Output List
Peripheral Device Indicator (see page 5-5)
Name of Array Containing Format Information

In the WRITE (i) list statement, no FORMAT statement is designated because it is automatically understood that binary output is requested whenever this version of the WRITE statement is used; no other format information is necessary. The entire string of physical records written by a single execution of a WRITE (i) list statement is termed a logical record.

Example:



If the output list has not been satisfied by the end of one line, tape record, or card, it is possible to print additional lines, write additional records, or punch additional cards by the same single execution of the output statement, until a value has been transmitted for every item in the list. In such cases, the programmer must include a record terminator (slash or right parenthesis terminating the FORMAT) at the appropriate place in the FORMAT statement (see page 5-45) to insure that no record exceeds the maximum size of 131 print positions, or 132 tape characters, or 80 punched columns. If no record terminator is given where required in the FORMAT statement and the field specifications call for more characters than the maximum permitted in one record on the particular output device, the extra characters will be ignored.

In either form of the WRITE statement, if the peripheral device is indicated by an integer variable (as in Example 2 above), the value of the variable must be set to the appropriate unit number prior to execution of the WRITE statement. The value of this variable may be changed during execution of the program. When using a variable device designation, the programmer must specify as a constant, somewhere in his job (possibly in a dummy input/output statement which need not be executed), the peripheral device number of the particular input or output device in question. Furthermore, this constant must appear in an input or output statement that is of exactly the same type as the statement in which the variable unit designation appears. Fulfillment of these requirements enables the compiler to allocate the physical devices and the buffer space for each physical device.

An error in a WRITE statement detected during execution of a job, such as an attempt to write onto a card reader, will cause job termination and printout of an error message. (See Appendix G.)

INPUT/OUTPUT LISTS

All READ and WRITE statements make use of lists of variables, arrays, and/or array elements to be transferred either to or from memory. The lists may be classified either as simple lists or as lists containing implied DO loops and nested pairs of implied DO loops. These lists are described in detail in this section. The lists apply equally to input (READ) and output (WRITE) statements. One main difference between input and output operations should be noted, however. An integer variable appearing as part of the control information for an implied DO loop or in a subscript in an output list must be assigned a value prior to execution of the output statement. In an input operation, the integer variable may be assigned a value during execution of the input statement by having the input data designate the value of the integer variable. Assignment of values to integer variables during execution of an input statement is illustrated in this section.

Simple Lists

A simple list is a series of names of variables, arrays, and/or array elements, with a comma separating each two successive names, e. g.:

```
A, B, C(1), ARRAY
```

Each name in the list is called a list item. Because the list is scanned from left to right, values are assigned to (or transmitted from) the leftmost list item first, then to (or from) the next leftmost item, and so on. Thus the transfer sequence for the above example is:

1. A
2. B
3. C(1)
4. ARRAY

That is, for input operations, the first incoming data field is assigned to variable A (under control of the first field specification of the associated FORMAT statement), the second incoming data field is assigned to B (under control of the second field specification of the FORMAT statement), etc. Similarly, for output operations, the value of variable A is the first value to be transmitted from memory to the output device (under control of the first field specification in the associated FORMAT statement), and so on.

A simple list may also be the name of a single variable, array, or array element.

Integer variables appearing in an input/output list may be used in subscript expressions elsewhere in the list.

Example 1: READ (2, 20) I, C (I)

Example 2: READ (2, 20) C(I), I

Because of the left-to-right scan of the list, the results are not equivalent in the two examples above. The following rule defines how the subscript expressions are evaluated at execution time in both cases:

> If the subscript expression appears later in the list than the integer variable which it employs (as in Example 1 above), the subscript expression is evaluated using the newly read-in value of the integer variable.
>
> If the subscript expression appears earlier in the list than the integer variable which it employs (as in Example 2 above), the subscript expression is evaluated using the value last defined for the integer variable.

To illustrate: Assume that integer variable I was previously assigned a value of 3. During execution of the READ statement, the previous value will be replaced with an incoming value of 5. Is C(I) evaluated as C(3) or C(5)? According to the rule given above, the answer is C(5) in the first example and C(3) in the second example.

The reason for the rule is as follows. During execution of an input statement, each list item receives the input value at the instant when the item is encountered in the sequence of scanning the list. Subscript expressions in the list are also evaluated at the time when they are actually encountered during the scan.

Thus, integer variable I receives its new value of 5 as soon as the scan encounters I while proceeding from left to right. In the first example, C(I) is evaluated as C(5) because I had just previously assumed that new value of 5. Thus, it is array element C(5) which receives the next incoming value. In the second example, however, C(I) is encountered before the incoming value of 5 is assigned to I. Thus, C(I) is evaluated with the value last assigned to I, which happens to be 3. Consequently, it is array element C(3) which receives the first value of incoming data. Then I assumes its new value of 5.

The form shown in Example 1, wherein the integer variable precedes its use in a subscript expression, is useful when it is desired to have the input data designate both the array element and the value for that array element.

On output, the integer variable must be assigned a value prior to execution of the output statement.

Short-List Notation for Input/Output of Entire Arrays

The inclusion of an array name without subscripts in an input/output list causes values to be transmitted for all elements of the array (assuming that the array has already been declared and dimensioned in a DIMENSION, COMMON, or data-type declaration statement). This usage is called short-list notation. The sequence in which the array elements are transmitted is the same as the storage sequence described on page 1-10. Only after all values of the complete array have been transmitted is the next list item considered.

Lists with Implied DO Loops

Implied DO loops and nested pairs of implied DO loops were defined and described briefly in relation to the lists of the DATA initialization statement. Because of the power of the implied DO loop to save laborious and repetitious effort in writing lists, this section both repeats the information contained on pages 4-11 through 4-13 and offers more detailed explanation and examples.

When several variables and array elements are to be transferred to or from memory, the programmer may find it convenient to incorporate an implied DO loop into the input or output list to reduce the writing and keypunching that would be necessary if each variable or array element were to be written as an individual item of a simple list. The implied DO loop is particularly useful when several elements of an array are to be transferred to or from memory but not the entire array or if all the elements are to be transferred in a sequence different from that obtained by using the short-list notation.[1] However, the implied DO loop is useful in any case in which iterative transfer of variables to and from memory is required.

Though not literally a DO loop, the implied DO loop has the same effect of carrying out an iterative process, causing a control variable to be incremented after each repetition. The portion of a list that contains an implied DO loop is called an implicit list.

The general form of the implied DO loop is:

$$\boxed{(v_1,\ v_2,\ \ldots,\ v_n,\ i{=}m_1,\ m_2,\ m_3)}$$

[1]Although an implied DO loop could be used to transfer all elements of an array in the sequence in which they are stored, the short-list notation method described previously is more convenient and faster.

Where: Each v is a variable or array element.

i is an integer variable that controls the implied DO loop and can be used as the subscript of array elements when present.

m_1 is the initial value of i.

m_2 is the terminal value of i.

m_3 is the value by which i is incremented at each iteration. It may be an unsigned integer constant or integer variable. If not explicitly stated, m_3 is understood to be 1.

Parentheses in a READ or WRITE statement that do not bound the peripheral device indicator and FORMAT label are assumed to be the limits of an implied DO loop. Any variable or array element within the bounds of the left parenthesis and the comma preceding the control variable i is repeated during each iteration of the loop.

Example 1: WRITE (3, 20) (A, B, C(I), I = 1, 3)

is equivalent to:

WRITE (3, 20) (A, B, C(1), A, B, C(2), A, B, C(3)

Example 2: WRITE (3, 20) (A, B, C, D, I = 1, 3)

is equivalent to:

WRITE (3, 20) A, B, C, D, A, B, C, D, A, B, C, D

Example 3: READ (2, 20) (B(1, I), I = 1, 5)

is equivalent to:

READ (2, 20) B (1, 1), B(1, 2), B(1, 3), B(1, 4), B(1, 5)

Example 4: READ (2, 20) (A(I), I = 1, 8)

is equivalent to:

READ (2, 20) A(1), A(2), A(3), A(4), A(5), A(6), A(7), A(8)

Example 5: READ (2, 20) (C(I, I), I = 1, 6, 1)

is equivalent to:

READ (2, 20) C(1, 1), C(2, 2), C(3, 3), C(4, 4), C(5, 5), C(6, 6)

Like a simple list, a list containing an implied DO loop is scanned from left to right until the implicit portion is encountered. Then the implied DO loop is fully evaluated before the scan continues on to any remaining list items. The following example illustrates this.

READ (2, 20) A, B, (C(I), I= 1, 5), D, E, F

is evaluated as:

READ (2, 20) A, B, C(1), C(2), C(3), C(4), C(5), D, E, F

Note that all items specified by the implied DO loop appear before items D, E, and F, which are written later in the list than the implied DO loop.

When a nested set of implied DO loops is encountered during the left-to-right scan, the complete nest is evaluated before the scan continues on to any remaining list items. (Nests of implied DO loops and the order in which they are evaluated are discussed in detail below.)

The control variable, i, and any parameter (m_1, m_2, or m_3) that is written as an integer variable may also appear elsewhere in the list, either as a single variable or in a subscript expression (as part of an array element name), subject to the following restriction:

> During input operations, none of the parameters of an implied DO loop may appear in a simple list that is enclosed within the bounding parentheses of the same implied DO loop.

This rule follows from the restriction governing conventional DO loops that none of the DO parameters may be altered within the range of the DO loop (see page 3-5).

Examples:

```
Valid:      READ (2, 20)   A, N, B(N),  (C(I),  I=1, N)
                                  Permissible

Invalid:    READ (2, 20)   A, (N, C(I),  I = 1, N)
                                  Illegal
                            Bounding              Bounding
                          Parenthesis           Parenthesis
```

In both examples above, N is a parameter of the implied DO loop, because it represents the terminal value (m_2) of the control variable. In the second example, N also appears in a simple list enclosed within the bounding parentheses of the implied DO loop of which N is a parameter. This condition is not permitted in an input list, because the parameters of an implied DO loop may not be altered within the range of the implied DO loop.

When a parameter of an implied DO loop appears elsewhere in the list without violating the above restriction, the value of the parameter at evaluation time depends upon which is encountered first in the left-to-right scan of the list, the other appearance(s) or the parameter. The rules are exactly analogous to those given for integer variables used as subscripts elsewhere in a list (pages 5-6 and 5-7). For the same reason, array element B(N) in the example above is evaluated with the new value of N that is received during execution of the READ statement, not the previous value. For the same reason, the value of parameter N in the above example is also the newly received value of N.

Nested Pairs of Implied DO Loops

Implied DO loops can be nested to a depth of two. Nested pairs of implied DO loops are particularly useful in describing the elements of a two-dimensional array. The general form of a nested pair of implied DO loops is shown below.

$$\boxed{\underbrace{(\underbrace{(v_1, v_2, \ldots, v_f, i=m_1, m_2, m_3)}_{\text{inner loop}}, u_1, u_2, \ldots, u_k, j=n_1, n_2, n_3)}_{\text{outer loop}}}$$

Where: $v_1, v_2, \ldots, v_f$	is a list of variables or array elements.
$u_1, u_2, \ldots, u_k$	is a list of variables or array elements or may be empty.
i	is the control variable of the inner DO loop.
j	is the control variable of the outer DO loop.
m_1, m_2, m_3	are respectively the initial, terminal, and incremental values of i.
n_1, n_2, n_3	are respectively the initial, terminal, and incremental values of j.

Example 1: READ (2, 20) ((ARRAY(I, J), J = 1, 4), I = 1, 9, 2)

is equivalent to:

```
READ (2, 20) ARRAY (1, 1), ARRAY (1, 2), ARRAY (1, 3), ARRAY (1, 4),
             ARRAY (3, 1), ARRAY (3, 2), ARRAY (3, 3), ARRAY (3, 4),
             ARRAY (5, 1), ARRAY (5, 2), ARRAY (5, 3), ARRAY (5, 4),
             ARRAY (7, 1), ARRAY (7, 2), ARRAY (7, 3), ARRAY (7, 4),
             ARRAY (9, 1), ARRAY (9, 2), ARRAY (9, 3), ARRAY (9, 4)
```

Interpretation: Read in 20 values, storing them as follows:

1. Store the first four as the first four elements of the first <u>row</u> of ARRAY.
2. Store the second four as the first four elements of the third row.
3. Store the next four as the first four elements of the fifth row.
4. Store the next four as the first four elements of the seventh row.
5. Store the last four as the first four elements of the ninth row.

This statement may be thought of as being equivalent to the nest of DO loops:

```
      DO 10 I = 1, 9, 2
      DO 10 J = 1, 4
10    READ (2, 20) ARRAY (I, J)
```

Note that the incrementing value of the inner implied DO loop (m_3) is automatically understood to be one, since it is not explicitly stated.

Example 2: READ (2, 20) ((ARRAY(I, J), I = 1, 4), J = 1, 9, 2)

is equivalent to:

```
READ (2, 20) ARRAY(1, 1), ARRAY(2, 1), ARRAY(3, 1), ARRAY(4, 1),
             ARRAY(1, 3), ARRAY(2, 3), ARRAY(3, 3), ARRAY(4, 3),
             ARRAY(1, 5), ARRAY(2, 5), ARRAY(3, 5), ARRAY(4, 5),
             ARRAY(1, 7), ARRAY(2, 7), ARRAY(3, 7), ARRAY(4, 7),
             ARRAY(1, 9), ARRAY(2, 9), ARRAY(3, 9), ARRAY(4, 9)
```

Interpretation: Read in 20 values, storing them as follows:

1. Store the first four as the first four elements of the first column of ARRAY.
2. Store the second four as the first four elements of the third column.
3. Store the next four as the first four elements of the fifth column.
4. Store the next four as the first four elements of the seventh column.
5. Store the last four as the first four elements of the ninth column.

I/O Lists Used with Binary Tape Input or Output

When an unformatted record is read in or written out under an I/O statement of one of the following forms:

READ (i) list
WRITE (i) list

care must be taken to see that the list variables match the data items in the record. If a list is longer than the number of data items in the record, the remainder of the list variables will be read in or written out using the last value in the floating-point or integer accumulator as appropriate. This practice is not recommended.

FORMAT STATEMENT

General Form of the FORMAT Statement

When incoming data is not already in binary format and when outgoing data requires format other than binary code, a FORMAT statement must accompany the READ or WRITE statement. The FORMAT statement describes the external arrangement and type of conversion of incoming or outgoing data in terms of field specifications. Because a thorough knowledge of the FORMAT statement is essential for Fortran programming and because of the many optional forms that the statement and its specifications may have, this section contains considerable detail. A guide to the contents of the section is given in Table 5-2.

Table 5-2. FORMAT Statement Summary

	Subject and Format		Explanation	Page Reference (Definition pages are in parentheses)
FORMAT statements	n FORMAT $(S_1, S_2, \ldots, S_m)$		Simple FORMAT statement n = statement label each S_i = one field specification.	(5-12-5-13)
	n FORMAT $(S_1, S_2, \ldots, S_m/S'_1, S'_2, \ldots, S'_m/S''_1, S''_2, \ldots, S''_m///)$		Multiple-record form where a slash (/) marks the end of a unit record.	(5-13), 5-45 to 5-51
	n FORMAT $(S_1, S_2, g(S_3, S_4, S_5), S_6, g'(S_7, g''(S_8, S_9)) \ldots, S_m)$		Group repetition form where each g shows the number of times the following group of field specifications is repeated.	(5-14), 5-40 to 5-41
Conversion Codes and the Field Specifications Applicable to Each Code	A	(Aw or rAw)	Alphabetic conversion.	(5-31)
	E	(Ew.d, rEw.d, sPEw.d, sPrEw.d)	Explicit exponent conversion.	(5-21) to 5-27
	F	(Fw.d, rEw.d, sPFw.d, sPrFw.d)	Fixed-point decimal conversion.	(5-21) to 5-25
	G	(Gw.d, rGw.d, sPGw.d, sPrGw.d)	Generalized (F or E) conversion.	(5-21) to 5-28
	H	(wH)	Hollerith conversion (includes Hollerith characters: $wHh_1h_2 \ldots h_n$).	(5-31), (5-33) to 5-36
	I	(Iw or rIw)	Integer conversion.	(5-18) to 5-20
	L	(Lw or rLw)	Logical conversion.	(5-29) to 5-31
	O	(Ow or rOw)	Octal conversion.	(5-28) to 5-29
	X	wX	Blank conversion.	(5-36) to 5-37
Other Components of the Field Specification	w		Field width (used in all field specifications).	(5-16) to 5-18
	d		Decimal position indicator (used in all E, F, and G specifications).	(5-18)
	r		Field repetition constant (optional).	(5-39) to 5-41
	sP		Scale factor (optional).	(5-41) to 5-45
Other Subjects Covered in this Section	Scanning		Scanning and rescanning of FORMAT statements to satisfy I/O lists.	(5-47) to 5-51
	Carriage Control		On printer output, 1st character of 1st field specification of a unit record controls carriage.	(5-37) to 5-39
	Object-time Formatting		Reading a format description into an array at object time.	(5-51) to 5-55

The general form of the FORMAT statement is:

$$\boxed{\text{n FORMAT } (S_1, S_2, \ldots\ldots, S_m)}$$

Where:

n is an identifying statement label (in columns 1-5).

$(S_1, S_2, \ldots, S_m)$ is a list of field specifications, and each S_i is a field specification describing one of the data fields to be transmitted by an input or output statement. The order in which the field specifications are written must correspond to the sequence in which the data fields exist (or will exist) in the external medium.

Each field specification, S_i, has one of the following forms:

Ew.d	Fw.d	Gw.d	Aw	Iw	Lw	Ow	wH	wX
rEw.d	rFw.d	rGw.d	rAw	rIw	rLw	rOw		
sPrEw.d	sPrFw.d	sPrGw.d						
sPEw.d	sPFw.d	sPGw.d						

Where: Each capital letter E, F, G, A, I, L, O, H, and X is a conversion code signifying a particular type of conversion from external to internal representation (or vice versa). The codes are defined in Table 5-5 and are discussed in detail on pages 5-16 through 5-37.

w is the field width of the data field.

d specifies the position of the decimal point (if any) in the data field.

s represents an optional scale factor followed by the letter P.

r represents an optional field-repetition constant.

Figure 5-1 illustrates some of the different components of the FORMAT statement. Shown in the figure is the following hypothetical FORMAT statement, describing 10 data fields, each of which requires a different type of conversion:

```
20 FORMAT (1X, E7.0, F5.1, G15.5, A3, I5, L4, O10, 10HHOLLERITHΔ, E10.2)
```

In Examples A, B, C, and D of the figure, different points of the same statement are highlighted: A emphasizes field specifications, B the conversion codes, C the field widths, and D the decimal positions. In E, a field-repetition constant precedes each field specification which may optionally include one. In F, a scale factor precedes each field specification which may optionally contain one.

There are two variations of the general FORMAT statement: the multiple-record form and the group-repetition form. The multiple-record form punctuates field specifications with slashes. A slash is a record terminator signifying the end of a unit record. A unit record is defined as any of the following:

1. On a printer page, it is a single line of up to 131 characters.
2. On a tabulating card, it is the entire card of up to 80 characters.
3. On a magnetic tape, it is either of the following:
   a. a formatted (i.e., binary-coded-decimal) record representing a card image or printer-line image. Such a record may contain up to 132 characters.
   b. a logical record composed of any number of physical records of data in the form of its internal representation.

Examples of the multiple-record form:

```
1.   20 FORMAT (4I6, F11.4/5F9.2/-3PF8.4, 0P2F7.2, F9.1/)
2.   20 FORMAT (4I6, F11.4//5F9.2///3I7, 4E6.3/)
```

Multiple records also result when a FORMAT statement contains fewer field specifications than the number of variables in its associated I/O list. When the right parenthesis is encountered, the FORMAT specifications will be rescanned until the I/O list is satisfied. Multiple-record forms and rescanning are described in detail on pages 5-45 through 5-51.

```
20 FORMAT ( 1X , E 7 . 0 , F 5 . 1 , G15 . 5 , A 3 , I 5 , L 4 , O 1 0 , 1 0 HHOLLERITHΔ , E10.2 )
```

A. Field Specification

```
20 FORMAT (1 X , E 7 . 0 , F 5 . 1 , G 15 . 5 , A 3 , I 5 , L 4 , O 1 0 , 1 0 H HOLLERITHΔ , E 10. 2 )
```

B. Conversion Code

```
20 FORMAT ( 1 X, E 7 . 0 , F 5 . 1 , G 15 . 5 , A 3 , I 5 , L 4 , O 1 0 , 10 HHOLLERITHΔ, E 10.2 )
```

C. Field Width, w

```
20 FORMAT (1X , E 7 . 0 , F 5 . 1 , G15 . 5 , A 3 , I 5 , L 4 , O 1 0 , 1 0 HHOLLERITHΔ, E 10.2 )
```

D. Decimal Position, d

```
20 FORMAT (1X, 2 E7 . 0 , 2 F5 . 1 , 2 G15.5 , 2 A 3 , 2 I 5 , 2 L 4, 2 O 1 0 , 1 0 HHOLLERITHΔ, E10. 2)
```

E. Field-Repetition Constant

```
20 FORMAT (1X, +3P E7.0, -3P F5. 1 0P 2G15. 5, A 3 , I 5 , O 1 0, 10HHOLLERITHΔ , E10. 2)
```

F. Scale Factor

Figure 5-1. Example Highlighting Different Components of FORMAT Statement

The second variation of the FORMAT statement is the group-repetition form. This form permits repetition of one or more field specifications without rewriting each specification. A repetition constant precedes the group of field specifications, which is set off by parentheses. For example:

```
20 FORMAT (F11.2, 2(I8, F9.2))
        is equivalent to
20 FORMAT (F11.2, I8, F9.2, I8, F9.2)
```

The group-repetition form is further described on pages 5-40 and 5-41.

Every FORMAT statement must be identified by a statement label, since it will be referenced in an input or output statement (or perhaps both). The word FORMAT must be followed by a left parenthesis. The last thing in the entire statement must be a right parenthesis. Successive field specifications must be separated from each other by a field separator, i.e., a comma, slash (/), or consecutive slashes. However, the comma and slash do not have the same meaning, since each slash also indicates the end of a unit record. If a right parenthesis is followed by a comma, the comma is redundant; it may or may not be written. If it is not written, the right parenthesis will serve as the field separator. A comma should not appear before the last right parenthesis, but any number of consecutive slashes may appear there.

An error in a FORMAT statement detected during execution of a job will cause job termination and printout of the following error message:

```
ILLEGAL CHARACTER IN FORMAT STATEMENT. SEE END OF LINE BELOW.
(Next line contains the FORMAT statement up to the point of error.)
```

Contents of the Field Specification

The field specification supplies the information shown in Table 5-3 concerning a data field.

Table 5-3. Contents of the Field Specification

Information	Representation of the Information in the Field Specification	Description
Conversion Code	A single capital letter (A, E, F, G, H, I, L, O, or X as given in Table 5-5.)	Designates the type of conversion required to transform an incoming value to binary or an outgoing value to the required external representation (Table 5-5).
Conversion Field Width	w For E, F, G, I, and O conversions, $w \leq 32$. For A, H, and X conversions, $w \leq$ unit record length, u, where: u = 80 for punched cards 132 for BCD tapes 131 for printer lines	Indicates the total number of columns to be used as a data field for a single conversion.
Decimal Position Indicator	d $(0 \leq d \leq 31)$	Indicates the number of places after the decimal point for conversions involving decimals, i.e., E, G, and F.
Field-Repetition Constant	r $(0 \leq r \leq 63)$	Indicates the number of times that the field specifications which follow are to be repeated.
Scale Factor	sP (s = a signed or unsigned integer is always followed by the capital letter P)	Indicates that the decimal point of the incoming and outgoing data is to be shifted right or left.

Every field specification must indicate the type of conversion involved, using the appropriate conversion code, and must also indicate the width of the field. In addition, all data represented internally as floating-point decimal (i. e., E, F, and G codes) must indicate the decimal position. Field-repetition constants and scale factors are optional information applicable only to certain conversion codes. In all, there are three general forms of field specifications as indicated in Table 5-4.

Table 5-4. Field Specification Formats

E, F, or G Conversion				A, I, L, or O Conversion		H or X Conversion	
E	w	.	d	A	w	w	H
F	w	.	d	I	w		
G	w	.	d	L	w	w	X
				O	w		
Conversion Code	Field Width	Decimal Point	Decimal Position Indicator	Conversion Code	Field Width	Field Width	Conversion Code
Scale factor and/or field-repetition con-constant may precede conversion code.				Field-repetition constant may precede the conversion code.		No options.	

Field specifications written without scale factors or field-repetition constants are called basic field specifications. The contents of basic field specifications are described in the following paragraphs.

Conversion Codes

There are nine conversion codes used in Fortran D language, which are defined in Table 5-5. The general form of field specification applicable to each conversion code is also given in the table.

Table 5-5. Conversion Codes

Data Type to be Converted	Code Used in Field Specification	External Representation of Data	Internal Representation of Data	General Forms of the Field Specification
Alphabetic	A	Characters of Fortran set	Fixed-point binary equivalent of external representation.	Aw rAw
Explicit exponent	E	Real	Floating-point decimal	Ew.d rEw.d sPrEw.d sPEw.d
Fixed-point decimal	F	Real, without explicit exponent	Floating-point decimal	Fw.d rFw.d sPrFw.d sPFw.d
Generalized	G	Real, with or without exponent	Floating-point decimal	Gw.d rGw.d sPrGw.d sPGw.d
Hollerith	H	Characters of Fortran set	Fixed-point binary equivalent of external representation.	wH
Integer	I	Integer	Fixed-point binary	Iw rIw
Logical	L	"T" or "F"	Fixed-point binary, using only the rightmost character of the field.	Lw rLw
Octal	O	Octal integer	Fixed-point binary equivalent of external representation.	Ow rOw
Blank or Skip	X	Not applicable	Not applicable	wX

Conversion Field Width

The field width for a single conversion, w, represents the total number of columns or positions assigned to a single datum. For all real conversions (E, F, and G) and for integer and octal conversions (I and O), the conversion field width cannot exceed 32. Hollerith, alphabetic, and blank conversions (H, A, and X) may be assigned a number of positions up to the limit for the unit record of the external medium. The unit record limit for punched cards is 80, for BCD tapes 132, and for printed lines 131. Every consecutive position of the external medium from column 1 through the last column used must be considered as part of the data field, including blanks.

Every distinct input value is said to occupy one data field of the input medium, regardless of how many card columns the value requires. One or several data fields can occupy a single punched card. Different values on the same card can be of different data types. In Figure 5-2, six data fields are shown; two are integer and four are real.

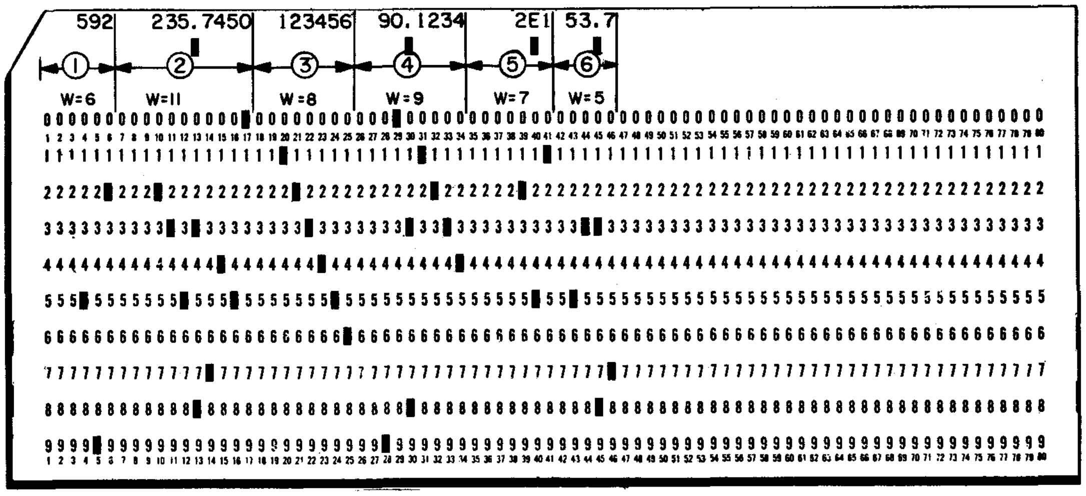


Figure 5-2. Data Fields and Field Widths

The limits of conversion field width are, in general, set for a larger number of positions than a single datum can occupy. When converting data to output format, only data converted under H and X conversion codes can occupy the full conversion field width of up to unit-record length. All other data are limited by the internal fixed-point or floating-point precision set by the programmer for a job and described in detail in Appendix C. If the conversion field width has additional positions, they will be filled with blanks. The representation of output is shown for each conversion code on pages 5-19 to 5-31. If the conversion field width is less than that required by the datum, an overflow condition will occur. Overflow conditions are also described for each conversion code on pages 5-19 to 5-31.

The programmer can specify the precision of an internal, floating-point number as between 2 and 20 characters in the mantissa, or he can permit automatic assignment of precision of seven characters. The number of digits in the mantissa of a real number being converted to output form under E, F, or G conversion can therefore be between 2 and 20, depending upon the floating-point precision. On output, real data are right justified within their allotted conversion field widths.

A logical datum is stored internally only in the low-order six bits of a location and appears on output as either T or F, right justified in the conversion field.

An octal datum is stored internally in a fixed-point field. From 6 to 24 octal digits can be stored in from 3 to 12 characters. Octal digits appear on output left justified in the conversion field.

An alphabetic datum is stored internally in a fixed-point field, which the programmer can specify between 3 and 12 characters. If not specified by the programmer, a precision of three characters is assigned. Alphabetic characters appear on output left justified in the conversion field.

An integer datum is stored internally in binary in a fixed-point field. From 5 to 20 integer digits can be stored internally in binary in from 3 to 12 characters. On output, integer digits are right justified in the conversion field width.

Decimal Position Indicator

The number of places to the right of a decimal point is expressed by d, an unsigned decimal number greater than or equal to 0 and less than or equal to 31. Decimal position indicators are expressed for all E, F, and G conversions. Use of a decimal point in an incoming datum for E, F, or G conversion is optional, since the decimal position indicator will specify its position. However, if a decimal point is expressed in the datum, its position in the datum will determine the value stored in memory. The decimal position indicator in the FORMAT specification, if different from the actual decimal point, will be ignored.

Basic Field Specification for Integer Conversion

The basic form of the Integer Conversion is:

$\boxed{\text{I w}}$

INPUT

Used in conjunction with an input statement, an Iw field specification converts an incoming integer to internal, fixed-point binary form. Precision of integer data is from 3 to 12 characters (5 to 20 digits), as described in Appendix C. When a minus sign precedes an integer, a space must be allotted in the field width for the sign. Use of a plus sign is optional and no space need be allotted for the positive sign. When a sign is included, it may be followed immediately by the integer or by any number of blanks, then by the integer. Any embedded or trailing blanks in the incoming integer data field will be stored as zeros. Note that Honeywell uses an upper case delta (Δ) to denote a blank.

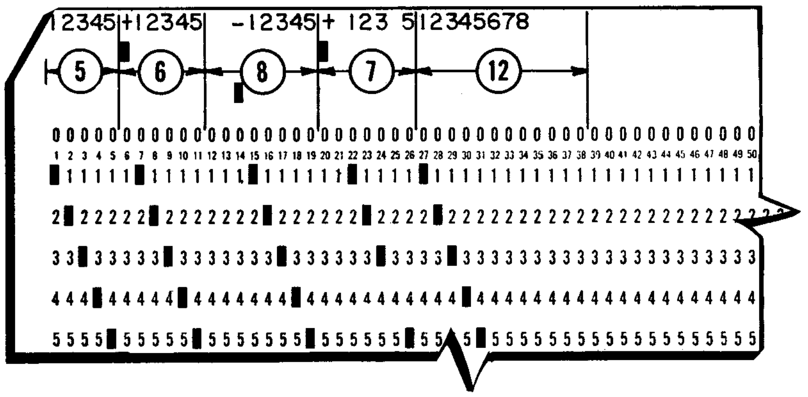
Examples:

Incoming Integer	Field Width	Field Specification	Decimal Representation of Value Stored Internally
12345	5	I5	+12345
+12345	6	I6	+12345
ΔΔ-12345	8	I8	-12345
+Δ123Δ5	7	I7	+12305
12345678ΔΔΔΔ	12	I12	+123456780000

The five integers in the preceding examples can be read into memory and converted to internal form by means of the READ and FORMAT statements shown in Figure 5-3.

An illegal character in the incoming integer data will cause termination of the job. An illegal character is any character not 0 through 9, a blank, or an initial plus or minus. An illegal data character causes the following printout:

ILLEGAL CHARACTER IN INPUT DATA. BAD RECORD IS PRINTED BELOW.
(Next line shows the contents of the bad record.)



```
      READ(2,20) I, J, K, L, M
   20 FORMAT   (I5,I6,I8,I7,I12)
```

Figure 5-3. Input of Integer Data

OUTPUT

Used in conjunction with an output statement, an Iw field specification causes conversion from internal, fixed-point binary form to an external integer. In the output data field, digits are right justified on a background of blanks when the field width is wider than necessary to accommodate all the characters. If the value is negative, a minus sign immediately precedes the number. Positive integers appear without the plus sign. However, a space is allowed for the sign, whether plus or minus, when determining the field width of the output data field.

Examples:

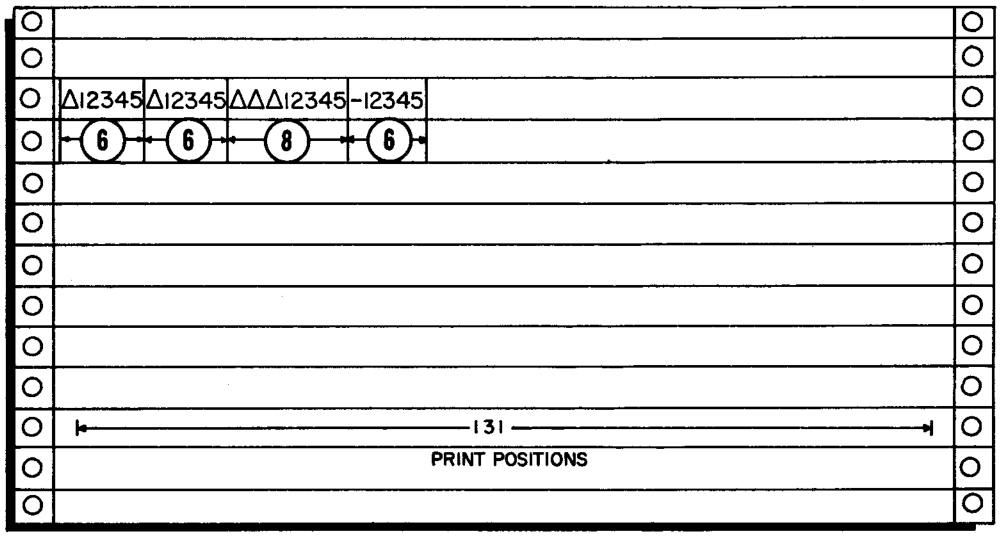
Integer Variable	Value Stored Internally	Field Specification	Presentation on Output Medium
I	+12345	I7	ΔΔ12345
J	+12345	I6	Δ12345
K	+12345	I8	ΔΔΔ12345
L	-12345	I6	-12345

The values stored internally in I, J, K, and L in the preceding examples can be converted to integer form and printed on the on-line printer by means of the WRITE and FORMAT statements shown in Figure 5-4. Note that the first field specification is I7 rather than I6. This value allows a blank first character for carriage control when printing. Carriage control is explained on pages 5-37 through 5-39.

If an outgoing integer requires more columns of the output medium than the allocated field width permits, truncation occurs at the low-order end of the integer. An asterisk is automatically inserted as the first character of the output field to indicate that truncation has occurred; the asterisk is followed by as many high-order digits as will fit in the remainder of the field with a negative sign, if present.

Examples:

	1	2
Values Stored Internally:	+12345678	-12345678
Minimum Field Width Required in Output Medium:	w = 8	w = 9
Field Width Actually Allocated in FORMAT Statement:	w = 7	w = 7
Presentation on Output Medium:	*123456	*-12345



```
    WRITE (3,20) I, J, K, L
20  FORMAT      (I7,I6,I8,I6)
```

Figure 5-4. Output of Integer Data

Input for Conversion of All Real Data

There are three FORMAT field specifications for the conversion of incoming real data. These are:

$\boxed{\text{Fw. d}}$

$\boxed{\text{Ew. d}}$

$\boxed{\text{Gw. d}}$

On input, all three conversions are performed in a similar manner. An Fw. d, Ew. d, or Gw. d causes conversion to internal floating-decimal form of an incoming real constant. Page 1-13 defines and describes real constants. Note that such constants may appear with or without exponents.

Use of a plus sign in incoming data is optional. Any blanks embedded in the mantissa of the constant are considered to be zeros. When an incoming constant has an exponent, the exponent is of the general form E±ee, where ee is the numeric exponent. However, several simplifications are permitted for convenience in keypunching input data, and the figure below shows equivalent ways of punching the exponent <u>plus</u> <u>two</u>. Blanks appearing in the exponent have no effect, since they are suppressed. A positive exponent may have its plus sign omitted or replaced with a blank. If the first digit of the exponent is zero, it may be omitted. If the exponent appears with a sign, the E may be omitted (as in the last two rows of Figure 5-5). The exponent need not be right justified in the input field.

E+02	E + 0 2	E +02	E+ 02	E+0 2
E02	E 0 2	E 02	E0 2	
E+2	E + 2	E +2	E+ 2	
E2	E 2			
+02	+ 0 2	+ 02	+0 2	
+2	+ 2			

Figure 5-5. Twenty-One Equivalent Ways of Keypunching an Exponent of Plus Two

The field width, w, is determined by counting the number of characters in the incoming datum. The following formula indicates how w is determined:

$$w = a + p + n + e + b$$

Where: a = the number of digits in the mantissa. For F format this would mean all digits in the datum.

p = 1 if a decimal point is present.
 = 0 if a decimal point is not present.

n = 1 if the sign of the mantissa is minus or is punched plus.
 = 0 if the sign of the mantissa is positive but not punched.

Where: e = 4 for any real datum with an exponent as follows:

1 character for E,
1 character for sign of the exponent,
2 characters for the exponent.

= 0 for any real datum without an exponent.

b = the number of leading or embedded blanks.

The decimal position indicator, d, represents the number of digits following the decimal point in the incoming real datum. The decimal point need not be present in the incoming datum; in this case, the d of the FORMAT specification will determine the position of the decimal point of the value stored in memory. If a decimal point is present in the incoming datum, the decimal position of the value stored in memory will be determined by the datum, not by the d of the FORMAT specification.

Following are a number of examples of correctly formatted input data.

Examples:

Real Input Value	Minimum Field Width, w	Decimal Position, d	Field Specification	Decimal Representation of Value Stored Internally
12345.12345	11	5	F11.5	+12345.12345
⊢ w = 11 ⊣ ⊢d=5⊣				
+234.	5	0	F5.0	+234.
ΔΔΔΔ-67.1234	12	4	F12.4	-67.1234
123Δ5.1Δ345	11	5	F11.5	+12305.10345
+12.34E02	9	2	E9.2	+1234.
-123456+02	10	4	E10.4	-1234.56
1234.567E+02	12	3	G12.3	+123456.7

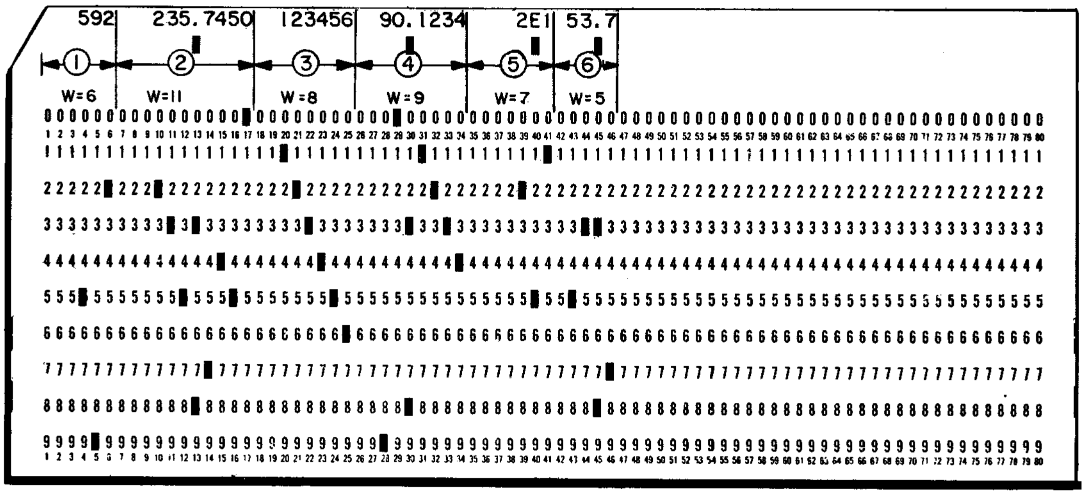
Note that any of the examples above could have an E, F, or G input specification. The following example illustrates this.

Real Input Value	Minimum Field Width, w	Decimal Position, d	Possible Field Specifications	Decimal Representation of Value Stored Internally
1234.567E+02	12	3	E12.3 or F12.3 or G12.3	+123456.7

Thus, F, E, and G conversions can be used interchangeably on input, provided that sufficient field width is allowed. Note in the next-to-last example given above that the decimal position indicator, d, has determined the storage of the input datum in the absence of a decimal point in the value.

The compiler mantissa parameter, F, determines the precision or number of significant digits which may be used on input. When a datum containing more than F significant digits is encountered on input of real data, only the high-order F digits will be stored in the mantissa. The remaining low-order digits will be ignored except in determining the proper exponent value of the datum. The programmer may set the mantissa parameter F on the *JOBID card at compile time within the range of 2 to 20 digits. If no parameter is specified, the compiler assumes F = 7. When the mantissa contains fewer significant digits than F, the incoming real datum is stored left justified with a fill of zeros.

Figure 5-6 repeats the incoming data card previously shown on page 5-17. Included in the figure are an appropriate I/O statement and a FORMAT statement with correctly formulated field specifications. All fields are real with decimal position indicators giving the decimal point where appropriate.



```
      READ (2, 20) A, B, C, D, E, F
20    FORMAT (F6.0, E11.4, G8.3, F9.4, E7.0, G5.1)
```

Decimal Representation of Values Stored Internally:

+592., +235.7450, +123.456, +90.1234, +20., +53.7

Figure 5-6. Input of Real Data

An illegal character in incoming real data will cause termination of the job. An illegal character is any character not 0 through 9, a blank, a plus or minus, a decimal point, or an E. The following diagnostic is printed out:

ILLEGAL CHARACTER IN INPUT DATA. BAD RECORD IS PRINTED BELOW.
(Next line shows the contents of the bad record.)

Output Conversion to Fixed-Point Decimals (Fw.d)

Used in conjunction with an output statement, an Fw.d field specification causes conversion from internal floating-decimal form to the form of a real constant expressed without an exponent (i.e., fixed-point decimal form).

In the output field, the value is right justified on a background of blanks when the field width is wider than necessary to accommodate all characters of the value. If the value is negative, a minus sign immediately precedes the value. Positive values appear without a plus sign, but a character position must be allowed for the sign. On Fw.d output conversion, the field width, w, is determined by the following formula:

$$w \geq a + d + 2$$

Where: a = the number of digits before the decimal point,

d = the number of digits after the decimal point,

2 columns are allotted for decimal point and sign.

The following examples illustrate the rules for determining the minimum field width for output under Fw.d field specifications.

Examples:

Decimal Representation of Value Stored Internally	Minimum Field Width for Output	Field Specification
+12345.12345	w = 12 (w = 5 + 5 + 2)	F12.5
-.123	w = 5 (w = 0 + 3 + 2)	F5.3
-23.1234	w = 8 (w = 2 + 4 + 2)	F8.4
+234.	w = 5 (w = 3 + 0 + 2)	F5.0

To transmit these values from memory, the following output and FORMAT statements can be used:

```
      WRITE (3, 20) A, B, C, D
20    FORMAT (1X, F12.5, F5.3, F8.4, F5.0)
```

The line would be printed as follows, starting in column 1:

```
      Δ12345.12345|-.123|-23.1234|Δ234.
```

If an outgoing value requires more columns of output medium than the allocated field width permits, an asterisk is set in the first column of the output field. If the field width, w, was greater than or equal to 7, the output value will have the format E(w-1).(w-7). When such overflow occurs, the least significant field position is rounded. That is, if the digit to the immediate right of the least significant position is five or more, one is added to the least significant position; otherwise the number in that position remains the same. The following is an example of overflow and rounding of a value having a field width of 7 or greater.

Value Stored Internally	Field Specification	Actual Field Specification Used	Presentation on Output Medium
-12345.12345	F11.5	E10.4	*-.1235E+05

If the field width, w, is less than 7, blanks will follow the asterisk in the output field. For example:

Value Stored Internally	Field Specification	Presentation on Output Medium
+1.23	F3.2	*ΔΔ

The number of places that will appear at the right of the decimal point is specified by d in Fw.d. The following examples show the effect of varying d and w. The same real constant is transmitted from memory using different Fw.d field specifications.

Examples:

Value Stored Internally	Output Field Specification	Presentation on Output Medium
+2.53	F5.2	Δ2.53
+2.53	F8.5	Δ2.53000
+2.53	F10.5	ΔΔΔ2.53000
+2.53	F4.1	Δ2.5
+2.53	F3.2	*ΔΔ
+2.53	F9.8	*.253E+01
+2.53	F10.9	Δ*.253E+01

Note that in the last three examples if the value stored internally had been +.253, the output field specification would have been great enough to permit transmission of the value under F conversion.

Output Conversion to Explicit Exponent (Ew.d)

Used in conjunction with an output statement, an Ew.d field specification causes conversion from internal floating-decimal form to the form of a real constant expressed with an exponent. The output form is shown in Figure 5-7. It consists of a sign followed by the mantissa and exponential part.

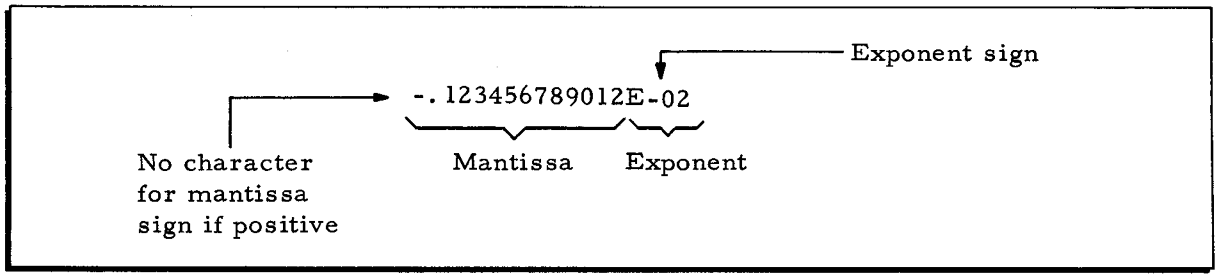


Figure 5-7. Output of Real Data in Exponential Form

When an output value is in explicit exponent form, the mantissa is given as a decimal fraction preceded by a decimal point and sign if negative. The number of significant digits in the mantissa is specified between 2 and 20 by the mantissa parameter, F, on the *JOBID card. If not specified, an output value can have up to 10 significant digits in the mantissa.

The exponential part consists of the letter E, followed by a sign, then by a two-digit exponent representing the power of ten by which the mantissa is multiplied.

The formula for determining the minimum field width for output is as follows:

$$w = m + 6$$

Where: m is the number of digits in the mantissa, and

6 spaces are allotted as follows

1 for mantissa sign
1 for decimal point
1 for the letter E
1 for the exponent sign
2 for the exponent

The value is right justified on a background of blanks when the field width is wider than necessary to accommodate all characters of the value. Following are examples of some correctly formatted output values using E conversion.

Examples:

Value Stored Internally	Minimum Field Width for Output	Field Specification	Presentation on Output Medium
$-.1234 \times 10^{-6}$	10 (4 digits in decimal fraction + 6)	E10.4	-.1234E-06
$-.12345 \times 10^{4}$	11 (5 digits in decimal fraction +6)	E11.5	-.12345E+04
$+.123456789012 \times 10^{12}$	18 (12 digits in decimal fraction + 6)	E18.12	.123456789012E+12

To transmit these values from memory, the following output and FORMAT statements can be used:

```
         WRITE (3, 20) A, B, C
   20    FORMAT (1X, E10.4, E11.5, E18.12)
```

The line would be printed as follows starting in column 1:

```
         -.1234E-06|-.12345E+04|Δ.123456789012E+12
```

If the field width, w, is less than m + 6, an asterisk is set in the first column of the output field. When such overflow occurs, the output consists of as many high-order digits as the field width can accommodate. The digit in the least significant position is rounded. For example:

Value Stored Internally	Field Specification Used	Presentation on Output Medium
$-.3214892\text{x}10^6$	E11.7	*-.3215E+06

A field specification of E13.7 would correct the difficulty.

Generalized Field Specification, Gw.d

Used with an output statement, a Gw.d field specification causes conversion from internal, floating-decimal form to a real constant. The magnitude of the real constant will determine whether the Gw.d is interpreted as an F or E conversion.

Comparison between the exponent of the stored value, e, and the number of decimal places, d, of the specification determines the type of conversion the compiler will use as follows:

1. If $e > d$, E conversion is used.
2. If $e \leq d$, F conversion is used according to the formula: F(w-4).(d-e), 4X
   Four blanks (4X) are appended to the right of the value.
3. If the value to be represented is less than $|.1|$, E conversion is always used.

Following are some correctly formatted output values using G conversion, which will indicate how the conversion formulae determine the output presentation.

Examples:

Given a field specification of G14.6.

Value Stored Internally	Conversion	Presentation on the Output Medium
$.12345123 \times 10^0$	F	ΔΔΔ.123451ΔΔΔΔ
$.12345123 \times 10^4$	F	ΔΔΔ1234.51ΔΔΔΔ
$.12345123 \times 10^8$	E	ΔΔΔ.123451E+08
$.12345123 \times 10^{10}$	E	ΔΔΔ.123451E+10

If the programmer does not allow sufficient field width, the rules for E output conversion under overflow conditions will apply. The following examples illustrates these conditions.

Examples:

The programmer selected a specification of G12.8 for output of the values below. In each case, w< m+6.

Value Stored Internally	Conversion	Presentation on the Output Medium
$-.12345123 \times 10^{8}$	E	*-.12345E+08
$-.12345678 \times 10^{10}$	E	*-.12346E+10

Basic Field Specification for Octal Conversion

Octal conversion has the basic form:

$\boxed{\text{O w}}$

INPUT

Used in conjunction with an input statement, an Ow field specification causes conversion of an incoming octal integer to internal, fixed-point binary representation. The incoming integer consists of digits in the range 0 to 7. Any characters other than the digits 0 through 7 are illegal and will cause termination of the object program at execution time. Embedded and trailing blanks are considered to be zero; leading blanks are ignored. Internally, octal data appear not as a value but as a string of octal characters, left-justified with a fill of zeros, since two octal characters occupy one character of a fixed-point field.

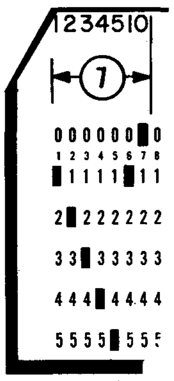
Example:

Incoming Octal Integer: 1234510

Internal Representation:

$$\underbrace{001}_{1}\quad \underbrace{010}_{2}\quad \underbrace{011}_{3}\quad \underbrace{100}_{4}\quad \underbrace{101}_{5}\quad \underbrace{001}_{1}\quad \underbrace{000}_{0}\quad 000\quad 000\quad 000\quad 000\quad 000\quad 000\quad 000$$

The value shown in the example can be read into memory and converted to internal form by means of the READ and FORMAT statement shown in Figure 5-8. Note that octal input data are identified as having an integer data type, since there is no way of declaring octal type in a data-type statement.



```
READ (2, 20)L
20 FORMAT(O7)
```

Figure 5-8. Input of Octal Datum

OUTPUT

Used in conjunction with an output statement, an Ow field specification causes conversion from internal representation to the form of an octal integer. Conversion takes place from left to right and proceeds until w octal digits have been converted.

Example:

Internal Representation

$\underbrace{001}_{1}\ \underbrace{010}_{2}\ \underbrace{011}_{3}\ \underbrace{100}_{4}\ \underbrace{101}_{5}\ \underbrace{001}_{1}\ \underbrace{000}_{0}\ 000\ 000\ 000\ 000\ 000\ 000\ 000$

Field specification for output: O7

Presentation on output medium: 1234510

Basic Field Specification for Logical Conversion

Logical conversion has the basic form:

$\boxed{Lw}$

INPUT

Used in conjunction with an input statement, an Lw specification causes an incoming truth value (true or false) to be converted to binary representation (zeros for false and ones for true). The first non-blank character in the input data field determines the resulting truth value. If the first non-blank character is T, a value of true will be stored. Any other character will result in a value of false being stored. It is recommended that the letter F be written as the first non-blank character if a value of false is desired.

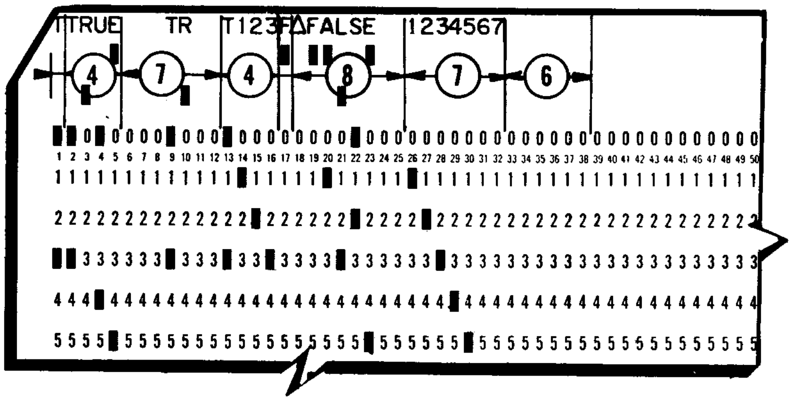
Examples:

Contents of Input Data Field	Field Specification for Input	Truth Value Stored Internally
T	L1	TRUE
TRUE	L4	TRUE

Examples (cont):

Contents of Input Data Field	Field Specification for Input	Truth Value Stored Internally
ΔΔΔTRΔΔ	L7	TRUE
T123	L4	TRUE
F	L1	FALSE
ΔFALSEΔΔ	L8	FALSE
1234567	L7	FALSE
ΔΔΔΔΔΔ	L6	FALSE
.TRUE.	L5	FALSE

The nine truth values of the preceding examples can be read into memory and converted to internal form by means of the READ and FORMAT statements shown in Figure 5-9.



```
   READ (2,20)   I, J, K, L, M, N, II, JJ,KK
20 FORMAT        (L1,L4,L7,L4,L1,L8,L7,L6,L5)
```

Figure 5-9. Input of Logical Data

OUTPUT

Used in conjunction,with an output statement, an Lw field specification converts the binary representation of a truth value to either the letter T (if TRUE) or F (if FALSE), right-justified in the output data field.

Examples:

Truth Value Stored Internally	Field Specification for Output	Presentation on Output Medium
TRUE	L1	T
FALSE	L3	ΔΔF
TRUE	L6	ΔΔΔΔΔT

Basic Field Specification for Alphabetic Conversion

Alphabetic conversion has the basic form:

$\boxed{\text{A w}}$

INPUT

Used in conjunction with an input statement, an Aw field specification causes characters up to the number permitted by the size of the fixed-point field of incoming alphabetic data to be stored internally; each character is stored as six bits. Valid input includes any character of the Honeywell character set; blanks are significant. Since there is no alphabetic type declaration, alphabetic data are identified as having an integer data type.

Example:

Assume that no fixed-point parameter has been specified and that it is desired to read into memory the complete English sentence:

THERE IS NO ALPHABETIC TYPE DECLARATION.

Three variations on the method are shown in Figure 5-10. A declared integer precision of 5 is assumed, and the sentence is divided into data fields of five alphanumeric characters. In methods 1 and 2, a separate variable is used to store the contents of each data field. The contents of the variables after execution of the READ statement are shown at the right of the illustration.

In method 3, the sentence is stored in an array. The array size is declared in a DIMENSION statement; an implied DO loop in the READ statement eliminates the need of writing out the names of all eight array elements in a simple list; a repetition constant is used in the FORMAT statement as a shortcut in writing the statement. The contents of each array element after execution of the READ statement are shown in the figure.

OUTPUT

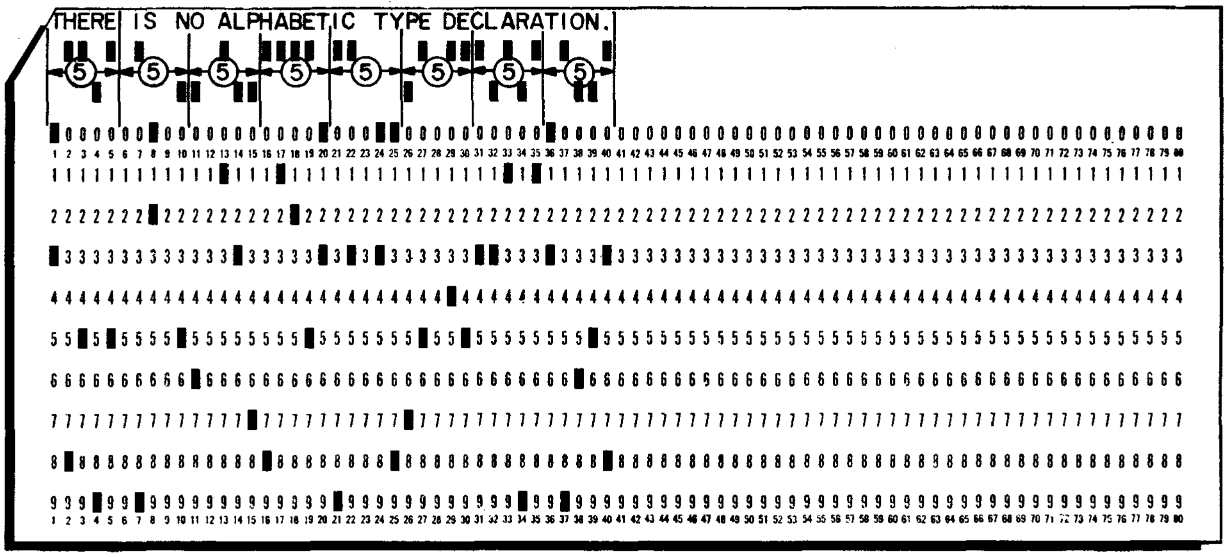
Used in conjunction with an output statement, an Aw field specification causes alphabetic data stored internally as six-bit characters to be converted to their equivalent forms in the Honeywell character set and transmitted to the external medium.

Field Specification for Hollerith Data

The Hollerith field specification has the form:

$\boxed{wHh_1h_2h_3...h_n}$

Where: wH are the field specification's field width and conversion code; and each h is a Hollerith character.



Method 1: Eight variables are used for storage.

```
    READ (2, 20) I, J, K, L1, L2, L3, M, N
20  FORMAT (A5, A5, A5, A5, A5, A5, A5, A5)
```

If the incoming contents of N were less than five alphabetic characters, the contents would be left justified.

Variable	Contents after Execution
I	THERE
J	ΔISΔN
K	OΔALP
L1	HABET
L2	ICΔTY
L3	PEΔDE
M	CLARA
N	TION.

Method 2: Same as method 1 except that a repetition constant is used to simplify writing field specifications (see page 5-39).

```
    READ (2, 20) I, J, K, L1, L2, L3, M, N
20  FORMAT (8A5)
```

Method 3: A one-dimensional array, containing eight array elements, is used for storage. Incoming characters of the last array element will be left justified if less than five.

```
    DIMENSION IALPH (8)
    READ (2, 20)(IALPH(I), I = 1, 8)
20  FORMAT (8A5)
```

Array Element	Contents after Execution
IALPH (1)	THERE
IALPH (2)	ΔISΔN
IALPH (3)	OΔALP
IALPH (4)	HABET
IALPH (5)	ICΔTY
IALPH (6)	PEΔDE
IALPH (7)	CLARA
IALPH (8)	TION.

Figure 5-10. Input of Alphabetic Data

The Hollerith field specification, therefore, differs from discussed specifications in that the item to be transmitted appears in the FORMAT statement itself, not on a data card. If only Hollerith characters are to be transmitted, the appropriate I/O statement does not need a list. Hollerith characters may be any of the Honeywell character set with blanks being significant.

OUTPUT

Used in conjunction with an output statement, this field specification provides the basic means of supplying appropriate headings, titular information, and vertical line-spacing for output reports. If the output device is a card punch, w should not exceed 80; if it is a magnetic tape unit, w should not exceed 132; if it is a printer, w should not exceed 131.

To illustrate a basic use of the Hollerith field specification, assume that it is desired to print the heading "POWER CALCULATIONS". This could be done with the following WRITE and FORMAT statements:

```
WRITE (3, 20)
20 FORMAT (19HΔPOWERΔCALCULATIONS)
```

At compilation time, the 19 characters composing the phrase ΔPOWERΔCALCULATIONS become an integral part of the program and will be stored in memory. When the WRITE statement is actually executed in the object program, the characters are transmitted to the on-line printer and printed in the first 18 columns of a line. The initial blank in the Hollerith specification is used for carriage control. (See pages 5-37 to 5-39.)

To center the heading on the page, the wX specification may be used to insert blanks before the Hollerith field. Blanks may be inserted through the Hollerith specification itself, as shown below, but this method is more cumbersome:

```
WRITE (3, 20)
20 FORMAT (26HΔΔΔΔΔΔΔΔ POWER CALCULATIONS)
```

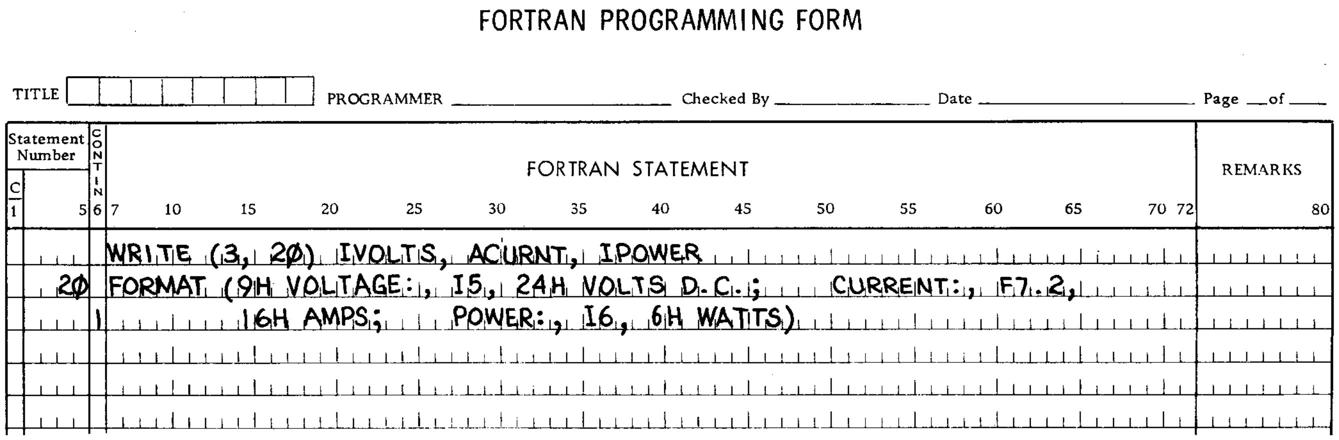
In effect, printing will start in column 8 instead of column 1, since seven blanks will precede the phrase POWER CALCULATIONS.

Like all field specifications, Hollerith specifications may be interspersed with other specifications in a FORMAT statement, as illustrated by the following example.

Example:

It is desired to print three values on the same line of an output report. The first value represents a voltage, the second a current, and the third a power calculation. For clarity, it is desired to label each calculation and to indicate the units in which it is expressed.

The WRITE and FORMAT statements shown in Figure 5-11 illustrate how all this information could be printed on the same line.


FORTRAN PROGRAMMING FORM

TITLE PROGRAMMER Checked By Date Page of

Statement Number	CONT	FORTRAN STATEMENT	REMARKS
		WRITE (3, 2Ø) IVOLTS, ACURNT, IPOWER	
2Ø		FORMAT (9H VOLTAGE:, I5, 24H VOLTS D.C.; CURRENT:, F7.2,	
	1	16H AMPS; POWER:, I6, 6H WATTS)	

Figure 5-11. Example of Output of Hollerith Data

The line would be printed as follows in columns 1 through 72:

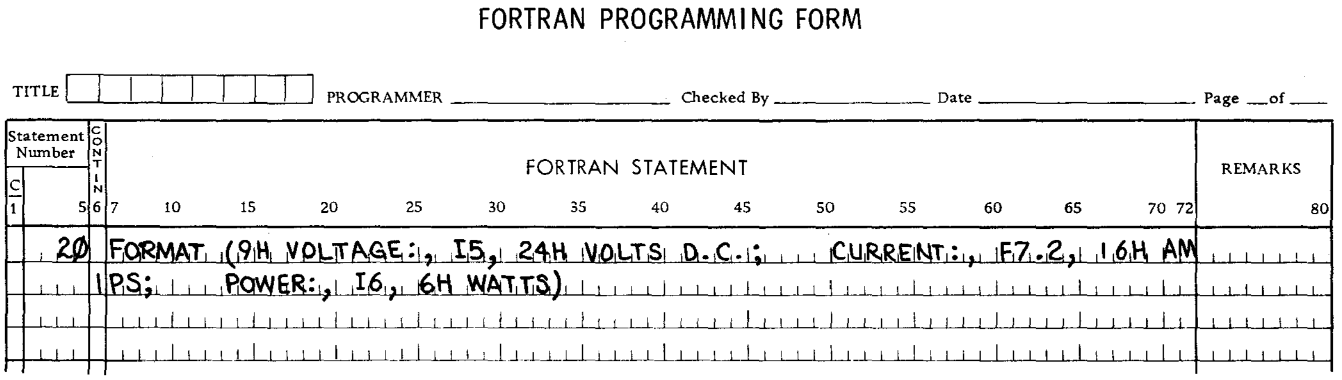
```
VOLTAGE:  115 VOLTS D. C. ;          CURRENT: 13.04 AMPS;         POWER:  1500 WATTS . . .
```

The same line is shown below with the blank spaces marked and the starting column of each data field indicated.

```
VOLTAGE:ΔΔ115ΔVOLTSΔD. C. ;ΔΔΔΔ CURRENT:ΔΔ13.04ΔAMPS;ΔΔΔΔPOWER:ΔΔ1500Δ WATTS
↓        ↓   ↓                          ↓     ↓                   ↓    ↓    ↓
1        9   14                         38    45                  61   67   72
```

The preceding example illustrates the scanning process that is discussed on pages 5-47 through 5-49 . The FORMAT statement is scanned from left to right. The first field specification encountered is the Hollerith one, 9H VOLTAGE:. The first space is used for printer carriage control and the eight literal characters following are transmitted from memory to the output medium. Scanning resumes without any list item having been transmitted. Next, the I5 field specification is matched with the integer variable IVOLTS in the output list, and the present value of IVOLTS (assumed to be 115) is transmitted from memory to the output medium. Scanning of the FORMAT statement continues. Now, the 24 characters specified in the next Hollerith specification are transmitted, and scanning resumes without any additional list item having been transmitted. Then, the F7.2 specification is matched with real variable ACURNT, and the present value of this variable (assumed to be 13.04) is transmitted, and so on.

When Hollerith field specifications are written in FORMAT statements, it may often be necessary to use one or more continuation lines to express the complete FORMAT statement, as in the preceding example. It is permissible for a Hollerith field specification to be divided between lines of a statement, since column 7 of a continuation line follows immediately after column 72 of the preceding line. Thus, the FORMAT statement of the preceding example could be written as shown in Figure 5-12, with a Hollerith field specification split between lines.



FORTRAN PROGRAMMING FORM

TITLE ____ PROGRAMMER ____ Checked By ____ Date ____ Page ___of ___

Statement Number	CONTIN	FORTRAN STATEMENT	REMARKS
20		FORMAT (9H VOLTAGE:, I5, 24H VOLTS D.C.; CURRENT:, F7.2, 16H AM	
	1	PS; POWER:, I6, 6H WATTS)	

Figure 5-12. Use of Continuation Line with Hollerith Specification

Note that in the original example, it does not matter where the continuation line starts, since no Hollerith field specification is split between lines, and blanks are normally suppressed if they are not part of a Hollerith field specification. In the second example, however, it is very important that the continuation line begin in column 7 if the results are to be identical, since blanks are significant when part of a Hollerith specification.

INPUT

When a FORMAT statement containing Hollerith data is referenced by an input statement, the actual characters listed in the Hollerith field specification are replaced by whatever characters appear in the corresponding field of the input record. If the same FORMAT statement is later used with an output statement, the replacement characters rather than the original Hollerith data will be transferred to the output record. By this means, titling information, such as the current date, may be conveniently changed from run to run, as shown in the following example.

Example:

Assume that a program contains the following FORMAT statement:

```
20 FORMAT (1X, 8HMM/DD/YY)
```

At object time, the current date, expressed as 11/22/65 and punched in columns 2-9 of an input card, is read into the memory area occupied by MM/DD/YY by means of the statement:

```
READ (2, 20)
```

Now, if the statement:

```
WRITE (3, 20)
```

is executed, the current date, 11/22/65, will be printed in the first eight print positions instead of MM/DD/YY.

It should be emphasized, however, that the Hollerith data replacing the original Hollerith characters are still not available to the programmer for use in any way other than for input or output. (The Aw field specification may be used to enter alphabetic data which can then be manipulated by the program.)

Another useful application of the Hollerith field specification is in controlling the vertical spacing of lines of printing. For a complete discussion of carriage control and of the use of the Hollerith specification for carriage control, see pages 5-37 to 5-39.

Field Specification for Blank Conversion

Blank conversion has the form:

$\boxed{wX}$

INPUT

On input, the field specification wX causes w columns of the input record to be bypassed. Any information contained in the skipped columns is disregarded, never being transmitted to memory. No associated name is required (or should be written) in the list of the READ statement which references the FORMAT statement containing the wX, since this particular field specification is not matched with any corresponding list item.

Example:

Six numerical values are punched on each card of a deck of input data. The six data fields per card have widths as shown in the sample card of Figure 5-13.

