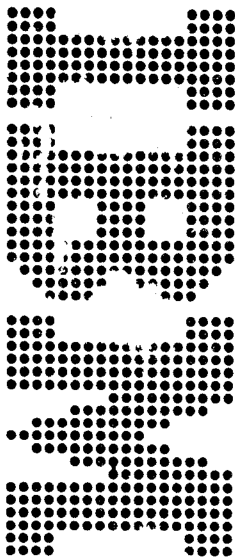
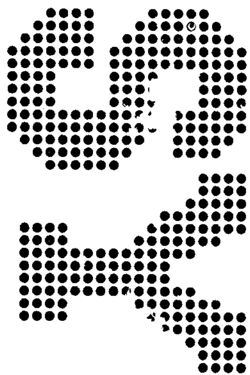
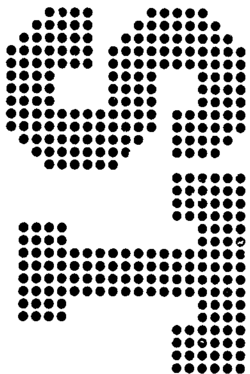
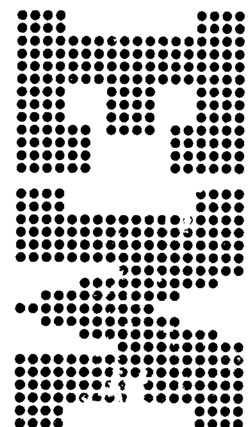





IBM System/3 FORTRAN IV Reference Manual

Program Numbers:

- 5702-FO1 (Model 10 Disk System)
- 5703-FO1 (Model 6)
- 5704-FO1 (Model 15)
- 5704-FO2 (Model 15)
- 5705-FO1 (Model 12)

SC28-6874-3
File No. S3-25

Program Product

Page of SC28-6874-3
Issued 29 September 1978
By TNL: SN21-5634

Fourth Edition (September 1976)

This is a major revision of, and obsoletes, SC28-6874-2 and Technical Newsletters SN21-5329 and SN21-5259. Changes are indicated by a vertical line at the left of the change.

This edition applies to the System/3 program versions listed below and to all subsequent versions and modification levels until otherwise indicated in new editions or technical newsletters:

System/3	Version	Modification	Program Number
Model 6	13	00	5703-FO1
Model 10	13	00	5702-FO1
Model 12	2	00	5705-FO1
Model 15	5	00	5704-FO1
Model 15	1	00	5704-FO2

Changes are continually made to the specifications herein; before using this publication in connection with the operation of IBM Systems, consult the latest *IBM System/3 Bibliography,* GC20-8080, for the editions that are applicable and current.

Use this publication only for the purposes stated in the *Preface.*

Publications are not stocked at the address below. Requests for copies of IBM publications and for technical information about the system should be made to your IBM representative or to the IBM branch office serving your locality.

This publication could contain technical inaccuracies or typographical errors. Use the Reader's Comment Form at the back of this publication to make comments about this publication. If the form has been removed, address your comments to IBM Corporation, Publications, Department 245, Rochester, Minnesota 55901. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

© Copyright International Business Machines Corporation 1972, 1974, 1976

Preface

This publication describes the System/3 FORTRAN IV language and the use of the System/3 FORTRAN IV processor to run programs written in the language.

This publication is intended to be used as a reference manual by persons writing programs in the System/3 FORTRAN IV language. You should have some knowledge of FORTRAN before using this publication. A useful source of this information is the set of programmed instruction texts describing the System/3 FORTRAN IV language, Order Numbers SR29-5015 through SR29-5019.

System/3 FORTRAN IV contains those features defined in American National Standard Basic FORTRAN X3.10-1966; language extensions supported by IBM 1130 Basic FORTRAN IV; and additional capabilities previously available only with certain IBM Full FORTRAN IV compilers.

The System/3 FORTRAN IV processor is a program product consisting of a compiler and a library of mathematical functions, service subprograms, and commercial subroutines.

The System/3 FORTRAN IV program product operates on the IBM System/3 Model 10 Disk System, IBM System/3 Model 6, IBM System/3 Model 12, and on the IBM System/3 Model 15.

The IBM System/3 Model 8 is supported by System/3 Model 10 Disk System control programming and System/3 Model 10 FORTRAN IV. The facilities described in this publication for the Model 10 are also applicable to the Model 8, even though the Model 8 is not referenced. Some of the devices discussed in this publication might not be applicable for the Model 8. Model 8 users should be familiar with the contents of *IBM System/3 Model 8 Introduction*, GC21-5114.

All references to the System/3 Model 15 in this publication apply to System/3 Models 15A, 15B, 15C, and 15D unless otherwise specified.

Direct Access Storage for Models 12 and 15

The IBM 3340 Direct Access Storage Facility attaches to System/3 Model 12 and to System/3 Models 15B, 15C, and 15D. Also, the IBM 3344 Direct Access Storage attaches to System/3 Model 15D. Certain areas on the 3340 and 3344 disks are treated as 5444 disks. These areas, known as 5444 simulation areas or simulation areas, are used for the program libraries and can also be used for data files. The remainder of the disk space, known as main data area, can only be used for data files.

References in this manual to 5444, 5445, and 3340 are to be interpreted according to which disk storage device(s) is (are) attached to the system. Use the following table to determine the meaning of the reference:

Reference	Model 15A Meaning	Models 12, 15B, 15C Meaning	Model 15D Meaning
5444	5444 Disk Storage Drive	5444 simulation area on 3340	Simulation area on 3340 or 3344
5445	5445 Disk Storage	Main data area on 3340	Main data area on 3340 or 3344
3340	Not applicable	Main data area on 3340	Main data area on 3340 or 3344

For further information, see the appropriate publications listed under *Related Publications.*

IBM System/3 5448 Disk Storage Drive

The IBM System/3 5448 Disk Storage Drive on System/3 Models 8 and 10 uses the same program product support as the IBM 5445 Disk Storage. However, a separate system control program feature is required for the 5448. In general, references to 5445 in this manual also apply to 5448. For specific information about 5448 operating characteristics and programming support, see the *IBM System/3 5448 Disk Storage Drive Program Reference Manual,* GC21-5168.

Related Publications

Publications containing information about the System/3 FORTRAN IV program product, System/3 Model 6, System/3 Model 8, System/3 Model 10 Disk System, System/3 Model 12, and System/3 Model 15 are:

Manuals	Order Number of Publication for				
	System/3 Model 6	System/3 Model 8	System/3 Model 10 Disk System	System/3 Model 12	System/3 Model 15
FORTRAN IV Commercial Subroutines	SC28-6875				
Introduction	GA21-9122	GC21-5114	GC21-7510	GC21-5116	GC21-5094
Operator's Guide	GC21-7501	GC21-7634	GC21-7508	GC21-5144	GC21-5075
System Generation Reference Manual	GC21-5126				GC21-7616
Halt Guide (System Messages)	GC21-7541	GC21-7540		GC21-5145	GC21-5076
System Control Programming Reference (Operation Control Language)	GC21-7516	GC21-7512		GC21-5130	5704-FO1 GC21-5077 5704-FO2:[1] GC21-5162
Overlay Linkage Editor Reference Manual	GC21-7561				
Disk Concepts and Planning Guide	GC21-7571				
User's Guide				GC21-5142	
Components Reference Manual	GA34-0001	GA21-9209	GA21-9103	GA21-9236	GA21-9193
3340 Reference					GC21-5111
3741 Reference Manual					5704-FO2 only GC21-5113

[1] Availability date for GC21-5162 manual is not the same as for this manual. Orders sent shortly after the issue date of this manual might be considered invalid.

Contents

HOW THIS MANUAL IS ORGANIZED 1

PART 1. SYSTEM/3 FORTRAN IV LANGUAGE . . . 3

CHAPTER 1. FORTRAN STATEMENTS 5
Coding FORTRAN Statements 6
Columns 1-5–Statement Number 7
Column 1–Comment Statement 7
Column 6–Continuation 7
Columns 7-72–FORTRAN Statement 7
Columns 73-80 (or 73-96)–Program Identification . . 7
Elements of the Language 8
Order of Statements in a FORTRAN Program 8

CHAPTER 2. CONSTANTS, VARIABLES, AND ARRAY ELEMENTS 9
Integer and Real Calculations 9
Data Types and Data Description 9
Operation Symbols 9
Constants 9
Integer Constants 10
Real Constants 10
Hexadecimal Constants 11
Literal Constants 12
Variables 12
Variable Names 12
Variable Types 13
Arrays 13
Arrangement of Arrays in Storage 14
Subscripts 14
Form of Subscripts 14

CHAPTER 3. ARITHMETIC ASSIGNMENT STATEMENTS AND EXPRESSIONS 15
Arithmetic Assignment Statements 15
Expressions 15
Rules for Constructing Expressions 15
Mode of an Arithmetic Statement 17

CHAPTER 4. CONTROL STATEMENTS 19
Unconditional GO TO Statement 19
Computed GO TO Statement 19
Relational IF Statement 20
Arithmetic IF Statement 21
DO Statement 21
CONTINUE Statement 23
PAUSE Statement 24
STOP Statement 24
PAUSE and STOP Algorithm 25
Character to Displacement 26
Decimal to Hex 26
END Statement 27

CHAPTER 5. FORMAT OF INPUT/OUTPUT 29
Format Statement 29
Various Forms of a Format Statement 30
Format Codes for Numeric Data 31
I Format Code (aIw) 31
F Format Code (aFw.d) 32
D and E Format Codes (aDw.d, aEw.d) 32
Scale Factor (nPc) 34
Format Codes for Alphameric Data 34
A Format Code (aAw) 34
H Format Code (wHstring) and Literals Enclosed in Apostrophes 35
Specifying Blank Fields in a Record (X Format Code) . . 36
Formatting the Record (T Format Code) 36
Lists for Transmission of Data 37
Indexing in Input/Output Lists 37
Implied DO Notation in Input/Output Lists 38
Additional Details of Input/Output Lists 38
Printed Output 39
Data Input to the Object Program 39
List-Directed Input Data 40
List-Directed Output Data 40

CHAPTER 6. INPUT/OUTPUT STATEMENTS . . . 41
SEQUENTIAL INPUT/OUTPUT STATEMENTS . . . 42
READ Statement 42
WRITE Statement 43
END FILE Statement 43
BACKSPACE Statement 43
REWIND Statement 44
DIRECT-ACCESS INPUT/OUTPUT STATEMENTS . . 45
Define FILE Statement 45
Direct-Access READ Statement 46
Direct-Access WRITE Statement 47
FIND Statement 48
General Example–Direct-Access Operations 49

CHAPTER 7. SPECIFICATION STATEMENTS . . . 51
TYPE Statements 51
IMPLICIT Statement 51
Explicit Specification Statements (INTEGER and REAL) 52
DIMENSION Statement 53
COMMON Statement 53
EQUIVALENCE Statement 56
Other Specification Statements 57
Data Initialization Statement 57

CHAPTER 8. SUBPROGRAMS 59
Naming Subprograms 59
Statement Functions 59
Function Definition 59
Function Reference 60
FUNCTION Subprograms 61
SUBROUTINE Subprograms 62
CALL Statement 63
RETURN Statement 64

Dummy Arguments in a Function or Subroutine Subprogram . . . 64
EXTERNAL Statement . . . 65
Automatic Function Selection . . . 66
GENERIC Statement . . . 66

CHAPTER 9. INTERPROGRAM COMMUNICATION . . . 69
PROGRAM Statement . . . 69
INVOKE Statement . . . 69
GLOBAL Statement . . . 70

CHAPTER 10. DEBUG FACILITY . . . 71
DEBUG Statement . . . 71
AT Statement . . . 72
TRACE ON Statement . . . 72
TRACE OFF . . . 73
Examples of the Debug Facility . . . 73

CHAPTER 11. EXAMPLES OF FORTRAN PROGRAMS . . . 75
Sample Program 1 . . . 75
Sample Program 2 . . . 76

PART 2. SYSTEM/3 FORTRAN IV USER'S GUIDE . . . 81

CHAPTER 12. OVERVIEW OF FORTRAN PROCESSING . . . 83
How a FORTRAN Program is Processed . . . 83
Using Operation Control Language (OCL) . . . 86
IBM System/3 FORTRAN-Supplied Procedures . . . 86
Libraries . . . 88
Compiler, Linkage Editor, and Load Module Output . . . 88
Defining Files . . . 88
Files Needed by the Compiler and Linkage Editor . . . 88
Files Needed by the FORTRAN Load Module . . . 88
Defining FORTRAN Files at Compilation Time . . . 88
Defining FORTRAN Files at Execution Time . . . 89
Logical Unit Numbers . . . 89

CHAPTER 13. COMPILATION . . . 93
Compiler Option Statements . . . 93
READ Device Option Statement . . . 94
PRINT and NOPRINTER Device Option Statements . . . 94
PUNCH Device Option Statement . . . 95
DAD40, DAD44, and DAD45 Device Option Statements . . . 95
SEQ40, SEQ44, and SEQ45 Device Option Statements . . . 96
TAPE Device Option Statement . . . 97
CORE Statement . . . 97
CATEGORY Statement . . . 98
*PROCESS Statement . . . 98
Batched Compilation . . . 102
Combining OCL and Compiler Option Statements . . . 102

CHAPTER 14. LINKAGE EDITOR PROCESSING . . . 105
Compiler Input to the Linkage Editor . . . 105
OCL Statements Needed for the Linkage Editor . . . 105
Linkage Editor Control Statements . . . 106
Linkage Editor Overlay Feature . . . 106

CHAPTER 15. LOAD MODULE EXECUTION . . . 107
OCL Statements Needed for Load Module Execution . . . 107
Program Data . . . 108
Combining Load Module OCL Statements with Compile Step Statements . . . 108

CHAPTER 16. JOB OUTPUT . . . 111
Object Module . . . 113
ESL Records . . . 113
RLD Records . . . 113
END Record . . . 114
Compiler Output Listing . . . 115
Messages . . . 115
Output from *PROCESS Options . . . 116
Object Module Card Deck . . . 118
Source Module Listing . . . 118
Storage Map . . . 119
Linkage Editor Output . . . 119
Load Module . . . 119
Linkage Editor Output Listing . . . 120
Output from Options . . . 120
Load Module Output . . . 124
Messages . . . 124
Program Output . . . 124
Using the FORTRAN Traceback Listing . . . 124

PART 3. PROGRAMMING CONSIDERATIONS . . . 129

CHAPTER 17. DIRECT-ACCESS PROGRAMMING CONSIDERATIONS . . . 131
Formatted I/O . . . 131
Unformatted I/O . . . 131
Sharing the Associated Variable Between Programs . . . 131
Minimizing Direct-Access I/O Time . . . 132
Buffer Assignment for Direct-Access Files . . . 132
Sharing Buffers . . . 134

CHAPTER 18. SEQUENTIAL DISK AND TAPE PROGRAMMING CONSIDERATIONS . . . 135
Formatted or List-Directed . . . 135
Unformatted I/O . . . 135
Buffer Assignment for Sequential Files . . . 138
Restrictions on the Order of Sequential I/O Operations . . . 138
Model 15 Multifile Tape Processing . . . 138
Tape Processing in Programs Using Overlays . . . 138

CHAPTER 19. FORTRAN IMPLEMENTATION CONSIDERATIONS . . . 139
Array Considerations . . . 139
Expression Considerations . . . 139
Real Number Considerations . . . 139
Assignment Statement Considerations . . . 139
Control Statement Considerations . . . 139
CALL Statement Considerations . . . 140
Subprogram Value Considerations . . . 140
Setting up a Job Stream for Programs Containing PROGRAM and INVOKE . . . 140
Separating Debug, DUMP, or PDUMP Output from Other Program Output . . . 140
Directing Program Output to Both a Printer and a Card Punch . . . 141
DEBUG Facility Considerations . . . 141
Assigning Names to User-Supplied Subprograms . . . 142

CHAPTER 20. SYSTEM CONSIDERATIONS . . . 145
Optimum Assignment of $WORK and $SOURCE Work Files . . . 145
Assignment of Work Files on One Disk . . . 145
Assignment of Work Files on Two Disks . . . 145
Model 15 Assignment of Work Files on 5445 or 3340 Disk Storage . . . 145
Linkage Between Modules Produced by System/3 Language Translators . . . 146
Standards . . . 148

Console Display Panel Dial Settings 148
Models 10 and 12 Dual Programming Considerations . . 149
Model 15 Considerations 149
Model 15 Spooled Environment and Multiprogramming 149
Model 15 CRT/Keyboard Support 149
Model 15 Double Buffering for Card Devices 150
Considerations/Restrictions 150
Differences Between 1130 and System/3 151
Unit Numbers 151
Device Options 152
Specifying the BCD Option 152
Read/Punch on the Same Card 152
Call Link 153
Associated Variables in Subroutines 153
Library Routines 153
Passing Arrays 154
Length Specification of Variables 154
Use of COMMON, EQUIVALENCE, and DEFINE FILE 154
Rounding 154
Passing Scalar Arguments to Subroutines 154
Forms Control 155
Commercial Subroutines 155
Decimal Data Format 155
A1 Data Format 155
Negative Zero 155
Number of Record Fields in the DEFINE FILE Statement 155

PART 4. REFERENCE 157

CHAPTER 21. FORTRAN STATEMENT REFERENCE . 159
Arithmetic Assignment Statement 159
Arithmetic IF Statement 159
AT Statement 160
BACKSPACE Statement 160
CALL Statement 160
COMMON Statement 160
Computed GO TO Statement 161
CONTINUE Statement 161
DATA Statement 161
DEBUG Statement 161
DEFINE FILE Statement 162
DIMENSION Statement 162
Direct-Access READ/WRITE Statement 162
DO Statement 162
END Statement 163
END FILE Statement 163
EQUIVALENCE Statement 163
Explicit Specification Statement 163
EXTERNAL Statement 163
FIND Statement 163
FORMAT Statement 164
Function Definition Statement 164
FUNCTION Statement 164
GENERIC Statement 164
GLOBAL Statement 164
GO TO Statement 165
IF Statement 165
IMPLICIT Statement 165
INTEGER Statement 165
INVOKE Statement 165
PAUSE Statement 165
PROGRAM Statement 166
READ Statement (Direct-Access) 166
READ Statement (Sequential) 166
REAL Statement 166
Relational IF Statement 167
RETURN Statement 167
REWIND Statement 167
Sequential READ/WRITE Statements 167
STOP Statement 167
SUBROUTINE Statement 167
TRACE OFF Statement 168
TRACE ON Statement 168
WRITE Statement (Direct-Access) 168
WRITE Statement (Sequential) 168

CHAPTER 22. SYSTEM/3 FORTRAN INTRINSIC AND EXTERNAL LIBRARY FUNCTIONS 169
Algorithms 169
Control of Program Exceptions in Mathematical Functions 173
Exponential Functions 174
EXP, REAL*4 Exponential Function (Module Names EXP,$FOMM,$FOMC) 174
DEXP, REAL*8 Exponential Function (Module Names DEXP,$FOMN,$FOMD) 174
Logarithmic Functions 175
ALOG, REAL*4 Natural Logarithm (Module Names ALOG,$FOM5,$FOMI) 175
ALOG10, REAL*4 Base 10 Logarithm (Module Names ALOG10,$FOMK,$FOM5) 176
DLOG, REAL*8 Natural Logarithm (Module Names DLOG,$FOMJ,$FOM6) 176
DLOG10, REAL*8 Base 10 Logarithm (Module Names DLOG10,$FOML,$FOM6) 176
Trigonometric Functions 177
SIN/COS, REAL*4 Sine/Cosine (Module Names SIN,COS,$FOM1,$FOM3) 177
ATAN, REAL*4 Arctangent (Module Names ATAN,$FOME) 178
DSIN/DCOS, REAL*8 Sine/Cosine (Module Names DSIN,DCOS,$FOM2,$FOM4) 179
DATAN, REAL*8 Arctangent (Module Names DATAN,$FOMF) 180
Square Root Functions 181
SQRT, REAL*4 Square Root (Module Names SQRT and $FOMG) 181
DSQRT, REAL*8 Square Root (Module Names DSQRT and $FOMH) 182
Hyperbolic Tangent Functions 183
TANH, REAL*4 Hyperbolic Tangent (Module Names TANH,$FOMO,$FOMM) 183
DTANH, REAL*8 Hyperbolic Tangent (Module Names DTANH,$FOMP,$FOMN) 183
Implicitly Invoked Exponentiation Subprograms . . . 184
$FOM7, Subprogram for I**J (Module Name $FOM7, Secondary Entry Point #FOM7) 184
$FOM8, Subprogram for A**J (Module Name $FOM8) 185
$FOM9, Subprogram for D**J (Module Name $FOM9) 185
$FOMA, Subprogram for A**B (Module Names $FOMA,$FOM5,$FOMC) 186
$FOMB, Subprogram for A**B (Module Names $FOMB,$FOM6,$FOMD) 186
$FOMC, REAL*4 Subprogram to Compute 2^x (Module Name $FOMC) 187
$FOMD, REAL*8 Subprogram to Compute 2^x (Module Name $FOMD) 188

CHAPTER 23. FORTRAN SERVICE SUBPROGRAMS . 189
Machine Indicator Test Subprograms 189
Pseudosense Light Subprogram (Entry Names: SLITE/SLITET) 189
Divide Check Subprogram (Entry Name: DVCHK) . . 189
Overflow Indicator Subprogram (Entry Name: OVERFL) 189
Utility Subprograms 190
End Execution Subprogram (Entry Name: EXIT) . . 190
Storage Dump Subprogram (Entry Names: DUMP/PDUMP) 190
Transaction Logging Subprogram (SUBR81) 5704-FO2 Only 191
Address/Data Switch Subprogram (Entry Name: DATSW) 192
Library Function Error Subprogram (Entry Name: FCTST) 193
ROLLOUT Support Subprogram (Entry Name: INQCHK) 195
Inquiry Support Subprogram (Entry Name: SETINQ) 195
Date/Time-of-Day Subprogram (Entry Name: CFTOD) 196

GLOSSARY 197

APPENDIX A. FORTRAN SAMPLE PROGRAM . . . 201
Fortran Sample Program 201

APPENDIX B. COMPILATION MESSAGES 209

INDEX 221

This page is intentionally left blank.

Part 1. System/3 FORTRAN IV Language

System/3 FORTRAN IV consists of a *language* and a *processor.*

The System/3 FORTRAN IV language is especially useful in writing ***source programs*** for applications that involve mathematical computations and other manipulation of numerical data. Source programs written in the System/3 FORTRAN IV language consist of a set of statements constructed from the language elements described in this publication.

The processor consists of a FORTRAN *compiler* and a *library* of FORTRAN subprograms that perform operations required during program execution. In a process called *compilation*, the FORTRAN compiler analyzes the source program statements and translates them into an *object module.* In a process called *link-editing*, the object module is combined with FORTRAN subprograms from the library, and with any user-written subprograms, and becomes a *load module*, suitable for execution. In a process called *load module execution*, the load module is run by the System/3 System Control Program.

Chapter 1. FORTRAN Statements

Source programs consist of a set of statements from which the compiler generates instructions, constants, and storage areas. A given FORTRAN statement:

- Causes certain operations to be performed.
- Specifies the nature of the data being handled.
- Specifies the characteristics of the source program.

FORTRAN statements are composed of certain FORTRAN keywords used in conjunction with the elements of the language: constants, variables and expressions.

There are two broad classes of FORTRAN statements—*executable* statements and *nonexecutable* statements:

Class	Purpose	Example
Executable Statement	Performs calculations	A=B+C
	Transfers data between main storage and an input/output device	WRITE, READ
	Controls operation of an input/output device	REWIND
	Changes the order of execution in the program	GO TO
	Terminates program execution	STOP
Nonexecutable Statement	Provides initial values for variables and arrays	DATA
	Specifies the form in which data is to appear	FORMAT
	Defines the properties of variables, arrays, and functions	REAL*8
	Declares the operations to be performed by statement functions	CALC(I,J)=3*I/J
	Names and specifies arguments for subprograms	SUBROUTINE SUB3(A,B)

Some examples of FORTRAN statements and their effects are:

Statement	Meaning
A=96.0	Assigns the value 96.0 to the variable named A.
C=A/3	The slash (/) indicates division. Thus, this statement means divide A by 3 and set C equal to the result. Using the data of the previous example, C would be given the value 32.

For purposes of clarity in coding FORTRAN elements in this publication, we use uppercase letters to indicate actual coding, and lowercase letters to indicate a value that must be supplied.

CODING FORTRAN STATEMENTS

Although the usual form of input to the computer is a deck of punched cards, the initial coding of FORTRAN statements is generally on a coding sheet. The FORTRAN coding form, order number GX28-7327, is shown in Figure 1.

IBM

FORTRAN Coding Form

GX28-7327 U/M050
Printed in U.S.A.

PROGRAM		PUNCHING INSTRUCTIONS	GRAPHIC	PAGE OF
PROGRAMMER	DATE		PUNCH	CARD ELECTRO NUMBER

C FOR COMMENT	STATEMENT NUMBER	CONT.	FORTRAN STATEMENT	IDENTIFICATION SEQUENCE
1	2 3 4 5	6	7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72	73 74 75 76 77 78 79 80

Figure 1. FORTRAN Coding Form

Columns 1-5—Statement Number

Columns 1-5 can be used to enter a number by which the statement is referred to. Statement numbers can be assigned in any order; the sequence of operations depends only on the order of the statements, not on their statement numbers. Blanks and leading zeros in statement numbers are ignored by the FORTRAN processor. Thus,

```
         00090
         90
         09 0
   and   90
```

are equivalent. However, a given statement number can be assigned only once in a program unit.

Column 1—Comment Statement

Comments that explain the program can be written in columns 2-72 of a line having a C in column 1. Comments are not processed by the FORTRAN compiler, but are printed in the compiler listing. A comment statement cannot appear between continuation lines of a statement.

Column 6—Continuation

If a statement is too long for one line, it can be continued on as many as 19 successive lines by placing a character other than zero or a blank in column 6. If desired, the characters in column 6 can be used to indicate the order of continuation lines; that is, the character A can be used for the first continuation line, B for the second, etc. A blank or zero in column 6 indicates that the line is the initial line of a statement.

Columns 7-72—FORTRAN Statement

The FORTRAN statement must be written only in columns 7-72 of each line of the coding form.

Blanks can be used to improve the readability of a FORTRAN program because the compiler ignores blanks except in certain limited cases (literal fields of FORMAT or DATA statements and column 6 of a card).

Thus

```
A=B(I,J)-D-(C/E)-F**K
```

and

```
A = B ( I , J ) - D - ( C / E ) - F * * K
```

are equivalent.

Columns 73-80 (or 73-96)—Program Identification

Columns 73-80 (or 73-96 for 96-column cards) can be used for identifying information. These columns are not analyzed by the compiler, although the information in columns 73-80 is printed in the compiler listing. When 96-column cards are used, columns 81-96 are not printed in the compiler listing.

ELEMENTS OF THE LANGUAGE

In order to write FORTRAN programs, it is necessary to learn the rules for writing the following elements of the language:

Name of Element	Purpose	Examples
Constant	Specifies a numeric value	27 or 3.14159
Variable	Refers to a particular area in storage	X or Y
Array element	A member of a collection of data that has identical attributes	X(I) or Y(3,2)
Mathematical expression	Causes mathematical computations	A+B or 3*J
Assignment statement	Assigns a quantity to a variable or array element	A = B/C
Control statement	Affects the order in which statements are executed	DO or GO TO
Specification statement	Provides information about the data used in the source program, and the amount of storage required for it	IMPLICIT, REAL, COMMON
Input/Output statement	Gets data into the computer or transfers data to an output device	READ, WRITE
Subprogram statement	Performs a series of statements to be performed	SUBROUTINE, FUNCTION

ORDER OF STATEMENTS IN A FORTRAN PROGRAM

The order of statements in a FORTRAN program is as follows:

1. Subprogram statement for a subprogram. PROGRAM statement, if any, for a main program.
2. IMPLICIT statement, if any.
3. Other specification statements, if any.
4. Statement function definitions, if any, to describe statement functions.
5. Executable statements, at least one of which must be present.
6. END statement, to indicate the end of the program.