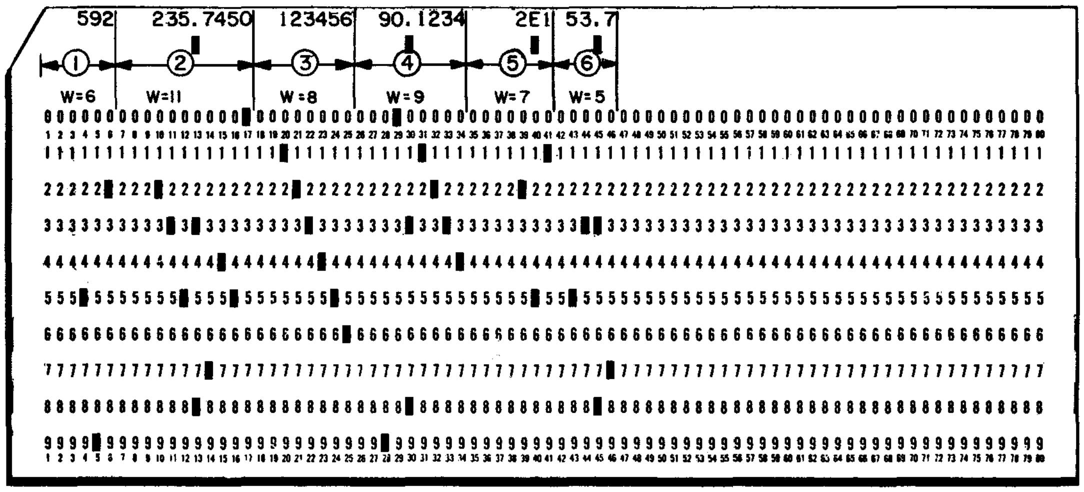


Figure 5-13. Sample Input Card

Assume that this particular object program does not need to process the data punched in field No. 2 (columns 7-17) or field No. 4 (columns 26-34) which have field widths of 11 and 9 columns, respectively. The two fields may be skipped by using the wX field specification while obtaining values for the other four fields in the manner:

```
      READ (2, 20) I, J, A, B
   20 FORMAT (I6, 11X, I8, 9X, E7.0, F5.1)
```

OUTPUT

On output, wX causes w blanks to be inserted into the output record. No associated name is required (or should be written) in the list of the output statement which references the FORMAT statement containing wX, since this particular field specification is not matched with any corresponding list item.

Example 1:

It is desired to print six blank columns before the phrase POWER CALCULATIONS. This can be done by means of the following WRITE and FORMAT statements:

```
      WRITE (3, 20)
   20 FORMAT (7X, 18HPOWER CALCULATIONS)
```

The line would be printed as follows in columns 1 through 24:

ΔΔΔ ΔΔΔ POWERΔCALCULATIONS

Where Δ represents a blank space.

Example 2:

It is desired to print three headings on the same line, each separated from the other by five blank spaces. The following WRITE and FORMAT statements can be used:

```
      WRITE (3, 20)
   20 FORMAT (10HΔHEADING 1, 5X, 9HHEADING 2, 5X, 9HHEADING 3)
```

The line would be printed as follows in columns 1 through 37:

```
       HEADING 1      HEADING 2      HEADING 3
       ↓              ↓              ↓
  Col. 1              15             29
```

The blank field specification does not require a terminating comma or other field separator. For example, the FORMAT statement of example 2 could be written:

```
   20 FORMAT (10HΔHEADING 1, 5X 9HHEADING 2,    5X    9HHEADING 3)
```

Carriage Control for Printer Output

It is important to remember that, in formatting output to the printer, the first field position is a carriage control indicator. A blank in the first field position indicates single vertical spacing. All carriage control indicators are listed below.

Δ	Single space before printing current line.
0	Double space before printing current line.
1	Space to head of form before printing current line (skip to next page).
2, 3, 4, 5, 6, 7, 8, 9	Space the indicated number of lines. (The number of blank lines will be one less than that given by the carriage control indicator.)

Therefore, the first character of the first field following the FORMAT statement operator and the first character of the first field following each unit record terminator (slash) is a carriage control indicator and must not be used as part of the field to be printed. For example, consider the following sequence of statements:

```
          I = 10
          WRITE (3, 20) I
     20   FORMAT (I2)
```

No blank for single spacing has been left in the FORMAT field specification. The character 1 of the value 10 is assumed to be the carriage control indicator. The printer will skip to head of form and print 0 in column 1. Any of the following FORMAT statements will correct the printer output by causing the printer to single space and print 10 in columns 1 and 2:

```
Example 1:  20 FORMAT (I3)
Example 2:  20 FORMAT (1X, I2)
Example 3:  20 FORMAT (1HΔ, I2)
```

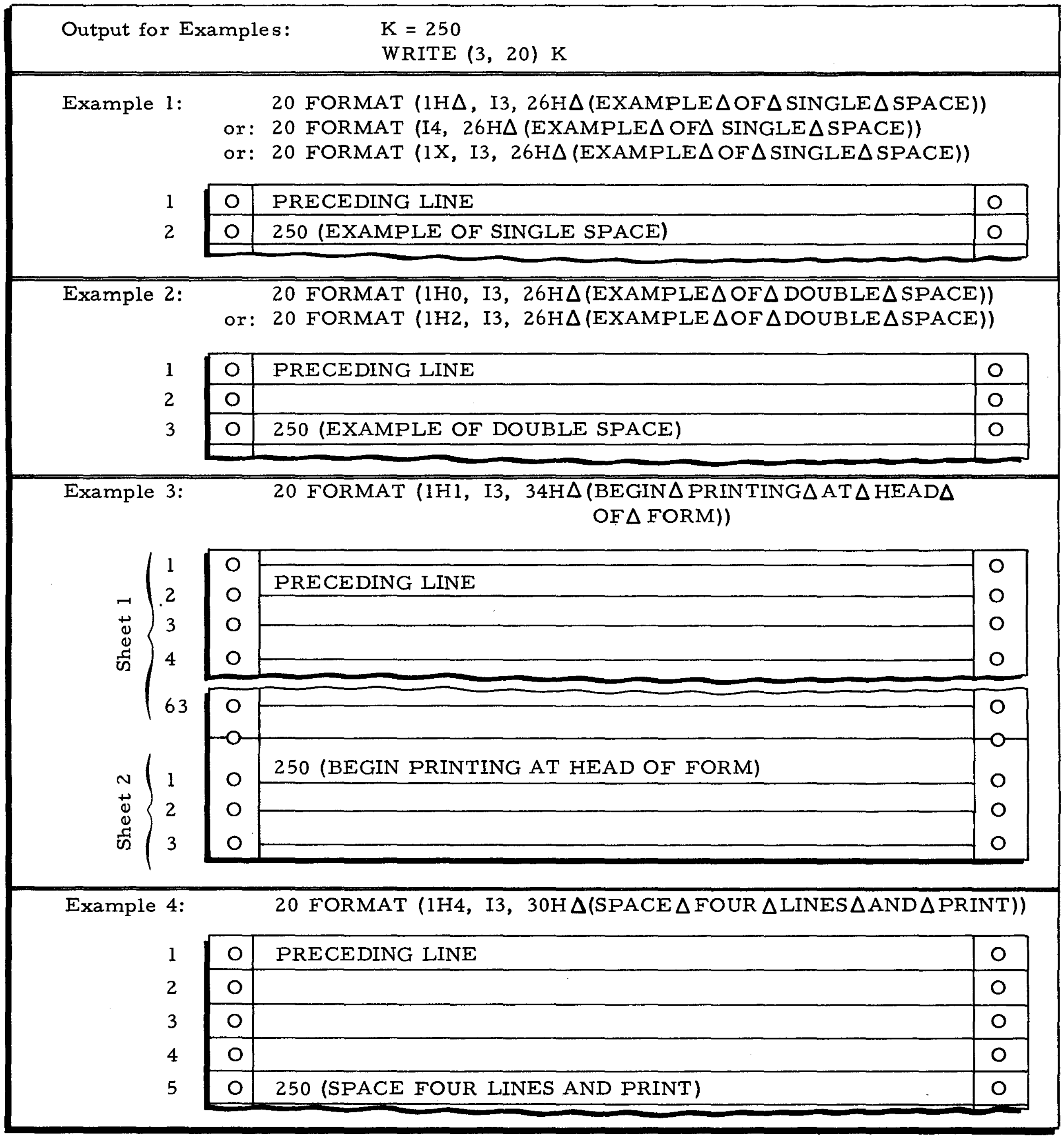
In Example 1, the value 10 is right justified in a three-character field, leaving the first character blank. In Example 2, a blank field specification is used to indicate a blank for carriage control. In Example 3, a Hollerith field specification indicates the blank.

The restriction in this section is limited to printer output only. It does not affect input formatting or output to tape or punched cards.

While several methods are available to indicate single spacing, the Hollerith field specification is commonly used to indicate other forms of carriage control. Examples of carriage control are given in Figure 5-14.

When a FORMAT statement describes more than one unit record, a carriage control indicator must be given for each unit record. See examples in Figure 5-15. For a complete discussion of multiple-record forms, see pages 5-45 to 5-51.

Since vertical spacing information applies only to printer output, when the output is to a punch or to a tape, the carriage control indicators are treated as characters.



```
Output for Examples:      K = 250
                          WRITE (3, 20) K

Example 1:        20 FORMAT (1HΔ, I3, 26HΔ(EXAMPLEΔOFΔSINGLEΔSPACE))
              or: 20 FORMAT (I4, 26HΔ(EXAMPLEΔOFΔSINGLEΔSPACE))
              or: 20 FORMAT (1X, I3, 26HΔ(EXAMPLEΔOFΔSINGLEΔSPACE))

        1   PRECEDING LINE
        2   250 (EXAMPLE OF SINGLE SPACE)

Example 2:        20 FORMAT (1H0, I3, 26HΔ(EXAMPLEΔOFΔDOUBLEΔSPACE))
              or: 20 FORMAT (1H2, I3, 26HΔ(EXAMPLEΔOFΔDOUBLEΔSPACE))

        1   PRECEDING LINE
        2
        3   250 (EXAMPLE OF DOUBLE SPACE)

Example 3:        20 FORMAT (1H1, I3, 34HΔ(BEGINΔPRINTINGΔATΔHEADΔ
                                   OFΔFORM))

Sheet 1   1
          2   PRECEDING LINE
          3
          4
         63
Sheet 2   1   250 (BEGIN PRINTING AT HEAD OF FORM)
          2
          3

Example 4:        20 FORMAT (1H4, I3, 30HΔ(SPACEΔFOURΔLINESΔANDΔPRINT))

        1   PRECEDING LINE
        2
        3
        4
        5   250 (SPACE FOUR LINES AND PRINT)
```

Figure 5-14. Carriage Control in Single-Record FORMAT Statements

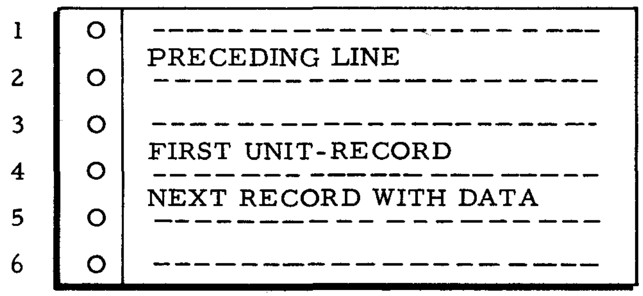
Field-Repetition Constant

When successive data fields are described by identical field specifications, it is not necessary to write each field specification separately. Instead, a field-repetition constant, r, in front of the first field specification, indicates the number of times the field is to be repeated. This abbreviated form may be used with any type of field specification except wH or wX. The constant, r, may be any unsigned integer greater than zero and less than or equal to 63.

As an example of use of the field-repetition constant:

```
    20 FORMAT (5F9.2, 2A5)
     is equivalent to:
    20 FORMAT (F9.2, F9.2, F9.2, F9.2, F9.2, A5, A5)
```

Example 1: 20 FORMAT (18H0FIRSTΔUNIT-RECORD/22HΔNEXTΔRECORDΔ WITHΔDATA)



Example 2: 20 FORMAT (18HΔFIRSTΔUNIT-RECORD/22H0NEXTΔRECORDΔ WITHΔDATA)

or: 20 FORMAT (18HΔFIRSTΔUNIT-RECORD//22HΔNEXTΔRECORDΔ WITHΔDATA)

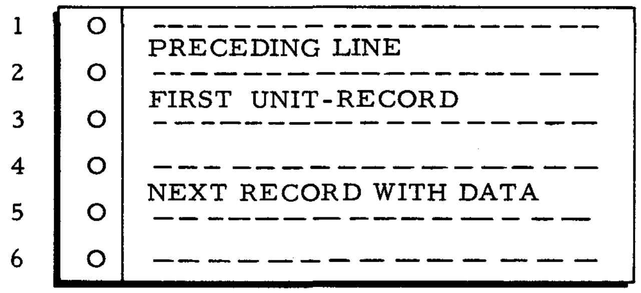


Figure 5-15. Carriage Control in Multiple-Record FORMAT Statements

Repetition of Groups of Field Specifications

As described briefly on page 5-14, an appropriate repetition constant can be used with groups of field specifications to provide a shortened notation in writing a FORMAT statement. For convenience, the group-repetition constant is designated g. While not a part of the field specification as is r, the group-repetition constant is used in a similar way as described below.

When two different field specifications alternate repetitively in a FORMAT statement, a group-repetition constant may be used in writing the FORMAT statement. This same short notation can also be used when more than two field specifications recur repetitively in the same relative sequence. A pair of parentheses is placed around the pair or group of field specifications which recur repetitively, and the appropriate repetition constant, g, is written before the opening parenthesis. The constant g may be any unsigned integer greater than zero and less than or equal to 63.

Example 1:

```
20 FORMAT (4(F9.2, I7))
   may be used in place of:
20 FORMAT (F9.2, I7, F9.2, I7, F9.2, I7, F9.2, I7)
```

The group-repetition constant, g, is equal to 4. The example shows the shortened notation used when repeating alternating pairs of field specifications.

Example 2:

```
20 FORMAT (4(F12.6, 3I7, F4.1, E6.3, O5))
   may be more conveniently written and keypunched than the equivalent:
20 FORMAT ( F12.6, 3I7, F4.1, E6.3, O5,  F12.6, 3I7, F4.1, E6.3, O5,
            F12.6, 3I7, F4.1, E6.3, O5,  F12.6, 3I7, F4.1, E6.3, O5)
```

g = 4. The example shows repetitive groups of field specifications.

Example 3:

```
20 FORMAT (O11, 3(F9.2, I7), 5A5, 4(I4, E6.3), F22.6)
   is equivalent to:
20 FORMAT (O11, F9.2, I7,  F9.2, I7,  F9.2, I7,  5A5,
   I4, E6.3,  I4, E6.3,  I4, E6.3,  I4, E6.3,  F 22.6)
```

In this example, single field specifications are interspersed with repetitive groups.

Scale Factor

In F, E, and G conversions the programmer has the option of using a scale factor as part of the field specification. The scale factor precedes other components of the field specification and is written:

$\boxed{sP}$

Where: s is a decimal value, positive, negative, or zero, indicating the number of decimal places that the decimal point is to be shifted ($-31 \leq s \leq +31$).

P always follows s and identifies s as a scale factor.

On input, the scale factor affects the value of the datum only if there is no explicit exponent in the data field. If the incoming datum is a decimal number without an expressed exponent, the value stored will be changed by the power of ten expressed by the scale factor. On output, the type of conversion, E, F, or G, determines the effect of the scale factor. For E-conversion output operations, the scale factor changes the value by the power of ten indicated by s, but the exponent is also modified so that the value remains the same although expressed differently.

Finally, for G-conversion output operations, the scale factor does not apply to F-type conversions. If E-type output conversion is used, the scale factor has the same effect as if E conversion had been specified originally, i.e., the decimal point is shifted but the exponent is modified so that the actual value remains the same. When an E, F, or G conversion has no given scale factor, the scale factor is understood to be zero and no shift occurs.

The direction in which the decimal point of the value is to be shifted is determined by whether the scale factor is positive or negative. By convention, the scale factor is said to be positive if the decimal point is shifted to the left on input, and negative if it is shifted to the right. For output, the convention is just the opposite: The scale factor is positive if the decimal point is shifted to the right and negative if it is shifted to the left. A positive scale factor may be written with or without the plus sign; a negative scale factor is preceded by a minus sign. Omission of the sign automatically implies a positive scale factor. Table 5-6 shows shifting of the decimal point by the scale factor in I/O operations.

Table 5-6. Scale Factor Shifting of Decimal Point

Sign of Scale Factor	I/O Operation	Direction of Shift
s or +s	Input	← (left)
-s	Input	→ (right)
s or +s	Output	→ (right)
-s	Output	← (left)

INPUT

The effect of the scale factor on input values may be expressed by the general equation:

$$I = X \cdot 10^{-s}$$

Where: I = Internal value.

X = External value.

s = Scale factor (a signed or unsigned integer in the range $-31 \leq s \leq +31$).

Thus, for a scale factor of 3 and an incoming value of 123.4567,

$$123.4567 \times 10^{-3} = .1234567$$

And for a scale factor of -3,

$$123.4567 \times 10^{-(-3)} = 123.4567 \times 10^{+3} = 123456.7$$

OUTPUT

The effect of the scale factor on output values may be expressed by the general equation:

$$X = I \cdot 10^{s}$$

Where: X, I, and s are as defined above (since the equation is the same one rearranged). Thus, for an internal value of 0.1234567 and a scale factor of 3, the external value is 123.4567:

$$X = .1234567 \times 10^3 = 123.4567$$

Table 5-7 shows the effects of use of a scale factor on input values for F conversion, and Table 5-8 gives the same information for output values. Table 5-9 shows the scale factor effect on output values in E conversion.

Table 5-7. Effects of Scale Factor on Input Values (F Conversion)

ON INPUT, A POSITIVE SCALE FACTOR SHIFTS DECIMAL POINT TO LEFT BY <u>s</u> PLACES

+s

Input Value Before Scaling	FORMAT Specification	Scale Factor	Input Value After Scaling
123.4567	F8.4	0	123.4567
123.4567	1PF8.4	1	12.34567
123.4567	2PF8.4	2	1.234567
123.4567	3PF8.4	3	0.1234567
123.4567	4PF8.4	4	0.01234567

ON INPUT, A NEGATIVE SCALE FACTOR SHIFTS DECIMAL POINT TO RIGHT BY <u>s</u> PLACES

-s

Input Value Before Scaling	FORMAT Specification	Scale Factor	Input Value After Scaling
123.4567	F8.4	0	123.4567
123.4567	-1PF8.4	-1	1234.567
123.4567	-2PF8.4	-2	12345.67
123.4567	-3PF8.4	-3	123456.7
123.4567	-4PF8.4	-4	1234567.

Table 5-8. Effects of Scale Factor on Output Values (F Conversion)

ON OUTPUT, A POSITIVE SCALE FACTOR SHIFTS DECIMAL POINT TO RIGHT BY <u>s</u> PLACES

+s

Output Value Before Scaling	FORMAT Specification	Scale Factor	Output Value After Scaling	Presentation on Output Medium
.1234567	F9.7	0	.1234567	.1234567
.1234567	1PF10.7	1	1.234567	1.2345670
.1234567	2PF11.7	2	12.34567	12.3456700
.1234567	3PF12.7	3	123.4567	123.4567000
.1234567	4PF13.7	4	1234.567	1234.5670000

Table 5-8 (cont). Effects of Scale Factor on Output Values (F Conversion)

ON OUTPUT, A NEGATIVE SCALE FACTOR SHIFTS DECIMAL POINT TO LEFT BY s PLACES

-s

Output Value Before Scaling	FORMAT Specification	Scale Factor	Output Value After Scaling	Presentation on Output Medium
.1234567	F9.7	0	.1234567	.1234567
.1234567	-1PF9.7	-1	.01234567	.0123457
.1234567	-2PF9.7	-2	.001234567	.0012346
.1234567	-3PF9.7	-3	.0001234567	.0001235
.1234567	-4PF9.7	-4	.00001234567	.0000123

Examples of scale factors written in the field specification of the FORMAT statement are:

General Form		Example
sPFw.d	without repetition constant	3PF8.4
sPEw.d		-3PE11.4
sPGw.d		+2PG14.3
sPrFw.d	with repetition constant	3P2F8.4
sPrEw.d		-3P2E11.4
sPrGw.d		+2P3G14.3

Table 5-9. Effects of Scale Factor on Output Values (E Conversions)

POSITIVE SCALE FACTOR SHIFTS DECIMAL POINT TO RIGHT BY s PLACES AND DECREASES EXPONENT BY s

Output Value Before Scaling	FORMAT Specification	Scale Factor	Output Value After Scaling	Presentation on Output Medium
$.1234 \times 10^6$	E10.4	0	$.1234 \times 10^6$	.1234E+06
$.1234 \times 10^6$	1PE11.4	1	1.234×10^5	1.2340E+05
$.1234 \times 10^6$	2PE12.4	2	12.34×10^4	12.3400E+04
$.1234 \times 10^6$	3PE13.4	3	123.4×10^3	123.4000E+03
$.1234 \times 10^6$	4PE14.4	4	$1234. \times 10^2$	1234.0000E+02

NEGATIVE SCALE FACTOR SHIFTS DECIMAL POINT TO LEFT BY s PLACES AND INCREASES EXPONENT BY s

Output Value Before Scaling	FORMAT Specification	Scale Factor	Output Value After Scaling	Presentation on Output Medium
$.1234 \times 10^6$	E10.4	0	$.1234 \times 10^6$	.1234E+06
$.1234 \times 10^6$	-1PE10.4	-1	$.01234 \times 10^7$	.0123E+07
$.1234 \times 10^6$	-2PE10.4	-2	$.001234 \times 10^8$	.0012E+08
$.1234 \times 10^6$	-3PE10.4	-3	$.0001234 \times 10^9$	.0001E+09
$.1234 \times 10^6$	-4PE10.4	-4	$.00001234 \times 10^{10}$	.0000E+10

At the instant when a FORMAT statement assumes control, a zero scale factor takes effect and remains in effect until it is superseded by the appearance of a nonzero scale factor in an E, G, or F field specification of the FORMAT statement. Once a new scale factor is established, it applies to all following field specifications involving E, G, or F conversions within the same FORMAT statement, including rescans, until it is superseded by another scale factor appearing later in the same FORMAT statement.

The following example shows several points about the continuity of the scale factor:

```
20 FORMAT (F10.2, 3PF8.3, E8.1, 5A3, O7, F9.1, 0PF8.2  G11.3)
 is equivalent to:
20 FORMAT (F10.2, 3PF8.3, 3PE8.1, 5A3, O7, 3PF9.1, 0PF8.2, 0PG11.3)
```

Note in the example above:

1. The first specification, F10.2, has an implied scale factor of 0 and is not affected by scale factors in specifications that follow.
2. The first given scale factor, 3, governs all E, F, and G conversions that follow until another scale factor is given. This includes the field specification F9.1. Thus, the scale factor is not affected by intervening alphabetic, octal, Hollerith, integer, and other conversions that do not use scale factors.
3. The scale factor is superseded when a new scale factor is given in the field specification 0PF8.2. This new scale factor then governs the scale factor for the G conversion that follows.
4. If the FORMAT statement given in the example were used in conjunction with an input statement, the scale factor would not affect the value of the conversions given in specifications E8.1 and G11.3.

A scale factor, once established in a FORMAT statement, remains in effect when the FORMAT statement is rescanned as described in pages 5-47 through 5-49.

The established scale factor applies to all unit records of a multiple-record format.

Multiple-Record Forms

As described briefly on page 5-13, a multiple-record form is a variation of the general FORMAT statement. The multiple-record form makes use of a slash (/) as a format field separator for unit records, where a unit record is defined in terms of the I/O medium (see Figure 5-16).

If the I/O statement contains an item list that will require more than the allotted limit for a unit record in the output medium, the programmer must provide record termination marks (/) at the appropriate places in the FORMAT statement. In this way, a new printed line, a new card, or a new tape record will begin before the maximum limit for the previous one has been exceeded.

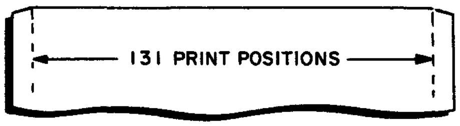
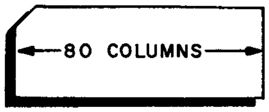
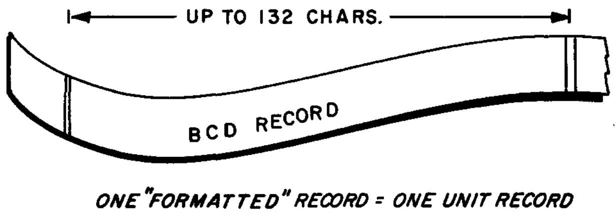
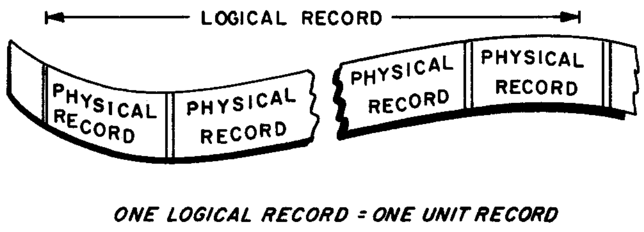
(a) A single line of up to 131 characters on a printer page.	
(b) One tabulating card of up to 80 characters.	
(c) A formatted (BCD) record on magnetic tape of up to 132 characters.	 UP TO 132 CHARS. BCD RECORD ONE "FORMATTED" RECORD = ONE UNIT RECORD
(d) A logical record composed of any number of physical records of data on magnetic tape in the form of its internal representation. (See Section X, "I/O Programming Tips," for the formula to determine the number of physical records in a logical record.)	 LOGICAL RECORD PHYSICAL RECORD PHYSICAL RECORD PHYSICAL RECORD PHYSICAL RECORD ONE LOGICAL RECORD = ONE UNIT RECORD

Figure 5-16. Definition of a Unit Record

Example 1:

It is desired to print 11 values, allotting 14 print positions to each value. The following WRITE and FORMAT statements will cause nine values to be printed on the first line and the remaining two on the next line:

```
      WRITE (3, 20) (ARRAY(I), I = 1, 11)
   20 FORMAT (1X, 9F14.3/1X, 2F14.3)
```

If no record-termination mark had been given at the appropriate place and the FORMAT statement had been written as

```
   20 FORMAT(1X, 11F14.3)
```

part of the 10th value and the 11th value would be printed on the next line. It is recommended that the programmer terminate records so that values will not be divided between lines.

Example 2:

It is desired to punch 13 values, allotting 11 columns to each. The following WRITE and FORMAT statements will cause seven values to be punched on the first card and the remaining six on the next card:

```
      WRITE (5, 20) (ARRAY(I), I = 1, 13)
   20 FORMAT (7F11.3 / 6F11.3)
```

When the list of an input or output statement is used to transfer more than one unit record and the different records have different formats, a slash (/) must be used to separate the format specifications of each record. This use of the record termination statement necessitates rescanning.

The FORMAT statement is scanned from left to right in conjunction with the list of an input or output statement. If additional items in the input or output list remain to be transmitted after the FORMAT statement has been completely scanned from left to right, the scan returns to the last first-level left parenthesis (defined below) of the same FORMAT statement and resumes its left-to-right cycle from that point, until the list is satisfied or until the end of the FORMAT statement is again reached. Then, if the list is still not satisfied because items remain to be transmitted, the scan once more returns to the same point as before (i.e., the last first-level left parenthesis), and the cycle repeats again, continuing to repeat until the list is finally satisfied. If there is no first-level parenthesis in the FORMAT statement, the scan returns to the beginning of the FORMAT statement and repeats from there until the list is satisfied. A group-repetition constant preceding the last first-level left parenthesis is detected during rescanning and has the desired effect. Each rescan starts with a new unit record.

As an example of the use of the multiple-record form with rescanning, consider the following FORMAT statement:

```
20 FORMAT (I6 / F10.6)
```

If this statement is used with a READ statement, the first data card is read under control of the I6 field specification, and the next card is read under control of F10.6. If the input list of the READ statement is still not satisfied, the FORMAT statement is automatically rescanned, and the third card is read under control of I6, the fourth under control of F10.6. Rescanning continues in this manner with all odd-numbered cards being read under control of I6 and all even-numbered cards under control of F10.6 until the list is satisfied.

Similarly, when the above FORMAT statement is used with a WRITE statement designating the printer, the first line is printed under control of I6, and the next line under control of F10.6. If the output list of the WRITE statement is still not satisfied, rescanning occurs as above, so that all odd-numbered lines are printed under control of I6, and all even-numbered lines under control of F10.6 until the list is satisfied.

Rescanning (or scanning) of a FORMAT statement stops as soon as the input/output list is satisfied, except for the following case. If the next field specification is a Hollerith (wH) field specification, the Hollerith characters specified in the FORMAT statement are transmitted before the input/output operation is considered concluded (see Example 6 on page 5-51). Any record-termination marks following the last-used field specification are also honored (see Example 5 on page 5-50). The action taken upon encountering n consecutive record-termination marks in a FORMAT statement is defined on pages 5-48 to 5-51.

Parentheses in a FORMAT statement may be "zero level," "first level," or "second level," as defined and illustrated in Figure 5-17. Figure 5-18 shows the rescan points for each of the examples of Figure 5-17.

Level	Definition	Examples (Shaded areas show parenthesis level being illustrated)
Zero	The opening(left) and closing (right) parentheses of the FORMAT.	20 FORMAT (I6, F9.3, E11.4, A5, F22.4, O6) 20 FORMAT (I6/(3F9.3, E11.4, A5, F22.4, O6))
First	A left parenthesis that is either the second parenthesis of a FORMAT statement or one that follows a first-level right parenthesis. A right parenthesis following a first-level left parenthesis or a right parenthesis following a second-level right parenthesis.	20 FORMAT (I6/ (3F9.3, E11. 4, A5, F22.4, O6)) 20 FORMAT (I6, 2 (3F9.3, 3(E11.4, I3)),F22.4) 20 FORMAT (I6 (3F9.3, I4) , 2 (E11.4, (I5, I3)), I8)
Second	A left parenthesis that is the next parenthesis after a first-level, left parenthesis. A right parenthesis following a second-level left parenthesis.	20 FORMAT (I6, 2(3F9.3, 3 (E11.4, I3)), F22.4) 20 FORMAT (I6, (3F9.3,I4), 2 (E11.4, (I5,I3)),I8)

Figure 5-17. Parenthesis Levels in a FORMAT Statement

Consecutive slashes can be used in FORMAT statements. Their significance depends on whether an input or output statement is being referenced. On input, n consecutive slashes in a FORMAT statement cause n-1 unit records to be skipped (i.e., bypassed). If the slashes appear at the end of the FORMAT statement, an additional record will be skipped by the action of the right parenthesis.

Example 1:

```
      READ (2, 20) INTEGR, A
   20 FORMAT (I6 /// F10.6)
```

Interpretation: Read first card under control of I6 and store value in INTEGR; skip next two cards; read fourth card under control of F10.6 and store value in A.

Rescan Point	Example
When only zero-level parentheses are present, the rescan begins at the start of the FORMAT statement.	Direction of Scan → 20 FORMAT (I6, F9.3, E11.4, A5, F22.4, O6) Rescan Point If I/O list is not completed when this point is reached, return to rescan point.
When only zero- and first-level parentheses are present, the rescan begins at the last (rightmost) first-level left parenthesis. If a second rescan is required, the new rescan begins at this same point.	Direction of Scan → 20 FORMAT (I6 / (3F9.3, E11.4, A5, F22.4, O6)) Rescan Point If I/O list is not completed when this point is reached, return to rescan point.
When zero-, first-, and second-level parentheses are present, the rescan will still begin at the last (rightmost) first-level left parenthesis.	Direction of Scan → 20 FORMAT (I6 / (3F9.3, 2(E11.4, I3)), F22.6) Rescan Point If I/O list is not completed when this point is reached, return to rescan point.
When a group repetition constant precedes the last first-level left parenthesis, rescans will include group repetition.	Direction of Scan → 20 FORMAT (I6, (3F9.3, I4), 2(E11.4, (I5, I3)), I8) Rescan Point If I/O list is not completed when this point is reached, return to rescan point.

Figure 5-18. Rescanning a FORMAT Statement

Example 2:

```
      READ (2, 20) INTEGR, A, J, B
 20   FORMAT (I6, /// F10.6)
```

Interpretation: Same as above, but continuing as follows: Read fifth card under control of I6 and store value in J; skip next two cards; read eighth card under control of F10.6 and store value in B.

Example 3:

```
      READ (2, 20) INTEGR, A
 20   FORMAT (I6, F10.6 ///)
```

Interpretation: Read first card; store in INTEGR the integer value found in columns 1-6; store in A the real value found in the next ten columns; skip rest of card and next two cards as indicated by the slashes and an additional card as indicated by the right parenthesis. (The next execution of a READ statement will read the fifth card.)

Example 4:

```
      READ (2, 20) INTEGR, A, J, B
   20 FORMAT (I6, F10.6 ///)
```

Interpretation: Same as above, but continuing as follows: Read fourth card; store in J the integer value found in columns 1-6; store in B the real value found in the next ten columns; skip rest of card and next two cards. (The next execution of a READ statement will read the eighth card.)

On output, n consecutive slashes in a FORMAT statement cause n-1 blank lines to be written, except when the slashes appear at the end of the FORMAT statement. In that case, and only in that case, an additional blank line is written before rescanning occurs, as indicated by the right parenthesis. Thus, n consecutive slashes written at the end of a FORMAT statement cause n blank lines to be written.

Example 1:

```
      WRITE (3, 20) INTEGR, A
   20 FORMAT (I6 /// F10.6)
```

Interpretation: On first line, print (under control of I6) the value stored in INTEGR; write two blank lines; on fourth line, print (under control of F10.6) the value stored in A.

Example 2:

```
      WRITE (3, 20) INTEGR, A, J, B
   20 FORMAT (I6 /// F10.6)
```

Interpretation: Same as above, but continuing as follows, due to rescanning: On fifth line, print (under control of I6) the value stored in J; write two blank lines; on eighth line, print (under control of F10.6) the value stored in B.

Example 3:

```
      WRITE (3, 20) INTEGR, A
   20 FORMAT (I6, F10.6 ///)
```

Interpretation: On first line, print (under control of I6 and F10.6, respectively) the values stored in INTEGR and A; write three blank lines.

Example 4:

```
      WRITE (3, 20) INTEGR, A, J, B
   20 FORMAT (I6, F10.6 ///)
```

Interpretation: Same as above, but continue as follows, due to rescanning: on fifth line, print (under control of I6 and F10.6, respectively) the values stored in J and B; write three blank lines.

Example 5:

```
      WRITE (3, 20) INTEGR, A, J
   20 FORMAT (I6 /// F10.6)
```

Interpretation: Same as Example 1 above, but continuing as follows: On fifth line, print (under control of I6) the value stored in J; write two blank lines.

Note that the scan stops as soon as the list is satisfied, but that the record-termination marks are heeded when they follow the last used field specification.

Example 6:

```
      WRITE (3, 20) INTEGR, A, J
   20 FORMAT (I6, 13HEND OF RECORD /// F10.6)
```

Interpretation: On first line, print (under control of I6) the value stored in INTEGR; starting in column 7 of same line, print "END OF RECORD"; write two blank lines; on fourth line, print (under control of F10.6) the value stored in A; on fifth line, print (under control of I6) the value stored in J; starting in column 7 of same line, print "END OF RECORD"; write two blank lines.

Note that the scan stops as soon as the list is satisfied, but that Hollerith fields are transmitted when they follow the last used field specification and that the record-termination marks are also heeded.

In a multiple-record FORMAT statement, it is possible to specify that the first record have one format and that all following records have another format. This is done by enclosing the last record specification in a second set of parentheses.

Example: `20 FORMAT (10I3 / (3F10.3, G10.6) )`

When this statement is used with an output statement, the first record will be printed (or recorded on tape) under control of the 10I3 field specification, and every subsequent record will be printed or recorded under control of the other two field specifications until the output list is satisfied.

Reading in FORMAT at Object Time

Occasions may arise where the format of the input data to a program will differ from run to run, or perhaps within the same run. In such cases, it would be advantageous if the format specification could be supplied along with the input data, instead of being rigidly and irrevocably prescribed in the program beforehand. To permit the changing or prescription of formats at object time, the following technique is used. In the object program, the programmer allocates adequate storage space for an array which will later be filled with the pertinent format information at execution time. During execution of the object program, the format information is read into the array before any input data are handled. Then, the input statement reading the input data references the format array, instead of a FORMAT statement. Since the contents of the array may be easily changed by reading in new format information when desired, the effect is equivalent to having variable format statements. Once an array has been used as a FORMAT statement, it may not appear on the righthand side of an assignment statement, within an IF clause, or in an output list until the array has been re-initialized. The array may only be used as a FORMAT until it has been re-initialized. If any change is desired in the formatted array once it has been used, it must be re-initialized.

To make use of object-time formatting, the programmer must:

1. Determine how large an array is required to handle the largest incoming format specification. If array storage space is readily available, the programmer can allow an arbitrary number of storage locations for the array. However, if array storage is at a premium, the programmer can compute the minimum array storage for the specification as shown later in this section.
2. Allocate appropriate array storage by declaring and dimensioning the array in a DIMENSION, COMMON, or data-type statement in the object program.
3. Include in the object program an input statement and a FORMAT statement which will read the incoming format specification into array storage. An alphabetic field specification (Aw) is used to read the actual format information into the array. The number of characters specified for the fixed-point field will determine the number of characters that can be read into each array (i.e., 5 for unspecified fixed-point fields).
4. Reference the format array (instead of a FORMAT statement) in the READ statement governing input of data.
5. Supply (at object time) the format information to be read into the array.

   NOTE: The format information is written exactly as though it were a FORMAT statement, except that the word FORMAT must be omitted and there is no statement label. However, both the left and right parentheses which bound the format specification list must be included. (See Figure 5-18.)

The following example illustrates the principle of reading in formats at object time:

Example:

A program has been written to accept three values from each card of several large decks of input data. However, the input decks have not been consistently punched, and the data fields do not start in the same column in each deck, although the relative sequence of the data fields is the same in all decks. To eliminate the problem caused by the inconsistent starting columns, the format specification of each card deck is read into an array just before the deck is processed. Shown in Figure 5-19 are some of the variations in format which might be encountered in the different input decks when the same three values are to be read.

The following statements in the object program will produce the desired result:

```
                                        (See Step 1 above.)
      DIMENSION  IRRAY (12)             (See Step 2.)
  3   READ (2, 20) (IRRAY (I), I=1, 10) (See Step 3.)
 20   FORMAT (10A5)                     (See Step 3.)
      READ (2, IRRAY)  J, B, C          (See Step 4.)
                 .
                 .
                 .
      GO TO 3
```

Every time a new input deck is to be read, the READ (2, 20) statement should be executed; this will cause the contents of the format specification card preceding the deck to be read into the array under control of the 10A5 field specification. Then the input data can be read by means of the READ (2, IRRAY) statement. J, B, and C are variables in which the incoming values are to be stored.

The example is repeated later with minimum array storage, instead of the arbitrary 12 locations.

When array storage is at a premium, the programmer can determine the minimum size of the format array by counting the number of characters in the largest expected incoming field specification, beginning with the left parenthesis bounding the field specification and ending with the right parenthesis terminating the field specification. All characters, including the parentheses and embedded blanks, are significant.

For example, if the largest expected format specification at object time is:

(5F10. 3, 3(4E10. 2, 2F12. 6), 4(I5, E10. 2, 3HΔBΔ))

and each fixed-point field has five characters, array storage in the object program could be allocated by the following statement:

DIMENSION IRRAY (10)

The format specification has 48 characters and therefore the minimum array storage space that can be allocated is 10 array storage locations.

Because of the modularity of the I/O execution package, array formatting will not of itself bring in the proper conversion routines. The programmer must include a FORMAT statement containing the conversion codes to be used in array formatting for each chain containing array formatting. The FORMAT statement may be a dummy statement.

Alternate Creation of Variable Formats

It was shown in the preceding section how a format description can be read into an array from cards at object time; then the format array, instead of a FORMAT statement, is referenced by an input or output statement.

Variable formats can alternatively be created by transferring into an array the contents of constants and/or variables having format data; then the array can be referenced by an input or output statement, as in the previous section.

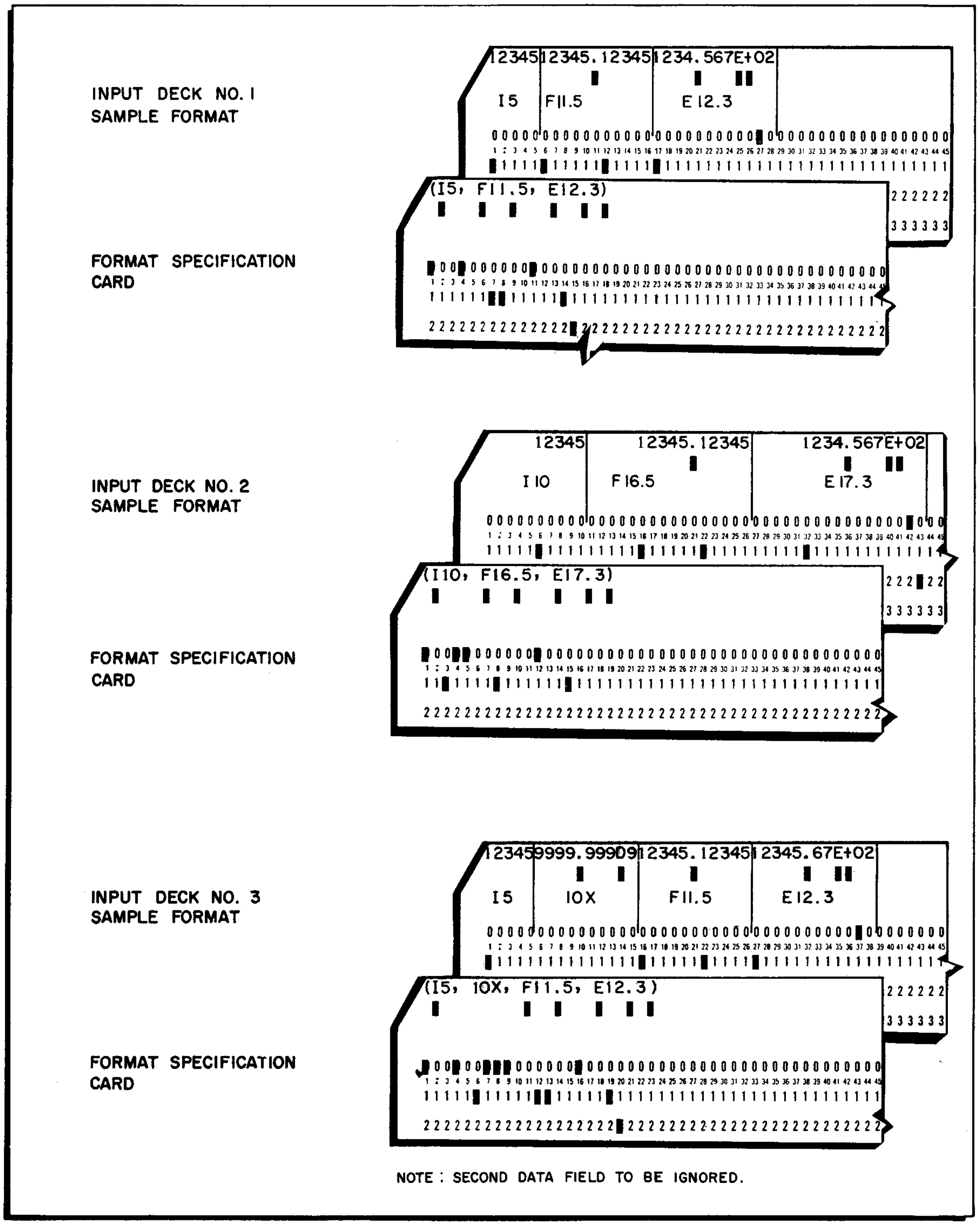

INPUT DECK NO. 1 SAMPLE FORMAT
12345 12345.12345 1234.567E+02
I5
F11.5
E12.3
FORMAT SPECIFICATION CARD
(I5, F11.5, E12.3)
INPUT DECK NO. 2 SAMPLE FORMAT
12345 12345.12345 1234.567E+02
I10
F16.5
E17.3
FORMAT SPECIFICATION CARD
(I10, F16.5, E17.3)
INPUT DECK NO. 3 SAMPLE FORMAT
12345 9999.99909 12345.12345 12345.67E+02
I5
10X
F11.5
E12.3
FORMAT SPECIFICATION CARD
(I5, 10X, F11.5, E12.3)
NOTE : SECOND DATA FIELD TO BE IGNORED.

Figure 5-19. Handling Variations in Format at Object Time

Example:

The format description of Figure 5-19

(I5, 10X, F11.5, E12.3)

is to be created by transferring the contents of constants and variables, rather than by reading the format description from an input card at object time. The following statements will accomplish the task:

```
DIMENSION IRRAY (5)
DATA IRRAY (1), IRRAY (2), IRRAY (3), IRRAY (4), IRRAY (5)/5H(I5,Δ, 5H10X,Δ,
                                                 5HF11.5, 5H,ΔE12, 5H.3)ΔΔ/
READ (2, IRRAY) J, A, B
```

The minimum array size of five is computed by adding the number of characters in the format specification. Appropriate array storage is then allocated in the DIMENSION statement. Using a DATA initialization statement, each array element is initialized with five Hollerith characters. The array can now be referenced by the READ statement. Variables J, A, and B are used to store the incoming values.

END FILE STATEMENT

The END FILE statement has the form:

```
END FILE i
```

Where: i represents the device code of a peripheral unit. It may be either an unsigned integer in the range 1 through 15 or an integer variable. If it is an integer variable, it must be assigned a value prior to execution of the statement. The value assigned to i must correspond to a physical device in the equipment configuration available to the program.

Examples:

```
END FILE 1
END FILE IUNIT
```

When addressing a magnetic tape unit, the END FILE statement writes two end-of-file records on the designated tape unit. The statement is ordinarily used to indicate that there is no more valid information on a tape, but it may also be used to separate groups of records into files for any convenient purpose. If upon writing an output tape, it is desired to rewind the tape and then read it, an END FILE statement should be executed before the REWIND statement to avoid reading beyond the written information.

An attempt to write end-of-file records on a card reader or punch will cause job termination and printout of the error message:

IMPROPER COMMAND TO THIS DEVICE.
(Subsequent line shows peripheral device number.)

REWIND STATEMENT

The REWIND statement has the form:

```
REWIND i
```

Where: i represents the device code of magnetic tape unit in the equipment configuration available to the program. i may be an unsigned integer in the range 1 through 15 or an integer variable. If it is an integer variable, it must be assigned a value prior to execution of the statement.

This statement is used to rewind to the beginning of tape the reel mounted on logical tape unit number i.

Examples:

```
REWIND 1
REWIND IUNIT
```

An attempt to rewind a device such as the printer or card reader will cause job termination and printout of the error message:

I/O ERR followed by the peripheral device indicator.

BACKSPACE STATEMENT

The general form is:

```
BACKSPACE i
```

Where: i represents the device code of a magnetic tape unit in the equipment configuration available to the program. i may be an unsigned integer in the range 1 through 15 or an integer variable. If it is an integer variable, it must be assigned a value prior to execution of the statement.

Examples:

```
BACKSPACE 1
BACKSPACE IUNIT
```

The statement moves back by one logical record the tape mounted on magnetic tape unit i. The statement may be used to backspace input or output tapes. In the case of binary tapes, one logical record may correspond to several physical records (see page 5-46); in the case of formatted tapes, a logical record is the same as a physical record.

An attempt to backspace a device such as the printer or punch will cause job termination and printout of the error message:

```
IMPROPER COMMAND TO THIS DEVICE.
```
(Subsequent line shows peripheral device number.)

SECTION VI
PROCEDURES

CATEGORIES OF PROCEDURES

There are two categories of procedures: functions and subroutines. Functions are further subdivided into categories. Figure 6-1 shows the different categories of procedures.

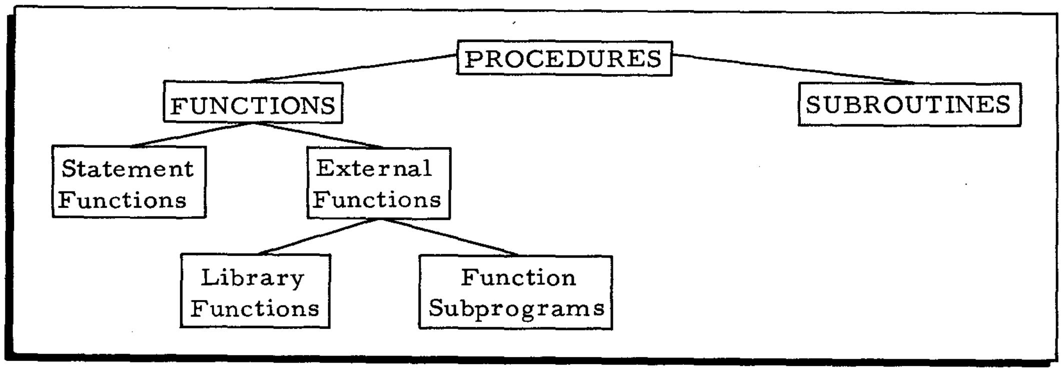


Figure 6-1. Categories of Procedures

All procedures except statement functions are external procedures. Functions that the proposed ASA Fortran specification defines as intrinsic are supplied with this compiler as part of the library functions.

SUBPROGRAMS

There are two categories of subprograms: function subprograms and subroutine subprograms. All subroutines are by definition subroutine subprograms. Function subprograms are headed by FUNCTION statements; subroutine subprograms are headed by SUBROUTINE statements. All subprograms are external procedures.

NAMING AND TYPING PROCEDURES

Procedure names consist of one to six alphanumeric characters, the first of which must be alphabetic. Subroutines do not have types. Functions, however, are typed in the same way as data. Rules for typing functions are:

1. Typing Statement Functions - Logical statement function names must be declared in a LOGICAL statement. Arithmetic statement function names can be typed implicitly using the I, J, K, L, M, N convention, or they can be declared either real or integer in a REAL or INTEGER statement.

2. Typing Function Subprograms - Arithmetic function subprograms can be typed implicitly by using the I, J, K, L, M, N naming convention. However, they can be explicitly declared, and logical function subprograms must be explicitly declared. Prefixing the FUNCTION statement with the appropriate type (REAL, INTEGER, or LOGICAL) explicitly types a function subprogram.

3. Library Functions - The types associated with library functions are predefined within the compiler.

FUNCTIONS

A function consists of a sequence of instructions to perform a mathematical or logical operation. Rather than writing such a sequence each time it is required during a program, the coding is supplied previously either with the Fortran compiler or by the programmer. A function is called when the function name is encountered in an executable statement. When evaluation of a function is complete, control returns to the expression in which the name of the function was embedded. These characteristics are true for all functions. They are listed in Table 6-1 below; in addition, the characteristics that differ with the category of function are listed in the table.

Table 6-1. Characteristics of Functions

Defining Characteristic	Category of Function		
	Statement Function	External Function	
		Library Function	Function Subprogram
Originating Source	Programmer-defined.	Honeywell-defined.	Programmer-defined.
Relative Size	One-statement definition.	More than one statement and may require a large number of statements.	
Number of Output Values Returned	One.	Usually one but may be more than one.	
Method of Compilation	Code is compiled as an integral part of the using program, but it appears only once regardless of the number of times it is used.	Code is compiled independently. Code appears only once, no matter how many times it is used.	
Method of Calling	Called when the function name is encountered in an executable statement.		
Point to Which Control is Returned after Evaluation	Expression in which the name of the function is embedded.		

Statement Functions

Statement functions are single statements written by the programmer. They consist of a single statement of the form:

$\underline{\text{Funame}}\ (\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_n) = \underline{\text{Expression}}$

Where: $\underline{\text{Funame}}$ = Name of the function.

$\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_n$ = The names of variables, which are dummy arguments of the function. ($1 \leq n \leq 63$)

$\underline{\text{Expression}}$ = Any arithmetic expression (if the data type associated with the function name is not LOGICAL), or:

Any logical expression (if the data type associated with the function name is LOGICAL).

The expression portion of the definition may contain non-Hollerith constants, variable references, and references to previously defined statement functions and external functions. It may not reference arrays or array elements, except when these are actual arguments of the statement function.

The name of a statement function may not appear in a COMMON, DIMENSION, EQUIVALENCE, data-type, or EXTERNAL statement, nor as the name of a subroutine in a CALL statement.

The dummy argument list $(arg_1, arg_2, \ldots, arg_n)$ is a list of variables that will be replaced by the actual call arguments when the function is used. Any of the variables in the expression may be written as dummy arguments of the function. There must be at least one dummy argument enclosed within the parentheses and, if there is more than one, commas must separate the arguments. The data type of each dummy argument must be declared prior to using the argument in the definition of a statement function. If the type is not explicitly declared in a data type statement, it is implied to be either integer or real, depending on the first letter of the argument name. Although a dummy argument may have the same name as a variable appearing elsewhere in the program, no two dummy arguments may have identical names.

To use a statement function once it is defined (i.e., to reference a statement function), the programmer writes the statement function, followed by the actual arguments in parentheses, in an arithmetic or logical expression. When the expression is executed, the statement function is evaluated according to its definition, using the actual arguments in place of the dummy arguments. The output of a statement function is a single numerical or logical value that is returned to the expression in which the statement function is embedded. Statement functions are defined only for the programs in which they appear, and the object coding is inserted only once. Statement functions must appear directly after any DATA statements in the input deck and must precede the first executable statement.

Because the actual arguments are to replace the corresponding dummy arguments in the statement defining the function, they must agree in sequence, number, and data type with the corresponding dummy arguments and must be separated by commas (if the list contains more than one actual argument). An actual argument may be the name of any constant, variable, or array element, or it may be any arithmetic or logical expression that is not the expression which uses the function.

Example: The following statement function is written before the first executable statement in the program:

```
ROOT (A, B, C) = (-B + SQRT (B**2 - 4.*A*C))/(2.*A)
```

Where: ROOT is the name of a real function.

A, B, and C are the dummy arguments that will be replaced by actual arguments when the function is used. (Their data types are implied to be real from their names, in the absence of any overriding data-type declaration.)

The right-hand side of the equals sign is the expression to be evaluated when actual arguments replace the dummy arguments at the time the function is called.

Assume that it is desired, later in the program, to evaluate the formula with A replaced by DATA(6), B replaced by 12.8, and C replaced by the absolute value of X minus Y. The following makes use of the previously defined statement function to obtain the result and to store it in a location called VALUE:

```
VALUE = ROOT (DATA(6), 12.8, ABS(X-Y) )
```

If it were further desired to perform other operations in the same statement (for example, adding Z^3 to the result), this could be done as follows:

```
VALUE = ROOT (DATA(6), 12.8, ABS(X-Y) ) + Z**3
```

Since the result takes on the same data type as the function, the quantity stored in VALUE would be a real number in both instances above.

The same statement function may be used in many other expressions in the same program with different values replacing dummy arguments A, B, and C.

Function Subprograms

Function subprograms are programmer-written external functions. A function subprogram can be written by the programmer to express a function that cannot be expressed in a single statement. The function subprogram is then called into operation just as any other function, simply by writing its name in a statement, together with the argument(s) to be employed.

The function subprogram can be compiled as an independent entity. It can consist of a number of statements and can contain any Fortran statement except those listed below under restrictions. The function subprogram, like the statement function, delivers a single value to the expression which called it. However, the function subprogram may deliver additional values by altering the values of some of the variables and/or array elements in its argument list or by addressing elements in the common region.

A function subprogram must begin with a FUNCTION statement and terminate with an END statement. Control returns to the calling program when a RETURN statement or the END statement is encountered. Control will return to the point in the executable statement of the calling program at which the function name occurred.

The general form of the FUNCTION statement is:

$\boxed{\underline{\text{type}}\ \text{FUNCTION}\ \underline{\text{funame}}\ (\text{arg}_1,\ \text{arg}_2, \ldots,\ \text{arg}_n)}$

Where:

type	is one of the following or does not appear at all: INTEGER, REAL, LOGICAL.
funame	is the symbolic name of the function.
$(\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_n)$	is a list of dummy arguments (variables, arrays, function subprograms, or dummy names of subroutines) which will be replaced by actual arguments from the corresponding positions of the statement calling the function into operation. $(1 \leq n \leq 63)$.

The FUNCTION statement is followed by the subprogram body, with the following restrictions:

1. The symbolic name of the function must also appear as a variable name in the defining subprogram. During every execution of the subprogram, this variable must be defined and, once defined, may be referenced or redefined. The value of the variable at the time of any return from this subprogram is called the value of the function.
2. The function name must appear only as the symbolic name of that function in the function subprogram and the calling program. It cannot appear as the name of any other function or as an array name.
3. The function name cannot appear in the function subprogram in a COMMON, DIMENSION, EQUIVALENCE, EXTERNAL, or data-type statement, nor as the name of a subroutine in a CALL statement.
4. The function subprogram cannot contain TITLE or SUBROUTINE statements, other FUNCTION statements, or any usage (either direct or indirect) of the function being defined. Any other Fortran statements can be used in the function subprogram.
5. The dummy argument names cannot appear in EQUIVALENCE, COMMON, or DATA statements in the function subprogram.
6. Each of the dummy arguments should appear at least once in an executable statement in the function subprogram. If the name of an array is to be used as a dummy argument, a statement defining the array must appear in the function subprogram prior to any reference to the array name. The dimensions of the array should be the same as those of any corresponding actual arguments.
7. If the function subprogram is to deliver additional values for dummy arguments in the FUNCTION statement, the actual arguments corresponding to those dummy arguments must be variable names, array elements, or array names.
8. The last statement of the function subprogram must be an END statement.

As indicated in the format for the FUNCTION statement, the data type can be declared in the statement. If it is not explicitly declared, the type is implied from the function name to be either integer or real, depending on the first letter of the name. The data type of the result delivered by the function subprogram is the same as the type declared in the FUNCTION statement or, if no type is declared there, the result is implied by the function name to be either integer or real.

To use a function subprogram, the programmer writes as a term in an arithmetic or logical expression the name of the function followed by a set of parentheses containing the list of actual arguments.

Because the actual arguments are to replace the corresponding dummy arguments in the statement calling the function subprogram, they must agree in sequence, number, and data type with the corresponding dummy arguments and must be separated by commas (if the list contains more than one actual argument). An actual argument may be the name of any constant, variable, or array element, any arithmetic or logical expression, any Hollerith constant, or the name of any other function subprogram or of any subroutine. However, when the name of a function or subroutine subprogram is used as an actual argument, the name must be declared. Hollerith constants must replace integer dummy arguments.

Example: A real function subprogram named VAL is defined. VAL has two dummy arguments, J and K. The example shows only the outline of the function subprogram. Note that more than one return statement can be used.

```
FUNCTION VAL(J, K)
....
....
RETURN
....
RETURN
END
```

Within the calling program, the function subprogram VAL may be called by writing its name in an executable statement with actual arguments. An outline of such a calling program is shown below.

```
Δ TITLE PROCES
....
....
....
QUAN = A * (B/VAL(LIN, NEX(5))
....
....
IF (VAL(NONE, NINE))20, 32, 15
....
....
```

In the example, the function VAL is referenced twice in a calling program with different actual arguments in each case.

Library Functions

Library functions are external functions supplied with the compiler. A list of these functions and their characteristics is given in Table 6-2. The data type of the result delivered by a library function must be the same as that indicated by the table.

Library functions include arithmetic, trigonometric, and Boolean functions. Arguments of arithmetic and trigonometric library functions for which the result is not mathematically defined are improper. For example, if the value of the second argument of AMOD, MOD, SIGN, or ISIGN is zero, the result will be undefined.

Library functions are referenced by writing the name of the function in an executable statement.

Library function names cannot be used to identify and reference a programmer-written function. However, if the library function does not appear in the job being compiled, the function name may be used to identify a variable or an array.

Table 6-2. Library Functions

Name	Function	Definition	Number of Arguments	Type of Argument	Type of Function	Function Used For
ATAN ATAN2	Arctangent	arctan (Arg) arctan ($\text{Arg}_1/\text{Arg}_2$)	1 2	Real	Real	Trigonometric Operations
COS	Trigonometric cosine	cos (Arg)	1	Real	Real	
SIN	Trigonometric sine	sin (Arg)	1	Real	Real	
TANH	Hyperbolic tangent	tanh (Arg)	1	Real	Real	
ABS IABS	Absolute value	$\lvert\text{Arg}\rvert$	1	Real Integer	Real Integer	Arithmetic Operations
AINT INT	Truncation	Sign of Arg times largest integer $\leq \lvert\text{Arg}\rvert$	1	Real Real	Real Integer	
ALOG	Natural logarithm	$\log_e$ (Arg)	1	Real	Real	
ALOG10	Common logarithm	$\log_{10}$ (Arg)	1	Real	Real	
AMOD MOD	Remaindering (see note below)	$\text{Arg}_1 - [\text{Arg}_1/\text{Arg}_2]\text{Arg}_2$	2	Real Integer	Real Integer	
AMAX0 AMAX1 MAX0 MAX1	Choosing largest value	Max ($\text{Arg}_1, \text{Arg}_2, \ldots$)	≥ 2	Integer Real Integer Real	Real Real Integer Integer	

Table 6-2 (cont). Library Functions

Name	Function	Definition	Number of Arguments	Type of Argument	Type of Function	Function Used For
AMIN0 AMIN1 MIN0 MIN1	Choosing smallest value	Min $(Arg_1, Arg_2, ...)$	≥ 2	Integer Real Integer Real	Real Real Integer Integer	Arithmetic Operations (cont)
EXP	Exponential	e^{Arg}	1	Real	Real	
FLOAT	Float	Conversion from integer to real	1	Integer	Real	
IFIX	Fix	Conversion from real to integer	1	Real	Integer	
SIGN ISIGN	Transfer of sign	Sign of Arg_2 times $\lvert Arg_1 \rvert$	2	Real Integer	Real Integer	
DIM IDIM	Positive difference	Arg_1 - Min (Arg_1, Arg_2)	2	Real Integer	Real Integer	
SQRT	Square root	$(Arg)^{(1/2)}$	1	Real	Real	
IAND	Logical AND	$J \cap K$	2	Integer	Integer	Boolean Operations
IOR	Inclusive OR	$J \cup K$	2	Integer	Integer	
ICOMPL	Logical Complement	$\overline{K}$	1	Integer	Integer	
IEXCLR	Exclusive OR	$J \cup K$ and $\overline{(J \cap K)}$	2	Integer	Integer	

NOTES: 1. Trigonometric functions express angles in radians.
2. The ATAN2 function is a four-quadrant arctangent routine.
3. The bracketed quantity Arg_1/Arg_2 represents the integral part of this value.

Examples:

1. Take the absolute value of the square root of X^3-X. Divide the result by X. Store result in Y.

```
Y = ABS(SQRT(X**3-X))/X
```

2. Add integer I and real variable Y by converting I to floating-point form. Store result in X.

```
X = Y + FLOAT(I)
```

3. Given two octal constants and a string of five alphabetic characters such that:

```
I1 = 0077777700
I2 = 7777000000
I3 = BOOLE
```

The following Boolean operations are performed:

```
I4 = IAND(I2, I3)
I5 = IOR (I1, I2)
I6 = ICOMPL (I1)
I7 = IEXCLR (I1, I2)
```

Then, I4 is set to BO000
I5 is set to 7777777700
I6 is set to 7700000077
I7 is set to 7700777700

For programmers who are unfamiliar with Boolean terminology, the Boolean operations are defined below in terms of bit manipulation.

Form	Definition	Example	
IAND (arg1, arg2)	In each bit position, the result is "1" if and only if both bits of the arguments in the corresponding position are "1".	arg1: arg2: result:	0 1 1 0 0 1 1 0 1 0 1 1 0 0 1 0 0 1
IOR (arg1, arg2)	In each bit position, the result is "1" if either or both bits are "1" and the result is "0" only if both are "0".	arg1: arg2: result:	0 1 1 0 0 1 1 0 1 0 1 1 1 1 1 0 1 1
ICOMPL (arg)	In each bit position, the result is "1" if the corresponding bit in the argument is "0" and the result is "0" if the corresponding bit in the argument is "1".	arg: result:	0 1 1 0 0 1 1 0 0 1 1 0
IEXCLR (arg1, arg2)	In each bit position, the result is "1" if either bit in the corresponding position of the arguments is "1" and the result is "0" if the corresponding bits are either both "0" or both "1".	arg1: arg2: result:	0 1 1 0 0 1 1 0 1 0 1 1 1 1 0 0 1 0

SUBROUTINES

A subroutine (or subroutine subprogram) is an independent entity which can be separately compiled and whose variable names are independent of those in the main program or any other subprogram. A subroutine begins with a SUBROUTINE statement and must be terminated by an END statement. Control returns to the calling program at the first RETURN statement encountered or at the END statement. A CALL statement (Section III) is used to call a subroutine. No data type is associated with the name of a subroutine.

The subroutine differs from the function subprogram, which normally delivers only a single result to an expression. The subroutine subprogram may deliver any desired number of output results (including none). The subroutine returns values, if any, either through its arguments or by addressing elements in the common region. A value may be returned to any variable or array element in the subroutine's argument list.

The general form of the SUBROUTINE statement is:

$$\boxed{\text{SUBROUTINE } \underline{\text{subnam}} \ (arg_1, arg_2, \ldots, arg_n)}$$

Where: <u>subnam</u>	is the subroutine subprogram's symbolic name.
$(arg_1, arg_2, \ldots, arg_n)$	is a list of dummy arguments to be replaced by actual arguments from the corresponding positions of the statement calling the subroutine into operation. The list may contain 1 to 63 arguments or be omitted.

The SUBROUTINE statement is followed by the subprogram body, with the following restrictions:

1. The name of the subroutine appears only in the SUBROUTINE statement.
2. Names of dummy arguments must not appear in EQUIVALENCE, COMMON, or DATA statements in the subprogram.
3. The subprogram can contain any statement except FUNCTION, another SUBROUTINE, or a statement referencing the subroutine being defined.
4. The subroutine subprogram can define or redefine one or more of its arguments to return required additional values.

A CALL statement giving the subroutine name is written in the calling program at the point at which the programmer wishes to enter the subroutine. When the CALL statement is encountered, control is transferred to the subroutine named. Statements of the subroutine are then executed until a RETURN statement or the END statement is encountered. Control is then returned to the first executable statement following the CALL in the calling program. If the subroutine contains more than one RETURN statement (alternate coding branches), the first logically encountered RETURN gives control back to the calling program.

A CALL statement can simply transfer control to the subroutine or it can supply a list of actual arguments. The actual argumen*s, which constitute the argument list of the CALL statement, must agree in order, number, and type with the corresponding dummy arguments in the SUBROUTINE statement. Note that Hollerith constants must replace integer dummy arguments. An actual argument used as a subroutine reference may be:

1. A Hollerith constant;
2. A variable name;
3. An array element name;
4. An array name;'
5. Any other expression; or
6. The name of an external procedure.

If an actual argument is an external function name, the corresponding dummy argument must be used as an external function name. If the actual argument is a subroutine name, the corresponding dummy argument must be used as a subroutine name.

If an actual argument corresponds to a dummy argument which is defined or redefined in the referenced subprogram, the actual argument must be a variable name, an array element name, or an array name.

Execution of a subroutine reference results in substitution of actual arguments for dummy arguments. The actual argument is specified by name, except in the case of an expression (5, above). For such an expression, the association is by value rather than name. When actual arguments are substituted for dummy arguments, the first executable statement of the defining subprogram is executed.

If a dummy argument of a SUBROUTINE statement is an array name, the corresponding actual argument must be an array name or array element name.

Unless it is a dummy argument, a subroutine is also referenced by the appearance of its symbolic name in an EXTERNAL statement.

The characteristics of a subroutine subprogram are summarized in Table 6-3.

Table 6-3. Characteristics of the Subroutine Subprogram

Characteristic	Subroutine Subprogram
Originating source	Programmer
Relative size	At programmer's option (minimum: 3 statements)
Method of compilation	Compiled independently. Coding appears only once, no matter how many times called.
Number of output values returned	Any number, including zero.
Method of calling into use	Entered when name is encountered in a CALL statement in calling program.
Point to which control is returned after evaluation	First executable statement following the CALL statement in the calling program. (If the CALL statement is the terminal statement of a DO loop, control returns to the update portion of the DO loop.)

Example: XSUB2 is defined and can be called by the calling program, AMAIN, using actual arguments for the dummy arguments in the defining program.

```
     Defining Program                      Calling Program

SUBROUTINE XSUB2 (M, N)            Δ TITLE AMAIN
....                                 ....
....                                 ....
....                                 ....
RETURN                               CALL XSUB2(INTRST, IPRINL)
END                              25  ....
                                     ....
```

Actual arguments, INTRST and IPRINL, are substituted for M and N respectively when XSUB2 is called. When evaluation of the subroutine is complete, control will return to the statement labeled 25 in the calling program if this is an executable statement.

SPECIAL SUBROUTINES

Supplied with the compiler are a group of special subroutines to assist users in the execution of Fortran programs.

Test Subroutines for Simulated Hardware and Hardware Features

A test can be made to determine whether one of the four SENSE switches is ON or OFF, as follows:

Subroutine Call	Purpose
CALL SSWTCH (n, j)	When the call is executed, integer expression n is evaluated. If n is 1, 2, 3, or 4, the corresponding SENSE switch is tested. Integer variable j is set to 1 if the SENSE switch is ON and to 2 if it is OFF.

Four tests of simulated hardware conditions are made as follows:

Subroutine Call	Purpose
CALL DVCHK (j)	Integer variable j is set to 1 if a divide-check condition is detected when the call is executed. Otherwise, j is set to 2. The internal error condition is reset.
CALL OVERFL (j)	Integer variable j is set to 1 if floating-point exponential overflow occurs when the call is executed. Otherwise, j is set to 2. Any internal overflow indication is reset.

Subroutine Call	Purpose
CALL SLITE (n)	Integer expression n is evaluated. If n is zero, all simulated sense lights are set to OFF. If n is 1, 2, 3, or 4, the corresponding sense light is set to ON.
CALL SLITET (n, j)	Integer expression n is evaluated. If n is 1, 2, 3, or 4, the corresponding simulated sense light is tested, then set to OFF. If the sense light is in the ON state when tested, integer variable j is set to 1; otherwise, j is set to 2.

I/O Condition Test Subroutines

The three I/O test subroutines test for end-of-file and end-of-tape indications and for bad parity. The subroutines may be used in any combination. They must be called immediately after the I/O operation to which the test condition applies. If the condition the subroutine is intended to sense does occur and the subroutine has not been called, the program is automatically terminated. The purpose of each subroutine is given below.