FORMAT and DATA statements can appear anywhere after the IMPLICIT statement, if present, and before the END statement. DATA statements, however, must follow any specification statements that contain the same variable or array names.

FORTRAN provides a means of expressing numeric constants, variable quantities, and array elements. The rules for expressing these quantities are quite similar to the rules of ordinary mathematical notation. However, each of these quantities can be expressed in one of two modes: integer or real (floating-point).

Integer and Real Calculations

Floating-point calculations are carried out between two real numbers in one of two degrees of precision--single precision (REAL*4) or double precision (REAL*8)—that can be specified by the user. The number of digits maintained for single precision is approximately 7, and for double precision approximately 17. Some typical floating-point calculations are:

Arithmetic Operation	Result of Calculation
A=.4301/1.7	A=.253
B=5./2.	B=2.5
C=1.6x.7	C=1.12
D=-2.7+1.2	D=-1.5

In floating-point calculations rounding does not occur; digits in excess of the precision are dropped.

Integer calculations are carried out differently; no decimal remainders are retained or used in computations. For example:

Arithmetic Operation	Result of Calculation
I=5/2	I=2 (instead of 2.5, because the .5 is truncated)
I=5/2+7/2	I=5 (intermediate truncation computes this as 2+3 rather than 12/2)
J=5x2	J=10
K=-4+1	K=-3

Data Types and Data Description

FORTRAN deals with arithmetic, literal, and hexadecimal data. Arithmetic data can be represented as a real or integer number, a constant, a variable, or an array (a group of related members that have related attributes).

Operation Symbols

Operation symbols used in FORTRAN are:

+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

To express the arithmetic operation A=.4301 divided by 1.7, the following statement can be used:

```
A=.4301/1.7
```

To express the operation J equals the product of 5 and 2, this statement can be used:

```
J=5*2
```

CONSTANTS

A constant is a number that is used in computations without change during execution of the program. It appears in its actual numeric form in the source statement. For example, in the statement

```
J=3*X
```

3 is a constant because it appears in actual numeric form.

Four types of constants can be specified in FORTRAN:

- Integer constant—written without a decimal point or exponent
- Real constant—written with a decimal point and/or an exponent
- Hexadecimal constant—the character Z followed by the hexadecimal characters 0 through 9 and A through F.
- Literal constant—a string of alphabetic, numeric, and/or special characters enclosed in apostrophes

The rules for writing each of these constants are given in the following sections.

Integer Constants

An *integer constant* is a whole number written without a decimal point, using the decimal digits 0, 1, . . . , 9. A preceding + or - sign is optional. An unsigned constant is assumed to be positive.

All integer constants occupy four storage locations (bytes). The magnitude of an integer constant cannot exceed 2,147,483,647 (2^{31}-1).

Examples:

Valid Integer Constants

0	
+9	
186	
-327	
6	
45	
123b̸456	b̸ (blank) is ignored by the processor

Invalid Integer Constants	Reason
4.321	Contains a decimal point
-3,675	Contains a comma
5436578656	Exceeds the magnitude permitted by the System/3 FORTRAN IV processor

Real Constants

A *real constant* is any of the following:

1. A *basic real constant* is a number written with a decimal point, using the decimal digits 0, 1, . . ., 9. A preceding + or - sign is optional. An unsigned constant is assumed to be positive.
2. A basic real constant followed by a D or E, followed by an exponent. The exponent is a signed or unsigned one- or two-digit integer constant. An unsigned exponent is assumed to be positive.
3. An integer constant followed by a D or E, followed by an exponent.

In the exponent, the letter E specifies a *single-precision* constant occupying four storage locations (bytes), and the letter D specifies a *double-precision* constant occupying eight storage locations. Unless it contains a D exponent, a real constant always occupies four storage locations.

Precision: Single precision—6 hexadecimal digits, or approximately 7.2 decimal digits.

Double precision—14 hexadecimal digits, or approximately 16.8 decimal digits.

Magnitude: Single-precision and double-precision constants have the same magnitude limitations: 0, or 16^{-65} (approximately 10^{-78}) through 16^{63} (approximately 10^{75}).

The decimal exponent permits the expression of a real constant as the product of a basic real constant or integer constant times 10 raised to a desired power.

Examples:

Valid Real Constants (single precision)	Equivalent To
+0.	
-999.9999	
7.0E+0	$7.0 \times 10^0 = 7.0$
19761.25E+1	$19761.25 \times 10^1 = 197612.5$
7.E3	
7.0E3	$7.0 \times 10^3 = 7000.0$
7.0E+03	
7E-03	$7.0 \times 10^{-3} = 0.007$
21.4354657687	*Note:* This level of precision cannot be accommodated in four storage locations. Positions to the right of the decimal point that cannot be accommodated are truncated.

Valid Real Constants (double precision)	Equivalent To
1234567890123456.D-93	
	$.1234567890123456 \times 10^{-77}$
7.9D03	
7.9D+03	$7.9 \times 10^3 = 7900.0$
7.9D+3	
7.9D0	$7.9 \times 10^0 = 7.9$
7D03	$7.0 \times 10^3 = 7000.0$

Invalid Real Constants	Reason
1	Missing a decimal point or exponent)
3,471.1	Embedded comma
1.E	Missing an integer constant following the E
1.2E+113	E is followed by a three-digit integer constant; must be one- or two-digits
23.5E+97	Magnitude outside the allowable range; 23.5×10^{97} greater than 16^{63}
21.3E-90	Magnitude outside the allowable range; 21.3×10^{-90} less than 16^{-65}

Hexadecimal Constants

A *hexadecimal constant* is used only in DATA initialization statements to specify initial values for variables and array elements. A hexadecimal constant contains the character Z followed by a hexadecimal number (0, 1, . . ., 9, A, B, C, D, E, F).

In System/3, the basic unit of storage is an 8-bit *byte*. Under this sytem, one byte contains two hexadecimal digits. The internal (binary) form of each of the 16 possible hex digits is as follows:

Hex	Binary	Hex	Binary
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

If a constant with an odd number of digits is specified, a leading hexadecimal zero is added on the left to fill the storage location.

Examples:

The eight-digit hexadecimal number Z1C49A2F1 represents the bit string 0001 1100 0100 1001 1010 0010 1111 0001

The seven-digit number ZBADFADE represents the bit string 0000 1011 1010 1101 1111 1010 1101 1110. The first four zero bits are implied because an odd number of hexadecimal digits was written.

Further information about hexadecimal constants can be found in the section that describes the DATA initialization statement.

Literal Constants

A *literal constant* is a string of alphabetic, numeric, and/or special characters, enclosed in apostrophes.

Each character requires one byte of storage. Note that the blank is considered a character. The appearance of a blank is significant only in literal constants and in statement continuation requirements. If an apostrophe is to appear in the literal, it must be represented by two consecutive apostrophes with no intervening blank.

Literals can be used as data initialization values in the DATA statement. Rules for this usage are presented in the Chapter 7 description of the DATA statement.

Examples:

```
'X-COORDINATE    Y-COORDINATE    Z-COORDINATE'
'3.14159'
'FRANCIS BACON''S ''HAMLET'''
```

VARIABLES

A FORTRAN variable is a data item, identified by a name, that occupies a storage area. The value specified by the name is always the current value stored in the area.

For example, in the statement

```
A=5.0+B
```

both A and B are variables. The value of B was determined by some previously executed statement. The value of A is calculated when the above statement is executed, and depends on the previously calculated value of B.

As with constants, a variable can be integer or real, depending on whether the value it represents is integer or real, respectively. Because a variable represents an area of storage, it is assigned a length, either implicitly or explicitly. A real variable can have a length of either four or eight bytes. An integer variable can have a length of either two or four bytes.

In order to distinguish between variables that derive their value from an integer, as opposed to a real number, the rules for naming each type of variable are different, although these rules can be overridden.

Variable Names

A variable name consists of from 1 to 6 alphabetic or numeric characters, of which the first must be alphabetic. In the context of this discussion, the character $ is considered an alphabetic character. Blanks in a variable name are ignored.

Examples:

```
I
ABCD
BILL23
I$2
ITEM1
I T E M 1
ITEM 1
```

In the above list, the last three names are considered identical.

The rules for naming variables allow for extensive selectivity. In general, it is easier to follow the flow of a program if meaningful names are used wherever possible. For instance, to compute distance it would be possible to use the statement:

```
A=B*C
```

but it would be more meaningful to write:

```
D=R*T
```

or:

```
DIST=RATE*TIME
```

Variable Types

The type of a variable corresponds to the type of the data the variable represents. Variable type can be specified in three ways: predefined, implicitly, or explicitly.

Predefined Convention

If the first character of the variable name is I, J, K, L, M, or N, the variable is considered to be an integer variable of length 4. If the first character of the variable name is *not* I, J, K, L, M, or N, the variable is a real variable of length 4.

Implicit Specification

By use of the IMPLICIT statement, the FORTRAN processor can be told to ignore the predefined convention for variable names whose initial letters are specified in the IMPLICIT statement. Thus, variables whose names begin with the letters I through N can be declared as type real, and those which begin with A through H or O through Z can be declared as type integer. In addition, variables can be declared to be one of the optional lengths: two bytes for integer variables, or eight bytes for real variables. The rules for using the IMPLICIT statement are given later.

Explicit Specification

Explicit specification of type (and, optionally, length) is made for individual variables by using the INTEGER or REAL type-specification statements. These statements override predefined and IMPLICIT specifications. The rules for using these statements are given later.

ARRAYS

An *array* is an ordered set of data items identified by a single name. A member of the array, called an *array element* is identified according to its position, by a quantity called a *subscript*.

Arrays, like variables and constants, have a type associated with them. The rules for naming arrays are the same as the rules for naming variables. The type of an array name is determined by the same conventions that govern the type of a variable name. Each element of an array is of the type and length specified for the array name.

Assume the following elements are contained in an array named NEXT:

15
12
18
42
19

Suppose you want to refer to the second element in the group; in ordinary mathematical notation this would be $NEXT_2$. In FORTRAN this would be

```
NEXT(2)
```

The quantity 2 is the subscript. Thus

NEXT(2) has the value 12
NEXT(4) has the value 42

Similarly, ordinary mathematical notation might use $NEXT_i$ to represent any element of the set, NEXT. In FORTRAN, this might be written as NEXT(I) where I equals 1, 2, 3, 4, or 5.

The array could be two dimensional, as for example, the array JO:

	Column 1	**Column 2**	**Column 3**
Row 1	82	4	7
Row 2	12	13	14
Row 3	91	1	31
Row 4	24	16	10
Row 5	2	8	2

Suppose you want to refer to the element in row 2, column 3; this would be

JO(2,3)

2 and 3 are the subscripts. Thus,

JO(2,3) has the value 14
JO(4,1) has the value 24

Similarly, ordinary mathematical notations might use JO_{ij} to represent any element of the set JO. In FORTRAN, this might be written as JO(I,J) where I equals 1, 2, 3, 4, or 5 and J equals 1, 2, or 3.

The above notation can extend to three-dimensional arrays:

JO(1,2,3)
JO(I,J,K)
JO(5,J,16)

The use of an array in the source program must be preceded by its declaration in either a DIMENSION statement, a COMMON or GLOBAL statement, or a type specification statement specifying the size of the array. These statements are explained later.

Arrangement of Arrays in Storage

An array is stored in ascending storage locations, with the value of the first of its subscripts increasing most rapidly, and the value of the last subscript increasing least rapidly.

For example, the array named A, described by one subscript that varies from 1 to 5, appears in storage as follows:

A(1) A(2) A(3) A(4) A(5)

The array named B, described by two subscripts, with the first varying from 1 to 5 and the second from 1 to 3, appears in storage as follows:

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

Note that B(1,2) and B(1,3) follow in storage B(5,1) and B(5,2), respectively.

The following list is the order of a three-dimensional array, C(3,3,3):

C(1,1,1) C(2,1,1) C(3,1,1) C(1,2,1) C(2,2,1) C(3,2,1)
C(1,3,1) C(2,3,1) C(3,3,1) C(1,1,2) C(2,1,2) C(3,1,2)
C(1,2,2) C(2,2,2) C(3,2,2) C(1,3,2) C(2,3,2) C(3,3,2)
C(1,1,3) C(2,1,3) C(3,1,3) C(1,2,3) C(2,2,3) C(3,2,3)
C(1,3,3) C(2,3,3) C(3,3,3)

Subscripts

A subscript is an integer quantity or a set of integer quantities separated by commas, enclosed in parentheses, and written immediately after the array name. The subscript is used to identify a particular element of an array. The number of subscript quantities in a subscript must be the same as the number of dimensions of the array. A maximum of three subscript quantities can appear in a subscript.

Form of Subscripts

A subscript must be in one of the following forms, (v represents any unsigned nonsubscripted integer variable, and c and c¹ any unsigned integer constant):

v
c
v+c or v–c
c*v
c*v+c¹ or c*v–c¹

Whichever subscript form you use, the evaluated result must be greater than 0, less than or equal to the range of the array, and less than or equal to 32,767. The evaluated result of a subscript is the product of all subscript quantities multiplied by the length of an element for the associated array. For example, if the subscript (200,4) is used to reference an element of an integer *4 array, the evaluated result is the product of the subscript quantities (200 x 4 = 800), multiplied by the length of an element (800 x 4 = 3200).

Examples:

Valid Subscripts	Form of Subscript
(IMAX)	v
(19)	c
(JOB+2)	v+c
(NEXT–3)	v–c
(8*IQUAN)	c*v
(5*L+7)	c*v+c¹
(4*M–3)	c*v–c¹

Invalid Subscripts	Reason
(–I)	The variable cannot be signed.
(0)	A subscript quantity cannot assume the value 0.
(–2*J)	The constant must be unsigned.
(I(3))	A subscript cannot be subscripted.
(K*2)	For multiplication, the constant must precede the variable, thus 2*K is correct.
(2+JOB)	For addition, the variable must precede the constant; thus, JOB+2 is correct.

ARITHMETIC ASSIGNMENT STATEMENTS

The arithmetic assignment statement defines a numeric calculation; it resembles a conventional arithmetic formula.

The general form of an arithmetic assignment statement is:

a=b

where

a is a variable or array element.

b is an expression as defined under *Expressions.*

Examples:

The following are valid arithmetic assignment statements:

```
A=B+C
D(I)=E(I)+2.-F
```

In an arithmetic assignment statement, the equal sign means *is to be replaced by* rather than *is equivalent to.* This distinction is important; for example, suppose an integer variable I has the value 3. Then, the statement

```
I=I+1
```

would give I the value 4. This feature enables the programmer to keep counts and perform other required operations in the solution of a problem.

The following is an example of a series of arithmetic assignment statements:

Statement	**Meaning**
A=3.0	Store the value 3.0 in A
B=2.0	Store the value 2.0 in B
C=A+B	Add the values in A and B and store in C(3.+2.=5.)
C=C+1	Add 1. to the value in C(5.+1.=6.)

EXPRESSIONS

An expression in FORTRAN is a sequence of constants, variables, array elements, and operation symbols that indicates a quantity or a series of calculations. It must be formed according to the rules for constructing expressions. It can include parentheses and can also include functions (which will be discussed later). It appears on the right-hand side of arithmetic assignment statements and in certain types of control and I/O statements.

Rules for Constructing Expressions

Because constants, variables, and array elements can be integer or real quantities, expressions can contain integer or real quantities; that is, two types can appear in the same expression.

In the following discussion, no mention is made of the rules for using integer and real quantities in functions. These rules are stated when functions are described and are considered as addenda to the following rules.

1. The simplest expression consists of a single constant, variable or array element. If the quantity is an integer type, the expression is said to be in the integer mode. If the quantity is a real type, the expression is said to be in the real mode.

   Examples:

Expression	**Type of Quantity**	**Mode of Expression**
3	Integer constant	Integer
3.0	Real constant	Real
I	Integer variable	Integer
A	Real variable	Real
I(J)	Integer array element	Integer
A(J)	Real array element	Real

   In the last example, note that the subscript, which must be an integer quantity, does not affect the mode of the expression. The mode of the expression is determined solely by the type of the quantity itself.

2. Real exponentiation of a quantity affects the mode of the expression; thus:

I**2.3	Real
I**J	Integer
A**I	Real
A**B	Real

3. Quantities can be preceded by plus or minus signs (+ or -), or can be connected by any of the operation symbols (+, -, *, /, **) to form expressions, provided:
   a. No two operation symbols appear consecutively. Quantities connected need not all be the same mode but are converted to the higher mode (in the order INTEGER*2, INTEGER*4, REAL*4, REAL*8) before the expression is evaluated. For example, in A+I, if A is real and I is integer, I is converted to real before the addition. Figure 2 shows the type and length of the result of arithmetic operations.

First Quantity \ Second Quantity	INTEGER*2	INTEGER*4	REAL*4	REAL*8
INTEGER*2	INTEGER*2	INTEGER*4	REAL*4	REAL*8
INTEGER*4	INTEGER*4	INTEGER*4	REAL*4	REAL*8
REAL*4	REAL*4	REAL*4	REAL*4	REAL*8
REAL*8	REAL*8	REAL*8	REAL*8	REAL*8

Figure 2. Determining the Type and Length of the Result of Arithmetic Operations that use Quantities of Different Mode

   b. No operation symbols are assumed; that is, no two quantities appear consecutively.

   Examples:

Valid Expressions

-A+B
B+C-J
I/J
K*L

Invalid Expressions	**Reason**
A+-B	Must be written as A+(-B).
3J	Must be written as 3*J if multiplication is intended.

4. The order of operations (hierarchy) must be considered when writing FORTRAN statements. The hierarchy of operations (from highest to lowest) is as follows:

   1. Evaluation of functions

   2. Exponentiation (**)

   3. Multiplication (*) and Division (/)

   4. Addition (+) and Subtraction (-)

   Because of this hierarchy, the expression

   A+B*C/D+E**F-G*H

   will be taken to mean

   $$A+\frac{B*C}{D}+E^{F}-(G*H)$$

Hierarchy determines which of two consecutive operations is performed first. If the hierarchy of the first operation is higher than or equal to the hierarchy of the second operation, the first operation is performed. If it is not, the hierarchy of the second operation is compared to the hierarchy of the third, etc. Once an operation is executed, the next comparison starts with the last operation that was skipped.

In the following examples, the operations are numbered in the order in which they will be performed:

```
X = A + B * C / D + E ** F - G * H
      3   1   2   5   4    7   6
```

When evaluating consecutive exponentiation operations, the order is from right to left:

```
X = A**B**C + D
     2  1   3
```

Parentheses can be used in arithmetic expressions, as in algebra, to specify the order in which operations are to be performed. Where parentheses are used, the expression within the innermost set of parentheses is evaluated first. For example:

```
X = ( ( A + B ) * C / D + E ) ** F - G * H
          1     2   3   4      5    7   6
```

Mode of an Arithmetic Statement

Expressions must be integer or real; however, the variable on the left-hand side of the equal sign in an arithmetic statement need not be of the same mode as the expression on the right-hand side.

If the variable on the left is an integer quantity and the expression on the right is real, the expression is first evaluated as a real quantity, the portion following the decimal point is dropped, and the remainder is converted to an integer quantity. Thus, if the result is +3.872, the integer stored is +3, not +4. If the variable on the left is real and the expression on the right is integer, the latter is evaluated as an integer expression, and the result is converted to real.

Examples:

Arithmetic Statement	Result of Calculation	Contents of Variable
A=3/2	1	A is assigned the value 1.0
A=3./2	1.5	A is assigned the value 1.5
I=3/2	1	I is assigned the value 1
I=3./2.	1.5	I is assigned the value 1
I=3./2	1.5	I is assigned the value 1

Normally, FORTRAN statements are executed sequentially. That is, after one statement is executed, the statement immediately following it is executed. However, it is often undesirable to proceed with each statement in this manner. This chapter describes the statements used to alter sequential execution.

UNCONDITIONAL GO TO STATEMENT

The general form of the unconditional GO TO statement is:

GO TO n

where n is a statement number.

This statement is used to interrupt sequential execution; it indicates the statement that is to be executed next.

Uses: The GO TO statement transfers control to the statement numbered n.

Considerations/Restrictions: The statement following the GO TO statement must have a statement number in order to be executed.

Examples:

```
GO TO 16
GO TO 137
```

A coding example is shown below:

```
       .
       .
       .
       A=3.
       B=4.
       GO TO 7
12     B=2.*A
7      A=2.*B
       .
       .
       .
```

Statement 12 is not executed. After the GO TO statement is executed, statement 7 is evaluated and A is assigned the value 8.0.

COMPUTED GO TO STATEMENT

The general form of the computed GO TO statement is:

GO TO $(n_1, n_2, \ldots, n_n)$, i

where

$n_1, n_2, \ldots, n_n$ are statement numbers
i is an integer variable having a value of 1 to n.

This statement also indicates which statement is executed next. However, it also gives you the ability to execute different statements during various stages in the program.

Uses: This statement transfers control to the statement numbered $n_1, n_2, \ldots n_n$ depending on whether the current value of i is 1,2, . . . ,n, respectively. If the value of i is less than 1 or greater than the number of statement numbers in the list, the next sequential statement is executed.

Considerations/Restrictions:

1. i must be given a value before the computed GO TO statement is executed.
2. No more than 60 statement numbers can be specified in a computed GO TO statement.

Examples:

GO TO(5,7,8,2,4),J	If J is 3, transfer control to statement 8.
GO TO(4,4,4,7,8,9),MAX	This example illustrates that several values of i can cause a transfer of control to the same statement. In this case, when MAX has the value 1, 2, or 3, control transfers to statement 4.

Further use of the computed GO TO statement is illustrated below:

```
        .
        .
        .
        A=3.
        B=4.
        C=5.
        K=0
 1      K=K+1
        GO TO(10,20,30),K
        .
        .
        .
 30     F=A-B
        GO TO 12
 20     E=A-C
        GO TO 1
 10     D=B-C
        GO TO 1
        .
        .
        .
 12     CONTINUE
```

As a study of this example shows, D, E and F are computed, in that order, and control proceeds to statement 12. Of course, the example itself is highly simplified; if these were the only required calculations in this series, the programmer would just compute D, E, and F sequentially, in any desired order and without using the Computed GO TO statement.

RELATIONAL IF STATEMENT

The general form of the relational IF statement is:

IF (a) s

where

a is a relational expression

s is the associated statement—any executable statement except a DO statement or another relational IF statement.

A relational expression is formed by combining two arithmetic expressions with one of the six relational operators:

.GT.	Greater Than
.LT.	Less Than
.EQ.	Equal to
.NE.	Not Equal to
.GE.	Greater than or Equal to
.LE.	Less than or Equal to

Periods must precede and follow the relational operators, as shown.

Uses: The relational IF statement permits the programmer to execute or skip an associated statement depending on whether the relational expression is true or false.

Considerations/Restrictions: The associated statement can not be a DO statement or another relational IF statement.

Examples:

```
IF(A.GT.1.0) GO TO 50
IF(A-B.LT.A+C) A=B
```

The associated statement is executed if the relational expression is true. Otherwise, the statement following the IF statement is executed next. In the second example, if the relational expression is true, A is set equal to B and then the statement following the IF statement is executed.

Suppose a series of records, each containing a variable code number, I, is being read and processed. Certain of the records, appearing at random but with special code numbers greater than 99, are to be processed differently. The FORTRAN statements to accomplish this might be as follows:

```
        .
        .
        .
        IF(I.GT.99) GO TO 20
        .
        .
        .
  20    A=B+C
        .
        .
        .
```

ARITHMETIC IF STATEMENT

The general form of the arithmetic IF statement is:

IF (a)n_1,n_2,n_3

where

a represents an expression.

n represents a statement number.

The expression, a, must be enclosed in parentheses; the statement numbers must be separated from one another by commas. The same statement number can be specified more than once.

Uses: This statement transfers control to the statement numbered n_1,n_2, or n_3, depending upon whether the value of the expression, a, is negative, zero, or positive, respectively.

Considerations/Restrictions: The statement following the arithmetic IF statement must have a statement number in order to be executed.

Examples:

```
IF(A-B)10,10,7
IF(A(I)/D)1,2,3
```

Control transfers to statement number n_1, n_2, or n_3 depending on whether the value of a is less than, equal to, or greater than zero, respectively. Note that in the first example, the same statement, numbered 10, is to be executed if a is less than or equal to 0.

As another example, suppose a value, A, is being computed. Whenever this value is positive, you wish to proceed with the program. Whenever the value of A is negative, an alternative routine starting at statement 12 is to be followed; and if A is zero, an error routine at statement 72 is to be followed. This may be coded as:

```
        .
        .
        .
        A=(B+C)/(D**E)-F
        IF(A)12,72,10
  10    .
        .
        .
  12    .
        .
        .
  72    .
```

DO STATEMENT

The general form of the DO statement is:

DO n i=m_1,m_2,m_3

where

n (end of range) is a statement number identifying the last statement to be executed in the range of the DO statement.

i (the DO variable) is a non-subscripted integer variable.

m_1(the initial value) is an unsigned integer constant greater than 0 or a non-subscripted integer variable. The value of m_1 should not exceed the value of m_2.

m_2(the test value) is an unsigned integer constant greater than 0 or a non-subscripted integer variable.

m_3(the increment) is an unsigned integer constant greater than 0 or a non-subscripted integer variable. m_3 is optional and if omitted is assumed to be 1. If m_3 is omitted, the preceding comma must also be omitted.

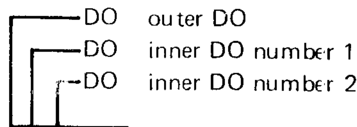
Uses: The DO statement is used to control multiple execution of the statements that physically follow the DO statement, up to and including the statement numbered n. These statements are called the range of the DO. For the first execution of the statements within the DO range, the variable, i, is set to the initial value, m_1. Each time statement n (the end of the range) is executed, the value of i is increased by the increment value, m_3, and checked against the test value, m_2. If i is less than or equal to m_2, the statements within the DO range are executed again. If the value of i is greater than the value m_2, the statement immediately following statement number n is executed and i becomes undefined.

Considerations/Restrictions:

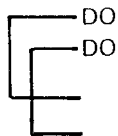
1. The statements within the DO range are always executed at least once.

2. A maximum of 25 nested DOs is permitted. A nested DO cannot extend beyond the range of the containing DO statement.

   For example, the following is a valid nest of DOs:

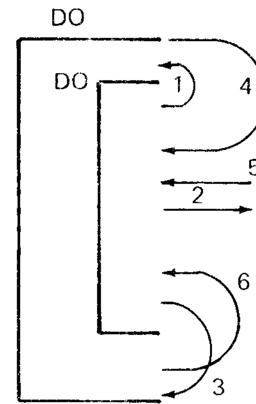


   The following is an invalid nest of DOs.



   Transfer of control to the end of the range of a nest of DOs is permitted only from the innermost DO. A transfer from any other DO in the nest causes the innermost DO to begin execution.

3. Transfer of control from inside the range of a DO to a statement outside its range (by means of a GO TO or an IF statement) is permitted at any time. However, the reverse is not allowed except as specified in item 7 in the following configuration. A transfer of control is not permitted into the range of any DO from outside its range. Thus, in the following configuration 1, 2 and 3 are permitted transfers of control, but 4, 5 and 6 are not.

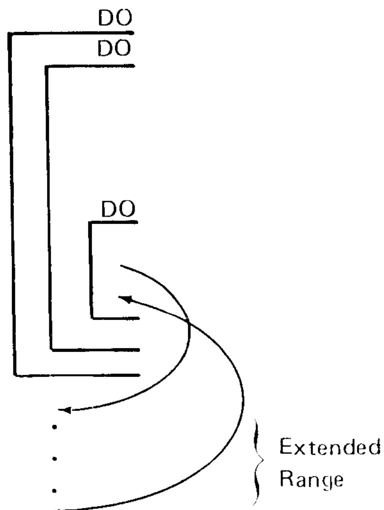


4. After a normal exit from a DO (when the DO becomes satisfied and control passes to the next statement after the range), the value of the DO variable is not defined, and the DO variable cannot be used again until it is defined. If exit occurs by a transfer of control out of the range, the current value of the DO variable is preserved for subsequent use.

5. No statement is permitted in the range of the DO that changes the value of any of the variable parameters (i, m_1, m_2, or m_3).

6. The last statement in the range of a DO must be an executable statement. It cannot be a transfer of control, such as a GO TO of any type, an arithmetic IF, another DO, PAUSE, STOP, or RETURN. A relational IF statement is valid only if the associated statement is not a transfer of control (PAUSE and STOP are not considered transfers of control when associated with a relational IF).

7. The *extended range* of a DO is defined as those statements that are executed between the transfer out of the *innermost* DO of a set of completely nested DOs, and the transfer back into the range of this innermost DO. In a set of completely nested DOs, the first DO is not in the range of any other DO, and each succeeding DO is in the range of every DO that precedes it. The following restrictions apply:
   a. Transfer into the range of a DO is permitted only if such a transfer is from the extended range of the DO.
   b. The extended range of a DO statement must not contain another DO statement that has an extended range if the second DO is within the same program unit as the first.
   c. The variable parameters (i, m_1, m_2, m_3) cannot be changed in the extended range of the DO.



8. The m_1, m_2, or m_3 values, if constants, cannot exceed 2^{31}-2.

Example 1:

```
        .
        .
        .
15    DO 25 J=1,1000
25    INV(J)=INV(J)-IOUT(J)
35    .
      .
      .
```

Statement 15 is a command to execute the following statements up to and including statement 25. The first time, J is 1; thereafter J is incrased by 1 for each execution of the loop until the loop is executed with J equal to 1000. After the loop is executed with J equal to 1000, the statement following statement 25 is executed.

Example 2:

```
      .
      .
      .
      K=0
      L=10
      DO 5 JOB=1, L, 2
      K=K+1
5     M(JOB)=N(JOB)-K*JOB
```

This causes the following computations:

```
M(1)=N(1)-1*1
M(3)=N(3)-2*3
M(5)=N(5)-3*5
M(7)=N(7)-4*7
M(9)=N(9)-5*9
```

CONTINUE STATEMENT

The general form of a CONTINUE statement is:

```
CONTINUE
```

Uses: The CONTINUE statement is used as a dummy statement that can be placed anywhere in the source program without affecting the sequence of execution. The primary use of the CONTINUE statement is as the last statement in the range of a DO statement, where you cannot use a transfer of control statement.

Considerations/Restrictions: A CONTINUE statement, without a statement number, is ignored.

Example: As an example of a program that requires a CONTINUE statement, consider the following:

```
        .
        .
        .
10    DO 12 I=1, 100
      IF (ARG.EQ.VALUE(I)) GO TO 20
12    CONTINUE
      .
      .
      .
```

This program scans the 100-element VALUE array until it finds an entry that equals the value of the variable ARG, whereupon it transfers control to statement 20 with the value of I available for use. If no entry in the array equals the value of ARG, a normal exit to the statement following the CONTINUE occurs.

PAUSE STATEMENT

The general form of the PAUSE statement is:

PAUSE
or
PAUSE n

where n is an unsigned integer constant not greater than 99999.

Uses: The PAUSE statement halts program execution and displays a halt code—b6 on the Model 10, Model 12, and Model 15; 4 on the Model 6. The rightmost two digits of the value n are displayed as a subhalt. If n is not specified, zeros are displayed.

The word PAUSE and the value n are printed as follows:

Model 6 — Printed unless // NOPRINTER has been specified.

Model 10 — Printed on the FORTRAN error logging device. See *PRINT and NOPRINTER Device Option Statements* in Chapter 13 for more information.

Model 12 — Printed on the line printer assigned as the system log device if that printer has not been allocated to the program prior to the PAUSE n execution. If // LOG CONSOLE is specified, the PAUSE n is printed on the console only.

Model 15 — Printed on the printer that is assigned as the system log device if that printer is not allocated to the program prior to PAUSE n execution. (PAUSE n is always displayed on the console.)

Note: The format of the Model 6 display is different from that on the Model 10, Model 12, and Model 15. Use the algorithm and tables in Figure 3 to convert one format to the other.

When execution is halted by the PAUSE statement, the following responses are permitted:

0 Continue execution of the program with the next FORTRAN statement
2 Controlled cancel
3 Immediate cancel

For an explanation of how to respond to a halt, see the appropriate Halt Guide, listed in the Preface under *Related Publications*.

Considerations/Restrictions:

1. The constant, n, cannot exceed 99999.
2. If n is not specified, zeros are displayed or printed.

Examples:

PAUSE
PAUSE 50
PAUSE 00002

STOP STATEMENT

The general form of the STOP statement is:

STOP
or
STOP n

where n is an unsigned integer constant not greater than 99999.

Uses: The STOP statement terminates execution of an object program. If n is specified, the STOP statement also displays a halt code—b6 on the Model 10, Model 12, and Model 15; 4 on the Model 6. The two rightmost digits of the STOP value, n, are displayed as a subhalt. If n is not specified, normal EOJ is reached.

The word STOP and the value n are printed as follows:

Model 6 — Printed unless // NOPRINTER is specified.

Models 10 and 12 — Printed on the FORTRAN error logging device. See *PRINT and NOPRINTER Device Option Statements* in Chapter 13 for more information. If // LOG CONSOLE is specified, the STOP n is printed on the console.

Model 15 — Printed on the FORTRAN error logging device. See *PRINT and NOPRINTER Device Option Statements* in Chapter 13 for more information.

Note: The format of the Model 6 display is different from that on the Model 10, Model 12, and Model 15. Use the algorithm and tables in Figure 3 to convert one format to the other.

The only responses to a halt caused by the STOP statement are:

2 Controlled cancel
3 Immediate cancel

For an explanation of how to respond to a halt, see the appropriate Halt Guide, listed in the Preface under *Related Publications.*

Considerations/Restrictions:

1. Execution of an object program halted by a STOP statement cannot be resumed.

2. The constant, n, cannot exceed 99999.

Examples:

```
STOP
STOP 25
```

PAUSE and STOP Algorithm

Subhalt, STOP, and PAUSE values are converted for display by the Model 6 according to the following algorithm:

1. Determine the displacement in the table (see *Character to Displacement,* Figure 3) of the first character.

2. Multiply the displacement by 22.

3. Determine the displacement of the second character and add it to the result of step 2.

4. Convert the result of step 3 from decimal to hexadecimal (see *Decimal to Hex,* Figure 3).

5. Write the 9-bit binary equivalent of the hex value obtained in step 4. The bits on indicate the display lights on.

For example, to convert the U2 halt:

```
U displacement = 7
2 displacement = 4

7 x 22 = 154
154 + 4 = 158
158 (dec) = 9E (hex)

9E (hex) = 0000  1001 1110
                 A BCD1 2345

Model 6 halt =   B    1 234
```

Following are exceptions to the conversion algorithm:

Disk System Halt	Model 6 Halt
EJ	ABCD12345
HE	BCD12345
5Y	A CD12 5
80	3 5
b̸0	12 45
0C	AB D12 45
7E	CD1 3
0A	AB D 2345
'Y	CD123

Any Disk System halt that ends with (-) except for (- -) cannot be converted. If Model 6 receives a disk system halt it cannot convert, Model 6 issues halt ABC 1 3.

Model 6 halts derived from disk system halts can be converted back to disk system halts by reversing the conversion algorithm. For example:

```
Model 6 halt  B1234
Displayed B   1 234
     0000 1001 1110 = 9E (hex)
```

9E (hex) = 158 (decimal) (see *Decimal to Hex*, Figure 3).
158/22 = 7 + 4

7 = displacement for U (see *Character to Displacement,* Figure 3).
4 = displacement for 2 (see *Character to Displacement,* Figure 3).

Disk system halt = U2

Character to Displacement

Character		8	6	E	2	0	C	U	L	J	9	5	3	Y	A	P	F	H	4	'	7	1	-
Displacement	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Decimal to Hex

High Order \ Low Order	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	0010	0011	0012	0013	0014	0015
1	0016	0017	0018	0019	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	0030	0031
2	0032	0033	0034	0035	0036	0037	0038	0039	0040	0041	0042	0043	0044	0045	0046	0047
3	0048	0049	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	0060	0061	0062	0063
4	0064	0065	0066	0067	0068	0069	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079
5	0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	0090	0091	0092	0093	0094	0095
6	0096	0097	0098	0099	0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	0110	0111
7	0112	0113	0114	0115	0116	0117	0118	0119	0120	0121	0122	0123	0124	0125	0126	0127
8	0128	0129	0130	0131	0132	0133	0134	0135	0136	0137	0138	0139	0140	0141	0142	0143
9	0144	0145	0146	0147	0148	0149	0150	0151	0152	0153	0154	0155	0156	0157	0158	0159
A	0160	0161	0162	0163	0164	0165	0166	0167	0168	0169	0170	0171	0172	0173	0174	0175
B	0176	0177	0178	0179	0180	0181	0182	0183	0184	0185	0186	0187	0188	0189	0190	0191
C	0192	0193	0194	0195	0196	0197	0198	0199	0200	0201	0202	0203	0204	0205	0206	0207
D	0208	0209	0210	0211	0212	0213	0214	0215	0216	0217	0218	0219	0220	0221	0222	0223
E	0224	0225	0226	0227	0228	0229	0230	0231	0232	0233	0234	0235	0236	0237	0238	0239
F	0240	0241	0242	0243	0244	0245	0246	0247	0248	0249	0250	0251	0252	0253	0254	0255
10	0256	0257	0258	0259	0260	0261	0262	0263	0264	0265	0266	0267	0268	0269	0270	0271
11	0272	0273	0274	0275	0276	0277	0278	0279	0280	0281	0282	0283	0284	0285	0286	0287
12	0288	0289	0290	0291	0292	0293	0294	0295	0296	0297	0298	0299	0300	0301	0302	0303
13	0304	0305	0306	0307	0308	0309	0310	0311	0312	0313	0314	0315	0316	0317	0318	0319
14	0320	0321	0322	0323	0324	0325	0326	0327	0328	0329	0330	0331	0332	0333	0334	0335
15	0336	0337	0338	0339	0340	0341	0342	0343	0344	0345	0346	0347	0348	0349	0350	0351
16	0352	0353	0354	0355	0356	0357	0358	0359	0360	0361	0362	0363	0364	0365	0366	0367
17	0368	0369	0370	0371	0372	0373	0374	0375	0376	0377	0378	0379	0380	0381	0382	0383
18	0384	0385	0386	0387	0388	0389	0390	0391	0392	0393	0394	0395	0396	0397	0398	0399
19	0400	0401	0402	0403	0404	0405	0406	0407	0408	0409	0410	0411	0412	0413	0414	0415
1A	0416	0417	0418	0419	0420	0421	0422	0423	0424	0425	0426	0427	0428	0429	0430	0431
1B	0432	0433	0434	0435	0436	0437	0438	0439	0440	0441	0442	0443	0444	0445	0446	0447
1C	0448	0449	0450	0451	0452	0453	0454	0455	0456	0457	0458	0459	0460	0461	0462	0463
1D	0464	0465	0466	0467	0468	0469	0470	0471	0472	0473	0474	0475	0476	0477	0478	0479
1E	0480	0481	0482	0483	0484	0485	0486	0487	0488	0489	0490	0491	0492	0493	0494	0495
1F	0496	0497	0498	0499	0500	0501	0502	0503	0504	0505	0506	0507	0508	0509	0510	0511

Figure 3. Halt Conversion Tables

END STATEMENT

The general form of the END statement is:

```
END
```

Uses: The END statement is a nonexecutable statement that defines the end of a main program or subprogram. Physically, it must be the last statement of each program unit. If any FORTRAN statements follow an END statement, they are ignored by the compiler and not printed. The END statement does not terminate program execution.

Considerations/Restrictions:

1. The END statement cannot have a statement number.
2. The END statement cannot have a zero in column 6.
3. The END statement cannot have a continuation.

Examples:

```
PROGRAM FIRST      SUBROUTINE SECOND
      .                   .
      .                   .
      .                   .
CALL SECOND        RETURN
      .            END
      .
      .
STOP 25
END
```

In order for data to be transmitted from an external storage medium such as a punched card, tape, or disk to the computer, or from the computer to an external medium, it is necessary that the computer know the form in which the data exists. The FORMAT statement describes the form of the data.

The FORMAT statement uses format codes to specify the type of conversion to be performed between the external and the internal representation of each quantity specified in a READ or WRITE statement I/O list.

FORMAT statements are associated by their statement numbers with specific READ or WRITE statements. Successive items in the I/O list are transmitted according to successive specifications in the FORMAT statement. FORMAT statements are not executed and can appear anywhere in the same program unit as their associated I/O statements, subject to the rules governing placement of PROGRAM, IMPLICIT, FUNCTION, SUBROUTINE, and END statements.

FORMAT STATEMENT

The general form of a FORMAT statement is:

xxxxx FORMAT($c_1 s_1 c_2 s_2 \ldots c_n$)

where

xxxxx	is a statement number (1 through 5 digits).
c	is a format code (described following).
s	is a separator, which may be either a comma or any number of slashes. Slashes are used to indicate the beginning of a new record. Any number of slashes may precede the first or follow the last format code.

Format Code	Purpose
aIw	Describes integer data fields.
aDw.d	Describes double precision data fields.
aEw.d	Describes real data fields.
aFw.d	Describes real data fields.
nPc	Describes a scale factor; if present, this code is specified as the first part of a D, E, or F field descriptor.
aAw	Describes character data fields.
wHstring	Describes literal data.
'literal'	Describes literal data.
wX	Indicates that a field is to be skipped on input or filled with blanks on output.
Tr	Indicates the position in a FORTRAN record where transfer of data is to begin.

where

a	is optional and is a repeat count, an unsigned integer constant that specifies the number of times the code is to be repeated. If a is omitted, the code is used once.
w	is an unsigned nonzero integer constant that specifies the number of characters in the field (field width).
d	is an unsigned integer constant specifying the number of decimal places to the right of the decimal point (the fractional portion). The decimal point between w and d portions of the specification is required.
n	is a negative or unsigned integer constant which is the scale factor; if the constant is unsigned, it is assumed to be positive.
c	is format code D, E, or F.
string	is a set of characters in a field.
r	is an unsigned integer constant designating a character position in a record.

Uses: The FORMAT statement is used in conjunction with the I/O lists in the READ and WRITE statements to specify the structure of FORTRAN records and the form of the data fields within the records. FORMAT statements are associated by their statement numbers with specific READ or WRITE statements.

Considerations/Restrictions:

1. A FORMAT statement is not executed–its function is to supply information to the object program. The FORMAT statement can be placed anywhere in the source program.
2. When defining a FORTRAN record with a FORMAT statement, it is necessary to consider the maximum size record allowed on the input/output device.
3. The specifications in a FORMAT statement must correspond in mode to the items in the I/O list–integer quantities require integer format codes and real quantities require real format codes.
4. The field width, w, of a format code cannot exceed 255.
5. The repeat count, a, whether of the form aDw.d or a(Dw.d), cannot exceed 255.
6. The field width, w, can be specified greater than required in order to provide spacing. For example, to write out an integer variable with the I format code, a specification of I10 reserves five leading blanks if the number does not exceed five digits, including the sign.

Examples:

```
05 FORMAT (I5,F8.4/20X,I5)
10 FORMAT (' THE ANSWER IS',I10)
33 FORMAT (E6.2,2(I3,2F2.1))
```

VARIOUS FORMS OF A FORMAT STATEMENT

All of the format codes in a FORMAT statement are enclosed in a pair of parentheses. Within these parentheses, the format codes are delimited by a separator–a comma or a slash. The comma delimits one format code from another; the slash is used to indicate the beginning of a new record.

For example, the statement

```
25 FORMAT (I3,F6.2/D10.3,F6.2)
```

describes the format of two records. The first record is transmitted according to the format codes I3 and F6.2; the second record is transmitted according to the format codes D10.3 and F6.2.

Consecutive slashes can be used to produce blank output records or to skip input records. If there are n consecutive slashes at the beginning or end of a FORMAT statement, n input records are skipped or n blank records are inserted between output records. If n consecutive slashes appear anywhere else in a FORMAT statement, the number of records skipped or blank records inserted is n–1.

For example, the statement

```
25 FORMAT (//I5//I5//)
```

describes seven record formats. If this FORMAT statement is used to transmit records to the printer, it produces two blank lines, one line of data, one blank line, another line of data, and two blank lines.

The group format specification is used to repeat a set of format codes and to control the order in which the format codes are used. The group repeat count, a, is the same as the repeat count, a, that can be placed in front of other format codes. For example, the following statements are equivalent:

```
25 FORMAT (2(E10.5,D12.6) ,I4)
25 FORMAT (E10.5,D12.6,E10.5,D12.6,I4)
```

An additional level of parentheses is not permitted. For example, the following statement is invalid:

```
25 FORMAT (2(3(I6,I8)))
```

Additional parentheses are valid, however, as long as they are not in the same group. For example:

```
25 FORMAT (2(I2),2(I4))
```

Each time a READ or WRITE statement is executed, successive items in the I/O list are transmitted according to successive format codes in the FORMAT statement. This continues until all the items in the list are transmitted. If there are more items in the list than there are format codes, control transfers to the last preceding left parenthesis of the FORMAT statement. This is either the left parenthesis at the beginning of the FORMAT statement or, if grouping is used, the left parenthesis of the last group in the FORMAT statement. For example, if the I/O list contains the variables A, B, C, D, and E and the FORMAT statement used is

```
25 FORMAT (F6.2,D10.3,F12.2)
```

the values transmitted for A, B, and C would use format codes F6.2, D10.3, and F12.2, respectively. Because the list is not exhausted, control returns to the preceding left parenthesis and the values transmitted for D and E would use format codes F6.2 and D10.3, respectively. If the FORMAT statement were coded

```
25 FORMAT (F6.2,D10.3,2(F12.2))
```

the values transmitted for A, B, C, D, and E would use format codes F6.2, D10.3, F12.2, F12.2, and F12.2, respectively.

If there are fewer items in the I/O list than there are format codes, the remaining format codes are ignored.

FORMAT CODES FOR NUMERIC DATA

Four types of format codes for numeric data are:

Internal	Format Code	External
Real	F	Real (without exponent)
Real	E	Real (with exponent; single precision)
Real	D	Real (with exponent; double precision)
Integer	I	Integer

Numbers printed by the F format code are printed in a decimal notation without an exponent. Typical output might be:

```
 12.3    -0.726    102.
-17.2     1.318   -968.
289.1     0.009    721.
```

Numbers printed by the D and E format codes are printed as a decimal fraction to a power of 10. These numbers are *normalized;* that is, their first significant digit is to the right of the decimal point. For example:

```
232.3    is printed as 0.2323E+03
.003     is printed as 0.30E-02
17.4     is printed as 0.174E+02
```

Numbers printed by the I format code are printed as integers. Typical output might be:

```
  12
 -17
2342
```

I Format Code (aIw)

The I format code is used to read integer data or to print a number that exists in the computer as an integer quantity.

On output, w print positions are reserved for the number. It is printed in a w-space field right-justified (that is, the units position is at the extreme right). If the number converted is greater than w spaces, asterisks are printed instead of the number. If the number has less than w digits, the leftmost spaces are filled with blanks. If the quantity is negative, the space preceding the leftmost digit contains a minus sign, for which a space must be reserved.

The following examples show how each of the quantities on the left is printed according to the specification I3 (ƀ is used to indicate blanks):

Internal Value	Printed
721	721
-721	***
-12	-12
9	ƀƀ9
8114	***
0	ƀƀ0
-5	ƀ-5

On input, w characters are read from an input device. Blanks on either side of a significant digit are treated as zeros. If the number is too large to be contained in a 2- or 4-byte integer variable, only the rightmost digits are used, and computations involving this variable are meaningless.

The following examples show the internal values of the given quantities if read under the I3 format code:

Input	Internal Value
3ƀƀ	300
ƀƀƀ	0
ƀ3ƀ	30

F Format Code (aFw.d)

On input, w is the total field width, including the exponent, if any, and d is the number of places to the right of the decimal point (the fractional portion). If a decimal point is present, its position overrides the d specification in the format code. Either a D, E, or signed integer exponent is acceptable as input with an F format code. Blanks are treated as zeros; thus, embedded and trailing blanks affect the value of the number.

For example, the following item is interpreted as having the value 1000:

Field Description	Input Record
F8.0	ƀƀƀƀ1ƀƀƀ

On output, w must provide sufficient space for an integer segment if it is other than zero, a fractional segment containing d digits, a decimal point, and, if the output value is negative, a sign. Thus, the value of w should be at least 1 greater than the value of d, and at least 2 greater if the number can be negative. If insufficient positions are provided for the integer portion, including decimal point and sign (if any), asterisks are written instead of the number. If excess positions are provided, the number is preceded by blanks. Fractional digits in excess of the number specified by d are dropped after rounding.

The following example shows how each of the quantities on the left is printed according to the specification F5.2:

Internal Value	Printed
12.125	12.13
-41.5	*****
-0.25	-0.25
7.375	ƀ7.38
-1.	-1.00
9.03125	ƀ9.03
187.625	*****
0.00390625	ƀ0.00
0.0078125	ƀ0.01

The last two examples demonstrate the effect of rounding.

D and E Format Codes (aDw.d, aEw.d)

D and E format codes are used in transmitting real (single precision or double precision) data. On input, the number optionally has a decimal point and/or a D, E, or signed integer constant exponent. All exponents must be preceded by a constant—that is, an optional sign followed by at least one decimal digit, with or without a decimal point. If the decimal point is present, its position overrides the position indicated by the d portion of the format specification, and the number of positions specified by w must include a place for it. Because leading, trailing, and embedded blanks are treated as zeros, embedded and trailing blanks affect the value of the item.

The D, E, and signed integer constant exponent specifications for input data are interchangeable. For example, given a REAL*4 item in an input list, and E format specification, the exponent specification in the data item can be a D, an E, or a signed integer constant, or have no exponent. The data item is treated as a REAL*4 constant in any case. Similarly, if the list item is REAL*8 and the FORMAT specification is D, the data item is treated as a double precision constant regardless of its exponent specification, if any. Note that the type and length of the list item must agree with that of the specification.

For output, unless a scale factor is present (the scale factor changes the location of the decimal point in real numbers; its use is explained later in this section under the heading *Scale Factor (nPc),* output consists of an optional sign (required if the value is negative), a decimal point, the

number of significant digits specified by d, and a D or E exponent requiring four positions: the D or E, a + or − sign, and a two-digit exponent. The w specification must provide spaces for all of these positions. Thus the value of w should always be at least d + 5, or d + 6 if the number can be negative. If additional space is available, a leading zero is written before the decimal point. If the value of w is not sufficient to print a decimal point and a four-position exponent, plus a minus sign if the value is negative, asterisks are printed instead of the number. Fractional digits in excess of the number specified by d are dropped after rounding.

Examples:

Given the following input record:

```
b6.73124E3bbbbbbb.23791D-06bb167123.141591.413962D-01b1.413962D+1
```

the following statements could be used to read this data:

```
      READ (1,10)A,B,I,C,D,E
10    FORMAT (E10.5,D16.10,I7,E7.5,2D12.7)
```

Then the data could be printed using essentially the same format codes:

```
20    FORMAT ('1',E10.5/'   ',D16.10/'   ',I7/'   ',
      E7.5,2(/' ',D12.7))
```

The '1' and ' ' specifications are carriage and record control specifications that are explained under the heading *Printed Output*. The following shows the printed results:

```
.67312E+04
0.2379100000D-06
  16712
*******
.1413962D+00
.1413962D+02
```

In the first data item, the decimal point was moved to the left and exponent adjusted. For the second data item, a leading zero was added for the integer portion, trailing zeros were added to the decimal portion, and the number is printed with ten digits after the decimal point. For the third item, two leading blanks were added to the integer field and the number is printed right-justified. Asterisks were printed instead of the fourth data item because the w specification, although sufficient for input (there was no exponent) is insufficient for output (where there is always an exponent with the D and E format codes). For the last two data items, identical except for their exponents, the decimal points were moved to the left and the exponents adjusted accordingly.

Scale Factor (nPc)

The P scale factor can be specified as the first part of a D, E, or F format code to change the location of the decimal point in real numbers.

Unless there is an exponent in the external input or output field, the effect of the scale factor for the F format code is:

external number = internal number x 10^n

where n is the scale factor.

On input, the scale factor in the format specification is ignored for any data item with an exponent in the external field. Otherwise, a positive scale factor decreases the magnitude of the data item and a negative scale factor increases its magnitude. For example, if the input data is in the form xx.xxxx and is to be used internally in the form .xxxxxx, then the format code used to effect this change is 2PF7.4. Or, if the same input is to be used in the form xxxx.xx, then the format code used to effect this change is -2PF7.4. If the external representation was xx.xxxxExx and the format code 2PE10.4, the scale factor would be ignored, and the value stored internally as xx.xxxxExx.

On output, the scale factor can be specified for F, E, or D format codes. For F format codes, the effect of the scale factor is the opposite of that for input; a positive scale factor increases the magnitude of the number and a negative scale factor decreases the magnitude. For example, if the number has the internal form xx.xxxx and it is to be written out in the form xxxx.xx, the format code used to effect this change is 2PF7.2.

For D or E format codes, the exponent is adjusted so that the magnitude of the number does not change. For example, if the internal number were printed according to the format E10.3, it would appear as 0.238E+03. If it were printed according to the format 1PE10.3, it would appear as 2.385E+02. Note that this results in greater precision.

Once a scale factor is established, it applies to all subsequently interpreted D, E, and F format codes in the FORMAT statement until another scale factor is established. A factor of 0 may be used to discontinue the effect of a previous scale factor. If no scale factor is given, 0 is used for all F, E, and D format codes.

Example:

```
30 FORMAT (E10.3,2PD20.10,E7.2,I5)
```

The scale factor 2 applies to both the D20.10 and the E7.2 specifications. If 5 data items were read using this code, it would also apply to the E10.3 code, which would be used to interpret the fifth item. To discontinue the effect of the code after the D20.10 specification, the statement should be coded:

```
30 FORMAT (E10.3,2PD20.10,0PE7.2,I5)
```

To discontinue it after the E7.2 code, if more than four data items are to be read using the FORMAT statement, the statement should read:

```
30 FORMAT (0PE10.3,2PD20.10,E7.2,I5)
```

Note that the 0PE10.3 specification is not necessary to discontinue the effect of a scale factor in a previous FORMAT statement, or in the previous use of this FORMAT statement.

FORMAT CODES FOR ALPHAMERIC DATA

There are three specifications available for input/output of alphameric information. The specification wH or a literal enclosed in apostrophes is used for alphameric data that is not going to be processed by the object program; the specification Aw is used for alphameric data that is to be processed by the program.