Subroutine Call	Purpose
CALL PARITY (j)	Integer variable j is set to 1 if the peripheral operation encountered an uncorrectable error. Otherwise, j is set to 2.
CALL EOF (j)	Integer variable j is set to 1 if an end-of-file record is sensed. Otherwise, j is set to 2.
CALL EOT (j)	Integer variable j is set to 1 if the peripheral operation encountered an end-of-tape indication. Otherwise, j is set to 2.

I/O Subroutine REREAD

Subroutine REREAD provides a means whereby the same data can be read twice. A call to REREAD can be made between two READ statements. The subroutine causes the last record read by the first READ statement to be the first record read by the second READ statement. The subroutine call has the following form:

```
CALL REREAD (i)
```

Where: i is the code identifying the input device.

Example:

```
      READ (2, 15) A, B
      CALL REREAD (2)
      READ (2, 18) I, J
   15 FORMAT (2F5.1)
   18 FORMAT (2I5)
```

The subroutine will cause the data for A and B and the data for I and J to be read from the same data card.

Dynamic Dumping Subroutines

Three subroutines provide for dumping an area of memory at object time. The only differences in the three subroutines are in the arguments used to delimit the area dumped and the location to which the subroutine returns after the dump. The three calls to dump memory at object time are as follows:

Subroutine Call	Purpose
CALL PDUMP (v_1, v_2)	v_1 and v_2 are variable or array element names. When PDUMP is called, the area between and including v_1 and v_2 will be dumped. The address of v_1 does not need to be less than that of v_2, and v_1 can equal v_2 if the dump of a single variable is desired. After the dump, the subroutine returns control to the next executable statement after the call in the calling program.
CALL DUMP (v_1, v_2)	A call to DUMP will cause a dump to be taken in exactly the same way as a call to PDUMP. However, after the dump is taken, the subroutine transfers control to the job exit location.
CALL MDUMP (n_1, n_2)	n_1 and n_2 are decimal addresses between 0 and 262, 144. A call to MDUMP will cause a dump to be taken between and including the contents of n_1 and n_2. After the dump, the subroutine returns control to the next executable statement after the call in the calling program.

Examples: CALL PDUMP (A(1), A(10))

CALL DUMP (C, F(5, 5))

CALL MDUMP (8192, 17037)

Exit-to-Monitor Subroutine

A call to EXIT is the equivalent of a STOP statement. The subroutine is called as follows:

Subroutine Call	Purpose
CALL EXIT	When the call is encountered, an automatic exit to the monitor occurs, causing final termination of the job's object program.

SECTION VII
SYSTEM CONTROL CARDS

RUN-LEVEL AND JOB-LEVEL CONTROL CARDS

The input deck for any run begins and terminates with run-level control cards. These starting and terminating cards are a Console Call card and an *ENDRUN card, respectively. They are described in Section IX, "Operating Procedures." The Console Call card has several hardware options with which the programmer should become familiar, but these run-level cards are primarily the operator's responsibility.

A Fortran run can consist of a number of jobs separated by appropriate control cards that define each job. These control cards are the responsibility of the programmer. A job-level control card must have an asterisk in column 1. The card designator always begins in column 2.

CONTROL CARDS FOR STANDARD OPERATION

Standard operation is considered to be load-and-go mode, in which a job is compiled, the run tape is generated, and the job is executed. The next compilation, run-tape generation, and execution follows, etc., until the end of the run is reached. For standard operation without options, two cards are required: the *JOBID card in front of the source deck for the job, and the *DATA card at the end of the source deck. If there are data cards, the *DATA card is placed at the end of the source deck and ahead of any data cards. The deck appears as shown in Figure 7-1.

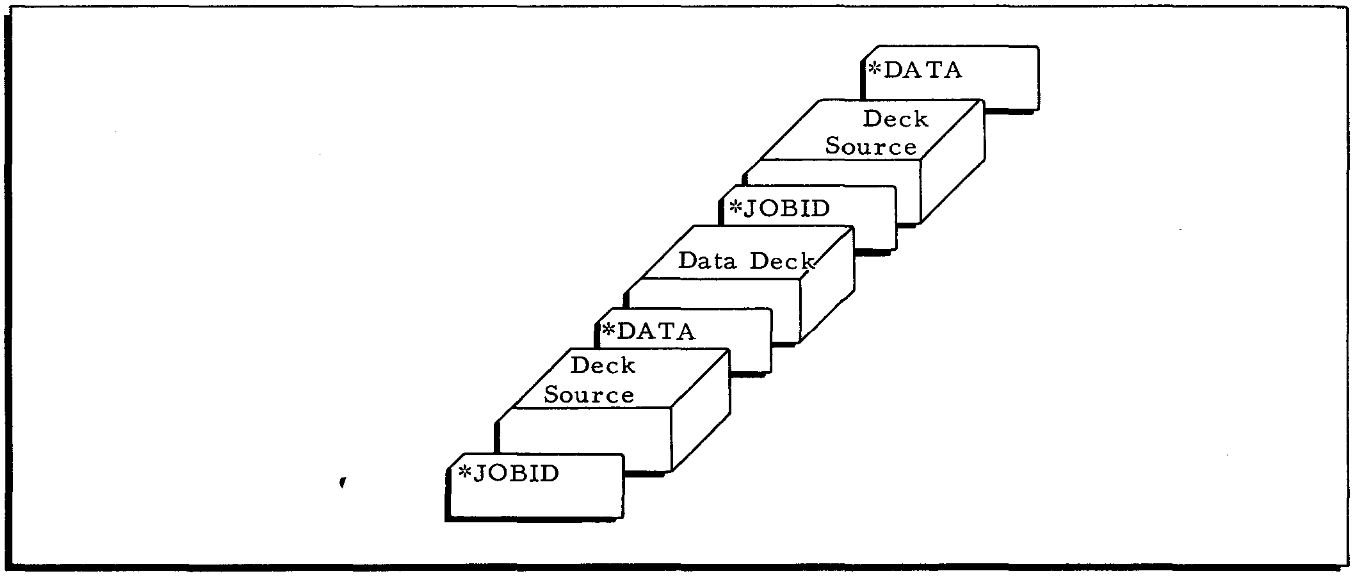


Figure 7-1. Input Deck for Standard Operation

NON-STANDARD OPERATION AND OPTIONS

Besides load-and-go, Fortran run modes include writing jobs onto a go-later tape for execution at a later time, execution of such batched go-later jobs, and processing Fortran II source decks into source-language decks acceptable to Fortran Compiler D. Some of these runs require control cards other than the standard load-and-go cards.

Within load-and-go runs, there are several options available to the programmer. Some can be included on the *JOBID card, and some require separate control cards. For example, diagnostic preprocessing of a source deck requires a separate control card, while punching a relocatable program deck is an option on the *JOBID card. The remainder of this section discusses each of the control cards and its options in detail.

*JOBID CARD

Every job must begin with a *JOBID card.[1] By itself, the *JOBID card usually indicates that the job following is to be compiled and then automatically executed (load-and-go mode). However, this interpretation can be modified by control cards that follow or precede the *JOBID.

Some or all programs of the job may have been previously compiled onto a stack tape or onto binary input decks. Presence of a *GET control card following a *JOBID indicates that the named program is on a stack tape and does not need to be compiled. It will be copied onto a binary program tape, generated onto the binary run tape in absolute form, and executed. Presence of a *BINARY control card following the *JOBID indicates that the binary deck following is to be copied onto the binary program tape, generated in absolute form, and executed. In both cases, compilation of source programs in the same job will not be inhibited.

A *DIAG card immediately preceding the *JOBID will bring in the preprocessor to check the job for source program errors. When a *SCREEN card precedes the *JOBID, that job and any following jobs in the run are processed by the Screen routine from Fortran II to Fortran Compiler D language format.

A *JOBID card has the form:

*JOBID   or   *JOBID, Option 1, Option 2, . . . , Option n

In the first form, no job options are included. *JOBID appears in columns one through six. In

[1] *JOBID cards are not required for Screen translation. However, if the programmer wishes his output deck from Screen to be in proper order for compilation, he should include *JOBID cards as required in the input deck as well as seeing that input programs are ordered according to the requirements for Fortran Compiler D.

the second form, *JOBID is followed by a comma in column 7. One or more options then follow; each option is separated from the one following by a comma.

When there are many options, a number of consecutive *JOBID cards may be used and the options divided among the cards. *JOBID must appear in columns one through six of each card. Job options may appear in any order following column 7 of the job card. Options may not appear beyond column 72 on any one card.

The options that may appear on the *JOBID card are the following:

Job name
Memory size for object program execution
Floating-point precision
Integer precision
Nonstandard assignment of card reader, printer, and punch
SAVE compiled programs by writing them onto a stack tape
PUNCH compiled programs into binary decks
Two listing options
Tape to be used as the common input device

Job Name Option

```
*JOBID, *jbnam
```

The use of an identifying name for each job is recommended. From one to six characters may be used for the job name including embedded blanks. The first character of the job name must be an asterisk. When a job name is included on a *JOBID card, the name will be printed on the listing; the asterisk will be suppressed. If no job name is included on the *JOBID card, the compiler assigns the tag *NONAM. A job name is required if a go-later tape is being generated.

Memory Size

```
*JOBID, Mdddddd
```

The letter M is followed by a 5- or 6-digit integer, representing the highest location in memory to be used during execution. Any location within the limits of memory can be chosen. Thus, a specification of M57344 means that all memory up to 57,344 characters can be used during execution.

The memory size option on the *JOBID card is used when programs are executed on a computer with a memory size different from that on which they were compiled. It can also be used when a programmer wishes to retain a program or programs in upper memory.

Floating-Point Precision

```
*JOBID, Fdd
```

The letter F is followed by one or two digits dd, where $2 \leq dd \leq 20$. The digit (or digits) specifies the maximum number of digits in the mantissa of a floating-point field. When not specified by the programmer on the *JOBID card, the maximum number of digits in the mantissa will be seven.

Integer Precision

```
*JOBID, Idd
```

The letter I is followed by one or two digits dd, where $3 \leq dd \leq 12$. The number indicates the maximum number of characters allotted to the integer field. When not specified, a standard value of three characters is allotted. Since integers are stored internally in binary, the option specified by the programmer is smaller than actual integer precision. See Appendix C for a tabular comparison of integer precision with the number of characters specified by the programmer.

Peripheral Device Assignments

```
*JOBID, IOiioopp
```

The input/output option permits specification of nonstandard assignments to the input, output, and punch devices. If the option is not used, the card reader is assigned to logical 02, the printer is assigned to logical 03, and the punch is assigned to logical 05.

When the I/O option is used, the letters IO are followed by the following information:

ii	is a two-digit logical address for the card reader ($01 \leq ii \leq 15$)
oo	is a two-digit logical address for the printer ($01 \leq oo \leq 15$)
pp	is a two-digit logical address for the punch ($01 \leq pp \leq 15$)

As an example, suppose that the user installation uses logical device 09 for the card reader and logical device 02 for the printer. A nonstandard option must then be included on all *JOBID cards as shown below:

```
*JOBID, IO090205
```

Note that even though a standard assignment is made to the card punch, all logical device addresses must be included if any one is changed.

SAVE Option

```
*JOBID, SAVE
```

The SAVE option is used to create or to add programs to a stack tape. This option requires that a fifth tape be mounted on logical tape drive No. 4 during compilation. When the SAVE option is encountered on the *JOBID card, an indicator is set in the communication region. At the same time that the run tape is generated, this indicator will cause the programs of the job to be written from the binary program tape (logical tape No. 3) to the end of the stack tape (logical tape No. 4). Note that in using the SAVE option, no deletion of duplicate-name programs on the stack tape will occur. The names of programs to be saved on a stack tape may not contain embedded blanks. This applies to Easycoder program names and Fortran main program names, which normally permit embedded blanks. Blanks in subroutine and function subprogram names are automatically suppressed.

PUNCH Option

```
*JOBID, PUNCH
```

The PUNCH option is used to create a deck of programs punched in relocatable binary form. This option requires that a card punch be initialized as part of the equipment for compilation. When the PUNCH option is encountered on the *JOBID card, an indicator is set in the communication region. At the same time that the run tape is generated, this indicator will cause the programs of the job to be punched from the binary program tape. If the SAVE and PUNCH options are requested for the same job, only the SAVE option will be performed.

Listing Options

There are three possible listings that can be produced by the system; an example of each can be found in Section VIII. If a binary run tape is generated for a job (absolute format), a memory map listing in absolute format is always printed out. The two options that can be specified on the *JOBID card are concerned with listings in relocatable format; these can be printed out even though a binary run tape is not generated.

The first listing option is as follows:

```
*JOBID, NOLIST
```

A NOLIST option will inhibit the printing of a relative memory map listing at compile time. This listing is printed out unless a NOLIST option is encountered and contains the address, relative position, and the name of each variable and constant. The memory map is so divided as to indicate whether the variable or constant being listed is contained in a common block, labeled common block, or noncommon block.

The second listing option is as follows:

```
*JOBID, LIST
```

A LIST option will cause the printing of a pseudo-Easycoder listing of the instruction string generated by compilation. Each line of the listing is edited to resemble a line of an Easycoder listing. The LIST option is most commonly used for program checkout and maintenance, but users may occasionally find the pseudo-Easycoder listing useful for tracing subtle source program errors. If the object program exceeds 8.5K characters of memory, the LIST option will be inhibited.

Tape Input

```
*JOBID, TAPEIP
```

This option informs the compiler that the common input device for the rest of the run is a card-image tape mounted on logical tape drive No. 5. When a TAPEIP option is encountered, not just the job following but the remainder of the run must use tape input. Input jobs on cards to be included in the same run must precede the jobs to be read in from tape. The tape on logical tape drive 5 must not be rewound during a TAPEIP run. In addition, no diagnostic preprocessing to tape can occur in the run after the TAPEIP option is encountered. (See the description of the *DIAG card with tape option below.)

Sample *JOBID Card with Options

```
*JOBID, I7, F20, IO070305, *SAMPLE, PUNCH, LIST, M22527
```

Shown above is an example of a *JOBID card with several of the possible options. The name of the job is SAMPLE. The job requires that the maximum floating-point accuracy (20 characters in the mantissa) be used, together with integer specification of seven characters (or integer precision up to 12 decimal digits). Logical input/output devices are given by option, since the card reader is assigned as logical device 7. Programs within the job are to be punched into relocatable binary decks at run-tape generation time. At the same time, a pseudo-Easycoder listing is to be printed in addition to the usual memory map listings. At object-program time, memory up to 22,527 characters is to be used.

*SCREEN CARD

When a user wishes to convert a source-program deck from Fortran II to Fortran Compiler D format, use of Screen conversion will result in a new source-program deck with translated I/O statements and statements containing function names. A *SCREEN card in the input deck indicates that all jobs following are to be translated. The compiler monitor will transfer control to F2TOF4, the Screen conversion routine.

A *SCREEN card has one of the following forms

*SCREEN or *SCREEN x

*SCREEN appears in columns 1 through 7. Any nonblank character anywhere in columns 8 through 72 will cause sequential numbering of the lines of each job in the listing produced by Screen and the cards of the new source-program deck. If columns 8 through 72 are blank, the listing lines and card deck will be numbered in the same way as the input deck.

A Screen run is terminated by the *ENDRUN card that signals the end of the run. Therefore, if a deck to be screened is included with compilation or preprocessor jobs, the Screen deck must be the last deck in the run.

*DIAG CARD

When a user wishes to check a job for source-program errors without using compilation time, he can use the diagnostic preprocessor. A *DIAG card in the input deck indicates that the job following is to be processed for possible source-program errors. The compiler monitor will transfer control to ACCPRA, the first segment of the diagnostic preprocessor routine. There are two options in using the diagnostic preprocessor. In the first option, the *DIAG card has the form:

*DIAG

*DIAG appears in columns 1 through 5 of the control card. This brings in the diagnostic preprocessor, which will examine source-program statements in the job following for possible diagnostics and print out a listing of all programs (source programs and binary programs if included) together with diagnostics for the source programs. An example of a source-program listing with diagnostics is given in Section VIII.

When the diagnostic preprocessor job has been completed, control returns to the compiler monitor to determine how the next job should be processed.

In the second option, the *DIAG card has the form:

```
*DIAG, T
```

*DIAG appears in columns 1 through 5 of the control card. A comma appears in column 6. A T anywhere in columns 7 through 72 indicates that preprocessing is to be to tape rather than to a listing. For this option a tape is mounted on logical tape drive 5. As the preprocessor scans the job for source-program errors, it writes all programs of the job (whether source programs or binary programs) onto tape 5. When the end of the job is encountered, tape 5 is rewound and a check for job fatality is made. Any error diagnosed by the preprocessor will cause job fatality. If the job is fatal, it is copied from tape 5 onto a printed listing with source-program diagnostics. Control is then transferred to the compiler monitor to process the next job.

If no errors were diagnosed in the job, control is passed to the compiler monitor, together with parameters informing the monitor to process the next job using tape 5 as the input device. Thus, by use of the tape option, a job can be preprocessed, compiled, and executed in one operation.

Any *DIAG card containing punches other than *DIAG, T will cause diagnostic preprocessing to a listing only.

*GET CARD

A *GET card specifies that the program named on the card is to be copied from the stack tape (logical tape drive 4) onto the binary program tape. Use of a *GET card presumes that a fifth (stack) tape is included in the compilation run. A *GET card has the form:

```
*GET, program name
```

*GET, appears in columns 1 through 5. The program name can appear anywhere in columns 6 through 72 and must be punched exactly as it appears on the stack tape. When a *GET card is encountered, the stack tape will be searched forward until the named program is found or until an end of file is found. If an end of file is found first, the tape will be rewound and searched until the named program is located. If, for any reason, two programs of the same name appear on the stack tape, the first program located will be the one copied to the binary program tape. If the named program cannot be found on the stack tape at the end of the second pass over the tape, a "job fatal" diagnostic is issued and processing continues with the next job.

*BINARY AND END CARDS

A *BINARY card indicates to the monitor that the program deck following the card is in relocatable binary form. The binary program deck will be copied onto the binary program tape. A *BINARY card has the form:

```
*BINARY
```

*BINARY appears in columns 1 through 7. If the binary deck is out of order, a "job fatal" diagnostic is issued and processing continues with the next job.

The programmer must terminate the binary deck with an END card to signal the end of binary input. The characters END must appear in columns 7 to 9 of this card as shown below.

```
7 9
↓ ↓
END
```

*CHAIN CARD

Chaining can be used when a job is too large to be treated as one memory load at object time. Such a job is divided into groups of programs. Each group of programs is called a chain and constitutes a separate memory load at object time. In the input deck each chain of programs is preceded by a *CHAIN card. A maximum of 30 chains is permitted in a single job. A *CHAIN card has the form:

```
*CHAIN, x
```

*CHAIN, must appear in columns 1 through 7. The x shown in the card form represents a single alphanumeric character that must appear somewhere in columns 8 through 72 to identify the program chain that follows. If the first chain of a job is not called in by later chains, it need not have a *CHAIN card. However, all subsequent chains must have *CHAIN cards.

*ALTER CARD

An *ALTER card contains one or more of four options. The options are SAVE, PUNCH, LIST, and NOLIST. The options are identical to those used on the *JOBID card. However, *JOBID options determine options for all programs within the job. An *ALTER card determines options only for the program that it precedes. Thus, an *ALTER card can be used to change initially set job options for a single program within the job. An *ALTER card has the form:

```
*ALTER, Option 1, ..., Option n
```

*ALTER, appears in columns 1 through 7. One or more of the options may appear in any desired order in columns 8 through 72; options must be separated by commas. If the SAVE and PUNCH options are requested for the same program, only the SAVE option will be performed.

*DATA CARD AND 1EOFΔ CARD

```
*DATA
```

The *DATA card is placed at the end of all program decks for a given job whenever execution of the job is desired. A *DATA must be present to trigger job execution. *DATA appears in columns 1 through 5.

When a job includes data to be read under the control of the object program, all data cards for the job must follow the *DATA card and will be read by the object program in the order in which they follow the *DATA card.

The absence of a *DATA card following program decks for a given job will inhibit execution of the job. The run tape generator will process the job, allocate memory, collect called library functions and subroutines from the compiler system tape, and satisfy any SAVE or PUNCH option. When compilation is complete, instead of attempting execution, the compiler passes control to the monitor to process the next job.

If a job proves fatal, when a *DATA card is encountered, the data cards are bypassed and the next job is initialized.

When a data deck follows a *DATA card, the programmer has the option of testing an end of file following the data cards. An end-of-file card following the data cards will cause the end-of-file record to be generated following the data deck. This card has the form:

```
1EOFΔ
```

Columns 1 through 5 contain 1EOFΔ.

*ENDATA CARD

```
*ENDATA
```

The *ENDATA card is used only when executing a go-later tape. In this run mode, a series of jobs previously written onto tape in relocated form are to be executed. For each job to be executed, the card reader must contain a console call card giving the name of the job on the go-later tape, followed by data for the job, and terminated by an *ENDATA card. The *ENDATA card indicates the end of input data for that particular job. *ENDATA appears in columns 1 through 7 of the card.

*DUMP CARD

```
*DUMP
```

If the programmer wishes to take a terminal dump when a job is executed, a *DUMP card must follow the *JOBID card. When the *DUMP card is encountered, the compiler monitor sets a dump indicator in the communication region. When the job is executed, an alphanumeric and octal dump of all memory will be printed. *DUMP appears in columns 1 through 5 of the card.

COMMENT CARDS

Any card with an asterisk in column 1 that is not immediately followed by JOBID, SCREEN, DIAG, DATA, ALTER, GET, BINARY, CHAIN, or ENDRUN will be treated as a comment card. The contents of the card will be printed on the listing. Comments may appear anywhere in columns 2 through 80 of such cards. (Note that these comment cards are not within an individual program. For comments within a source program, see page 1-4.)

When the compiler monitor begins scanning cards after run initialization, it searches for a *JOBID, *SCREEN, or *DIAG card. Any asterisk card, control or otherwise, encountered before the compiler monitor locates a *JOBID, *SCREEN, or *DIAG card will be treated as a comment card and will have no effect upon compilation and execution of the run.

Card-Image Tape Input

At installations not using a card reader, system control information and source programs can be read in from a card-image tape mounted on logical tape drive No. 5. The initial console call for the run is keyed in by the operator at the console. The console key-in takes the place of a TAPEIP option on a *JOBID control card in a card reader to initiate a run using tape input. See Sections VIII and IX for additional information.

SECTION VIII
SYSTEM DESCRIPTION

SYSTEM SUMMARY

System Modules

The system supplied to each installation consists of the software modules described in Table 8-1. These modules are supplied on a CST (Compiler System Tape). A list of the complete contents of this tape is found in Appendix F.

Table 8-1. System Modules

Module	Function
Compiler	Translates Fortran source-program units into relocatable machine language and writes them on a BPT (binary program tape).
Run-Tape Generator	Accepts relocatable machine-language program units from several sources (BPT, Fortran library, and stack tape in relocatable machine-language format), and relocates them into loadable executable jobs on a BRT (binary run tape). There are two run-tape generators, one for the 3-character address mode and one for the 4-character address mode.
Execution Package	Consists of loader-monitor, floating-point package, fixed-point package, object I/O packages, etc., required for program execution. The module is segmented so that only the required segments are in memory for any given job. Two loader-monitors are included, one for loading programs in the 3-character address mode and one for loading programs in the 4-character address mode.
Fortran Library	A library of Honeywell-supplied Fortran mathematical functions and special subroutines in relocatable machine language suitable as input to the run-tape generator.
Diagnostic Preprocessor	Checks for source programming errors and issues diagnostics.
Screen Routine	Converts Fortran II I/O statements and function names to Fortran Compiler D source program form.
Debugging Aids	Source-program listing, memory map of relocatable machine coding, object memory map, generated pseudo-Easycoder listing, and dynamic and terminal memory dumping facilities.
Punch	Punches program units onto cards in relocatable machine language suitable as input to the run-tape generator.
Stack	Writes program units onto an optional tape in relocatable machine language suitable as input to the run-tape generator.

Run Options

There are four run options for Fortran D. They are:

Load-and-Go Mode

Writing a Go-Later Tape

Executing a Go-Later Tape

Screen Conversion

During load-and-go operation, jobs are serially compiled, relocated, and executed. The load-and-go mode uses the system modules for compilation, run-tape generation, execution, and debugging aids. Optionally, the Fortran library, punch, and stack can be used during load-and-go operation. Diagnostic preprocessing jobs can be interspersed with load-and-go jobs and do not require a separate run.

Writing a go-later tape consists of compiling and relocating a group of jobs without executing them. The tape on which the relocated jobs are written is saved for later execution. Writing a go-later tape uses the system modules for compilation, run-tape generation, and debugging aids. Optionally, the Fortran library, punch, and stack can be used during this run mode, and diagnostic preprocessing jobs can be interspersed.

Executing the jobs batched in relocated code on a go-later tape represents another run option. Only the execution routines are used in this run mode.

Conversion of Fortran II I/O statements and function names to Fortran D format uses only the Screen routine and constitutes a run mode. Optionally, a Screen run can follow a load-and-go run without operator intervention if the load-and-go input does not terminate with an *ENDRUN card or card image. The presence of a *SCREEN card or card image terminates the load-and-go processing.

Tape assignments for all runs are described in Section IX.

STANDARD FORTRAN PROCESSING — LOAD-AND-GO OPERATION

The standard mode of processing Fortran source programs is called load-and-go operation. In the load-and-go mode, program units that make up a job are compiled into relocatable machine language and written onto a work tape, which becomes the binary program tape (BPT). Then the run-tape generator collects these program units on the BPT, together with any called functions or subroutines from the Fortran library and any program units called from the stack tape, relocates them, and writes them onto the binary run tape (BRT). The job is then immediately executed. Then the next job is compiled, relocated, and executed, etc., until the end of the run is encountered.

A minimum of six peripheral devices and a maximum of 15 can be used for load-and-go operation, as well as other modes of execution. A simple flow diagram for the minimum system is shown in Figure 8-1.

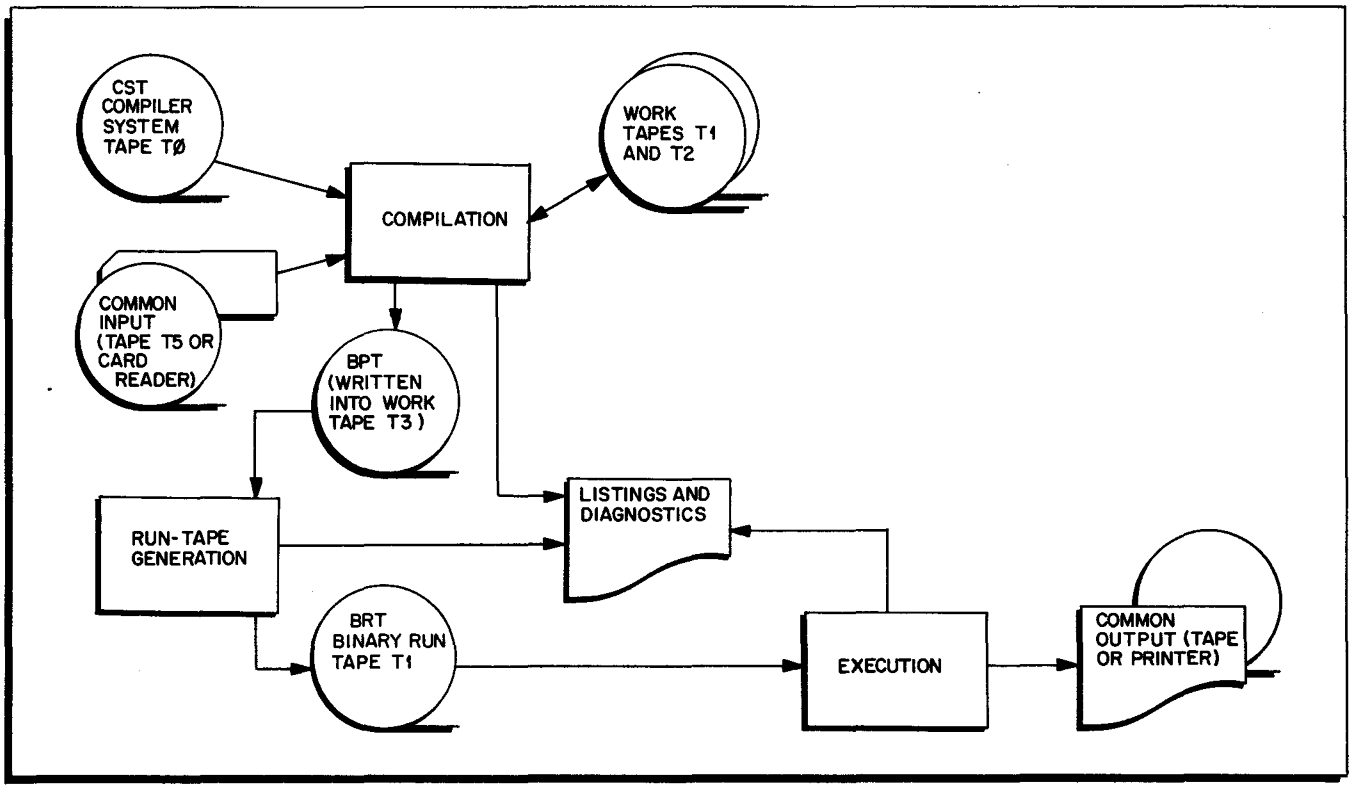

CST COMPILER SYSTEM TAPE TØ
COMPILATION
WORK TAPES T1 AND T2
COMMON INPUT (TAPE T5 OR CARD READER)
BPT (WRITTEN INTO WORK TAPE T3)
LISTINGS AND DIAGNOSTICS
RUN-TAPE GENERATION
BRT BINARY RUN TAPE T1
EXECUTION
COMMON OUTPUT (TAPE OR PRINTER)

Figure 8-1. Standard Load-and-Go Flow Diagram

The input deck for a load-and-go run is shown in Figure 8-2. The Console Call card and *ENDRUN card begin and terminate the run, respectively, and are the responsibility of the operator. A *JOBID card must begin each job deck, and a *DATA card must terminate the source programs. If there are data cards, the *DATA card precedes them. These control cards are the programmer's responsibility.

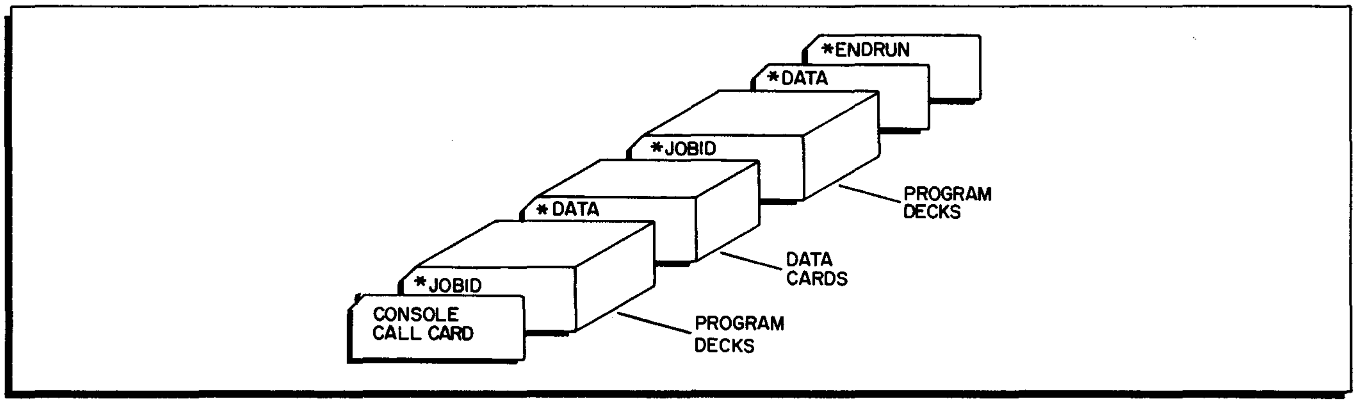


Figure 8-2. Input Deck for Load-and-Go

Chaining a Load-and-Go Job

When it is probable that a job will overflow memory at execution time, the job should be divided into two or more memory loads. Each memory load (or chain) begins with a *CHAIN control card that names the chain. An example of chained input is shown in Figure 8-3. Programming tips on chaining are given in Section X.

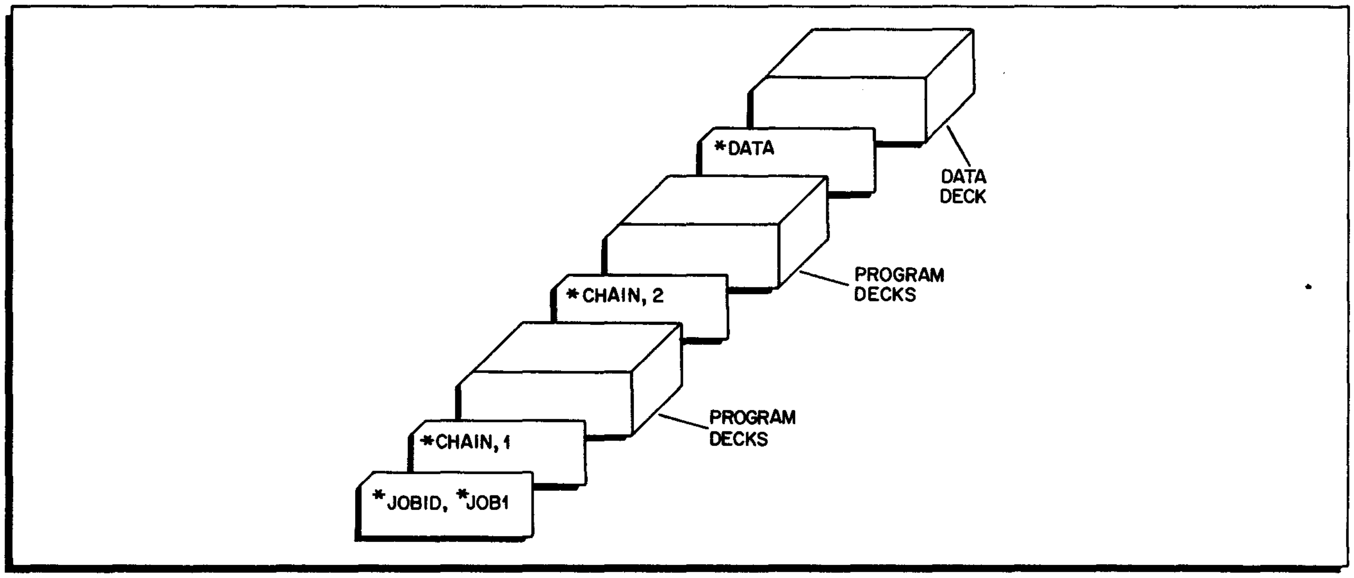


Figure 8-3. Job Divided into Two Chains of Program Units

GO-LATER – BATCHED JOB PROCESSING

As an alternative to load-and-go processing, Fortran D System permits the programmer to compile and relocate a series of jobs onto a BRT. The BRT is saved, and the jobs on it can later be executed as a separate run.

Writing a Go-later Tape

Compiling and relocating jobs onto a go-later BRT is a run-level processing mode, specified by an option on the Console Call card. There are two options that trigger the run. One indicates that a new go-later tape is to be created from a work tape, and the other specifies that jobs are to be added to an already existing go-later tape. Further details on these options can be found in Section IX, "Operating Procedures."

Every job to be written onto a go-later BRT must have its identifying job name specified on the *JOBID card. The job name appears on the go-later tape as a six-character name, the initial character of which must be an asterisk. If a job name longer than six characters is specified on the *JOBID card, the name will be truncated at the right-hand side as shown below:

*JOBID Card with Job Name	Job Name on Go-Later Tape
*JOBID *SAMPLE	*SAMPL

Figure 8-4 shows a sample input deck for writing a go-later tape. Figure 8-5 gives the minimum system configuration for writing a go-later tape.

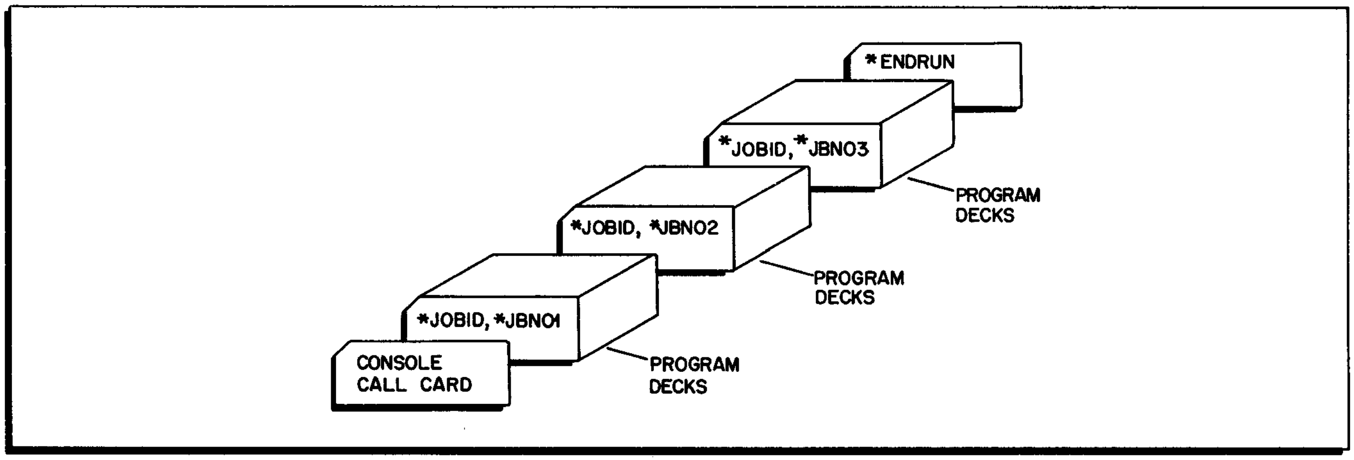


Figure 8-4. Sample Input Deck to Write Go-Later Tape

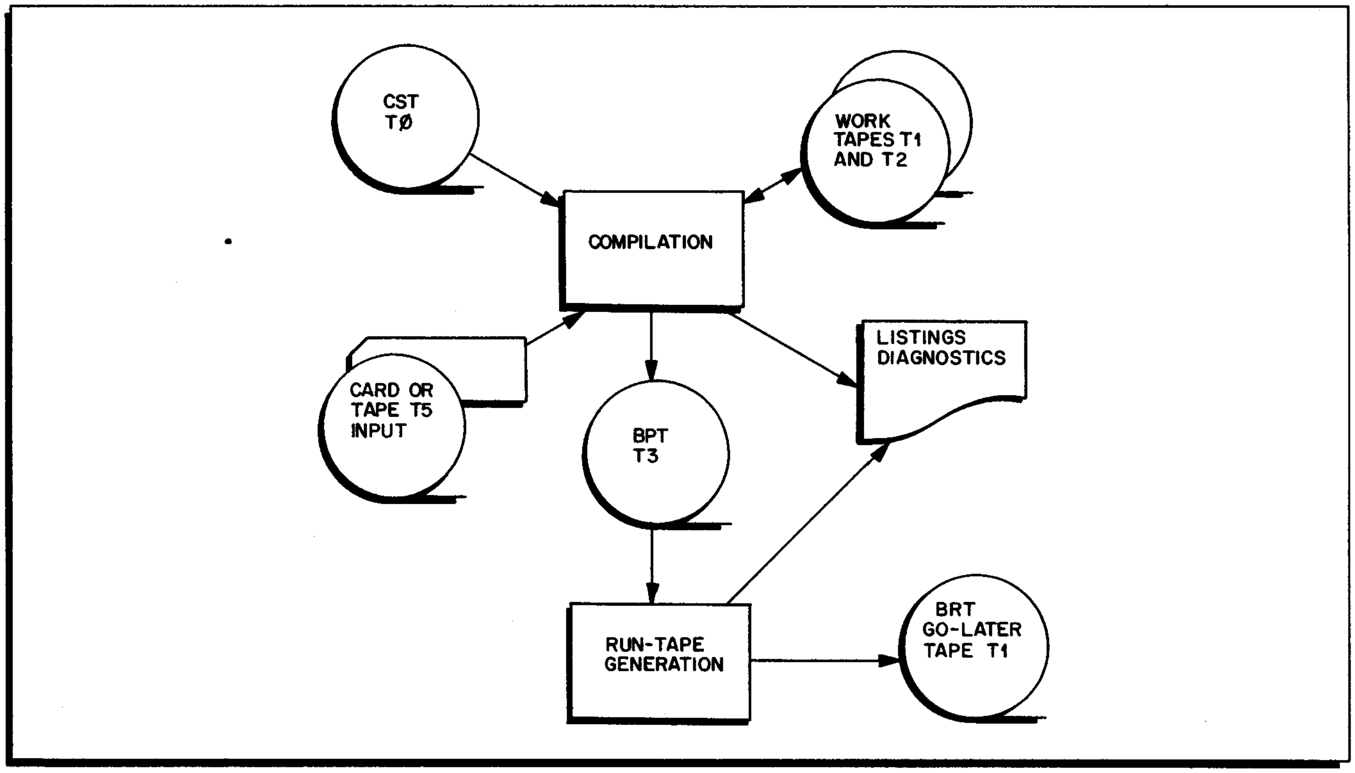


Figure 8-5. Flow Diagram to Write Go-Later Multi-job Tape

The console call, described in Section IX, provides three go-later options. One permits jobs to be written onto a BRT that already contains go-later jobs. The second option permits a new go-later tape to be created, and the third option is used to reposition the go-later tape to the last good job if a run restart is necessary. There is, in addition, a SENSE switch option that allows the use of card input for creating a go-later tape and using tape input for later execution. This option is described in Section IX.

Executing a Go-later Tape

Executing batched jobs from a go-later BRT is a run-level processing mode, triggered by a console call that contains the name of the first job to be executed from the go-later tape. The programmer can select a job or several jobs on the go-later tape that he wishes to run, since the individual jobs each have identifying names.

The common input device for the run contains a series of console calls, giving the names of jobs on the go-later tape. Each console call is followed by the data for the job. Data for each job must be terminated by an *ENDATA card or card image. When the first *ENDATA card is encountered, the system accepts the next console call from the common input device, searches the go-later tape for the specified job name, and executes the job. Execution of selected jobs continues until an *ENDRUN card is encountered. Greater efficiency in executing a go-later tape is achieved if the jobs are executed in the order in which they appear on the tape, thus avoiding extra search time.

When a tape unit is used as the common input device, the first console call can be keyed in at the console, but the remaining job-identifying console calls can appear as card images.

NOTE: If the off-line printing and/or punching options are used when generating a go-later tape, they must be indicated on the Console Call cards used when running the go-later tape.

A sample input deck for executing a go-later tape and a flow diagram for go-later execution are shown in Figures 8-6 and 8-7.

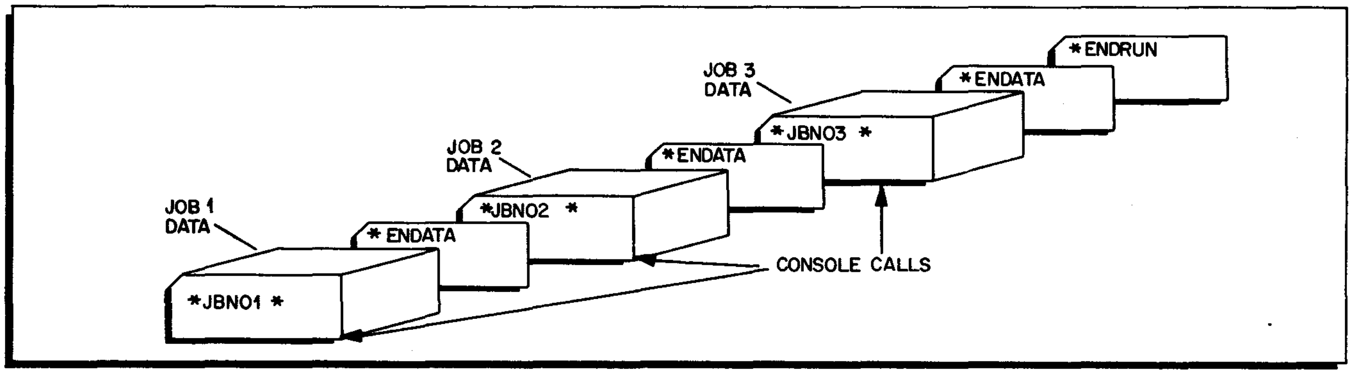


Figure 8-6. Sample Input Deck to Execute Go-later Tape

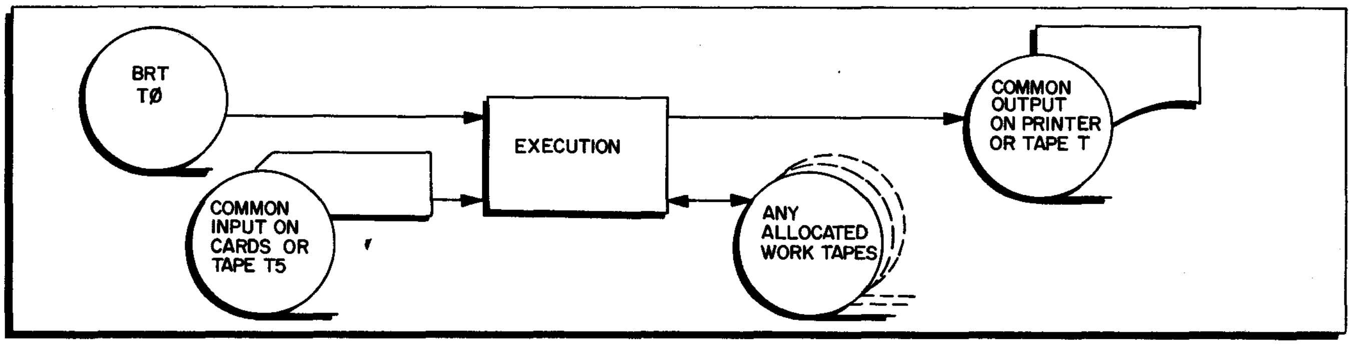

BRT
TØ
EXECUTION
COMMON OUTPUT ON PRINTER OR TAPE T
COMMON INPUT ON CARDS OR TAPE T5
ANY ALLOCATED WORK TAPES

Figure 8-7. Flow Diagram to Execute Go-later Tape

SYSTEM OPTIONS

Stack Tape

Compiled program units in relocatable form can be written onto a fifth tape, called the stack tape, during the course of compilation and run-tape generation. The stack tape, on logical tape drive 4, is saved. Writing the program units onto the stack tape does not inhibit their relocation and execution during a load-and-go run or their relocation onto the go-later tape when writing a go-later tape. Use of the SAVE option on the *JOBID card causes that job to be placed on the stack tape. Use of the SAVE option on the *ALTER card triggers the writing of the program unit following onto the stack tape.

Program units previously saved can be called for relocation by use of a *GET card containing the program unit name. The name of the program unit as it appears on the *GET card must exactly match the program unit name as it appears on the stack tape. To insure uniformity of names, the programmer must suppress all embedded blanks when writing the name of a program unit to be saved on the stack tape. Use of the *GET option is permitted during load-and-go operation and when writing a go-later tape.

The console call, described in Section IX, provides two stack tape options. One permits a new stack tape to be created during a run, and the other permits jobs to be written onto an already existing stack tape.

Punch Option

Compiled program units can be punched onto cards in relocatable format during the course of a load-and-go run or while writing a go-later tape. Punching program units does not inhibit their relocation by the run-tape generator or their execution if the run is load-and-go. The PUNCH option on the *JOBID card causes punching of all program units in the job. The PUNCH option on the *ALTER card causes the program unit that follows to be punched. If both the SAVE and the PUNCH option appear on a *JOBID or *ALTER card, only the SAVE option is processed.

Use of a *BINARY card preceding a deck created by the PUNCH option causes the deck to be relocated onto the BRT by the run-tape generator. Use of the *BINARY option is permitted when writing a go-later tape or during a load-and-go run.

Jobs Containing *GET and *BINARY Program Units

A job can consist primarily of previously compiled programs in relocatable form if the job contains at least one source program to be compiled. The source program is placed last in the job and must allocate maximum common storage for the job and reference logical device numbers of any I/O devices used in the job. Figure 8-8 shows a job input deck containing previously compiled program units and terminating with a source program containing common and I/O statements.

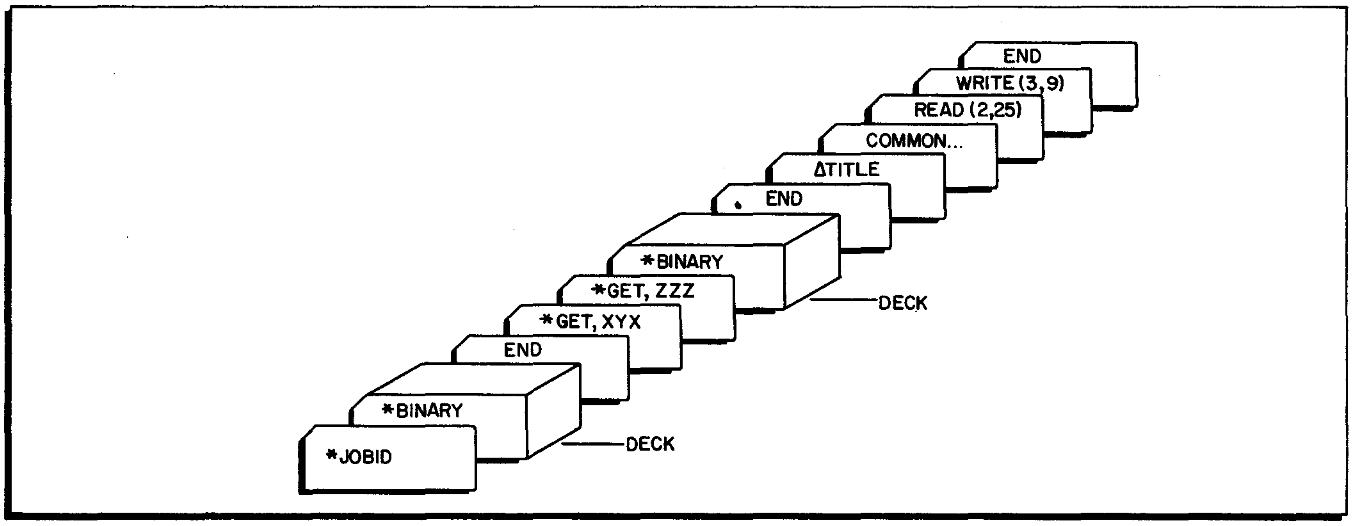


Figure 8-8. Job Containing Previously Compiled Program Units

Common Input Device

The common input device can be a card reader or a tape unit. If a card reader is the input device, the run begins with the Console Call card, as described in Section IX.

Tape drive No. 5 is used for common input from tape. This card-image tape can be written off-line as a series of jobs by Simultaneous Media Conversion C. The console call is keyed in by the operator at the console; or if a card reader is available, the Console Call card can be read in from the card reader followed by a *JOBID card with a TAPEIP option.

A second method exists whereby jobs can be read from tape 5 following diagnostic preprocessing of a job onto the tape. If no source-program errors are found during diagnostic preprocessing and a tape option was specified, the diagnostic preprocessor will produce the job on tape and load-and-go processing will automatically follow. This diagnostic preprocessing option is described in detail later in this section.

Since the diagnostic preprocessor uses tape 5 as a work tape, the option for diagnostic preprocessing to tape is not permitted when using tape 5 as the input device (TAPEIP).

Common Output and Common Punch Device Options

The common output device can be a printer or a tape unit. A punch or common punch tape is optional for Fortran D execution as well as the previously described PUNCH option. Note that when the common output device is a tape unit, it can be used as the common punch tape as well, with printed and punched output interspersed.

Bypassing Execution

During load-and-go operation, omission of a *DATA card following the source deck of a job causes control to return to the compiler monitor after the job is written onto the binary run tape. Processing of the next job then begins. This option is used to obtain debugging information from compilation and run-tape generation.

Diagram of System Options

Figure 8-9 is a diagram of a load-and-go run with all system options shown.

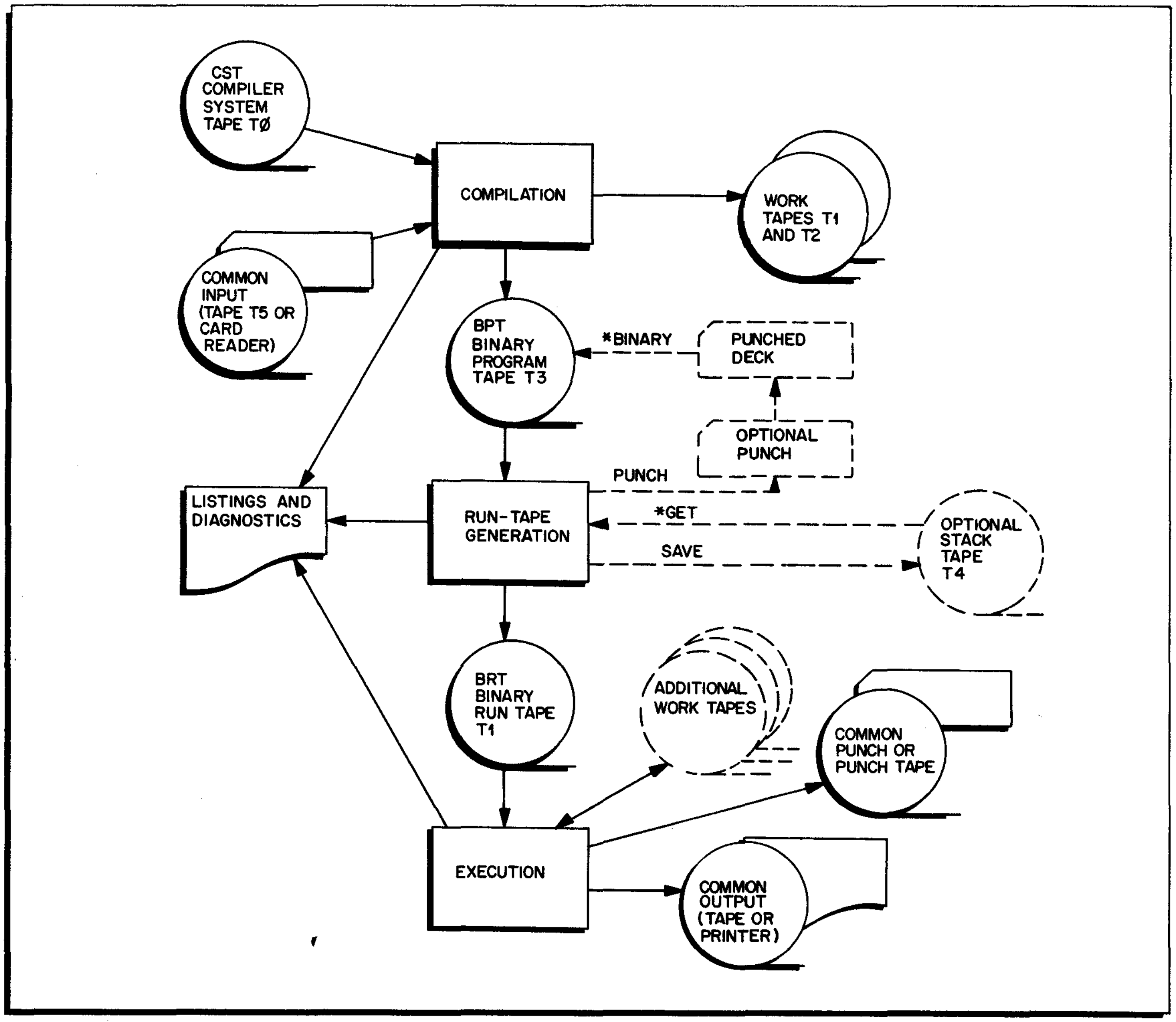


Figure 8-9. Load-and-Go Run With System Options

DEBUGGING AIDS

Source-Program Listing

During compilation, a listing of the source program is always generated. A sample source-program listing is shown in Figure 8-10.

Relocatable Memory Map

During compilation, a memory map giving the address, relative position, and symbol name of each variable or constant is printed. The relocatable memory map is printed unless the NOLIST option is specified on the *JOBID card. A sample relocatable memory map is shown in Figure 8-11.

The memory map indicates in a heading whether the data following is in noncommon, unlabeled common, or labeled common blocks. Under this heading is a two-column listing with the headings: ADDRESS, RELATIVE POSITION, and SYMBOL. Symbols are sorted in ascending order by address value.

For each labeled common block, the name, size, and base of the block are included in the header information. Constants in the noncommon area are only those explicitly defined within the source program. Constants generated by the compiler do not appear.

Logical constants for true and false are given as .TRUE. and .FALSE. Integer constants are given as = value_{10} (for example, =9). Floating constants are given as .mantissa E ± exponent (for example, .7E-02). Hollerith and octal constants are given as = value in Hollerith or octal (for example, =25KC).

No explicit information indicates whether a variable has been defined as a dummy variable or a dimensioned variable. However, dummy variables appear as the first set of noncommon variables with three characters allocated to each. Dimensioned variables can be identified by the amount of storage allocated to each.

All addresses are given in relocatable form, relative to the base 10000_8, and are not the absolute addresses assigned in execution. The absolute addresses can be computed by adding the address given under REL POS to the base address of the BASE LOCN DATA on the object memory map described in the next paragraph.

```
FORTRAN      200         SOURCE LISTING AND DIAGNOSTICS          PROGRAM: XSUB1

001                SUBROUTINE XSUB1(N)
002                COMMON /XSUB1/I(10)
           C       WHEN N=1 LOADS I AND SINGLES M1,M2.
           C       WHEN N=2 CHECKS I AND SINGLES M1,M2.
003                IF (N-1) 100,100,200
004            100 WRITE (3,1) N
005                DO 110 KK=1,10
006                I(KK)=KK
007            110 WRITE (3,2) KK,I(KK)
010              2 FORMAT (3H I( I3,2H)= I5)
011              1 FORMAT (14H XSUB1 WITH N= I3)
012                M1=-1
013                M2=-2
014                WRITE (3,3) M1,M2
015                RETURN
016            200 WRITE (3,1) N
017                DO 210 KK=1,10
020            210 WRITE (3,2) KK,I(KK)
021                WRITE (3,3) M1,M2
022              3 FORMAT (4H M1= I5,4H M2= I5)
023                RETURN
024                END
```

Figure 8-10. Source-Program Listing

```
XSUB1                   MEMORY MAP

                NON COMMON DATA

ADDRESS   REL-POS   SYMBOL                                 ADDRESS   REL-POS   SYMBOL

10047     00030     N                                      10065     00046     =10
10054     00035     =1                                     10135     00116     M1
10057     00040     =3                                     10140     00121     M2
10062     00043     KK                                     10143     00124     =2


                LABELED COMMON BLOCK: XSUB1                BASE: 77704   SIZE: 00036

ADDRESS   REL-POS   SYMBOL                                 ADDRESS   REL-POS   SYMBOL


77706     00002     I

          SUBPROGRAMS REFERENCED :
          ACBFXP   ACBOIO
```

Figure 8-11. Relocatable Memory Map

Object Memory Map

Whenever a binary run tape (BRT) is generated, an object memory map is printed. This listing is not optional. The object memory map gives base locations in absolute code for:

1. Data;
2. Program instructions of all chains in a job; and
3. Any I/O devices that have nonstandard logical address assignments.

The object memory map is useful in debugging object coding. A sample object memory map is shown in Figure 8-12.

```
*DATA
                                      OBJECT MEMORY MAP
                  PROGRAM/DATA AREAS    BASE LOCN DATA    BASE LOCN PROG
                CHAIN 01
                  UNLAB COM                04527
                  LABEL COM                04540
                   ACBFXP                  04634              04634
                   ACBOIO                  06064              06304
                   XMAINS                  11356              11655
                   XMAIN2                  12203              12260
                   XSUB2                   12367              12630
                   XSUB1                   13725              14105
                   BCDCON                  15062              15143
                   INTCON                  20217              20217
                   IODIAG                  21302              21302
                   ACBFLO                  21320              21320
                   IABS                    21733              21750
                   ACBFPP                  22060              22060

                             HIGHEST LOCATION                 24306
```

Figure 8-12. Object Memory Map

The object memory map has three columns. The first gives the name of the item concerned — program, common or labeled common data area, or buffer device having a nonstandard assignment. The second column contains the base location for data. In this column are the base locations of program data, common, and labeled common areas. In addition, the base address of any buffer device having a nonstandard address will appear in this column. The third column is used only with programs and gives the base address of each program instruction string. Information is printed in the following order:

1. Any nonstandard buffer devices;
2. Chain designation;
3. Unlabeled common area;
4. Labeled common area; and
5. Programs in the chain.

If there is insufficient memory to execute a given chain, the object memory map prints at the end of the chain:

INSUFFICIENT MEMORY xxxxx CHARACTERS NEEDED.

Machine-Code (Pseudo-Easycoder) Listing

During compilation, a pseudo-Easycoder listing is printed if the LIST option is specified on the *JOBID card. The compiler-generated instruction string is printed, using the programmer-defined variable names whenever possible. The pseudo-Easycoder listing is usually used in program checkout and maintenance, but some users may occasionally find it useful for checking source-program errors. A sample pseudo-Easycoder listing is shown in Figure 8-13.

```
      XSUB1          MACHINE CODE LISTING

BEGADD AL MACHINE CHARACTERS  R         LOC       OPCODE OPERANDS AND VARIANTS

010177 R                                $BEGIN    RESV   0
010177  W 2401022370                              SCR    $BEGIN+20,70
010204  W 65010224                                B      $BEGIN+21
010210  W 1001001700000257                        EXM    REGIONI+15,X1-2,57
010220  W 65000000                                B      0
010224  W 1071022101004757                        EXM    (IFN#004-62),N,57
010234  W 2401022367                              SCR    $BEGIN+20,67
010241  W 1000000201001757                        EXM    X1-2,REGIONI+15,57
010251  W 14710047004453                          MCW    (N),$FXPTA+11
010260  W 35010054004453                          BS     =1,$FXPTA+11
010267  W 5401031700445102                        BCC    IFN#004,$FXPTA+9,02
010277  W 33004462004453                          C      $FXPTZ,$FXPTA+11
010306  W 6501031742                              B      IFN#004,42
010313  W 65010664                                B      IFN#016
010317 R                                IFN#004   RESV   0
010317  W 14010057004455                          MCW    =3,$LODCH
010326  W 65004757                                B      ACBOIO
010332  W 01010620            I                   DSA    =10+17,20
010336  W 14710047004453                          MCW    (N),$FXPTA+11
010345  W 43                  W                   CSM
010346  W 65004757                                B      ACBOIO
010352  W 01010677            I                   DSA    =10+17,77
010356  W 15010054010062                          LCA    =1,KK
010365  W 14010062004453                          MCW    KK,$FXPTA+11
010374  W 65004756                                B      ACBFXP
010400  W 01017630            I                   DSA    =3,30
010404  W 14004453010172                          MCW    $FXPTA+11,$S+3
010413  W 65010426                                B      IFN#004+71
010417  W 34010176010172                          BA     =3,$S+3
010426  W 14010172000010                          MCW    $S+3,X2
010435  W 15010062277703                          LCA    KK,LABEL+62083+X2
010444 R                                IFN#007   RESV   0
010444  W 14010057004455                          MCW    =3,$LODCH
010453  W 65004757                                B      ACBOIO
010457  W 01006620            I                   DSA    =10+1,20
010463  W 14010062004453                          MCW    KK,$FXPTA+11
010472  W 43                  W                   CSM
010473  W 14010172000004                          MCW    $S+3,X1
010502  W 14177703004453                          MCW    LABEL+62083+X1,$FXPTA+11
010511  W 43                  W                   CSM
010512  W 65004757                                B      ACBOIO
010516  W 01006677            I                   DSA    =10+1,77
010522  W 34010054010062                          BA     =1,KK
010531  W 33010065010062                          C      =10,KK
010540  W 6501041743                              B      IFN#004+64,43
010545  W 35004453                                BS     $FXPTA+11
010551  W 35010054004453                          BS     =1,$FXPTA+11
010560  W 15004453010135                          LCA    $FXPTA+11,M1
010567  W 35004453                                BS     $FXPTA+11
010573  W 35010143004453                          BS     =2,$FXPTA+11
010602  W 15004453010140                          LCA    $FXPTA+11,M2
010611  W 14010057004455                          MCW    =3,$LODCH
```

Figure 8-13. Pseudo-Easycoder Listing

```
       XSUB1          MACHINE CODE LISTING

BEGADD AL MACHINE CHARACTERS   R        LOC      OPCODE OPERANDS AND VARIANTS

010620  W 65004757                              B      ACBOIO
010624  W 01014420            I                 DSA    =2+1,20
010630  W 14010135004453                        MCW    M1,$FXPTA+11
010637  W 43                  W                 CSM
010640  W 14010140004453                        MCW    M2,$FXPTA+11
010647  W 43                  W                 CSM
010650  W 65004757                              B      ACBOIO
010654  W 01014477            I                 DSA    =2+1,77
010660  W 65010210                              B      $BEGIN+9
010664 R                               IFN#016  RESV   0
010664  W 14010057004455                        MCW    =3,$LODCH
010673  W 65004757                              B      ACBOIO
010677  W 01010620            I                 DSA    =10+17,20
010703  W 14710047004453                        MCW    (N),$FXPTA+11
010712  W 43                  W                 CSM
010713  W 65004757                              B      ACBOIO
010717  W 01010677            I                 DSA    =10+17,77
010723  W 15010054010062                        LCA    =1,KK
010732  W 14010062004453                        MCW    KK,$FXPTA+11
010741  W 65004756                              B      ACBFXP
010745  W 01017630            I                 DSA    =3,30
010751  W 14004453010172                        MCW    $FXPTA+11,$S+3
010760  W 65010773                              B      IFN#020
010764  W 34010176010172                        BA     =3,$S+3
010773 R                               IFN#020  RESV   0
010773  W 14010057004455                        MCW    =3,$LODCH
011002  W 65004757                              B      ACBOIO
011006  W 01006620            I                 DSA    =10+1,20
011012  W 14010062004453                        MCW    KK,$FXPTA+11
011021  W 43                  W                 CSM
011022  W 14010172000004                        MCW    $S+3,X1
011031  W 14177703004453                        MCW    LABEL+62083+X1,$FXPTA+11
011040  W 43                  W                 CSM
011041  W 65004757                              B      ACBOIO
011045  W 01006677            I                 DSA    =10+1,77
011051  W 34010054010062                        BA     =1,KK
011060  W 33010065010062                        C      =10,KK
011067  W 6501076443                            B      IFN#016+64,43
011074  W 14010057004455                        MCW    =3,$LODCH
011103  W 65004757                              B      ACBOIO
011107  W 01014420            I                 DSA    =2+1,20
011113  W 14010135004453                        MCW    M1,$FXPTA+11
011122  W 43                  W                 CSM
011123  W 14010140004453                        MCW    M2,$FXPTA+11
011132  W 43                  W                 CSM
011133  W 65004757                              B      ACBOIO
011137  W 01014477            I                 DSA    =2+1,77
011143  W 65010210                              B      $BEGIN+9
011147  W 65010210                              B      $BEGIN+9
011153  W 40                  W                 NOP
```

Figure 8-13 (cont). Pseudo-Easycoder Listing

Each instruction is edited to resemble a line of an Easycoder listing. On the left side of the page, the beginning address is given, followed by left punctuation, followed by machine code, followed by right punctuation.

For each programmer-defined EFN (statement label) that is not associated with a Format statement, a tag is generated that corresponds to the IFN (internal formula number). This tag appears on a line with the command RESV 0.

DSA, DCW, and DC statements, as well as all instructions, appear with mnemonic operation codes and with the one or two address fields edited symbolically. Variants appear in octal; address arithmetic appears in decimal.

In array references, the symbolic tag does not always appear to be correct because of subscripting. The address field of a variably subscripted array element contains the base address incremented by the constant portions of the subscripts and decremented by an allocation constant. This may cause the address value to appear within the range of another array and thus cause the appearance of a wrong tag. Users must remember that this is not a compiler error.

Error Diagnostics

During compilation, run-tape generation, and execution, error messages can be issued by several system modules: the compiler, compiler monitor, run-tape generator, object I/O routines, and Fortran library. Error messages issued by segments other than the compiler are in English; they are listed in Appendix G. All are "job fatal."

Most compiler diagnostics are printed as error numbers, although a few diagnostics are issued as English sentences. Where possible during the source-program listing, diagnostic numbers for detected errors are printed next to the appropriate statements. These error printouts usually consist of up to three digits. If the error is fatal, the number is printed in columns 109 to 111 of the listing on the same line as the statement. If the error is not fatal, the number is printed in columns 116 to 118. If the error number has less than three digits, it is right justified in the appropriate columns. Errors detected after the source program has been listed are printed beneath the source program with an IFN (internal formula number) to the left of the error number so that the statement to which it applies can be located in the listing.

Compiler error numbers and their meanings are given in Appendix G. The magnitude of the compiler error number indicates the system module that detected the error. Subheadings in Appendix G show which module is responsible for detecting a given error.

In general, the types of errors listed below are not detected by the compiler and could lead to unspecified results if the program gets into execution:

FORMAT errors,

Punctuation errors,

Illegal characters,

Subscripting errors,

Mixed mode in arithmetic statements,

Errors in I/O statements,

FORMAT I/O list mismatches.

However, some of the errors above can be detected by diagnostic preprocessing, described later.

Memory Dumps

Three Honeywell-supplied subroutines provide for dynamic memory dumps of all or part of memory, as described in Section VI. A terminal dump of memory is taken if the *DUMP card is part of the job deck.

DIAGNOSTIC PREPROCESSING

The diagnostic preprocessor checks source programs for errors without using compilation time, and it can provide diagnostic information about the construction of the source program that the compiler is unable to supply.

Diagnostic preprocessing is a job-level option; therefore, jobs submitted for diagnostic preprocessing can be interspersed with jobs submitted for load-and-go operation or for writing on a go-later tape. The presence of a *DIAG or *DIAG, T card preceding a *JOBID card causes transfer of control to the diagnostic preprocessor. A *DIAG card causes the preprocessor to list all control cards and all programs of a job with appropriate diagnostic information. A *DIAG, T card causes the diagnostic preprocessor to write a job onto logical tape 5 in card-image form. The compiler is then called in, and compilation of the job on tape 5 is automatically carried out, unless the job is fatal.

Preprocess-Only Option - *DIAG

When only preprocessing is requested, the diagnostic preprocessor lists on the common output device the source program and any diagnostics. A sample input deck and a flow diagram of a diagnostic preprocessor job of this type are shown in Figure 8-14 and 8-15. Note that the work tapes shown in Figure 8-15 are not required for preprocessing. However, since compilation would normally precede or follow diagnostic preprocessing, the tapes are shown mounted.

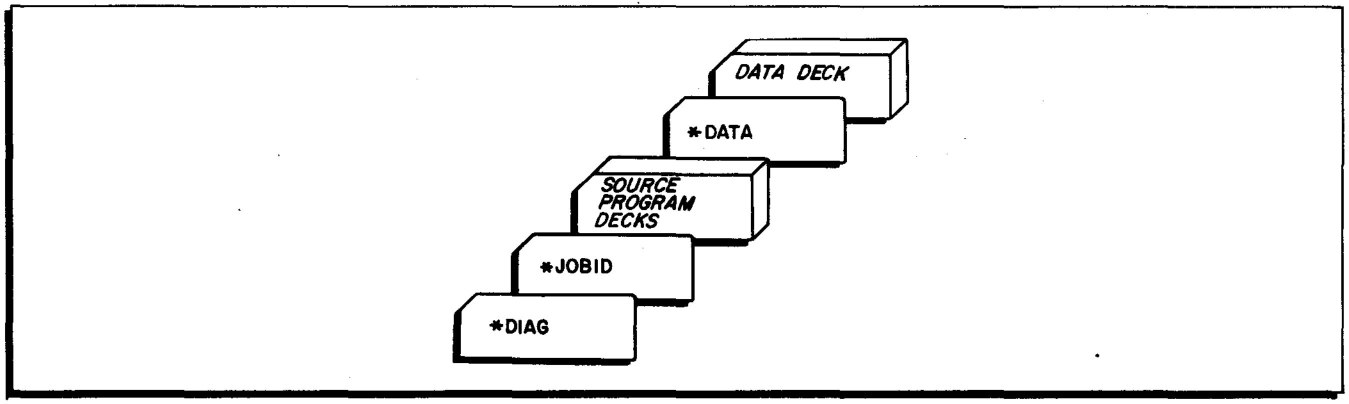


Figure 8-14. Input Deck for Diagnostic Preprocessing

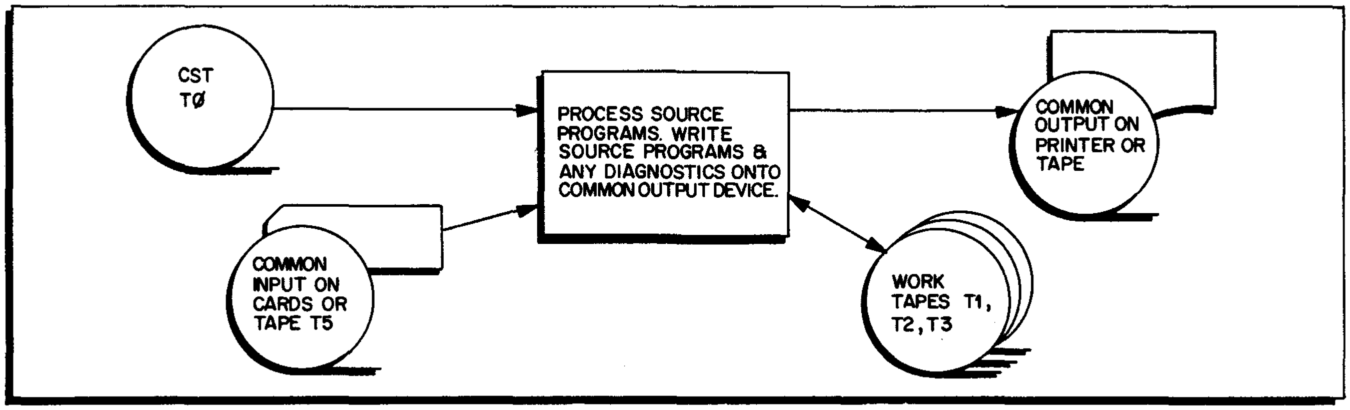

CST TØ
PROCESS SOURCE PROGRAMS. WRITE SOURCE PROGRAMS & ANY DIAGNOSTICS ONTO COMMON OUTPUT DEVICE.
COMMON OUTPUT ON PRINTER OR TAPE
COMMON INPUT ON CARDS OR TAPE T5
WORK TAPES T1, T2,T3

Figure 8-15. Flow Diagram for Diagnostic Preprocessing — Preprocess-Only Option

When a *DIAG card is encountered, all programs, subprograms, control cards, and binary decks following are listed until a *DATA card or other card indicating the end of that job is encountered. Components other than source programs are listed without diagnostic action. Data are not listed.

On the printer, each source program is listed on a separate page, together with diagnostic messages if errors were detected. At the end of each program, a list of subprogram references is printed. Following the program is the line: *END OF PROGRAM.

Preprocessing to Tape - *DIAG, T

A card-to-tape option is specified on the *DIAG control card by a comma in column 6 and the character T somewhere in columns 7-72. When this option is specified, all program decks of the job are preprocessed and written onto logical tape 5. When the preprocessor senses the end of job (*DATA card or other end-of-job card), logical tape 5 is rewound and a check made for job fatality. Any source-program error detected by the preprocessor will cause job fatality.

If the job is fatal, the complete job listing is copied from logical tape 5 onto the common output device with the preprocessor source-program diagnostics. Control is then passed to the compiler monitor to process the next job in the input deck.

However, if no job-fatal error was detected when the tape is rewound, control is passed to the compiler monitor with parameters informing the monitor to use tape 5 as input to load-and-go processing or writing a go-later tape according to the run option. Thus, by using the tape option, a job can be preprocessed, compiled, relocated, and executed in one operation. Figure 8-16 shows a flow diagram of diagnostic preprocessing to tape. Note that work tapes are required for this job, since compilation automatically follows preprocessing.

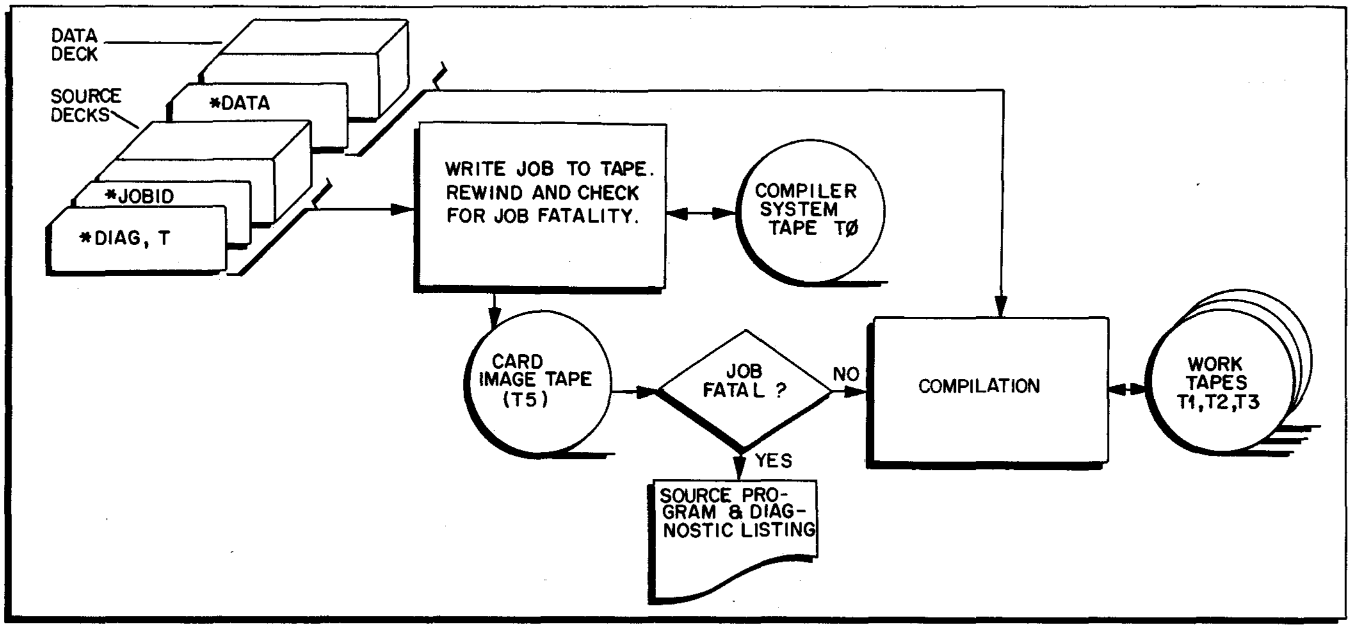


Figure 8-16. Diagnostic Preprocessor Flow Diagram - Tape Option

Diagnostics

The preprocessor writes diagnostic messages in English. On the printer, the diagnostic begins in column 40 of the line directly beneath the statement in which the error occurs. In diagnosing an error in an executable statement, the preprocessor often includes as part of the diagnostic a portion of the source statement. This portion begins in column 70 of the line directly beneath the statement in error and indicates the point in the source statement at which the error occurred. For most FORMAT statement diagnostics, a column number is given instead of a portion of the source statement. The column number indicates the position in the FORMAT specification at which the error occurred. In indicating the column number, the diagnostic preprocessor does not count either the opening left parenthesis of the specification or embedded blanks. A flag always follows the diagnostic message, indicating that the error is fatal.

In most instances, the preprocessor continues to analyze a statement already found to be erroneous, sometimes producing a series of error diagnostics for a single statement. Since the initial error may cause the introduction of spurious diagnostics in continued analysis, the programmer should take considerable care in correcting statements containing multiple errors; and in many cases, he may wish to ignore all diagnostics after the first one issued.

Limitations in the use of the diagnostic preprocessor are as follows:

1. Statement operators must not cross card boundaries.
2. EQUIVALENCE statements are bypassed by the preprocessor.
3. DATA initialization statements are bypassed by the preprocessor. (These statements are diagnosed by the compiler.)
4. The preprocessor does not detect missing or duplicate chains or subprograms.

Appendix G lists diagnostics issued by the preprocessor. Figure 8-17 shows the output from the preprocessor in the form of a printed listing.

```
                                    PREPROCESSOR  DIAGNOSTIC  LISTING

*JOBID
 TITLEDIAGBG
C
C
C
C     CHECK DIAGNOSTICS PROBLEMS
C
C
C     CHECK RIGHT PARENS AS LAST CHARACTER ON CARD,NEXT IS CONTINUING
C
      WRITE(3,12)
  12  FORMAT (    4HTEST,F13.9,     F13.9,        F13.9,       )

                           THERE IS DATA AFTER THE TERMINATING RIGHT PARENTHESIS.
                                                                  ■■■■■■■■■■

     1I=8
C
C     CHECK DIAGNOSTIC ON IH FORMAT, SHOULD HAVE BEEN I4
C
      WRITE(3,13)
  13  FORMAT(2IH)

                           DUPLICATE FIELD SPECIFICATION OR MISSING COMMA.
                                          COLUMN NUMBER016          ■■■■■■■■■■

C
C     CHECK DIAGNOSTIC ON MORE THAN 3 SETS OF PARENS
C
      WRITE(3,14)
  14  FORMAT(3(2(2(4HSCAN,1X),4HSCAN),4HSCAN))

                           MORE THAN THREE NESTED PARENTHESES.
                                          COLUMN NUMBER019          ■■■■■■■■■■

C
C     CHECK DIAGNOSTIC FOR MISSING SPECIFICATION
C
      WRITE(3,16)
  16  FORMAT(15,5.3)

                           EITHER A MEANINGLESS DECIMAL POINT OR MISSING FIELD SPECIFICATION.
                                          COLUMN NUMBER018          ■■■■■■■■■■
      WRITE(3,17)
  17  FORMAT(2HAF12.5)

                           EITHER A MEANINGLESS DECIMAL POINT OR MISSING FIELD SPECIFICATION.
                                          COLUMN NUMBER020          ■■■■■■■■■■
      WRITE(3,18)
  18  FORMAT(6I)

                           AN A, H, I, L, O, OR X FIELD IS BLANK OR ZERO.
                                          COLUMN NUMBER016          ■■■■■■■■■■
      STOP
      END
* END OF PROGRAM
```

Figure 8-17. Diagnostic Preprocessor Listing

SCREEN CONVERSION

The Screen routine provides limited conversion of user source programs written in Fortran II into source programs in Fortran D language. Screen replaces Fortran II I/O statements and names of library functions with their equivalents in Fortran D. Screening is a run-level option.

Presence of a *SCREEN control card will cause all source programs to be translated until an *ENDRUN card is encountered. Data decks should not be included in a Screen run.

As output, Screen punches a new source deck and produces a listing of the new source deck. Output can be on line or off line as desired. If a character is punched anywhere in columns 8 to 72 of the *SCREEN card, each card of each job is numbered sequentially, both in the new card deck and in the listing. The sequential numbers appear in columns 75 to 77 and are three-digit decimal numbers beginning with 001. When the routine encounters a *JOBID card in the input deck, the numbering sequence is reinitialized. (*JOBID cards are not required for a Screen run, but their use is recommended to separate job decks.)

Screen processing can be performed as a separate run, or it can follow load-and-go processing. An input deck for Screen is shown in Figure 8-18 and the flow diagram is given in Figure 8-19.

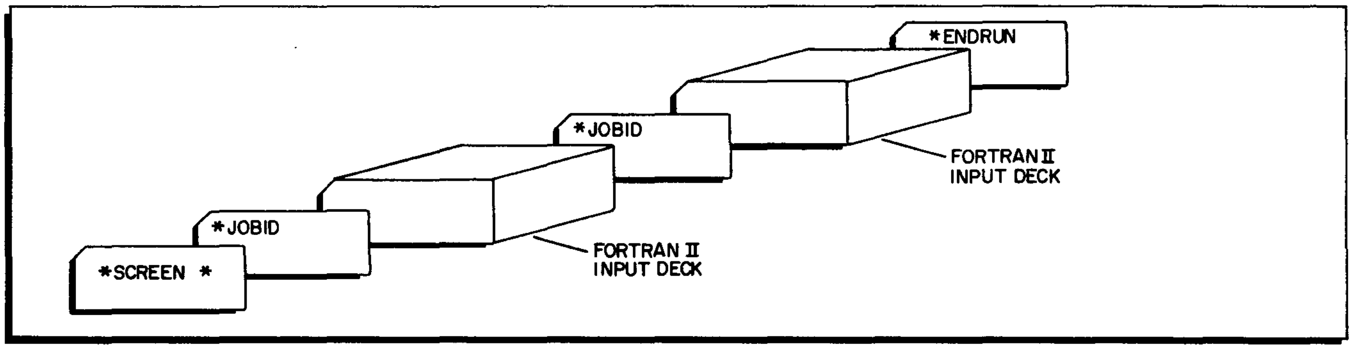


Figure 8-18. Input Deck for Screen

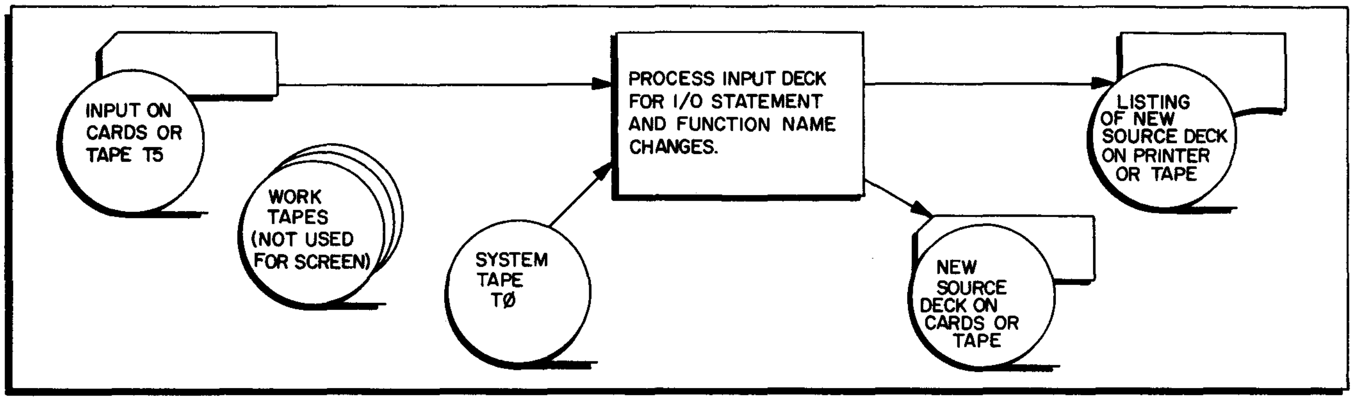

INPUT ON CARDS OR TAPE T5
WORK TAPES (NOT USED FOR SCREEN)
SYSTEM TAPE TØ
PROCESS INPUT DECK FOR I/O STATEMENT AND FUNCTION NAME CHANGES.
LISTING OF NEW SOURCE DECK ON PRINTER OR TAPE
NEW SOURCE DECK ON CARDS OR TAPE

Figure 8-19. System Flow of Screen

Figure 8-20 shows the changes made in input/output statements by Screen. In each case, i is the number identifying the I/O device, n is either a FORMAT statement label or the name of an array, and List represents a correctly sequenced list of names of variables, array elements, and arrays.

Fortran II	Fortran D
READ INPUT TAPE i, n, List	READ (i, n) List
READ TAPE i, List	READ (i) List
READ n, List	READ (i, n) List
WRITE OUTPUT TAPE i, n, List	WRITE (i, n) List
WRITE TAPE i, List	WRITE (i) List
PRINT n, List	WRITE (i, n) List
PUNCH n, List	WRITE (i, n) List

Figure 8-20. Screen Conversion of I/O Statements

Fortran II function names appearing anywhere in a source-program statement are converted. Figure 8-21 shows the conversion of library function names.

Fortran II	Fortran D
ABSF	ABS
XABSF	IABS
INTF	AINT
XINTF	INT
MODF	AMOD
XMODF	MOD
SIGNF	SIGN
XSIGNF	ISIGN
MAX0F	AMAX0
XMAX0F	MAX0
MAX1F	AMAX1
XMAX1F	MAX1
MIN0F	AMIN0
XMIN0F	MIN0
MIN1F	AMIN1
XMIN1F	MIN1
FLOATF	FLOAT
FIXF XFIXF	IFIX
DIMF	DIM
XDIMF	IDIM
LOGF	ALOG
SINF	SIN
COSF	COS
EXPF	EXP
SQRTF	SQRT
ATANF	ATAN
TANHF	TANH

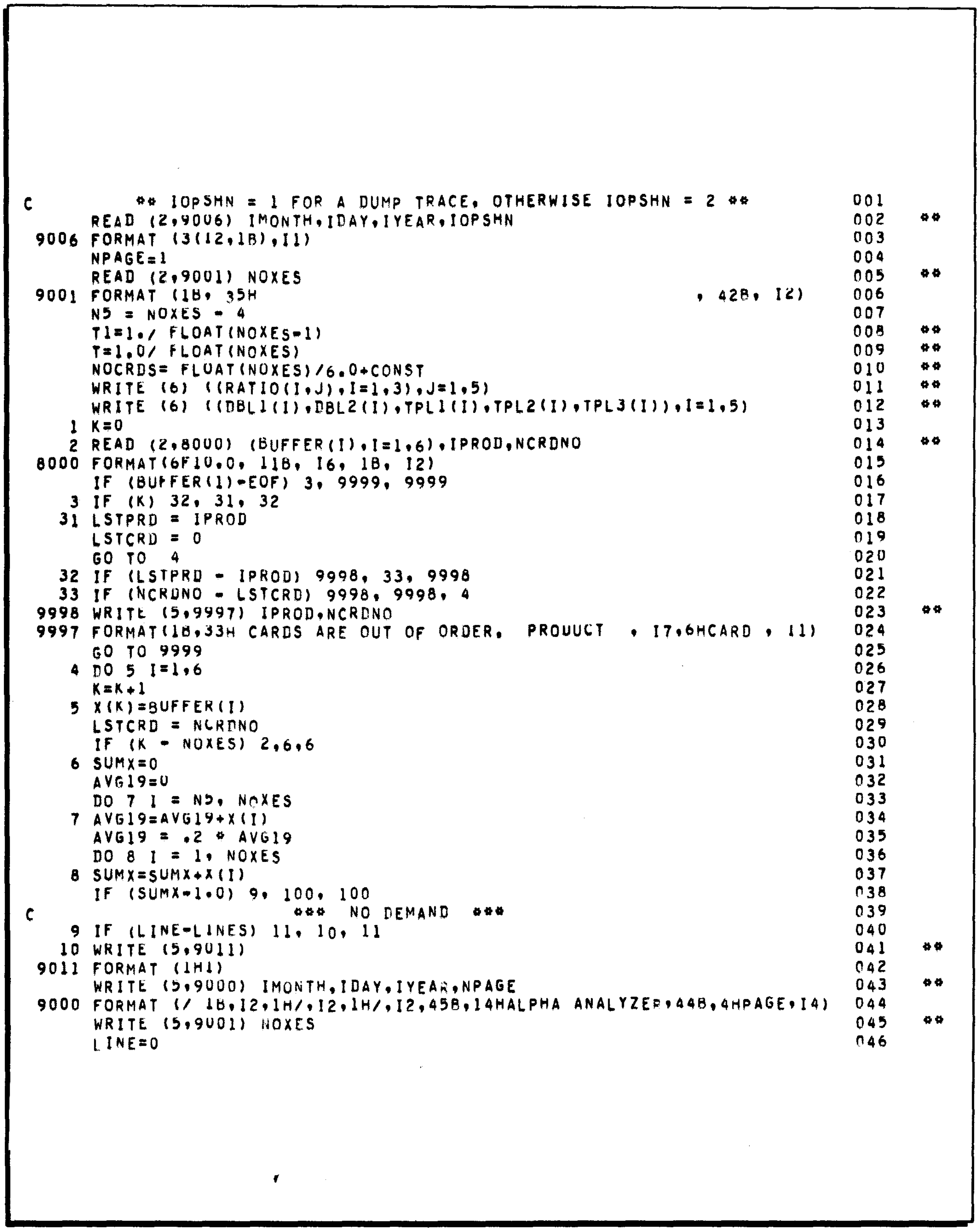
Figure 8-21. Screen Conversion of Library Function Names

The format of statements processed by Screen is as follows:

1. Statements that are not I/O statements and do not have function names are reproduced without change. Statements already in converted format are also reproduced without change.
2. A converted I/O statement starts in column 7. All blanks are suppressed except those on each side of the parentheses enclosing the logical device address and/or the FORMAT statement number.
3. Converted statements containing function names start in column 7. All blanks are suppressed except those resulting from shortening of function names in conversion.
4. When a converted statement is longer than the original, a continuation card is generated if required. Continuation cards are numbered in column 6.
5. If a Fortran II statement required a continuation card but the converted statement is 66 characters or less, the new statement appears on a single card.
6. Occasionally, a Fortran II statement may have continuation cards with comment cards interspersed. When converted, the statement, with as many continuation cards as needed, is generated first. Comment cards are reproduced without change and follow the statement cards.
7. Two asterisks in columns 81 and 82 of the listing indicate that a change has been made in the statement.
8. Columns 75-77 of the card deck and listing contain a three-digit number when the sequential numbering option is used. Otherwise, columns 73-80 are reproduced without change.
9. Screen will handle Fortran II statements with up to 19 continuation cards if the number of nonblank characters does not exceed 800. In no case can there be more than 24 cards between the first cards of two consecutive statements. This includes continuation, comment, and control cards. No data cards are permitted in a Screen run.
10. Figure 8-22 shows a page of input to Screen. Figure 8-23 shows the Screen output for the same program. Note that the sequential numbering option has been used.
11. There are two possible error printouts from Screen. These are listed in Appendix G.

```
C         ** IOPSHN = 1 FOR A DUMP TRACE, OTHERWISE IOpSHN = 2 **
      READ 9006, IMONTH,IDAY,IYEAR,IOPSHN
 9006 FORMAT (3(I2,1B),I1)
      NPAGE=1
      READ 9001, NOXES
 9001 FORMAT (1B, 35H                                       , 42B, I2)
      N5 = NOXES - 4
      T1 = 1./FLOATF(NOXES - 1)
      T = 1.0 / FLOATF(NOXES)
      NOCRDS=FLOATF(NOXES)/6.0+CONST
      WRITE TAPE 6, ((RATIO(I,J), I = 1,3), J = 1,5)
      WRITE TAPE 6, ((DBL1(I), DBL2(I), TPL1(I), TPL2(I), TPL3(I)),I=1,5
     1)
    1 K=0
    2 READ 8000,(BUFFER(I),I=1,6),IPROD,NCRDNO
 8000 FORMAT(6F10.0, 11B, I6, 1B, I2)
      IF (BUFFER(1)-EOF) 3, 9999, 9999
    3 IF (K) 32, 31, 32
   31 LSTPRD = IPROD
      LSTCRD = 0
      GO TO  4
   32 IF (LSTPRD - IPROD) 9998, 33, 9998
   33 IF (NCRDNO - LSTCRD) 9998, 9998, 4
 9998 WRITE OUTPUT TAPE 5, 9997, IPROD, NCRDNO
 9997 FORMAT(1B,33H CARDS ARE OUT OF ORDER,  PRODUCT   , I7,6HCARD , I1)
      GO TO 9999
    4 DO 5 I=1,6
      K=K+1
    5 X(K)=BUFFER(I)
      LSTCRD = NCRDNO
      IF (K - NOXES) 2,6,6
    6 SUMX=0
      AVG19=0
      DO 7 I = N5, NOXES
    7 AVG19=AVG19+X(I)
      AVG19 = .2 * AVG19
      DO 8 I = 1, NOXES
    8 SUMX=SUMX+X(I)
      IF (SUMX-1.0) 9, 100, 100
C                           ***  NO DEMAND  ***
    9 IF (LINE-LINES) 11, 10, 11
   10 WRITE OUTPUT TAPE 5,9011
 9011 FORMAT (1H1)
      WRITE OUTPUT TAPE 5,9000, IMONTH,IDAY,IYEAR,NPAGE
 9000 FORMAT (/ 1B,I2,1H/,I2,1H/,I2,45B,14HALPHA ANALYZER,44B,4HPAGE,I4)
      WRITE OUTPUT TAPE 5, 9001, NOXES
      LINE=0
```

Figure 8-22. Listing of Card Input to Screen



```
C          ** IOPSHN = 1 FOR A DUMP TRACE, OTHERWISE IOPSHN = 2 **
      READ (2,9006) IMONTH,IDAY,IYEAR,IOPSHN                                    **
 9006 FORMAT (3(I2,1B),I1)
      NPAGE=1
      READ (2,9001) NOXES                                                       **
 9001 FORMAT (1B, 35H                                   , 42B, I2)
      N5 = NOXES - 4
      T1=1./ FLOAT(NOXES-1)                                                     **
      T=1.0/ FLOAT(NOXES)                                                       **
      NOCRDS= FLOAT(NOXES)/6.0+CONST                                            **
      WRITE (6) ((RATIO(I,J),I=1,3),J=1,5)                                      **
      WRITE (6) ((DBL1(I),DBL2(I),TPL1(I),TPL2(I),TPL3(I)),I=1,5)               **
    1 K=0
    2 READ (2,8000) (BUFFER(I),I=1,6),IPROD,NCRDNO                              **
 8000 FORMAT(6F10.0, 11B, I6, 1B, I2)
      IF (BUFFER(1)-EOF) 3, 9999, 9999
    3 IF (K) 32, 31, 32
   31 LSTPRD = IPROD
      LSTCRD = 0
      GO TO  4
   32 IF (LSTPRD - IPROD) 9998, 33, 9998
   33 IF (NCRDNO - LSTCRD) 9998, 9998, 4
 9998 WRITE (5,9997) IPROD,NCRDNO                                               **
 9997 FORMAT(1B,33H CARDS ARE OUT OF ORDER,  PROUUCT  , I7,6HCARD , I1)
      GO TO 9999
    4 DO 5 I=1,6
      K=K+1
    5 X(K)=BUFFER(I)
      LSTCRD = NCRDNO
      IF (K - NOXES) 2,6,6
    6 SUMX=0
      AVG19=0
      DO 7 I = N5, NOXES
    7 AVG19=AVG19+X(I)
      AVG19 = .2 * AVG19
      DO 8 I = 1, NOXES
    8 SUMX=SUMX+X(I)
      IF (SUMX-1.0) 9, 100, 100
C                           ***  NO DEMAND  ***
    9 IF (LINE-LINES) 11, 10, 11
   10 WRITE (5,9011)                                                            **
 9011 FORMAT (1H1)
      WRITE (5,9000) IMONTH,IDAY,IYEAR,NPAGE                                    **
 9000 FORMAT (/ 1B,I2,1H/,I2,1H/,I2,45B,14HALPHA ANALYZER,44B,4HPAGE,I4)
      WRITE (5,9001) NOXES                                                      **
      LINE=0
```

Figure 8-23. Output Listing from Screen

SECTION IX
OPERATING PROCEDURES

FORTRAN RUN OPTIONS

There are four possible run modes for Fortran D. These are:

Load-and-Go

Screen

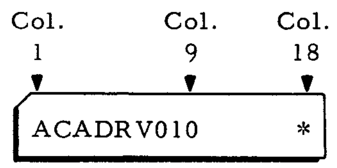
Writing a Go-Later Tape

Executing a Go-Later Tape

Operating procedures for all run modes are very similar. Where setup and operating procedures vary, the differences are explained for each run mode. Otherwise, standard operating procedures should be followed.

STANDARD CONSOLE CALL

For every run except execution of a go-later tape, a standard console call is used. The run is initiated either by a Console Call card at the beginning of the card input or by a keyin at the console. The console call can contain a number of options as shown in Table 9-1. However, when no options are indicated, the console call in card form appears as follows:



When the console call is keyed in, the operator follows the standard starting procedure, given in the following paragraph, up to the second loader halt (B-address register = 17002). He then keys in the console call. A minimum console call corresponding to the Console Call card above is given below.

Octal Keyin	Octal Location	Equivalent Card Column
WM01	100	9
WM21	104	1
23	105	2
21	106	3
24	107	4
51	110	5
65	111	6
WM00	112	7
01	113	8
RM54	125	18

The indicated record and word marks are required.

The console call for executing a go-later tape is discussed in the paragraph entitled "Executing a Go-Later Tape."

EQUIPMENT REQUIREMENTS

A minimum memory of 16,384 characters is required for Fortran D. Minimum peripheral requirements are six devices: four tape units, a standard input device (card reader or tape), and a standard output device (printer or tape). Equipment setup and optional devices are shown later for each run mode.

TAPE LOADER-MONITORS

Fortran D can be loaded by any of the Series 200 tape loader-monitors, either fixed or floating. However, the Fortran D compiler system tape contains two loader-monitors. In the absence of options, the Fortran D system brings in Tape Loader-Monitor C in the 3-character addressing mode. At installations running in the 4-character mode, an option in the console call and a console keyin to location 124_8 will bring in Tape Loader-Monitor C in the 4-character addressing mode.

STARTING PROCEDURE

Since Fortran D uses a Series 200 tape loader-monitor, starting procedures for the system follow the standard starting procedures for the loader-monitor used. Given below is the starting procedure for Tape Loader-Monitor C with a minimum system configuration of 4 tape units, card reader, and printer. The run is presumed to be either load-and-go or one in which a go-later tape is written. Variations on the standard starting procedure for Screen runs, go-later execution, and other options are contained in the six notes that follow the starting procedure.

1. Press the STOP button on the console.
2. If not already mounted, mount the compiler system tape on tape drive 0 in protect status.[1]
3. Mount work tapes on tape drives 1, 2, and 3 in permit status.[2]
4. Press INITIALIZE.[3]
5. Place the card deck in card reader hopper, making sure that the first card is the Console Call card. Cycle up the card reader and printer.[4]
6. If the compiler system was on tape drive 0, make sure it is rewound.
7. Set the CONTENTS buttons to octal 40 (100000).
8. Press BOOTSTRAP.[5]
9. Set the CONTENTS buttons to octal 40.
10. Press BOOTSTRAP.
11. Press RUN.
12. Display the contents of the B-address register. These contents should be octal 17001.

13. Press RUN.

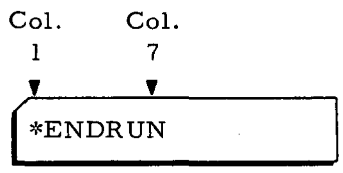
14. Display the contents of the B-address register. These contents should be octal 17002.[6]

15. Press RUN.

NOTES:
1. When executing a go-later tape (saved BRT), the go-later tape is mounted on tape drive 0 in protect status and no compiler system tape is used.
2. Work tapes are not required for Screen.
3. See the paragraph on writing a go-later tape. There is an option in which the operator sets SENSE switch 1 ON after pressing INITIALIZE.
4. A punch or a punch tape is required for Screen.
5. When running in the 4-character address mode, key in octal 20 to location 124_8 after step 12 above.
6. The console call is keyed in immediately after the halt at 17002 when not using a Console Call card.

TERMINATING A RUN

Runs terminate automatically when an *ENDRUN card or card image is encountered. It is the operator's responsibility to place the *ENDRUN card at the end of the deck. When performing off-line conversion from cards to tape, the operator must place the *ENDRUN card at the end of the deck, so that the card image will appear at the end of the input tape. The *ENDRUN card is shown below:



There is one exception in which a run need not terminate with an *ENDRUN card. If a load-and-go run is immediately followed by a Screen run, presence of the *SCREEN card in the input deck terminates the load-and-go run and brings the Screen routine in automatically. However, the Screen run must itself be terminated by an *ENDRUN card.

CONSOLE CALL OPTIONS

The console call can contain options that indicate additional equipment or specify the way in which equipment is to be used during the run. These options are indicated in columns 10 to 17 of the Console Call card or by the equivalent console keyin to octal locations 115 to 124. The options specify the following.

- Additional core memory above 16,384 characters
- Use of a stack tape on logical tape drive No. 4

- Punch Option
- Multiply/divide hardware
- Options used in writing a go-later tape
- Use of tape as the standard output device
- Use of four-character address mode

When there are no options, columns 10 through 17 of the Console Call card or their equivalent keyin can be used for a date.

Table 9-1. Console Call Options

CARD		CONSOLE		
Card Column	Contents	Octal Location	Octal Keyin	How Used
10	*	115	WM54	Flag to indicate that options follow in columns 11-17. Required whenever there is any option.
11		116		A designator for the amount of memory over 16,384 characters used for the run. A blank or any character not specified in the list will cause 16K memory to be used. When more than 16K is used, the proper option is required.
(Memory Size Options)	E-T		25-63	E (25) = 20,480 F (26) = 24,576 G (27) = 28,672 H (30) = 32,768 I (31) = 40,960 J (41) = 49,152 K (42) = 57,344 L (43) = 65,536 M (44) = 81,920 N (45) = 98,304 O (46) = 114,688 P (47) = 132,072 Q (50) = 163,840 R (51) = 196,608 S (62) = 229,376 T (63) = 262,144
12 (Stack Tape Options)		117		This location is used by installations having a stack tape (logical tape address T4). A blank or any character except A or G in this location prevents any tape mounted on T4 from being allocated as a work tape.
	A		21	Allocate the tape on T4 as a work tape during execution. (T4 will be allocated after T2 and T3.)
	G		27	Initialize the tape on T4 as a stack tape. Do not allocate it at object time. This option is used when there are no programs already stacked on T4.
13[1] (Punch Options)		120		This location is used by installations having a common punch device. A blank or any character not specified indicates the absence of a punch device.

Table 9-1 (cont). Console Call Options

CARD		CONSOLE		
Card Column	Contents	Octal Location	Octal Keyin	How Used
(Punch Options) (cont)	P		47	Common punch used.
	0-7		00-07	Logical tape address (T0 to T7) of common punch tape.
14 (Multiply/ Divide Hardware)	D	121	24	A D must be stored in this location by those installations having multiply/divide hardware. A blank or any character except D in this location indicates multiply/divide software.
15		122		This location is used only for a run mode in which a go-later tape is written. A blank or any character other than B, E, or L or their equivalent keyins in this location indicates that this is not a run in which a go-later tape is written.
(Write Go-later Options)	B		22	Logical tape drive T1 has a BRT (go-later) tape on it. The tape is positioned to the 1ERI record and jobs are added to the already existing tape.
	L		43	Logical tape drive T1 has a work tape on it. A go-later tape is to be generated by copying the loader onto the work tape and then writing go-later jobs on the tape.
	E		25	Required for emergency, restart of a go-later run. The tape on T1 is repositioned in a backward direction to the end of the last good job before the run continues. The tape on T1 <u>must</u> <u>not</u> be rewound.
16[1] (Tape Used as Output Device)	0-7	123	00-07	This location is used only when common output is on tape. It contains the logical address of the common output tape. A blank or any character other than zero to seven in this location indicates a printer as common output. (Note that the same tape can be substituted for printing and punching, i.e., card column 13 can be the same as card column 16 if desired.)
17		124		This location is used only at installations having the 4-character address mode. A blank or any character except 4 in the location indicates the three-character address mode.
(4-Character address mode)	4		04	Four-character address mode.

[1]If these options are used when generating a go-later tape, they must be indicated on the Console Call cards used when running the go-later tape.

CODED HALTS DURING FORTRAN RUNS

In addition to tape loader-monitor halts, there are five other halts that the user can program. They are described in Table 9-2.

Table 9-2. Possible Halts During a Fortran Run

Halt	General Display Pattern on the Console		Meaning
	A Address in Octal	B Address in Octal	
Tape Loader-Monitor C Halts	- - - - -	- - - - -	See Order No. 221 for Tape Loader-Monitor C, No. 005 for Floating Tape Loader-Monitor C.
Fortran Driver Halts	- - - - -	0 p p 1 d 0 p p 2 d	pp = peripheral control unit number d = device number 1 = uncorrectable read 2 = uncorrectable write Check tape for dirt and damage. (See Equipment Operators' Manual (Model 200), Order Number 040.) Depress RUN to try to reread or rewrite.
Fortran Compiler Halt	- - - - -	0 6 0 0 t	Physical tape t desired. Change tape and depress RUN.
Fortran PAUSE	- - - - -	0 4 0 0 0	Perform operations indicated on run request and depress RUN.
Fortran PAUSE $n_1n_2n_3n_4n_5n_6$ STOP $n_1n_2n_3n_4n_5n_6$	$n_1n_2n_3n_4n_5$	0 5 0 0 n_6	Examine the A and B addresses for the STOP or PAUSE number. ($n_1n_2n_3n_4n_5n_6$). Perform the operations indicated for this number on the run request. Depress RUN.

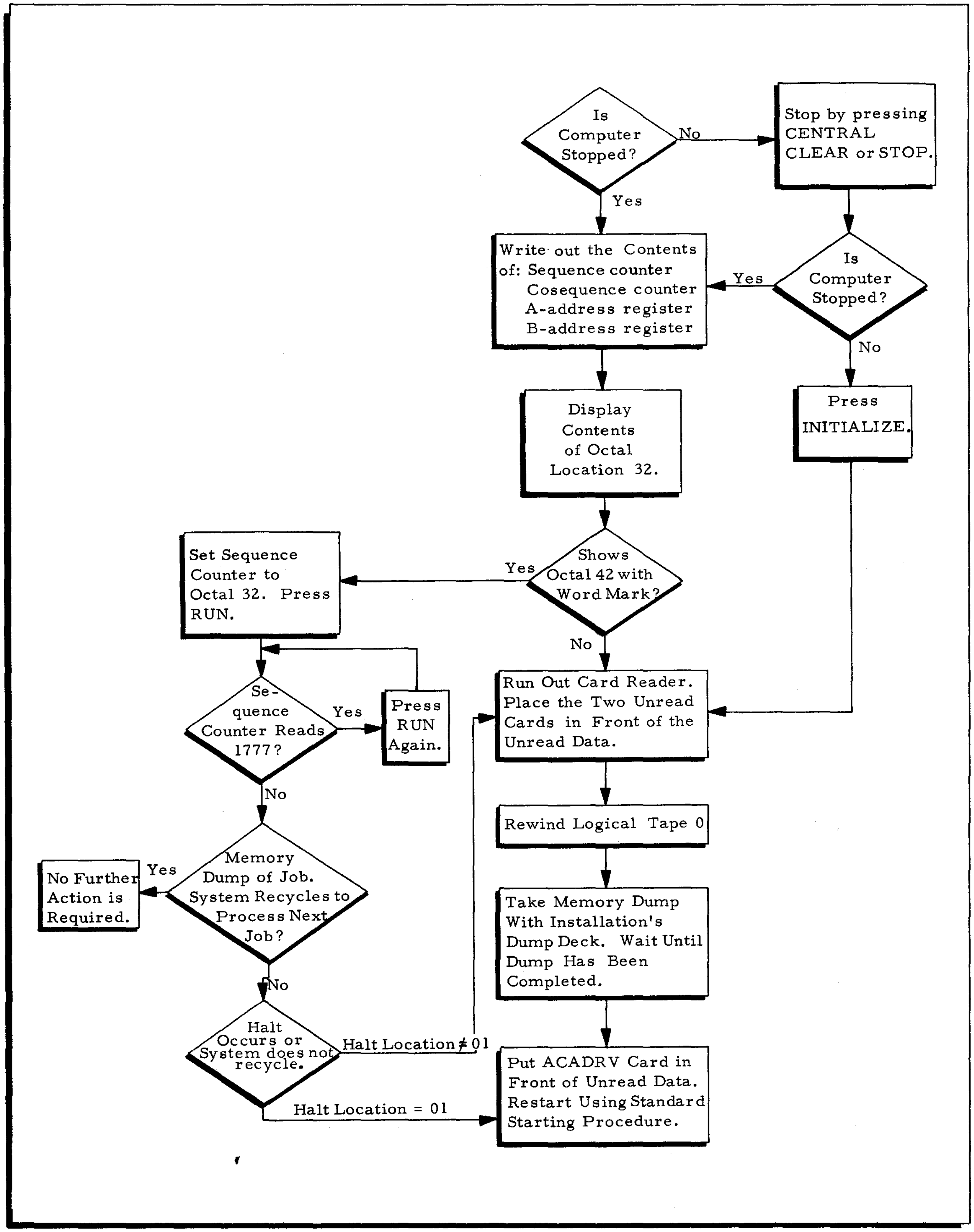
Unprogrammed Halts and Looping

The following action should be taken if the run should loop or come to some unspecified halt.

1. Stop the machine (if not already in the stop mode) by depressing the CENTRAL CLEAR or STOP button. Write down the contents of the sequence counter, cosequence counter, A- and B-address registers. Follow the installation's hang-up procedures for keeping this information with the deck in error.

   If the machine cannot be halted by the method explained above, depress INITIALIZE button and proceed to step 4. Otherwise, proceed as follows.

2. Display the contents of location octal 32. If it contains an octal 42 with a word mark, proceed as follows. Otherwise, proceed to step 4.



Is Computer Stopped?
No
Stop by pressing CENTRAL CLEAR or STOP.
Yes
Write out the Contents of: Sequence counter Cosequence counter A-address register B-address register
Yes
Is Computer Stopped?
No
Display Contents of Octal Location 32.
Press INITIALIZE.
Set Sequence Counter to Octal 32. Press RUN.
Yes
Shows Octal 42 with Word Mark?
No
Sequence Counter Reads 1777?
Yes
Press RUN Again.
Run Out Card Reader. Place the Two Unread Cards in Front of the Unread Data.
No
Rewind Logical Tape 0
No Further Action is Required.
Yes
Memory Dump of Job. System Recycles to Process Next Job?
Take Memory Dump With Installation's Dump Deck. Wait Until Dump Has Been Completed.
No
Halt Occurs or System does not recycle.
Halt Location ≠ 01
Put ACADRV Card in Front of Unread Data. Restart Using Standard Starting Procedure.
Halt Location = 01

Figure 9-1. Operator Action in Unprogrammed Halt or Looping

3. Set the sequence counter to octal 32 and depress the RUN button. This will cause a memory dump to be taken, and the system will recycle automatically to process the next job.

   NOTE: If a no-locate halt (sequence counter = 1777) occurs, depress the RUN button once more.

   If for some reason the system does not recycle or the dump loops or halts at some location other than 01, proceed to step 4. If the dump halts at location 01, proceed to step 7.

4. Run out the card reader and place the two unread cards in front of the unread data.

5. Rewind logical tape 0.

6. Take a memory dump with the dump deck provided at the installation.

7. After the dump has terminated, place the Console Call card in front of the unread data. Proceed with the standard starting and running procedure given on page 9-2.

If the run was a Screen run, a *SCREEN card must immediately follow the Console Call card in front of the unread data. Then proceed with the standard starting and running procedure.

Figure 9-1 shows the operator action in case of an unprogrammed halt or a loop in flowchart format.

LOAD-AND-GO RUN

The standard mode of Fortran D operation is load-and-go, in which a job is compiled and executed, then the next job is compiled and executed, etc., until the end of the run (*ENDRUN or *SCREEN) is encountered.

Load-and-Go Equipment

Table 9-3 show the equipment that is required and optional for load-and-go operation. Figure 9-2 shows the minimum equipment configuration for such operation.

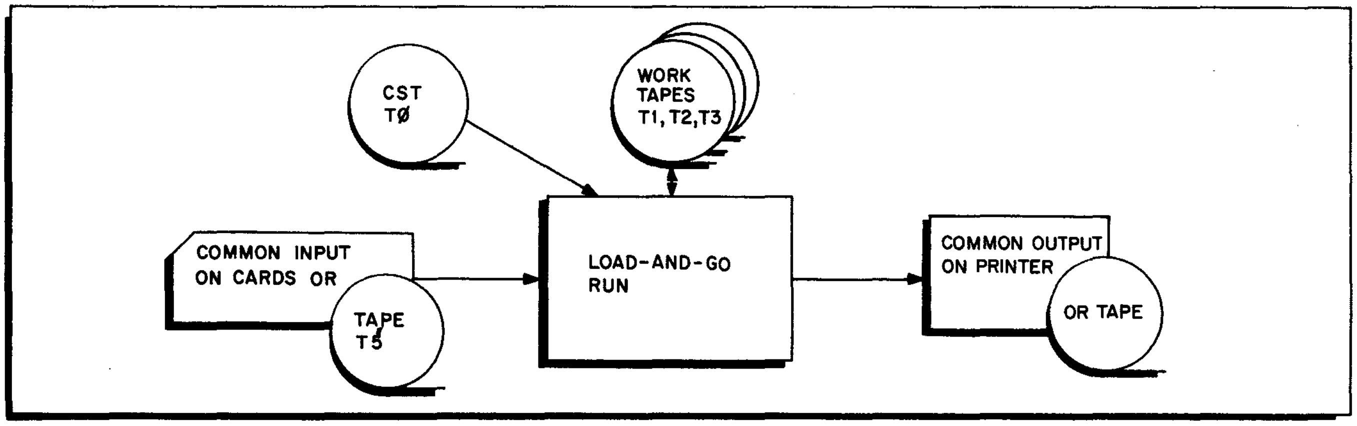


Figure 9-2. Minimum Equipment Configuration for Load-and-Go Operation

Table 9-3. Equipment for Load-and-Go Operation

Peripheral Device	Logical Tape Drive Address	Tape Status	Other Peripheral Equipment	Required or Optional
Compiler System Tape	0	protect		Required
Work Tape	1	permit		Required
Work Tape	2	permit		Required
Work Tape	3	permit		Required
Common Input Device				One Device Required
On-Line	—	—	Card Reader	
Off-line	5	protect	—	
Common Output Device				One Device Required
On-line	—	—	Printer	
Off-line	Any tape address not otherwise assigned	permit	—	
Stack Tape	4	permit		Optional
Work Tapes	6, 7, then 0 to 7 on channel 3	permit		Optional
Common Punch Device				One Device Optional (Punch tape and Print tape may be the same tape.)
On-line	—	—	Punch	
Off-line	Any tape address not otherwise assigned	permit	—	

Stack Tape (T4)

Where more than the minimum four tape drives are available at an installation, logical tape drive 4 can be used for a tape containing a library of user programs. By option in the console call, a tape on logical tape drive 4 can be used as a work tape and allocated during execution. A second option permits a stack tape to be initially generated from a work tape mounted on drive 4. However, in the absence of options, the tape is treated as an already existing stack tape and positioned to the end of the last good job at the beginning of a run.

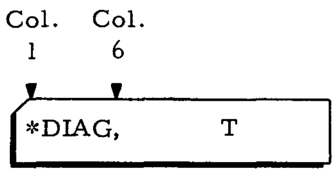
Allocation of Work Tapes

Allocation of work tapes at execution time, in the absence of any options, proceeds as follows: logical tapes 2 and 3; when available, logical tapes 5 and 6; when available, logical tapes 0 to 7 of channel 3. When off-line input or output is used, tapes on these drives are not allocated. A tape on logical drive 4 (stack tape) is only allocated if the console call option permits allocation. If the allocation option is used, tape 4 is allocated after tapes 2 and 3.

Input Tape (T5)

Logical tape address T5 is always used for tape input to a load-and-go run. This tape is in protect status when programs have been previously placed upon the tape.

However, when a load-and-go run includes diagnostic preprocessing to tape, a work tape in permit status is mounted as tape 5. The diagnostic preprocessor writes programs onto the tape and these are then processed by the load-and-go run. This is the only load-and-go run for which tape 5 is in permit status. The presence of one or more *DIAG, T cards in the input deck causes diagnostic preprocessing to tape. These cards have the following form:



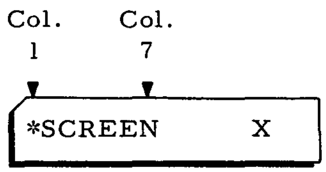
A T somewhere in columns 7 to 72 indicates preprocessing to tape. When the option to preprocess to tape is used, input must be from cards.

Output to Tape

Any unassigned logical tape address except T5 can be used for a standard output device. The tape address must be indicated by a console call option. The same tape may be used in place of output to a printer and to a common punch during execution.

SCREEN RUN

A Screen run converts certain statements written in Fortran II language to the language of Fortran D. The standard console call is used for Screen. A punching option must be indicated in the console call. The console call is followed by a *SCREEN control card or card image on tape, which has the following format:



When a Screen run immediately follows a load-and-go run (no setup or operating procedures required), the initial console call serves for both runs and must contain the punching option. A Screen run must be terminated by an *ENDRUN card or card image.

Required and optional equipment for a Screen run is listed in Table 9-4. Figure 9-3 shows the minimum equipment setup for a Screen run.

Table 9-4. Equipment for Screen

Peripheral Address	Logical Tape Drive Address	Tape Status	Other Peripheral Equipment	Required or Optional
Compiler System Tape	0	protect		Required
Common Input Device				One Device Required
On-line	—	—	Card Reader	
Off-line	5	permit	—	
Common Output Device				One Device Required
On-line	—	—	Printer	
Off-line	Any tape address not otherwise assigned	permit	—	
Common Punch Device				One Device Required (Punch tape and Print tape may be the same tape.)
On-line	—	—	Punch	
Off-line	Any tape address not otherwise assigned	permit	—	

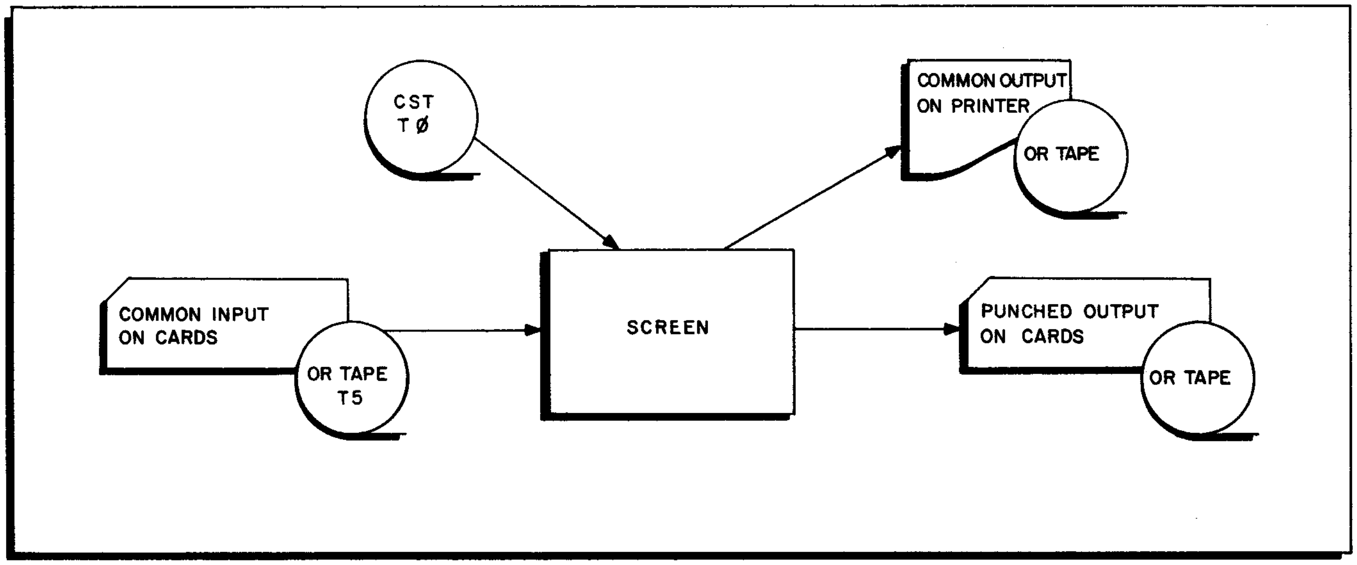


Figure 9-3. Screen Equipment Configuration

WRITING A GO-LATER TAPE

In this run mode, one or more jobs are compiled, relocated, and written onto a binary run tape but not executed. The binary run tape is then dismounted and saved for execution in another run. Required and optional equipment is the same as for a load-and-go-run. See Table 9-3 for equipment and Figure 9-2 for the minimum run configuration.

Starting and Terminating a Write Go-Later Tape Run

For card input the Console Call card must contain the appropriate go-later option in column 15. If there is a binary run tape on tape drive 1, the programs already on the tape must be protected by positioning the tape to the end of the last job on the BRT. The B option in column 15 is used to indicate a BRT on tape 1. However, if tape 1 is a work tape, a BRT must be generated by copying the loader onto the tape and then writing go-later jobs. An L option in column 15 causes the loader to be copied and a BRT to be generated onto the work tape.

It is essential that a run which writes a go-later tape be terminated with an *ENDRUN card. If the card or card image is not present, the jobs written onto the go-later tape will be destroyed.

Emergency Restart Option

Note that column 15 can contain an E option. This option is used only when a run restart is necessary. The go-later tape (T1) must not be rewound. The operator follows the run procedures given in Figure 9-1 for unprogrammed halts or looping. A Console Call card with an E in column 15 is placed in front of the next *JOBID card in the input deck. Then standard starting procedures are followed. The presence of the E option causes the go-later tape to be backspaced to the end of the last good job on the tape. A console call with an E must be used or the BRT will be rewound and all jobs destroyed.

Card and Tape Input Option

If tape input is used to write a go-later tape, it is presumed that tape input will be used to execute the tape. In the same way, if card input is used in writing a go-later tape, card input is presumed for the data to execute the tape.

Some installations, however, may wish to use card input to write the go-later tape and tape input for execution of the same tape. This is permitted if at the beginning of the run that writes the go-later tape, the operator presses SENSE switch 1 on the console. The SENSE switch is turned ON in this manner immediately after the operator presses the INITIALIZE button, as indicated in the paragraph on starting procedures.

EXECUTING A GO-LATER TAPE

When a binary run tape has been written and saved as previously described, it can then be used as input to the Execute Go-later run. In this run one or more jobs on a saved BRT will be executed. Equipment requirements are shown in Table 9-5. The minimum run configuration is shown in Figure 9-4. Work tapes for executing the go-later run were allocated when the go-later tape was written.

Table 9-5. Equipment to Execute Go-Later Jobs

Peripheral Device	Logical Tape Drive Address	Tape Status	Other Peripheral Equipment	Required or Optional
Binary Run Tape	0	protect		Required
Common Input Device				One Device Required
On-line	—	—	Card Reader	
Off-line	5	protect	—	
Common Output Device				One Device Required
On-line	—	—	Printer	
Off-line	Any tape address not otherwise assigned	permit	—	
Common Punch Device				One Device Optional (Punch tape and Print tape may be the same tape.)
On-line	—	—	Punch	
Off-line	Any tape address not otherwise assigned	permit	—	

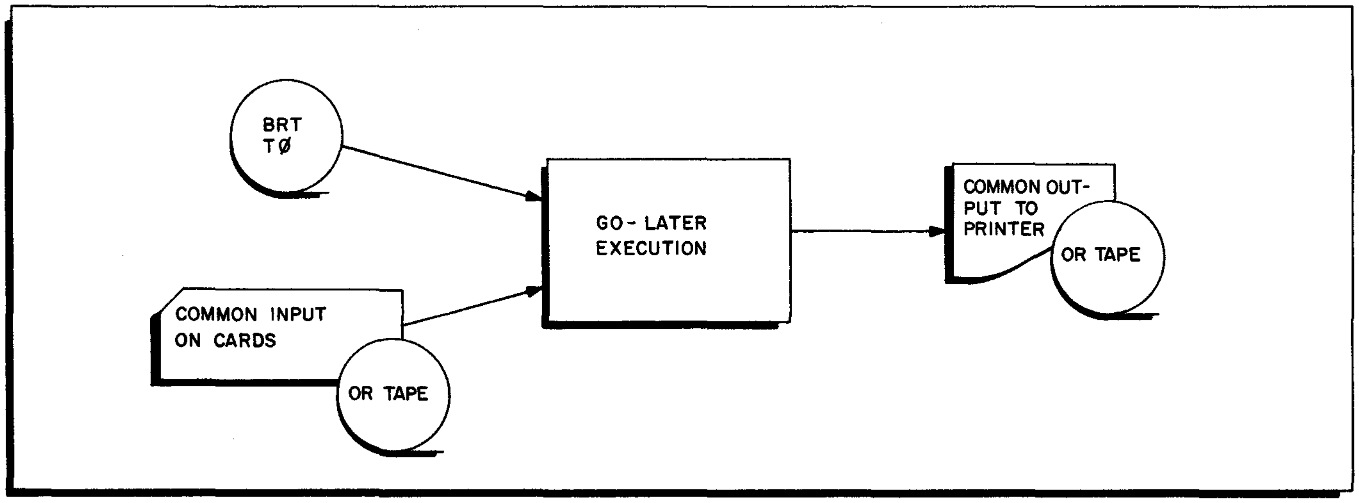
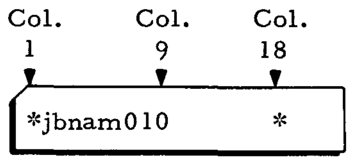


Figure 9-4. Minimum Equipment Configuration for Go-Later Execution

Starting a Go-Later Execution Run

The console calls for execution of a go-later tape contain the names of the jobs on the tape to be executed. These are six-character names with an asterisk as the first character. The console call in card form is as follows:



The characters *jbnam represent the name of a job on the go-later tape. The only option that may appear on the console call is the punch option.

The console call can be keyed in at the second loader halt. For example, if the name of the first program on the tape to be run is *JOB25, the appropriate minimum console keyin would be:

Octal Keyin	Octal Location
WM01	100
WM54	104
41	105
46	106
22	107
02	110
05	111
WM00	112
01	113
RM54	125

Starting procedures other than the contents of the console call are the same as for other runs. An *ENDRUN card must terminate the input deck.

CREATING A COMPILER SYSTEM TAPE (CST)

Distribution of the Fortran Compiler D System is in the form of a symbolic program tape (SPT). To create a CST, the installation performs an assembly (actually, a dummy assembly) of the SPT onto a transaction binary tape (TBT). Then the TBT is used as input to an update and select run in which the compiler system tape is generated. Honeywell supplies card decks for the dummy assembly and the update and select runs.

When the CST is created, the input SPT should be stored for possible later use if symbolic updates are released to the field before the release of a new SPT.

For the assembly run, tapes are mounted as follows:

T0	Easycoder Assembler Program Tape	Protect
T1	Input SPT (supplied)	Protect
T2	Work tape (becomes the TBT)	Permit

For the update and select run, tapes are mounted as follows:

T0	Update and Select Program Tape	Protect
T2	Input TBT (leave tape mounted from assembly)	Protect
T4	Work tape (becomes the CST)	Permit

Card formats and operating procedures for Easycoder assembly and for update and select are given in the software bulletin entitled Operating System - Mod 1 Operating Procedure Summaries, Order No. 069.

SECTION X

GENERAL PROGRAMMING CONSIDERATIONS

LANGUAGE LIMITATIONS

Table 10-1 describes the language limitations.

Table 10-1. Language Limitations

Limitation	Maximum	Page Reference
Programs and Specification Statements		
Number of chains in a job	30	1-1
Continuation lines in a statement	9	1-3
Number of characters in a name	6	1-9
Highest label number assigned to a statement	99999	1-7
Dimensions of an array	2	1-10
Labeled common areas in a chain	15	4-2
Number of arguments in a function or subroutine subprogram	63	6-6, 6-9
Unrelated equivalence sets in a program	64	4-6
Number of programs in a job or chain of a job for 16K memory	26	---
Assignment Statements		
Number of nested parentheses in an assignment statement	63	---
Control Statements		
Number of statement labels in a computed GO TO	63	3-2
Depth of DO loop nesting	10	3-6
Number of variables in subscript expressions within a DO loop	20	---
Number of redefined variables in a DO loop	15	---
I/O Statements		
Number of logical devices useable in the object program	15	5-2
Depth of implied DO loop nesting	2	5-10
Field width permissible in an E, F, G, I or O conversion	32	5-15
Field width permissible in an A, H, or X conversion	132	5-15
Number of times a specification field can be repeated	132	5-15
Depth of nested parentheses in a FORMAT	3	5-47
Record Width:		
Printer (not including printer control character)	131	5-46
Punch	80	5-46
BCD tape record	132	5-46
Total number of statement labels (Each FORMAT statement label is counted twice for the total.)	157	10-2

SOURCE PROGRAM SIZE LIMITATIONS

Four tables that are built in memory during compilation restrict the size of source programs. These tables are:

Source Table,

Token Table,

IEFN Table,

FORMAT Table.

When memory available for compilation is restricted to 16K characters, the source and token tables share a common block of memory. As the source table is built forward in memory, the token table is built backward. The IEFN and FORMAT tables also share a common block of memory when only 16K is available for compilation. When larger memory is available, more space is allotted to the four tables, and the source and FORMAT tables have separate blocks of memory, while the IEFN and token tables share a block.

Appendix F shows how memory is allocated to the tables at each memory size level. If table overflow occurs, the compiler issues a diagnostic.

Source Table

For each source statement, the source table has the following entries:

4 characters for IFN (Internal Formula Number), type of statement, and terminator;

2 characters for each variable name;

2 characters for each reference to a statement label; and

1 character for each operator in an arithmetic statement.

Thus, A = B + C would result in 12 characters in the source table.

Token Table

Every unique variable or constant in the source program has a token table entry equal to the number of characters in the variable or constant, plus five more characters of information. Dimensioned variables have an additional three token table characters for each dimension.

Thus, A = B + C would result in 18 characters in the token table. Note that if each variable in the example had a six-character tag, the number of characters in the token table would have been 33.

The simple arithmetic statement, A = B + C, therefore occupies 30 characters out of a maximum of 4096 in the source/token table block. When source programs overflow any table, the job is diagnosed as fatal.

Note that there is only one entry in the token table for each variable or constant, while each reference to a variable or constant has an entry in the source table.

IEFN Table

The IEFN table uses five characters for each EFN (statement label) and two characters for each corresponding IFN (internal formula number). Thus, each statement label in the source program requires seven characters of memory.

FORMAT Table

The FORMAT table uses seven characters of memory for each FORMAT statement in the source program. Since every FORMAT statement is labeled, the combined storage cost of a FORMAT statement for both the IEFN and FORMAT tables is 14 characters.

SIZE OF PROGRAM STRING

On a 16K computer, the maximum size of one program unit — main program or subprogram — is about 8.5K characters. Computers with greater memory can have a proportionately larger program string up to a maximum of about 20K characters in three-character address form.

Compilation is performed in three-character address form whether the three- or four-character address mode is used. The coding generated by the compiler is expanded to four-character address form by the run-tape generator when using the four-character address mode.

COMPILER CHARACTERISTICS AND LIMITATIONS

1. Statement operators cannot cross continuation cards.
2. When the data deck for a job contains more data than is actually read by the job, the additional cards will be listed.
3. Any attempt to reference a variable that has not been previously defined will probably cause part of memory to be wiped out. "Previously defined" means that the variable must have had information stored in it by appearing on the left-hand side of an arithmetic statement, in a DATA statement, or in the list of a READ statement.

   Example:
   ```
     SUBROUTINE SUB
     GO TO 5
     N = 1
   5 J = N+1
   ```

   Transfer of control has bypassed storing anything in the variable N. In some compilers, N would have been initialized to zero before attempting execution, so that no problem other than improper execution would result. However, in the Fortran Compiler D, a large portion of memory would probably be wiped out by lack of punctuation.

4. Jobs containing binary decks or stack tape programs must include a source program that references common storage and/or I/O devices for each chain. The source program may only be a dummy program.

5. Statement labels are not permitted on continuation cards.

6. An END statement must be used to terminate programs.

7. Frequently called subroutines should appear early in the source deck of a job.

8. All statement operators are reserved words. The reserved words are:

ASSIGN	DO	GO TO	RETURN
BACKSPACE	END	IF	REWIND
CALL	END FILE	INTEGER	STOP
COMMON	EQUIVALENCE	LOGICAL	SUBROUTINE
CONTINUE	EXTERNAL	PAUSE	TITLE
DATA	FORMAT	READ	WRITE
DIMENSION	FUNCTION	REAL	

The following rules apply to reserved words:

a. An IF followed by a left parenthesis at the beginning of a statement is always considered to be a statement operator.

b. A DO immediately followed by a digit at the beginning of a statement is always assumed to be a statement operator.

c. A FORMAT at the beginning of a statement is always assumed to be a statement operator.

d. When any other reserved word begins a statement and the next delimiter is not an equal sign, the reserved word is assumed to be a statement operator.

e. When the words IF, DO, or FORMAT appear anywhere except at the beginning of a statement, they are assumed to be user names.

f. When any reserved word except IF, DO or FORMAT appears under circumstances different from those described in rule d., it is assumed to be a user name.

g. Use of embedded blanks and continuation cards does not alter rules a. through f.

h. The rules above set the minimum restrictions on use of reserved words. The rules are intended primarily for users converting programs used on other compilers who wish to make minimum changes. When writing programs for the Fortran Compiler D, it is best to observe the following rule:

> Do not use a reserved word as a variable, array, or function name.

TIPS FOR SAVING SPACE AND TIME

1. Terminate DO loops with CONTINUE statements.

2. Use labeled common storage instead of arguments when calling a subroutine.
3. Use statement labels only when required.
4. Chains that call each other frequently should be adjacent within a job.
5. Use integer exponents whenever possible:

   A = B**2 takes less storage than A = B**2.0

6. When an exponential expression occurs only once in a program, use multiplication instead of exponentiation. For example:

   A * A takes less storage than A**2.

7. When iterating an evaluation of an exponential expression:

```
     C=ALOG(B)                   )   takes less    (    DO 4  I=1, 100
     DO 4  I=1, 100              }   storage than  {
4    A(I)=EXP((FLOAT(I)+.5)*C)   )                 (4   A(I)=B**(FLOAT(I)+.5)
```

   In the statement sequence on the right, the natural logarithm of B must be taken for each of the 100 iterations, whereas on the left the logarithm is taken before the DO loop and the resultant value is used in each of the iterations without recomputation.

8. Use the smallest integer and/or floating-point precision that will permit accurate data manipulation without overflow.

CHAINING

There are no set rules that can be given on the best method of chaining programs within a job. Experience at chaining will help the programmer determine how jobs may best be chained. A job should not be chained unless it is probable that it will not fit into memory as a unit, since chains are brought in from tape, slowing down execution.

A number of factors affect whether or not a job requires chaining — memory size available, length of the program string, data storage required, and the number and size of the execution packages used.

One method of dividing a large job into chains is to put all I/O operations in one chain, all internal computations in another chain, and library function computations in still another chain as follows:

Chain 1 - Input/Output operations;

Chain 2 - Internal computations; and

Chain 3 - Computations using library functions.

Figure 10-1 shows a simple example of chaining, using three chains. These are:

Chain 1 - Input operations;

Chain 2 - Internal computations; and

Chain 3 - Output operations.

Note that this is only an example and that a minimum deck configuration is used. Remember the following in determining how to divide a job into chains:

1. A chain may include up to 26 program units (main programs and subprograms) if they will fit into a 16K memory. Larger memory configurations can have more programs in a chain.
2. A chain may be called by other chains as many times as required during a job.
3. The first chain of the job does not need a chain identifier (*CHAIN, x card). However, if it has no identifier, it cannot be called by later chains.
4. The ordering of chains should be suited to the job. In Figure 10-1, for example, it might be more advantageous to use two chains: the first for computations and the second for all I/O. By setting switches in common storage, the job could switch back and forth between the two chains to read, compute, write, read, compute, write, etc.
5. All communication between chains is carried on through common storage. Variables in unlabeled common storage are stored on a job basis. Variables in labeled common storage are stored on a chain basis.

Once a chain is entered, the programmer may wish to branch immediately to another part of the chain. To do so, set a variable in unlabeled common storage to 1, 2, 3, etc. Then use the variable in a computed GO TO statement at the beginning of the chain. This technique is particularly useful in dividing large existing programs into chains for Fortran Compiler D.