Information handled with the A specification is given a variable or array name and hence can be referred to by this name for processing and/or modification. Information handled with the H format code is not given a name and cannot be referred to or manipulated in any way.

A Format Code (aAw)

The specification Aw causes w characters to be read into, or written from, a variable or array element. The type of the variable or array is immaterial, because no conversion takes place. Thus, the A format code can be used for numeric fields, but not for numeric fields requiring arithmetic.

The maximum width of w can be 255.

The maximum number of characters stored in internal storage depends on the length of the variable in the I/O list. If w is greater than the variable length, say v, then the leftmost w–v characters in the field of the input card are skipped and the remaining v characters are read and stored in the variable; truncation occurs on the left. If w is less than v, then w characters from the field in the input card are read and the remaining rightmost characters in the variable are filled with blanks.

If w is greater than the length (v) of the variable in the I/O list, then the output field contains v characters right-justified in the field, preceded by leading blanks. If w is less than v, the leftmost w characters from the variable are printed and the rest of the data is truncated; truncation occurs on the right.

Example 1:

Assume that B was specified as REAL*8, that N and M are INTEGER*4, and that the following statements are given:

```
25    FORMAT (3A7)
      READ (1,25)B,N,M
```

When the READ statement is executed, one input card is read from the file associated with logical unit number 1 into the variables B, N, and M in the format specified by FORMAT statement number 25. The following list shows the values stored for the given input cards.

Input Card	B	N	M
ABCDEFG46ƀATƀ11234567	ABCDEFGƀ	ATƀ1	4567
HIJKLMN76543213334445	HIJKLMNƀ	4321	4445

Example 2:

Assume that A and B are real variables of length 4, that C is a real variable of length 8, and that the following statements are given:

```
26    FORMAT (A6,A5,A6)
      WRITE (3,26)A,B,C
```

When the WRITE statement is executed, one line is written on the file associated with logical unit number 3 from the variables A, B, and C in the format specified by FORMAT statement 26. The printed output for values of A, B, and C is as follows:

A	B	C	Printed Line
A1B2	C3D4	E5F6G7H8	ƀƀA1B2ƀC3D4E5F6G7

H Format Code (wHstring) and Literals Enclosed in Apostrophes

The specification wH is followed in the FORMAT statement by a string of w alphameric characters. For example,

24HƀTHISƀISƀALPHAMERICƀDATA

This specification can also be coded using apostrophes to enclose the string of characters:

'ƀTHISƀISƀALPHAMERICƀDATA'

The apostrophe specification method may be more convenient for specifying long character strings.

Note that blanks are considered alphameric characters and must be included as part of the count, w.

The effect of wH or literal specification depends on whether it is used with input or output.

On input, w characters, or as many characters as are enclosed in apostrophes, are extracted from the input record and replace the characters included with the specification.

On output, the string of characters following the specification or the literal string are written as part of the output record unless characters have replaced them as a result of an input operation, in which case, the replacement characters are written.

For example, suppose that the following statements are executed:

```
      WRITE (3,2)
2     FORMAT (20HTIME/QUANTITYƀREPORT)
```

These would cause the following output to be printed:

TIME/QUANTITY REPORT

On the other hand, assume that a card containing the characters ƀNOƀƀ238 is read using these statements:

```
      READ (1,1)I
1     FORMAT ('YES',I5)
```

The statement

WRITE (3,1)I

would cause the following printed output:

ƀNOƀƀ238

SPECIFYING BLANK FIELDS IN A RECORD (X FORMAT CODE)

Blank characters can be provided in an output record, or characters of an input record can be skipped, by means of the specification wX where w is the number of blanks provided or characters skipped. When the specification is used with an input record, w characters are skipped.

For example, if a card has six 10-column fields for integers, and you do not want to read the second quantity, then the statement

```
FORMAT (I10,10X,4I10)
```

can be used with the appropriate READ statement. The T format code can also be used for this purpose.

FORMATTING THE RECORD (T FORMAT CODE)

The FORMAT statement positions data from left to right, according to the specifications given within the FORMAT statement. Very often, though, it is useful to start printing other than in print position 1, or to print data at specific print locations. The T format code can be used for this purpose.

The T format code is specified as

```
Tr
```

where r is an unsigned integer constant specifying the position in the record where data transfer is to begin. On a printed line, r indicates the print position plus 1 because the first character is used for carriage control.

A blank is inserted into any character position that was not previously filled.

Example 1:

```
1 FORMAT (T61,'CALCULATION')
```

This statement prints the word *calculation* in positions 60 through 70.

Example 2:

```
2 FORMAT (T21,D20.7)
```

This statement prints the data item whose specification is D20.7 starting at position 20.

When using the T specification for printed output, the carriage control character must still be provided. If the specification T1 is used, the first character of the output image is used for carriage control. If something other than T1 is specified as the first print position, (for example T40), then a blank is used as the carriage control character.

Example 3:

```
3 FORMAT (T40,'PLUS',T1,'1MINUS')
```

This statement prints the word *minus* starting in position 1, with the 1 in the '1MINUS' specification being used for carriage control. The word *plus* is printed starting in position 39.

Note that more than one T specification can be used in a FORMAT statement, and that the order in which the print positions are specified need not be sequential.

When output is to tape or disk, the first character of the record is treated as data. In this case, the T specification represents the exact position at which transfer is to begin, rather than the position plus 1. Thus, if FORMAT statement 3 were used to write a tape or disk record, 1MINUS would be written starting in position 1, and PLUS would be written starting in position 40.

Example 4:

```
4 FORMAT (T41,F7.2,T1,' X=',T61,'RESULT')
```

This statement prints the following line:

```
X=           1234.56       RESULT
↑            ↑             ↑
Print        Print         Print
Position 1   Position 40   Position 60
```

The T format code can be used in conjunction with any other format code. The positions of all codes following a T format code are in effect governed by the T format code until another T format code is encountered.

Example 5:

```
5 FORMAT (T101,F10.3,I10,' NOTE 1',T51,E9.3,
          T1,' ANS = ')
```

This statement prints the following line:

ANS =	0.354E+02	111082.986	536453	NOTE 1
↑	↑	↑	↑	↑
Print Position 1	Print Position 50	Print Position 100	Print Position 110	Print Position 120

When formatting a line using the T format code, care must be taken not to overlap print positions.

Example 6:

```
6 FORMAT (T41,D20.15,T51,F6.3)
```

If the preceding statement were used, the F6.3 specification would be written over six of the decimal places of the D20.15 specification.

The T format code can also be used for input.

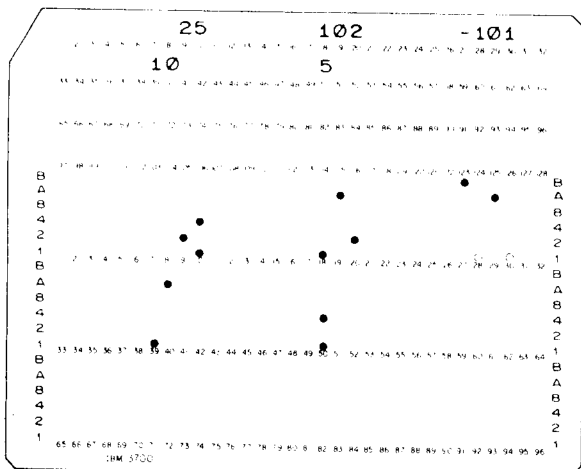
Example 7:

```
      READ (1,7) INPUT
  7   FORMAT (T15,I5)
```

These statements cause the first 14 columns of the input record to be ignored, and the next five to be transmitted to the variable INPUT.

LISTS FOR TRANSMISSION OF DATA

The list in an input/output statement specifies what quantities to transmit. For example, assume that a card is punched as follows:



Further assume that the following statement appears in the source program:

```
      READ(1,100) I,J,K,L,M
100   FORMAT(5I10)
```

The card is read and the program processes the data as though the following statements were written:

```
I=25
J=102
K=-101
L=10
M=5
```

If control passes back to the READ statement, I, J, K, L, and M have new values depending upon what is punched in the next card read.

Indexing in Input/Output Lists

DO-type notation can be used in lists for the transmission of data. For example, suppose you want to transmit the five quantities A(1), A(2), A(3), A(4) and A(5). This can be accomplished by writing:

```
10   FORMAT(5F10.0)
12   READ(1,10)(A(I),I=1,5)
```

The above statements cause a record to be read and cause the value contained in the first ten positions of the record to be converted to a real number and stored into A(1), the next ten positions into A(2), etc.

This is equivalent to writing

```
12   READ(1,10)A(1),A(2),A(3),A(4),A(5)
```

In other words, I is given the value 1 and the first quantity becomes the value of A(1). I is then increased by 1, and the second quantity becomes the value of A(2). This continues until the fifth quantity read becomes the value of A(5).

As with DO statements, a third indexing parameter can be used to specify the amount by which the index is incremented at each iteration. Thus,

```
READ(1,50) (A(I),I=1,10,2)
```

causes transmission of values for A(1), A(3), A(5), A(7) and A(9).

Implied DO Notation in Input/Output Lists

The general form of the Implied DO notation is:

$(y_1\ (i), y_2\ (i), \ldots, y_n\ (i), i=m_1, m_2, m_3)$

where

y is an array name.

i is composed of one through three integer variables, separated by commas, representing a value of a subscript in the array, based on the value of m_1.

m_1, m_2, and m_3 are each either an unsigned integer constant or an integer variable. If m_3 is not stated, it is assumed to be 1.

The maximum nesting level in implied DOs is 15.

Note: As with DOs, i is the index, m_1 is the initial value, m_2 is the test value, and m_3 is the increment. In addition, this notation can be nested.

Example:

```
((C(I,J),D(I,J),I=1,5),J=1,4)
```

transmits data in the form

```
C(1,1),D(1,1),C(2,1),D(2,1), . . . ,C(5,1),D(5,1),
C(1,2),D(1,2), . . . ,C(5,4),D(5,4)
```

Additional Details of Input/Output Lists

Any number of quantities can appear in a single list. Integer and floating point quantities can be transmitted by the same statement. However, if formatted I/O is used, each quantity must have the correct format as specified in a corresponding FORMAT statement.

For formatted I/O, if there are more quantities to transmit than there are in the list, only the number of quantities specified in the list are transmitted, and remaining quantities are ignored. Thus, if a card contains three quantities and a list contains two, the third quantity is not used by the program.

A list for an unformatted READ must not contain more quantities than the input record.

When an array name appears in a list in nonsubscripted form, all of the quantities of the array receive data or are transmitted. For example, if A is an array with 25 elements, the statement

```
READ(1,100)A
```

causes all of the quantities A(1), . . . A(25) to receive data.

A more complex list is

```
A,B(3),(C(I),D(I,K),I=1,10),((E(I,J),I=1,10,2),
F(J,3),J=1,K)
```

This list receives or transmits data in the order

```
A,B(3),C(1),D(1,K),C(2),D(2,K), . . . ,C(10),D(10,K),
E(1,1),E(3,1), . . . ,E(9,1),F(1,3),E(1,2),E(3,2), . . . ,
E(9,2),F(2,3), . . . E(9,K),F(K,3)
```

Note that each item in the list is separated by a comma, that the range of the implied DO statement is clearly defined by means of parentheses, and that constants do not appear in the list except as indexing parameters or subscripts. The variable indexing parameter (K) is assumed to be previously defined by the program, although in an input list it could have been defined by an item in the list itself, providing that it appeared *before* its use as an index.

PRINTED OUTPUT

When formatted records are prepared for a printer, the first character of the record is not printed, it is treated as a carriage control character. It can be specified in the FORMAT statement with either of the two forms of literal data: either 'x' or nHx, where x is one of the following:

x	Meaning
blank	Advance one line before printing
0	Advance two lines before printing
1	Advance to first line of next page
+	No advance

Due to hardware restrictions, the 3284 printer does not support the + carriage control for no advance. A print request to the 3284 with a + for carriage control will be equivalent to the blank carriage control character.

The carriage control character can stand by itself as a format code, or it can be included as part of a larger literal specification. Consider the following statements:

```
20   FORMAT ('1THE FOLLOWING IS A LIST OF
     PRIMES')
            or
20   FORMAT ('1',33HTHE FOLLOWING IS A LIST
     OF PRIMES)
            or
20   FORMAT (34H1THE FOLLOWING IS A LIST
     OF PRIMES)
```

Each of these statements would print the heading at the top of the next page.

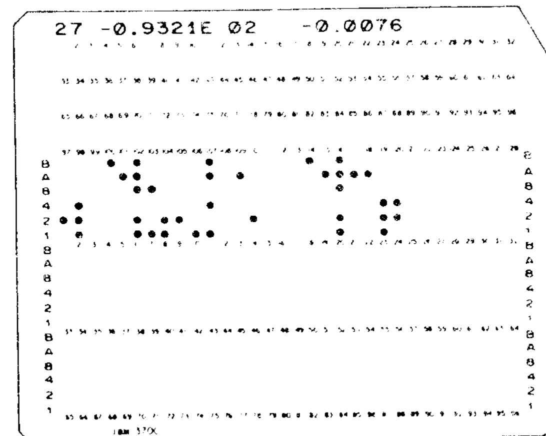
For media other than the printer (for example, tape, disk, or card punch), the first character of the record is treated as data, and a carriage control character should not be specified. Thus, the same FORMAT statement should not be used for both printing and punching. An example specifying the use of FORMAT statements for both printing and punching may be found in *Directing Program Output to Both a Printer and a Card Punch* in the section *Programming Considerations* in Part 2 of this publication.

DATA INPUT TO THE OBJECT PROGRAM

Numeric input data to be read by a READ statement when the object program is executed must be in essentially the same format as given in the previous examples. Thus, a card to be read according to

```
FORMAT (I2,E12.4,F10.4)
```

might be punched:



Within each field, all information is considered right-justified; embedded blanks and trailing blanks in numeric fields are read as zeros and affect the item's value. Plus signs can be omitted or indicated by a +. Minus signs must be punched if the number is negative or has a negative exponent. Input for E and D format codes can contain any number of digits, but only the high order digits of accuracy are retained if the number exceeds the capacity of the system.

To permit economy in punching, certain relaxations in input data format are permitted.

1. Numbers for D- and E-type format codes need not have 4 columns devoted to the exponent field. The start of the exponent field must be marked by an E, or if that is omitted, by a + or - (not a blank). Thus, E2, E+2, +2, +02, E02, and E+02 are all permissible exponent fields.

2. Numbers for D, E, and F format codes need not have their decimal point punched. If it is not punched, the format specification supplies it. For example, the number -09321+2 with the code E12.4 is treated as though the decimal point were punched between the 0 and the 9. If the decimal point is punched in the card, its position overrides the position indicated in the FORMAT statement.

LIST-DIRECTED INPUT DATA

A record containing list-directed input data consists of an alternation of constants and separators. The record can be read from tape or sequential disk, in addition to unit-record devices.

An input constant can be any valid FORTRAN numeric data type. Blanks cannot be embedded in any list-directed constant since they would be interpreted as separators. Numeric constants can optionally be signed, but there must be no embedded blanks between the sign and the constant.

Each constant must agree in type with the corresponding list element. The decimal point can be omitted from a real constant. If omitted, it is assumed to follow the right-most digit of the constant.

With the exceptions noted below, a separator is either a comma or a blank. In addition, for console input, an end indicator is a separator. For punched card input, an end-of-card condition is also a separator. Blanks can optionally occur between the comma and the carrier return or end-of-card.

A separator can be surrounded by any number of blanks, horizontal tabs, carrier returns, or end-of-card conditions. Any such combination (with no intervening constants) constitutes a single separator. At the inception of execution of a list-directed READ, a preceding separator is assumed; and initial blanks, horizontal tabs, carrier returns, or end-of-card conditions, if present, are considered part of that separator.

A null item is represented by two consecutive commas with no intervening constant. Any number of blanks, horizontal tabs, carrier returns, or end-of-card conditions can be embedded between the commas. If a null item is specified, the corresponding list item is skipped; its current value remains unaltered.

A repeat factor can be specified for a constant or null item. For a constant, the form is

```
i*constant
```

and for a null item, the form is

```
i*
```

In each instance, i is a nonzero unsigned integer constant, which indicates that the following constant or null item is to occur i times. Neither of these forms can contain embedded blanks. The separators surrounding a repeated null item need not be commas.

A slash (/) serves as a special-purpose separator, indicating that no more data is to be read during the current execution of a READ statement. If the list has not been satisfied, the values of the remaining list elements remain unaltered. If the list has been satisfied, the slash is optional.

Example: A list-directed READ statement is used to read a record containing constants of various types into main storage. For this example, the character / is used to represent a carrier return. A carrier return is the terminal equivalent of an end-of-card condition.

```
READ (1,*) (ARRAY(I),I=1,50),A,B,C,D,E,F,G,H,
J,P,Q,R,S

50*0
2.17E+15,3.14E0 , 1.,2.0,0.125D-3 2*, 87,,/
```

When this statement is executed, the value 0. is read into each of the first 50 elements of ARRAY; real values are read into the variables A, B, C, D, and E; variables F and G are skipped because of the repeated null specification; the value 87. is read into the real variable H; variable J is skipped because of the null specification, and variables P, Q, R, and S receive no data because of the slash.

LIST-DIRECTED OUTPUT DATA

List-directed output data can be directed to tape or sequential disk in addition to any unit-record output device such as a printer or card punch. List-directed output data can contain any producible form of data that is readable as list-directed input. However, certain forms which are permissible as list-directed input are not produced as list-directed output. These forms are:

null items
i* repeat factor
/ special-purpose separator

In list-directed output, the width of the data field depends upon the type of variable to be written:

Type of Variable	Width of Data Field
REAL*8	24
REAL*4	14
INTEGER*4	11
INTEGER*2	6

A blank is inserted as a separator between data fields. The total width of the data fields plus separators must be considered when output is going to sequential devices.

Input/output statements transfer data and control the flow of data between main storage and an input/output device such as a card reader, printer, punch, magnetic tape unit, or disk storage unit.

Input/output statements in FORTRAN are primarily concerned with the transfer of data between main storage locations defined in a FORTRAN program and records that are external to the program. On input, data is taken from a record and placed into main storage locations that are not necessarily contiguous. On output, data is gathered from diverse main storage locations and placed into a record. An I/O list is used to specify which main storage locations are used.

System/3 FORTRAN IV provides two types of input/output statements—*sequential* and *direct-access*. Sequential I/O statements read or write records consecutively. Direct-access I/O statements generally read or write records randomly.

Most input/output devices are sequential; that is, if there are 100 records to be read, they would have to be read in the order 1, 2, 3, . . . , 99, 100. Record 57 cannot be read before record 56. Examples of sequential devices are card readers, printers, and magnetic tape.

The 5444, 5445, and 3340 disk storage devices are the only System/3 devices that can read or write records randomly; that is, in an order determined by the programmer. When records are to be read or written randomly, direct-access statements must be used. Note, however, that sequential I/O statements can be used to read or write records consecutively on the disk, and that direct-access I/O statements can also be used for reading or writing records consecutively.

Each input/output statement uses a logical unit number to specify which input/output device (or file on a device) is to be used in the operation. *Logical unit numbers* relate particular devices or files to the system. For example, in this section we use the number 1 to refer to a card reader, the number 3 for a printer, the numbers 8 and 9 for disk files, and the number 10 for a tape file. A full explanation of logical unit number assignments is contained in Part 2 of this manual, in Chapter 13, *Compilation*.

READ and WRITE statements can be used to access *formatted* records, *unformatted* records, or *list-directed* records.

A formatted record has a FORMAT statement associated with it. A FORMAT statement describes the form of the data and how it is to be transmitted. Any number of records can be read or written with one execution of a formatted READ or WRITE statement. The data in the records are converted according to specifications listed in the FORMAT statement and are assigned to (or taken from) elements listed in the READ or WRITE statement. A partially filled, formatted record used for output is padded on the right with blanks.

An unformatted record has no FORMAT statement associated with it. If the file is sequential, only one record can be transmitted per execution of an unformatted I/O statement. The unformatted READ is used to read records that were written on tape or disk by an unformatted WRITE statement. A partially filled, unformatted record used for output is padded on the right with zeros.

A list-directed record is similar to an unformatted record in that it has no FORMAT statement associated with it. Its use, however, is to transmit records to and from any sequential file, such as a card reader, printer, tape, or disk. A partially filled, list-directed record used for output is padded on the right with blanks.

Sequential Input/Output Statements

There are five sequential input/output statements: READ, WRITE, END FILE, REWIND, and BACKSPACE.

The READ and WRITE statements are used to transmit data between sequential input/output devices and main storage. The END FILE, REWIND, and BACKSPACE statements are used only for sequential disk files and tape files. For a further discussion of sequential disk files and tape files, see *Sequential Disk & Tape Programming Considerations* in Part 3, *Programming Considerations* of this publication.

READ STATEMENT

The general forms of the READ statement are:

Formatted read:	READ (u, f,END=s,ERR=t) list
Unformatted read:	READ (u,END=s,ERR=t) list
List-directed read:	READ (u,*,END=s,ERR=t) list

where

u is an unsigned integer constant or INTEGER*4 variable that is the logical unit number of the device to read from.

f is the statement number of the FORMAT statement describing the data items to read.

*specifies list-directed data mode for sequential devices, that is, without use of a FORMAT statement.

END=s is optional and specifies the number (s) of a statement to which to transfer control if end-of-file is encountered.

ERR=t is optional and specifies the number (t) of a statement to which to transfer control if a transmission error occurs during the data transfer. This parameter is ignored if the file being processed is not on a disk or tape drive.

If END or ERR is not specified, the preceding comma is omitted.

list is an I/O list; it can contain variable names, array elements, array names, or a form called an implied DO. The I/O list is optional if f is specified.

Uses: The READ statement transmits data from a device, such as a card reader or magnetic tape unit, to main storage.

Considerations/Restrictions:

1. The program terminates if the END= parameter is not specified and an end-of-file record is encountered.

2. The program terminates if an ERR= parameter is not specified and a transmission error occurs.

3. There is no end-of-file from a 5406 console keyboard.

Examples:

```
READ (9,100) D,E,F
```

The preceding formatted READ statement reads data from the file whose logical unit number is 9, (assume a disk file) into the variables D, E, and F, in the format specified by the FORMAT statement numbered 100.

```
READ (1,98) A,B,(C(I,K),I=1,10)
```

The preceding formatted READ statement reads data from the file whose logical unit number is 1 (assume a card reader) into the variables A and B, and the array elements C(1,K), C(2,K), . . . , C(10,K) in the format specified in the FORMAT statement whose statement number is 98.

```
READ (J) A,B,C
```

The preceding unformatted READ statement reads data from the file whose logical unit number is the current value of J into the variables A, B, and C.

```
READ (1,*,END=200) (ARRAY(I),I=1,25),B(1),C(6)
```

The preceding list-directed READ statement reads data from the file whose logical unit number is 1 into the 27 array elements specified by the list. If an end-of-file record is encountered, control is transferred to the statement numbered 200.

WRITE STATEMENT

The general forms of the WRITE statement are:

Formatted write: WRITE (u,f) list
Unformatted write: WRITE (u) list
List-directed write: WRITE (u,*) list

where

u is an unsigned integer constant or INTEGER*4 variable that is the logical unit number of the device to be written on.

f is the statement number of the FORMAT statement that describes the data items to be written.

* specifies list-directed data mode for sequential devices, that is, without the use of a FORMAT statement.

list is an I/O list; it can contain variable names, array elements, array names, or a form called an implied DO. The I/O list is optional if f is specified.

Uses: The WRITE statement transmits data from main storage to a device, such as a printer or disk.

Considerations/Restrictions: The END and ERR parameters cannot be specified in the WRITE statement.

Examples:

```
WRITE (3,75) A,(B(I,3),I=1,10,2),C
```

The preceding formatted WRITE statement writes data from the variables A and C and array elements B(1,3), B(3,3), B(5,3), B(7,3), and B(9,3) onto the file whose logical unit number is 3 (a printer) according to the format specified by the FORMAT statement whose number is 75.

```
WRITE (4) ARRAY
```

The preceding unformatted WRITE statement writes data from the variable ARRAY onto the file whose logical unit number is 4. If ARRAY is an array with 25 elements, all 25 elements are written. Since the record is unformatted, no FORMAT statement number is given, and none should be specified when the record is read back into storage.

```
WRITE (2,*) I,N(I)
```

The preceding list-directed WRITE statement writes the data from the variable I and the array element N(I) on the unit-record device whose logical unit number is 2.

END FILE STATEMENT

The general form of the END FILE statement is:

```
END FILE i
```

where
i is an unsigned integer constant or INTEGER*4 variable that is the logical unit number of a magnetic tape or a sequential disk file.

Uses: The END FILE statement writes an end-of-file record on the tape or sequential disk that has logical unit number i.

Considerations/Restrictions: An END FILE request followed by another END FILE to the same logical unit causes a terminating error.

Examples:

```
END FILE 10
END FILE K
```

BACKSPACE STATEMENT

The general form of the BACKSPACE statement is:

```
BACKSPACE i
```

where
i is an unsigned integer constant or INTEGER*4 variable that is the logical unit number of a magnetic tape or sequential disk file.

Uses: The BACKSPACE statement causes a tape or sequential disk file with a logical unit number of i to be backspaced one record.

Examples:

```
BACKSPACE 10
BACKSPACE K
```

REWIND STATEMENT

The general form of the REWIND statement is:

REWIND i

where

i is an unsigned integer constant or INTEGER*4 variable that is the logical unit number of a magnetic tape or sequential disk file.

Uses: The REWIND statement causes a tape or sequential disk file with a logical unit number of i to be rewound to the beginning of the file. The REWIND statement also closes the file.

Considerations/Restrictions: When a file is rewound, another REWIND request to that file is ignored.

Examples:

```
REWIND 10
REWIND K
```

Direct-Access Input/Output Statements

The direct-access statements permit a programmer to read and write records randomly from any location within a file. They contrast with the sequential input/output statements, described previously, that process records, one after the other, from the beginning of a file to its end. With the direct-access statements, a programmer can go directly to any point in the file, process a record, and go directly to any other point without having to process all the records in between.

There are four direct-access input/output statements: READ, WRITE, DEFINE FILE, and FIND.

The READ and WRITE statements transfer data into or out of main storage. These statements allow the user to specify the location within a file from which data is to be read or into which data is to be written.

The DEFINE FILE statement describes the characteristics of the file(s) to be used during a direct-access operation.

The FIND statement points to the next record required.

In addition to these four statements, the FORMAT statement (described previously) specifies the form in which to transmit data. The direct-access READ and WRITE statements and the FIND statement are the only input/output statements that can refer to a logical unit number defined by a DEFINE FILE statement.

Each record in a direct-access file has a unique record number associated with it. The programmer must specify in the READ, WRITE, and FIND statements not only the logical unit number, as for sequential input/output statements, but also the number of the record to be read, written, or found. Specifying the record number permits processing selected records of the file, instead of records in their sequential order.

The number of the record physically following the one just processed is made available to the program in an integer variable called the *associated variable*. Thus, if the associated variable is used in a READ or WRITE statement to specify the record number, sequential processing is automatically secured. The associated variable is specified in the DEFINE FILE statement, which also gives the number, size, and type of the records in the direct-access file.

For a further discussion of direct-access files, see *Direct-Access Programming Considerations* section of this publication.