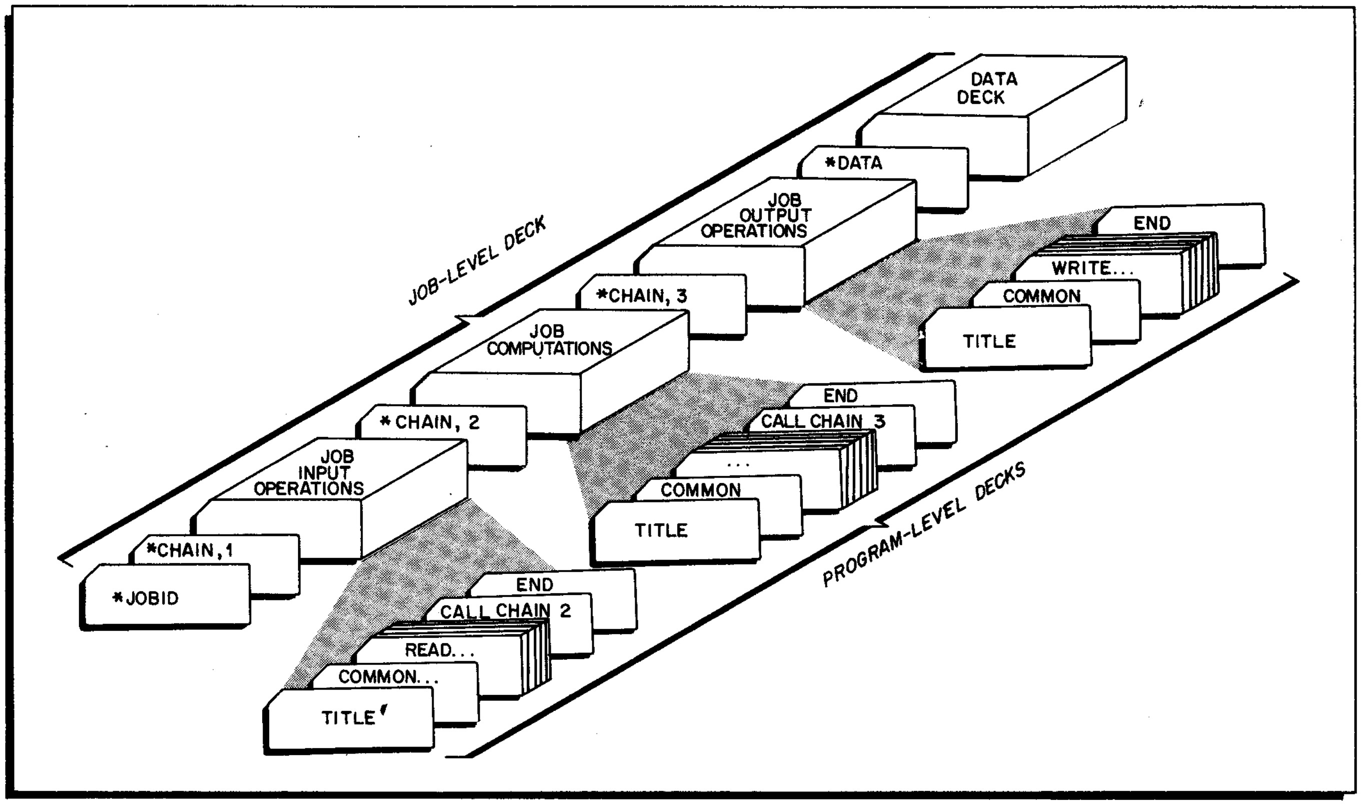


Figure 10-1. Example of Chaining

I/O PROGRAMMING TIPS

Following is a list of tips on using the I/O routines. Some of these tips are found elsewhere in this manual and are repeated here as a reminder to the programmer.

1. E, F, and G conversions must be used with floating-point variables.
2. E, F, G, I or O conversions have a maximum field width of 32.
3. A, I, and O conversions must be used with integer variables.
4. L conversions must be used with logical variables.
5. Every field specification should be followed by a field separator — , or / or).
6. The terminal right parenthesis in a FORMAT statement causes a new record to be read or written if the I/O list is not satisfied.
7. It is illegal to read on a device when the previous instruction to that device was either a WRITE or an END FILE statement.
8. REWIND statements are issued to rewind all tapes before use.
9. A WRITE statement after a REWIND on binary tapes causes a Fortran header to be written.
10. If it is desired to print six lines with five values on each line:

    This sequence causes incorrect results: FORMAT (6(1HΔ, 5E20. 10))

    Use this sequence: FORMAT (1HΔ, 5E20. 10)
11. Allocation of devices at execution time:

    All runs except go-later. No "A" punch in column 14 of Console Call card:

    Card reader, printer and punch to 2, 3, and 5 (or to iioopp of *JOBID I/O option). Remaining logical devices to T2, T3, T5, T6, T7; then channel 3: T0, T1, . . . , T7

    All runs except go-later. "A" punch in column 14 of Console Call card:

    Card reader, printer, and punch to 2, 3, and 5 (or to iioopp of *JOBID I/O option). Remaining logical devices to T2, T3, T4, T5, T6, T7; then channel 3: T0, T1, . . . , T7.

    Go-later runs.

    Card reader, printer, and punch to 2, 3, and 5 (or to iioopp of *JOBID I/O option). Remaining logical devices to T1, T2, T3, T4, T5, T6, T7; then channel 3: T0, T1, . . . , T7.
12. Vertical spacing (carriage control) of the printer is specified by the first character in a data record. If the character is 0 or 1, ASA Fortran conventions for single-spacing or spacing to the head of form are followed. If the character is between 2 and 9, that number of lines will be spaced before printing. If the character is nonnumeric, the compiler takes the low-order 4 bits and interprets them as a number. For example, an E (octal 25) will cause 5 lines to be spaced before printing.
13. An asterisk * cannot be the first character in a BCD data record being read. It will cause job fatality and result in a diagnostic.
14. Each nonstandard device allocation causes buffers to be allotted. The cost in characters for nonstandard allocation is as follows:

Punch	- 168 characters
BCD Tape	- 272 characters
Binary Tape	- 272 characters

15. Easycoder subroutines should normalize any floating-point data they generate or work with if this data is to be operated on by Fortran-generated code.

16. The following statements are illegal if unit device i is a card reader, printer, or punch:

    REWIND i   BACKSPACE i   END FILE i

17. To determine the number of physical records in a logical binary record, compute:

$$C = n_f(p_f+3) + n_i(p_i+1) + 2n_L$$

Where: n_f = number of floating-point numbers in the record.

n_i = number of integer numbers in the record.

p_f = floating-point precision.

p_i = integer precision.

n_L = number of logical numbers in the record.

Then, the smallest integer greater than or equal to C/124 is the number of physical records in the logical binary record.

18. Object I/O coding is segmented into a series of modules. Only the modules required for execution of a given job are loaded at object time, thus making additional space available for the program string. See Appendix D for a list of the object I/O modules, when each is brought into memory, and the approximate number of characters required for the module.

19. Appendix F shows the layouts of binary and BCD tapes. Use of I/O statements to write header and trailer records on these tapes and to perform other binary and BCD tape operations is explained in that appendix.

CONVERSION TECHNIQUES

Programs written in Fortran II should be checked for consistency with Fortran D language before attempting to use the compiler. The following procedure is recommended:

1. Screen the Fortran II program to convert I/O statements and names of library functions.
2. Check the Screen listing for other Fortran II statements that are handled by subroutines in Fortran D, such as IF SENSE SWITCH, IF END OF FILE, etc. Replace these with subroutine calls. (See Section VI.)
3. Check for illegal statements peculiar to a given compiler, such as SPACE, SKIP, EXECUTE PROCEDURE, etc. Replace these with Fortran D statements appropriate to the operation.
4. Preprocess the deck for diagnostics. Make any corrections required. (Note that if illegal statements remain in the screened deck, spurious diagnostics may be issued. Check any spurious diagnostic to determine if an illegal statement is still in the program.)
5. If corrections required were extensive, preprocess and correct again.
6. Compile.

Programs written in Automath 800/1800 should be checked for consistency with Fortran D and for memory size required. The following procedure is recommended.

1. Check the length of the program against the general rules for maximum length given in this section under "Chaining."
2. Check for illegal statements, such as BLOCK DATA or ABNORMAL.
3. Check for carriage control discrepancies. In Fortran Compiler D, the first character of a data record is used for carriage control. (See I/O programming tip No. 14 in this section.) In Automath, the first character is used for carriage control only when a 1H__ field specification appears in the FORMAT statement.
4. Use chaining techniques and replace illegal statements with appropriate Fortran D statements.
5. Preprocess the deck for diagnostics. Make any corrections required. (As in Fortran II decks, spurious diagnostics may indicate illegal statements that have not yet been replaced.)
6. If corrections are extensive, preprocess again.
7. Compile.

SECTION XI
THREE-CHARACTER AND FOUR-CHARACTER ADDRESS MODES

The Fortran D System can be run in either the three-character or the four-character addressing mode. Installations having central processors with 32,768 or more characters of memory have the option of running programs in the four-character mode. Installations with less than 32,768 characters of memory, which is the limit that can be accessed by three-character addressing, must run in the three-character mode.

Program units are always compiled in the three-character mode. The run-tape generator converts the compiler-generated code to the four-character mode when four-character addressing is specified. This mode is specified by a console call option — a 4 in column 17 of the Console Call card or an 04 keyed into location 124_8 by the operator if the console call is keyed in. The string of object coding produced during conversion to four-character addressing is about 1.285 times as long as in three-character addressing. Three- to four-character interfaces are used to provide communication between the four-character object string and the execution routines.

Object programs are run in the three-character address mode unless the four-character mode is specified. The four-character loader is brought in by the operator by keying an octal 20 into location 124_8 during bootstrapping.

When a go-later tape is executed, each job requires a separate console call; therefore, three- and four-character address mode jobs can be interspersed. All other types of Fortran D runs require that only one address mode be used for a given run.

The four-character addressing mode allows programs to access memory above location 32,768 at execution time. However, execution in the four-character mode is necessarily somewhat slower than in the three-character mode, and some memory space is required for four-character interfacing. If a job can be executed in 32K characters of memory, it is recommended that three-character addressing be used.

The choice of the three- or four-character addressing mode affects the conditions for the use of regionalized Easycoder programs in Fortran jobs. See Appendix E for a description of how Easycoder programs are coded for Fortran jobs using both the three- and four-character addressing modes.

APPENDIX A
OCTAL-DECIMAL CONVERSION PROCEDURE

Table A-1. Octal-Decimal Conversion Table

	DECIMAL INCREMENT																									
LOW-ORDER OCTAL DIGIT																										LOW-ORDER OCTAL DIGIT
0	000	008	016	024	032	040	048	056	064	072	080	088	096	104	112	120	128	136	144	152	160	168	176	184	192	0
1	001	009	017	025	033	041	049	057	065	073	081	089	097	105	113	121	129	137	145	153	161	169	177	185	193	1
2	002	010	018	026	034	042	050	058	066	074	082	090	098	106	114	122	130	138	146	154	162	170	178	186	194	2
3	003	011	019	027	035	043	051	059	067	075	083	091	099	107	115	123	131	139	147	155	163	171	179	187	195	3
4	004	012	020	028	036	044	052	060	068	076	084	092	100	108	116	124	132	140	148	156	164	172	180	188	196	4
5	005	013	021	029	037	045	053	061	069	077	085	093	101	109	117	125	133	141	149	157	165	173	181	189	197	5
6	006	014	022	030	038	046	054	062	070	078	086	094	102	110	118	126	134	142	150	158	166	174	182	190	198	6
7	007	015	023	031	039	047	055	063	071	079	087	095	103	111	119	127	135	143	151	159	167	175	183	191	199	7
DECIMAL BASE NO.																										DECIMAL BASE NO.
0000	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17	20	21	22	23	24	25	26	27	30	0000
0200	31	32	33	34	35	36	37	40	41	42	43	44	45	46	47	50	51	52	53	54	55	56	57	60	61	0200
0400	62	63	64	65	66	67	70	71	72	73	74	75	76	77	100	101	102	103	104	105	106	107	110	111	112	0400
0600	113	114	115	116	117	120	121	122	123	124	125	126	127	130	131	132	133	134	135	136	137	140	141	142	143	0600
0800	144	145	146	147	150	151	152	153	154	155	156	157	160	161	162	163	164	165	166	167	170	171	172	173	174	0800
1000	175	176	177	200	201	202	203	204	205	206	207	210	211	212	213	214	215	216	217	220	221	222	223	224	225	1000
1200	226	227	230	231	232	233	234	235	236	237	240	241	242	243	244	245	246	247	250	251	252	253	254	255	256	1200
1400	257	260	261	262	263	264	265	266	267	270	271	272	273	274	275	276	277	300	301	302	303	304	305	306	307	1400
1600	310	311	312	313	314	315	316	317	320	321	322	323	324	325	326	327	330	331	332	333	334	335	336	337	340	1600
1800	341	342	343	344	345	346	347	350	351	352	353	354	355	356	357	360	361	362	363	364	365	366	367	370	371	1800
2000	372	373	374	375	376	377	400	401	402	403	404	405	406	407	410	411	412	413	414	415	416	417	420	421	422	2000
2200	423	424	425	426	427	430	431	432	433	434	435	436	437	440	441	442	443	444	445	446	447	450	451	452	453	2200
2400	454	455	456	457	460	461	462	463	464	465	466	467	470	471	472	473	474	475	476	477	500	501	502	503	504	2400
2600	505	506	507	510	511	512	513	514	515	516	517	520	521	522	523	524	525	526	527	530	531	532	533	534	535	2600
2800	536	537	540	541	542	543	544	545	546	547	550	551	552	553	554	555	556	557	560	561	562	563	564	565	566	2800
3000	567	570	571	572	573	574	575	576	577	600	601	602	603	604	605	606	607	610	611	612	613	614	615	616	617	3000
3200	620	621	622	623	624	625	626	627	630	631	632	633	634	635	636	637	640	641	642	643	644	645	646	647	650	3200
3400	651	652	653	654	655	656	657	660	661	662	663	664	665	666	667	670	671	672	673	674	675	676	677	700	701	3400
3600	702	703	704	705	706	707	710	711	712	713	714	715	716	717	720	721	722	723	724	725	726	727	730	731	732	3600
3800	733	734	735	736	737	740	741	742	743	744	745	746	747	750	751	752	753	754	755	756	757	760	761	762	763	3800
4000	764	765	766	767	770	771	772	773	774	775	776	777	1000	1001	1002	1003	1004	1005	1006	1007	1010	1011	1012	1013	1014	4000
4200	1015	1016	1017	1020	1021	1022	1023	1024	1025	1026	1027	1030	1031	1032	1033	1034	1035	1036	1037	1040	1041	1042	1043	1044	1045	4200
4400	1046	1047	1050	1051	1052	1053	1054	1055	1056	1057	1060	1061	1062	1063	1064	1065	1066	1067	1070	1071	1072	1073	1074	1075	1076	4400
4600	1077	1100	1101	1102	1103	1104	1105	1106	1107	1110	1111	1112	1113	1114	1115	1116	1117	1120	1121	1122	1123	1124	1125	1126	1127	4600
4800	1130	1131	1132	1133	1134	1135	1136	1137	1140	1141	1142	1143	1144	1145	1146	1147	1150	1151	1152	1153	1154	1155	1156	1157	1160	4800
5000	1161	1162	1163	1164	1165	1166	1167	1170	1171	1172	1173	1174	1175	1176	1177	1200	1201	1202	1203	1204	1205	1206	1207	1210	1211	5000
5200	1212	1213	1214	1215	1216	1217	1220	1221	1222	1223	1224	1225	1226	1227	1230	1231	1232	1233	1234	1235	1236	1237	1240	1241	1242	5200
5400	1243	1244	1245	1246	1247	1250	1251	1252	1253	1254	1255	1256	1257	1260	1261	1262	1263	1264	1265	1266	1267	1270	1271	1272	1273	5400
5600	1274	1275	1276	1277	1300	1301	1302	1303	1304	1305	1306	1307	1310	1311	1312	1313	1314	1315	1316	1317	1320	1321	1322	1323	1324	5600
5800	1325	1326	1327	1330	1331	1332	1333	1334	1335	1336	1337	1340	1341	1342	1343	1344	1345	1346	1347	1350	1351	1352	1353	1354	1355	5800
6000	1356	1357	1360	1361	1362	1363	1364	1365	1366	1367	1370	1371	1372	1373	1374	1375	1376	1377	1400	1401	1402	1403	1404	1405	1406	6000
6200	1407	1410	1411	1412	1413	1414	1415	1416	1417	1420	1421	1422	1423	1424	1425	1426	1427	1430	1431	1432	1433	1434	1435	1436	1437	6200
6400	1440	1441	1442	1443	1444	1445	1446	1447	1450	1451	1452	1453	1454	1455	1456	1457	1460	1461	1462	1463	1464	1465	1466	1467	1470	6400
6600	1471	1472	1473	1474	1475	1476	1477	1500	1501	1502	1503	1504	1505	1506	1507	1510	1511	1512	1513	1514	1515	1516	1517	1520	1521	6600
6800	1522	1523	1524	1525	1526	1527	1530	1531	1532	1533	1534	1535	1536	1537	1540	1541	1542	1543	1544	1545	1546	1547	1550	1551	1552	6800
7000	1553	1554	1555	1556	1557	1560	1561	1562	1563	1564	1565	1566	1567	1570	1571	1572	1573	1574	1575	1576	1577	1600	1601	1602	1603	7000
7200	1604	1605	1606	1607	1610	1611	1612	1613	1614	1615	1616	1617	1620	1621	1622	1623	1624	1625	1626	1627	1630	1631	1632	1633	1634	7200
7400	1635	1636	1637	1640	1641	1642	1643	1644	1645	1646	1647	1650	1651	1652	1653	1654	1655	1656	1657	1660	1661	1662	1663	1664	1665	7400
7600	1666	1667	1670	1671	1672	1673	1674	1675	1676	1677	1700	1701	1702	1703	1704	1705	1706	1707	1710	1711	1712	1713	1714	1715	1716	7600
7800	1717	1720	1721	1722	1723	1724	1725	1726	1727	1730	1731	1732	1733	1734	1735	1736	1737	1740	1741	1742	1743	1744	1745	1746	1747	7800
8000	1750	1751	1752	1753	1754	1755	1756	1757	1760	1761	1762	1763	1764	1765	1766	1767	1770	1771	1772	1773	1774	1775	1776	1777	2000	8000
8200	2001	2002	2003	2004	2005	2006	2007	2010	2011	2012	2013	2014	2015	2016	2017	2020	2021	2022	2023	2024	2025	2026	2027	2030	2031	8200
8400	2032	2033	2034	2035	2036	2037	2040	2041	2042	2043	2044	2045	2046	2047	2050	2051	2052	2053	2054	2055	2056	2057	2060	2061	2062	8400
8600	2063	2064	2065	2066	2067	2070	2071	2072	2073	2074	2075	2076	2077	2100	2101	2102	2103	2104	2105	2106	2107	2110	2111	2112	2113	8600
8800	2114	2115	2116	2117	2120	2121	2122	2123	2124	2125	2126	2127	2130	2131	2132	2133	2134	2135	2136	2137	2140	2141	2142	2143	2144	8800
9000	2145	2146	2147	2150	2151	2152	2153	2154	2155	2156	2157	2160	2161	2162	2163	2164	2165	2166	2167	2170	2171	2172	2173	2174	2175	9000
9200	2176	2177	2200	2201	2202	2203	2204	2205	2206	2207	2210	2211	2212	2213	2214	2215	2216	2217	2220	2221	2222	2223	2224	2225	2226	9200
9400	2227	2230	2231	2232	2233	2234	2235	2236	2237	2240	2241	2242	2243	2244	2245	2246	2247	2250	2251	2252	2253	2254	2255	2256	2257	9400
9600	2260	2261	2262	2263	2264	2265	2266	2267	2270	2271	2272	2273	2274	2275	2276	2277	2300	2301	2302	2303	2304	2305	2306	2307	2310	9600
9800	2311	2312	2313	2314	2315	2316	2317	2320	2321	2322	2323	2324	2325	2326	2327	2330	2331	2332	2333	2334	2335	2336	2337	2340	2341	9800
10,000	2342	2343	2344	2345	2346	2347	2350	2351	2352	2353	2354	2355	2356	2357	2360	2361	2362	2363	2364	2365	2366	2367	2370	2371	2372	10,000
10,200	2373	2374	2375	2376	2377	2400	2401	2402	2403	2404	2405	2406	2407	2410	2411	2412	2413	2414	2415	2416	2417	2420	2421	2422	2423	10,200
10,400	2424	2425	2426	2427	2430	2431	2432	2433	2434	2435	2436	2437	2440	2441	2442	2443	2444	2445	2446	2447	2450	2451	2452	2453	2454	10,400
10,600	2455	2456	2457	2460	2461	2462	2463	2464	2465	2466	2467	2470	2471	2472	2473	2474	2475	2476	2477	2500	2501	2502	2503	2504	2505	10,600
10,800	2506	2507	2510	2511	2512	2513	2514	2515	2516	2517	2520	2521	2522	2523	2524	2525	2526	2527	2530	2531	2532	2533	2534	2535	2536	10,800
11,000	2537	2540	2541	2542	2543	2544	2545	2546	2547	2550	2551	2552	2553	2554	2555	2556	2557	2560	2561	2562	2563	2564	2565	2566	2567	11,000
11,200	2570	2571	2572	2573	2574	2575	2576	2577	2600	2601	2602	2603	2604	2605	2606	2607	2610	2611	2612	2613	2614	2615	2616	2617	2620	11,200
11,400	2621	2622	2623	2624	2625	2626	2627	2630	2631	2632	2633	2634	2635	2636	2637	2640	2641	2642	2643	2644	2645	2646	2647	2650	2651	11,400
11,600	2652	2653	2654	2655	2656	2657	2660	2661	2662	2663	2664	2665	2666	2667	2670	2671	2672	2673	2674	2675	2676	2677	2700	2701	2702	11,600
11,800	2703	2704	2705	2706	2707	2710	2711	2712	2713	2714	2715	2716	2717	2720	2721	2722	2723	2724	2725	2726	2727	2730	2731	2732	2733	11,800
12,000	2734	2735	2736	2737	2740	2741	2742	2743	2744	2745	2746	2747	2750	2751	2752	2753	2754	2755	2756	2757	2760	2761	2762	2763	2764	12,000
12,200	2765	2766	2767	2770	2771	2772	2773	2774	2775	2776	2777	3000	3001	3002	3003	3004	3005	3006	3007	3010	3011	3012	3013	3014	3015	12,200
12,400	3016	3017	3020	3021	3022	3023	3024	3025	3026	3027	3030	3031	3032	3033	3034	3035	3036	3037	3040	3041	3042	3043	3044	3045	3046	12,400
12,600	3047	3050	3051	3052	3053	3054	3055	3056	3057	3060	3061	3062	3063	3064	3065	3066	3067	3070	3071	3072	3073	3074	3075	3076	3077	12,600
12,800	3100	3101	3102	3103	3104	3105	3106	3107	3110	3111	3112	3113	3114	3115	3116	3117	3120	3121	3122	3123	3124	3125	3126	3127	3130	12,800
13,000	3131	3132	3133	3134	3135	3136	3137	3140	3141	3142	3143	3144	3145	3146	3147	3150	3151	3152	3153	3154	3155	3156	3157	3160	3161	13,000
13,200	3162	3163	3164	3165	3166	3167	3170	3171	3172	3173	3174	3175	3176	3177	3200	3201	3202	3203	3204	3205	3206	3207	3210	3211	3212	13,200
13,400	3213	3214	3215	3216	3217	3220	3221	3222	3223	3224	3225	3226	3227	3230	3231	3232	3233	3234	3235	3236	3237	3240	3241	3242	3243	13,400
13,600	3244	3245	3246	3247	3250	3251	3252	3253	3254	3255	3256	3257	3260	3261	3262	3263	3264	3265	3266	3267	3270	3271	3272	3273	3274	13,600
13,800	3275	3276	3277	3300	3301	3302	3303	3304	3305	3306	3307	3310	3311	3312	3313	3314	3315	3316	3317	3320	3321	3322	3323	3324	3325	13,800
14,000	3326	3327	3330	3331	3332	3333	3334	3335	3336	3337	3340	3341	3342	3343	3344	3345	3346	3347	3350	3351	3352	3353	3354	3355	3356	14,000
14,200	3357	3360	3361	3362	3363	3364	3365	3366	3367	3370	3371	3372	3373	3374	3375	3376	3377	3400	3401	3402	3403	3404	3405	3406	3407	14,200
14,400	3410	3411	3412	3413	3414	3415	3416	3417	3420	3421	3422	3423	3424	3425	3426	3427	3430	3431	3432	3433	3434	3435	3436	3437	3440	14,400
14,600	3441	3442	3443	3444	3445	3446	3447	3450	3451	3452	3453	3454	3455	3456	3457	3460	3461	3462	3463	3464	3465	3466	3467	3470	3471	14,600
14,800	3472	3473	3474	3475	3476	3477	3500	3501	3502	3503	3504	3505	3506	3507	3510	3511	3512	3513	3514	3515	3516	3517	3520	3521	3522	14,800
15,000	3523	3524	3525	3526	3527	3530	3531	3532	3533	3534	3535	3536	3537	3540	3541	3542	3543	3544	3545	3546	3547	3550	3551	3552	3553	15,000
15,200	3554	3555	3556	3557	3560	3561	3562	3563	3564	3565	3566	3567	3570	3571	3572	3573	3574	3575	3576	3577	3600	3601	3602	3603	3604	15,200
15,400	3605	3606	3607	3610	3611	3612	3613	3614	3615	3616	3617	3620	3621	3622	3623	3624	3625	3626	3627	3630	3631	3632	3633	3634	3635	15,400
15,600	3636	3637	3640	3641	3642	3643	3644	3645	3646	3647	3650	3651	3652	3653	3654	3655	3656	3657	3660	3661	3662	3663	3664	3665	3666	15,600
15,800	3667	3670	3671	3672	3673	3674	3675	3676	3677	3700	3701	3702	3703	3704	3705	3706	3707	3710	3711	3712	3713	3714	3715	3716	3717	15,800
16,000	3720	3721	3722	3723	3724	3725	3726	3727	3730	3731	3732	3733	3734	3735	3736	3737	3740	3741	3742	3743	3744	3745	3746	3747	3750	16,000
16,200	3751	3752	3753	3754	3755	3756	3757	3760	3761	3762	3763	3764	3765	3766	3767	3770	3771	3772	3773	3774	3775	3776	3777	4000	4001	16,200
16,400	4002	4003	4004	4005	4006	4007	4010	4011	4012	4013	4014	4015	4016	4017	4020	4021	4022	4023	4024	4025	4026	4027	4030	4031	4032	16,400
	HIGH-ORDER OCTAL DIGITS																									

Consider the decimal number to be converted as a base and an increment. Locate the base (the next lower number which is evenly divisible by 200) in the margin of the lower chart and the increment in the body of the upper chart. The intersection of the row and column thus defined contains the high-order digits of the octal equivalent. The low-order digit appears in the margins of the upper chart opposite the increment. For example, to convert 7958 to octal, the base is 7800 and the increment is 158. Locate 158 in the upper chart and read down this column to the 7800 row below. The high-order octal result is 1742. Then read out to the margin of the upper chart to obtain the low-order digit of 6. Append (do not add) this digit to 1742 for an octal equivalent of 17,426.

To convert an octal number to decimal, locate the high-order digits in the body of the lower chart and the low-order digit in the margin of the upper chart. Then perform the converse of the above operation.

APPENDIX B

LANGUAGE SUMMARY

This appendix summarizes language features described in this manual. The appendix has three parts. The first part compares the differences between the language of Fortran Compiler D and the ASA proposed Fortran as of March, 1965. The second part is a tabular summary of the Fortran statements used in Fortran Compiler D with a brief explanation of their purpose and an example of how they are used in the source program. The third part defines some of the language terms used in the handbook.

COMPARISON WITH ASA PROPOSED FORTRAN

Additional Statements

TITLE	– a nonexecutable statement followed by a main program name. This statement can be used optionally to name a main program. (See page 4-9).
END	– the END line of ASA proposed Fortran is replaced by an END statement of the same form. This statement must be used to terminate a main program or subprogram. (See page 3-9).
CALL CHAINx	– Chaining is the overlay technique used by the compiler. Within a job, a group of programs that occupies a separate memory load is called a chain. The CALL CHAINx statement causes transfer of control to the chain of programs named x. (See pages 1-1 and 3-8).

Terminology

Honeywell uses the term job. A job is an executable unit that can consist of (1) a single program, (2) a group of programs, or (3) up to 30 chains, each chain consisting of either (1) or (2). A chain is a single memory load within a job. (See page 1-1).

Honeywell uses the term library functions for functions that are called intrinsic functions and basic external functions in the ASA proposed specification. (See page 6-1).

Unless otherwise specified in the text, the term variable refers to a non-subscripted variable. Array element is used in place of subscripted variable. (See pages 1-10 and 1-11).

Additional Language Features

DATA Initialization	– Implied DO loops and short-list array notation are permitted in DATA initialization statements. (See page 4-9).

Formatting in an Array	– A Hollerith field descriptor is permitted to be part of a FORMAT specification in an array. (See page 5-51).
Boolean Functions	– Four Boolean functions are supplied with the compiler – Logical AND, Inclusive OR, Logical Complement, and Exclusive OR. (See pages 6-8 and 6-9).
Special Dump Subroutines	– Three subroutines (MDUMP, DUMP, and PDUMP) are supplied with the compiler to provide dynamic dumping facilities. (See page 6-14).
Hardware Test Subroutine	– One test subroutine for SENSE switches (SSWTCH) is supplied with the compiler). (See page 6-12).
Test Subroutines for Simulated Indicators	– Four test subroutines, supplied with the compiler, test simulated indicators (DVCHK, OVERFL, SLITE, and SLITET). (See page 6-12).
I/O Condition Test Subroutines	– Three subroutines (PARITY, EOF, and EOT) are supplied with the compiler to test I/O conditions. (See page 6-13).
Special I/O Subroutine	– Subroutine REREAD is supplied with the compiler to provide a facility for re-reading data. (See page 6-13).
Octal Data	– Input and output of octal data using an octal conversion specification in a FORMAT statement is permitted. Octal data may also be initialized in a DATA initialization statement. (See pages 1-14, 5-28, 4-9).

Restrictions

Array Dimensioning	– Only one- or two-dimensioned arrays are permitted. (See page 1-10).
Data Types	– No complex or double-precision data are permitted. (See page 1-11).
Specification Subprograms	– No BLOCK DATA subprograms are permitted. (See Section IV).
Extended Range DO	– No extended ranges are permitted in DO nests. (See pages 3-4 to 3-7).

Change in Parameter

The compiler permits up to six octal digits to follow a PAUSE or STOP statement as an identifier, in contrast to the five specified by the ASA proposed Fortran. (See pages 3-8 and 3-9).

Table B-1. Fortran Statement Summary

Specification Statements

Statement and Page Reference	Explanation	Example
DIMENSION $a_1(i_1, i_2), a_2(i_1, i_2), \ldots, a_n(i_1, i_2)$ Page 4-1	A DIMENSION statement declares the arrays to be used in a program and gives the number of dimensions and the size of each array dimension. Each a represents an array name. Each i_1 indicates the number of rows and each i_2 indicates the number of columns in the array. i_2 is present only if the array is two dimensional.	DIMENSION A(20), T(3, 4), J(7, 7)
Unlabeled Common: COMMON $a_1, a_2, a_3, \ldots, a_n$	Each a is a single variable or array name assigned to the common area in the order in which it appears in the COMMON statement. Unlabeled common storage is accessible to all programs within a single job.	COMMON A, B(5, 6), K, M(4, 4)
Labeled Common: COMMON/N_1/$a_1, \ldots, a_n$/N_2/$a_1, \ldots, a_n$/N_n/$a_1, \ldots, a_n$ Page 4-2	Each list of names of variables or arrays (a_1 to a_n) is assigned to the common block named within the slashes preceding the list. Labeled common blocks are accessible to all programs in a chain.	COMMON/BLK1/C(3, 3),D,V/BLK2/E,F(2,5)
	A single statement can be used for unlabeled or labeled common storage. The area between slashes is left blank for unlabeled common storage. Only one unlabeled common block is permitted.	COMMON/BLK1/C(3,3),D,V/BLK2/E,F(2,5)//A, B(5, 6), K, M(4, 4)

Table B-1 (cont). Fortran Statement Summary

Specification Statements (cont)

Statement and Page Reference	Explanation	Example
EQUIVALENCE $(v_1,\ldots,v_n)$, $(v_1',\ldots,v_n')$, ... Page 4-5	An EQUIVALENCE statement either renames variables or array elements or permits two or more variables or array elements to share a single memory location. Each set of parentheses encloses variables and array elements assigned to the same location. Up to 64 unrelated equivalenced sets are permitted.	EQUIVALENCE (A(2),B(3, 1)),(V,C(2,4), S) Related and Unrelated Sets EQUIVALENCE (A, B), (A, C), (D, E) The example above is counted as two unrelated equivalenced sets. Since A is a variable in two of the sets, those two sets are counted as related.
Type Statements: INTEGER $v_1, v_2,\ldots, v_n$ LOGICAL $v_1, v_2,\ldots, v_n$ REAL $v_1, v_2,\ldots, v_n$ Page 4-7	Type statements explicitly declare types of arithmetic and logical variables, arrays, or functions. Logical quantities must be explicitly declared. Real or integer values are declared to override implicit typing.	LOGICAL A, B(3, 3), C(4), D REAL M(5, 4), INT, MORTG
EXTERNAL $f_1, f_2,\ldots, f_n$ Page 4-8	Each f is the name of a function or subroutine that appears in the argument list of a function or subroutine subprogram and is not otherwise previously declared in a CALL statement (for subroutines) or an arithmetic/logical expression (for functions).	FUNCTION PHI(DUMMY1, X, Y) PHI = DUMMY1(X)/Y RETURN END } function EXTERNAL PHI .... CALL X(PHI, B) .... } part of calling program
DATA $v_1,\ldots,v_n/c_1,\ldots,c_n/$ Page 4-9	A DATA statement assigns an initial constant c to the corresponding variable v at object time. Types of corresponding constants and variables must match.	DATA G,H,I,J,K/8.75,1.0,35,3HRPM,2037/ (Note that in Fortran D, octal and Hollerith values are typed as integer data.)

Table B-1 (cont). Fortran Statement Summary

Assignment Statements

Statement and Page Reference	Explanation	Example

Arithmetic Statement:
a = b
Page 2-4
	The value of the arithmetic expression b replaces the real or integer variable a.	...
		I = I + 1
		A = I
		X = A*12./(Z-2.)

Logical Statement: a = b Page 2-6	The value of the logical expression b replaces the logical variable a. Logical variables must be explicitly declared.	LOGICAL C, Z .... C = X.GT.2.5.AND.NOT.Z
Statement Function		
Statement Function: Funame $(a_1, a_2, \ldots, a_n)$ = b Page 6-2	The statement function assigns an arithmetic or logical expression b containing dummy arguments a_1 to a_n to a function name. A list of the dummy arguments appears following the function name. The statement function permits writing an expression only once for a program in which it will be used repetitively with different actual arguments. A statement function is called by writing its name and actual arguments in an executable statement.	CALC(X, Y, Z)=X**2.*SIN(Y)+(Z-15.) .... D=BAL + CALC(A, B, C) .... IF (CALC(E, F, G) - 24.) 2, 3, 4 ....
Control Statements		
GO TO n Page 3-1	Unconditional GO TO. n is the statement label of an executable statement to which control is transferred.	.... GO TO 75 .... 75 A=25.*X ....

Table B-1 (cont). Fortran Statement Summary

Control Statements (cont)

Statement and Page Reference	Explanation	Example

GO TO $(n_1, n_2, \ldots, n_m)$, i
 	
Page 3-2
	A computed GO TO statement. The n's are statement labels. m cannot exceed 63. i is an integer variable. The value of i is from 1 through m; it indicates to which n control is transferred.	`.....`
		`      J=0`
		`5     J=J+1`
		`      GO TO (10, 20, 30), J`
		`....`
		`....`
		`10    A=B+C`
		`      GO TO 5`
		`20    D=B-C`
		`      GO TO 5`
		`30    E=A-C`
		`....`

ASSIGN n TO i .... GO TO i, $(n_1, n_2, \ldots, n_m)$ Page 3-2	An assigned GO TO and an ASSIGN statement are used together. All n's are statement labels. i is an integer variable that is the same for the ASSIGN and its matching GO TO statement. Control is transferred to the statement given in the ASSIGN when the GO TO is encountered.	ASSIGN 90 TO K .... GO TO K (66, 90, 34, 82) When the GO TO statement is encountered, transfer is made to statement 90.
IF(e) n_1, n_2, n_3 Page 3-3	e is an integer or real arithmetic expression and the n's are labels of executable statements. The n to which control is transferred depends on whether the expression is evaluated as negative, 0, or positive.	IF(X**4-16.) 7, 12, 12 If X is less than 2., control goes to statement 7. If X is equal to or greater than 2., control goes to statement 12.
IF (e) S Page 3-3	e is a logical expression and S if any executable statement except a DO or another logical IF. S is executed only if e is true.	IF (D. OR. E) GO TO 38 If either D or E is true, control transfers to statement 38; otherwise the next statein order is executed.

Table B-1 (cont). Fortran Statement Summary

Control Statements (cont)

Statement and Page Reference	Explanation	Example

DO n i = m_1, m_2, m_3
or
DO n i = m_1, m_2
 	
Page 3-4
	The DO statement permits a sequence of statements following to be executed repetitively. n is the last statement of the sequence. i is an integer variable. m_1 is the initial value and m_2 is the final value of i. m_3 is the increment of i; when m_3 is absent, the increment is 1.	`....`
		`G=0`
		`DO 25 I=1, 10, 2`
		`G=G+1`
		`25 A(I) = B(I)-G*C(I)`
		`.....`
		
		The following computations would result:
		A(1) = B(1) - 1*C(1)
		A(3) = B(3) - 2*C(3)
		A(5) = B(5) - 3*C(5)
		A(7) = B(7) - 4*C(7)
		A(9) = B(9) - 5*C(9)

CALL subname
or
CALL subname (a_1, a_2, . . . ,a_n)
 	
Page 3-7
	Used to transfer control to a procedure subroutine. The subroutine name may be followed by actual arguments (a_1 to a_n where $n \leq 63$) to be substituted for dummy arguments given in the SUBROUTINE statement that begins the subroutine subprogram.	`....`
		`CALL TEST (A, B, REMO)`
		`25 .....`
		`.....`
		`.....`
		`END` } main or calling program
		
		`SUBROUTINE TEST (X,Y,Z)`
		`....`
		`....`
		`....`
		`RETURN`
		`END` } sub-routine

RETURN Page 3-8	A RETURN statement transfers control from a function or subroutine subprogram back to the calling program. Control returns to the first executable statement after the CALL for a subroutine and to the statement in which the function is embedded for a function.	X, Y, and Z are dummy arguments of subroutine subprogram TEST. A,B, and REMO are the actual arguments substituted when TEST is called. At the RETURN statement, control is transferred to statement 25.

Table B-1 (cont). Fortran Statement Summary

Control Statements (cont)

Statement and Page Reference	Explanation	Example
CONTINUE Page 3-8	CONTINUE is used to terminate DO loops that would otherwise be illegally terminated by a GO TO, DO, arithmetic IF, RETURN, or STOP statement.	... DO 50 I = 1, 10, 2 .... IF (A(I) - B(I)) 4, 5, 6 50 CONTINUE
CALL CHAIN x Page 3-8	The CALL CHAIN statement transfers control to the named chain of programs within a job. The x is a character (letter or digit) identifying the chain.	..... CALL CHAIN 3 .....
END Page 3-9	An END statement is always required to terminate a program or subprogram.	.... END
PAUSE or PAUSE n Page 3-8	A PAUSE halts execution of a program so that the operator can take some action designated by the programmer. An n can be up to 6 octal digits displayed in the A- and B-address registers. The n indicates which of several numbered actions should be taken at this pause.	..... PAUSE 33 .....
STOP or STOP n Page 3-9	A STOP without the identification constant n causes termination of execution and exit to process the next job. A STOP with an identification constant n causes a halt as does the PAUSE statement. When the operator has taken the action indicated and resumes processing, an exit to the monitor occurs to process the next job.	.... STOP 217 END

Table B-1 (cont). Fortran Statement Summary

Input/Output Statements

Statement and Page Reference	Explanation	Example
READ (i, n) list or READ (i) list Page 5-1	The statement indicates that the list of names of variables, arrays, and array elements given is to be read from the device numbered i into memory in accordance with the input format shown in the FORMAT statement numbered n. When no n is present, the input list is unformatted. The list for both READ and WRITE statements may be a sequenced group of names separated by commas or can take the form of an implied DO loop enclosed in parentheses. Single list items can precede an implied DO loop.	READ (2, I7) A(2), B, J, P READ (2, 30) (D(I), I=1, 5) READ (2, 47) A(2), J, (D(I) I=1, 5) The final READ example reads the list into memory in the following order: A(2), J, D(1), D(2), D(3), D(4), D(5)
WRITE (i, n) list or WRITE (i) list Page 5-3	The WRITE statement is identical to the READ statement, except that the device i is an output device and the format referenced is the output format.	WRITE (3, 9) (A(2), J, D(I), I=1, 5) The WRITE example would list on the output device the following: A(2), J, D(1), A(2), J, D(2), A(2), J, D(3), A(2), J, D(4), A(2), J, D(5)
END FILE i Page 5-55	i is a peripheral device code. The effect of the END FILE statement depends upon the device being addressed. Two end-of-file records are written on a magnetic tape. Do not END FILE the punch, printer, or card reader. This causes termination of the run.	END FILE 3

Table B-1 (cont). Fortran Statement Summary

Input/Output Statements (cont)

Statement and Page Reference	Explanation	Example
REWIND i Page 5-56	The tape mounted on logical tape unit i will be rewound to the beginning of tape.	REWIND 5
BACKSPACE i Page 5-56	The tape mounted on logical tape unit i will be backspaced one logical record.	BACKSPACE 4

n FORMAT (s_1, s_2, ... s_n)
 	
Section V
	Each s is a specification for formatting output or indicating incoming format for the corresponding list item in the WRITE or READ statement that references n, the FORMAT statement number.
	
	Each specification s indicates the type of conversion (A, E, F, G, H, I, L, O, or X) and the width of the field on the external medium. Floating-point conversions (E, F, and G) include the position of the decimal point and may include a scale factor. All conversions except Hollerith (H) and blank (X) must have corresponding list items in a READ or WRITE statement. All conversions except Hollerith and blank may have repetition constants.
		`.....`
		`.....`
		`READ (2, 10) A, B, C, I, J, K, LGL`
		`.....`
		`.....`
		`.....`
		`10  FORMAT (5X, F4.1, E6.1, G8.2, I3,`
		`    O4, A6, I2, 15HΔFORMATΔ`
		`    EXAMPLE)`
		`.....`
		`.....`
		--------------------------------- 	
		Appearance of data to be read in:
		`ΔΔΔΔΔ-2.7Δ5.3E2Δ1765.85Δ25Δ777ΔALPHAΔT`

Table B-1 (cont). Fortran Statement Summary

Program Header Statements

Statement and Page Reference	Explanation	Example
Δ TITLE name Page 4-9	The TITLE statement is an optional first statement in a main program. It permits the programmer to name a main program.	Δ TITLE SURVEY .... END } main body of program
type FUNCTION name (a_1, a_2,...,a_n) or FUNCTION name (a_1, a_2,...,a_n) Page 6-4	A FUNCTION statement names a function subprogram and lists the dummy arguments ($0 \leq a_n \leq 63$) of the subprogram. The function subprogram can be explicitly typed as REAL, INTEGER, or LOGICAL, or it may follow implicit typing name conventions. The FUNCTION statement precedes the function subprogram statements. A function subprogram is called by writing its name in an executable statement together with its actual arguments.	REAL FUNCTION INTRST (X, Y, Z) .... RETURN END
SUBROUTINE name (a_1, a_2,...a_n) Page 6-9	A SUBROUTINE statement names a subroutine subprogram and lists the dummy arguments ($0 \leq a_n \leq 63$) of the subprogram. A subroutine subprogram is not typed. The SUBROUTINE statement precedes the subroutine subprogram statements. A subroutine subprogram is called by a CALL statement giving its name and any actual arguments.	SUBROUTINE EVALU (D, I, C, J) .... RETURN ..... END

GLOSSARY

Some of the language terms used in this manual which have not previously been defined or which have similar or partly similar meanings to other terms are here defined.

Fortran Syntax: This comprises the rules governing the structure of the Fortran language.

Fortran Language Elements: The Fortran language elements are operators, delimiters, and names.

Delimiters: Delimiters separate (delimit) elements of a Fortran statement and are:

+ (-) = , * /

Operators: Operators indicate action to be taken upon another element of Fortran and are:

Arithmetic operators
Logical operators
Relational operators
Statement operators

Arithmetic Operators: Four delimiters are used to indicate five arithmetic operations as follows:

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

Logical Operators: Logical operators combine with a logical constant, logical variable, logical array element, and/or reference to a logical function to produce a truth value (.TRUE. or .FALSE.). The logical operators are:

.NOT.	logical negation
.AND.	logical conjunction
.OR.	inclusive disjunction

Relational Operators: Relational operators define a relationship between arithmetic expressions such that if the logical relation is satisfied, the truth value is .TRUE. and if the logical relation is not satisfied, the truth value is .FALSE. The relational operators are:

.EQ.	equal to
.GE.	greater than or equal to
.GT.	greater than

.LE.	less than or equal to
.LT.	less than
.NE.	not equal to

Statement Operators: Statement operators are reserved words used to begin Fortran statements and indicate the action to be taken. The statement operators are:

ASSIGN	END FILE	PAUSE
BACKSPACE	EQUIVALENCE	READ
CALL	EXTERNAL	REAL
CALL CHAIN	FORMAT	RETURN
COMMON	FUNCTION	REWIND
CONTINUE	GO TO	STOP
DATA	IF	SUBROUTINE
DIMENSION	INTEGER	TITLE
DO	LOGICAL	WRITE
END		

Names: Names are assigned to all constants, variables, array elements, arrays, functions, and subroutines according to the Fortran conventions described in Section I.

Operand: An operand is a general term for a Fortran element that is neither an operator nor a delimiter. It may be a constant, variable, array element, array, function reference, or the result of previous evaluation of an expression.

Expression: An expression is a sequence of Fortran language elements that can be evaluated as a unit. Expressions are arithmetic or logical. Complete rules for defining either expression are given in Section II.

APPENDIX C

INTERNAL REPRESENTATION OF NUMBERS

BIT REPRESENTATION

Each decimal character specified by the programmer on the *JOBID card as the precision option represents six bits of storage. Of the allocated bits, the leftmost represents the sign for integer data.

FIXED-POINT NUMBERS

Fixed-point (integer) numbers are stored in binary. In considering requirements of precision, overflow conditions, etc., the programmer must therefore think, not in terms of the conventional character representation, but in terms of bits available. For example, a minimum allocation of three characters permits representation of an integer in up to 17 bits.

Length - From 3 to 12 characters can be specified by parameter card at the beginning of program compilation. If not specified, a length of three characters is assumed. Let N equal the number of characters specified.

Magnitude - For any fixed-point number i, the magnitude is restricted to:

$$0 < i \leq 2^{(6N-1)} - 1$$

Precision - Precision can be determined from the table following.

Parameter Specified by Programmer	Integer Precision in Digits	Parameter Specified by Programmer	Integer Precision in Digits
3	5	8	14
4	6	9	15
5	8	10	17
6	10	11	19
7	12	12	20

Format - The number is stored in binary in a group of N characters. If the number is negative, it will have a value equal to the twos complement of the positive number of equal magnitude. Note that this assures that a negative number has a high-order bit of 1.

FLOATING-POINT NUMBERS

A mantissa of 2 to 20 characters can be specified by control card at the beginning of program compilation. Two additional characters to the right of the mantissa represent the exponent. If not specified, a mantissa of seven characters, plus two characters for the exponent, is assumed. Both the mantissa and exponent are signed. Since real numbers are stored in floating-point decimal, the mantissa size specified by the programmer (in characters) represents the number of decimal digits that can be stored.

STORAGE OF OTHER DATA

Alphabetic, Hollerith, octal, and logical data are stored as fixed-point data. Examples illustrating storage of these data, together with examples of storage of fixed- and floating-point numbers, are given in this appendix.

ACCURACY OF CALCULATIONS

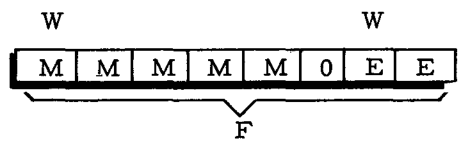
The degree of precision specified for values in a Fortran program will not necessarily be reflected in the results of calculations. For example, if floating-point precision is 10 places, the two values 88888.0 and 6666666.0 can be read into memory without overflow. However, if these values are multiplied, the result will exceed specified precision and prove accurate only to nine places. Results of further calculations with the result of this multiplication may reflect greater decreases in precision. In selecting fixed- and floating-point precision, therefore, it is important to determine the degree of precision required in calculations, not the precision required to read in initial values.

FORMATS OF DATA IN MEMORY AT OBJECT TIME

Variables are stored in memory with sign information. Real constants always have a positive mantissa and a signed (plus or minus) exponent. Integer constants are always positive.

Real Data

Real data are stored in memory in decimal. After a real number assignment or input, storage is as shown:



Where: F = 4 to 22 characters (mantissa precision from 2 to 20 plus 2 for exponent)
each M = 1 digit in the mantissa
each E = 1 digit in the exponent
W= word mark

The mantissa is normalized and stored left-justified with a zero fill at the right. If the mantissa is negative, the B and A bits in the low-order character of the mantissa are set to 10; all other settings of these two bits indicate a positive mantissa. A negative exponent is indicated when the B and A bits of the low-order character of the exponent are set to 10.

Examples:

1. This example has a precision of two decimal digits and shows the resultant data bits of the four characters. A= -9.3E — 29 gives the following in memory for the variable A:

MANTISSA		EXPONENT	
WORD MARK		WORD MARK	
B A 8 4 2 1	B A 8 4 2 1	B A 8 4 2 1	B A 8 4 2 1
0 0 1 0 0 1	1 0 0 0 1 1	0 0 0 0 1 0	1 0 1 0 0 0

The following shows how the constant (9.3E – 29) is stored in memory:

MANTISSA		EXPONENT	
WORD MARK		WORD MARK	
B A 8 4 2 1	B A 8 4 2 1	B A 8 4 2 1	B A 8 4 2 1
0 0 1 0 0 1	0 0 0 0 1 1	0 0 0 0 1 0	1 0 1 0 0 0

2. With a specified precision of 20 decimal digits, B = 97425.7761834581975E + 78 gives the following in memory for the variable and constant:

W																				W	
9	7	4	2	5	7	7	6	1	8	3	4	5	8	1	9	7	5	0	0	8	3

Integer Data

Integer storage in memory is in binary. After fixed-point input:

$$\overset{W}{\underbrace{\boxed{000XXX}}_{I}}$$

I	=	3 to 12 characters (5 to 20 digits)
W	=	word mark
XXX	=	data stored

Data are stored right-justified with zero fill on the left. For a negative number the twos complement of the binary equivalent of the number is stored in the variable location, instead of the binary equivalent itself.

Examples:

1. With a precision of four characters, I = 3947 gives the following in the constant and variable locations:

$$\underbrace{\overset{W}{\boxed{0}}\boxed{0}\boxed{0}\boxed{0}\boxed{7}\boxed{5}\boxed{5}\boxed{3}}_{\text{shown in octal}}$$

2. With a precision of three characters, I = -2997 gives the following in the variable location for I:

$$\underbrace{\overset{W}{\boxed{7}}\boxed{7}\boxed{2}\boxed{1}\boxed{1}\boxed{3}}_{\text{shown in octal}}$$

The same datum gives the following in the constant location since constants are always positive:

$$\underbrace{\overset{W}{\boxed{0}}\boxed{0}\boxed{5}\boxed{6}\boxed{6}\boxed{5}}_{\text{shown in octal}}$$

Octal Data

Octal data are stored in integer form. Integer precision between 3 and 12 characters permits between 6 and 24 octal digits to be stored, as shown below.

$$\underbrace{\overset{W}{\boxed{\text{XXX000}}}}_{I}$$

I = 6 to 24 octal digits

Data are stored left-justified with zero fill on the right.

Example:

With a precision of four characters, 7O1234567 gives the following in the variable and constant locations:

$$\underbrace{\overset{W}{\boxed{1}}\boxed{2}\boxed{3}\boxed{4}\boxed{5}\boxed{6}\boxed{7}\boxed{0}}_{\text{shown in octal}}$$

Hollerith and Alphabetic Data

The data in a Hollerith or alphabetic input are stored in integer form. Integer precision between 3 and 12 characters permits between 3 and 12 alphabetic or Hollerith characters to be stored as shown below.

$$\underbrace{\overset{W}{\boxed{\text{XXX}\Delta\Delta\Delta}}}_{I}$$

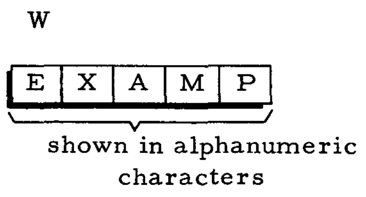
I = 3 to 12 Hollerith or alphabetic characters

Data are stored left-justified on a field of blanks.

With a precision of 10 characters, 7HEXAMPLE gives the following in the variable and constant locations:

$$\underbrace{\overset{W}{\boxed{E}}\boxed{X}\boxed{A}\boxed{M}\boxed{P}\boxed{L}\boxed{E}\boxed{\Delta}\boxed{\Delta}\boxed{\Delta}}_{\text{shown in alphanumeric characters}}$$

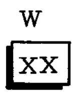
When overflow occurs, Hollerith and alphabetic data are truncated. With a precision of five characters, 7HEXAMPLE gives



Logical Data

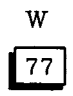
CONSTANTS

Logical constants are assigned one character of memory space as follows:



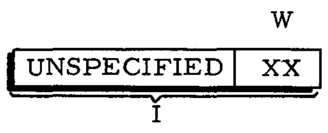
XX are two octal digits where 77 = . TRUE.
00 = . FALSE.

Example:

LOG = . TRUE. gives  in memory for the constant . TRUE. .

VARIABLES

Logical variables are stored in integer fields with only the right-hand character specifying the value, as shown below.



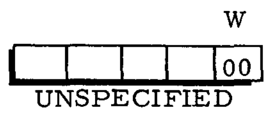
I = 3 to 12 characters

XX = two octal digits
77 (for . TRUE.) or 00 (for . FALSE.)

When a field is read into memory with a READ statement, the result will be . TRUE. only if the first nonblank character encountered is a "T".

Examples:

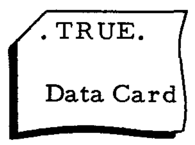
1. With "I" having a precision of five characters:

I = . FALSE. gives  in memory for the variable "I".

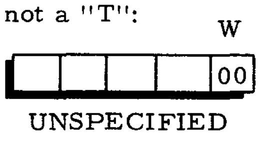
2. With a precision of six characters

READ (2, 12) LL

12 FORMAT (L6)



The above input gives the following in memory for the variable LL, because the first character of the field is not a "T":



APPENDIX D
PROCEDURES AND ROUTINES SUPPLIED WITH THE COMPILER

Table D-1 lists all library functions, subroutines, and execution routines supplied on the compiler system tape, together with the memory they require, how they are called, and the direct and indirect calls made by the routines to other routines. This information is of particular importance in determining execution time memory limits and when jobs must be chained.

Memory requirements for some routines are given by formula. To determine the number of characters required for each such routine, substitute the floating-point mantissa precision specified on the *JOBID card for the variable, mant., in the appropriate formula. The memory requirements for default precision are also listed for each variable formula.

The routines of the standard execution package — I/O modules and floating- and fixed-point packages — are described in this appendix. Described in detail also are the library functions SIN, COS, EXP, ALOG, ALOG10, ATAN, ATAN2, SQRT, and TANH.

Table D-1. Procedures and Execution Routines on the Compiler System Tape

Routine Name and Section Reference		Function of Routine and How Called	Calls by Routine		Octal Locations of Memory Required	
			Direct	Indirect	3-Character	4-Character
ABS	VI	Programmer calls ABS to take absolute value of real datum.	--	--	71	106
ACADGN	D	Execution package error routine called by system when an error occurs in a library function. Issues appropriate diagnostic.	--	--	--	--
ACBCCH	D	This chain-calling routine is brought in by the system when a job is divided into chains. Presence of CALL CHAIN statements causes the system to call ACBCCH.	--	--	73	110
ACBFIX	D	An execution routine brought in by the system when real-to-integer conversion is required, i.e., I = A. A programmer call to IFIX will also bring in ACBFIX.	--	--	472	611
ACBFLO	D	An execution routine brought in by the system when integer-to-real conversion is required, i.e., A = I. A programmer call to FLOAT will also bring in ACBFLO.	--	--	413	516
ACBFPH	D	Floating-point execution package for users with multiply/divide hardware, furnished with all central processors except Type 201. Brought in by the system when a floating-point arithmetic expression is encountered.	--	--	1715	--
ACBFPP	D	Calling routine for either ACBFPH or ACBFPS.	ACBFPH or ACBFPS	--	--	--
ACBFPR	D	Floating-point relational routine of execution package. Brought in by system whenever floating-point values in an IF or an assignment statement are used with .LT., .LE., .GT., .GE., .NE., or .EQ.	ACBFPP	ACBFPH or ACBFPS	263	--
ACBFPS	D	Floating-point execution package for users with multiply/divide software. Brought in by the system when a floating-point arithmetic expression is encountered.	--	--	2455 + 6 mant.	--
ACBFXP	D	Fixed-point execution package using simulated hardware. Brought in by system whenever fixed-point values are multiplied or divided and whenever a subscript containing a variable is encountered.	--	--	1230	--

Table D-1 (cont). Procedures and Execution Routines on the Compiler System Tape

Routine Name and Section Reference	Function of Routine and How Called	Calls by Routine		Octal Locations of Memory Required	
		Direct	Indirect	3-Character	4-Character
ACBFXR D	Fixed-point relational routine. Brought in by system whenever fixed-point values in an IF or assignment statement are used with .LT., .LE., .GT., .GE., .NE., or .EQ..	ACBFXP	--	252	--
ACBIIE D	An execution package routine brought in by system whenever integer-to-integer exponentiation is encountered (I**J).	ACBFXP	--	620	756
ACBMEM D	An execution memory dump routine brought in whenever a call to MDUMP, PDUMP, or DUMP is encountered.	--	--	1465	2006
ACBOIO and its modules:	Execution internal I/O calling routine brought in if any I/O is used. The routine calls whatever modules of internal I/O are required.	as required	--	3254	--
BACKSP D	I/O module brought in whenever BACKSPACE statements are used.	--	--	521	--
BCDCON D	I/O module brought in whenever a formatted READ or formatted WRITE statement is used.	as required	--	3135	--
BINARY D	I/O module brought in whenever an unformatted READ or unformatted WRITE statement is used.	--	--	1562	--
EFGCNV D	I/O module brought in whenever a real conversion is used (E, F, or G conversion code).	--	--	3140	--
ENDFIL D	I/O module brought in whenever an END FILE statement is used.	--	--	236	--
EOFPAR D	I/O module brought in whenever a call to EOF or PARITY is encountered.	--	--	177	--
INTCON D	I/O module brought in whenever integer conversion is used (I conversion code).	--	--	1063	--
IODIAG D	I/O diagnostic module brought in if all I/O modules are not loaded. If an I/O module not in memory is referenced, IODIAG causes an appropriate diagnostic.	--	--	16	--
LOGOCT D	I/O module brought in whenever a logical or octal conversion is used (L or O conversion codes).	--	--	1066	--
VFORMT D	I/O module brought in whenever arrays are used for formatting.	--	--	354	--
ACBRIE D	An execution routine brought in by the system whenever real-to-integer exponentiation is encountered (A**I).	ACBFIX	--	777	1162
ACBRRE D	An execution routine brought in by the system whenever real-to-real exponentiation is encountered (A**B).	ALOG, EXP, ACBFPP	ACBFPH or ACBFPS	325	400
AINT VI	A truncation library function called by the programmer.	--	--	373	473
ALOG VI	Library function for natural logarithm, called by programmer.	ACBFPP	ACBFPH or ACBFPS	1330 + 26 mant.	1562 + 26 mant.
ALOG10 VI	Library function for base-10 logarithm, called by programmer.	ALOG, ACBFPP	ACBFPH or ACBFPS	71 + 1 mant.	105
AMAX0 VI	Library function to select largest value, called by programmer.	ACBFXR, ACBFLO	ACBFXP, IABS	222	271
AMAX1 VI	Library function to select largest value, called by programmer.	ACBFPR	ACBFPP	220	266
AMIN0 VI	Library function to select smallest value, called by programmer.	ACBFXR, ACBFLO	ACBFXP, IABS	222	271
AMIN1 VI	Library function to select smallest value, called by programmer.	ACBFPR	ACBFPP	220	266
AMOD VI	Library function for remaindering, called by programmer.	AINT, ACBFPP	ACBFPH or ACBFPS	237	302
ATAN VI, D	Library function for obtaining arctangent in two quadrants, called by programmer.	ACBFPP	ACBFPH or ACBFPS	2365 + 65 mant.	2702 + 65 mant.
ATAN2 VI, D	Library function for obtaining arctangent in four quadrants, called by programmer.	ATAN, ACBFPP	ACBFPH or ACBFPS	337 + 4 mant.	416 + 4 mant.
COS VI, D	Library function for cosine, called by programmer.	SIN, ACBFPP	ACBFPH or ACBFPS	114 + 2 mant.	140 + 2 mant.
DIM VI	Library function for positive difference, called by programmer.	ACBFPP	ACBFPH or ACBFPS	120	143
DUMP VI	Special subroutine to call memory dump routine, ACBMEM.	ACBMEM	--	143	175
DVCHK VI	Special subroutine to check for illegal division.	--	--	64	100
EOF VI	Special subroutine to check for end of file.	EOFPAR	--	64	101
EOT VI	Special subroutine to check for end of tape.	--	--	63	100

Table D-1 (cont). Procedures and Execution Routines on the Compiler System Tape

Routine Name and Section Reference		Function of Routine and How Called	Calls by Routine		Octal Locations of Memory Required	
			Direct	Indirect	3-Character	4-Character
EXP	VI, D*	Exponential library function, called by programmer.	ACBFPP	ACBFPH or ACBFPS	740 + 22 mant.	1100 + 22 mant.
FLOAT	VI	Library function that can be specifically called by the programmer or is called by the system for assigning an integer value to a real variable (A=I).	ACBFLO	IABS	57	73
IABS	VI	Library function for obtaining absolute value, called by programmer.	--	--	125	153
IAND	VI	Library function for logical conjunction, called by programmer.	--	--	56	72
ICOMPLE	VI	Library function for logical complement, called by programmer.	--	--	67	102
IDIM	VI	Library function for obtaining positive difference, called by programmer.	--	--	116	142
IEXCLR	VI	Library function for exclusive OR, called by programmer.	--	--	56	72
IFIX	VI	Library function that can be specifically called by the programmer or is called by the system for assigning a real value to an integer variable (I=A).	ACBFIX	--	57	73
INT	VI	Library function for truncation, called by programmer.	ACBFIX	--	57	73
IOR	VI	Library function for inclusive OR, called by programmer.	--	--	103	125
IO4CHI	D	Execution, 4-character interface routine, called by system whenever interface with ACBOIO is required.	ACBOIO	as required	--	671 to 2634 dependent on the type of formatting
ISIGN	VI	Library function for transfer of sign, called by programmer.	--	--	247	323
MAX0	VI	Library function to select largest value, called by programmer.	ACBFXR	ACBFXP	215	263
MAX1	VI	Library function to select largest value, called by programmer.	ACBFIX, ACBFPR	ACBFPP, ACBFPH or ACBFPS	223	272
MDUMP	VI	Special subroutine to call memory dump routine.	ACBMEM	--	152	206
MIN0	VI	Library function to select smallest value, called by programmer.	ACBFXR	ACBFXP	215	263
MIN1	VI	Library function to select smallest value, called by programmer.	ACBFPR, ACBFIX	ACBFPP	223	272
MOD	VI	Library function for remaindering, called by programmer.	ACBFXP	--	164	217
OVERFL	VI	Special subroutine to test for overflow.	--	--	64	100
PARITY	VI	Special subroutine to test for correct parity.	EOFPAR	--	53	66
PDUMP	VI	Special subroutine to call memory dump routine.	ACBMEM	--	140	171
PP4CHI	D	Execution, 4-character interface routine, called by system whenever interface with ACBFPP is required.	ACBFPP	ACBFPH or ACBFPS	130	--
PR4CHI	D	Execution, 4-character interface routine, called by system whenever interface with ACBFPR is required.	ACBFPR	ACBFPP	130	--
REREAD	VI	Special subroutine to reread data.	--	--	122	147
SIGN	VI	Library function for transfer of sign, called by programmer.	--	--	134	163
SIN	VI, D	Library function for sine, called by programmer.	ACBFPP	ACBFPH or ACBFPS	1205 + 17 mant.	1441 + 17 mant.
SLITE	VI	Special subroutine for sense light test.	--	--	145	175
SLITET	VI	Special subroutine for sense light test.	--	--	216	263
SSWTCH	VI	Special subroutine for SENSE switch test.	--	--	154	207
SQRT	VI, D	Library function to compute square root, called by programmer.	ACBFPP	ACBFPH or ACBFPS	520 + 11 mant.	650 + 11 mant.
TANH	VI, D	Library function to compute hyperbolic tangent, called by programmer.	ACBFPP	ACBFPH or ACBFPS	326 + 3 mant.	405 + 3 mant.
XP4CHI	D	Execution, 4-character interface routine, called by system whenever interface with ACBFXP is required.	ACBFXP	--	115	--
XR4CHI	D	Execution, 4-character interface routine, called by system whenever interface with ACBFXR is required.	ACBFXR	ACBFXP	140	--

LIBRARY FUNCTION ERRORS AT EXECUTION TIME

During execution of a job under the conditions described in the table, the routines listed in Table D-2 will set a value in a communication cell indicating a library error condition and the routine in which the error occurred. At job termination, compiler segment ACADGN is always brought in to test the value of this cell. If a library error occurred, the value found by ACADGN will cause the segment to print out an execution-time error message indicating the routine involved and the nature of the error. A list of library error messages is given in Appendix G.

Table D-2. Library Error Conditions

Routine	Condition	Reason
AMOD MOD SIGN ISIGN	$Arg_2 = 0$	An attempt to divide by zero.
SIN COS	$\lvert Arg \rvert \geq 20\pi$	An attempt to perform calculations outside the stated limit for the library function causes undefined results.
EXP	$\lvert Arg \rvert \geq 230$	
SQRT	$Arg < 0$	
ALOG ALOG10	$Arg \leq 0$	
ACBRRE	Base 0 and Exponent $\neq 0$	An attempt to raise a negative base to a real exponent causes undefined results.

FLOATING-POINT PACKAGES

There are two floating-point packages supplied with the compiler. The package selected to perform floating-point operations depends upon the equipment configuration of the Series 200 computer at a given installation.

When an installation has a Series 200 computer with multiply/divide hardware, the Console Call card ACADRV must indicate the presence of this hardware by a D in column 14. This hardware option of the Console Call card is described in Section IX. Presence of a D in column 14 of the Console Call card triggers a call to floating-point routine ACBFPH whenever an arithmetic expression involving real addition, subtraction, multiplication, or division is encountered. ACBFPH uses the Series 200 hardware to perform multiplication and division, while simulating floating-point hardware for other arithmetic operations.

Timing of ACBFPH is:

Addition or Subtraction - $T_{AS} = 2(492 + 19.5P + 49L)$

Multiplication	$- T_M = 2(299 + 20P + 5P^2)$
Division	$- T_D = 2(333 + 22P + 7P^2)$

Where: P = characters of mantissa precision,
L = number of leading zeros in the result.

When an installation does not have multiply/divide hardware, column 14 of the Console Call card is left blank. Absence of the option causes floating-point routine ACBFPS to be brought in whenever an arithmetic expression involving real addition, subtraction, multiplication, or division is encountered. ACBFPS simulates floating-point hardware to perform all arithmetic operations.

Timing of ACBFPS is:

Addition or Subtraction	$- T_{AS} = 2(492 + 19.5P + 49L)$
Multiplication	$- T_M = 2(400 + 223P + 10P^2)$
Division	$- T_D = 2(558 + 271P + 22P^2)$

Where: P = characters of mantissa precision,
L = number of leading zeros in the result.

Error conditions cause the following results:

1. An attempt to divide by zero yields a result of all 9's in the accumulator (approximation to infinity) and causes a switch to be set for DVCHK.
2. Exponential overflow stores a resulting value of all 9's and causes a switch to be set for OVERFL.
3. Exponential underflow stores a resulting value of zero. No indicator is set.

OBJECT I/O MODULES

Object I/O coding in Fortran D is modularized so that only those routines required for the object program are loaded, thus saving object memory space. Modules are loaded as they are needed on a chain basis.

When any I/O statement is included in the program, ACBOIO is brought in. ACBOIO contains the main logic for object I/O, including calls to the driver and rewind capability, and is the calling routine for all other object I/O modules.

ACBOIO makes direct calls to EOFPAR, BINARY, BCDCON, BACKSP, and ENDFIL. BCDCON, the I/O module brought in whenever a formatted READ or WRITE statement is encountered, can in turn call VFORMT, EFGCON, INTCON, or LOGOCT. If a required I/O module is not loaded, the error diagnostic routine IODIAG is called in. Sizes for each of the various I/O modules are given in Table D-1 under ACBOIO.

There is a restriction in the use of variable formats. In a chain when variable formats are used, each conversion code in the variable format must also appear in a FORMAT statement in the same chain. If desired, the FORMAT statement may be a dummy statement.

LIBRARY FUNCTIONS

The library functions described here are:

Title	Type of Function
SIN	Sine
COS	Cosine
EXP	Exponential
ALOG	Natural Logarithm
ALOG10	Common Logarithm
ATAN	Arctangent (arctan (Arg))
ATAN2	Arctangent (arctan (Arg_1 / Arg_2))
SQRT	Square Root
TANH	Hyperbolic Tangent

The arguments for each of these functions must be in floating decimal format. The length of the mantissa, K, is variable and may range from 2 to 20 digits. Evaluation of each function, followed by a flowchart showing linkage, is presented in this Appendix.