DEFINE FILE STATEMENT

The general form of the DEFINE FILE statement is:

DEFINE FILE u_1 (r_1,s_1,f_1,v_1),u_2 (r_2,s_2,f_2,v_2), . . . , u_n (r_n,s_n,f_n,v_n)

where

each u is an unsigned integer constant that is the logical unit number.

each r is an unsigned integer constant that specifies the number of records in the file associated with u. System/3 does not use the number of records field (r) in the DEFINE FILE statement as a check for accessing records outside the range of the file. The range or size of the file is set by the FILE statement via the tracks or records parameter.

each s is an unsigned integer constant that specifies the maximum size of each record associated with u. The record size is measured in characters (bytes), storage locations (bytes), or storage units (words). (A storage unit is the number of storage locations divided by four and rounded to the next highest integer.) The method used to measure the record size depends upon the specification for f.

each f specifies that the file is to be read or written either with or without format control; f can be one of the following letters:

L to indicate that the file is to be read or written either with or without format control, and that the record size is measured in number of storage locations (bytes).

E to indicate that the file is to be read or written with format control (as specified by a FORMAT statement), and that the record size is measured in number of characters (bytes).

U to indicate that the file is to be read or written without format control, and that the record size is measured in number of storage units (words). The DEFINE FILE statement with U specified measures record size in four-byte words. The word size and data length should be considered when determining record length.

each v is an integer variable called an ***associated variable.*** At the conclusion of each read or write option, v is set to a value that points to the record that immediately follows the last record transmitted. At the conclusion of a find operation, v is set to a value that points to the record found. The value of the associated variable must be set before the first read or write operation on the file.

Uses: The DEFINE FILE statement describes the characteristics of one or more direct-access files. To use the direct-access READ, WRITE, and FIND statements in a program, the file must be described with a DEFINE FILE statement. Each direct-access file must be described once, and only once, in the main program.

Considerations/Restrictions:

1. The DEFINE FILE statement cannot be used in a subprogram.

2. The associated variable cannot appear in the I/O list of a READ or WRITE statement for a file with which it is associated.

3. The value of the associated variable should be set before the first read or write operation.

4. An associated variable passed to a subprogram as an argument in a CALL statement *is not* automatically updated when input/output operations are performed in the subroutine. (The associated variable *is* updated if it is passed to a subprogram via COMMON or GLOBAL, instead of through the argument list.)

Examples:

```
DEFINE FILE 8(50,100,L,I2),9(100,50,L,J3)
```

The preceding DEFINE FILE statement describes two files, referred to by logical unit numbers 8 and 9. The data in the first file consists of 50 records, each with a maximum length of 100 storage locations. The L specifies that the data is to be transmitted either with or without format control. I2 is the associated variable that points to the next record.

The data in the second file consists of 100 records, each with a maximum length of 50 storage locations. The L specifies that the data is to be transmitted either with or without format control. J3 is the associated variable that points to the next record.

If an E is substituted for each L in the preceding DEFINE FILE statement, a FORMAT statement is required and the data is transmitted under format control.

If the data is to be transmitted without format control, the DEFINE FILE statement can be written as:

```
DEFINE FILE 8(50,25,U,I2),9(100,13,U,J3)
```

DIRECT-ACCESS READ STATEMENT

The general form of the direct-access READ statement is:

```
READ (u'r,f,ERR=s) list
```

where

u is an unsigned integer constant or an integer variable of length 4 that represents a logical unit number; u must be followed by an apostrophe (').

r, the relative record number, is an integer expression that represents the relative position of a record within the file associated with u. The relative record number of the first record of a direct-access file is 1.

f is optional and, if given, specifies the statement number of the FORMAT statement that describes the data being read.

ERR=s is optional and s is the number of a statement in the same program unit as the READ statement to which control is given when a device error condition is encountered during data transfer from device to storage.

list is an I/O list; it can contain variable names, array elements, array names, or a form called an implied DO. The I/O list is optional if f is specified.

Uses: The direct-access READ statement transfers data from a direct-access device to main storage.

Considerations/Restrictions:

1. The file being read must be defined by a DEFINE FILE statement.

2. The I/O list must not contain the associated variable defined in the DEFINE FILE statement for file u.

3. The END= parameter cannot be specified in a direct-access READ statement.

Example:

```
      DEFINE FILE 8(500,100,L,ID1),9(100,28,L,ID2)
      DIMENSION M(10)
               .
               .
               .
         ID2= 21
               .
               .
               .
 10      FORMAT (5I20)
  9      READ (8'16,10) (M(K),K=1,10)
               .
               .
               .
 13      READ (9'ID2+5) A,B,C,D,E,F,G
```

In the preceding example, READ statement 9 transmits data from the file associated with logical unit number 8, under control of FORMAT statement 10; transmission begins with record 16. Ten data items of 20 characters each are read as specified by the I/O list and FORMAT statement 10. Two records are read to satisfy the I/O list, because each record, as defined by the FORMAT statement, contains only five data items (100 characters). The associated variable ID1 is set to a value of 18 at the conclusion of the operation.

READ statement 13 transmits data from the file associated with logical unit number 9, without format control; transmission begins with record 26. Data is read until the I/O list for statement 13 is satisfied or until the end of the record, whichever occurs first. Because the DEFINE FILE statement for file 9 specified the record length as 28 storage locations, the I/O list of statement 13 calls for the same amount of data (the seven variables are type real and each occupies four storage locations). The associated variable ID2 is set to a value of 27 at the conclusion of the operation. If the value of ID2 is unchanged, the next execution of statement 13 reads record 32.

The DEFINE FILE statement in the previous example can also be written as:

```
DEFINE FILE 8(500,100,E,ID1),9(100,7,U,ID2)
```

The FORMAT statement can also control the point at which reading starts. For example, if statement 10 in the example is

```
10 FORMAT (//5I20)
```

records 16 and 17 are skipped, record 18 is read, records 19 and 20 are skipped, record 21 is read, and ID1 is set to a value of 22 at the conclusion of the READ operation in statement 9.

DIRECT-ACCESS WRITE STATEMENT

The general form of the direct-access WRITE statement is:

```
WRITE (u'r,f) list
```

where

u is an unsigned integer constant or INTEGER*4 variable that represents a logical unit number; u must be followed by an apostrophe (').

r, the relative record number, is an integer expression that represents the relative position of a record within the file associated with u.

f is optional and, if given, specifies the statement number of the FORMAT statement that describes the data being written.

list is an I/O list; it can contain variable names, array elements, array names, or a form called an implied DO. It is optional if f is specified.

Uses: The direct-access WRITE statement transfers data from main storage to a direct-access device.

Considerations/Restrictions:

1. The file being written must be defined by a DEFINE FILE statement.

2. The I/O list must not contain the associated variable defined in the DEFINE FILE statement for file u.

3. The ERR= and END= parameters cannot be specified on a direct-access WRITE statement

Example:

```
        DEFINE FILE 8(500,100,L,ID1),9(100,28,L,ID2)
        DIMENSION M(10)
            .
            .
            .
        ID2=21
            .
            .
            .
 10     FORMAT (5I20)
  8     WRITE (8'16,10) (M(K),K=1,10)
            .
            .
            .
 11     WRITE (9'ID2+5) A,B,C,D,E,F,G
```

In the preceding example, WRITE statement 8 transmits data into the file associated with logical unit number 8, under control of FORMAT statement 10; transmission begins with record 16. Ten data items of 20 characters each are written as specified by the I/O list and FORMAT statement 10. Two records are written to satisfy the I/O list because each record contains five data items (100 characters). The associated variable ID1 is set to a value of 18 at the conclusion of the operation.

WRITE statement 11 transmits data into the file associated with logical unit number 9, without format control; transmission begins with record 26. The contents of 28 storage locations are written as specified by the I/O list for statement 11. The associated variable ID2 is set to a value of 27 at the conclusion of the operation. Note the correspondence between the records described (28 storage locations per record) and the number of items called for by the I/O list (7 variables, type real, each occupying four storage locations).

The DEFINE FILE statement in the example can also be written as:

```
    DEFINE FILE 8(500,100,E,ID1),9(100,7,U,ID2)
```

As with the READ statement, a FORMAT statement can also be used to control the point at which writing starts.

FIND STATEMENT

The general form of the FIND statement is:

```
    FIND (u'r)
```

where

u is an unsigned integer constant or INTEGER*4 variable that represents a logical unit number; u must be followed by an apostrophe (').

r, the relative record number, is an integer expression that represents the relative position of a record within the file associated with u.

Uses: The FIND statement updates the associated variable for the direct-access file with a logical unit number, u, to the value of the relative record number, r.

Considerations/Restrictions:

1. No actual I/O operations are performed.

2. The file referred to in the FIND statement must be defined by a DEFINE FILE statement.

Example:

```
        DEFINE FILE 8(1000,80,L,IVAR)
 10     FIND (8'50)
            .
            .
 15     READ (8'50) A,B
```

After the FIND statement is executed, the value of IVAR is 50. After the READ statement is executed, the value is 51.

General Example—Direct-Access Operations

```
      DEFINE FILE 8(1000,72,L,ID8)
      DIMENSION A(100),B(100),C(100),D(100),
      E(100),F(100)
 15   FORMAT (6F12.4)
      FIND (8'1)
      DO 100 I=1,100
100   READ (8'ID8+4,15)A(I),B(I),C(I),D(I),E(I),F(I)
       .
       .
      DO 200 I=1,100
200   WRITE (8'ID8+4,15)A(I),B(I),C(I),D(I),E(I),F(I)
       .
       .
      END
```

The general example illustrates the ability of direct access statements to gather and disperse data in an order designated by the user. The first DO loop in the example fills arrays A through F with data from the fifth, tenth, fifteenth, . . . , and five hundredth record associated with logical unit number 8. Array A receives the first value in every fifth record, B the second value and so on, as specified by FORMAT statement 15 and the I/O list of the READ statement. At the end of the READ operation, the records are dispersed into arrays A through F. At the conclusion of the first DO loop, ID8 has a value of 501.

The second DO loop in the example groups the data items from each array, as specified by the I/O list of the WRITE statement and FORMAT statement 15. Each group of data items is placed in the file associated with logical unit number 8. Writing begins at the five hundred fifth record and continues at intervals of five, until record 1000 is written, if ID8 is not changed between the last READ and the first WRITE.

The FIND statement initially sets ID8 to 1.

The specification statements are nonexecutable statements that provide the compiler with information about the nature of data used in the source program. In addition, they supply the information required to allocate locations in storage for this data.

Specifications must precede statement function definitions, which must precede the program part containing at least one executable statement. The PROGRAM statement, if present, must precede all statements in a main program. The IMPLICIT statement, if present, must precede all statements in a main program except the PROGRAM statement, and all statements in a subprogram except the FUNCTION or SUBROUTINE statement.

TYPE STATEMENTS

There are two kinds of type statements: the IMPLICIT specification statement, and the explicit specification statements, REAL and INTEGER.

The IMPLICIT statement enables you to:

- Specify the type (including length) of all variables, arrays, and user-supplied functions whose names begin with a particular letter

The explicit specification statements enable you to:

- Specify the type (including length) of a variable, array, or user-supplied function of a particular name
- Specify the dimensions of an array

The explicit specification statements override the IMPLICIT statement, which, in turn, overrides the predefined convention for specifying type.

IMPLICIT Statement

The general form of the IMPLICIT statement

IMPLICIT $type_1$ $*s_1$ $(a_{11}, a_{12}, \ldots)$, ... $type_n$ $*s_n$ $(a_{n1}, a_{n2}, \ldots)$

where

type is either INTEGER or REAL.

each *s is optional and represents one of the permissible length specifications for its associated type.

each a is a single alphabetic character or a range of characters drawn from the set A, B, . . . , Z, $, in that order. The range is denoted by the first and last characters of the range separated by a minus sign (for example, (A-D)).

Uses: This statement specifies the type (including length) of all variables, arrays, and user-supplied functions whose names begin with a particular letter. The types that a variable, array, or function can assume, along with the permissible length specifications, are as follows:

Type	Length Specification
INTEGER	2 or 4 (standard length is 4)
REAL	4 or 8 (standard length is 4)

If the standard length is desired, the *s can be omitted. If the optional length is desired, the *s must be included in the IMPLICIT statement.

Considerations/Restrictions:

1. In a main program, if IMPLICIT is specified, it must immediately follow the PROGRAM statement, or be the first statement if PROGRAM is not specified.

2. In a subprogram, if IMPLICIT is specified, it must immediately follow the FUNCTION or SUBROUTINE statement.

3. There can be only one IMPLICIT statement per program or subprogram.

4. The IMPLICIT statement overrides the predefined convention for specifying type.

Examples:

IMPLICIT REAL (A-H,O-$),INTEGER(I-N)

All variables beginning with the characters I through N are declared as INTEGER. Because no length specification is explicitly given (that is, the *s was omitted), four bytes, the standard length for INTEGER, are allocated for each variable.

All other variables (those beginning with the characters A through H and O through Z and $) are declared as REAL with four bytes allocated for each.

Note that the statement in this example defines the type for variables the same as the predefined convention.

IMPLICIT INTEGER*2(A-H),REAL*8(I-K)

All variables beginning with the characters A through H are declared as integer with two bytes allocated for each. All variables beginning with the characters I through K are declared as real with eight bytes allocated for each.

Because the remaining letters of the alphabet, L through Z (and $), are left undefined by the IMPLICIT statement, the predefined convention remains in effect. Thus, the variables beginning with the letters L, M, and N are integer, each with a standard length of four bytes, and variables beginning with the letters O through $ are real, each with a standard length of four bytes.

Explicit Specification Statements (INTEGER and REAL)

The general form of explicit specification statements is:

type*s $a_1(k_1),a_2(k_2), \ldots a_n(k_n)$

where

type is INTEGER or REAL.

*s is optional and represents one of the permissible length specifications for its associated type.

a is a variable, array, or function name.

k is optional and gives dimension information for arrays. Each k is composed of one through three unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array.

Uses: This statement specifies the type (including length) of a variable, array, or user-supplied function by its name, rather than by its initial character. This differs from the other ways of specifying the type (that is, predefined convention and the IMPLICIT statement). In addition, the information necessary to allocate storage for arrays (dimension information) can be included within the statement.

Considerations/Restrictions:

1. The explicit specification statements override the IMPLICIT statements for specifying type.

2. Explicit specification statements must follow any IMPLICIT statement, and must precede executable statements in the program.

Examples:

INTEGER*2 ITEM,VALUE

This statement declares that the variables, ITEM and VALUE, are of type integer, each with two storage locations reserved.

REAL*8 BAKER,HOLD,VALUE,ITEM(5,5)

This statement declares that the variables, BAKER, HOLD, VALUE, and the array named ITEM, are of type real with a length of 8 bytes. In addition, it declares that the array named ITEM has 200 bytes reserved (8 for each element in the array). Note that when the length is associated with the type (for example, REAL*8), the length applies to each variable in the statement.

In the same manner in which the IMPLICIT statement overrides the predefined convention, the explicit specification statements override the IMPLICIT statement and predefined convention. If the length specification (*s) is omitted, the standard length per type is assumed.

DIMENSION STATEMENT

The general form of the DIMENSION statement is:

DIMENSION $a_1(k_1), a_2(k_2), \ldots, a_n(k_n)$

where

a is an array name.

k is composed of one to three unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array.

Uses: This statement provides information necessary to allocate storage for arrays used in the source program.

Considerations/Restrictions: The DIMENSION statement must precede executable statements and any DATA statements initializing array a.

Examples:

```
DIMENSION A(10),ARRAY(5,5,5),LIST(10,100)
DIMENSION B(25,25),TABLE(5,10,15)
```

The first statement defines three arrays, A, ARRAY, and LIST. The array, A, is a single dimension array consisting of ten elements. The array, ARRAY, is a three-dimensional array; and LIST is a two-dimensional array. The second statement defines a two-dimensional array, B, and a three-dimensional array, TABLE.

COMMON STATEMENT

The general form of the COMMON statement

COMMON $a_1(k_1), a_2(k_2), \ldots a_n(k_n)$

where

a is a variable name or array name that is not a dummy argument.

k is optional and is composed of one through three unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array.

Uses: The COMMON statement is used to share storage by two or more program units, and to specify the names of variables and arrays that are to occupy this area. Storage sharing can be used for two purposes: to conserve storage, by avoiding more than one allocation of storage for variables and arrays used by several program units; and to make arguments available between a calling program and a subprogram. Arguments passed in a common area do not appear in the argument lists of either the calling program or subprogram. Arguments in common are subject to the same rules with regard to type, length, etc., as arguments passed in an argument list. (These rules are described in the section about subprograms.)

Considerations/Restrictions:

1. There is no restriction as to the number of program units that can have COMMON statements, except that a COMMON statement in a single program unit serves no purpose other than strictly ordering the arrangement of variables in common. It should normally have at least one counterpart in another program unit.

2. There can be more than one COMMON statement in a program unit. Only one common storage area is allocated, containing all the variables and arrays in the order of their specification. A variable or array name cannot appear more than once in a COMMON statement, in more than one COMMON statement in the same program unit, or in both a COMMON and GLOBAL statement.

3. Not all program units need refer to all of the variables and arrays in common. Thus, in order to maintain correct positioning, so-called *dummy* variables can be inserted into the COMMON statement list. These dummy arguments are not referred to anywhere else in the program unit. Their function is to allow you to position variable and array names that otherwise would be in the wrong locations in a COMMON statement.

   Example:

Main Program	Subprogram
COMMON A,B,C,D	COMMON DUMMY1,BETA, DUMMY3,DELTA

4. Because the main program and subprograms have access to common storage locations via the COMMON statement, they have a way of communicating with each other. This means that a value computed in one program unit and placed in common storage can be used by another program unit in much the same manner as if it were passed as an argument. This idea becomes clearer when CALL statements and function references are discussed, later in this part of the publication.

5. The GLOBAL statement can be used to pass additional variables to some subprograms that are not needed by other subprograms. (The GLOBAL statement is fully described in the section *Interprogram Communication.*)

   Example:

```
PROGRAM MAIN
GLOBAL A,B,C
COMMON X,Y,Z
CALL SUB1
CALL SUB2


SUBROUTINE SUB1
GLOBAL A1,A2,A3


SUBROUTINE SUB2
COMMON B1,B2,B3
```

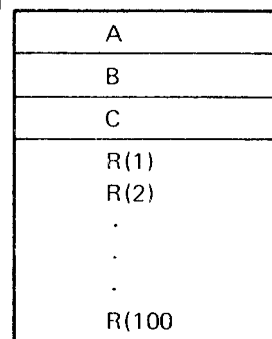
   SUB1 would share with MAIN variables A, B, and C, while SUB2 would share X, Y, and Z. In this case, it is not necessary to introduce dummy variables into the common block.

Example 1:

Because the entries in a common area share storage locations, the order in which they are entered is significant when the common area is used to transmit arguments. Consider the following example:

Main Program	Subprogram
COMMON A,B,C,R(100)	SUBROUTINE MAPMY
REAL A,B,C	COMMON X,Y,Z,S(100)
INTEGER R	REAL X,Y,Z
.	INTEGER S
.	.
.	.
CALL MAPMY	.
.	
.	
.	

The statement COMMON A,B,C,R(100) in the main program reserves 412 storage locations (four locations per variable) in the following order:



4 Storage Locations

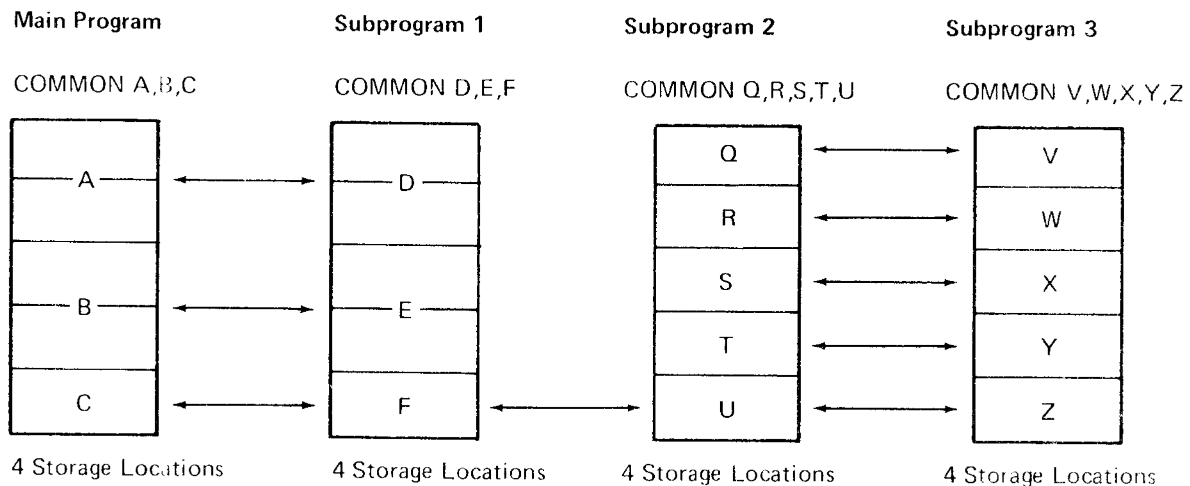
The statement COMMON X, Y, Z, S(100) in the subprogram then causes the variables X, Y, Z, and S(1), . . . ,S(100) to share the same storage space as A, B, C, and R(1), . . . , R(100).

Example 2:

Assume a common area is defined in a main program and in three subprograms as follows:

Main Program	COMMON	A,B,C (A and B are 8 storage locations; C is 4 storage locations.)
Subprogram 1	COMMON	D,E,F (D and E are 8 storage locations; F is 4 storage locations.)
Subprogram 2:	COMMON	Q,R,S,T,U (4 storage locations each.)
Subprogram 3:	COMMON	V,W,X,Y,Z (4 storage locations each.)

The correspondence of these variables within common can be illustrated as follows:



The main program can transmit values for A, B, and C to subprogram 1, provided that C is of the same type as F. However, the main program and subprogram 1 cannot, by assigning values to the variables A and B, or D and E, transmit values to the variables Q, R, S, and T in subprogram 2, or V, W, X, and Y in subprogram 3, because the lengths of their common variables differ. Likewise, subprograms 2 and 3 cannot transmit values to variables A and B, or D and E.

Values can be transmitted between variables C, F, U, and Z, assuming that each is of the same type. With the same assumption, values can be transmitted between Q and V, R and W, S and X, and T and Y. Note, however, that assignment of values to A or D destroys any values assigned to Q, R, V, and W, (and vice versa) and that assignment to B and E destroys the values of S, T, X, and Y (and vice versa).

EQUIVALENCE STATEMENT

The general form of the EQUIVALENCE statement is:

EQUIVALENCE $(a_{11},a_{12},a_{13},\ldots),(a_{21},a_{22},a_{23},\ldots),\ldots$

where

a is a variable or array element. a cannot be a dummy argument. The subscripts of array elements can have either of two forms:

If the array element has a single subscript, the subscript refers to the linear position of the element in the array (that is, its position relative to the first element in the array: third element, seventeenth element, two hundred fifty-ninth element).

If the array element has more than one subscript (with the number of subscripts equal to the number of dimensions of the array), the subscript refers to position in the same manner as in an arithmetic statement (that is, its position relative to the first element of each dimension of the array). In either case, the subscripts themselves must be unsigned integer constants.

Uses: The EQUIVALENCE statement provides the option for controlling the allocation of data storage within a single program unit. In particular, when the logic of the program permits it, the number of storage locations used can be reduced by causing locations to be shared by two or more variables (of the same or different types). Equivalence between variables implies storage sharing.

Considerations/Restrictions:

1. a cannot be a dummy argument.

2. All the elements within a single set of parentheses share the same storage locations. The order of appearance of names within an equivalence group is immaterial.

3. Mathematical equivalence of variables or array elements is implied only when they are of the same type, when they share exactly the same storage, and when the value assigned to the storage is of that type.

4. Because arrays are stored in a predetermined order as discussed previously, equivalencing two elements of two different arrays will implicitly equivalence other elements of the two arrays. (The one exception is when the first element of an array is equivalenced to the last element of another array.) The EQUIVALENCE statement must not contradict itself or any previously established equivalences.

5. The EQUIVALENCE statement is the only statement in which a single subscript can be used to refer to an element (or elements) in a multidimensional array.

6. Variables that appear in COMMON or GLOBAL statements cannot be made equivalent to each other. However, a variable can be made equivalent to a variable in common or global. If the variable that is equivalenced to a variable in common or global is an element of an array, the implicit equivalencing of the rest of the elements of the array can extend the size of common or global as shown in the following examples. The size of common or global cannot be extended so that elements are added before the beginning of the established common or global area.

Example 1:

Assume that in the initial part of a program, an array, C, of size 10x10 is needed; in the final stages of the program, C is no longer used, but arrays A and B of sizes 5x5 and 10, respectively, are used. The elements of all three arrays are of the type REAL*4. Storage space can then be saved by using the statements:

```
DIMENSION C(10,10),A(5,5),B(10)
EQUIVALENCE (C(1),A(1)),(C(26),B(1))
```

The array A, which has 25 elements, can occupy the same storage as the first 25 elements of array C because the arrays are not both needed at the same time. Similarly, the array B can share storage with elements 26 to 35 of array C.

Example 2:

```
DIMENSION B(5),C(10,10),D(5,10,15)
EQUIVALENCE (A,B(1),C(5,3)),(D(5,10,2),E)
```

This EQUIVALENCE statement specifies that the variables A, B(1), and C(5,3) are assigned the same storage locations and that variables D(5,10,2) and E are assigned the same storage locations. It also implies that the array elements B(2) and C(6,3), etc., are assigned the same storage locations. Note that further equivalence specification of B(2) with any element of array C other than C(6,3) is invalid.

Example 3:

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

This establishes a common area containing the variables A, B, and C. The EQUIVALENCE statement would then cause the variable D(1) to share the same storage location as B, D(2) to share the same storage location as C, and D(3) would extend the size of the common area, in the following manner:

```
A         (lowest location of the common area)
B,D(1)
C,D(2)
  D(3)    (highest location of the common area)
```

The following EQUIVALENCE statement is invalid:

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

because it would force D(1) to precede A, as follows:

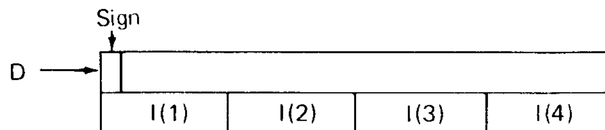
```
  D(1)
A,D(2)    (lowest location of the common area)
B,D(3)
C         (highest location of the common area)
```

Example 4:

A real variable (REAL*8) is equivalenced to several elements in an integer (INTEGER*2) array. This can be done to address two bytes of an eight-byte field. The storage space can be shared by using the following statements:

```
REAL*8 D
INTEGER*2 I(4)
EQUIVALENCE (D,I(1))
```

The four elements of array I share storage with variable D as follows:



OTHER SPECIFICATION STATEMENTS

There are four other specification statements: EXTERNAL, GENERIC, PROGRAM, and GLOBAL. EXTERNAL and GENERIC are discussed in the following chapter, which deals with subprograms. PROGRAM and GLOBAL are discussed in the chapter *Interprogram Communication.*

DATA INITIALIZATION STATEMENT

The general form of the DATA statement is:

DATA $k_1/i_1*d_1/,k_2/i_2*d_2/, \ldots ,k_n/i_n*d_n/$

where

k is a list containing variables, array elements (in which case, the subscript quantities must be unsigned integer constants), or array names.

d is a list of constants (integer, real, hexadecimal, or literal).

i* is optional and is an unsigned integer constant appearing before d, indicating that d is to be specified i times.

Uses: This statement defines initial values of variables, array elements, or arrays.

Considerations/Restrictions:

1. There must be a one-to-one correspondence between data elements (k) and initializing constants (d). However, this correspondence can be violated when initializing arrays with literal data. An array element can be initialized by subscripting the array name. Only one element is initialized this way. If any excess characters are specified, they are truncated and not placed into the next element. If there are not enough characters specified, the element is padded on the right with blanks.