SIN

PURPOSE: To evaluate, in floating decimal, sine x for an argument of the form:

$$x = M \cdot 10^P \text{ radians}$$

METHOD: The routine normalizes the argument until:

$$-1/4\pi \leq x \leq 1/4\pi$$

and evaluates the series:

$$Z = \sum_{i=1}^{n} C_i \, y^{2i-1}$$

where: $C_i = \dfrac{(-1)^{i-1} \, (2\pi)^{2i-1}}{(2i-1)!}$

and $y = x$, $\sin x = Z$ if $|x| \leq 1/12\pi$

or $y = 1/3\,x$, $\sin x = 3Z-4Z^3$ if $1/12\pi \leq |x| \leq 1/4\pi$

The number of terms, n, in the series evaluation, depends on the precision, K, which is specified as follows:

K	n
2, 3, 4	3
5, 6, 7	4
8, 9	5
10, 11, 12	6
13, 14, 15	7
16, 17, 18	8
19, 20	9

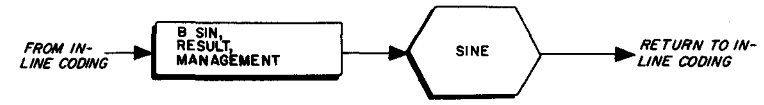
ACCURACY: $|\epsilon_r| < 7 \cdot 10^{-K}$

where: $\epsilon_r = \dfrac{f(x) - g(x)}{1 + |g(x)|}$

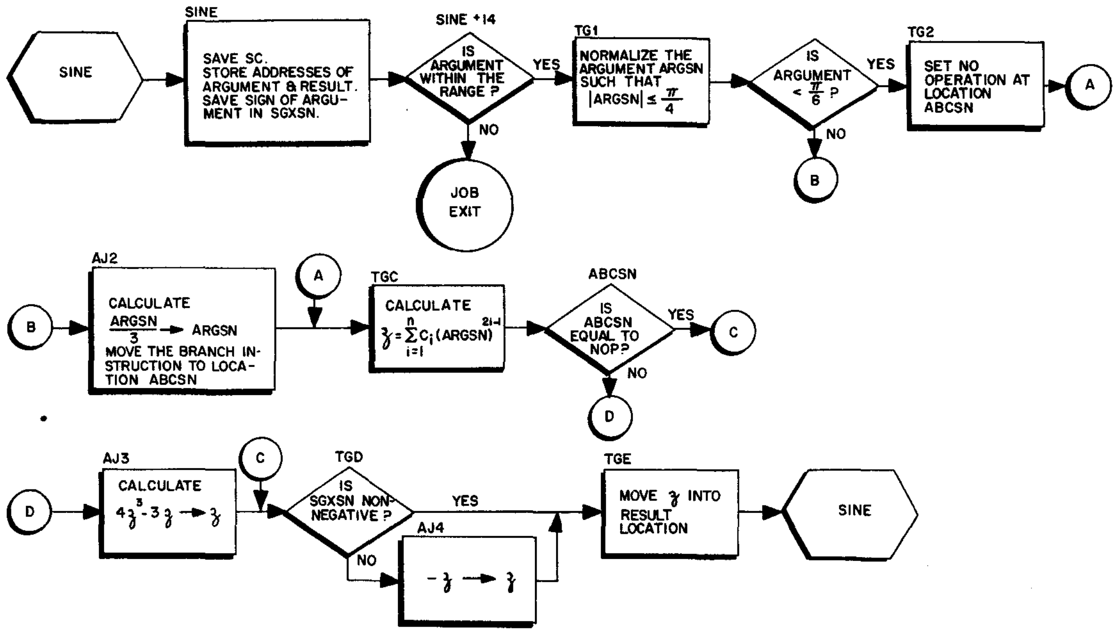
and f(x) = exact value of function
g(x) = computed value of function

LIMIT: $|x| < 20\pi$

LINKAGE:



SUBROUTINE:


SINE
SINE
SAVE SC. STORE ADDRESSES OF ARGUMENT & RESULT. SAVE SIGN OF ARGUMENT IN SGXSN.
SINE +14
IS ARGUMENT WITHIN THE RANGE ?
YES
NO
JOB EXIT
TG1
NORMALIZE THE ARGUMENT ARGSN SUCH THAT |ARGSN| ≤ π/4
IS ARGUMENT < π/6 ?
YES
NO
B
TG2
SET NO OPERATION AT LOCATION ABCSN
A
AJ2
B
CALCULATE ARGSN/3 → ARGSN MOVE THE BRANCH INSTRUCTION TO LOCATION ABCSN
A
TGC
CALCULATE ʒ = Σ C_i (ARGSN)^{2i-1}
ABCSN
IS ABCSN EQUAL TO NOP?
YES
C
NO
D
AJ3
D
CALCULATE 4ʒ³-3ʒ → ʒ
C
TGD
IS SGXSN NON-NEGATIVE ?
YES
NO
AJ4
-ʒ → ʒ
TGE
MOVE ʒ INTO RESULT LOCATION
SINE

COS

PURPOSE: To evaluate, in floating decimal, cosine x for an argument of the form: $x = M \cdot 10^P$ radians

METHOD: Cos x = sin $(\frac{\pi}{2} - x)$. Sin $(\frac{\pi}{2} - x)$ is evaluated using the method described in SIN.

ACCURACY: $|\epsilon_r| < 2\text{x}10^{-(K-1)}$

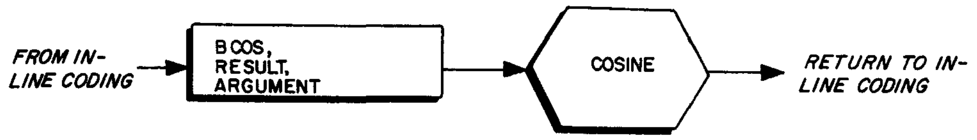
where: $\epsilon_r = \dfrac{f(x) - g(x)}{1 + |g(x)|}$

and $f(x)$ = exact value of function

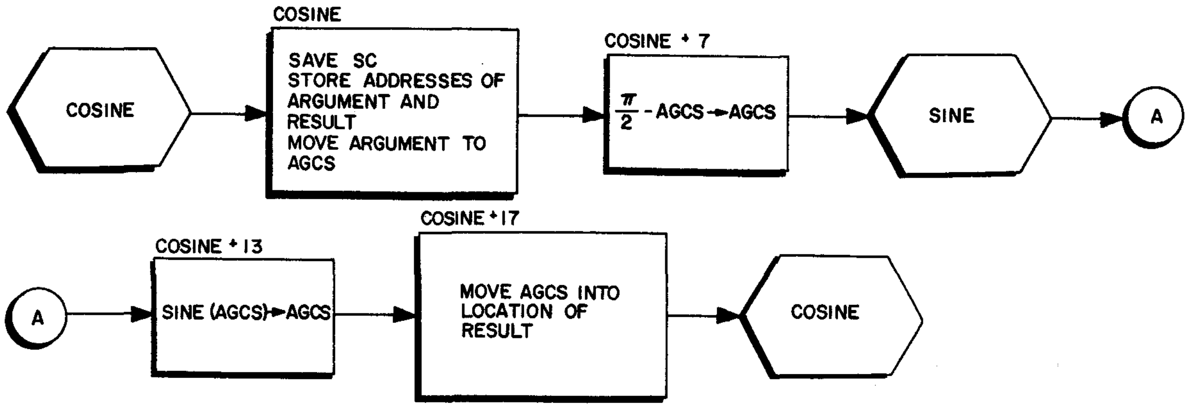
$g(x)$ = computed value of function

LIMIT: $|x| < 20\pi$

LINKAGE:



SUBROUTINE:


COSINE
COSINE
SAVE SC STORE ADDRESSES OF ARGUMENT AND RESULT MOVE ARGUMENT TO AGCS
COSINE + 7
π/2 - AGCS→AGCS
SINE
A
A
COSINE + 13
SINE (AGCS)→AGCS
COSINE + 17
MOVE AGCS INTO LOCATION OF RESULT
COSINE

EXP

PURPOSE: To evaluate, in floating decimal, e^x for an argument of the form:

$x = M \cdot 10^P$

METHOD: The routine first calculates:

$$\frac{X}{10g_e 10}$$

and separates the integer, I, and the fraction, F. It then determines:

$e^{F \log_e 10} = (e^Z)^{16}$

where: $Z = \dfrac{F \log_e 10}{16}$

by evaluating the series:

$$e^Z = \sum_{i=1}^{n} C_i Z^i$$

where: $C_i = \dfrac{1}{i!}$

Finally, $e^x = 10^I \cdot e^{F \log_e 10}$

The parameter, n, in the series evaluation, is dependent upon the precision, K, being used.

K	n
2, 3, 4	3
5	4
6, 7	5
8	6
9, 10	7
11	8
12, 13	9
14, 15	10
16, 17	11
18	12
19, 20	13

ACCURACY: $|\epsilon_r| < 2 \times 10^{-(K-2)}$

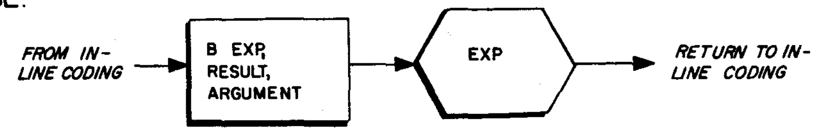
where: $\epsilon_r = \frac{f(x) - 2(x)}{1 + |g(x)|}$

and f (x) = actual value of function

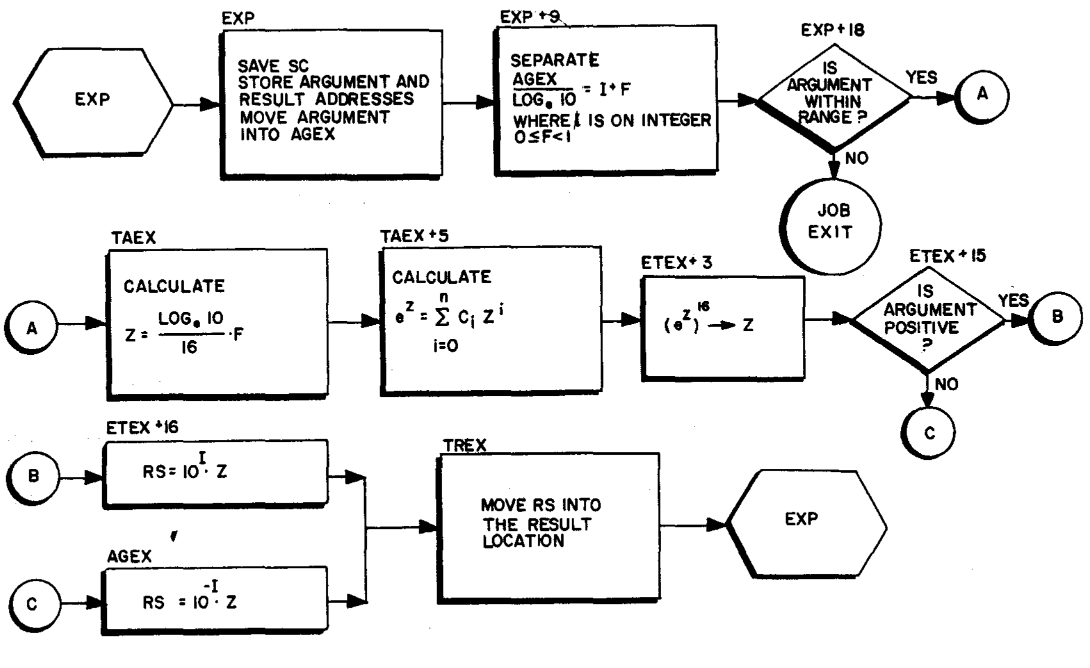
g (x) = computed value of function

LIMIT: $-230 \leq x \leq 227$

LINKAGE:



SUBROUTINE:


EXP
EXP
SAVE SC STORE ARGUMENT AND RESULT ADDRESSES MOVE ARGUMENT INTO AGEX
EXP +9
SEPARATE $\frac{AGEX}{LOG_e 10} = I + F$ WHERE I IS ON INTEGER $0 \leq F < 1$
EXP+18
IS ARGUMENT WITHIN RANGE ?
YES
NO
A
JOB EXIT
A
TAEX
CALCULATE $Z = \frac{LOG_e 10}{16} \cdot F$
TAEX+5
CALCULATE $e^Z = \sum_{i=0}^{n} C_i Z^i$
ETEX+3
$(e^Z)^{16} \to Z$
ETEX+15
IS ARGUMENT POSITIVE ?
YES
NO
B
C
B
ETEX+16
$RS = 10^I \cdot Z$
C
AGEX
$RS = 10^{-I} \cdot Z$
TREX
MOVE RS INTO THE RESULT LOCATION
EXP

SQRT

PURPOSE: To compute, in floating decimal, the square root of a positive floating decimal number X.

METHOD: Given a normalized floating-point argument of the form:

$$X = M \cdot 10^P$$

where: the mantissa M consists of digits $m_1 \ldots m_{10}$, define P' and M' as follows:

$$P' = \begin{cases} P \text{ if } P \text{ is even} \\ P{+}1 \text{ if } P \text{ is odd} \end{cases}$$

$$M' = \begin{cases} m_1 \ldots m_{10}\underbrace{0 \ldots 0}_{10 \text{ zeros}}, & \text{if P is even} \\ 0\, m_1 \ldots m_{10}\underbrace{0 \ldots 0}_{9 \text{ zeros}}, & \text{if P is odd} \end{cases}$$

$$= m_1' \, m_2' \ldots m_{20}'$$

Then $X = M' \cdot 10^{P'}$ and $\sqrt{X} = \sqrt{M'} \cdot 10^{P' 1/2}$

To find $\sqrt{M'}$, regard M' as a 20-digit integer field consisting of 10 segments, $S_1 \ldots S_{10}$, such that S_i contains 2i digits. To get the first digit of $\sqrt{M'}$, n_1, let $S_1 = m_1' \, m_2'$. Subtract as many consecutive odd integers from S_1 as possible without getting a negative result. The number of successful subtractions is n_1.

The amount subtracted from S_1 is $(n_1)^2$. This gives the new segment

$$S_1' = S_1 - 1 - 3 - \ldots - (2n_1 - 1) = S_1 - n_1^2 = m_1' \, m_2' - n_1^2$$

and $m_1' \, m_2' < (n_1+1)^2$.

S_2 is formed by adding the next two digits $m_3' \, m_4'$ to the right of S_1'. Hence, $S_2 = m_1' \ldots m_4' - (n_1 0)^2$.

To get the (i+1)st digit, n_{i+1}, of $\sqrt{M'}$, subtract $2(n_1 \ldots n_i 0)+1$, $2(n_1 \ldots n_i 0)+3 \ldots$ from the (i+1)st segment, S_{i+1} until one more subtraction would give a negative result. The number of successful subtractions is n_{i+1} and $2(n_1 \ldots n_i 0)(n_{i+1}) + (n_{i+1})^2$ is subtracted. Hence the new segment is:

$$\begin{aligned} S'_{i+1} &= S_{i+1} - (2(n_1 \ldots n_i 0)+1) - \ldots - (2(n_1 \ldots n_i 0) + (2n_{i+1}-1)) \\ &= S_{i+1} - 2(n_1 \ldots n_i 0)(n_{i+1}) - (n_{i+1})^2 \\ &= S_{i+1} - (n_1 \ldots n_i\, n_{i+1})^2 + (n_1 \ldots n_i 0)^2 \\ &= (m_1' \ldots m'_{2i+2}) - (n_1 \ldots n_{i+1})^2 \end{aligned}$$

and $(m_1' \ldots m'_{2i+2}) < (n_1 \ldots (n_{i+1}+1))^2$.

S_{i+2} is formed by adding $m'_{2i+3}\ m'_{2i+4}$ to the right of S'_{i+1} . Thus,

$$S_{i+2} = (m'_1 \ldots m'_{2i+4}) - (n_1 \ldots n_{i+1})^2 .$$

At each step, $(n_1 \ldots n_i)^2 \leq (m_1' \ldots m'_{2i}) < (n_1 \ldots (n_i+1))^2$ and therefore:

$$N = n_1 \ldots n_{10} \approx \sqrt{M'} .$$

ACCURACY: $|\epsilon_r| < 2 \times 10^{-K}$

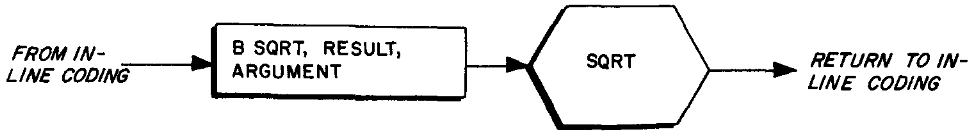
where: $\epsilon_r = \dfrac{f(x) - g(x)}{1 + |g(x)|}$

and $f(x)$ = exact value of function

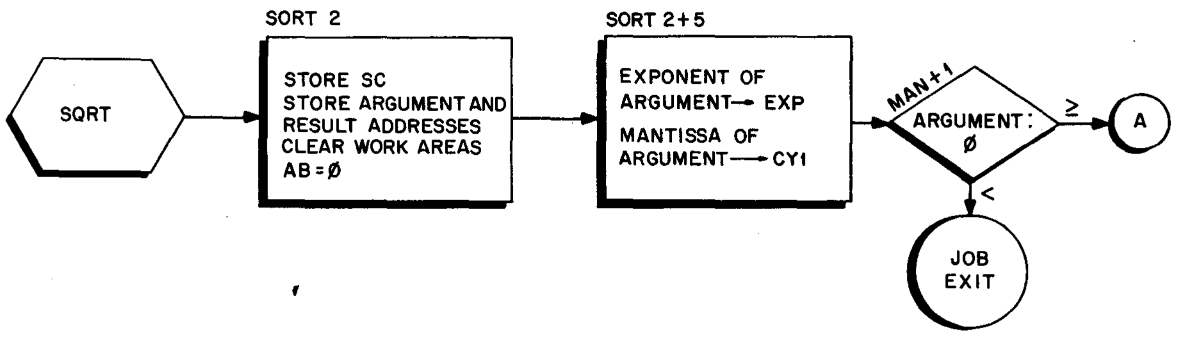
$g(x)$ = computed value of function

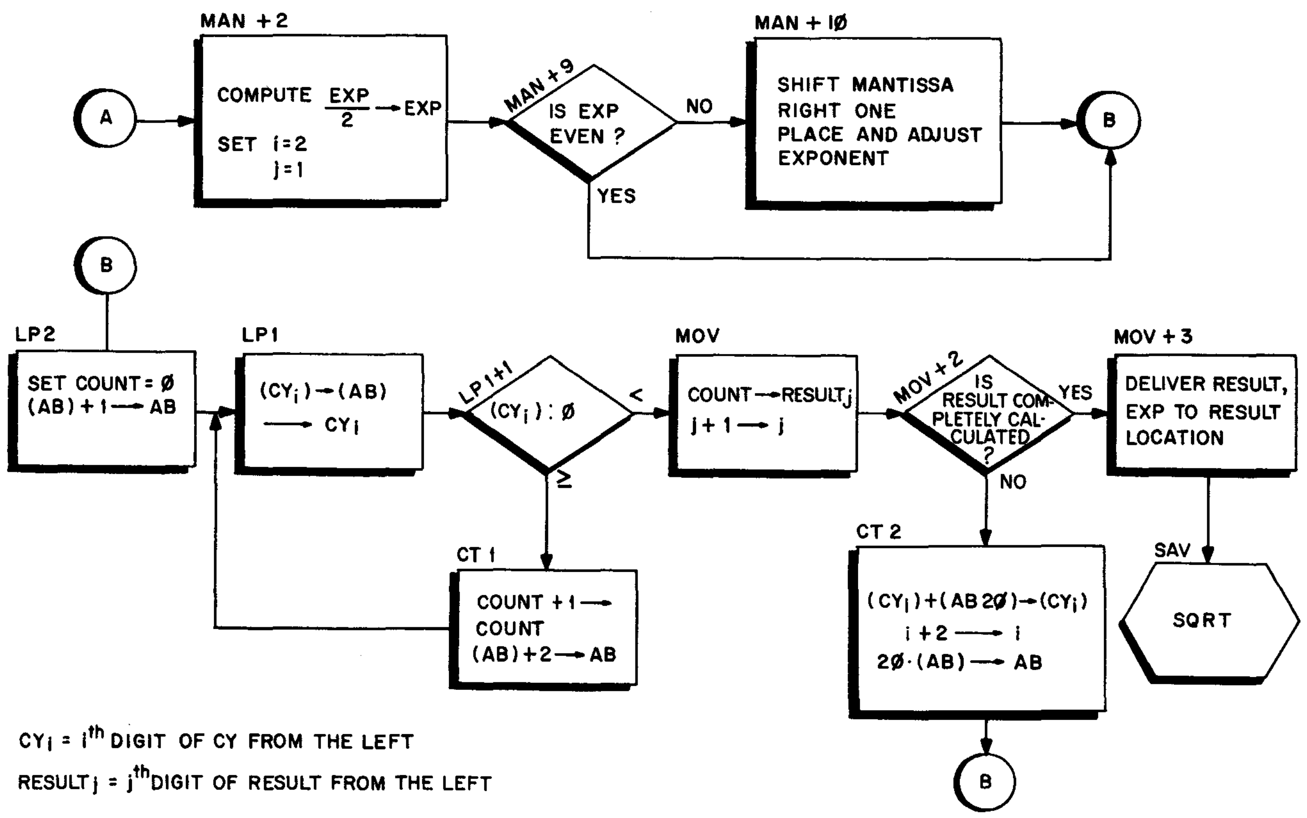
LIMIT: $X < 0$.

LINKAGE:



SUBROUTINE:


SQRT
SQRT 2
STORE SC
STORE ARGUMENT AND RESULT ADDRESSES
CLEAR WORK AREAS
AB=Ø
SQRT 2+5
EXPONENT OF ARGUMENT→ EXP
MANTISSA OF ARGUMENT→ CY1
MAN+1
ARGUMENT: Ø
≥
A
<
JOB EXIT



TANH

PURPOSE: To evaluate, in floating decimal, tanh x for an argument of the form:

$$x = M \cdot 10^P$$

METHOD: If $x < -23$, then $\tanh x = -1.0$

If $x > +23$, then $\tanh x = +1.0$

For all other values of x, the routine calculates:

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

e^{2x} is calculated using the method described in EXP.

ACCURACY: $|\epsilon_r| < 3 \times 10^{-(k-2)}$

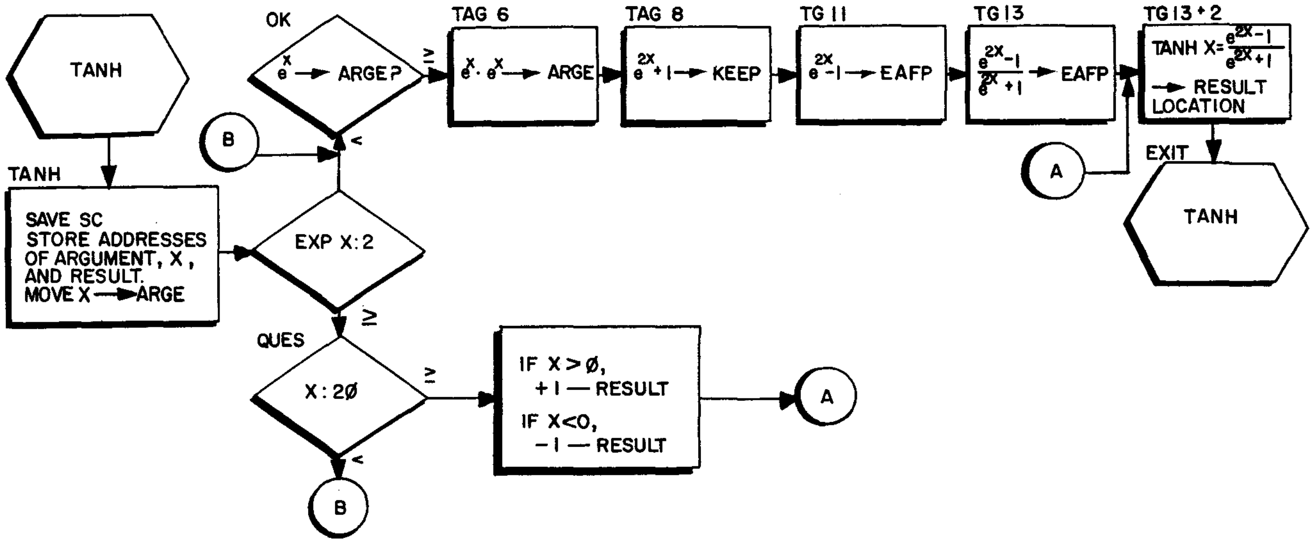
where: $\epsilon_r = \frac{f(x) - g(x)}{1 + |g(x)|}$

and f(x) = exact value of function

g(x) = computed value of function

LIMIT: None

SUBROUTINE:



ALOG

PURPOSE: To evaluate, in floating decimal, $\log_e x$ for an argument of the form:

$X = M \cdot 10P$

METHOD: The routine calculates $\log_e x$ by evaluating the series:

$$\log_e x = P \log_e 10 + 2 \sum_{i=1}^{n} C_i \; y^{2i-1}$$

where: $x = M \cdot 10^P$, $0.1 \leq M < 1$ and $y = \frac{M-1}{M+1}$

The number of terms in the series, n, depends on the precision, K, to be used.

K	n
2, 3, 4, 5	2
6, 7	3
8	4
9, 10	5
11	6
12, 13	7
14	8
15, 16	9
17	10
18, 19, 20	11

ACCURACY: $|\epsilon_r| < 1 \times 10^{-(K-1)}$

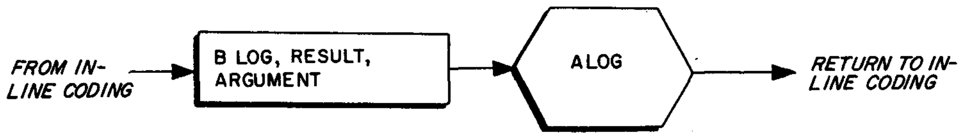
where: $\epsilon_r = \frac{f(x) - g(x)}{1 + |g(x)|}$

and f (x) = exact value of function

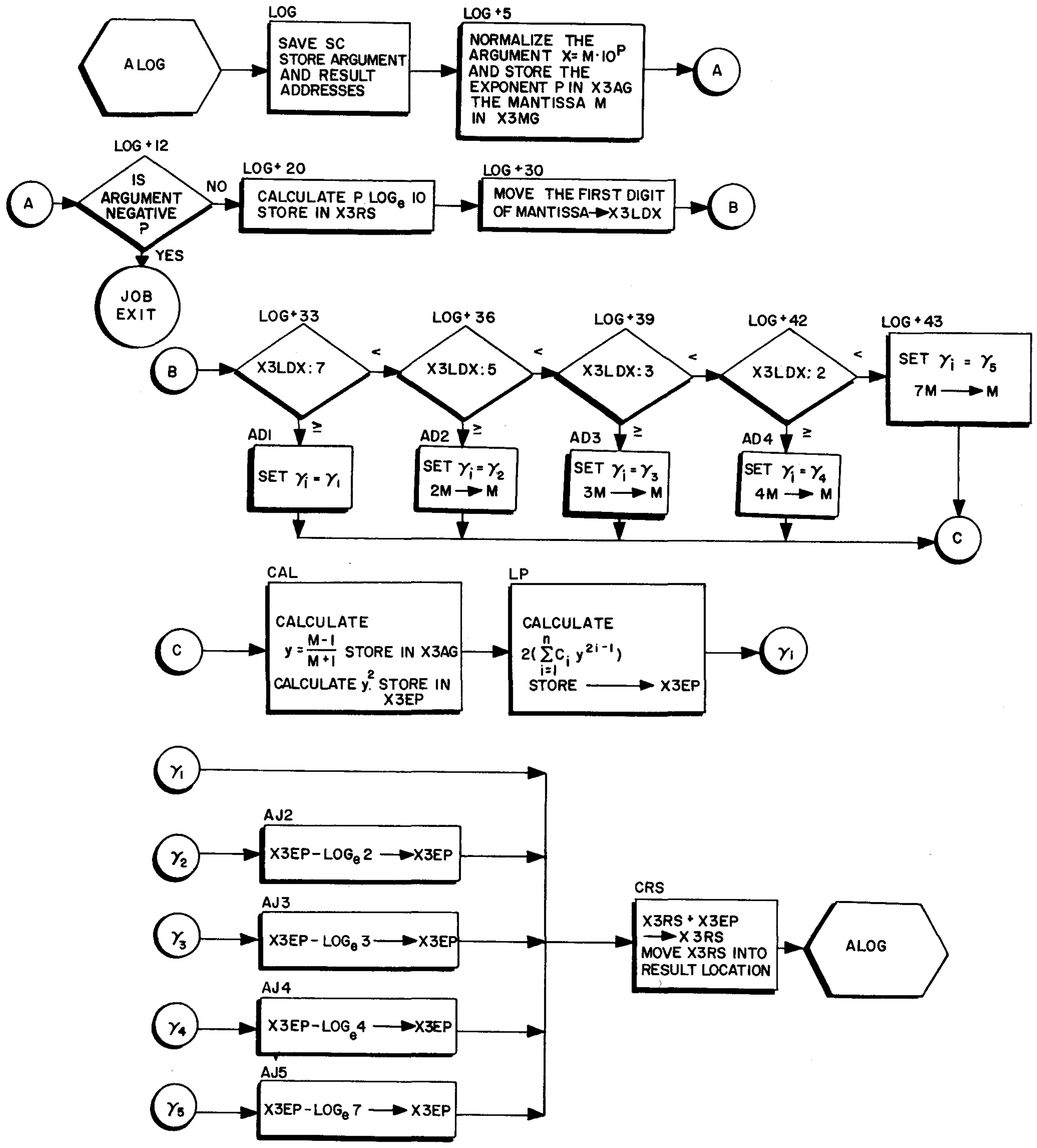
g (x) = computed value of function

LIMIT: $X > 0$

LINKAGE:



SUBROUTINE:



ALOG
LOG
SAVE SC STORE ARGUMENT AND RESULT ADDRESSES
LOG+5
NORMALIZE THE ARGUMENT X= M·10^P AND STORE THE EXPONENT P IN X3AG THE MANTISSA M IN X3MG
A
LOG+12
IS ARGUMENT NEGATIVE ?
NO
YES
JOB EXIT
LOG+20
CALCULATE P LOGe 10 STORE IN X3RS
LOG+30
MOVE THE FIRST DIGIT OF MANTISSA → X3LDX
B
LOG+33
X3LDX:7
LOG+36
X3LDX:5
LOG+39
X3LDX:3
LOG+42
X3LDX:2
LOG+43
SET γi = γ5 7M → M
<
≥
AD1
SET γi = γ1
AD2
SET γi = γ2 2M → M
AD3
SET γi = γ3 3M → M
AD4
SET γi = γ4 4M → M
C
CAL
CALCULATE y = (M-1)/(M+1) STORE IN X3AG CALCULATE y² STORE IN X3EP
LP
CALCULATE 2(Σ_{i=1}^{n} Ci y^{2i-1}) STORE → X3EP
γi
γ1
γ2
γ3
γ4
γ5
AJ2
X3EP-LOGe2 → X3EP
AJ3
X3EP-LOGe3 → X3EP
AJ4
X3EP-LOGe4 → X3EP
AJ5
X3EP-LOGe7 → X3EP
CRS
X3RS + X3EP → X3RS MOVE X3RS INTO RESULT LOCATION
ALOG

ALOG10

PURPOSE: To evaluate, in floating decimal, $\log_{10}x$ for an argument of the form:

$X = M \cdot 10P$

METHOD: $\log_{10}x = \log_e x \cdot \log_{10}e$. The routine calculates $\log_e x$ according to the method described in ALOG. $\log_{10}e$ is stored as a constant and multiplied by $\log_e x$ to obtain the desired result.

ACCURACY: $|\epsilon_r| < 1 \times 10^{-(K-1)}$

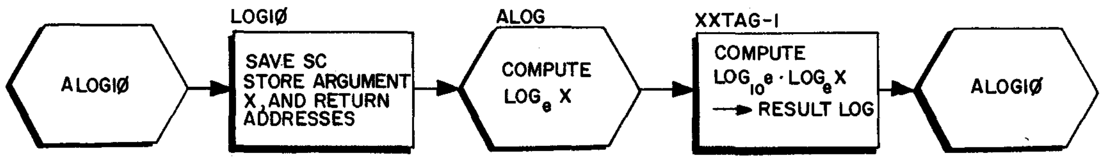
where: $\epsilon_r = \dfrac{f(x) - g(x)}{1 + |g(x)|}$

and $f(x)$ = exact value of function

$g(x)$ = computed value of function

LIMIT: $X > 0$

SUBROUTINE:


ALOGIØ
LOGIØ
SAVE SC STORE ARGUMENT X, AND RETURN ADDRESSES
ALOG
COMPUTE LOG$_e$ X
XXTAG-1
COMPUTE LOG$_{10}$e · LOG$_e$X → RESULT LOG
ALOGIØ

ATAN

PURPOSE: To evaluate, in floating decimal, $\tan^{-1}x$ for an argument of the form $x = M \cdot 10P$ obtaining a positive angle in the first quadrant or a negative angle in the fourth quadrant measured in radians.

METHOD: The routine evaluates the series:

$$\tan^{-1}y = \sum_{i=1}^{n} C_i \, y^{2i-1}$$

where: $C_i = \dfrac{(-1)^{i-1}}{2i-1}$

and $y = \dfrac{x-a}{1+ax}$

$$\tan^{-1}x = \tan^{-1}y + \tan^{-1}a$$

The constant, a, is selected from the following table.

Range of x	Value of a	Range of x	Value of a
$11.0 < x \le 10^{98}$	11.0	$.77 < x \le .92$	.77
$5.5 < x \le 11.0$	5.5	$.64 < x \le .77$	.64
$3.7 < x \le 5.5$	3.7	$.52 < x \le .64$	.52
$2.8 < x \le 3.7$	2.8	$.41 < x \le .52$	.41
$2.2 < x \le 2.8$	2.2	$.31 < x \le .41$	.31
$1.8 < x \le 2.2$	1.8	$.22 < x \le .31$	.22
$1.5 < x \le 1.8$	1.5	$.13 < x \le .22$	.13
$1.3 < x \le 1.5$	1.3	$.09 < x \le .13$	.09
$1.1 < x \le 1.3$	1.1	$.00 < x \le .09$	.00
$.92 < x \le 1.1$	.92	------	---

The number of terms, n, in the series, depends on the specified precision, K, as follows:

K	n
2, 3	1
4, 5	2
6, 7	3
8, 9, 10	4
11, 12	5
13, 14	6
15, 16	7
17, 18	8
19, 20	9

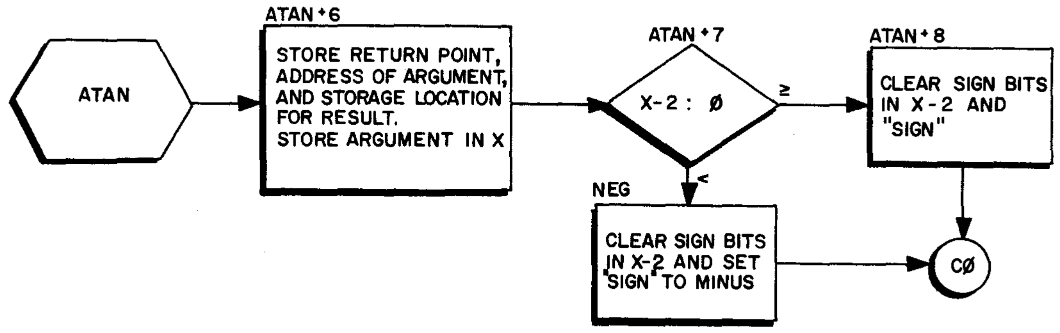
ACCURACY: $|\epsilon_r| < 2 \times 10^{-k}$

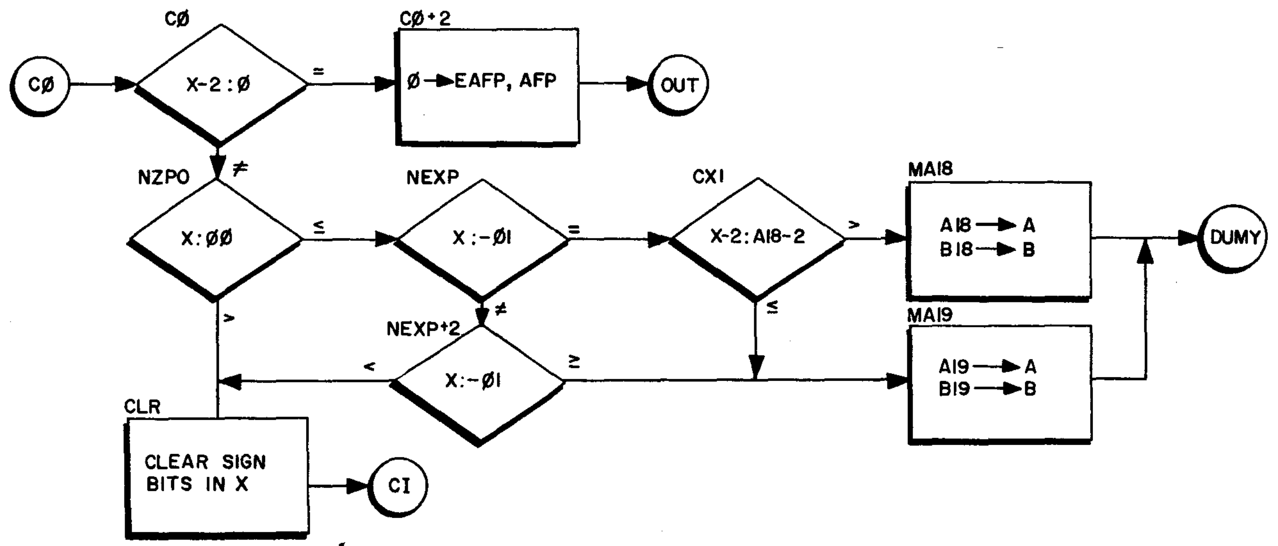
where: $\epsilon_r = \dfrac{f(x) - g(x)}{1 + |g(x)|}$

and $f(x)$ = exact value of function

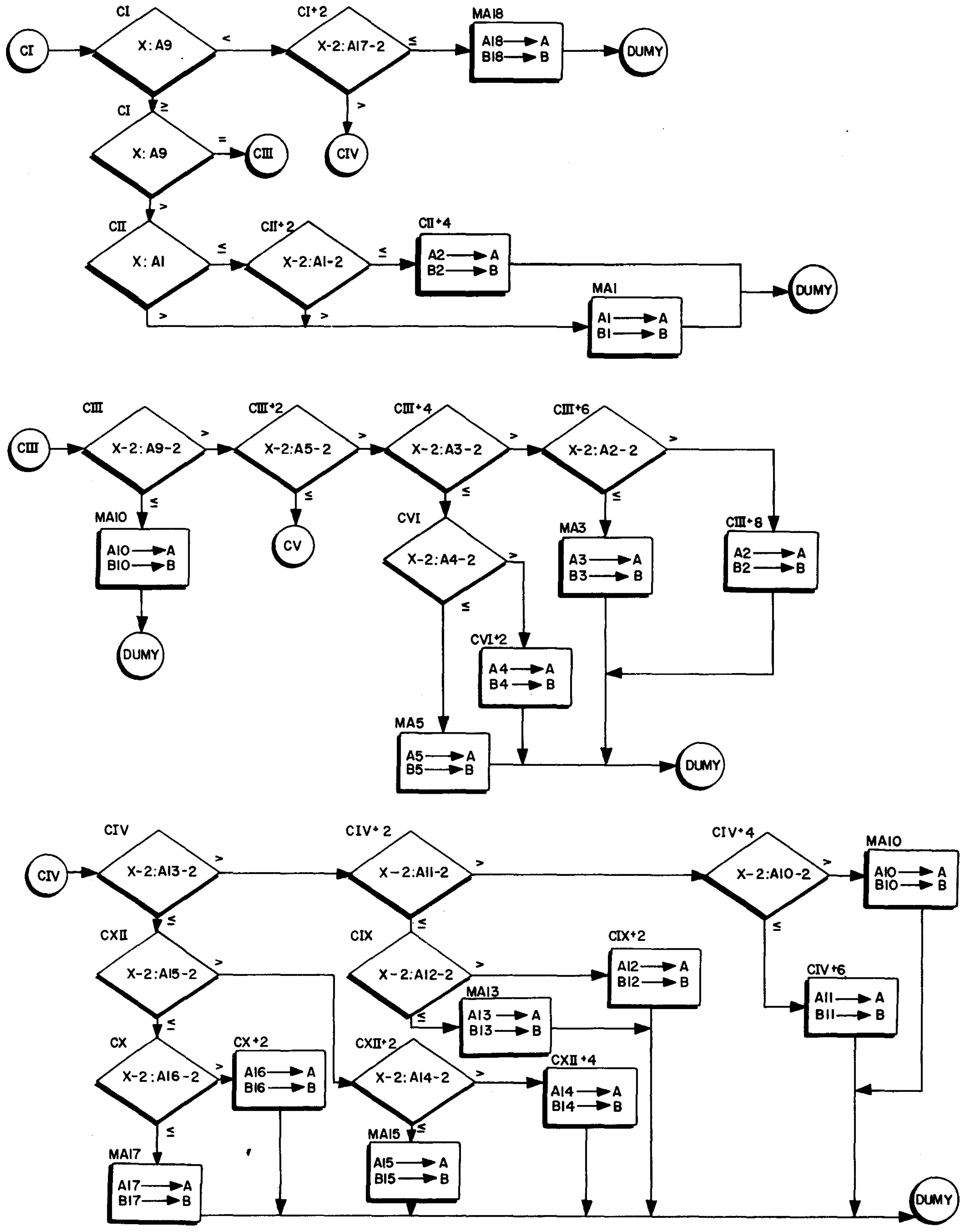
$g(x)$ = computed value of function

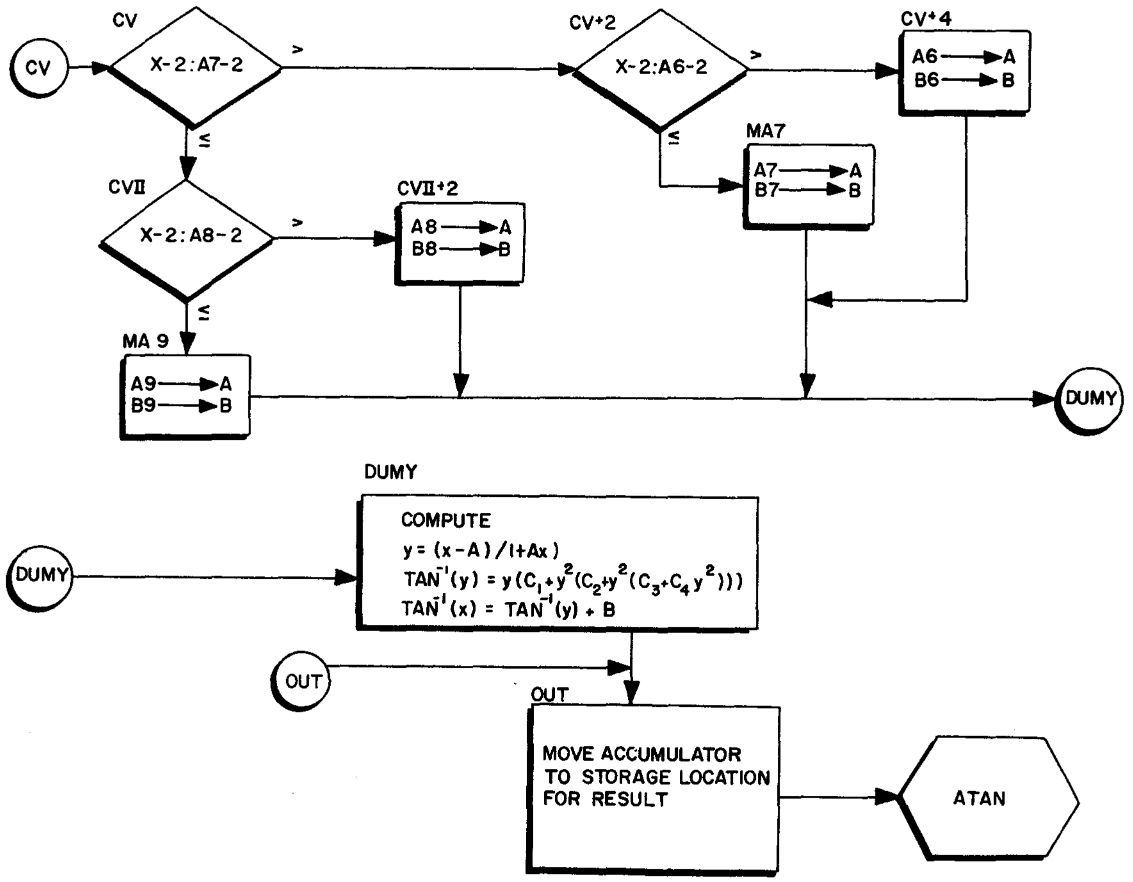
SUBROUTINE:


ATAN
ATAN+6
STORE RETURN POINT, ADDRESS OF ARGUMENT, AND STORAGE LOCATION FOR RESULT. STORE ARGUMENT IN X
ATAN+7
X-2 : Ø
≥
<
ATAN+8
CLEAR SIGN BITS IN X-2 AND "SIGN"
NEG
CLEAR SIGN BITS IN X-2 AND SET SIGN TO MINUS
CØ



NOTE: THE NUMBER THAT MODIFIES AN ADDRESS (I.E., CØ+2) IS ONLY AN AID IN FOLLOWING THE FLOW CHART. THIS NUMBER REFERS TO THE NUMBER OF CODING LINES AFTER THE TAG, NOT THE NUMBER OF CHARACTERS.


CI
X:A9
CI+2
X-2:A17-2
MA18
A18 → A
B18 → B
DUMY
X:A9
CIII
CIV
CII
X:A1
CII+2
X-2:A1-2
CII+4
A2 → A
B2 → B
MA1
A1 → A
B1 → B
CIII+2
X-2:A9-2
X-2:A5-2
CIII+4
X-2:A3-2
CIII+6
X-2:A2-2
MA10
A10 → A
B10 → B
CV
CVI
X-2:A4-2
MA3
A3 → A
B3 → B
CIII+8
CVI+2
A4 → A
B4 → B
MA5
A5 → A
B5 → B
CIV
X-2:A13-2
CIV+2
X-2:A11-2
CIV+4
X-2:A10-2
CXII
X-2:A15-2
CIX
X-2:A12-2
CIX+2
A12 → A
B12 → B
MA13
A13 → A
B13 → B
CIV+6
A11 → A
B11 → B
CX
X-2:A16-2
CX+2
A16 → A
B16 → B
CXII+2
X-2:A14-2
CXII+4
A14 → A
B14 → B
MA15
A15 → A
B15 → B
MA17
A17 → A
B17 → B



ATAN2

PURPOSE: To evaluate, in floating decimal, $\tan^{-1} A/B$ for arguments of A and B of the form:

$$A = M \cdot 10P$$

$$B = M' \cdot 10P'$$

The result is a positive angle between 0 and 2π, measured in radians.

METHOD: The routine evaluates $\tan^{-1} A/B$ by the method described in ATAN, obtaining a positive angle in the first quadrant or a negative angle in the fourth quadrant. It then examines the signs of A and B and computes an angle in the proper quadrant.

ACCURACY: $|\epsilon_r| < 5 \times 10^{-K}$

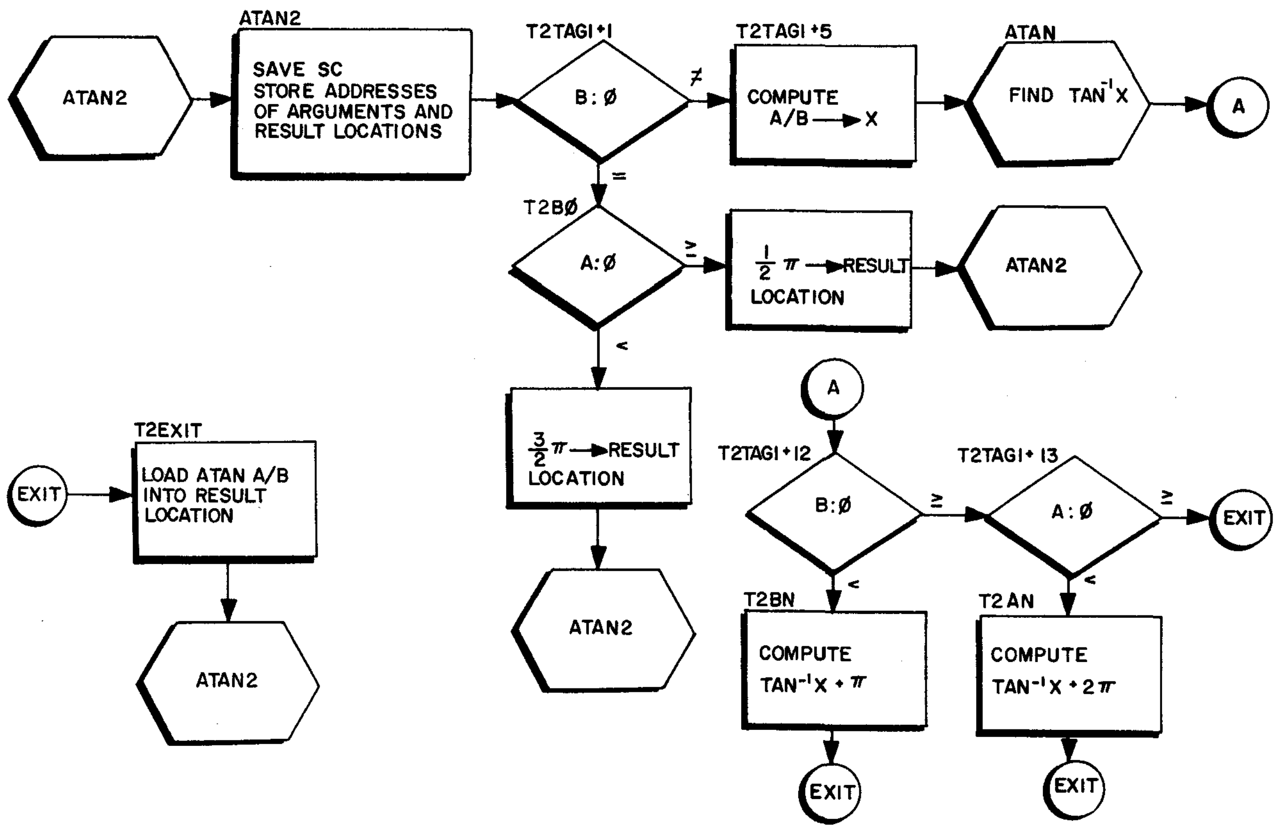
where: $\epsilon_r = \frac{f(x) - g(x)}{1 + |g(x)|}$

and f(x) = exact value of function.

g(x) = computed value of function.

LIMIT: $A/B \leq ,10^{97}$

SUBROUTINE:


ATAN2
ATAN2
SAVE SC STORE ADDRESSES OF ARGUMENTS AND RESULT LOCATIONS
T2TAGI+1
B:Ø
≠
=
T2TAGI+5
COMPUTE A/B → X
ATAN
FIND TAN⁻¹X
A
T2BØ
A:Ø
≥
<
½π → RESULT LOCATION
ATAN2
3/2π → RESULT LOCATION
ATAN2
A
T2TAGI+12
B:Ø
≥
<
T2TAGI+13
A:Ø
≥
<
EXIT
T2BN
COMPUTE TAN⁻¹X + π
EXIT
T2AN
COMPUTE TAN⁻¹X + 2π
EXIT
T2EXIT
EXIT
LOAD ATAN A/B INTO RESULT LOCATION
ATAN2

APPENDIX E
EASYCODER SYMBOLIC PROGRAM UNITS

Program units which are written in Easycoder language[1] and assembled onto a BRT can be combined into Fortran D jobs. The program units are fetched by name, as directed by *GET statements during compilation. Alternatively, binary decks of assembled code can be called using *BINARY control cards. Refer to Section VII for the formats of these directors.

The Easycoder program unit must be organized into code regions in a manner similar to compiled program units. Certain addressing conventions must be used to allow proper relocation and intercommunication between the Easycoder program unit and other program units in the Fortran job. Regionalization and code conventions are discussed below.

REGIONALIZATION

The program origin of the Easycoder program unit must be 4096_{10}. Five DSA statements at this origin define six coding region boundaries. Each region must be contiguous and consecutive in memory, and each contains specific data types to which a corresponding relocation factor applies.

Region 1 contains constants, data storage areas, and argument DSA statements. This region may be empty. All information and punctuation is reproduced when this region is relocated.

Region 2 contains the names of all other program units which are "called" by the present Easycoder program unit. Each name appears left justified in a six-character field. A word mark on the leftmost character is the only punctuation. Region 2 may be empty; it is deleted during relocation.

Region 3 is the instruction string and must contain at least one instruction. The first character in this region is the "turn-on point" to which control will be transferred when the Easycoder program unit is called. There must be a word mark on the leftmost character of each instruction; no other word marks are permitted. All address fields in this region are relocated. Calls to other program units are indicated by branches to the addresses of the left-justified names in region 2. These addresses will be replaced by the relocated turn-on points of the called subprograms when this region is processed.

[1]Refer to Models 200/1200/2200 Programmers' Reference Manual, Order No. 139, or Model 120 Programmers' Reference Manual, Order No. 141.

Region 4 contains DSA statements that are relocated in the same manner as address fields within instructions in region 3. This region may be empty.

Region 5 contains constants, data storage areas, or argument DSA statements. Thus, it serves as an auxiliary storage region which can be conveniently adjusted by initialization code (see below). This region may be empty.

Region 6 contains initialization coding. This code is executed before the Easycoder program unit is relocated. The A address of the Easycoder END card defines the turn-on point of the initialization code. Job parameters such as integer or mantissa precision, memory size, etc., can be interrogated by the initialization code and the other regions, usually region 5, adjusted accordingly. If the region dimensions are altered by such initialization, the region-defining DSA statements must also be adjusted. This region is deleted when the Easycoder program is relocated. At the conclusion of initialization, region 6 branches indirectly to location $CMNEX which is defined below.

Region-Defining DSA Statements

Location	DSA	Contents	Notes
4096 4097 4098	1	Points to the first location not in Region 1	Normally the first location in Region 2.
4099 4100 4101	2	Turn-on point	When Region 2 is empty, DSA 1 is the same as DSA 2.
4102 4103 4104	3	Points to the last opcode in Region 3	If Region 4 is empty, DSA 3 should be repeated as DSA 4.
4105 4106 4107	4	Points to the last location in Region 4	If Region 4 is empty, DSA 3 should be repeated as DSA 4.
4108 4109 4110	5	Point to the first character not in Region 5.	DSA 5 should initially point to the first location in Region 6.

Note that the above DSA statements serve only to define regions. They are deleted when the Easycoder program unit is relocated.

SYSTEM PROCESSING

The Easycoder program unit can be placed upon the object run tape by the following sequence of system events:

1. A *GET or *BINARY control card is encountered; the associated Easycoder program unit is transferred directly onto the compilation output tape (BPT).
2. When all contents of a job are transferred (or compiled) onto the BPT, the object run tape generation process begins. The BPT is rewound and the tape loader-monitor is instructed to load each program unit on the tape, one by one.
3. According to its sequence on the BPT, the Easycoder program is loaded.
4. After loading, the tape loader-monitor transfers control to the location specified by the END card of the Easycoder program unit. This location is the first instruction of initialization coding in region 6.
5. When region 6 has performed any necessary specialization of the other regions, control is passed back to the Fortran D system by executing:

   B (\$CMNEX)

6. The Easycoder regions are now processed according to content:
   a. Region 1 contains data literals and reserve areas. These contents are not modified, but are sent directly to the object run tape.
   b. Region 2 defines external references to other program units in the job. This region is not needed during execution and therefore will not be placed on the object run tape.
   c. Region 3 contains the instruction string. All address fields must be interpreted and relocated before this region is sent to the object run tape.
   d. Region 4 contains DSA statements. These are relocated in the same manner as address fields in region 3 before this region is sent to the object run tape.
   e. Region 5 is processed in the same manner as Region 1.
7. After the above processing, the system requests the tape loader-monitor to load the next program unit from the BPT. Fortran-compiled program units are processed in the same manner as Easycoder program units with the following exceptions:
   a. Region 4 and region 5 are always empty in Fortran programs.
   b. Initialization is bypassed. After loading Fortran program units, the tape loader-monitor transfers control immediately back to the Fortran D system.

Region 3 Address Interpretation

Each instruction in region 3 begins with a word-marked character and continues through to the location preceding the next word mark. Address fields within instructions are determined by an instruction length algorithm summarized by Table E-1.

Table E-1. Determination of Address Lengths in Region 3

Instruction Length (Characters)	Secondary Characteristic	No of Address Fields	Character Positions of Addresses	Example
1	. . .	0	. . .	NOP
2	. . .	0	. . .	CAM 00
3	. . .	1	123	DSA addr
4	No item mark on last character	1	234	BS addr
	Item mark on last character. Branch Instruction		234	B addr
	Item mark on last character. Not a Branch Instruction		123	DSA addr, V
5 or 6	. . .	1	234	SCR addr, V
7 or more	PDT or PCB instruction	1	234	PDT addr, V, V, V
7 or more	Not a PDT or PCB instruction	2	234 567	MCW addr, addr

Relocation of Address Fields

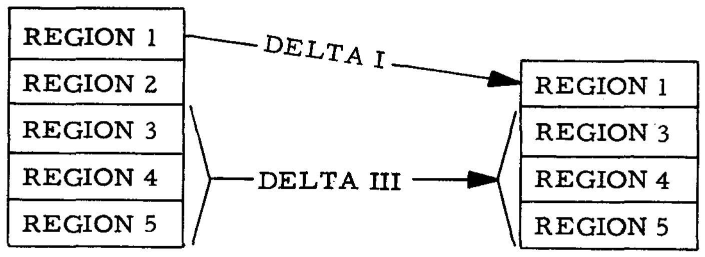
Punctuation of a direct or indirect address field indicates whether it is to be relocated and which relocation delta should be used. The item marks to control relocation can be set by initialization coding. Indexed addresses are not relocated in Easycoder routines.

Punctuation	Type of Address
Item mark on first character	Absolute address - no relocation
Item mark on middle character	Unlabeled common address
Item marks on first and middle characters	Labeled common address
No item marks on first and middle characters	Not a common address; use appropriate regional delta as given in the paragraphs following

If there are no item marks on the first and middle characters of the address and the program is an Easycoder program, the relocation delta appropriate to the region is chosen by considering the address according to the following table.

$0 \leq \text{ADDR} < 4096_{10}$	do not relocate
$4096 \leq \text{ADDR} < \text{DSA } 1$	add delta of region 1
$\text{DSA } 1 \leq \text{ADDR} < \text{DSA } 2$	replace by address of first instruction of program
$\text{DSA } 2 \leq \text{ADDR} < \text{DSA } 5$	add delta of region 3
$\text{DSA } 5 \leq \text{ADDR} < 32,767_{10}$	do not relocate

Note that the same relocation delta is used for regions 3, 4 and 5, while that for region 1 is unique. This results from the fact that region 2 is not relocated but is deleted, so that blocks are moved as shown below.



Relocation of Fortran Program Units

When there are no item marks on characters 1 and 2 and the program unit is a Fortran program unit, the following rules govern relocation:

1. The delta is chosen in the same way as for Easycoder program units when the address is either direct or indirect.
2. The delta for region 1 is always used if the address is indexed.

Note that if the op code of an instruction is 00, the op code is replaced by 40 (NOP) and the index bits of the A address are replaced by zero bits.

PROGRAMMING PROCEDURES

Calling Sequences

Any Easycoder program unit that expects to call a Fortran program unit or be called by a Fortran program unit must employ the Fortran compiler calling sequences given below. (The same calling sequences can be used when an Easycoder program unit calls another Easycoder program unit within a Fortran job.)

TYPE 1	TYPE 2	TYPE 3
B SUBROUTINE	B SUBROUTINE	® B SUBROUTINE
DSA ARG1	DSA ARG1	
.	.	
.	.	
.	.	
.	DSA ARGn	
® DSA ARGn	® DC 1Cnn	

Note that in each type there is an item mark right on the last character of the calling sequence. The only exceptions occur when the calling sequence is a B ($CCHEN) or B ($CMNEX). $CCHEN and $CMNEX are explained below under "Use of Communication."

The address, SUBROUTINE, generally refers to region 2, in which the name of the program unit is located.

If the subroutine called is a Fortran subroutine, each DSA statement must use either direct or indirect addressing. Indexed addressing must not be used. If the subroutine refers to an argument as an array, direct addressing must be used. If the subroutine called is an Easycoder program unit, any type of addressing, including indexed, can be used.

If the calling program unit is a Fortran program unit, it expects return to be made to the location immediately following the item mark.

Each subroutine should save and restore any index registers that it uses.

If the subprogram is used as a function subprogram, the first DSA statement is that of the result location.

Use of Common Storage

Since common storage is relocated, Easycoder program units that reference common storage must follow the procedures below.

1. Each address referencing common storage must have the proper item-mark bits set in it.
2. There must be Fortran source programs in the chain that name the appropriate common blocks, which in turn must have the correct size.
3. The address of the base of unlabeled common storage is 10000_8, and it runs forward in memory.
4. Allocation of labeled common storage begins with the first appearance of a label name. The last location of the first block of labeled common storage is 77777_8. Remaining locations for the first block are allocated forward in memory. Blocks of labeled common storage appear sequentially in memory, so that the last location of each additional block will be the one immediately before the first location of the block previously defined.

Use of Communication

A number of registers in the communication region are usable by the object programs.

At run-tape generation time, the following registers are usable.

$CMNEX	— contains a DSA statement of the monitor exit.
$INTEG	— contains the number of characters specified for integer size.
$RUNMD	— 03: 3-character address mode of execution. 04: 4-character address mode of execution.

$FLOAT	— contains the number of characters specified for the floating-point mantissa.
$FLTPA+22	— is the rightmost character of the floating accumulator.
$FXPTA+11	— is the rightmost character of the fixed accumulator.
$SUBPR	— contains relevant flags to the run-tape generator. This location is reset to zero before each program is loaded. If this location is loaded, it is interpreted as follows:

B and A bits both zero	—	Easycoder program unit
B and A not both zero	—	Fortran program unit
1 bit = 1	—	Save this program unit
4 bit = 1	—	Punch this program unit

At object time, the following registers are usable in addition to the above:

$FLOAT	— contains the floating-point size, including exponent.
$SUBPR	— has no meaning.
$LOGSW	— is the logical accumulator, used during logical input/output.
$FPO	— is exponential overflow indicator.
$DBZ	— is the divide-by-zero indicator.

Restrictions

All addresses must be in the three-character mode.

MORG statements (and MAT instructions that rely on correct interpretation of MORG statements) are ignored, since the program is relocated modulo 1.

Halt instructions must be avoided unless absolutely necessary. In case they are used, halts should conform to Series 200 Fortran conventions.

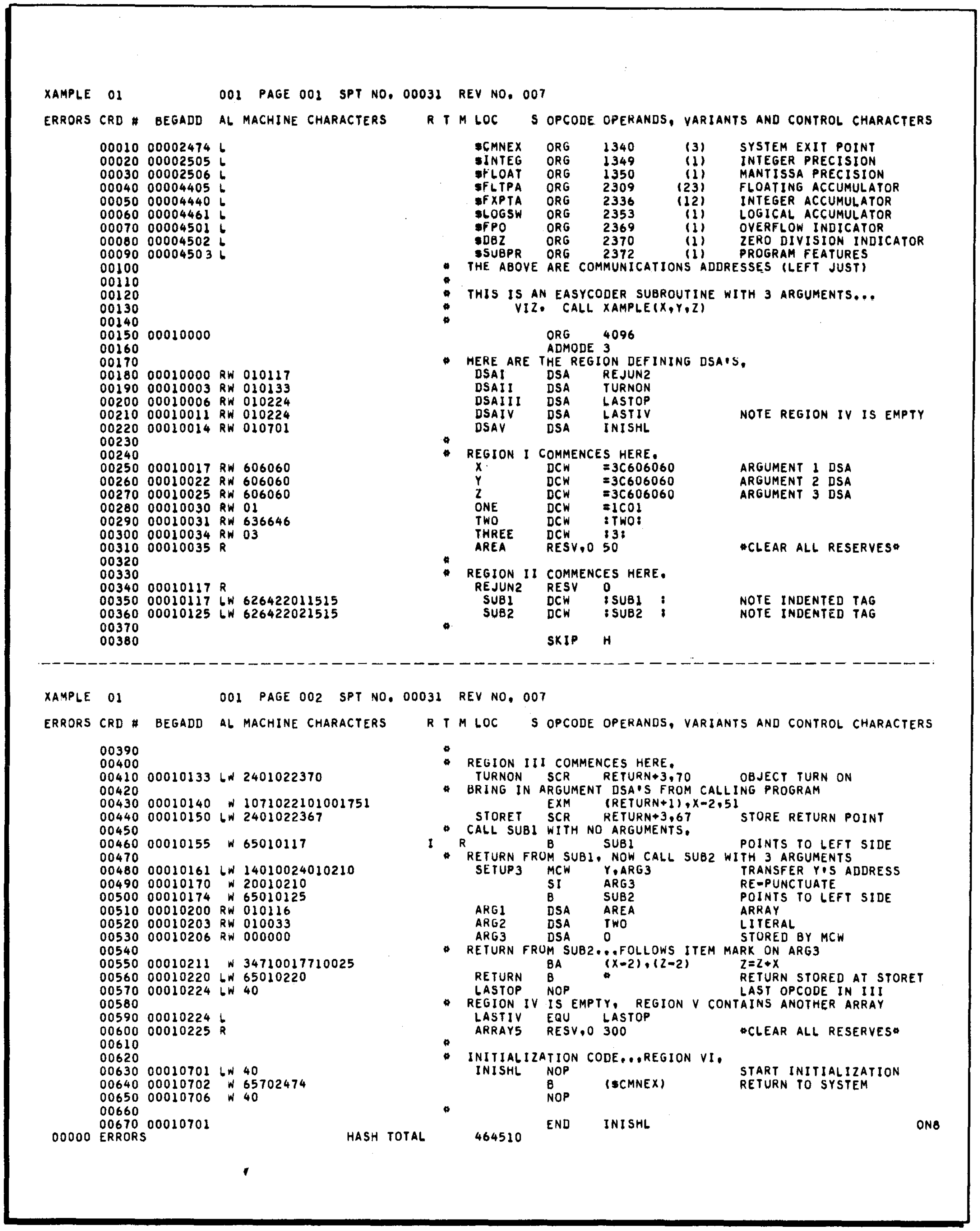
Any constants coded outside the specified regions will not be loaded at object time.

PDT and PCB instructions should be used only when absolutely necessary. Easycoder program units should, instead, refer to object I/O routines either by using the correct calling sequence or via the common peripheral driver.

Easycoder programs should normalize any floating-point data that they generate or work with if this data is to be operated on by Fortran-generated code.

Sample Program

Figure E-1 shows a sample regionalized Easycoder program unit.



```
XAMPLE  01             001  PAGE 001  SPT NO, 00031  REV NO, 007

ERRORS CRD #  BEGADD  AL MACHINE CHARACTERS     R T M LOC    S OPCODE OPERANDS, VARIANTS AND CONTROL CHARACTERS

       00010 00002474 L                                $CMNEX   ORG    1340       (3)    SYSTEM EXIT POINT
       00020 00002505 L                                $INTEG   ORG    1349       (1)    INTEGER PRECISION
       00030 00002506 L                                $FLOAT   ORG    1350       (1)    MANTISSA PRECISION
       00040 00004405 L                                $FLTPA   ORG    2309      (23)    FLOATING ACCUMULATOR
       00050 00004440 L                                $FXPTA   ORG    2336      (12)    INTEGER ACCUMULATOR
       00060 00004461 L                                $LOGSW   ORG    2353       (1)    LOGICAL ACCUMULATOR
       00070 00004501 L                                $FPO     ORG    2369       (1)    OVERFLOW INDICATOR
       00080 00004502 L                                $DBZ     ORG    2370       (1)    ZERO DIVISION INDICATOR
       00090 00004503 L                                $SUBPR   ORG    2372       (1)    PROGRAM FEATURES
       00100                                         * THE ABOVE ARE COMMUNICATIONS ADDRESSES (LEFT JUST)
       00110                                         *
       00120                                         * THIS IS AN EASYCODER SUBROUTINE WITH 3 ARGUMENTS...
       00130                                         *       VIZ.  CALL XAMPLE(X,Y,Z)
       00140                                         *
       00150 00010000                                         ORG    4096
       00160                                                  ADMODE 3
       00170                                         * HERE ARE THE REGION DEFINING DSA'S,
       00180 00010000 RW 010117                        DSAI     DSA    REJUN2
       00190 00010003 RW 010133                        DSAII    DSA    TURNON
       00200 00010006 RW 010224                        DSAIII   DSA    LASTOP
       00210 00010011 RW 010224                        DSAIV    DSA    LASTIV           NOTE REGION IV IS EMPTY
       00220 00010014 RW 010701                        DSAV     DSA    INISHL
       00230                                         *
       00240                                         * REGION I COMMENCES HERE.
       00250 00010017 RW 606060                        X        DCW    =3C606060        ARGUMENT 1 DSA
       00260 00010022 RW 606060                        Y        DCW    =3C606060        ARGUMENT 2 DSA
       00270 00010025 RW 606060                        Z        DCW    =3C606060        ARGUMENT 3 DSA
       00280 00010030 RW 01                            ONE      DCW    =1C01
       00290 00010031 RW 636646                        TWO      DCW    :TWO:
       00300 00010034 RW 03                            THREE    DCW    :3:
       00310 00010035 R                                AREA     RESV,0 50               *CLEAR ALL RESERVES*
       00320                                         *
       00330                                         * REGION II COMMENCES HERE.
       00340 00010117 R                                REJUN2   RESV   0
       00350 00010117 LW 626422011515                   SUB1    DCW    :SUB1  :         NOTE INDENTED TAG
       00360 00010125 LW 626422021515                   SUB2    DCW    :SUB2  :         NOTE INDENTED TAG
       00370                                         *
       00380                                                  SKIP   H
-------------------------------------------------------------------------------------------------------------

XAMPLE  01             001  PAGE 002  SPT NO, 00031  REV NO, 007

ERRORS CRD #  BEGADD  AL MACHINE CHARACTERS     R T M LOC    S OPCODE OPERANDS, VARIANTS AND CONTROL CHARACTERS

       00390                                         *
       00400                                         * REGION III COMMENCES HERE.
       00410 00010133 LW 2401022370                    TURNON   SCR    RETURN+3,70      OBJECT TURN ON
       00420                                         * BRING IN ARGUMENT DSA'S FROM CALLING PROGRAM
       00430 00010140  W 1071022101001751                       EXM    (RETURN+1),X-2,51
       00440 00010150 LW 2401022367                    STORET   SCR    RETURN+3,67      STORE RETURN POINT
       00450                                         * CALL SUB1 WITH NO ARGUMENTS.
       00460 00010155  W 65010117                 I   R         B      SUB1             POINTS TO LEFT SIDE
       00470                                         * RETURN FROM SUB1. NOW CALL SUB2 WITH 3 ARGUMENTS
       00480 00010161 LW 14010024010210                SETUP3   MCW    Y,ARG3           TRANSFER Y'S ADDRESS
       00490 00010170  W 20010210                               SI     ARG3             RE-PUNCTUATE
       00500 00010174  W 65010125                               B      SUB2             POINTS TO LEFT SIDE
       00510 00010200 RW 010116                        ARG1     DSA    AREA             ARRAY
       00520 00010203 RW 010033                        ARG2     DSA    TWO              LITERAL
       00530 00010206 RW 000000                        ARG3     DSA    0                STORED BY MCW
       00540                                         * RETURN FROM SUB2...FOLLOWS ITEM MARK ON ARG3
       00550 00010211  W 34710017710025                         BA     (X-2),(Z-2)      Z=Z+X
       00560 00010220 LW 65010220                      RETURN   B      *                RETURN STORED AT STORET
       00570 00010224 LW 40                            LASTOP   NOP                     LAST OPCODE IN III
       00580                                         * REGION IV IS EMPTY.  REGION V CONTAINS ANOTHER ARRAY
       00590 00010224 L                                LASTIV   EQU    LASTOP
       00600 00010225 R                                ARRAY5   RESV,0 300              *CLEAR ALL RESERVES*
       00610                                         *
       00620                                         * INITIALIZATION CODE...REGION VI.
       00630 00010701 LW 40                            INISHL   NOP                     START INITIALIZATION
       00640 00010702  W 65702474                               B      ($CMNEX)         RETURN TO SYSTEM
       00650 00010706  W 40                                     NOP
       00660                                         *
       00670 00010701                                           END    INISHL                                ON8
 00000 ERRORS                        HASH TOTAL        464510
```

Figure E-1. Sample Regionalized Easycoder Program Unit

NOTES ON THE FOUR-CHARACTER ADDRESS MODE

In addition to the system processes outlined above, the Easycoder program unit is transformed from the three-character to the four-character mode when the 4-character execution run option is set. For these instances, the following coding precautions should be observed:

1. Fields in region 1 or 5 which are used to store addresses during execution must be coded in argument DSA form.
2. Address modification, by adding constants, will produce incorrect results unless the constants are specialized during initialization in accordance with the address mode of execution.
3. All CAM instructions will be specialized to CAM 60 unless the op code is item marked.