   Several consecutive elements of an array can be initialized with a single literal constant by specifying the array name without a subscript. Data spill (overflow data from one element to the next) occurs through as many elements as necessary to insert the constant. If the last element initialized is only partially filled, it is padded on the right with blanks. (Any subsequent array elements are not initialized.) Truncation occurs if the constant exceeds the limit of the array.

2. For real and integer types, each constant must agree in type with the variable or array element it is initializing.

3. A variable, array element, or array that is in a common or global area cannot be initialized with a DATA statement.

4. Dummy arguments cannot appear in list k.

5. The DATA statement must precede any statement that uses the initialized variable, and must follow any specification statement describing the variable.

Examples:

```
DIMENSION D(5,10)
DATA A,B,C/5.0,6.1,7.3/,D,E/25*1.0,25*2.0,5.1/
```

The DATA statement indicates that the variables A, B, and C are to be initialized to the values 5.0, 6.1, and 7.3 respectively. In addition, the statement specifies that the first 25 elements of the array D are to be initialized to the value 1.0, the remaining 25 elements of D to the value 2.0, and the variable E to the value 5.1.

```
DIMENSION A(5), B(3,3)
DATA A/5*1.0/, B/9*2.0/,C/'FOUR'/,STRING/Z0F/
```

The DATA statement specifies that all the elements in the arrays A and B are to be initialized to the values 1.0 and 2.0, respectively. In addition, the variable C is initialized with the literal data constant FOUR, and the variable STRING is initialized with the hexadecimal data 0F.

```
DIMENSION A(10),B(9),C(2)
DATA A(1),A(2),A(4),A(5)/'ABCD','QRSTUVW',
 '123','6666'/
DATA B/'ABCDEFGHIJKLMNOPQRSTUVWXYZ'/
DATA C/'ABCDEFGHIJKL'/,X/'MNOP'/
```

Through the DATA statements above, storage would be initialized as follows:

A(1) contains ABCD.

A(2) contains QRST.

A(3) is not initialized (spill does not occur for a subscripted array name).

A(4) contains 123ƀ.

A(5) contains 6666.

A(6) through A(10) are not initialized.

B(1) contains ABCD.

B(2) contains EFGH.

B(3) contains IJKL.

B(4) contains MNOP.

B(5) contains QRST.

B(6) contains UVWX.

B(7) contains YZƀƀ.

B(8) is not initialized.

B(9) is not initialized.

C(1) contains ABCD.

C(2) contains EFGH. (The remainder of the constant is truncated.)

X contains MNOP.

It is sometimes necessary to write a program which, at various points, requires the same computation to be performed with different data for each calculation. It would simplify the writing of that program if the statements required to perform the desired computation could be written only once and then referred to freely, with each subsequent reference having the same effect as though these instructions were written at the point in the program where the reference was made.

For example, to take the cube root of a number, a program must be written with this object in mind. If a general program were written to take the cube root of any number, it would be desirable to be able to combine that program (or subprogram) with other programs where cube root calculations are required.

The FORTRAN language provides for the above situation through the use of subprograms. There are two classes of subprograms: FUNCTION subprograms and subroutine subprograms. In addition, there is a group of FORTRAN-supplied subprograms. Function subprograms return at least one value to the calling program, whereas subroutine subprograms need not return any. In addition, the method of referring to the two kinds of subprograms is different.

A subprogram must never refer to itself directly or indirectly.

Statement functions are also discussed in this section because they are similar to function subprograms. The difference is that subprograms are not in the same program unit as the program unit referring to them, whereas statement function definitions and references are in the same program unit.

Naming Subprograms

A subprogram names consists of from one through six alphameric characters, the first of which must be alphabetic but not a $. A subprogram name cannot contain special characters. The type of a function determines the type of the result that can be returned from it.

Type Declaration of a Statement Function: The declaration can be accomplished in one of three ways: by the predefined convention, by the IMPLICIT statement, or by the explicit specification statements. Thus, the rules for declaring the type of variables apply to statement functions.

Type Declaration of FUNCTION Subprograms: The declaration can be made by the predefined convention, by the IMPLICIT statement, by an explicit specification in the FUNCTION statement, or by an explicit specification statement within the function subprogram. Note that if the predefined convention is not used, the function must specify the type both in the function subprogram and in each program unit that refers to the function.

No type is associated with a subroutine name because the types of results that are returned to the calling program are dependent only on the types of the variable names appearing in the argument list of the calling program and the implicit arguments in common or global.

STATEMENT FUNCTIONS

A function is a statement of the relationship between a number of variables. To use a function in FORTRAN, it is necessary to:

1. Define the function (that is, specify which calculations to perform)

2. Refer to the function by name when you want to use it in the program

Function Definition

There are three steps in the definition of a function in FORTRAN:

1. The function must be assigned a name by which it can be called

2. The dummy arguments of the function must be stated

3. The procedure for evaluating the function must be stated

Items 2 and 3 are discussed in detail in the sections dealing with the specific subprograms, statement functions and FUNCTION subprograms.

Function Reference

When the name of a function, followed by a list of its arguments, appears in any FORTRAN expression, it refers to the function and performs the computations as indicated by the function definition. The resulting quantity (the function value) replaces the function reference in the expression and assumes the type of the function. The type of the name used for the reference must agree with the type of the name used in the definition.

Using the result of one function as an argument to another function is called nesting. Nesting of functions cannot be more than 20 levels deep.

The general form of a statement function definition is:

name(a_1,a_2,a_3, . . . ,a_n)=expression

where

name is the statement function name. The name consists of from one to six alphabetic or numeric characters, the first of which must be alphabetic but not $.

a is a dummy argument. It must be a distinct variable (it can appear only once within the list of arguments). There must be at least one dummy argument; and no more than 15.

expression is any arithmetic expression that does not contain array elements. Any statement function appearing in this expression must be defined previously.

Uses: A statement function definition specifies operations to be performed whenever that statement function name appears as a function reference in another statement in the same program unit.

The expression to the right of the equal sign defines the operations to perform when a reference to this function appears in a statement elsewhere in the program unit. The expression defining the function must not contain a reference to the function it is defining.

The dummy arguments enclosed in parentheses following the function name are dummy variables. The arguments given in the function reference are substituted for the dummy variables when the function reference is encountered. The same dummy arguments can be used in more than one statement function definition, and can be used as variables outside the statement function definitions. An actual argument in a statement function reference can be any expression of the same type as the corresponding dummy argument.

Considerations/Restrictions:

1. Dummy arguments can appear only once in the list of arguments.
2. There must be at least one dummy argument, (a), and no more than 15.
3. Any statement functions appearing in the expression must be defined previously.
4. The expression defining the function must not contain a reference to the function it is defining.
5. Arguments can be variables only (the expression can contain constants).
6. All statement function definitions used in a program must precede the first executable statement of the program.
7. Expressions cannot contain array elements.

Example: The statement:

```
FUNC(A,B)=3.*A+B**2.+X+Y+Z
```

defines the statement function FUNC, where FUNC is the function name and A and B are the dummy arguments. The expression to the right of the equal sign defines the operations to be performed when the function reference appears in an arithmetic statement.

The function reference might appear in a statement as follows:

```
C=FUNC(D,E)
```

This is equivalent to:

```
C=3.*D+E**2.+X+Y+Z
```

Note the correspondence between the dummy arguments A and B in the function definition and the actual arguments D and E in the function reference.

The following are *valid* statement function definitions and statement function references:

Function Definition	Function Reference
SUM(A,B,C,D)= A+B+C+D	NET=GROS-SUM(TAX,OLDAGE, HOSP,STOCK)
FUNC(Z)=A+X*Y*Z	ANS=FUNC(RESULT)

The following are *invalid* statement function *definitions*:

Function Definition	Reason
SUBPRG(3,J,K)= 3*I+J**3	Arguments must be variables.
SOMEF(A(I),B)= A(I)/B+3.	Arguments must not be array elements.
SUBPROGRAM (A,B)=A**2+B**2	Function name exceeds the limit of six characters.
3FUNC(D)=3.14*E	Function name must begin with an alphabetic character.
ASF(A)=A+B(I)	Expressions cannot contain an array element.
BAD(A,B)=A+B+ BAD(C,D)	Definition not permitted to refer to itself.
NOGOOD(A,A)= A*A	Arguments are not distinct variable names.

The following are *invalid* statement function references (the functions are defined previously):

Function Reference	Reason
WRONG=SUM (TAX,FICA)	Number of arguments does not agree with definition.
MIX=FUNC(I)	Type of argument does not agree with above definition.

FUNCTION SUBPROGRAMS

The general form of the FUNCTION statement

type FUNCTION name*s $(a_1,a_2,a_3,\ldots,a_n)$

where

type is INTEGER or REAL. Its inclusion is optional.

name is the name of the FUNCTION, consisting of from one to six alphabetic or numeric characters, the first of which must be alphabetic but not $.

s represents one of the permissible length specifications for its associated type. It can be included optionally only when type is specified.

a is a dummy argument. It must be a distinct variable or array name (that is, it can appear only once within the statement) or dummy name of a SUBROUTINE or other function subprogram. There must be at least one argument in the argument list.

Uses: The function subprogram is a FORTRAN subprogram consisting of a FUNCTION statement followed by other statements including a RETURN and an END statement. It is an independently written program that is executed when its name is referred to in another program.

A type declaration for a function name can be made by the predefined convention, by an IMPLICIT statement, by an explicit specification in the FUNCTION statement, or by an explicit specification statement within the function subprogram. The function name must also specify type in the program units that refer to it if the predefined convention is not used.

Because the subprogram is a separate program unit, there is no conflict if the variable names and statement numbers within it are the same as those in other program units.

Considerations/Restrictions:

1. The FUNCTION statement must be the first statement in the subprogram.
2. The function subprogram can contain any FORTRAN statement except:
   - a SUBROUTINE statement
   - another FUNCTION statement
   - a DEFINE FILE statement
   - a PROGRAM statement
3. If an IMPLICIT statement is used in a function subprogram, it must immediately follow the FUNCTION statement.
4. The name of the function must be assigned a value at least once in the subprogram—as:
   - the variable name to the left of the equal sign in an arithmetic statement
   - an argument of a CALL statement
   - an external function reference that is assigned a value by a subroutine referred to
   - an item in a list of a READ statement in the subprogram

5. The dummy arguments in a function subprogram cannot be redefined (that is, cannot appear to the left of an equal sign).

6. The number of dummy arguments in a FUNCTION statement cannot exceed 25.

Example:

The relationship between variable names used as arguments in the calling program and the dummy variables used as arguments in the function subprogram is illustrated in the following example:

Calling Program	Function Subprogram
.	FUNCTION CALC(A,B,J)
.	.
.	.
ANS=ROOT1* CALC(X,Y,I)	.
.	I=J*2
.	.
.	.
	.
	CALC=A**I/B
	RETURN
	END

In this example, the values of X, Y, and I are used in the FUNCTION subprogram as the values of A, B, and J respectively. The value of CALC is computed, and this value is returned to the calling program where the value of ANS is computed. The variable I in the argument list of CALC in the calling program is not the same as the variable I appearing in the subprogram.

Calling Program	Function Subprogram
INTEGER*2 CALC	INTEGER FUNCTION CALC* 2(I,J,K)
.	.
.	.
.	.
ANS=ROOT1* CALC(N,M,L)	CALC=I+J+K**2
.	.
.	.
.	.
	RETURN
	END

The function subprogram, CALC, is declared as type INTEGER of length 2.

SUBROUTINE SUBPROGRAMS

The general form of the SUBROUTINE statement is:

SUBROUTINE name $(a_1, a_2, a_3, \ldots, a_n)$

where

name is the SUBROUTINE name, consisting of from one to six alphabetic or numeric characters, the first of which must be alphabetic but not $.

a is a distinct dummy argument (that is, it can appear only once within the statement). There need not be any arguments, in which case, the parentheses must be omitted. Each argument used must be a variable or array name or the dummy name of another subroutine or function subprogram.

Uses: The subroutine subprogram is similar to the function subprogram in many respects. The rules for naming function and subroutine subprograms are similar. They both require a RETURN and an END statement, and they both contain the same sort of dummy arguments. Like the function subprogram, the subroutine subprogram is a set of commonly used computations, but unlike the function subprogram, it need not return any results to the calling program. The subroutine subprogram is referred to by the CALL statement.

Considerations/Restrictions:

1. The SUBROUTINE statement must be the first statement in the subprogram. The subroutine subprogram can contain any FORTRAN statement except a FUNCTION statement, another SUBROUTINE statement, a DEFINE FILE statement, or a PROGRAM statement. If an IMPLICIT statement is used in a subroutine subprogram, it must immediately follow the SUBROUTINE statement.

2. The subroutine subprogram can use one or more of its arguments to return values to the calling program. An argument so used appears on the left side of an arithmetic assignment statement, in the list of a READ statement within the subprogram, or as an argument in a CALL statement or function reference that is assigned a value by the subroutine referred to. The subroutine name must not appear in any other statement in the subroutine subprogram.

3. The dummy arguments ($a_1, a_2, a_3, \ldots, a_n$) are considered dummy names that are replaced at the time of execution by the actual arguments supplied in the CALL statement. Additional information about dummy arguments is in the section *Dummy Arguments in a Function or Subroutine Subprogram.*

4. Each distinct dummy argument (a) can appear only once in the list of arguments and the number of dummy arguments cannot exceed 25.

5. If there are no dummy arguments, the parentheses must be omitted.

6. A subroutine subprogram must contain an END statement and at least one RETURN statement.

7. Because the subprogram is a separate program unit, there is no conflict if the variable names and statement numbers within it are the same as those in other program units.

Examples:

```
SUBROUTINE COPY (A,B,N)
SUBROUTINE NULL
```

CALL Statement

The CALL statement is used to call a subroutine subprogram.

The general form of the CALL statement is:

CALL name ($a_1, a_2, a_3, \ldots, a_n$)

where

name is the name of a subroutine subprogram.

Each a is an actual argument that is being supplied to the subroutine subprogram. The argument can be a variable, array element, array name, constant, arithmetic expression, or subprogram name.

Uses: The CALL statement transfers control to a subroutine subprogram. The CALL statement associates the dummy arguments named in the SUBROUTINE statement with the value of the actual arguments in the CALL statement.

Considerations/Restrictions:

1. The arguments cannot be constants or expressions if the subprogram changes their value.

2. There cannot be more than 25 arguments in a CALL statement.

Examples:

```
CALL OUT
CALL MATMPY(X,5,40,Y,7,2)
CALL QDRTIC(X,Y,Z,ROOT1,ROOT2)
CALL SUB1(X+Y*5,ABDF,SINE)
```

The CALL statement transfers control to the subroutine subprogram, and associates the dummy variables with the value of the actual arguments that appear in the CALL statement.

Example:

Calling Program		**Subroutine Subprogram**
DIMENSION X(100),Y(100)		
.		SUBROUTINE COPY(A,B,N)
.		DIMENSION A(100),B(100)
.		DO 10(I)=1,N
CALL COPY (X,Y,100)	10	B(I)=A(I)
.		RETURN
.		END
.		

The preceding example shows the relationship between variable names used as arguments in the calling program and the dummy variables used as arguments in the SUBROUTINE subprogram.

Subroutine COPY *copies* array A into array B within the subprogram. In this particular call, the subroutine arrays, A and B, are associated with the calling program arrays, X and Y, respectively, and the variable N in the subroutine is associated with the value 100. Thus a call to subroutine COPY in this instance results in the 100 elements of array X being copied into the 100 elements of array Y

RETURN STATEMENT

The general form of the RETURN statement is:

RETURN

Uses: This statement is used to exit from a function or subroutine subprogram. It signifies the conclusion of a series of computations. The subprogram transmits argument values and returns control to the calling program.

Considerations/Restrictions:

1. There can be several RETURN statements in a subprogram.
2. The RETURN statement cannot appear in a main program.

Example:

```
       FUNCTION DAV(D,E,F)
       IF (D-E)10,20,30
   10  A=D+2.0*E
            .
            .
            .
    5  A=F+2.0*E
            .
            .
            .
   20  DAV=A+D**2
            .
            .
            .
       RETURN
   30  DAV=D**2
            .
            .
            .
       RETURN
       END
```

If the result of (D-E) is negative or zero, the first RETURN statement is executed. If the result is positive, the second RETURN is executed.

DUMMY ARGUMENTS IN A FUNCTION OR SUBROUTINE SUBPROGRAM

The dummy arguments of a subprogram appear after the function or subroutine name and are enclosed in parentheses. They are associated at the time of execution with the actual arguments supplied in the CALL statement or function reference in the calling program.

The dummy arguments must correspond in number, order, and type to the actual arguments. For example, if an actual argument is an integer constant, then the corresponding dummy argument must be an integer variable of length 4. If a dummy argument is an array, the corresponding actual argument must be (1) an array, or (2) an array element. In the first instance, the size of the dummy array must not exceed the size of the actual array. In the second, the size of the dummy array must not exceed the size of that portion of the actual array that follows and includes the designated element.

The actual arguments can be:

- An arithmetic constant
- Any type of variable or array element
- Any type of array name
- Any type of arithmetic expression
- The name of a function or subroutine subprogram

An actual argument that is the name of a subprogram must be identified by an EXTERNAL statement in the calling program unit containing that name. Hexadecimal constants cannot be actual arguments.

A dummy argument is an array when an appropriate DIMENSION or explicit specification statement appears in the subprogram. None of the dummy arguments can appear in an EQUIVALENCE, COMMON, or GLOBAL statement.

Dummy arguments or common or global elements cannot be assigned new values in a function subprogram. If a dummy argument is assigned a value in a subroutine subprogram, the corresponding actual argument must be a variable, an array element, or an array. A constant or expression should not be written as an actual argument unless the programmer is certain that the corresponding dummy argument is not assigned a value in the subprogram.

A referenced subroutine cannot assign new values to dummy arguments that are associated with other dummy arguments within the subroutine or with variables in common or global areas. For example, if the subroutine DERIV is defined as

```
SUBROUTINE DERIV (X, Y, Z)
COMMON W
```

and if the following statements are included in the calling program

```
COMMON B
   .
   .
   .
CALL DERIV (A,B,A)
```

then X, Y, Z, and W cannot be assigned new values by the subroutine DERIV. Dummy arguments X and Z cannot be defined because they are both associated with the same argument, A; nor dummy argument Y, because it is associated with an argument, B, which is in common; nor the variable W, because it is also associated with B.

EXTERNAL STATEMENT

The general form of the EXTERNAL statement is:

EXTERNAL $a_1, a_2, a_3, \ldots, a_n$

where
each a is a name of a subprogram that is passed as an argument to other subprograms.

Uses: The EXTERNAL statement identifies the names of subprograms that are passed as arguments to another subprogram.

Considerations/Restrictions:

1. The EXTERNAL statement is a specification statement, and must precede statement function definitions and all executable statements.

2. If the name of a FORTRAN-supplied intrinsic function is used in an EXTERNAL statement, the function from the System/3 FORTRAN library is not used when it appears as a function reference. Instead, it is assumed that the function is supplied by the user.

Examples:

The name of any subprogram that is passed as an argument to another subprogram must appear in an EXTERNAL statement in the calling program. For example, assume that SUB is a subroutine subprogram and MULT is a function subprogram in the following statements:

Calling Program	Subprogram
EXTERNAL MULT	SUBROUTINE SUB(K,M,Z)
.	IF (K)4,6,6
.	4 D=M(K,Z**2)
.	.
CALL SUB(J,MULT,C)	.
.	.
.	6 RETURN
.	END

In this example, the subprogram name MULT is used as an argument in the subprogram SUB. The subprogram name MULT is passed to the dummy variable M as are the variables J and C passed to the dummy variables K and Z, respectively. The subprogram MULT, is executed only if the value of K is negative.

Calling Program	Subprogram
.	SUBROUTINE SUB(W,X,
.	M,N)
.	.
CALL SUB(A,B,	.
MULT(C,D),37)	.
.	RETURN
.	END
.	

In this example, an EXTERNAL statement is not required because the subprogram named MULT is not an argument; it is executed first and the result becomes the argument.

AUTOMATIC FUNCTION SELECTION

The automatic function selection facility allows you to use a single generic name when requesting a FORTRAN-supplied function that has several names depending on argument type. The proper function is selected by the FORTRAN compiler, based on the type of the argument(s) of the function.

With this facility you can, for example, use the generic name, SIN, to refer to any sine routine, rather than explicitly calling SIN for REAL*4 arguments and DSIN for REAL*8 arguments. The facility is requested by including the GENERIC statement in each executable program unit in which it is to be used.

GENERIC Statement

The general form of the GENERIC statement is:

```
GENERIC
```

Uses: The GENERIC statement indicates that for FORTRAN-supplied functions having several names depending on argument type, the correct function is to be selected by the FORTRAN compiler.

The use of the GENERIC statement declares the set of names in the first column of Figure 4 to be generic. Specific built-in and library function names can be interspersed with generic names in the same program unit.

Considerations/Restrictions:

1. As a specification statement, GENERIC must precede statement function definitions and all executable program statements, and must follow any PROGRAM, FUNCTION, SUBROUTINE, or IMPLICIT statement.

2. The use of a generic name in an explicit type statement overrides its definition as generic, because generic names have no type. If the generic name of an intrinsic function appears in an EXTERNAL statement, its definition as generic is also overridden, because it is thereby considered an external procedure. (The intrinsic functions are underlined in the first column of Figure 4.) A generic name that does not coincide with an intrinsic function name can appear in an EXTERNAL statement and still be considered generic.

3. Generic names cannot be passed as arguments to external procedures. The automatic function selection facility will not substitute the appropriate function for the generic name in an argument list when the generic name is used without arguments.

   Note: There is no way to make such a selection, because the name being passed as an argument has no arguments of its own. Thus, a function name is specific for use as an argument, even if the same name is generic for use as a function reference.

Example:

```
GENERIC
EXTERNAL COS
REAL*8A,B,C,D
C=COS(A)
D=DCOS(B)
CALL SUB(COS)
```

Because automatic function selection is invoked, the function DCOS is called to calculate the value of C and D. The specific name COS is passed to the subroutine SUB. Because COS is not an intrinsic function name, it can be used in an EXTERNAL statement and still be used as a generic name.

Generic Function Name (intrinsic function)	Specific Function Name By Type and Length of Arguments Permissible			Function Value	
	INTEGER*4	REAL*4	REAL*8	Type	Length
ABS(1)	IABS	ABS	DABS	Arg ③	Arg
AINT(1)		AINT		Real	Arg
ATAN		ATAN	DATAN	Real	Arg
COS(1)		COS	DCOS	Arg	Arg
DIM(2)	IDIM	DIM		Arg	Arg
EXP(1)		EXP	DEXP	Arg	Arg
INT(1)		INT =IFIX ②	IDINT	Integer	4
LOG(1)		ALOG =LOG ①	DLOG	Arg	Arg
LOG10(1)		ALOG10 =ALOG10 ①	DLOG10	Real	Arg
MAX(≥2)	MAX0 =MAX ①	AMAX1	DMAX1	Arg	Arg
MIN(≥2)	MIN0 =MIN ①	AMIN1	DMIN1	Arg	Arg
MOD(2)	MOD	AMOD	DMOD	Arg	Arg
SIGN(2)	ISIGN	SIGN	DSIGN	Arg	Arg
SIN(1)		SIN	DSIN	Arg	Arg
SQRT(1)		SQRT	DSQRT	Arg	Arg
TANH(1)		TANH	DTANH	Real	Arg

① The function name is an alias. The functions in the left column are aliases for those in the right column. They are aliases both when GENERIC is specified and when it is not.

Alias	Function
LOG	ALOG
LOG10	ALOG10
MAX	MAX0
MIN	MIN0

② IFIX performs the same function as INT, and is shown as a member of the GENERIC family INT.

③ The abbreviation arg is used to indicate that the type and/or length of the function value is the same as the argument.

Parentheses, (), indicate the number of arguments.

Figure 4. Generic Functions

Chapter 9. Interprogram Communication

System/3 FORTRAN IV provides a means of loading and executing main programs successively. Each program can share a common storage area that is not overlaid when the next program is invoked.

Interprogram communication enables you to avoid the impasse that results when the main storage available for a FORTRAN job is insufficient for the data and object code required to process it. It also allows for more efficient execution of programs, that because of their complexity, require calls to numerous subroutines.

Interprogram communication is provided by three System/3 FORTRAN statements—PROGRAM, INVOKE, and GLOBAL. The PROGRAM statement assigns a user name to a main program that is called (loaded into main storage) with the INVOKE statement. The GLOBAL statement specifies a storage area (and the variables and arrays it contains) that will not be overlaid when the invoked program is loaded.

The following situation illustrates the usefulness of interprogram communication.

Assume that 16,000 bytes of main storage is available for use by a FORTRAN program. This program must read a large amount of data, process it, and write it out. Assume that the data requires 10,000 bytes of storage, leaving 6,000 bytes for object code which includes routines for reading, processing, and writing. If *each* of these three steps required the full amount of storage available (6,000 bytes), the program could not be executed even if it were divided into a main program and two subroutines (because a subroutine does not overlay the program that calls it). The minimum amount of storage required for processing code would still be 12,000 bytes.

However, with the INVOKE and GLOBAL statements, the program could be divided into three main programs of 6,000 bytes each, with the 10,000-byte data area designated as global. When the first main program is through with its processing, it invokes the second program, which overlays the first (occupies the same 6,000 bytes of storage). The 10,000 bytes of data in the global area is not destroyed. When the second program is through processing, it, in turn, invokes the third program.

PROGRAM STATEMENT

The general form of the PROGRAM statement is:

```
PROGRAM name
```

where name is the name of the main program. The name consists of from one to six alphabetic or numeric characters, the first of which must be alphabetic, but not $.

Uses: The PROGRAM statement assigns a name to a main program.

Considerations/Restrictions:

1. The PROGRAM statement must be specified if another program calls the main program using the INVOKE statement.
2. The PROGRAM statement must be specified if the program is to be compiled, link-edited, and stored on a disk pack (for execution at a later time).
3. A PROGRAM statement, if present, must be the first statement in a main program.
4. A PROGRAM statement cannot be used in a subprogram.

Examples:

```
PROGRAM PROG3
PROGRAM COST
```

INVOKE STATEMENT

The general form of the INVOKE statement is:

```
INVOKE name
```

where name is the name of a main program specified in a PROGRAM statement.

Uses: The INVOKE statement causes the named program to overlay the invoking program and receive control. A program that is invoked begins execution at its first executable instruction.

Considerations/Restrictions:

1. The INVOKE statement can appear in a main program or a subprogram.

2. When using the INVOKE statement on a non-DPF system, the program being invoked must not be larger than the amount of user main storage. When using the INVOKE statement on a DPF or multiprogramming system, the program being invoked must not be larger than the program that was initially loaded into main storage (the program specified on the LOAD statement). Also, the invoked program cannot be larger than the program level or partition size when using a PARTITION statement.

   The CORE statement can be included in the program initially loaded into main storage to specify the size of the largest program to be invoked (refer to *CORE Compiler Option Statement*).

3. The program being invoked and the invoking program must reside in the object library on the same drive.

4. When the invoking program and the program being invoked are both reading cards from the MFCU1 on the Model 10 or any card input device on the Model 15, that device should be single buffered. If the device is double buffered, a data record will be lost when the program is invoked.

5. If an invoking program uses a sequential disk file, a following invoked program must refer to that file by the same logical unit number.

Examples:

```
PROGRAM FIRST          PROGRAM SECOND
      .                       .
      .                       .
      .                       .
INVOKE SECOND
```

GLOBAL STATEMENT

The general form of the GLOBAL statement is:

GLOBAL $a_1(k_1),a_2(k_2), \ldots ,a_n(k_n)$

where a is the name of a variable or array. k is an optional subscript composed of from one to three unsigned integer constants separated by commas, representing the maximum value of each subscript in the array.

Uses: The GLOBAL statement provides the sharing of a main storage area, (and the variables and arrays contained in it) by two or more main programs. It is like a COMMON statement except that it can be used to communicate between two or more main programs in addition to program units in the same program.

Considerations/Restrictions:

1. The GLOBAL statement can appear in a main program or a subprogram. A program unit can contain any number of GLOBAL statements. All entries in these statements are strung together in the order of their specification. An entry cannot be specified more than once in a GLOBAL statement, in more than one GLOBAL statement, or in both a GLOBAL and COMMON statement.

2. The global data area is for interprogram communication, although a main program may share a global data area with a subprogram. The *local* COMMON statement can still be used for this intraprogram communication.

3. Rules regarding the use of equivalence are the same for a global data area as for a common data area. Specifically, the EQUIVALENCE statement cannot be used to extend the size of the global area by adding elements before the beginning of the global block. In addition, a variable or array cannot be associated by equivalence to a global variable, array, or array element if the variable or array so associated is itself in a global or common block.

Example:

```
PROGRAM FIRST          PROGRAM SECOND
GLOBAL A,B,C(5,5),     GLOBAL X,Y,Z(25),
D(10,1000)             DATA(10,1000)
                              .
      .                       .
      .                       .
      .                STOP
INVOKE SECOND          END
      .
      .
      .
END
```

The debug facility is a programming aid that enables you to locate errors in a FORTRAN source program. The debug facility traces the flow within a program, traces the flow between programs, and checks the validity of subscripts.

The debug facility consists of a DEBUG statement, an AT debug packet identification statement, and two executable statements. These statements specify the debugging operations for a single program unit in source language. (A program unit is a single main program or a subprogram.)

The source deck arrangement consists of the source language statements that constitute the program, followed by the DEBUG specification statement, followed by the debug packets, followed by the END statement.

The statements that make up a program debugging operation must be grouped in one or more debug packets. A debug packet consists of an AT statement followed by a TRACE ON or TRACE OFF statement.

DEBUG STATEMENT

The general form of the DEBUG statement is:

DEBUG option, . . . ,option

where option can be any of the following:

SUBCHK(n_1,n_2, . . . ,n_n)
: where n is an array name. The validity of the subscripts used with the named arrays is checked by comparing the subscript combination with the size of the array. If the subscript exceeds its dimension bounds, a message is placed in the debug output file. Program execution continues, using the incorrect subscript. If the list of array names is omitted, all arrays in the program are checked for valid subscript usage. If the entire option is omitted, no arrays are checked for valid subscripts.

  When subscript checking occurs, only two bytes of the subscript are examined. If an INTEGER*4 value is used as a subscript, the leftmost two bytes of the subscript value are ignored.

  SUBCHK cannot be applied to a dummy array.

TRACE
: This option must be in the DEBUG statement of each program or subprogram for which tracing is requested. If this option is omitted, there is no display of program flow by statement number within this program. Even when this option is used, a TRACE ON statement must appear in the first debug packet in which tracing is requested.

INIT(m_1,m_2, . . .,m_n)
: where m is the name of a variable or an array that is displayed in the debug output file only when the variable or the array values change.

  If m is a variable name, the name and value are displayed whenever the variable is assigned a new value in either an assignment or a READ statement.

  If m is an array name, the changed element is displayed. If the list of names is omitted, a display occurs whenever the value of a variable or an array element is changed.

  If the entire option is omitted, no display occurs when values change.

  INIT cannot be applied to a dummy variable.

SUBTRACE
: This option specifies that the name of this subprogram is displayed whenever it is entered. The message, RETURN, is displayed whenever execution of the subprogram is completed.

Uses: The DEBUG statement sets the conditions for operation of the debug facility and designates debugging operations that apply to the entire program unit.

Considerations/Restrictions:

1. There must be one, and only one, DEBUG statement for each program or subprogram to be debugged; it must follow the last executable statement (such as STOP or RETURN). The debug packets must immediately follow the DEBUG statement.

2. The options in a DEBUG statement can be given only once, can appear in any order, and must be separated by commas.

3. Output from a debug operation is directed to the FORTRAN error logging device. (See *PRINT and NOPRINTER Device Option Statements* in Chapter 13 for more information.)

AT STATEMENT

The general form of the AT statement is:

```
AT n
```

where
n is the statement number of an executable statement in the program or subprogram to be debugged.

Uses: The AT statement identifies the beginning of a debug packet and indicates the statement number in the program unit where statement tracing is to begin or end.

Considerations/Restrictions:

1. There must be one AT statement for each debug packet. (There can be many debug packets for one program or subprogram.)

2. The TRACE option must be specified in the DEBUG statement if AT is specified.

3. A TRACE ON statement or TRACE OFF statement must follow the AT statement.

Example:

```
200  X = Y + Z
     .
     .
     .
     DEBUG TRACE
     AT 200
     TRACE ON
     END
```

TRACE ON STATEMENT

The general form of the TRACE ON statement is:

```
TRACE ON
```

Uses: The TRACE ON statement initiates the display of program flow beginning at the statement number indicated by the AT statement. Each time a statement that has a statement number is executed, the statement number is printed out on the debug output file.

Considerations/Restrictions:

1. For the TRACE ON statement to be valid, the TRACE option must be specified in the DEBUG statement.

2. Tracing begins immediately before the execution of the statement specified in the AT statement.

3. The TRACE ON statement stays in effect through any level of subprogram call or return. However, if a TRACE ON statement is in effect and control is given to a program in which the TRACE option was not specified, the statement numbers in that program are not traced.

Example:

```
200  X=Y+Z
     .
     .
     .
     DEBUG TRACE
     AT 200
     TRACE ON
     END
```

TRACE OFF

The general form of the TRACE OFF statement is:

```
TRACE OFF
```

Uses: The TRACE OFF statement stops the display of program flow beginning at the statement number indicated by the AT statement.

Considerations/Restrictions: For this statement to be valid, the TRACE option must be specified in the DEBUG statement.

Example:

```
200     X=Y+Z
        .
        .
        .
210     X=Y-Z
        .
        .
        .
        DEBUG TRACE
        AT 200
        TRACE ON
        AT 210
        TRACE OFF
        END
```

EXAMPLES OF THE DEBUG FACILITY

Example 1:

```
 DIMENSION STOCK(1000),OUT(1000)
        .
        .
        .
        DO 30 I=1,1000
25      STOCK (I)=STOCK(I)-OUT(I)
30      CONTINUE
35      A=B+C
        .
        .
        .
        DEBUG SUBCHK(STOCK),INIT(STOCK)
        END
```

All of the invalid subscripts processed in STOCK, and all of the values of STOCK are to be displayed.

Example 2:

```
10      A = 1.5
12      L = 1
15      B = A + 1.5
20      DO 22 I = 1,5
        .
        .
        .
22      CONTINUE
25      C = B + 3.16
30      D = C/2
        STOP
        .
        .
        .
        DEBUG TRACE
C       DEBUG PACKET NUMBER 1
        AT 10
        TRACE ON
C       DEBUG PACKET NUMBER 2
        AT 20
        TRACE OFF
C       DEBUG PACKET NUMBER 3
        AT 30
        TRACE ON
        END
```

When statement 10 is encountered in the preceding example, tracing begins as indicated by debug packet 1. When statement 20 is encountered, tracing stops as indicated by the TRACE OFF statement in debug packet 2. When statement 30 is encountered, debug packet 3 tracing begins again.

In this example, trace output is produced for statement numbers 10, 12 15, and 30. No debug output is produced for statement numbers 20, 22, and 25.

SAMPLE PROGRAM 1

This sample program (Figure 5) is designed to find all of the prime numbers between 2 and 1000. A prime number is an integer greater than 1 that cannot be evenly divided by any integer except itself and 1. Thus 2, 3, 5, 7, 11, . . . are prime numbers. The number 9 is not a prime number because it can be evenly divided by 3.

IBM FORTRAN Coding Form

GX28 7327 U/M050 Printed in U.S.A.

PROGRAM SAMPLE PROGRAM 1 | PROGRAMMER | DATE | PUNCHING INSTRUCTIONS: GRAPHIC, PUNCH | PAGE 1 OF 1 | CARD ELECTRO NUMBER*

```
C PRIME NUMBER GENERATOR
      WRITE (3,1)
    1 FORMAT ('1FOLLOWING IS A LIST OF PRIME NUMBERS FROM 2 TO 1ØØØ'/
     *19X,'2'/19X,'3')
      DO 4 I=5,1ØØØ,2
      K=SQRT(FLOAT(I))
      DO 2 J=3,K,2
      IF (MOD(I,J)) 2,4,2
    2 CONTINUE
      WRITE (3,3) I
    3 FORMAT (I20)
    4 CONTINUE
      WRITE (3,5)
    5 FORMAT (' THIS IS THE END OF THE PROGRAM')
      STOP
      END
```

Figure 5. Sample Program 1

SAMPLE PROGRAM 2

The n points (x_i, y_i) are to be used to fit an m degree polynomial by the least-squares method.

$$y = a_o + a_1 x + a_2 x^2 + \ldots + a_m x^m$$

In order to obtain the coefficients $a_o, a_1, \ldots, a_m$, it is necessary to solve the normal equations:

$$\begin{array}{ll} (1) & W_o a_o + W_1 a_1 + \ldots + W_m a_m = Z_o \\ (2) & W_1 a_o + W_2 a_1 + \ldots + W_{m+1} a_m = Z_1 \\ & \vdots \\ (m+1) & W_m a + W_{m+1} a_1 + \ldots + W_{2m} a_m = Z_m \end{array}$$

where:

$$W_o = n \qquad Z = \sum_{i=1}^{n} y_i$$

$$W_1 = \sum_{i=1}^{n} x_i \qquad Z_1 = \sum_{i=1}^{n} y_i x_i$$

$$W_2 = \sum_{i=1}^{n} x_i^2 \qquad Z_2 = \sum_{i=1}^{n} y_i x_i^2$$

$$\vdots$$

$$Z_m = \sum_{i=1}^{n} y_i x_i^m$$

$$\vdots$$

$$Z_2 m = \sum_{i=1}^{n} x_i^{2m}$$

After the Ws and Zs are computed, the normal equations are solved by the method of elimination, which is illustrated by the following solution of the normal equations for a second degree polynomial (m = 2).

$$\begin{array}{ll} (1) & W_o a_o + W_1 a_1 + W_2 a_2 = Z_o \\ (2) & W_1 a_o + W_2 a_1 + W_3 a_2 = Z_1 \\ (3) & W_2 a_o + W_3 a_1 + W_4 a_2 = Z_2 \end{array}$$

The forward solution is as follows:

1. Divide equation (1) by W_o
2. Multiply the equation resulting from step 1 by W_1 and subtract from equation (2)
3. Multiply the equation resulting from step 1 by W_2 and subtract from equation (3)

The resulting equations are:

$$\begin{array}{ll} (4) & a_o + b_{12} a_1 + b_{13} a_2 = b_{14} \\ (5) & b_{22} a_1 + b_{23} a_2 = b_{24} \\ (6) & b_{32} a_1 + b_{33} a_2 = b_{34} \end{array}$$

where:

$$\begin{array}{lll} b_{12} = W_1/W_o, & b_{13} = W_2/W_o, & b_{14} = Z_o/W_o \\ b_{22} = W_2 - b_{12} W_1, & b_{23} = W_3 - b_{13} W_1, & b_{24} = Z_1 - b_{14} W_1 \\ b_{32} = W_3 - b_{12} W_2, & b_{33} = W_4 - b_{13} W_2, & b_{34} = Z_2 - b_{14} W_2 \end{array}$$

Steps 1 and 2 are repeated using equations (5) and (6), with b_{22} and b_{32} instead of W_o and W_1. The resulting equations are:

$$\begin{array}{ll} (7) & a_1 + c_{23} a_2 = c_{24} \\ (8) & c_{33} a_2 = c_{34} \end{array}$$

where:

$$c_{23} = b_{23}/b_{22} \quad , c_{24} = b_{24}/b_{22}$$

$$c_{33} = b_{33} - c_{23} b_{32} \; , \; c_{34} = b_{34} - c_{24} b_{32}$$

The backward solution is as follows:

(9)	$a_2 = c_{34}/c_{33}$	from equation (8)
(10)	$a_1 = c_{24} - c_{23} a_2$	from equation (7)
(11)	$a_o = b_{14} - b_{12} a_1 - b_{13} a_2$	from equation (4)

Figure 6 is a sample FORTRAN program for carrying out the calculations for the case: $n = 100$, $m \leq 10$. $W_0, W_1, W_2, \ldots, W_{2m}$ are stored in W(1), W(2), W(3), . . . , W(2M+1), respectively. $Z_0, Z_1, Z_2, \ldots, Z_m$ are stored in Z(1), Z(2), Z(3), . . . , Z(M+1), respectively.

IBM FORTRAN Coding Form

GX28-7327 U/M050 Printed in U.S.A.

PROGRAM: SAMPLE PROGRAM 2 — PAGE 1 OF 3

```
      REAL X(100), Y(100), W(21), Z(11), A(11), B(11,12)
    1 FORMAT (I2,I3/(4F14.7))
    2 FORMAT (5E15.6)
      READ (1,1) M,N,(X(I),Y(I),I=1,N)
      LW = 2*M+1
      LB = M+2
      LZ = M+1
      DO 5 J=2,LW
    5 W(J) = 0.0
      W(1) = N
      DO 6 J=1,LZ
    6 Z(J) = 0.0
      DO 16 I=1,N
      P = 1.0
      Z(1) = Z(1)+Y(I)
      DO 13 J=2,LZ
      P = X(I)*P
      W(J) = W(J)+P
   13 Z(J) = Z(J)+Y(I)*P
      DO 16 J=LB,LW
      P = X(I)*P
```

Figure 6 (Part 1 of 3). Sample Program 2

FORTRAN Coding Form

GX28-7327 U/M050 Printed in U.S.A.

PROGRAM SAMPLE PROGRAM 2 — PAGE 2 OF 3

```
   16 W(J) = W(J)+P
   17 DO 2Ø  I=1,LZ
      DO 2Ø  K=1,LZ
      J = K+I
   2Ø B(K,I) = W(J-1)
      DO 22  K=1,LZ
   22 B(K,LB) = Z(K)
   23 DO 31  L=1,LZ
      DIVB = B(L,L)
      DO 26  J=L,LB
   26 B(L,J) = B(L,J)/DIVB
      I1 = L+1
      IF (I1-LB) 28,33,33
   28 DO 31  I=I1,LZ
      FMULTB = B(I,L)
      DO 31  J=L,LB
   31 B(I,J) = B(I,J)-B(L,J)*FMULTB
   33 A(LZ) = B(LZ,LB)
      I = LZ
   35 SIGMA = Ø.Ø
      DO 37  J=I,LZ
```

Figure 6 (Part 2 of 3). Sample Program 2

FORTRAN Coding Form

GX28-7327 U/M050 Printed in U.S.A.

PROGRAM SAMPLE PROGRAM 2 — PAGE 3 OF 3

```
   37 SIGMA = SIGMA+B(I-1,J)*A(J)
      I = I-1
      A(I) = B(I,LB)-SIGMA
   4Ø IF (I-1) 41,41,35
   41 WRITE (3,2) (A(I),I=1,LZ)
      STOP
      END
```

Figure 6 (Part 3 of 3). Sample Program 2

The elements of the W array, except W(1), are set equal to zero. W(1) is set equal to N. For each value of I, X_i and Y_i are selected. The powers of X_i are computed and accumulated in the correct W counters. The powers of X_i are multiplied by Y_i and the products are accumulated in the correct Z counters. In order to save machine time when the object program is being run, the previously computed power of X_i is used when computing the next power of X_i. Note that the use of variables as index parameters. By the time control has passed to statement 17, the counters are set as follows:

$$W(1) = N \qquad Z(1) = \sum_{I=1}^{N} Y_I$$

$$W(2) = \sum_{I=1}^{N} X_I \qquad Z(2) = \sum_{I=1}^{N} Y_I X_I$$

$$W(3) = \sum_{I=1}^{N} X_I^2 \qquad Z(3) = \sum_{I=1}^{N} Y_I X_I^2$$

$$\vdots$$

$$Z(M+1) = \sum_{I=1}^{N} Y_I X_I^M$$

$$W(2M+1) = \sum_{I=1}^{N} X_I^{2M}$$

By the time control passes to statement 23, the values of $W_0, W_1, \ldots, W_{2m+1}$ are placed in the storage locations corresponding to columns 1 through M+1, row 1 through M+1, of the B array, and the values of $Z_0, Z_1, \ldots, Z_m$ have been stored in the locations corresponding to the column M+2 of the B array. For example, in the illustrative problem (M = 2), columns 1 through 4, rows 1 through 3, of the B array would be set to the following computed values:

$$\begin{matrix} W_0 & W_1 & W_2 & Z_0 \\ W_1 & W_2 & W_3 & Z_1 \\ W_2 & W_3 & W_4 & Z_2 \end{matrix}$$

This matrix represents equations (1), (2), and (3), the normal equations for M=2.

The forward solution, which results in equations (4), (7), and (8), is carried out by statements 23 through 31. By the time control passes to statement 33, the coefficients of the AI terms in the M+1 equations, which would be obtained in manual calculations, have replaced the contents of the locations corresponding to columns 1 through M+1, rows 1 through M+1, of the B array, and the constants on the right-hand side of the equations have replaced the contents of the locations corresponding to column M+2, rows 1 through M+1, of the B array. Columns 1 through 4, rows 1 through 3, of the B array are set to the following computed values:

$$\begin{matrix} 1 & b_{12} & b_{13} & b_{14} \\ 0 & 1 & c_{23} & c_{24} \\ 0 & 0 & c_{33} & c_{34} \end{matrix}$$

This matrix represents equations (4), (7), and (8).

The backward solution, which results in equations (9), (10), and (11), is carried out by statements 33 through 40. By the time control passes to statement 41, which prints the values of the A(I) terms, the M+1 values of the A(I) terms are stored in the M+1 locations for the A array. The A array would contain the following computed values for a_2, a_1, and a_0, respectively:

Location	**Contents**
A(3)	c_{34}/c_{33}
A(2)	$c_{24}-c_{23}a_2$
A(1)	$b_{14}-b_{12}a_1-b_{13}a_2$

The resulting values of the A(I) terms are then printed according to the FORMAT statement number 2.

Part 2. System/3 FORTRAN IV User's Guide

This section contains:

- Overview of FORTRAN processing
- Compilation
- Linkage editor processing
- Load module execution
- Job output

A FORTRAN program is processed by the FORTRAN compiler under control of the IBM System/3 System Control Program. The FORTRAN compiler is a program that translates FORTRAN statements into instructions that can be understood and executed by the System/3. The system control program is a program that controls the operation of the System/3.

For you to make the best use of System/3, you must know how to tell the System Control Program about your FORTRAN program, how to define FORTRAN files, and what kind of output to expect. This introductory section summarizes basic information you need in order to use System/3. This section briefly describes:

1. How a FORTRAN program is processed.
2. Communicating with System/3 through operation control language (OCL).
3. Program output.
4. Defining FORTRAN files and other files needed by System/3.

HOW A FORTRAN PROGRAM IS PROCESSED

Before your FORTRAN program can be executed, it must be converted into a form that can be understood by System/3. The compiler converts the program. The linkage editor combines the program with whatever other programs are required to form an executable unit. For example, if your program uses the SQRT library function, the linkage editor retrieves SQRT from the FORTRAN library of subprograms and joins it with your program.

The three steps that must be taken to convert and execute a FORTRAN program are *compilation, link-editing,* and *load module execution.* The FORTRAN source program is the input to the compilation step. The output is the group of translated statements, called an *object module,* which becomes the input to the link-edit step. (Other terms used to describe the object module are *routine* and *nonexecutable object program*; we use the term object module throughout this publication.) The output of the link-editing step is the object module combined with other modules to form a *load module,* or *object program* (we use the term load module). The load module is the program executed in the load module execution step.

Although these three steps must be taken in sequence to execute a program, it is not necessary that they occur at one time. For example, you can choose the compilation step only, with the other steps to follow at a later time. Assume that you have coded a particularly complex FORTRAN program. The first time you submit it, you might only compile it, so that you can correct any source program errors. The compiler examines each FORTRAN statement for correct syntax and issues error messages for FORTRAN language violations. After correcting these errors you could have the program compiled, link-edited, and executed at one time. Further assume that you intend to use the program many times. Once the program is successfully compiled, it would be pointless for you to compile it every time you use it. You could choose to store the compiled object module in a file of object modules, called an *object library.* Then, each time you want to execute the program, you could tell the System/3 to bypass the compile step and use the object module as its input. These are some of the alternatives you have when executing your program. You tell the system which alternatives to select through use of the operation control language statements.

Using FORTRAN consists of the general operations illustrated in Figure 7:

1. Define the job. The programmer defines the job requirements for the specific task. Usually, the following questions must be answered when the job is defined:
   a. What information is provided as input to the program?
   b. What calculations are to be performed?
   c. What output information should be generated by the program?

2. Write the source program. After the programmer defines the job, he or she develops the FORTRAN source program.

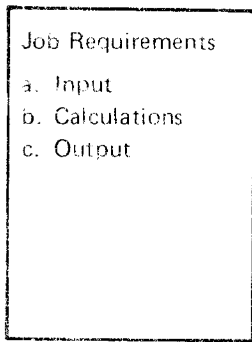
3. Record the source statements on disk or cards. After the source program is written, it is recorded on punched cards or entered into the system from the keyboard.

4. Compile the source statements. The source program, preceded by the required OCL statements, is processed by the FORTRAN IV compiler under control of the system control program. At the end of this processing (compilation), the object program is stored as an R module in the object library on disk or punched on cards. This program contains all the instructions required to perform the job.

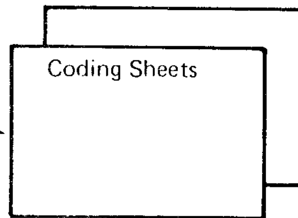
5. Link-edit the object program. The object program is processed by the Linkage Editor under control of the System Control Program. This is done to resolve all addresses and external references. At the end of this processing (link-editing), the load module is stored as an O module in the object library on disk or punched on cards. The program is now ready to be executed.

6. Execute the program. The load module is read from disk or cards; then the input and output are processed by the system under control of the FORTRAN program.

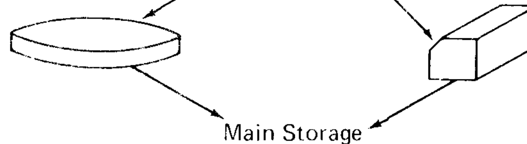
1. Define the job.



2. Write source program.

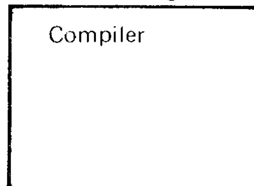


3. Record source statements on disk, diskette, or cards.



Main Storage

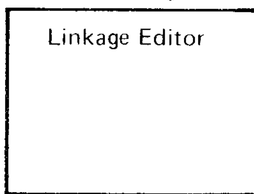
4. Compile the source statements. The resulting object program is recorded on disk, diskette, or cards.



FORTRAN IV compiler on disk.

Main Storage

5. Link-edit the object program. The resulting load module is recorded on disk, diskette, or cards.



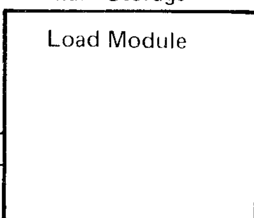
Linkage editor program on disk.

Main Storage

6. Execute the program. The load module is read from disk, diskette, or cards; then the input and output are processed by the system under control of the FORTRAN program.

Input Data

Disk
Cards
Tape
Console (Models 10 and 12)
Keyboard (Model 15)



Output Data

Printer
Disk
Cards
Tape
Console (Model 10)
CRT (Model 15)

Figure 7. Processing a FORTRAN Program

USING OPERATION CONTROL LANGUAGE (OCL)

Operation control language (OCL) is your means of communicating with IBM System/3. You must write a set of OCL statements for each program you want to run. OCL statements describe the program to the system, which reads the set of OCL statements and runs the program. When the program ends, the system reads the next set of OCL statements, runs the program it describes, and repeats the procedure until all OCL statements and programs have been run.

OCL statements that are essential in running a FORTRAN program are LOAD, FILE, and RUN.

A complete description of all OCL statements can be found in the applicable *system control programming reference manuals.* Refer to *Related Publications* for the order number.

A program submitted for compilation, link-editing, and execution might be arranged as follows:

```
// LOAD $FORT,F1
// FILE NAME-$WORK . . .
// FILE NAME-$SOURCE . . .
// RUN
*PROCESS LINK
FORTRAN source program
/*
// LOAD ABC,F1
// FILE (statement for load module, if any)
// RUN
Input data cards, if any
/*
```

Some OCL statements are used on a recurring basis, such as the LOAD $FORT and the FILE statements for $WORK and $SOURCE. To avoid recoding these statements every time they are to be used, they can be stored as sets in a source library. These sets are called *procedures.* A complete discussion of procedures can be found in the previously mentioned reference manuals.

IBM System/3 FORTRAN-Supplied Procedures

IBM supplies a number of procedures for use with System/3 FORTRAN. Figure 8 describes the procedures used for compiling, link-editing, and executing FORTRAN programs.

Note: The Model 15 also allows $WORK and $SOURCE files on 5445 or 3340 Disk Storage. The user must change the procedures if he wants to use 5445 or 3340 for $WORK and $SOURCE. See *Optimum Assignment of $WORK and $SOURCE Work Files* in Chapter 20, *System Considerations.*

Procedure Name	Function	Statements in the Procedure	Calling the Procedure

FORTRN	Compile	// LOAD $FORT,F1
		// FILE NAME-$WORK,UNIT-R1,PACK-R1R1R1,TRACKS-20,RETAIN-S
		// FILE NAME-$SOURCE,UNIT-R1,PACK-R1R1R1,TRACKS-20,RETAIN-S
		// RUN
			// CALL FORTRN,unit
			// RUN
			FORTRAN source program
			/*

FORTB	Compile (Model 6 conversational utilities)	// LOAD $FORT,R1
		// FILE NAME-$WORK,UNIT-R1,PACK-R1R1R1,TRACKS-20,RETAIN-S
		// FILE NAME-$SOURCE,UNIT-R1,PACK-R1R1R1,TRACKS-20,RETAIN-S
		// COMPILE OBJECT-R1,SOURCE-xx,UNIT-xx
		// RUN
			// CALL FORTB,unit
			// RUN

FORTG	Execute	// LOAD ##MAIN,F1
		// RUN
			// CALL FORTG,unit
			// RUN
			Data input cards
			/*

FORTL	Link-edit	// LOAD $OLINK,F1
		// FILE NAME-$WORK,UNIT-R1,PACK-R1R1R1,TRACKS-20,RETAIN-S
		// FILE NAME-$SOURCE,UNIT-R1,PACK-R1R1R1,TRACKS-20,RETAIN-S
		// RUN
			// CALL FORTL,unit
			// RUN
			Linkage editor control statement;
			Card object module (optional)
			/*

Figure 8. System/3 FORTRAN-Supplied Procedures

Libraries

Each disk can contain a *source library* and an *object library.* Although both libraries can be located anywhere on a disk, the source library always immediately precedes the object library.

The source library is an area for storing OCL procedures and FORTRAN source programs. OCL procedures or programs are added to a source library using the Library Maintenance utility program. Programs may be compiled from the source library by using the COMPILE statement. (Refer to the description of the COMPILE statement in the applicable system control programming reference manual.) (Utility programs are described in the applicable system control programming reference manual. Refer to *Related Publications* for the order number.)