APPENDIX F
TAPE AND MEMORY LAYOUTS

SYMBOLIC PROGRAM TAPE

The system is distributed in the form of a symbolic program tape, from which the compiler system tape is generated in accordance with the operating procedures described in Section IX.

COMPILER SYSTEM TAPE

The compiler system tape is a binary run tape that contains all segments of the compiler, the diagnostic preprocessor, the translation routine Screen, the Fortran functions, and Honeywell-supplied subroutines. The organization of the compiler system tape is shown in Figure F-1.

BINARY RUN TAPES

Binary run tapes are produced by the run-tape generator and contain jobs in executable form. This tape, together with data cards, is the input to object-program execution. The organization of a binary run tape for a single job is shown in Figure F-2.

OBJECT TAPES

Object tapes are always rewound by a REWIND statement before use.

BCD Tapes

Formatted data records of up to 132 characters per unit record may be generated onto a BCD tape. The BCD tape begins with an 80-character header record and terminates with two end-of-file records. The organization of the BCD tape is shown in Figure F-3.

Binary Tapes

A binary tape is produced by an unformatted WRITE statement. Each WRITE statement produces one logical record consisting of as many physical records as required to contain every item on the list. The first record is an 80-character 1HDRΔ header record and the last two records are end-of-file records. The organization of binary tapes is shown in Figure F-4.

BINARY PROGRAM TAPES

Binary program tapes contain program units compiled by Fortran Compiler D or assembled by Easycoder Assembler C. Binary program tapes are used as input to the run-tape generator. The program units on the BPT are relocatable modulo 1 (i.e., the run-tape generator can relocate the instruction sequence starting at any available address).

Stack Tape

The stack tape is in binary program tape format.

Common Input Tape

The common input tape contains records in card-image format.

Common Output Tape

The common output tape contains records in printer-image format, punched card format, or both.

MEMORY MAP (COMPILATION TIME)

Figure F-5 shows the communication and tabular information held in memory during compilation. Refer to Section X for a discussion of the contents of the FORMAT/IEFN and source/token tables.

MEMORY MAP (EXECUTION TIME)

Figure F-6 shows memory allocations at execution time for system communication, object programs, library functions, the execution package of Fortran routines and tabular information.

Segment Name	Function
AAAMON	Tape Loader-Monitor C (Three-Character)
AAAMON	Tape Loader-Monitor C (Four-Character)
ACADRV	Common Peripheral Driver
ACASKP	Diagnostic Preprocessor Bypass Routine
ACCPRA	Diagnostic Preprocessor - Segment A
ACCPRB	Diagnostic Preprocessor - Segment B
ACCPRC	Diagnostic Preprocessor - Segment C
ACASTP	Compiler Header for Preprocessor Bypass
ACAIIO	Compiler Internal I/O Package
ACAVIS	Visibility Selection Routine
ACAMNA	Compiler Monitor - Segment A
ACACMP	BRT Compiler Header Record
ACASPA	Specification Analyzer
ACADAR	Data Allocation Routine
ACAFIL	Filter
ACADAT	Data Initialization Routine
ACAEDT	Edit Routine and Executable Statement Analyzer

Figure F-1. Compiler System Tape Organization

Segment Name	Function
ACASSA	Subscript Analyzer - Segment A
ACASSB	Subscript Analyzer - Segment B
ACASSC	Subscript Analyzer - Segment C
ACACIA	Control and I/O Processor - Segment A
ACACIB	Control and I/O Processor - Segment B
ACAARA	Arithmetic Statement Processor - Segment A
ACAARB	Arithmetic Statement Processor - Segment B
ACAGNA	Generator - Segment A
ACAGNB	Generator - Segment B
ACAGNC	Generator - Segment C
ACALSA	Listing Options - Segment A
ACALSB	Listing Options - Segment B
ACALSC	Listing Options - Segment C
ACALSD	Listing Options - Segment D
ACAMNB	Compiler Monitor - Segment B
ACARTG	Run-Tape Generator
ACARIV	Four-Character Run-Tape Generator
ACAERR	Dump Calling and Loading Routine
ACAMEM	Memory Dump with Duplicate Line Suppression
ACADGM	Bypasses ACADGN if compilation or collection fails
ACADGN	Post-Execution Diagnostic Package
ACAMNC	Compiler Monitor - Segment C
ACABOO	Four-Character Chain Allocator
ACBBLB	Three-Character Chain Allocator
ACBFPR	Floating-Point Relational Routine
ACBFXR	Fixed-Point Relational Routine
ACBFPP	Calling Routine for Floating-Point Package
ACBFPH	Floating-Point Package for Hardware Multiply/Divide
ACBFPS	Floating-Point Package for Software Multiply/Divide
ACBFXP	Fixed-Point Package
ACBOIO	Object I/O Control Routine
BCDCON	Object I/O Routine for formatted READ and WRITE Statements
EFGCNV	Object I/O Routine for E, F, and G Conversions
INTCON	Object I/O Routine for I Conversions
BINARY	Object I/O Routine for unformatted READ and WRITE Statements

Figure F-1 (cont). Compiler System Tape Organization

Segment Name	Function
LOGOCT	Object I/O Routine for L and O Conversions
BACKSP	Object I/O Routine for BACKSPACE statements
ENDFIL	Object I/O Routine for END FILE statements
EOFPAR	Object I/O Routine for EOF and PARITY subroutines
IODIAG	Object I/O Routine which issues a diagnostic if a required Object I/O Routine is not loaded
VFORMT	Object I/O Routine for formatting in an array
IO4CHI	4-Character Address Mode Interface for I/O
PR4CHI	4-Character Address Mode Interface for Floating-Point Relational Routine
XR4CHI	4-Character Address Mode Interface for Fixed-Point Relational Routine
PP4CHI	4-Character Address Mode Interface for Floating-Point Package
XP4CHI	4-Character Address Mode Interface for Fixed-Point Package
IFIX	Real-to-Integer Function
INT	Truncation
FLOAT	Integer-to-Real Function
ACBRRE	Real-to-Real Exponentiation
ACBIIE	Integer-to-Integer Exponentiation
ACBRIE	Real-to-Integer Exponentiation
TANH	Hyperbolic Tangent
ATAN2	Two-Argument Arctangent
ATAN	Arctangent
COS	Cosine
EXP	Exponential
ALOG10	Base-10 Logarithm
ALOG	Natural Logarithm
SIN	Sine
SQRT	Square Root
ABS	Absolute Value
AMAX1 MAX1 MAX0 AMAX0	Largest Value Functions

Figure F-1 (cont). Compiler System Tape Organization

Segment Name	Function
AMIN1	
MIN1	
MIN0	Smallest Value Functions
AMIN0	
AMOD	Remaindering
ACBFLO	Integer-to-Real Conversion
IABS	Absolute Value
ACBFIX	Real-to-Integer Conversion
AINT	Truncation
IOR	Inclusive "OR"
ICOMPL	Logical Complement
IEXCLR	Exclusive "OR"
IAND	Logical AND
PARITY	
DVCHK	
OVERFL	
EOT	
EOF	Special Subroutines
SSWTCH	
SLITET	
SLITE	
SIGN	Sign Transfer Functions
ISIGN	
MOD	Remaindering
IDIM	
DIM	Positive Difference Functions
DUMP	
PDUMP	Dump-Call Subroutines
MDUMP	
REREAD	Reread Subroutine
ACBMEM	Dynamic Memory Dumper
ACBCCH	Call Chain Routine
ACBFPR	Floating-Point Relational Routine
ACBFXR	Fixed-Point Relational Routine
ACBFPP	Floating-Point Package Calling Routine

Figure F-1 (cont). Compiler System Tape Organization

Segment Name	Function
ACBFPH	Floating-Point Package for Hardware Multiply/Divide
ACBFPS	Floating-Point Package for Software Multiply/Divide
ACBFXP	Fixed-Point Package
ACBELB	Execution Initializer
ACAERR	Dump Calling and Loading Routine
ACAMEM	Memory Dump with Duplicate Line Suppression
ACADGM	Bypasses ACADGN if compilation or collection fails
ACADGN	Post-Execution Diagnostic Package
ACAMNC	Compiler Monitor - Segment C
ACAMND	Compiler Monitor - Segment D
ACAMNE	Compiler Monitor - Segment E
ACAMNF	Compiler Monitor - Segment F
F2TOF4	Screen Routine

Figure F-1 (cont). Compiler System Tape Organization

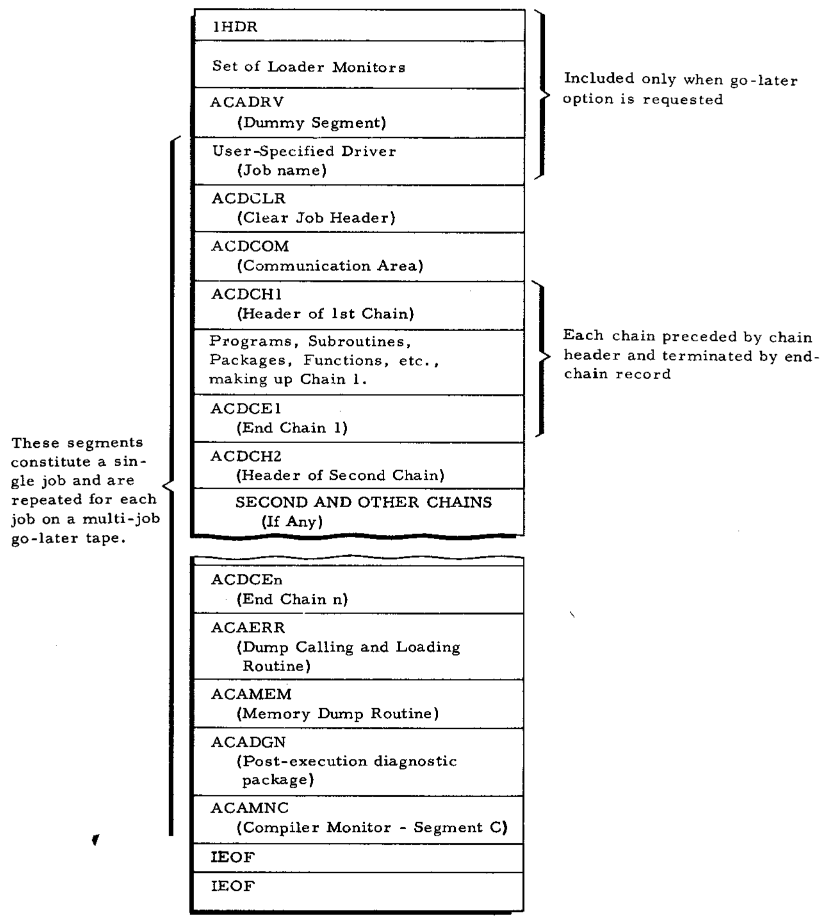

1HDR
Set of Loader Monitors
ACADRV (Dummy Segment)
User-Specified Driver (Job name)
ACDCLR (Clear Job Header)
ACDCOM (Communication Area)
ACDCH1 (Header of 1st Chain)
Programs, Subroutines, Packages, Functions, etc., making up Chain 1.
ACDCE1 (End Chain 1)
ACDCH2 (Header of Second Chain)
SECOND AND OTHER CHAINS (If Any)
ACDCEn (End Chain n)
ACAERR (Dump Calling and Loading Routine)
ACAMEM (Memory Dump Routine)
ACADGN (Post-execution diagnostic package)
ACAMNC (Compiler Monitor - Segment C)
IEOF
IEOF
Included only when go-later option is requested
Each chain preceded by chain header and terminated by end-chain record
These segments constitute a single job and are repeated for each job on a multi-job go-later tape.

Figure F-2. Organization of the Binary Run Tape

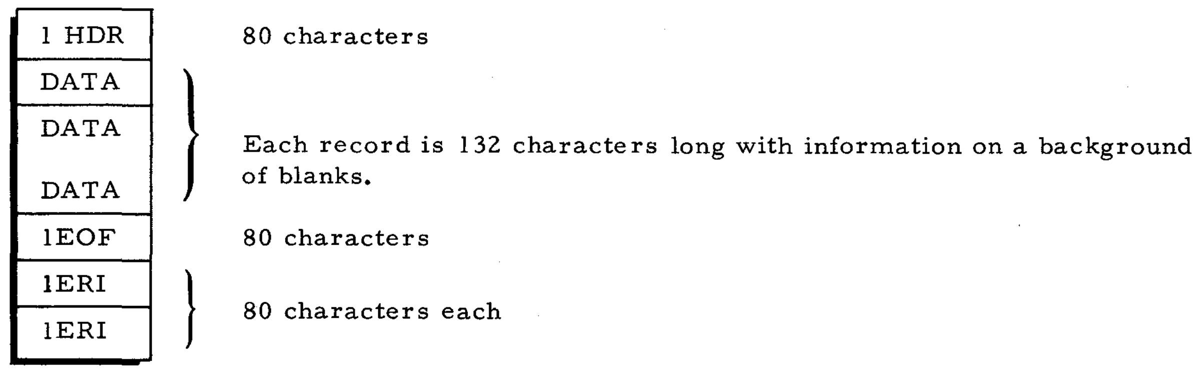
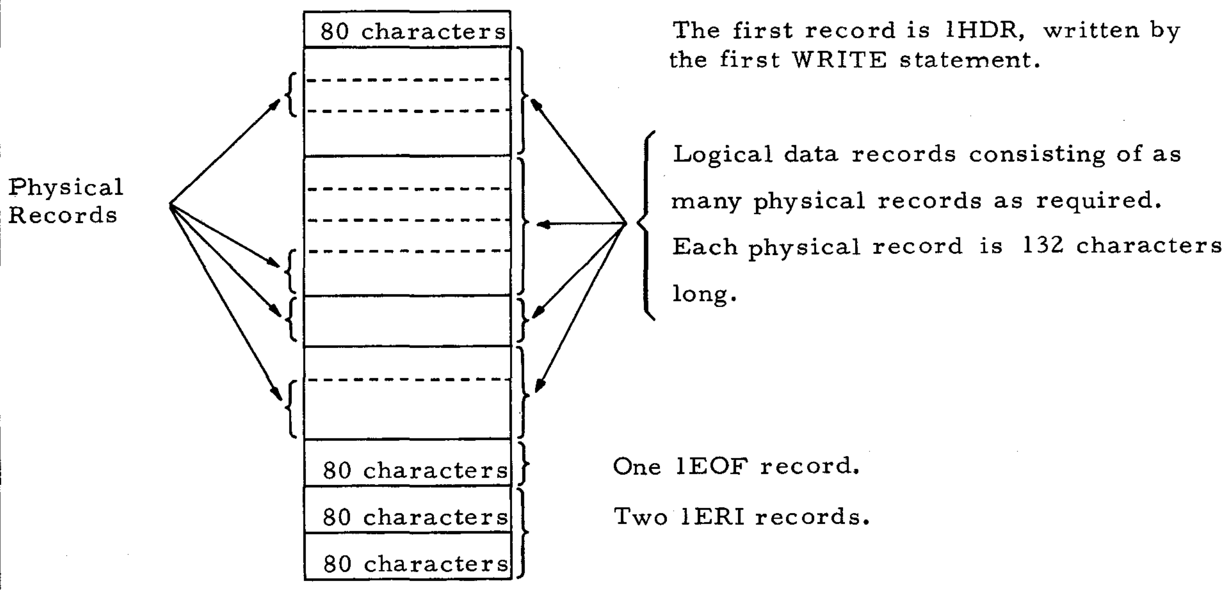

1 HDR
DATA
DATA
DATA
1EOF
1ERI
1ERI
80 characters
Each record is 132 characters long with information on a background of blanks.
80 characters
80 characters each

Figure F-3. Organization of BCD Tapes


80 characters
The first record is 1HDR, written by the first WRITE statement.
Physical Records
Logical data records consisting of as many physical records as required. Each physical record is 132 characters long.
80 characters
One 1EOF record.
80 characters
80 characters
Two 1ERI records.

LAYOUT OF PHYSICAL RECORDS ON BINARY TAPES

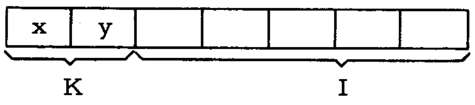
A	B	B	B	C	C	07	DATA FIELDS	77	Δ}	{Δ

1. A – First character tells where this record is in the logical record by one of the following values:
   a. 50 – First physical record.
   b. 41 – A physical record between the first and the last record.
   c. 44 – Last physical record.
   d. 54 – This is the only physical record in this logical record.
2. BBB – The number of information characters in this record from A to and including the 77.
3. CC – Sequence number of the physical record in the logical record.
4. 07 – The seventh character is always an octal 07.

Figure F-4. Organization of Binary Tapes

5. Data Fields:

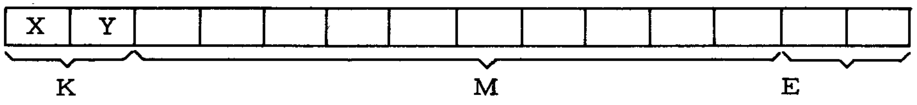
a. INTEGER:



K – control characters: x - the high-order bit is always 1 for an integer. Y - These five bits tell how many characters there are in the fixed-point number I.

I – This is the fixed-point number in binary as it appears in memory (see Appendix C).

b. REAL:

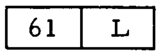


K – control character: X - the high-order bit is always 0 for a real datum. Y — The number of characters in M and E.

M – The mantissa in decimal and normalized as it appears in memory (see Appendix C).

E – The exponent in decimal as it appears in memory (see Appendix C).

c. LOGICAL:



61 means one character of logical data follows:

L - 00 = . FALSE.

XX = any other number is . TRUE.

6. 77 – Termination character for this physical record.

7. If a record contains 131 or fewer characters, it is filled out to the required 132 characters by the use of space fillers such as octal 15's.

Figure F-4 (cont). Organization of Binary Tapes

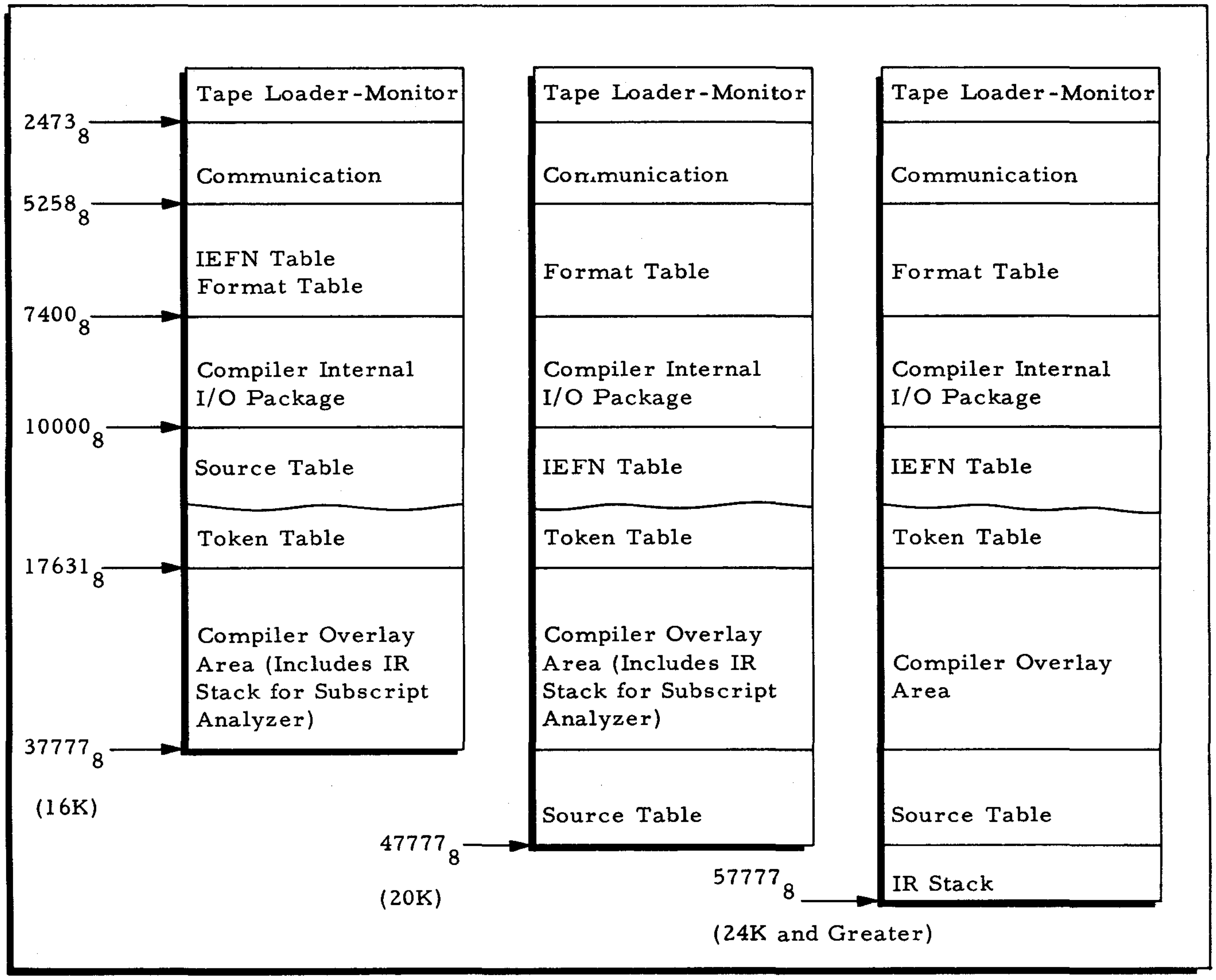


Figure F-5. Memory Map (Compilation Time)

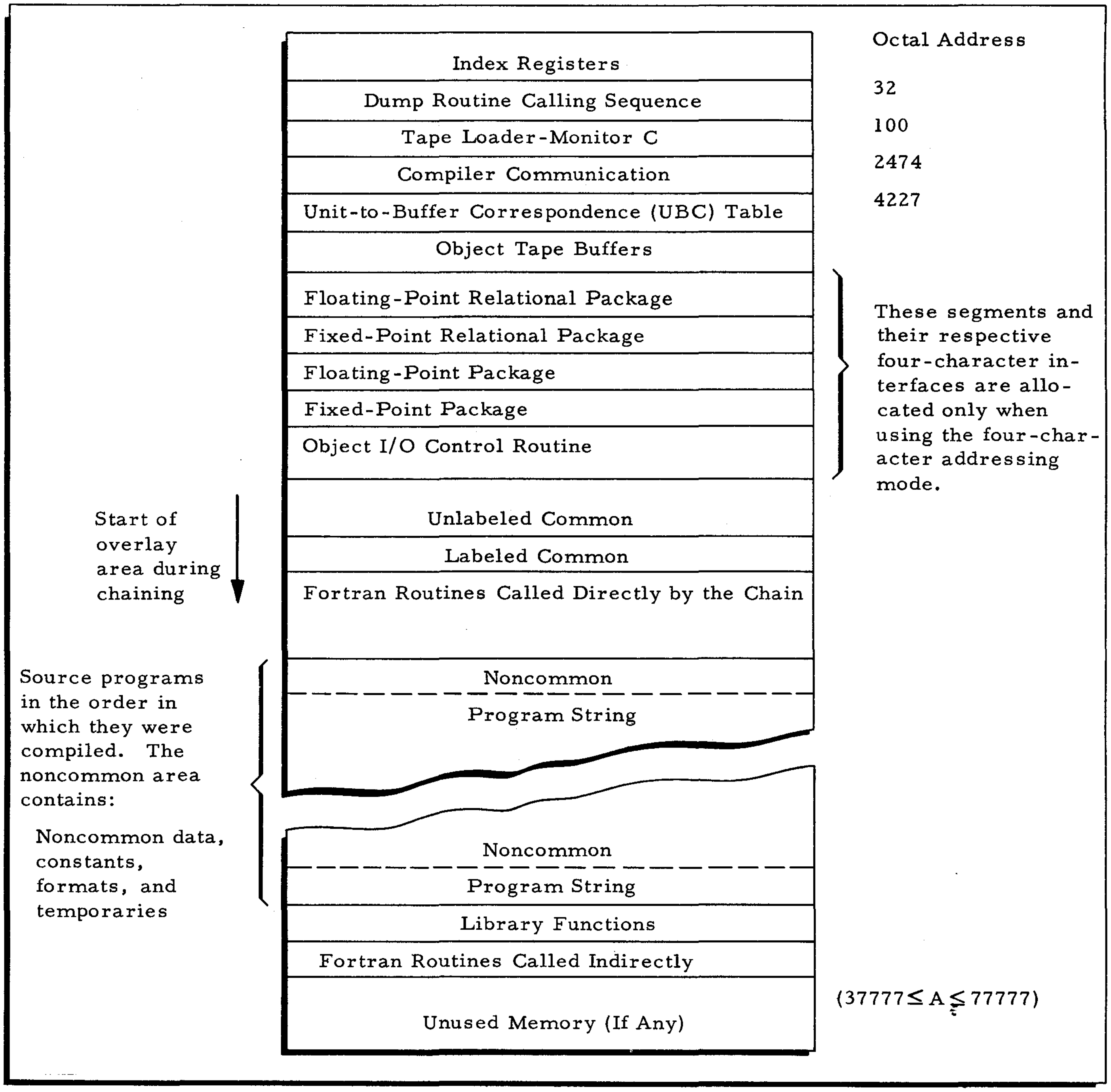

Octal Address
Index Registers
32
Dump Routine Calling Sequence
100
Tape Loader-Monitor C
2474
Compiler Communication
4227
Unit-to-Buffer Correspondence (UBC) Table
Object Tape Buffers
Floating-Point Relational Package
Fixed-Point Relational Package
Floating-Point Package
Fixed-Point Package
Object I/O Control Routine
These segments and their respective four-character interfaces are allocated only when using the four-character addressing mode.
Start of overlay area during chaining
Unlabeled Common
Labeled Common
Fortran Routines Called Directly by the Chain
Source programs in the order in which they were compiled. The noncommon area contains:
Noncommon data, constants, formats, and temporaries
Noncommon
Program String
Noncommon
Program String
Library Functions
Fortran Routines Called Indirectly
Unused Memory (If Any)
(37777≤A≤77777)

Figure F-6. Memory Map (Execution Time)

APPENDIX G
ERROR MESSAGES

PREPROCESSOR ERROR MESSAGES

Errors in Specification Statements

DATA APPEARS AFTER THE TERMINATING RIGHT PARENTHESIS.

ILLEGAL CHARACTER - TREATED AS BLANK.

YOU MAY NOT BEGIN A STATEMENT WITH A ________.

THIS CARD IS OUT OF ORDER AND WILL BE IGNORED ENTIRELY.

THIS CHARACTER ________ IS ERRONEOUS.

VARIABLES MAY NOT HAVE MORE THAN TWO DIMENSIONS.

DIMENSION MUST BE EXPRESSED NUMERICALLY - ZERO HAS BEEN SUBSTITUTED.

________ APPEARS MORE THAN ONCE IN A COMMON STATEMENT.

________ HAS BEEN DIMENSIONED MORE THAN ONCE - ACCEPTING FIRST.

YOU HAVE OVERFLOWED THE VARIABLE TABLE.

________ HAS BEEN DEFINED IN MORE THAN ONE MODE STATEMENT - ACCEPTING FIRST.

CONTINUATION CARDS MAY NOT HAVE STATEMENT NUMBERS - NUMBER IS BEING IGNORED.

NO TERMINATING RIGHT PARENTHESIS, OR BAD PUNCTUATION CAUSED IT TO BE IGNORED.

FUNCTION NAME MUST BE FOLLOWED BY ARGUMENT LIST.

TAG HAS MORE THAN 6 CHARACTERS.

A VARIABLE MAY NOT BEGIN WITH A NUMBER.

THIS STATEMENT IS NOT ALLOWED.

TOO MANY CONTINUATION CARDS.

MISSING COMMA.

VARIABLE NOT FOLLOWED BY LEFT PAREN.

THIS STATEMENT IS NOT COMPLETE BUT APPEARS TO HAVE NO CONTINUATION CARD.

BAD PUNCTUATION.

BLANKS ARE NOT ACCEPTABLE AS DIMENSIONS.

PREPROCESSOR ERROR MESSAGES (cont)

Errors In Arithmetic and Logical Expressions

FIRST ELEMENT OF STATEMENT IS NOT A VARIABLE.

THE SINGLE VARIABLE ON THE LEFT-HAND SIDE OF THE STATEMENT IS NOT FOLLOWED BY AN EQUAL SIGN.

THE ARGUMENT LIST OF THIS STATEMENT FUNCTION IS IN ERROR.

THE SUBROUTINE NAME IS NOT FOLLOWED BY A LEFT PARENTHESES.

THE OPERAND IS ILLEGAL IN THIS POSITION.

TOO MANY RIGHT PARENTHESES.

AN OPERATOR HAS OPERANDS OF DIFFERENT TYPES WHERE THIS IS ILLEGAL.

THE OPERATOR .NOT. IS FOLLOWED BY A NON-LOGICAL EXPRESSION.

A CALL STATEMENT HAS CHARACTERS FOLLOWING THE RIGHT PARENTHESIS.

TOO MANY LEFT PARENTHESES.

STATEMENT CAUSED TABLE OVERFLOW. (unevaluated-expression stack overflow)

A SOURCE ELEMENT APPEARS IN AN ILLEGAL RELATION TO PREVIOUS ELEMENTS. (syntactic error)

PARENTHESES ARE NESTED TO A DEPTH GREATER THAN 63.

THE EXPRESSION PRECEDING OR FOLLOWING AN .AND. OR .OR. OPERATOR IS NOT LOGICAL.

BOTH SIDES OF THIS EXPRESSION ARE NOT LOGICAL.

A LOGICAL, HOLLERITH, OR OCTAL EXPRESSION FOLLOWS A + OR -.

AN ARITHMETIC OPERATOR-- /OR * OR ** --IS NOT PRECEDED BY AN OPERAND.

ONE OR BOTH OPERANDS OF AN EXPONENTIAL EXPRESSION IS OF LOGICAL, OCTAL OR HOLLERITH TYPE.

FUNCTION OR SUBROUTINE CALL NOT TERMINATED BY A RIGHT PARENTHESIS.

A LOGICAL, OCTAL, OR HOLLERITH EXPRESSION FOLLOWS A * OR A / OR A RELATIONAL OPERATOR.

Errors in Construction and Use of Subscripts

THIS APPEARS TØ BE AN ERRONEOUS SUBSCRIPT.

A NON-INTEGER NUMBER OCCURS IN THE SUBSCRIPT EXPRESSION.

NON-INTEGER VARIABLE IN A SUBSCRIPT.

PREPROCESSOR ERROR MESSAGES (cont)

Errors in Construction and Use of Subscripts (cont)

A NON-NUMERIC FOLLOWS A + OR - IN THE SUBSCRIPT.

SUBSCRIPT EXPRESSION IS NOT FOLLOWED BY A COMMA OR RIGHT PARENTHESIS.

SUBSCRIPT EXPRESSION IS TERMINATED BY END OF STATEMENT. (NO COMMA OR RIGHT PARENTHESIS).

SUBSCRIPT CONTAINS MORE EXPRESSIONS THAN THE NUMBER OF DIMENSIONS IN THE ASSOCIATED ARRAY.

SUBSCRIPT CONTAINS FEWER EXPRESSIONS THAN THE NUMBER OF DIMENSIONS IN THE ASSOCIATED ARRAY.

A SUBSCRIPT CANNOT APPEAR IN A STATEMENT FUNCTION DEFINITION.

Statement Label Errors

CONTINUATION CARDS MAY NOT HAVE STATEMENT NUMBERS.

DUPLICATE STATEMENT NUMBER.

ZERO STATEMENT NUMBER NOT ALLOWED.

STATEMENT NUMBERS MAY NOT CONTAIN ALPHABETICS - NUMBER IS BEING IGNORED.

EFN TABLE OVERFLOW.

THIS STATEMENT NUMBER WAS PREVIOUSLY USED IN A FORMAT CONTEXT.

STATEMENT NUMBER WAS PREVIOUSLY USED IN AN EXECUTABLE STATEMENT CONTEXT.

FORMAT STATEMENTS MUST HAVE STATEMENT NUMBERS.

DO LOOPS ARE IMPROPERLY NESTED.

A DO LOOP HAS NOT BEEN TERMINATED.

AN UNDEFINED STATEMENT NUMBER HAS BEEN REFERENCED.

STATEMENT NUMBER CANNOT HAVE MORE THAN FIVE DIGITS.

FORMAT Statement Errors

MISSING DECIMAL POINT.

AN A, I, O, L, OR X FIELD IS TOO LARGE.

MEANINGLESS NUMERIC.

PREPROCESSOR ERROR MESSAGES (cont)

FORMAT Statement Errors (cont)

DUPLICATE FIELD SPECIFICATION OR MISSING COMMA.

EITHER A MEANINGLESS DECIMAL POINT OR MISSING FIELD SPECIFICATION.

NUMBER BEFORE THE DECIMAL POINT EXCEEDS 32.

THE NUMBER BEFORE THE DECIMAL POINT IS ZERO OR BLANK.

THE NUMBER AFTER THE DECIMAL POINT IS TOO LARGE.

MORE THAN THREE NESTED PARENTHESES.

HOLLERITH FIELD CANNOT EXCEED 132.

NO BEGINNING LEFT PARENTHESIS.

AN E, F, OR G FIELD IS NOT COMPLETE.

AN A, I, L, O, X, OR H FIELD IS BLANK OR ZERO.

THERE IS DATA AFTER THE TERMINATING RIGHT PARENTHESIS.

SCALE FACTOR LEGAL ONLY WITH E, F, OR G FIELD SPECIFICATIONS.

+ OR - LEGAL ONLY WITH SCALE FACTOR.

ZERO REPEAT IS TREATED LIKE A ONE.

THIS STATEMENT CONTAINS NO INFORMATION.

MISSING CONTINUATION CARD.

Errors in Construction of Program Constants

AN OCTAL OR HOLLERITH VARIABLE SPECIFICATION CONTAINS MORE THAN THREE DECIMAL DIGITS.

AN OCTAL OR HOLLERITH VARIABLE SPECIFICATION IS ZERO.

STATEMENT ENDED BEFORE SATISFYING THE OCTAL OR HOLLERITH VARIABLE SPECIFICATION.

DECIMAL POINT IS FIRST CHARACTER OF OPERAND OR OPERATOR BUT IS FOLLOWED BY ANOTHER OPERATOR.

THE DECIMAL POINT APPEARS TO BEGIN A LOGICAL OR RELATIONAL OPERATOR OR A LOGICAL CONSTANT.

INCORRECT EXPONENT.

THIS REAL CONSTANT APPEARS TO CONTAIN TWO DECIMAL POINTS.

PREPROCESSOR ERROR MESSAGES (cont)

Control and I/O Statement Errors

DO STATEMENTS MAY NOT HAVE PARAMETERS OF ZERO.

TOO MANY RIGHT PARENTHESES.

AN INTEGER VARIABLE NAME REQUIRED HERE.

SOME ERROR IN STATEMENT CONSTRUCTION.

THE WORD TO IS NOT FOUND.

INVALID CHAIN DETECTOR FOUND.

ILLEGAL CHARACTER PRESENT.

INTEGER HAS INCORRECT CONSTRUCTION.

MORE THAN TEN NESTED DO LOOPS.

ARRAY NAME REQUIRED HERE.

ARRAY IS NOT ONE DIMENSIONAL.

ARRAY IS NOT TWO DIMENSIONAL.

RIGHT PARENTHESIS MISSING.

THE INDEXED VARIABLE USED IN AN IMPLIED DO LIST IS NOT THE CONTROL VARIABLE OF THAT LIST.

THIS INTEGER IS LIMITED TO 63.

A MAIN PROGRAM MAY NOT CONTAIN A RETURN STATEMENT.

ONLY SIX OCTAL CHARACTERS ARE ACCEPTED IN A STOP OR PAUSE STATEMENT.

ONLY OCTAL CHARACTERS ARE ACCEPTED IN A STOP OR PAUSE STATEMENT.

THE STATEMENT NUMBER CONTAINS A NON-NUMERIC CHARACTER.

THIS STATEMENT APPEARS TO CONTAIN NO INFORMATION.

A DO STATEMENT MAY NOT FOLLOW A LOGICAL IF.

Errors in IF or CALL Statements or the Use of Functions or Subroutines

THE STATEMENT FUNCTION HAS BEEN DEFINED OR USED PREVIOUSLY.

THE SUBPROGRAM NAME HAS BEEN USED AS BOTH FUNCTION AND SUBROUTINE.

STATEMENT FUNCTION DEFINITION OUT OF PLACE.

PREPROCESSOR ERROR MESSAGES (cont)

Errors in IF or CALL Statements or the Use of Functions or Subroutines (cont)

THIS APPEARS TO BE AN IF STATEMENT BUT HAS NO LEFT PARENTHESIS AFTER IF.

THIS APPEARS TO BE A CALL STATEMENT BUT NO SUBROUTINE NAME FOLLOWS CALL.

CALL IS FOLLOWED BY A NAME THAT PREVIOUSLY APPEARED IN A DIMENSION STATEMENT.

THE CHARACTER AFTER THE RIGHT PARENTHESIS IS NEITHER A DIGIT NOR A LETTER.

A LOGICAL IF WITH A NON-LOGICAL EXPRESSION, OR AN ARITHMETIC IF WITH A LOGICAL EXPRESSION.

TOO MANY FUNCTION/SUBROUTINE NAMES HAVE BEEN ENTERED IN THE FUNCTION/ SUBROUTINE TABLE.

LOGICAL IF FOLLOWING LOGICAL IF.

Keypunch and Other Miscellaneous Errors

THIS STATEMENT APPEARS TO BE INCOMPLETE.

AN OTHERWISE VALID STATEMENT IS IMPROPERLY TERMINATED.

TAG HAS MORE THAN SIX CHARACTERS.

A VARIABLE BEGINS WITH A NUMBER.

A SPECIFICATION STATEMENT IS OUT OF ORDER.

THIS STATEMENT IS NOT ALLOWED.

MORE THAN NINE CONTINUATION CARDS.

MISSING COMMA.

THIS STATEMENT IS TOO GARBLED TO ANALYZE.

MISSING LEFT PARENTHESIS.

ILLEGAL CHARACTER.

STATEMENT MAY NOT BEGIN WITH A DELIMITER - DELIMITER IS BEING IGNORED.

THERE SHOULD BE A CONTINUATION CARD.

BAD PUNCTUATION.

THIS CHARACTER IS ERRONEOUS.

COMPILER ERROR MESSAGES

All compiler error messages are fatal unless otherwise specified.

Error Number	Meaning
	Errors Detected by the Arithmetic Processor
1	First element of a statement is a constant or it is a function name not followed by a left parenthesis.
2.	The variable on the lefthand side is not followed by either an equals sign or a subscript expression.
3	The variable on the lefthand side (including statement function argument list or subscript expression) is not followed by an equals sign.
4	An arithmetic expression begins with an operator on the lefthand side.
5	The lefthand side has a constant followed by a left parenthesis.
6	In a statement function definition, the dummy argument list must consist only of operands separated by commas.
7	The dummy argument list of a statement function definition must consist only of operands separated by commas.
8	The dummy argument list of a statement function definition must consist only of operands separated by commas.
9	The first operand of two or more on the righthand side is an octal or Hollerith constant.
10	The delimiter, relational operator, logical operator, or period following an equals sign, comma, left parenthesis, logical operator, or relational operator is illegal.
13	A function or subroutine name is neither the last element of a source program nor it followed by a left parenthesis.
14	An array name is followed by another operand.
15	An arithmetic delimiter (+ - * / **) is not followed by an operand or left parenthesis.

Error Number	COMPILER ERROR MESSAGES Meaning
	<u>Errors Detected by the Arithmetic Processor (cont)</u>
16	An operand or right parenthesis is followed by an operand.
17	An operand or right parenthesis (including subscript expression or argument list) is followed by a left parenthesis.
18	Too many right parentheses.
19	An integer is being raised to a real power.
21	Previous processing by the arithmetic statement processor is now detected as erroneous. Dump for Software Support at the occurrence of this error.
22	The operator .NOT. is followed by an arithmetic (non-logical) expression. (The error occurs only when the arithmetic expression is in a pseudo-accumulator at the same time that the .NOT. should be compiled.)
23	A right parenthesis or operand is followed by a .NOT.
24	Previous processing by the arithmetic statement processor is now detected as erroneous. Dump for Software Support at the occurrence of this error.
25	A CALL statement has characters following the terminating right parenthesis of the argument list. Check for too many right parentheses.
26	Too many left parentheses.
27	Table overflow - the statement is too long.
	<u>Errors Detected by the Edit</u>
50	A statement number (EFN) is used on a continuation card.
51	The I-EFN/format table has overflowed. The number of EFN or FORMAT statements must be reduced.
52	An unequal number of parentheses occurred in the statement.
54	A syntactical error has occurred in the statement.
55	Duplicate statement numbers.
56	More than 63 EFN's in a computed GO TO statement were detected.
57	A PRINT or PUNCH statement appears in the source program.
58	A unit number greater than 15 was detected in an I/O statement.
59	Incomplete Hollerith constant at the end of a line was not completed on the continuation line.
60	A FORMAT statement was completed at the end of the last line but a continuation line appears to be part of the same statement.
PRINT-OUT	SORCE TBL OVFLOW (Source or token table has overflowed.)
PRINT-OUT	XXX EFN UNDEFINED
	<u>Errors Detected by the Control and I/O Processor</u>
150	No format statement with the referenced statement label can be found.
151	The I/O list has an illegal delimiter, logical or relational operator, or period.

Error Number	COMPILER ERROR MESSAGES Meaning
	Errors Detected by the Control and I/O Processor (cont)
152	Implied DO loops may not be nested to a depth greater than two.
153	An implied DO loop is not terminated. (Probably a missing right parenthesis.)
154	DO loops may not be nested to a depth greater than ten.
155	This DO loop is incorrectly nested.
156	No executable statement with this statement label can be found.
	Errors Detected by the Subscript Processor
	A single printout indicates that one of nine tables has overflowed. The printout and the possible tables are given below. THE SUBSCRIPT ANALYZER HAS COMPILED THIS JOB WITH FATAL ERRORS BECAUSE THE ________ TABLE HAS OVERFLOWED.
SUPVL	A list of variables that are redefined in any statement within a DO loop and that have previously been used in subscript expressions in the loop. Variables that are said to be redefined are variables on the lefthand side of arithmetic statements, function/subroutine call arguments, all Common variables if a function or subroutine call is present, READ statement variables, and control variables of implied DO loops. The SUPVL table can contain up to 30 variables.
SNDVTL	A list of the variable parts of subscript expressions computed within a block.[1] Each list entry consists of a variable name and its constant multiplier. No DO control variables are included. The SNDVTL table can contain up to 40 entries.
SDVTL	A list of DO control variables identical in form to the SNDVTL list. The SDVTL table can contain up to 20 entries.
SPACK	Previous processing by the subscript processor is now detected as being in error. This printout is normally accompanied by one of the other table overflow printouts. Dump for Software Support if this table, and no other table, overflows.
SFTL	A list of temporary locations that have been released for reuse. Overflow of SFTL should be accompanied by overflow of either SNDVTL or SDVTL.
SCONTB	A list of constants needed at object time by subscript-generated code. The SCONTB table can contain up to 52 entries.
SLHST	A list of lefthand-side variables in a block. The SLHST table can contain up to 60 entries.
SDOTAB	This table overflows if DO loops are nested to a depth greater than 10.

[1] A block is defined as a set of physically sequential source program statements. Blocks may be subdivided into smaller blocks for processing. The possible blocks are listed below.

An innermost DO loop (cannot be further subdivided).
A DO loop.
A group of statements, the first of which has a statement label and the last of which precedes the next labeled statement.
A group of statements begining with the first executable and ending with the last executable statement of a program or subprogram.

Error Number	COMPILER ERROR MESSAGES Meaning
	<u>Errors Detected by the Subscript Processor (cont)</u>
SIRSTK	A list of intermediate representation (IR) items necessary for sorting. It is emptied at the end of each block. Overflow indicates that the block is too large. This usually means that there is too much subscripting in a DO loop.
	<u>Errors Detected by the Filter</u>
200	A variable name has more than six characters.
201	Either the source or the token table has overflowed. The program must be modified or segmented in order to compile.
202	A subroutine name appears in a Data-Type statement.
203	A name in an EXTERNAL statement also appears on the lefthand side of an arithmetic statement.
204	An illegal constant of one of the following kinds: 1. Illegal character embedded in a real or integer constant 2. Two decimal points 3. Two E's 4. An E not followed by a digit, a plus, or a minus sign 5. Two many signs in the exponent 6. Too many digits in the exponent 7. A decimal point followed by an O or H 8. An E followed by an O or H
	<u>Errors Detected by the Specification Processor</u>
	(Error Nos. 300 to 311 are analyzer errors. As with most compiler-detected errors, processing resumes with the next statement. Error Nos. 320 and 321 are allocator errors. Error 320 causes immediate termination of allocation. If error 321 occurs, an attempt is made to continue allocation.)
300	More than six characters in a COMMON block label.
301	More than 15 COMMON block labels.
302	Illegal delimiter in the statement.
303	Statement is incomplete. Probably no right parenthesis in a DIMENSION statement.
304	Too many digits in a DIMENSION size.
305	Too many DIMENSION sizes. A maximum of two dimensions is allowed in an array.
306	More than six characters in a name.
307	Illegally embedded FUNCTION or SUBROUTINE statement.
308	Name of a function is not followed by a left parenthesis.
309	Too many unrelated EQUIVALENCE sets. (Maximum is 64.)
310	Too many digits in an EQUIVALENCE subscript. (Maximum is five.)
311	An array name appearing in an EQUIVALENCE statement with two dimensions has only one dimension in the DIMENSION statement.

Error Number	COMPILER ERROR MESSAGES Meaning
	Errors Detected by the Specification Processor (cont)
320	The token table has overflowed.
321	The preceding COMMON block label has been used more than once in the chain but the sizes specified for the block are not identical.
	Errors Detected by the DATA Initialization Statement Processor
400 Not Fatal	A statement number (EFN) is not permitted in a DATA statement and will be ignored.
401	Statement contains a character not in the Fortran character set.
402	References to labeled or unlabeled common storage are not permitted in DATA statements.
403	External (function or subroutine name) references are not permitted in DATA statements.
404	Dummy variables are not permitted in DATA statements.
405	A faulty delimiter in the variable list of a DATA statement.
406	Garbled string of characters in a variable list. (Probably a leading numeric.)
407	DATA statement lacks a constant list.
408	Garbled string of characters in constant list. (Probably a leading alphabetic.)
409	Illegal character string in constant list. (Probably an embedded illegal character.)
410 Not Fatal	The subscript exceeds the range of the array.
411	Illegal subscript in the variable list.
412	Type of variable and its corresponding constant do not agree.
413	Program deck contains an embedded control card.
414	Continuation card seems to have garbage in columns 1 to 5.
415	An illegal delimiter follows a logical TRUE or FALSE in constant list.
416	Repetition factor in constant list is not in unsigned integer form.
417	Constant list appears to contain a garbled Hollerith constant.
418	Variable name in DATA list exceeds six characters.
419	Illegal delimiter in constant list.
420 Not Fatal	Constant list exceeds the variable list. When the variable list is satisfied, remaining constants will be ignored.
421	Variable list exceeds the constant list.
422 Not Fatal	Constant list should be terminated by a slash.
423	Number of characters in Hollerith constant exceeds maximum integer size.
424	Illegal delimiter follows a Hollerith constant.
425	Incomplete Hollerith constant appears at end of statement.
426	Faulty implied DO format in variable list.
427	Faulty subscript in the variable list.

Error Number	COMPILER ERROR MESSAGES Meaning
	Errors Detected by the DATA Initialization Statement Processor (cont)
428	Illegal delimiter in subscript of implied DO list.
429	Faulty specification in implied DO list.
430	Implied DO subscript is not of integer type.
431	A constant list element contains an embedded sign.
432	DATA statement operator not followed by a variable or constant list.
	Errors Detected by the Run Tape Generator
	INSUFFICIENT MEMORY
	INCOMPLETE PROGRAM CANNOT FIND (followed by a list of all missing subprograms.)
	UNDEFINED CHAIN CALL (chain named in call not found in job.)
	TABLE OVERFLOW (Unresolved call table has overflowed. Re-arrange source deck so that frequently called subroutines appear early in the job.)
	Errors Detected by the Generator
550	Either the subscript packet or forward reference table has overflowed.
551	An illegal IR item has been detected.
552	A call has been made to a nonfunction or nonsubroutine (i.e., variable, array, or constant).
PRINTOUT	TOTAL PROGRAM SIZE IS XXXXX (8) WHICH IS TOO LARGE FOR THIS SYSTEM.
	Diagnostic Issued by the Pseudo-Easycoder LIST Processor
PRINTOUT	THIS PROGRAM IS TOO LARGE TO BE LISTED.

COMPILER MONITOR ERROR MESSAGES

THE PROGRAM XXXXXX CANNOT BE FOUND ON THE STACK TAPE.

(This printout occurs when the program named on a *GET card cannot be located on the stack tape. This error causes job fatality.)

THE BINARY DECK BEING PROCESSED APPEARS TO BE OUT OF SEQUENCE.

(A binary deck which is out of sequence causes job fatality.)

THE *DIAG CARD IS NOT DIRECTLY FOLLOWED BY A *JOBID CARD. THIS JOB IS FATAL.

THIS JOB IS FATAL.

(This printout occurs at the end of compilation if the job fatality indicator has been set by a previous diagnostic.)

SYSTEM MEMORY SIZE IS TOO LARGE — 32K SYSTEM SIZE WILL BE USED.

(This printout results when running in the 3-character mode if the memory specified in the console call is greater than 32K.)

AMOUNT OF MEMORY SPECIFIED IS NOT SUFFICIENT FOR FOUR-CHARACTER MODE.

OBJECT MEMORY SIZE IS TOO LARGE — SYSTEM MEMORY SIZE WILL BE SUBSTITUTED.

(This printout occurs when running in the 3-character mode with a *JOBID card specifying more than 32K.)

RUN-TAPE GENERATOR ERROR MESSAGES

INCOMPLETE PROGRAM CANNOT FIND (followed by a list of all missing subprograms)

UNDEFINED CHAIN CALL

INSUFFICIENT MEMORY. xxxxx CHARACTERS NEEDED.

TABLE OVERFLOW.

(This printout occurs when the unresolved call table overflows. If the diagnostic is issued, the programmer should attempt to rearrange the source deck so that frequently called subroutines appear early in the job.)

PUNCH REQUEST IGNORED FOR FLOATING LOADER ABOVE 32K.

INSUFFICIENT PRECISION FOR ASSIGNED GO TO.

EXECUTION-TIME ERROR MESSAGES

Errors in input/output routines or in library functions that are detected during job execution will result in English-language error printouts of either two or three lines each, followed by job exit. There are 19 possible input/output or library function error messages. In every case the first line of the error message is:

ERROR CONDITION TERMINATED EXECUTION OF THIS JOB. SEE BELOW.

Library Function Error Messages

When a library function causes an error, the subsequent line or lines are as follows:

Library Function In Error	Message
AMOD	AMOD WAS CALLED WITH ARG2=0. THIS IS AN ILLEGAL ARGUMENT.
MOD	MOD WAS CALLED WITH ARG2=0. THIS IS AN ILLEGAL ARGUMENT.
SIGN	SIGN ROUTINE WAS CALLED WITH ARG2=0. ILLEGAL ARGUMENT.
ISIGN	ISIGN WAS CALLED WITH ARG2=0. ILLEGAL ARGUMENT.
ALOG or ALOG10	ALOG OR ALOG10 CALLED WITH A ZERO OR MINUS ARGUMENT — ILLEGAL. (Subsequent line shows the value of the illegal argument in E-conversion format.)
SIN or COS	SIN OR COS CALLED WITH ARGUMENT .GT. 20.*PI (62.83185307). (Subsequent line shows value of illegal argument in E-conversion format.)
SQRT	SQRT WAS CALLED WITH A NEGATIVE ARGUMENT — ILLEGAL ARGUMENT. (Subsequent line shows value of illegal argument in E-conversion format.)

Library Function In Error	LIBRARY FUNCTION ERROR MESSAGES Message
ACBRRE	A NEGATIVE REAL NUMBER CANNOT BE RAISED TO A REAL EXPONENT. (Subsequent line shows the value of the illegal exponentiated number in E-conversion format.)
EXP	EXP CALLED WITH ARGUMENT EXCEEDING PLUS OR -227.95592420699. (Subsequent line shows value of illegal argument in E-conversion format.)

Input/Output Error Messages

Two types of input/output errors are detected. The first type involves improper use of a peripheral device. The second type occurs when an illegal character is detected in a FORMAT statement or in data input. The first line of the error message is the same as for library function errors. The subsequent lines are shown below:

IMPROPER COMMAND TO THIS DEVICE.
(Subsequent line shows peripheral device number.)

THIS DEVICE IS NOT ALLOCATED AT OBJECT TIME.
(Subsequent line shows peripheral device number.)

READ MAY NOT FOLLOW WRITE OR ENDFILE.
(Subsequent line shows peripheral device number.)

LIST FOR BINARY READ ON THIS DEVICE EXCEEDS DATA IN RECORD.
(Subsequent line shows peripheral device number.)

END-FILE MARKER ENCOUNTERED.
(Subsequent line shows peripheral device number.)

END OF TAPE ENCOUNTERED.
(Subsequent line shows peripheral device number.)

UNCORRECTABLE READ ERROR.
(Subsequent line shows peripheral device number.)

ASTERISK (*) IN COL. 1 OF INCOMING DATA TERMINATED THE JOB.
(Subsequent line shows peripheral device number.)

ILLEGAL CHARACTER IN FORMAT STATEMENT. SEE END OF LINE BELOW.
(Subsequent line shows FORMAT statement up to point of error.)

ILLEGAL CHARACTER IN INPUT DATA. BAD RECORD IS PRINTED BELOW.
(Subsequent line shows contents of bad record containing illegal character.)

CALLING I/O SEGMENT WHICH ISN'T LOADED.
(Printout occurs when variable formatting is used in the source program without a dummy FORMAT statement containing the required specifications.)

SCREEN ERROR MESSAGES

****FATAL****MORE THAN 16 COMMENT AND/OR CONTROL APPEAR IN SEQUENCE.

****FATAL****THERE ARE MORE THAN 24 CARDS BETWEEN 2 INITIAL STATEMENT CARDS.

APPENDIX H

SERIES 200 CHARACTER CODES

Key Punch	Card Code	Central Processor Code	Octal	High Speed Printer	Key Punch	Card Code	Central Processor Code	Octal	High Speed Printer
0	0	000000	00	0	$\bar{0}$ or -	X, 0 or X(1)	100000	40	-
1	1	000001	01	1	J	X, 1	100001	41	J
2	2	000010	02	2	K	X, 2	100010	42	K
3	3	000011	03	3	L	X, 3	100011	43	L
4	4	000100	04	4	M	X, 4	100100	44	M
5	5	000101	05	5	N	X, 5	100101	45	N
6	6	000110	06	6	O	X, 6	100110	46	O
7	7	000111	07	7	P	X, 7	100111	47	P
8	8	001000	10	8	Q	X, 8	101000	50	Q
9	9	001001	11	9	R	X, 9	101001	51	R
	8, 2	001010	12	'		X, 8, 2	101010	52	#
#	8, 3	001011	13	=	$	X, 8, 3	101011	53	$
@	8, 4	001100	14	:	*	X, 8, 4	101100	54	*
Space	Blank	001101	15	Blank		X, 8, 5	101101	55	"
	8, 6	001110	16	> (2)		X, 8, 6	101110	56	≠ (2)
	8, 7	001111	17	&	- or $\bar{0}$	X or X, 0(1)	101111	57	1/2 or ! (2)
$\overset{\&}{0}$ or &	R, 0 or R(1)	010000	20	+		8, 5*	110000	60	< (2)
A	R, 1	010001	21	A	/	0, 1	110001	61	/
B	R, 2	010010	22	B	S	0, 2	110010	62	S
C	R, 3	010011	23	C	T	0, 3	110011	63	T
D	R, 4	010100	24	D	U	0, 4	110100	64	U
E	R, 5	010101	25	E	V	0, 5	110101	65	V
F	R, 6	010110	26	F	W	0, 6	110110	66	W
G	R, 7	010111	27	G	X	0, 7	110111	67	X
H	R, 8	011000	30	H	Y	0, 8	111000	70	Y
I	R, 9	011001	31	I	Z	0, 9	111001	71	Z
	R, 8, 2	011010	32	;		0, 8, 2	111010	72	@
.	R, 8, 3	011011	33	.	,	0, 8, 3	111011	73	,
□	R, 8, 4	011100	34	)	%	0, 8, 4	111100	74	(
	R, 8, 5	011101	35	%		0, 8, 5	111101	75	C_R
	R, 8, 6	011110	36	■		0, 8, 6	111110	76	□ (2)
& or $\overset{\&}{0}$	R or R, 0(1)	011111	37	? (2)		0, 8, 7	111111	77	¢ (2)

(1) Special Code. This card code-central processor code equivalency is effective when control character 26 is coded in a card read or punch PCB instruction.

(2) Indicates symbol which will be printed by a printer which has a 63-character drum (Types 122 and 222 printers).

ACCURACY OF CALCULATIONS, C-2
ACTION
      OPERATOR ACTION IN UNPROGRAMMED HALT OR LOOPING, 9-7
ADDITIONAL
    " LANGUAGE FEATURES, B-1
    " LIST PAIRS, 4-11
    " STATEMENTS, B-1
ADDRESS
    " FIELDS,
          RELOCATION OF ADDRESS FIELDS, E-4
    " LENGTHS,
          DETERMINATION OF ADDRESS LENGTHS IN REGION 3,
                E-4
    " MODE,
          NOTES ON THE FOUR-CHARACTER ADDRESS MODE, E-9
          THREE-CHARACTER AND FOUR-CHARACTER ADDRESS
                MODES, 11-1
      REGION 3 ADDRESS INTERPRETATION, E-3
AIDS
      DEBUGGING AIDS, 8-10
ALLOCATION OF WORK TAPES, 9-9
ALOG, D-13
ALOG10, D-15
ALPHABETIC
    " CONVERSION,
          BASIC FIELD SPECIFICATION FOR ALPHABETIC
                CONVERSION, 5-31
    " DATA, 1-15
          HOLLERITH AND ALPHABETIC DATA, C-4
          HOLLERITH, OCTAL AND ALPHABETIC DATA, 1-14
          INPUT OF ALPHABETIC DATA, 5-32
ALTER CARD
      *ALTER CARD, 7-9
ALTERNATE CREATION OF VARIABLE FORMATS, 5-53
AREA
      COMMON AREA,
          USE OF DUMMY ARRAY TO SPACE OVER COMMON AREA,
                4-5
ARITHMETIC
    " AND LOGICAL EXPRESSIONS AND STATEMENTS, 2-1
      ERRORS IN ARITHMETIC AND LOGICAL EXPRESSIONS, G-2
    " EXPRESSIONS, 2-1
          EXAMPLES OF ARITHMETIC EXPRESSIONS, 2-2
    " IF STATEMENT, 3-3
    " OPERATION SYMBOLS, 2-1
    " OPERATIONS,
          HIERARCHY OF ARITHMETIC OPERATIONS, 2-2
    " STATEMENTS, 2-4
ARRAY
      ARRAYS, ARRAY ELEMENTS, AND SUBSCRIPTING, 1-10
      DUMMY ARRAY,
          USE OF DUMMY ARRAY TO SPACE OVER COMMON AREA,
                4-5
    " ELEMENTS,
          ARRAYS, ARRAY ELEMENTS, AND SUBSCRIPTING, 1-10
      ONE-DIMENSIONAL ARRAY, STORAGE SEQUENCE OF ELEMENTS,
          1-10
      RIGHT-HAND PORTION OF ARRAY TO BE INITIALIZED, 4-13
      SHORT-LIST NOTATION FOR INPUT/OUTPUT OF ENTIRE
          ARRAYS, 5-7
      TWO-DIMENSIONAL ARRAY, STORAGE SEQUENCE OF ELEMENTS,
          1-10
ASA PROPOSED FORTRAN
      COMPARISON WITH ASA PROPOSED FORTRAN, B-1
ASSIGN STATEMENTS
      ASSIGNED GO TO AND ASSIGN STATEMENTS, 3-2
ASSIGNED GO TO AND ASSIGN STATEMENTS, 3-2
ASSIGNMENTS
      PERIPHERAL DEVICE ASSIGNMENTS, 7-4
ATAN, D-15
ATAN2, D-18
BACKSPACE STATEMENT, 5-56
BASIC FIELD SPECIFICATION
    " FOR ALPHABETIC CONVERSION, 5-31
    " FOR INTEGER CONVERSION, 5-18
    " FOR LOGICAL CONVERSION, 5-29
    " FOR OCTAL CONVERSION, 5-28
BATCHED JOB PROCESSING
      GO-LATER - BATCHED JOB PROCESSING, 8-4
BCD TAPES, F-1
      ORGANIZATION OF BCD TAPES, F-7
BINARY
      *BINARY AND END CARDS, 7-9
    " PROGRAM TAPES, F-1
    " PROGRAM UNITS,
          JOBS CONTAINING *GET AND *BINARY PROGRAM UNITS,
                8-7
          (CONT.)

BINARY (CONT.)
    " RUN TAPE,
          BINARY RUN TAPES, F-1
          ORGANIZATION OF THE BINARY RUN TAPE, F-6
    " TAPE INPUT,
          I/O LISTS USED WITH BINARY TAPE INPUT OR OUTPUT,
                5-11
    " TAPES, F-1
          ORGANIZATION OF BINARY TAPES, F-7
BIT REPRESENTATION, C-1
BLANK
    " CHARACTERS, 1-2
    " CONVERSION,
          FIELD SPECIFICATION FOR BLANK CONVERSION, 5-36
BLOCKS
      LABELED BLOCKS,
          COMMON STATEMENT FOR THREE LABELED BLOCKS, 4-4
BYPASSING EXECUTION, 8-9
CALCULATIONS
      ACCURACY OF CALCULATIONS, C-2
CALL
    " CHAIN STATEMENT, 3-8
    " OPTIONS,
          CONSOLE CALL OPTIONS, 9-3, 9-4
      STANDARD CONSOLE CALL, 9-1
    " STATEMENT, 3-7
          ERRORS IN IF OR CALL STATEMENTS OR THE USE OF
                FUNCTIONS OR SUBROUTINES, G-5
CALLING SEQUENCES, E-5
CARD
      *ALTER CARD, 7-9
      *CHAIN CARD, 7-9
      *DIAG CARD, 7-7
      *DUMP CARD, 7-11
      *ENDATA CARD, 7-10
      *GET CARD, 7-8
      *JOBID CARD, 7-2
      *SCREEN CARD, 7-7
    " AND TAPE INPUT OPTION, 9-12
      COMMENT CARDS, 7-11
      CONTROL CARDS FOR STANDARD OPERATION, 7-1
      END CARDS,
          *BINARY AND END CARDS, 7-9
    " INPUT,
          LISTING OF CARD INPUT TO SCREEN, 8-23
      JOB-LEVEL CONTROL CARDS,
          RUN-LEVEL AND JOB-LEVEL CONTROL CARDS, 7-1
      SAMPLE *JOBID CARD WITH OPTIONS, 7-6
      SAMPLE INPUT CARD, 5-36
      SYSTEM CONTROL CARDS, 7-1
      1EOF CARD,
          *DATA CARD AND 1EOF CARD, 7-10
CARD-IMAGE TAPE INPUT, 7-11
CARRIAGE CONTROL
    " FOR PRINTER OUTPUT, 5-37
    " IN MULTIPLE-RECORD FORMAT STATEMENTS, 5-40
    " IN SINGLE-RECORD FORMAT STATEMENTS, 5-39
CATEGORIES
      FORTRAN STATEMENT CATEGORIES, 1-5
    " OF PROCEDURES, 6-1
CHAIN
    " CARD,
          *CHAIN CARD, 7-9
    " STATEMENT,
          CALL CHAIN STATEMENT, 3-8
CHAINING, 10-5
    " A LOAD-AND-GO JOB, 8-4
      EXAMPLE OF CHAINING, 10-6
CHAINS
      JOB DIVIDED INTO TWO CHAINS OF PROGRAM UNITS, 8-4
CHANGE IN PARAMETER, B-2
CHARACTER
    " CODES,
          SERIES 200 CHARACTER CODES, H-1
    " SET,
          FORTRAN CHARACTER SET, 1-2
          HONEYWELL CHARACTER SET, 1-2
CHARACTERISTICS
      COMPILER CHARACTERISTICS AND LIMITATIONS, 10-3
    " OF FUNCTIONS, 6-2
    " OF THE SUBROUTINE SUBPROGRAM, 6-11
      STATEMENT CHARACTERISTICS, 1-5
CHARACTERS
      BLANK CHARACTERS, 1-2
CODED HALTS DURING FORTRAN RUNS, 9-6
CODES
      CONVERSION CODES, 5-16
          (CONT.)

CODES (CONT.)
SERIES 200 CHARACTER CODES, H-1
CODING
" FORM,
FORTRAN CODING FORM, 1-3
" FORMAT,
SOURCE PROGRAM CODING FORMAT, 1-4
COMMENT CARDS, 7-11
COMMON
" AREA,
USE OF DUMMY ARRAY TO SPACE OVER COMMON AREA, 4-5
" INPUT DEVICE, 8-8
" INPUT TAPE, F-2
" OUTPUT AND COMMON PUNCH DEVICE OPTIONS, 8-8
" OUTPUT TAPE, F-2
" PUNCH DEVICE OPTIONS,
COMMON OUTPUT AND COMMON PUNCH DEVICE OPTIONS, 8-8
" REGION,
ILLEGAL EXTENSION OF COMMON REGION, 4-7
LEGAL EXTENSION OF COMMON REGION, 4-7
" STATEMENT, 4-2
COMMON STATEMENT FOR THREE LABELED BLOCKS, 4-4
" STORAGE,
USE OF COMMON STORAGE, E-6
COMMUNICATION
USE OF COMMUNICATION, E-6
" VIA POSITIONAL CORRESPONDENCE, 4-4
COMPARISON WITH ASA PROPOSED FORTRAN, B-1
COMPILATION TIME
MEMORY MAP (COMPILATION TIME), F-2, F-9
COMPILED PROGRAM UNITS
JOB CONTAINING PREVIOUSLY COMPILED PROGRAM UNITS, 8-8
COMPILER
" CHARACTERISTICS AND LIMITATIONS, 10-3
" ERROR MESSAGES, G-7
" MONITOR ERROR MESSAGES, G-12
PROCEDURES AND ROUTINES SUPPLIED WITH THE COMPILER, D-1
" SYSTEM TAPE, F-1
CREATING A COMPILER SYSTEM TAPE (CST), 9-14
PROCEDURES AND EXECUTION ROUTINES ON THE COMPILER SYSTEM TAPE, D-1
" SYSTEM TAPE ORGANIZATION, F-2
COMPONENTS
EXAMPLE HIGHLIGHTING DIFFERENT COMPONENTS OF FORMAT STATEMENT, 5-14
COMPUTED GO TO, 3-2
CONDITION TEST SUBROUTINES
I/O CONDITION TEST SUBROUTINES, 6-13
CONDITIONS
LIBRARY ERROR CONDITIONS, D-4
CONFIGURATION
MINIMUM EQUIPMENT CONFIGURATION FOR GO-LATER EXECUTION, 9-13
MINIMUM EQUIPMENT CONFIGURATION FOR LOAD-AND-GO OPERATION, 9-8
CONSIDERATIONS
GENERAL PROGRAMMING CONSIDERATIONS, 10-1
CONSOLE CALL
" OPTIONS, 9-3, 9-4
STANDARD CONSOLE CALL, 9-1
CONSTANT
CONSTANTS, C-5, 1-9
EXAMPLES OF CONSTANTS, 1-9
FIELD - REPETITION CONSTANT, 5-39
PROGRAM CONSTANTS,
ERRORS IN CONSTRUCTION OF PROGRAM CONSTANTS, G-4
CONSTRUCTION
ERRORS IN CONSTRUCTION AND USE OF SUBSCRIPTS, G-2
ERRORS IN CONSTRUCTION OF PROGRAM CONSTANTS, G-4
CONTENTS
" OF THE FIELD SPECIFICATION, 5-15
CONTINUATION LINE
USE OF CONTINUATION LINE WITH HOLLERITH SPECIFICATION, 5-35
CONTINUE STATEMENT, 3-8
CONTROL
" AND I/O STATEMENT ERRORS, G-5
" CARDS,
CONTROL CARDS FOR STANDARD OPERATION, 7-1
RUN-LEVEL AND JOB-LEVEL CONTROL CARDS, 7-1
SYSTEM CONTROL CARDS, 7-1
CARRIAGE CONTROL FOR PRINTER OUTPUT, 5-37
CARRIAGE CONTROL IN MULTIPLE-RECORD FORMAT (CONT.)
CONTROL (CONT.)
STATEMENTS, 5-40
CARRIAGE CONTROL IN SINGLE-RECORD FORMAT STATEMENTS, 5-39
ILLEGAL TRANSFERS OF CONTROL, 3-7
LEGAL TRANSFERS OF CONTROL, 3-7
" STATEMENTS, 3-1
CONVERSION
ALPHABETIC CONVERSION,
BASIC FIELD SPECIFICATION FOR ALPHABETIC CONVERSION, 5-31
BLANK CONVERSION,
FIELD SPECIFICATION FOR BLANK CONVERSION, 5-36
" CODES, 5-16
EFFECTS OF SCALE FACTOR ON INPUT VALUES (F CONVERSION), 5-43
EFFECTS OF SCALE FACTOR ON OUTPUT VALUES (E CONVERSION), 5-44
EFFECTS OF SCALE FACTOR ON OUTPUT VALUES (F CONVERSION), 5-43
" FIELD WIDTH, 5-16
INPUT FOR CONVERSION OF ALL REAL DATA, 5-21
INTEGER CONVERSION,
BASIC FIELD SPECIFICATION FOR INTEGER CONVERSION, 5-18
LOGICAL CONVERSION,
BASIC FIELD SPECIFICATION FOR LOGICAL CONVERSION, 5-29
OCTAL CONVERSION,
BASIC FIELD SPECIFICATION FOR OCTAL CONVERSION, 5-28
OUTPUT CONVERSION TO EXPLICIT EXPONENT (EW.D), 5-25
OUTPUT CONVERSION TO FIXED-POINT DECIMALS (FW.D), 5-24
" PROCEDURE,
OCTAL-DECIMAL CONVERSION PROCEDURE, A-1
SCREEN CONVERSION, 8-19
SCREEN CONVERSION OF I/O STATEMENTS, 8-21
SCREEN CONVERSION OF LIBRARY FUNCTION NAMES, 8-21
" TABLE,
OCTAL-DECIMAL CONVERSION TABLE, A-1
" TECHNIQUES, 10-8
COS, D-7
CREATING A COMPILER SYSTEM TAPE (CST), 9-14
CREATION
ALTERNATE CREATION OF VARIABLE FORMATS, 5-53
CST
CREATING A COMPILER SYSTEM TAPE (CST), 9-14
DATA
ALPHABETIC DATA, 1-15
HOLLERITH AND ALPHABETIC DATA, C-4
HOLLERITH, OCTAL AND ALPHABETIC DATA, 1-14
INPUT OF ALPHABETIC DATA, 5-32
" CARD,
*DATA CARD AND 1EOF CARD, 7-10
" FIELDS AND FIELD WIDTHS, 5-17
FORMATS OF DATA IN MEMORY AT OBJECT TIME, C-2
HOLLERITH DATA, 1-14
EXAMPLE OF OUTPUT OF HOLLERITH DATA, 5-34
FIELD SPECIFICATION FOR HOLLERITH DATA, 5-31
" INITIALIZATION STATEMENT, 4-9
INTEGER DATA, C-3, 1-12
INPUT OF INTEGER DATA, 5-19
OUTPUT OF INTEGER DATA, 5-20
LOGICAL DATA, C-5, 1-14
INPUT OF LOGICAL DATA, 5-30
" NAMES, 1-8
OCTAL DATA, C-4, 1-15
OCTAL DATUM,
INPUT OF OCTAL DATUM, 5-29
READ DATA, 1-13
REAL DATA, C-2
INPUT FOR CONVERSION OF ALL REAL DATA, 5-21
INPUT OF REAL DATA, 5-23
OUTPUT OF REAL DATA IN EXPONENTIAL FORMAT, 5-26
STORAGE OF OTHER DATA, C-2
" TYPES, 1-11
DATA-TYPE STATEMENTS, 4-7
DEBUGGING AIDS, 8-10
DECIMAL
" POINT,
SCALE FACTOR SHIFTING OF DECIMAL POINT, 5-42
" POSITION INDICATOR, 5-18
DECIMALS
FIXED-POINT DECIMALS,
OUTPUT CONVERSION TO FIXED-POINT DECIMALS (FW.D), 5-24
DECK (CONT.)

DECK
    INPUT DECK FOR DIAGNOSTIC PREPROCESSING, 8-17
    INPUT DECK FOR LOAD-AND-GO, 8-3
    INPUT DECK FOR SCREEN, 8-20
    INPUT DECK FOR STANDARD OPERATION, 7-1
    SAMPLE INPUT DECK TO EXECUTE GO-LATER TAPE, 8-6
    SAMPLE INPUT DECK TO WRITE GO-LATER TAPE, 8-5
DEFINING LOGICAL RELATIONS
    RELATIONAL OPERATORS DEFINING LOGICAL RELATIONS, 1-7
DEFINITION, 2-4
    " AND EVALUATION, 2-1
    PROGRAM DEFINITIONS, 1-1
DELIMITERS, 1-8
DESCRIPTION
    SYSTEM DESCRIPTION, 8-1
DEVICE
    " ASSIGNMENTS,
        PERIPHERAL DEVICE ASSIGNMENTS, 7-4
    COMMON INPUT DEVICE, 8-8
    " OPTIONS,
        COMMON OUTPUT AND COMMON PUNCH DEVICE OPTIONS, 8-8
DIAG
    " CARD,
        *DIAG CARD, 7-7
    PREPROCESS-ONLY OPTION - *DIAG, 8-16
    PREPROCESSING TO TAPE - *DIAG, T, 8-17
DIAGNOSTIC
    " PREPROCESSING, 8-16
        FLOW DIAGRAM FOR DIAGNOSTIC PREPROCESSING - PREPROCESS-ONLY OPTION, 8-17
        INPUT DECK FOR DIAGNOSTIC PREPROCESSING, 8-17
    " PREPROCESSOR FLOW DIAGRAM - TAPE OPTION, 8-18
    " PREPROCESSOR LISTING, 8-19
DIAGNOSTICS, 8-18
    ERROR DIAGNOSTICS, 8-15
DIAGRAM
    DIAGNOSTIC PREPROCESSOR FLOW DIAGRAM - TAPE OPTION, 8-18
    FLOW DIAGRAM FOR DIAGNOSTIC PREPROCESSING - PREPROCESS-ONLY OPTION, 8-17
    FLOW DIAGRAM TO EXECUTE GO-LATER TAPE, 8-6
    FLOW DIAGRAM TO WRITE GO-LATER MULTIJOB TAPE, 8-5
    " OF SYSTEM OPTIONS, 8-9
    STANDARD LOAD-AND-GO FLOW DIAGRAM, 8-3
DIFFERENT COMPONENTS
    EXAMPLE HIGHLIGHTING DIFFERENT COMPONENTS OF FORMAT STATEMENT, 5-14
DIMENSION STATEMENT, 4-1
DIVIDED
    JOB DIVIDED INTO TWO CHAINS OF PROGRAM UNITS, 8-4
DO
    " LOOPS,
        IMPLIED DO LOOPS, 4-11
        LEGAL AND ILLEGAL NESTING OF DO LOOPS, 3-6
        LISTS WITH IMPLIED DO LOOPS, 5-7
        NESTED PAIRS OF IMPLIED DO LOOPS, 4-12
    " STATEMENT, 3-4
        DO STATEMENT AND ITS RANGE, 3-5
DSA STATEMENTS
    REGION-DEFINING DSA STATEMENTS, E-2
DUMMY ARRAY
    USE OF DUMMY ARRAY TO SPACE OVER COMMON AREA, 4-5
DUMP CARD
    *DUMP CARD, 7-11
DUMPING SUBROUTINES
    DYNAMIC DUMPING SUBROUTINES, 6-14
DUMPS
    MEMORY DUMPS, 8-16
DYNAMIC DUMPING SUBROUTINES, 6-14
EASYCODER SYMBOLIC PROGRAM UNITS, E-1
EFFECTS
    " OF SCALE FACTOR ON INPUT VALUES (F CONVERSION), 5-43
    " OF SCALE FACTOR ON OUTPUT VALUES (E CONVERSION), 5-44
    " OF SCALE FACTOR ON OUTPUT VALUES (F CONVERSION), 5-43
ELEMENTS
    ARRAYS, ARRAY ELEMENTS, AND SUBSCRIPTING, 1-10
    ONE-DIMENSIONAL ARRAY, STORAGE SEQUENCE OF ELEMENTS, 1-10
    TWO-DIMENSIONAL ARRAY, STORAGE SEQUENCE OF ELEMENTS, 1-10
EMERGENCY RESTART OPTION, 9-12
END
    " CARDS,
        *BINARY AND END CARDS, 7-9
        (CONT.)
END (CONT.)
    " FILE STATEMENT, 5-55
    " STATEMENT, 3-9
ENDATA CARD
    *ENDATA CARD, 7-10
EQUIPMENT
    " CONFIGURATION,
        MINIMUM EQUIPMENT CONFIGURATION FOR GO-LATER EXECUTION, 9-13
        MINIMUM EQUIPMENT CONFIGURATION FOR LOAD-AND-GO OPERATION, 9-8
    " FOR LOAD-AND-GO OPERATING, 9-9
    " FOR SCREEN, 9-11
    LOAD-AND-GO EQUIPMENT, 9-8
    " REQUIREMENTS, 9-2
    " CONFIGURATION,
        SCREEN EQUIPMENT CONFIGURATION, 9-11
    " TO EXECUTE GO-LATER JOBS, 9-13
EQUIVALENCE STATEMENT, 4-5
EQUIVALENT
    TWENTY-ONE EQUIVALENT WAYS OF KEYPUNCHING AN EXPONENT OF PLUS TWO, 5-21
ERROR
    " CONDITIONS,
        LIBRARY ERROR CONDITIONS, D-4
    " DIAGNOSTICS, 8-15
    " MESSAGES, G-1
        COMPILER ERROR MESSAGES, G-7
        COMPILER MONITOR ERROR MESSAGES, G-12
        EXECUTION-TIME ERROR MESSAGES, G-13
        INPUT/OUTPUT ERROR MESSAGES, G-14
        LIBRARY FUNCTION ERROR MESSAGES, G-13
        PREPROCESSOR ERROR MESSAGES, G-1
        RUN-TAPE GENERATOR ERROR MESSAGES, G-13
        SCREEN ERROR MESSAGES, G-14
ERRORS
    FORMAT STATEMENT ERRORS, G-3
    I/O STATEMENT ERRORS,
        CONTROL AND I/O STATEMENT ERRORS, G-5
    " IN ARITHMETIC AND LOGICAL EXPRESSIONS, G-2
    " IN CONSTRUCTION AND USE OF SUBSCRIPTS, G-2
    " IN CONSTRUCTION OF PROGRAM CONSTANTS, G-4
    " IN IF OR CALL STATEMENTS OR THE USE OF FUNCTIONS OR SUBROUTINES, G-5
    " IN SPECIFICATION STATEMENTS, G-1
    KEYPUNCH AND OTHER MISCELLANEOUS ERRORS, G-6
    LIBRARY FUNCTION ERRORS AT EXECUTION TIME, D-4
    STATEMENT LABEL ERRORS, G-3
EASYCODER PROGRAM UNIT
    SAMPLE REGIONALIZED EASYCODER PROGRAM UNIT, E-8
EVALUATION
    DEFINITION AND EVALUATION, 2-1
    LOGICAL EVALUATION USING LOGICAL OPERATORS, 2-5
EW.D
    OUTPUT CONVERSION TO EXPLICIT EXPONENT (EW.D), 5-25
EXAMPLE
    EXAMPLES OF ARITHMETIC EXPRESSIONS, 2-2
    EXAMPLES OF CONSTANTS, 1-9
    " HIGHLIGHTING DIFFERENT COMPONENTS OF FORMAT STATEMENT, 5-14
    " OF CHAINING, 10-6
    " OF OUTPUT OF HOLLERITH DATA, 5-34
EXECUTE GO-LATER
    " JOBS,
        EQUIPMENT TO EXECUTE GO-LATER JOBS, 9-13
    " TAPE,
        FLOW DIAGRAM TO EXECUTE GO-LATER TAPE, 8-6
        SAMPLE INPUT DECK TO EXECUTE GO-LATER TAPE, 8-6
EXECUTING
    " A GO-LATER TAPE, 8-6, 9-12
EXECUTION
    BYPASSING EXECUTION, 8-9
    GO-LATER EXECUTION,
        MINIMUM EQUIPMENT CONFIGURATION FOR GO-LATER EXECUTION, 9-13
    " ROUTINES,
        PROCEDURES AND EXECUTION ROUTINES ON THE COMPILER SYSTEM TAPE, D-1
    " RUN,
        STARTING A GO-LATER EXECUTION RUN, 9-13
    " TIME,
        LIBRARY FUNCTION ERRORS AT EXECUTION TIME, D-4
        MEMORY MAP (EXECUTION TIME), F-2, F-10
EXECUTION-TIME ERROR MESSAGES, G-13
EXIT-TO-MONITOR SUBROUTINE, 6-14
EXP, D-8
EXPLICIT EXPONENT
        (CONT.)

EXPLICIT EXPONENT (CONT.)
    OUTPUT CONVERSION TO EXPLICIT EXPONENT (EW.D), 5-25
EXPONENT
    EXPLICIT EXPONENT,
        OUTPUT CONVERSION TO EXPLICIT EXPONENT (EW.D), 5-25
    TWENTY-ONE EQUIVALENT WAYS OF KEYPUNCHING AN EXPONENT OF PLUS TWO, 5-21
EXPONENTIAL FORMAT
    OUTPUT OF REAL DATA IN EXPONENTIAL FORMAT, 5-26
EXPRESSIONS
    ARITHMETIC EXPRESSIONS, 2-1
        EXAMPLES OF ARITHMETIC EXPRESSIONS, 2-2
    LOGICAL EXPRESSIONS, 2-4
        ARITHMETIC AND LOGICAL EXPRESSIONS AND STATEMENTS, 2-1
        ERRORS IN ARITHMETIC AND LOGICAL EXPRESSIONS, G-2
EXTENSION
    ILLEGAL EXTENSION OF COMMON REGION, 4-7
    LEGAL EXTENSION OF COMMON REGION, 4-7
EXTERNAL STATEMENT, 4-8
FACTOR
    SCALE FACTOR, 5-41
        EFFECTS OF SCALE FACTOR ON INPUT VALUES (F CONVERSION), 5-43
        EFFECTS OF SCALE FACTOR ON OUTPUT VALUES (E CONVERSION), 5-44
        EFFECTS OF SCALE FACTOR ON OUTPUT VALUES (F CONVERSION), 5-43
    " SHIFTING,
        SCALE FACTOR SHIFTING OF DECIMAL POINT, 5-42
FEATURES
    ADDITIONAL LANGUAGE FEATURES, B-1
    HARDWARE FEATURES,
        TEST SUBROUTINES FOR SIMULATED HARDWARE AND HARDWARE FEATURES, 6-12
FIELD
    " - REPETITION CONSTANT, 5-39
    ADDRESS FIELDS,
        RELOCATION OF ADDRESS FIELDS, E-4
    DATA FIELDS AND FIELD WIDTHS, 5-17
    " SPECIFICATION,
        BASIC FIELD SPECIFICATION FOR ALPHABETIC CONVERSION, 5-31
        BASIC FIELD SPECIFICATION FOR INTEGER CONVERSION, 5-18
        BASIC FIELD SPECIFICATION FOR LOGICAL CONVERSION, 5-29
        BASIC FIELD SPECIFICATION FOR OCTAL CONVERSION, 5-28
        CONTENTS OF THE FIELD SPECIFICATION, 5-15
        FIELD SPECIFICATION FOR BLANK CONVERSION, 5-36
        FIELD SPECIFICATION FOR HOLLERITH DATA, 5-31
        GENERALIZED FIELD SPECIFICATION, GW.D, 5-27
        REPETITION OF GROUPS OF FIELD SPECIFICATIONS, 5-40
    " SPECIFICATION FORMATS, 5-15
    " WIDTH,
        CONVERSION FIELD WIDTH, 5-16
        DATA FIELDS AND FIELD WIDTHS, 5-17
FILE STATEMENT
    END FILE STATEMENT, 5-55
FIXED-POINT
    " DECIMALS,
        OUTPUT CONVERSION TO FIXED-POINT DECIMALS (FW.D), 5-24
    " NUMBERS, C-1
FLOATING-POINT
    " NUMBERS, C-1
    " PACKAGES, D-4
    " PRECISION, 7-4
FLOW
    " DIAGRAM,
        DIAGNOSTIC PREPROCESSOR FLOW DIAGRAM - TAPE OPTION, 8-18
        FLOW DIAGRAM FOR DIAGNOSTIC PREPROCESSING - PREPROCESS-ONLY OPTION, 8-17
        FLOW DIAGRAM TO EXECUTE GO-LATER TAPE, 8-6
        FLOW DIAGRAM TO WRITE GO-LATER MULTIJOB TAPE, 8-5
        STANDARD LOAD-AND-GO FLOW DIAGRAM, 8-3
    SYSTEM FLOW OF SCREEN, 8-20
FORM
    FORTRAN CODING FORM, 1-3
    GENERAL FORM OF THE FORMAT STATEMENT, 5-11
    MULTIPLE-RECORD FORMS, 5-45
FORMAT (CONT.)

FORMAT
    EXPONENTIAL FORMAT,
        OUTPUT OF REAL DATA IN EXPONENTIAL FORMAT, 5-26
    FIELD SPECIFICATION FORMATS, 5-15
    FORMATS OF DATA IN MEMORY AT OBJECT TIME, C-2
    HANDLING VARIATIONS IN FORMAT AT OBJECT TIME, 5-54
    READING IN FORMAT AT OBJECT TIME, 5-51
    SOURCE PROGRAM CODING FORMAT, 1-4
    " STATEMENT, 5-11
        CARRIAGE CONTROL IN MULTIPLE-RECORD FORMAT STATEMENTS, 5-40
        CARRIAGE CONTROL IN SINGLE-RECORD FORMAT STATEMENTS, 5-39
        EXAMPLE HIGHLIGHTING DIFFERENT COMPONENTS OF FORMAT STATEMENT, 5-14
        GENERAL FORM OF THE FORMAT STATEMENT, 5-11
        PARENTHESIS LEVELS IN A FORMAT STATEMENT, 5-48
        RESCANNING A FORMAT STATEMENT, 5-49
    " STATEMENT ERRORS, G-3
    " STATEMENT SUMMARY, 5-12
    " TABLE, 10-3
    VARIABLE FORMATS,
        ALTERNATE CREATION OF VARIABLE FORMATS, 5-53
FORMATTING
    " FORTRAN STATEMENTS, 1-3
    SOURCE PROGRAM FORMATTING, 1-2
FORTRAN
    ASA PROPOSED FORTRAN,
        COMPARISON WITH ASA PROPOSED FORTRAN, B-1
    " CHARACTER SET, 1-2
    " CODING FORM, 1-3
    " PROCESSING,
        STANDARD FORTRAN PROCESSING - LOAD-AND-GO OPERATION, 8-2
    " PROGRAM UNITS,
        RELOCATION OF FORTRAN PROGRAM UNITS, E-5
    " RUN,
        CODED HALTS DURING FORTRAN RUNS, 9-6
        POSSIBLE HALTS DURING A FORTRAN RUN, 9-6
    " RUN OPTIONS, 9-1
    " STATEMENT CATEGORIES, 1-5
    " STATEMENT SUMMARY, B-3
    " STATEMENTS,
        FORMATTING FORTRAN STATEMENTS, 1-3
FOUR-CHARACTER ADDRESS MODE
    NOTES ON THE FOUR-CHARACTER ADDRESS MODE, E-9
    THREE-CHARACTER AND FOUR-CHARACTER ADDRESS MODES, 11-1
FUNCTION
    " ERROR MESSAGES,
        LIBRARY FUNCTION ERROR MESSAGES, G-13
    " ERRORS,
        LIBRARY FUNCTION ERRORS AT EXECUTION TIME, D-4
    " NAMES,
        SCREEN CONVERSION OF LIBRARY FUNCTION NAMES, 8-21
    " SUBPROGRAMS, 6-4
FUNCTIONS, 6-2
    CHARACTERISTICS OF FUNCTIONS, 6-2
    ERRORS IN IF OR CALL STATEMENTS OR THE USE OF FUNCTIONS OR SUBROUTINES, G-5
    LIBRARY FUNCTIONS, D-6, 6-7
    STATEMENT FUNCTIONS, 6-2
FW.D
    OUTPUT CONVERSION TO FIXED-POINT DECIMALS (FW.D), 5-24
GENERAL
    " FORM OF THE FORMAT STATEMENT, 5-11
    " PROGRAMMING CONSIDERATIONS, 10-1
GENERALIZED FIELD SPECIFICATION, GW.D, 5-27
GENERATOR ERROR MESSAGES
    RUN-TAPE GENERATOR ERROR MESSAGES, G-13
GET
    " CARD,
        *GET CARD, 7-8
    JOBS CONTAINING *GET AND *BINARY PROGRAM UNITS, 8-7
GLOSSARY, B-12
GO
    ASSIGNED GO TO AND ASSIGN STATEMENTS, 3-2
    COMPUTED GO TO, 3-2
    UNCONDITIONAL GO TO, 3-1
GO-LATER
    " - BATCHED JOB PROCESSING, 8-4
    " EXECUTION,
        MINIMUM EQUIPMENT CONFIGURATION FOR GO-LATER EXECUTION, 9-13
    " EXECUTION RUN,
        (CONT.)

GO-LATER (CONT.)
STARTING A GO-LATER EXECUTION RUN, 9-13
" JOBS,
EQUIPMENT TO EXECUTE GO-LATER JOBS, 9-13
" MULTIJOB TAPE,
FLOW DIAGRAM TO WRITE GO-LATER MULTIJOB TAPE, 8-5
" TAPE,
EXECUTING A GO-LATER TAPE, 8-6, 9-12
FLOW DIAGRAM TO EXECUTE GO-LATER TAPE, 8-6
SAMPLE INPUT DECK TO EXECUTE GO-LATER TAPE, 8-6
SAMPLE INPUT DECK TO WRITE GO-LATER TAPE, 8-5
WRITING A GO-LATER TAPE, 8-4, 9-11
" TAPE RUN,
STARTING AND TERMINATING A WRITE GO-LATER TAPE RUN, 9-12
GROUPS
REPETITION OF GROUPS OF FIELD SPECIFICATIONS, 5-40
GW.D
GENERALIZED FIELD SPECIFICATION, GW.D, 5-27
HALT
CODED HALTS DURING FORTRAN RUNS, 9-6
POSSIBLE HALTS DURING A FORTRAN RUN, 9-6
UNPROGRAMMED HALT,
OPERATOR ACTION IN UNPROGRAMMED HALT OR LOOPING, 9-7
UNPROGRAMMED HALTS AND LOOPING, 9-6
HANDLING VARIATIONS IN FORMAT AT OBJECT TIME, 5-54
HARDWARE
" FEATURES,
TEST SUBROUTINES FOR SIMULATED HARDWARE AND HARDWARE FEATURES, 6-12
SIMULATED HARDWARE,
TEST SUBROUTINES FOR SIMULATED HARDWARE AND HARDWARE FEATURES, 6-12
HIERARCHY
" OF ARITHMETIC OPERATIONS, 2-2
" OF LOGICAL OPERATIONS, 2-6
HIGHLIGHTING DIFFERENT COMPONENTS
EXAMPLE HIGHLIGHTING DIFFERENT COMPONENTS OF FORMAT STATEMENT, 5-14
HOLLERITH
" OCTAL AND ALPHABETIC DATA, 1-14
" AND ALPHABETIC DATA, C-4
" DATA, 1-14
EXAMPLE OF OUTPUT OF HOLLERITH DATA, 5-34
FIELD SPECIFICATION FOR HOLLERITH DATA, 5-31
" SPECIFICATION,
USE OF CONTINUATION LINE WITH HOLLERITH SPECIFICATION, 5-35
I/O
" CONDITION TEST SUBROUTINES, 6-13
" LISTS USED WITH BINARY TAPE INPUT OR OUTPUT, 5-11
" MODULES,
OBJECT I/O MODULES, D-5
" PROGRAMMING TIPS, 10-7
" STATEMENT ERRORS,
CONTROL AND I/O STATEMENT ERRORS, G-5
" STATEMENTS, 5-1
SCREEN CONVERSION OF I/O STATEMENTS, 8-21
" SUBROUTINE REREAD, 6-13
IEFN TABLE, 10-3
ILLEGAL
" EXTENSION OF COMMON REGION, 4-7
" NESTING,
LEGAL AND ILLEGAL NESTING OF DO LOOPS, 3-6
" TRANSFERS OF CONTROL, 3-7
IMPLIED DO LOOPS, 4-11
LISTS WITH IMPLIED DO LOOPS, 5-7
NESTED PAIRS OF IMPLIED DO LOOPS, 4-12
INDICATOR
DECIMAL POSITION INDICATOR, 5-18
INITIALIZATION STATEMENT
DATA INITIALIZATION STATEMENT, 4-9
INITIALIZED
RIGHT-HAND PORTION OF ARRAY TO BE INITIALIZED, 4-13
INPUT, 5-18, 5-28, 5-29, 5-31, 5-35, 5-36, 5-42
BINARY TAPE INPUT,
I/O LISTS USED WITH BINARY TAPE INPUT OR OUTPUT, 5-11
" CARD,
SAMPLE INPUT CARD, 5-36
CARD INPUT,
LISTING OF CARD INPUT TO SCREEN, 8-23
CARD-IMAGE TAPE INPUT, 7-11
" DECK,
INPUT DECK FOR DIAGNOSTIC PREPROCESSING, 8-17
(CONT.)

INPUT (CONT.)
INPUT DECK FOR LOAD-AND-GO, 8-3
INPUT DECK FOR SCREEN, 8-20
INPUT DECK FOR STANDARD OPERATION, 7-1
SAMPLE INPUT DECK TO EXECUTE GO-LATER TAPE, 8-6
SAMPLE INPUT DECK TO WRITE GO-LATER TAPE, 8-5
" DEVICE,
COMMON INPUT DEVICE, 8-8
" FOR CONVERSION OF ALL REAL DATA, 5-21
" OF ALPHABETIC DATA, 5-32
" OF INTEGER DATA, 5-19
" OF LOGICAL DATA, 5-30
" OF OCTAL DATUM, 5-29
" OF REAL DATA, 5-23
" OPTION,
CARD AND TAPE INPUT OPTION, 9-12
" TAPE,
COMMON INPUT TAPE, F-2
INPUT TAPE (T5), 9-10
TAPE INPUT, 7-6
" VALUES,
EFFECTS OF SCALE FACTOR ON INPUT VALUES (F CONVERSION), 5-43
INPUT/OUTPUT
" ERROR MESSAGES, G-14
" LISTS, 5-5
SHORT-LIST NOTATION FOR INPUT/OUTPUT OF ENTIRE ARRAYS, 5-7
" STATEMENTS, 5-1
INTEGER
" CONVERSION,
BASIC FIELD SPECIFICATION FOR INTEGER CONVERSION, 5-18
" DATA, C-3, 1-12
OUTPUT OF INTEGER DATA, 5-20
" PRECISION, 7-4
INTERNAL REPRESENTATION OF NUMBERS, C-1
JOB
" CONTAINING PREVIOUSLY COMPILED PROGRAM UNITS, 8-8
" DIVIDED INTO TWO CHAINS OF PROGRAM UNITS, 8-4
EXECUTE GO-LATER JOBS,
EQUIPMENT TO EXECUTE GO-LATER JOBS, 9-13
JOBS CONTAINING *GET AND *BINARY PROGRAM UNITS, 8-7
LOAD-AND-GO JOB,
CHAINING A LOAD-AND-GO JOB, 8-4
" NAME OPTION, 7-3
" PROCESSING,
GO-LATER - BATCHED JOB PROCESSING, 8-4
JOB-LEVEL CONTROL CARDS
RUN-LEVEL AND JOB-LEVEL CONTROL CARDS, 7-1
JOBID CARD
*JOBID CARD, 7-2
SAMPLE *JOBID CARD WITH OPTIONS, 7-6
KEYPUNCH AND OTHER MISCELLANEOUS ERRORS, G-6
KEYPUNCHING
TWENTY-ONE EQUIVALENT WAYS OF KEYPUNCHING AN EXPONENT OF PLUS TWO, 5-21
LABEL ERRORS
STATEMENT LABEL ERRORS, G-3
LABELED BLOCKS
COMMON STATEMENT FOR THREE LABELED BLOCKS, 4-4
LABELING STATEMENTS, 1-6
LANGUAGE
" FEATURES,
ADDITIONAL LANGUAGE FEATURES, B-1
" LIMITATIONS, 10-1
" SUMMARY, B-1
LAYOUTS
MEMORY LAYOUTS,
TAPE AND MEMORY LAYOUTS, F-1
LEGAL
" AND ILLEGAL NESTING OF DO LOOPS, 3-6
" EXTENSION OF COMMON REGION, 4-7
" TRANSFERS OF CONTROL, 3-7
LENGTHS
ADDRESS LENGTHS,
DETERMINATION OF ADDRESS LENGTHS IN REGION 3, E-4
LEVELS
PARENTHESIS LEVELS IN A FORMAT STATEMENT, 5-48
LIBRARY
" ERROR CONDITIONS, D-4
" FUNCTION ERROR MESSAGES, G-13
" FUNCTION ERRORS AT EXECUTION TIME, D-4
" FUNCTION NAMES,
SCREEN CONVERSION OF LIBRARY FUNCTION NAMES, 8-21
(CONT.)

LIBRARY (CONT.)
    " FUNCTIONS, D-6, 6-7
LIMITATIONS
        COMPILER CHARACTERISTICS AND LIMITATIONS, 10-3
        LANGUAGE LIMITATIONS, 10-1
        SOURCE PROGRAM SIZE LIMITATIONS, 10-2
LINE
        CONTINUATION LINE,
            USE OF CONTINUATION LINE WITH HOLLERITH
                SPECIFICATION, 5-35
LIST PAIRS
        ADDITIONAL LIST PAIRS, 4-11
LISTING
        DIAGNOSTIC PREPROCESSOR LISTING, 8-19
        MACHINE-CODE (PSEUDO-EASYCODER) LISTING, 8-12
    " OF CARD INPUT TO SCREEN, 8-23
    " OPTIONS, 7-5
        OUTPUT LISTING FROM SCREEN, 8-24
        PSEUDO-EASYCODER LISTING, 8-13
        SOURCE-PROGRAM LISTING, 8-10, 8-11
LISTS
        I/O LISTS USED WITH BINARY TAPE INPUT OR OUTPUT,
            5-11
        INPUT/OUTPUT LISTS, 5-5
        SIMPLE LISTS, 5-5
    " WITH IMPLIED DO LOOPS, 5-7
LOAD-AND-GO
    " EQUIPMENT, 9-8
    " FLOW DIAGRAM,
            STANDARD LOAD-AND-GO FLOW DIAGRAM, 8-3
        INPUT DECK FOR LOAD-AND-GO, 8-3
    " JOB,
            CHAINING A LOAD-AND-GO JOB, 8-4
    " OPERATING,
            EQUIPMENT FOR LOAD-AND-GO OPERATING, 9-9
    " OPERATION,
            MINIMUM EQUIPMENT CONFIGURATION FOR LOAD-AND-GO
                OPERATION, 9-8
            STANDARD FORTRAN PROCESSING - LOAD-AND-GO
                OPERATION, 8-2
    " RUN,
            LOAD-AND-GO RUN, 9-8
            LOAD-AND-GO RUN WITH SYSTEM OPTIONS, 8-9
LOADER-MONITORS
        TAPE LOADER-MONITORS, 9-2
LOGICAL
    " CONVERSION,
            BASIC FIELD SPECIFICATION FOR LOGICAL
                CONVERSION, 5-29
    " DATA, C-5, 1-14
            INPUT OF LOGICAL DATA, 5-30
    " EVALUATION USING LOGICAL OPERATORS, 2-5
    " EXPRESSIONS, 2-4
            ARITHMETIC AND LOGICAL EXPRESSIONS AND
                STATEMENTS, 2-1
            ERRORS IN ARITHMETIC AND LOGICAL EXPRESSIONS,
                G-2
    " IF STATEMENT, 3-3
    " OPERATIONS,
            HIERARCHY OF LOGICAL OPERATIONS, 2-6
    " OPERATORS, 1-7, 2-5
            LOGICAL EVALUATION USING LOGICAL OPERATORS, 2-5
    " RELATIONS,
            RELATIONAL OPERATORS DEFINING LOGICAL RELATIONS,
                1-7
    " STATEMENTS, 2-6
LOOPING
        OPERATOR ACTION IN UNPROGRAMMED HALT OR LOOPING, 9-7
        UNPROGRAMMED HALTS AND LOOPING, 9-6
LOOPS
        DO LOOPS,
            LEGAL AND ILLEGAL NESTING OF DO LOOPS, 3-6
        IMPLIED DO LOOPS, 4-11
            LISTS WITH IMPLIED DO LOOPS, 5-7
            NESTED PAIRS OF IMPLIED DO LOOPS, 4-12
MACHINE-CODE (PSEUDO-EASYCODER) LISTING, 8-12
MAP
        MEMORY MAP (COMPILATION TIME), F-2, F-9
        MEMORY MAP (EXECUTION TIME), F-2, F-10
        OBJECT MEMORY MAP, 8-11, 8-12
        RELOCATABLE MEMORY MAP, 8-11
MEMORY
    " DUMPS, 8-16
        FORMATS OF DATA IN MEMORY AT OBJECT TIME, C-2
    " LAYOUTS,
            TAPE AND MEMORY LAYOUTS, F-1
    " MAP,
            (CONT.)

MEMORY (CONT.)
            MEMORY MAP (COMPILATION TIME), F-2, F-9
            MEMORY MAP (EXECUTION TIME), F-2, F-10
            OBJECT MEMORY MAP, 8-11, 8-12
            RELOCATABLE MEMORY MAP, 8-11
    " SIZE, 7-3
MESSAGES
        COMPILER ERROR MESSAGES, G-7
        COMPILER MONITOR ERROR MESSAGES, G-12
        ERROR MESSAGES, G-1
        EXECUTION-TIME ERROR MESSAGES, G-13
        INPUT/OUTPUT ERROR MESSAGES, G-14
        LIBRARY FUNCTION ERROR MESSAGES, G-13
        PREPROCESSOR ERROR MESSAGES, G-1
        RUN-TAPE GENERATOR ERROR MESSAGES, G-13
        SCREEN ERROR MESSAGES, G-14
MINIMUM EQUIPMENT CONFIGURATION
    " FOR GO-LATER EXECUTION, 9-13
    " FOR LOAD-AND-GO OPERATION, 9-8
MODE
        FOUR-CHARACTER ADDRESS MODE,
            NOTES ON THE FOUR-CHARACTER ADDRESS MODE, E-9
        FOUR-CHARACTER ADDRESS MODES,
            THREE-CHARACTER AND FOUR-CHARACTER ADDRESS
                MODES, 11-1
MODULES
        OBJECT I/O MODULES, D-5
        SYSTEM MODULES, 8-1
MONITOR ERROR MESSAGES
        COMPILER MONITOR ERROR MESSAGES, G-12
MULTIJOB TAPE
        FLOW DIAGRAM TO WRITE GO-LATER MULTIJOB TAPE, 8-5
MULTIPLE-RECORD
    " FORMAT STATEMENTS,
            CARRIAGE CONTROL IN MULTIPLE-RECORD FORMAT
                STATEMENTS, 5-40
    " FORMS, 5-45
NAME OPTION
        JOB NAME OPTION, 7-3
NAMES, 1-8
        DATA NAMES, 1-8
        LIBRARY FUNCTION NAMES,
            SCREEN CONVERSION OF LIBRARY FUNCTION NAMES,
                8-21
NAMING AND TYPING PROCEDURES, 6-1
NESTED PAIRS OF IMPLIED DO LOOPS, 4-12
NESTING
        ILLEGAL NESTING,
            LEGAL AND ILLEGAL NESTING OF DO LOOPS, 3-6
NON-STANDARD OPERATION AND OPTIONS, 7-2
NOTATION
        SHORT-LIST NOTATION FOR INPUT/OUTPUT OF ENTIRE
            ARRAYS, 5-7
NUMBERS
        FIXED-POINT NUMBERS, C-1
        FLOATING-POINT NUMBERS, C-1
        INTERNAL REPRESENTATION OF NUMBERS, C-1
OBJECT
    " I/O MODULES, D-5
    " MEMORY MAP, 8-11, 8-12
    " TAPES, F-1
    " TIME,
            FORMATS OF DATA IN MEMORY AT OBJECT TIME, C-2
            HANDLING VARIATIONS IN FORMAT AT OBJECT TIME,
                5-54
            READING IN FORMAT AT OBJECT TIME, 5-51
OCTAL
    " CONVERSION,
            BASIC FIELD SPECIFICATION FOR OCTAL CONVERSION,
                5-28
    " DATA, C-4, 1-15
        HOLLERITH, OCTAL AND ALPHABETIC DATA, 1-14
OCTAL-DECIMAL CONVERSION
    " PROCEDURE, A-1
    " TABLE, A-1
ONE-DIMENSIONAL ARRAY, STORAGE SEQUENCE OF ELEMENTS, 1-10
OPERATING
        LOAD-AND-GO OPERATING,
            EQUIPMENT FOR LOAD-AND-GO OPERATING, 9-9
    " PROCEDURES, 9-1
OPERATION
        ARITHMETIC OPERATIONS,
            HIERARCHY OF ARITHMETIC OPERATIONS, 2-2
        LOAD-AND-GO OPERATION,
            MINIMUM EQUIPMENT CONFIGURATION FOR LOAD-AND-GO
                OPERATION, 9-8
        LOGICAL OPERATIONS,
            (CONT.)

OPERATION (CONT.)
        HIERARCHY OF LOGICAL OPERATIONS, 2-6
    NON-STANDARD OPERATION AND OPTIONS, 7-2
    STANDARD FORTRAN PROCESSING - LOAD-AND-GO OPERATION, 8-2
    STANDARD OPERATION,
        CONTROL CARDS FOR STANDARD OPERATION, 7-1
        INPUT DECK FOR STANDARD OPERATION, 7-1
    " SYMBOLS,
        ARITHMETIC OPERATION SYMBOLS, 2-1
OPERATOR ACTION IN UNPROGRAMMED HALT OR LOOPING, 9-7
OPERATORS, 1-7
    " DEFINING LOGICAL RELATIONS,
        RELATIONAL OPERATORS DEFINING LOGICAL RELATIONS, 1-7
    LOGICAL OPERATORS, 1-7, 2-5
        LOGICAL EVALUATION USING LOGICAL OPERATORS, 2-5
    RELATIONAL OPERATORS, 2-6
OPTION
    COMMON PUNCH DEVICE OPTIONS,
        COMMON OUTPUT AND COMMON PUNCH DEVICE OPTIONS, 8-8
    CONSOLE CALL OPTIONS, 9-3, 9-4
    DIAGNOSTIC PREPROCESSOR FLOW DIAGRAM - TAPE OPTION, 8-18
    EMERGENCY RESTART OPTION, 9-12
    FLOW DIAGRAM FOR DIAGNOSTIC PREPROCESSING - PREPROCESS-ONLY OPTION, 8-17
    FORTRAN RUN OPTIONS, 9-1
    JOB NAME OPTION, 7-3
    LISTING OPTIONS, 7-5
    NON-STANDARD OPERATION AND OPTIONS, 7-2
    PREPROCESS-ONLY OPTION - *DIAG, 8-16
    PUNCH OPTION, 7-5, 8-7
    RUN OPTIONS, 8-1
    SAMPLE *JOBID CARD WITH OPTIONS, 7-6
    SAVE OPTION, 7-5
    SYSTEM OPTIONS, 8-7
        DIAGRAM OF SYSTEM OPTIONS, 8-9
        LOAD-AND-GO RUN WITH SYSTEM OPTIONS, 8-9
    TAPE INPUT OPTION,
        CARD AND TAPE INPUT OPTION, 9-12
ORGANIZATION
    COMPILER SYSTEM TAPE ORGANIZATION, F-2
    " OF BCD TAPES, F-7
    " OF BINARY TAPES, F-7
    " OF THE BINARY RUN TAPE, F-6
OUTPUT, 5-19, 5-29, 5-30, 5-31, 5-33, 5-37, 5-42
    COMMON OUTPUT AND COMMON PUNCH DEVICE OPTIONS, 8-8
    " CONVERSION,
        OUTPUT CONVERSION TO EXPLICIT EXPONENT (EW.D), 5-25
        OUTPUT CONVERSION TO FIXED-POINT DECIMALS (FW.D), 5-24
    EXAMPLE OF OUTPUT OF HOLLERITH DATA, 5-34
    I/O LISTS USED WITH BINARY TAPE INPUT OR OUTPUT, 5-11
    " LISTING FROM SCREEN, 8-24
    " OF INTEGER DATA, 5-20
    " OF REAL DATA IN EXPONENTIAL FORMAT, 5-26
    PRINTER OUTPUT,
        CARRIAGE CONTROL FOR PRINTER OUTPUT, 5-37
    " TAPE,
        COMMON OUTPUT TAPE, F-2
    " TO TAPE, 9-10
    " VALUES,
        EFFECTS OF SCALE FACTOR ON OUTPUT VALUES (E CONVERSION), 5-44
        EFFECTS OF SCALE FACTOR ON OUTPUT VALUES (F CONVERSION), 5-43
PACKAGES
    FLOATING-POINT PACKAGES, D-4
PAIRS
    ADDITIONAL LIST PAIRS, 4-11
    NESTED PAIRS OF IMPLIED DO LOOPS, 4-12
PARAMETER
    CHANGE IN PARAMETER, 8-2
PARENTHESIS LEVELS IN A FORMAT STATEMENT, 5-48
PAUSE STATEMENT, 3-8
PERIPHERAL DEVICE ASSIGNMENTS, 7-4
PLUS
    TWENTY-ONE EQUIVALENT WAYS OF KEYPUNCHING AN EXPONENT OF PLUS TWO, 5-21
POINT
    DECIMAL POINT,
        SCALE FACTOR SHIFTING OF DECIMAL POINT, 5-42
PORTION
        (CONT.)

PORTION (CONT.)
    RIGHT-HAND PORTION OF ARRAY TO BE INITIALIZED, 4-13
POSITION INDICATOR
    DECIMAL POSITION INDICATOR, 5-18
POSITIONAL
    COMMUNICATION VIA POSITIONAL CORRESPONDENCE, 4-4
PRECISION
    FLOATING-POINT PRECISION, 7-4
    INTEGER PRECISION, 7-4
PREPROCESS-ONLY OPTION
    " - *DIAG, 8-16
    FLOW DIAGRAM FOR DIAGNOSTIC PREPROCESSING - PREPROCESS-ONLY OPTION, 8-17
PREPROCESSING
    DIAGNOSTIC PREPROCESSING, 8-16
        FLOW DIAGRAM FOR DIAGNOSTIC PREPROCESSING - PREPROCESS-ONLY OPTION, 8-17
        INPUT DECK FOR DIAGNOSTIC PREPROCESSING, 8-17
    " TO TAPE - *DIAG, T, 8-17
PREPROCESSOR
    " ERROR MESSAGES, G-1
    " FLOW DIAGRAM,
        DIAGNOSTIC PREPROCESSOR FLOW DIAGRAM - TAPE OPTION, 8-18
    " LISTING,
        DIAGNOSTIC PREPROCESSOR LISTING, 8-19
PRINTER OUTPUT
    CARRIAGE CONTROL FOR PRINTER OUTPUT, 5-37
PROCEDURE
    CATEGORIES OF PROCEDURES, 6-1
    OCTAL-DECIMAL CONVERSION PROCEDURE, A-1
    OPERATING PROCEDURES, 9-1
    PROCEDURES, 6-1
    PROCEDURES AND EXECUTION ROUTINES ON THE COMPILER SYSTEM TAPE, D-1
    PROCEDURES AND ROUTINES SUPPLIED WITH THE COMPILER, D-1
    PROGRAMMING PROCEDURES, E-5
    STARTING PROCEDURE, 9-2
    TYPING PROCEDURES,
        NAMING AND TYPING PROCEDURES, 6-1
PROCESSING
    GO-LATER - BATCHED JOB PROCESSING, 8-4
    STANDARD FORTRAN PROCESSING - LOAD-AND-GO OPERATION, 8-2
    SYSTEM PROCESSING, E-2
PROGRAM
    " CODING FORMAT,
        SOURCE PROGRAM CODING FORMAT, 1-4
    " CONSTANTS,
        ERRORS IN CONSTRUCTION OF PROGRAM CONSTANTS, G-4
    " DEFINITIONS, 1-1
    " FORMATTING,
        SOURCE PROGRAM FORMATTING, 1-2
    SAMPLE PROGRAM, E-7
    " SIZE LIMITATIONS,
        SOURCE PROGRAM SIZE LIMITATIONS, 10-2
    " STATEMENTS,
        SEQUENCE OF PROGRAM STATEMENTS, 1-6
    " STRING,
        SIZE OF PROGRAM STRING, 10-3
    " SUMMARY,
        SOURCE PROGRAM SUMMARY, 1-1
    " TAPE,
        BINARY PROGRAM TAPES, F-1
        SYMBOLIC PROGRAM TAPE, F-1
    " UNIT,
        EASYCODER SYMBOLIC PROGRAM UNITS, E-1
        JOB CONTAINING PREVIOUSLY COMPILED PROGRAM UNITS, 8-8
        JOB DIVIDED INTO TWO CHAINS OF PROGRAM UNITS, 8-4
        JOBS CONTAINING *GET AND *BINARY PROGRAM UNITS, 8-7
        RELOCATION OF FORTRAN PROGRAM UNITS, E-5
        SAMPLE REGIONALIZED EASYCODER PROGRAM UNIT, E-8
PROGRAMMING
    " CONSIDERATIONS,
        GENERAL PROGRAMMING CONSIDERATIONS, 10-1
    " PROCEDURES, E-5
    " TIPS,
        I/O PROGRAMMING TIPS, 10-7
PROPOSED FORTRAN
    COMPARISON WITH ASA PROPOSED FORTRAN, B-1
PSEUDO-EASYCODER
    " LISTING, 8-13
    MACHINE-CODE (PSEUDO-EASYCODER) LISTING, 8-12
PUNCH (CONT.)

PUNCH
    " DEVICE OPTIONS,
        COMMON OUTPUT AND COMMON PUNCH DEVICE OPTIONS, 8-8
    " OPTION, 7-5, 8-7
RANGE
    DO STATEMENT AND ITS RANGE, 3-5
READ
    " DATA, 1-13
    " STATEMENT, 5-1
READING IN FORMAT AT OBJECT TIME, 5-51
REAL DATA, C-2
    INPUT FOR CONVERSION OF ALL REAL DATA, 5-21
    INPUT OF REAL DATA, 5-23
    OUTPUT OF REAL DATA IN EXPONENTIAL FORMAT, 5-26
RECORD
    UNIT RECORD,
        DEFINITION OF A UNIT RECORD, 5-46
REGION
    COMMON REGION,
        ILLEGAL EXTENSION OF COMMON REGION, 4-7
        LEGAL EXTENSION OF COMMON REGION, 4-7
    DETERMINATION OF ADDRESS LENGTHS IN REGION 3, E-4
    " 3 ADDRESS INTERPRETATION, E-3
REGION-DEFINING DSA STATEMENTS, E-2
REGIONALIZATION, E-1
REGIONALIZED EASYCODER PROGRAM UNIT
    SAMPLE REGIONALIZED ESAYCODER PROGRAM UNIT, E-8
RELATIONAL OPERATORS, 2-6
    " DEFINING LOGICAL RELATIONS, 1-7
RELATIONS
    RELATIONAL OPERATORS DEFINING LOGICAL RELATIONS, 1-7
RELOCATABLE MEMORY MAP, 8-10, 8-11
RELOCATION
    " OF ADDRESS FIELDS, E-4
    " OF FORTRAN PROGRAM UNITS, E-5
REPETITION
    " CONSTANT,
        FIELD - REPETITION CONSTANT, 5-39
    " OF GROUPS OF FIELD SPECIFICATIONS, 5-40
REPRESENTATION
    BIT REPRESENTATION, C-1
    INTERNAL REPRESENTATION OF NUMBERS, C-1
REQUIREMENTS
    EQUIPMENT REQUIREMENTS, 9-2
REREAD
    I/O SUBROUTINE REREAD, 6-13
RESCANNING A FORMAT STATEMENT, 5-49
RESTART OPTION
    EMERGENCY RESTART OPTION, 9-12
RESTRICTIONS, B-2, E-7
RETURN STATEMENT, 3-8
REWIND STATEMENT, 5-56
RIGHT-HAND PORTION OF ARRAY TO BE INITIALIZED, 4-13
ROUTINES
    EXECUTION ROUTINES,
        PROCEDURES AND EXECUTION ROUTINES ON THE COMPILER SYSTEM TAPE, D-1
    " SUPPLIED,
        PROCEDURES AND ROUTINES SUPPLIED WITH THE COMPILER, D-1
RUN
    FORTRAN RUN,
        POSSIBLE HALTS DURING A FORTRAN RUN, 9-6
    FORTRAN RUNS,
        CODED HALTS DURING FORTRAN RUNS, 9-6
    GO-LATER EXECUTION RUN,
        STARTING A GO-LATER EXECUTION RUN, 9-13
    LOAD-AND-GO RUN, 9-8
    LOAD-AND-GO RUN WITH SYSTEM OPTIONS, 8-9
    " OPTIONS, 8-1
        FORTRAN RUN OPTIONS, 9-1
    SCREEN RUN, 9-10
    " TAPE,
        BINARY RUN TAPES, F-1
        ORGANIZATION OF THE BINARY RUN TAPE, F-6
    TERMINATING A RUN, 9-3
    WRITE GO-LATER TAPE RUN,
        STARTING AND TERMINATING A WRITE GO-LATER TAPE RUN, 9-12
RUN-LEVEL AND JOB-LEVEL CONTROL CARDS, 7-1
RUN-TAPE GENERATOR ERROR MESSAGES, G-13
SAMPLE
    " JOBID CARD,
        SAMPLE *JOBID CARD WITH OPTIONS, 7-6
    " PROGRAM, E-7
    " REGIONALIZED EASYCODER PROGRAM UNIT, E-8
SAMPLE INPUT (CONT.)
SAMPLE INPUT
    " CARD, 5-36
    " DECK,
        SAMPLE INPUT DECK TO EXECUTE GO-LATER TAPE, 8-6
        SAMPLE INPUT DECK TO WRITE GO-LATER TAPE, 8-5
SAVE OPTION, 7-5
SAVING SPACE
    TIPS FOR SAVING SPACE AND TIME, 10-4
SCALE FACTOR, 5-41
    EFFECTS OF SCALE FACTOR ON INPUT VALUES (F CONVERSION), 5-43
    EFFECTS OF SCALE FACTOR ON OUTPUT VALUES (F CONVERSION), 5-44
    EFFECTS OF SCALE FACTOR ON OUTPUT VALUES (F CONVERSION), 5-43
    " SHIFTING OF DECIMAL POINT, 5-42
SCREEN
    " CARD,
        *SCREEN CARD, 7-7
    " CONVERSION, 8-19
        SCREEN CONVERSION OF I/O STATEMENTS, 8-21
        SCREEN CONVERSION OF LIBRARY FUNCTION NAMES, 8-21
    EQUIPMENT FOR SCREEN, 9-11
    " EQUIPMENT CONFIGURATION, 9-11
    " ERROR MESSAGES, G-14
    INPUT DECK FOR SCREEN, 8-20
    LISTING OF CARD INPUT TO SCREEN, 8-23
    OUTPUT LISTING FROM SCREEN, 8-24
    " RUN, 9-10
    SYSTEM FLOW OF SCREEN, 8-20
SEQUENCE
    CALLING SEQUENCES, E-5
    " OF PROGRAM STATEMENTS, 1-6
    ONE-DIMENSIONAL ARRAY, STORAGE SEQUENCE OF ELEMENTS, 1-10
    TWO-DIMENSIONAL ARRAY, STORAGE SEQUENCE OF ELEMENTS, 1-10
SERIES 200 CHARACTER CODES, H-1
SET
    CHARACTER SET,
        HONEYWELL CHARACTER SET, 1-2
    FORTRAN CHARACTER SET, 1-2
SHIFTING
    SCALE FACTOR SHIFTING OF DECIMAL POINT, 5-42
SHORT-LIST NOTATION FOR INPUT/OUTPUT OF ENTIRE ARRAYS, 5-7
SIMPLE LISTS, 5-5
SIMULATED HARDWARE
    TEST SUBROUTINES FOR SIMULATED HARDWARE AND HARDWARE FEATURES, 6-12
SIN, D-6
SINGLE-RECORD FORMAT STATEMENTS
    CARRIAGE CONTROL IN SINGLE-RECORD FORMAT STATEMENTS, 5-39
SIZE
    " LIMITATIONS,
        SOURCE PROGRAM SIZE LIMITATIONS, 10-2
    MEMORY SIZE, 7-3
    " OF PROGRAM STRING, 10-3
SONFIGURATION
    SCREEN EQUIPMENT CONFIGURATION, 9-11
SORT, D-10
SOURCE PROGRAM
    " CODING FORMAT, 1-4
    " FORMATTING, 1-2
    " SIZE LIMITATIONS, 10-2
    " SUMMARY, 1-1
SOURCE TABLE, 10-2
SOURCE-PROGRAM LISTING, 8-10, 8-11
SPACE
    SAVING SPACE,
        TIPS FOR SAVING SPACE AND TIME, 10-4
    USE OF DUMMY ARRAY TO SPACE OVER COMMON AREA, 4-5
SPECIAL SUBROUTINES, 6-12
SPECIFICATION
    BASIC FIELD SPECIFICATION FOR ALPHABETIC CONVERSION, 5-31
    BASIC FIELD SPECIFICATION FOR INTEGER CONVERSION, 5-18
    BASIC FIELD SPECIFICATION FOR LOGICAL CONVERSION, 5-29
    BASIC FIELD SPECIFICATION FOR OCTAL CONVERSION, 5-28
    FIELD SPECIFICATION,
        CONTENTS OF THE FIELD SPECIFICATION, 5-15
    FIELD SPECIFICATION FOR BLANK CONVERSION, 5-36
    FIELD SPECIFICATION FOR HOLLERITH DATA, 5-31
    FIELD SPECIFICATIONS,
        (CONT.)

SPECIFICATION (CONT.)
REPETITION OF GROUPS OF FIELD SPECIFICATIONS, 5-40
" FORMATS,
FIELD SPECIFICATION FORMATS, 5-15
GENERALIZED FIELD SPECIFICATION, GW.D, 5-27
HOLLERITH SPECIFICATION,
USE OF CONTINUATION LINE WITH HOLLERITH SPECIFICATION, 5-35
" STATEMENTS, 4-1
ERRORS IN SPECIFICATION STATEMENTS, G-1
STACK TAPE, F-2, 8-7
" (TF), 9-9
STANDARD
" CONSOLE CALL, 9-1
" FORTRAN PROCESSING - LOAD-AND-GO OPERATION, 8-2
" LOAD-AND-GO FLOW DIAGRAM, 8-3
" OPERATION,
CONTROL CARDS FOR STANDARD OPERATION, 7-1
INPUT DECK FOR STANDARD OPERATION, 7-1
STARTING
" A GO-LATER EXECUTION RUN, 9-13
" AND TERMINATING A WRITE GO-LATER TAPE RUN, 9-12
" PROCEDURE, 9-2
STATEMENT
ADDITIONAL STATEMENTS, B-1
ARITHMETIC AND LOGICAL EXPRESSIONS AND STATEMENTS, 2-1
ARITHMETIC IF STATEMENT, 3-3
ARITHMETIC STATEMENTS, 2-4
ASSIGN STATEMENTS,
ASSIGNED GO TO AND ASSIGN STATEMENTS, 3-2
BACKSPACE STATEMENT, 5-56
CALL CHAIN STATEMENT, 3-8
CALL STATEMENT, 3-7
CALL STATEMENTS,
ERRORS IN IF OR CALL STATEMENTS OR THE USE OF FUNCTIONS OR SUBROUTINES, G-5
" CATEGORIES,
FORTRAN STATEMENT CATEGORIES, 1-5
" CHARACTERISTICS, 1-5
COMMON STATEMENT, 4-2
COMMON STATEMENT FOR THREE LABELED BLOCKS, 4-4
CONTINUE STATEMENT, 3-8
CONTROL STATEMENTS, 3-1
DATA INITIALIZATION STATEMENT, 4-9
DATA-TYPE STATEMENTS, 4-7
DIMENSION STATEMENT, 4-1
DO STATEMENT, 3-4
DO STATEMENT AND ITS RANGE, 3-5
END FILE STATEMENT, 5-55
END STATEMENT, 3-9
EQUIVALENCE STATEMENT, 4-5
" ERRORS,
CONTROL AND I/O STATEMENT ERRORS, G-5
FORMAT STATEMENT ERRORS, G-3
EXTERNAL STATEMENT, 4-8
FORMAT STATEMENT, 5-11
EXAMPLE HIGHLIGHTING DIFFERENT COMPONENTS OF FORMAT STATEMENT, 5-14
GENERAL FORM OF THE FORMAT STATEMENT, 5-11
PARENTHESIS LEVELS IN A FORMAT STATEMENT, 5-48
RESCANNING A FORMAT STATEMENT, 5-49
FORMATTING FORTRAN STATEMENTS, 1-3
" FUNCTIONS, 6-2
I/O STATEMENTS, 5-1
SCREEN CONVERSION OF I/O STATEMENTS, 8-21
INPUT/OUTPUT STATEMENTS, 5-1
" LABEL ERRORS, G-3
LABELING STATEMENTS, 1-6
LOGICAL IF STATEMENT, 3-3
LOGICAL STATEMENTS, 2-6
MULTIPLE-RECORD FORMAT STATEMENTS,
CARRIAGE CONTROL IN MULTIPLE-RECORD FORMAT STATEMENTS, 5-40
PAUSE STATEMENT, 3-8
PROGRAM STATEMENTS,
SEQUENCE OF PROGRAM STATEMENTS, 1-6
READ STATEMENT, 5-1
REGION-DEFINING DSA STATEMENTS, E-2
RETURN STATEMENT, 3-8
REWIND STATEMENT, 5-56
SINGLE-RECORD FORMAT STATEMENTS,
CARRIAGE CONTROL IN SINGLE-RECORD FORMAT STATEMENTS, 5-39
SPECIFICATION STATEMENTS, 4-1
ERRORS IN SPECIFICATION STATEMENTS, G-1
(CONT.)
STATEMENT (CONT.)
STATEMENTS, 1-3
STOP STATEMENT, 3-9
" SUMMARY,
FORMAT STATEMENT SUMMARY, 5-12
FORTRAN STATEMENT SUMMARY, B-3
TITLE STATEMENT, 4-9
WRITE STATEMENT, 5-3
STOP STATEMENT, 3-9
STORAGE
COMMON STORAGE,
USE OF COMMON STORAGE, E-6
" OF OTHER DATA, C-2
" SEQUENCE,
ONE-DIMENSIONAL ARRAY, STORAGE SEQUENCE OF ELEMENTS, 1-10
TWO-DIMENSIONAL ARRAY, STORAGE SEQUENCF OF ELEMENTS, 1-10
STRING
PROGRAM STRING,
SIZE OF PROGRAM STRING, 10-3
SUBPROGRAM
FUNCTION SUBPROGRAMS, 6-4
SUBPROGRAMS, 6-1
SUBROUTINE SUBPROGRAM,
CHARACTERISTICS OF THE SUBROUTINE SUBPROGRAM, 6-11
SUBROUTINE
DYNAMIC DUMPING SUBROUTINES, 6-14
ERRORS IN IF OR CALL STATEMENTS OR THE USE OF FUNCTIONS OR SUBROUTINES, G-5
EXIT-TO-MONITOR SUBROUTINE, 6-14
I/O CONDITION TEST SUBROUTINES, 6-13
" REREAD,
I/O SUBROUTINE REREAD, 6-13
SPECIAL SUBROUTINES, 6-12
" SUBPROGRAM,
CHARACTERISTICS OF THE SUBROUTINE SUBPROGRAM, 6-11
SUBROUTINES, 6-9
TEST SUBROUTINES FOR SIMULATED HARDWARE AND HARDWARE FEATURES, 6-12
SUBSCRIPTING
ARRAYS, ARRAY ELEMENTS, AND SUBSCRIPTING, 1-10
SUBSCRIPTS
ERRORS IN CONSTRUCTION AND USE OF SUBSCRIPTS, G-2
SUMMARY
FORMAT STATEMENT SUMMARY, 5-12
FORTRAN STATEMENT SUMMARY, B-3
LANGUAGE SUMMARY, B-1
SOURCE PROGRAM SUMMARY, 1-1
SYSTEM SUMMARY, 8-1
SUPPLIED
ROUTINES SUPPLIED,
PROCEDURES AND ROUTINES SUPPLIED WITH THE COMPILER, D-1
SYMBOLIC PROGRAM
" TAPE, F-1
" UNITS,
EASYCODER SYMBOLIC PROGRAM UNITS, E-1
SYMBOLS
ARITHMETIC OPERATION SYMBOLS, 2-1
SYNTAX, 1-7
SYSTEM
" CONTROL CARDS, 7-1
" DESCRIPTION, 8-1
" FLOW OF SCREEN, 8-20
" MODULES, 8-1
" OPTIONS, 8-7
DIAGRAM OF SYSTEM OPTIONS, 8-9
LOAD-AND-GO RUN WITH SYSTEM OPTIONS, 8-9
" PROCESSING, E-2
" SUMMARY, 8-1
" TAPE,
COMPILER SYSTEM TAPE, F-1
CREATING A COMPILER SYSTEM TAPE (CST), 9-14
PROCEDURES AND EXECUTION ROUTINES ON THE COMPILER SYSTEM TAPE, D-1
" TAPE ORGANIZATION,
COMPILER SYSTEM TAPE ORGANIZATION, F-2
TABLE
FORMAT TABLE, 10-3
IEFN TABLE, 10-3
OCTAL-DECIMAL CONVERSION TABLE, A-1
SOURCE TABLE, 10-2
TOKEN TABLE, 10-2
TANH, D-12
TAPE (CONT.)

TAPE
    " AND MEMORY LAYOUTS, F-1
    BCD TAPES, F-1
        ORGANIZATION OF BCD TAPES, F-7
    BINARY PROGRAM TAPES, F-1
    BINARY RUN TAPE,
        ORGANIZATION OF THE BINARY RUN TAPE, F-6
    BINARY RUN TAPES, F-1
    BINARY TAPES, F-1
        ORGANIZATION OF BINARY TAPES, F-7
    COMMON INPUT TAPE, F-2
    COMMON OUTPUT TAPE, F-2
    COMPILER SYSTEM TAPE, F-1
        CREATING A COMPILER SYSTEM TAPE (CST), 9-14
        PROCEDURES AND EXECUTION ROUTINES ON THE
            COMPILER SYSTEM TAPE, D-1
    EXECUTE GO-LATER TAPE,
        FLOW DIAGRAM TO EXECUTE GO-LATER TAPE, 8-6
        SAMPLE INPUT DECK TO EXECUTE GO-LATER TAPE, 8-6
    GO-LATER TAPE,
        EXECUTING A GO-LATER TAPE, 8-6, 9-12
        WRITING A GO-LATER TAPE, 8-4, 9-11
    " INPUT, 7-6
        CARD-IMAGE TAPE INPUT, 7-11
        I/O LISTS USED WITH BINARY TAPE INPUT OR OUTPUT,
            5-11
    " INPUT OPTION,
        CARD AND TAPE INPUT OPTION, 9-12
    INPUT TAPE (T5), 9-10
    " LOADER-MONITORS, 9-2
    OBJECT TAPES, F-1
    " OPTION,
        DIAGNOSTIC PREPROCESSOR FLOW DIAGRAM - TAPE
            OPTION, 8-18
    " ORGANIZATION,
        COMPILER SYSTEM TAPE ORGANIZATION, F-2
    OUTPUT TO TAPE, 9-10
    PREPROCESSING TO TAPE - *DIAG, T, 8-17
    " RUN,
        STARTING AND TERMINATING A WRITE GO-LATER TAPE
            RUN, 9-12
    STACK TAPE, F-2, 8-7
    STACK TAPE (TF), 9-9
    SYMBOLIC PROGRAM TAPE, F-1
    WORK TAPES,
        ALLOCATION OF WORK TAPES, 9-9
    WRITE GO-LATER MULTIJOB TAPE,
        FLOW DIAGRAM TO WRITE GO-LATER MULTIJOB TAPE,
            8-5
    WRITE GO-LATER TAPE,
        SAMPLE INPUT DECK TO WRITE GO-LATER TAPE, 8-5
TECHNIQUES
    CONVERSION TECHNIQUES, 10-8
TERMINATING
    " A RUN, 9-3
    STARTING AND TERMINATING A WRITE GO-LATER TAPE RUN,
        9-12
TERMINOLOGY, B-1
TEST SUBROUTINES
    " FOR SIMULATED HARDWARE AND HARDWARE FEATURES, 6-12
    I/O CONDITION TEST SUBROUTINES, 6-13
TF
    STACK TAPE (TF), 9-9
THREE-CHARACTER AND FOUR-CHARACTER ADDRESS MODES, 11-1
TIME
    EXECUTION TIME,
        LIBRARY FUNCTION ERRORS AT EXECUTION TIME, D-4
    MEMORY MAP (COMPILATION TIME), F-2, F-9
    MEMORY MAP (EXECUTION TIME), F-2, F-10
    OBJECT TIME,
        FORMATS OF DATA IN MEMORY AT OBJECT TIME, C-2
        HANDLING VARIATIONS IN FORMAT AT OBJECT TIME,
            5-54
        READING IN FORMAT AT OBJECT TIME, 5-51
    TIPS FOR SAVING SPACE AND TIME, 10-4
TIPS
    " FOR SAVING SPACE AND TIME, 10-4
    I/O PROGRAMMING TIPS, 10-7
TITLE STATEMENT, 4-9
TOKEN TABLE, 10-2
TRANSFERS
    ILLEGAL TRANSFERS OF CONTROL, 3-7
    LEGAL TRANSFERS OF CONTROL, 3-7
TWENTY-ONE EQUIVALENT WAYS OF KEYPUNCHING AN EXPONENT
        OF PLUS TWO, 5-21
TWO-DIMENSIONAL ARRAY, STORAGE SEQUENCE OF ELEMENTS, 1-10
TYPES
    DATA TYPES, 1-11
TYPING PROCEDURES
    NAMING AND TYPING PROCEDURES, 6-1
T5
    INPUT TAPE (T5), 9-10
UNCONDITIONAL GO TO, 3-1
UNIT
    BINARY PROGRAM UNITS,
        JOBS CONTAINING *GET AND *BINARY PROGRAM UNITS,
            8-7
    COMPILED PROGRAM UNITS,
        JOB CONTAINING PREVIOUSLY COMPILED PROGRAM
            UNITS, 8-8
    EASYCODER SYMBOLIC PROGRAM UNITS, E-1
    FORTRAN PROGRAM UNITS,
        RELOCATION OF FORTRAN PROGRAM UNITS, E-5
    PROGRAM UNITS,
        JOB DIVIDED INTO TWO CHAINS OF PROGRAM UNITS,
            8-4
    " RECORD,
        DEFINITION OF A UNIT RECORD, 5-46
    SAMPLE REGIONALIZED EASYCODER PROGRAM UNIT, E-8
UNPROGRAMMED HALT
    OPERATOR ACTION IN UNPROGRAMMED HALT OR LOOPING, 9-7
    UNPROGRAMMED HALTS AND LOOPING, 9-6
VALUES
    INPUT VALUES,
        EFFECTS OF SCALE FACTOR ON INPUT VALUES (F
            CONVERSION), 5-43
    OUTPUT VALUES,
        EFFECTS OF SCALE FACTOR ON OUTPUT VALUES (E
            CONVERSION), 5-44
        EFFECTS OF SCALE FACTOR ON OUTPUT VALUES (F
            CONVERSION), 5-43
VARIABLE FORMATS
    ALTERNATE CREATION OF VARIABLE FORMATS, 5-53
VARIABLES, C-5, 1-9
VARIATIONS
    HANDLING VARIATIONS IN FORMAT AT OBJECT TIME, 5-54
WIDTH
    CONVERSION FIELD WIDTH, 5-16
    FIELD WIDTHS,
        DATA FIELDS AND FIELD WIDTHS, 5-17
WORK TAPES
    ALLOCATION OF WORK TAPES, 9-9
WRITE GO-LATER
    " MULTIJOB TAPE,
        FLOW DIAGRAM TO WRITE GO-LATER MULTIJOB TAPE,
            8-5
    " TAPE,
        SAMPLE INPUT DECK TO WRITE GO-LATER TAPE, 8-5
    " TAPE RUN,
        STARTING AND TERMINATING A WRITE GO-LATER TAPE
            RUN, 9-12
WRITE STATEMENT, 5-3
WRITING
    " A GO-LATER TAPE, 8-4, 9-11
1EOF CARD
    *DATA CARD AND 1EOF CARD, 7-10
200 CHARACTER CODES
    SERIES 200 CHARACTER CODES, H-1

HONEYWELL EDP TECHNICAL PUBLICATIONS
USERS' REMARKS FORM

TITLE: SERIES 200 FORTRAN COMPILER D SOFTWARE MANUAL

DATED: JUNE, 1966

FILE NO: 123.1305.001D.2-027

ERRORS NOTED:

Fold

Cut Along Line

SUGGESTIONS FOR IMPROVEMENT:

Fold

FROM: NAME ______________________ DATE __________

COMPANY ______________________

TITLE ______________________

ADDRESS ______________________

____

FIRST CLASS
PERMIT NO. 39531
WELLESLEY HILLS
MASS.

Cut Along Line

BUSINESS REPLY MAIL

No postage stamp necessary if mailed in the United States

POSTAGE WILL BE PAID BY

HONEYWELL
ELECTRONIC DATA PROCESSING DIVISION
60 WALNUT STREET
WELLESLEY HILLS, MASS. 02181

ATT'N: TECHNICAL COMMUNICATIONS DEPARTMENT

Honeywell
ELECTRONIC DATA PROCESSING