The object library is an area for storing object modules and load modules. FORTRAN modules are stored by requesting the OBJECT parameter (for an object module) or the LINK parameter (for a load module) on the *PROCESS compiler option statement, used exclusively in FORTRAN processing. This statement is described in greater detail in Chapter 13, *Compilation.*

COMPILER, LINKAGE EDITOR, AND LOAD MODULE OUTPUT

In addition to object and load modules, the compiler and linkage editor produce other forms of output that help you analyze the job. The compiler informs you of the success of the compilation by issuing messages, including a *severity code* for any error encountered. The severity code indicates whether the compilation was entirely or partially successful. An unsuccessful compilation results in no object module being produced.

The compiler can also produce output in the form of a storage map that lists the names and storage locations of variables and statement numbers in the object module.

The link-editing step can produce, in addition to a load module, a core usage map that shows the location of the modules in the load module, a card deck of the object or load module, and a cataloged entry of the object or load module on disk.

Load module execution produces the program output that you request in the source program.

DEFINING FILES

Files are collections of records, such as an object module, a library of subprograms, a source program, or a data card deck.

Files Needed by the Compiler and Linkage Editor

The compiler needs the file named $WORK to store the object module if the linkage editor is called to process any of the compiler options DECK, GODECK, OBJECT, or LINK. The linkage editor needs the two files named $WORK and $SOURCE.

For information about the assignment of $WORK and $SOURCE, see *Optimum Assignment of $WORK and $SOURCE Work Files* in Chapter 20, *System Considerations.*

Files Needed by the FORTRAN Load Module

The FORTRAN load module uses files only if it includes input/output statements or calls to any I/O commercial subroutines. Commercial subroutines are described in the publication *IBM System/3 FORTRAN IV Commerical Subroutines,* SC28-6875.

Defining FORTRAN Files at Compilation Time

Device option statements are used to tell the compiler which files and devices to use. The compiler uses the information in the device option statements to allocate buffer space and data management control blocks to the files. The compiler sets up a table of logical unit numbers, which contains an entry for each device or file that is specified. Up to a maximum of 40 files can be specified at compile time.

For more information about device option statements, see *Compiler Option Statements* in Chapter 13, *Compilation.*

Defining FORTRAN Files at Execution Time

OCL FILE statements are used to define disk or tape files at execution time. As with any FILE statement, you must specify the file name, the file size, and the disk or tape unit.

FORTRAN file names are assigned in the form FTnnnnn where nnnnn is the logical unit number that is specified on the device option statement. The following shows the relationship between the FORTRAN logical unit number and file definition:

FORTRAN Logical Unit Number	**File Definition at Compilation Time:**	**File Definition at Execution Time:**
FORTRAN input/output statement	Device option statement	FILE statement, if file is on disk or tape.
Examples:		
1. Defining a card file: READ(1,100)A	// READ DEVICE-MFCU1	None
2. Defining a disk file: DEFINE FILE 15 (50,25,L,ID) WRITE(15'5,100)B	// DAD44 UNITNO-15	// FILE NAME-FT00015

Logical Unit Numbers

You refer to a file by coding its logical unit number in the FORTRAN input/output statement. For example, the following READ statement refers to a file having logical unit number 5:

```
READ (5,f)list
```

Logical unit numbers are assigned to System/3 devices used in FORTRAN processing. Figure 9 lists these assignments. If your program reads input data from a card reader in a System/3 Model 10 installation, the logical unit number that must be coded in the READ statement is 1 for the primary hopper of the MFCU device, 2 for the secondary hopper, 5 for the 5471 printer/keyboard, or 9 for the 1442 card read punch. Each logical unit number can be assigned to only one file in a program; for example, if logical unit number 1 is used to define the MFCU1, it cannot be used to define a disk file.

To make the proper link between the logical unit numbers in the FORTRAN program and the devices used by System/3, you must define your files at compilation time and, for tape and disk files, at load module execution time as well.

Device	Function	Logical Unit Number
Model 10 Installation:		
5424 MFCU primary hopper (MFCU1)	Read	1
5424 MFCU secondary hopper (MFCU2)	Read/punch/print ③	2
5203 or 1403 printer	Print	3
5471 printer/keyboard	Read	5
5471 printer/keyboard	Print	6
1442 card read punch	Read/punch ③	9
3410/3411 magnetic tape ①	Read/write sequential formatted and unformatted records	n ②
5444 disk	Read/write sequential and direct-access formatted and unformatted records	n ②
5445 disk	Read/write sequential and direct-access formatted and unformatted records	n ②
Model 6 Installation:		
5406 console keyboard	Read card images	1
5496 data recorder	Read/punch ③	2
5213 or 2222 printer	Print	3
5444 disk	Read/write sequential and direct-access formatted and unformatted records	n ②

① Model 10 supports only one file per tape volume.

② Can be any number from 1 to 32767 that is not used in the source program to define another device.

③ Cannot be specified as both an input and output device when using FORTRAN READ and WRITE statements. (Can be both an input and output device when using the FORTRAN commercial subroutines.)

Figure 9 (Part 1 of 3). Logical Unit Number Assignment

Device	Function	Logical Unit Number
Model 12 Installation:		
5424 MFCU primary hopper (MFCU1)	Read	1
5424 MFCU secondary hopper (MFCU2)	Read/punch/print (3)	2
5203 or 1403 printer	Print	3
5471 printer/keyboard	Read	5
5471 printer/keyboard	Print	6
1442 card read punch	Read/punch (3)	9
3410/3411 magnetic tape (1)	Read/write sequential formatted and unformatted records	n (2)
3340 direct access storage facility	Read/write sequential and direct-access formatted and unformatted records	n (2)

(1) Model 12 supports only one file per tape volume.

(2) Can be any number from 1 to 32767 that is not used in the source program to define another device.

(3) Cannot be specified as both an input and output device when using FORTRAN READ and WRITE statements. (Can be both an input and output device when using the FORTRAN commercial subroutines.)

Figure 9 (Part 2 of 3). Logical Unit Number Assignment

Device	Function	Logical Unit Number
Model 15 Installation:		
5424 MFCU primary hopper (MFCU1)①	Read	1
5424 MFCU secondary hopper (MFCU2)①	Read/punch/print③	2
2560 MFCM primary hopper (MFCM1)①	Read	1
2560 MFCM secondary hopper (MFCM2)①	Read/punch/print③	2
1403 printer	Print	3
3284 printer	Print	4
3277 display station	Read	5
3277 display station	Print	6
3741 data station or programmable work station⑥	Read/punch③	7
2501 card reader	Read	8
1442 card read punch	Read/punch③	9
3410/3411 magnetic tape④	Read/write sequential formatted and unformatted records	n②
5444 disk⑤	Read/write sequential and direct access formatted and unformatted records	n②
5445 disk⑤	Read/write sequential and direct access formatted and unformatted records	n②

① 5424 and 2560 are mutually exclusive.

② Can be any number from 1 to 32767 that is not used in the source program to define another device.

③ Cannot be specified as both an input and output device when using FORTRAN READ and WRITE statements. (Can be both an input and output device when using the FORTRAN commercial subroutines.)

④ Model 15 supports multifile tape volumes.

⑤ For Model 15, if system configuration has 3340 disk storage, these files will be located on the 3340.

⑥ Supported on 5704-FO2 only.

COMPILER OPTION STATEMENTS

Compiler option statements provide control information to the compiler. They follow the OCL RUN statement and are interpreted by the initial compiler phase after the OCL statements are read.

Most of the compiler option statements are coded in the same manner as OCL statements. Compiler option statements consist of:

- Device option statements
- The CORE statement
- The CATEGORY statement
- The *PROCESS statement

Device option statements are used by the compiler to define input/output devices used at execution time. To continue these statements, place a comma after each option in every card or line except the last option. Begin each new card or line with a // in positions 1 and 2. Leave one or more blanks between the // and the first option in the card or line.

Example:

```
// DAD44 UNITNO-'10,12,14',
// BLOCKSIZE-'256,512,768'
```

READ, PRINT, and PUNCH device option statements contain all the information necessary to make such devices available at execution time. Device option statements for disk and tape files (SEQ40, SEQ44, SEQ45, DAD40, DAD44, DAD45, and TAPE) require corresponding FILE statements at execution time to complete the file definition.

The CORE and CATEGORY statements cannot be continued. However, to continue the *PROCESS statement, place a comma after each option in every card or line except the last option. Begin each new card or line with a nonzero or nonblank character in position 6.

Example:

```
*PROCESS MAP,GOSTMT,DECK,
     1GODECK
```

Compiler option statements can be specified in any order except that the *PROCESS statement, if used, must be the last card before the FORTRAN source program. The *PROCESS statement can be specified for subprograms and main programs; the CATEGORY statement for subprograms only; and the other statements for main programs only.

The formats and uses of the compiler option statements are identical for all Models. The compiler option statements are described in Figure 10.

Statement	Function
Device option statements:	
// READ	Specifies the input device used at execution time.
// PRINT	Specifies the printer(s) used at execution time.
// NOPRINTER	Specifies that no printer is to be used at execution time.
// PUNCH	Specifies the card punch used at execution time.

// DAD40 (2)
// DAD44
// DAD45 (1)
	Specifies direct-access files used at execution time.

// SEQ40 (2)
// SEQ44
// SEQ45 (1)
	Specifies sequential files used at execution time.

// TAPE (1)	Specifies tape files used at execution time.
// CORE	Specifies the amount of storage to be used by the load module. The linkage editor uses this value to create the load module.
// CATEGORY	Specifies a priority permitting a subprogram to remain in main storage in an overlay environment.
*PROCESS	Specifies compiler options.

(1) Does not apply to Model 6.

(2) For Model 12 and Model 15 only.

Note: A maximum of 40 tape and disk files can be specified.

Figure 10. Compiler Option Statements

READ Device Option Statement

The format of this statement is:

// READ DEVICE-' device,device, . . . , device',
RECL-nn(1)

where

device is any of the following:

For Model10—MFCU1, MFCU2, 5471, 1442, MFCU1*.

For Model 6—5406, 5496.

For Model 12—MFCU1, MFCU2, 5471, 1442, MFCU1*.

For Model 15—MFCU1, MFCU1*, MFCU2, MFCU2*, MFCM1, MFCM1*, MFCM2, MFCM2*, 3277, 3277S, 1442, 1442*, 2501, 2501*, 3741(1), 3741*(1).

The asterisk following a device indicates one buffer is requested. The default is two buffers. You cannot specify both MFCU1, MFCU1* in one program. If this occurs by error, MFCU1*(one buffer) overrides MFCU1 (two buffers). This information applies to all other devices that allow the *indication. (See *Model 15 Double Buffering for Card Devices* in Chapter 20, *System Considerations.*)

The S following 3277 indicates that split screen support is requested. (See *Model 15 CRT/Keyboard Support* in Chapter 20.)

If only one device is selected, the enclosing apostrophes can be omitted.

RECL-nn specifies the number (nn) of bytes in a logical 3741 record. This number must be from 1 through 128. The parameter must be coded if the 3741 was specified as a device. If the parameter is coded and the 3741 is not specified, an OPTIONS ERROR will be set.

Example:

// READ DEVICE-5471

The example states that the 5471 will be used as a read device.

// READ DEVICE-'2501,3741',RECL-96

This example states that the 2501 and the 3741 will be used as read devices. Both devices will use two buffers and the logical record length for the 3741 is 96 bytes.

PRINT and NOPRINTER Device Option Statements

The format of the PRINT statement is:

// PRINT DEVICE-'device,device, . . . ,device'

where

device is any of the following:

For Model 10—5203, 5471, MFCU2, or 1403. If you do not use a PRINT or NOPRINTER statement, the 5203 printer is assumed. The MFCU2 is a card punch and is specified on the PRINT device option statement when printing is to appear on the card.

For Model 6—5213 or 2222. The default is the 5213.

For Model 12—5203, 5471, MFCU2 or 1403. If you do not use a PRINT or NOPRINTER statement, the 5203 printer is assumed. The MFCU2 is a card punch and is specified on the PRINT device option statement when printing is to appear on the card.

For Model 15—1403, 1403D, 3284, MFCU2*, MFCU2, MFCM2*, MFCM2, 3277, 3277S:

If you do not use a PRINT or NOPRINTER statement, the 1403 printer is assumed. Both the MFCU2 and MFCM2 are card punches and are specified on the PRINT device option statement when printing is to appear on the card.

The D following 1403 indicates the deferred print option. (1403D changes the FORTRAN language space before commands to a deferred space after command.)

The asterisk for MFCU2 and MFCM2 indicates one buffer for that device. (See *Model 15 Double Buffering for Card Devices* in Chapter 20, *System Considerations.*) To maintain the same performance level, two buffers should be used, because MFCU2 print uses two buffers for Model 10.

Enclosing apostrophes can be omitted if only one device is selected.

(1) For 5704-FO2 only.

FORTRAN Error Logging Device

For Model 6, the FORTRAN error logging device is the printer specified.

For Models 10 and 12, the FORTRAN error logging device is designated to the first printer specified in the printer list. The possible devices are the 5471, 5203, and the 1403.

For Model 15, the FORTRAN error logging device is designated to the first printer specified in the printer list. The possible devices are the 1403 and the 3284.

Output from execution of a STOP n statement, DUMP or PDUMP subprograms, DEBUG statement, or error traceback is directed to the FORTRAN error logging device.

Output from execution of a PAUSE statement on the Models 6 and 10 is directed to the FORTRAN error logging device. For Models 12 and 15, this output is directed to the system log device if it is available; if the system log device is not available, the output is displayed.

Example:

```
// PRINT DEVICE-'5203,MFCU2'
```

The example states that two printer devices, the 5203 and the MFCU2, are to be used.

The PRINT statement is always assumed by default; therefore, to avoid the default, you must specify the NOPRINTER statement, as follows:

```
// NOPRINTER
```

No attempt should be made to execute statements that require printer output if the NOPRINTER statement is specified.

PUNCH Device Option Statement

The format of the PUNCH statement is:

// PUNCH DEVICE-'device,device',RECL-nn(1)

where

device is any of the following:

For Model 10—MFCU2, 1442

For Model 6—5496

For Model 12—MFCU2, 1442

For Model 15—MFCU2, MFCU2*, MFCM2, MFCM2*, 1442, 1442*, 3741(1), 3741*(1).

The asterisk following a device indicates one buffer is requested. The default is two buffers. You cannot specify a device to have both one and two buffers in one program. If this occurs by error, one buffer overrides two buffers. (See *Model 15 Double Buffering for Card Devices* in Chapter 20, *System Considerations*.)

If only one device is specified, the enclosing apostrophes can be omitted.

RECL-nn specifies the number (nn) of bytes in a logical 3741 record. This number must be from 1 through 128. The parameter must be coded if the 3741 was specified as a device. If the parameter is coded and the 3741 is not specified, an OPTIONS ERROR will be set.

Example:

```
// PUNCH DEVICE-5496
```

The example states that the 5496 will be used as a punch device.

```
// PUNCH DEVICE-3741*,RECL-80
```

This example states that the 3741 will be used as a punch device. One buffer is requested and the logical record length is 80 bytes.

DAD40, DAD44, and DAD45 Device Option Statements

```
// DAD44 UNITNO-'unit,unit, . . . ,unit',
// BLOCKSIZE-'block,block, . . . ,block',
// BUFFERS-'number,number, . . . ,number',
// CLEAR-'Y,N, . . . ,Y',
// UPDATE-'Y,N, . . . N'
```

where

DAD44 indicates a direct-access file on the 5444 device

(1) For 5704-FO2 only.

UNITNO is used to specify the logical unit number (unit). This number is used in the associated I/O statements to refer this file. A corresponding FILE statement using this number in the NAME parameter must be submitted at execution time. The enclosing apostrophes can be omitted if only one number is specified.

BLOCKSIZE is used to specify the size of the block of records to be transmitted for each corresponding file. This value (block) must be 256 or a multiple of 256 (for example, 512 or 1024). The enclosing apostrophes can be omitted if only one block is specified. If this parameter is not specified, a default value is calculated using the record size from the DEFINE FILE statement. (See index entry *buffers.*) I/O time can be decreased by specifying as large a block as possible. (See Chapter 17, *Direct-Access Programming Considerations.*)

BUFFERS is used to specify the number of buffers to be used for the corresponding file. This number can be 1 or 2. If this parameter is not specified, the default value is 1. When sharing buffers only 1 buffer can be specified. I/O time can be decreased when two buffers are assigned for each file and the file is accessed sequentially. (See Chapter 17, *Direct-Access Programming Considerations.*) The enclosing apostrophes can be omitted if only one number is specified.

CLEAR (Model 15 only) is used to request (Y) or prohibit (N) clearing a new direct-access file. If this parameter is omitted, the default is (Y) for each corresponding file. I/O time is decreased if clearing is prohibited when creating files, but the user is responsible for formatting the disk space and ensuring that all records are valid. Enclosing apostrophes can be omitted if only one option is specified.

UPDATE is used to request input with update (Y) or input only (N) access to a direct-access file. If this parameter is omitted, the default is (Y) for each corresponding file. The parameter is not used when creating a new file. If two programs in two partitions or an interrupted (rolled out) program and an inquiry program access the same file, at least one of the programs must specify UPDATE-N.[1] The enclosing apostrophes can be omitted if only one option is specified.

Example:

```
// DAD44 UNITNO-'15,20',BLOCKSIZE-'256,512',
// BUFFERS-'1,2,'
```

The example states that unit numbers 15 and 20 are referred to as direct-access files. The size of the block of records for file 15 is 256 and for file 20 is 512. One buffer is allocated for file 15 and two buffers for file 20. At execution time, files named FT00015 and FT00020 must be defined on FILE statements.

The DAD45 statement is the same as the DAD44 except that it indicates a direct-access file on the 5445 device.

```
// DAD45 UNITNO-'unit,unit, . . . ,unit',
// BLOCKSIZE-'block,block, . . . ,block',
// BUFFERS-'number,number, . . . ,number',
// CLEAR-'Y,N, . . . ,Y',
// UPDATE-'Y,N, . . . N'
```

As on the DAD44 statement, the CLEAR parameter is for the Model 15 only.

The DAD40 statement is the same as the DAD44 except that it indicates a direct-access file in the main data area on the 3340 Direct Access Storage Facility. Only the Model 12 and Model 15 use the DAD40 statement.

```
// DAD40 UNITNO-'unit,unit, . . . ,unit',
// BLOCKSIZE-'block,block, . . . ,block',
// BUFFERS-'number,number, . . . ,number',
// CLEAR-'Y,N, . . . ,Y',
// UPDATE-'Y,N, . . . ,N'
```

The CLEAR parameter is only for the Model 15.

SEQ40, SEQ44, and SEQ45 Device Option Statements

```
// SEQ44 UNITNO-'unit, . . . ,unit',
// BLOCKSIZE-'block, . . . ,block'
```

where

SEQ44 indicates a sequential file on the 5444 device.

UNITNO is used to specify the logical unit number (unit). This number is used in the associated I/O statements to refer to this file. A corresponding FILE statement using this number in the NAME parameter must be submitted at execution time. The enclosing apostrophes can be omitted if only one number is specified.

[1] This restriction does not apply to 5704-FO2.

BLOCKSIZE is the size of the block containing a record to be transmitted for each corresponding file. A BLOCKSIZE must be specified for each unit given in the UNITNO parameter. The value of *block* must be 256, 128, 64, 32, or 16. The enclosing apostrophes can be omitted if only one block is specified. For best I/O performance, and most efficient use of disk space, specify a block size as the smallest submultiple of 256 that is equal to or larger than your file's record size. (See Chapter 18, *Sequential Disk and Tape Programming Considerations.)*

Example:

```
// SEQ44 UNITNO-'14,15',BLOCKSIZE-'256,256'
```

The example states that unit numbers 14 and 15 are to be referred to as sequential files on the 5444. Files named FT00014 and FT00015 must be defined on FILE statements at execution time.

This page intentionally left blank.

The SEQ45 statement is the same as the SEQ44 statement, except that it indicates a sequential file on the 5445 device.

```
// SEQ45 UNITNO-'unit, . . . ,unit',
// BLOCKSIZE-'block, . . . ,block'
```

The SEQ40 statement is the same as the SEQ44 statement except that it indicates a sequential file in the main data area on the 3340 Direct Access Storage Facility. Only the Model 12 and Model 15 use the SEQ40 statement.

```
// SEQ40 UNITNO-'unit,unit, . . . ,unit',
// BLOCKSIZE-'block,block, . . . ,block
```

TAPE Device Option Statement

```
// TAPE UNITNO-'unit, . . . ,unit',
// BLOCKSIZE-'block, . . . ,block'
```

where

UNITNO is used to specify the logical unit number (unit). This number is used in the associated I/O statements to refer to this file. A corresponding FILE statement using this number in the NAME parameter must be submitted at execution time. The enclosing apostrophes can be omitted if only one number is specified.

BLOCKSIZE is used to define the record length for each corresponding file. A BLOCKSIZE must be specified for each unit given in the UNITNO parameter. The value of *block* must be a number between 18 and 32767. The enclosing apostrophes can be omitted if only one block is specified. The System/3 BLOCKSIZE parameter must be equal to or greater than the logical record length for formatted I/O. (See Chapter 18, *Sequential Disk and Tape Programming Considerations.*)

Example:

```
// TAPE UNITNO-10,BLOCKSIZE-512
```

The example states that unit number 10 on a magnetic tape device has a record length of 512 bytes.

The file named FT00010 must be defined on a FILE (tape) statement at execution time.

Note (Model 15 only): If records are being transmitted to or from a tape drive using multifile tape volumes, the OCL FILE statement must specify the file's sequence number (SEQNUM). A further description of multifile tape volumes can be found in *IBM System/3 Model 15 System Control Programming Reference Manual,* GC21-5077.

CORE Statement

The CORE statement specifies how much main storage is required for the load module at execution time. This value is passed to the linkage editor for a main program to be link-edited immediately following compilation (the LINK compiler option was specified). If you do not include a CORE statement, the default is the actual size of the program.

The format of the statement is:

```
// CORE SIZE-annK
```

where

SIZE-annK indicates the amount of storage required:

nnK represents a number multiplied by K (1024 bytes). For example, 05K indicates 5K, or 5120 bytes.

a indicates an additional fractional K; it is given one of the values: Q (additional quarter K), H (additional half K), T (additional three-quarter K), or 0 (no additional K). For example, Q indicates an additional 1/4K or 256 bytes.

Example:

```
// CORE SIZE-H10K
```

The example states that 10K bytes, plus an additional 1/2K (512 bytes), are to be allocated.

Note: For program 5704-FO2 only, when main storage size is specified by the CORE statement, external buffer storage space is not included. External buffers, when specified in the *PROCESS statement, are allocated outside of the specified CORE space but within the partition. External buffer storage requirements are listed in the overlay linkage editor map for main programs.

CATEGORY Statement

The CATEGORY statement specifies a category value for a FORTRAN subprogram. The category value is used by the linkage editor to determine the priority of the subprogram for remaining in main storage in an overlay environment.

The value is specified as a number between 0 and 128, the lower the number the greater the priority.

A category value of 0 means that the subprogram will never be overlaid. Category values 1-7 are used by system modules and modules in the FORTRAN library. If no category is specified for a program, its value is assumed to be 20.

The format of the statement is:

```
// CATEGORY VALUE-m
```

where
VALUE-m indicates the category value, 0 through 128.

Example:

```
// CATEGORY VALUE-15
```

The example states that the FORTRAN subprogram has a priority of 15; only programs having a value 0 through 14 would have a higher priority.

*PROCESS Statement

The *PROCESS statement allows the user to specify which actions the compiler is to perform. For example, by selecting the SOURCE option, you request the compiler to print a listing of the source program; by selecting the DECK option, you request the compiler to call the linkage editor to punch a card deck of the object module.

The *PROCESS statement appears in the input stream as the last compiler option statement before the FORTRAN source program. Keep in mind that the *PROCESS statement is more closely related to compiler processing than the other statements discussed here. The other statements specify information regarding operations to be performed at later stages of processing, but the *PROCESS statement specifies options directly related to compiler operations.

The statement must start in column one; column 72 is the last usable column for the statement. Continuation lines can be used. (See *Continuing FORTRAN Statements* in Chapter 1.)

The format of the statement is:

```
*PROCESS option,option, . . . ,option
```

where
option is one of the *PROCESS options summarized in Figure 11. In that figure, defaults are underlined and need not be specified if you want the default values to apply. Options can be specified in any order. Blanks can be inserted between options but not within an option.

Option	Meaning	Remarks
<u>SOURCE</u> NOSOURCE	Whether the source program input is to be printed.	
MAP <u>NOMAP</u>	Whether the relative addresses assigned by the compiler to source program data items are to be printed, and whether the addresses assigned by the linkage editor to the load module are to be printed. A relative address is the location of the data item calculated from the beginning of the program code. The compiler replaces each data item name with its relative address. (At execution time, branches are made to relative addresses rather than to names.)	MAP causes the linkage editor to perform extra time-consuming operations necessary to generate a storage map. Therefore, if the linkage editor is to be executed, MAP should be specified only if processing time is not critical.
DECK <u>NODECK</u>	Whether the object module is to be punched by the linkage editor.	If DECK is specified, the $SOURCE and $WORK files are required.
GOSTMT <u>NOGOSTMT</u>	Whether diagnostic traceback messages issued at execution time are to contain the internal statement numbers assigned by the compiler. An internal statement number (ISN) is assigned to each source program statement for identification. ISNs are assigned in ascending order beginning with 1 for the first statement, 2 for the second statement, etc.	
<u>EBCDIC</u> BCD	Whether the source program was coded in EBCDIC (Extended Binary Coded Decimal Interchange Code) or in BCD (Binary Coded Decimal).	If the input deck contains both BCD and EBCDIC, specify BCD.
<u>NOGODECK</u> GODECK	Whether the load module is to be punched.	Can be used with main programs only. If GODECK is specified, the $SOURCE and $WORK files are required.
<u>NOSHRBUFF</u> SHRBUFF	Whether a buffer is to be shared between two or more direct-access 5444, 5445, or 3340 files. NOSHRBUFF must be specified if: two buffers are specified for any direct-access file in a program, or EXTBUF is specified for DAD files (Program Number 5704-FO2 only). *Note*: If you specify one or both of the above with the SHRBUFF option, the program defaults to the NOSHRBUFF option.	Can be used with main programs only.

Figure 11 (Part 1 of 3). *PROCESS Statement Options

Option	Meaning	Remarks
NOOBJECT OBJECT (T,LIB (unit)) P R	Whether an object module is to be stored in the object library maintained by the system. T means the object module is temporary. P means that it is permanent. R means that it is to replace the existing module by the same name. Unit indicates where the object library resides, specified as one of the codes: R1–removable disk, drive 1 F1–fixed disk, drive 1 R2–removable disk, drive 2 F2–fixed disk, drive 2 If LIB(unit) is not specified, the library resides on the program pack, that is, the pack indicated in the unit parameter of the LOAD statement calling the compiler. The default value for the OBJECT option is NOOBJECT for a main program, and OBJECT (program pack) for a subprogram.	If OBJECT is specified, the $SOURCE and $WORK files are required.
NOLINK LINK(T,LIB(unit)) P R	Whether a load module is to be stored into the object library maintained by the system. T means that the load module is temporary. P means that it is permanent. R means that it is to replace the existing module by the same name. Unit indicates where the load module is to be stored, using the same codes as specified in the OBJECT option. If LIB(unit) is not specified, the load module resides on the program pack. The default value is LINK(program pack) for a main program (cannot be specified for a subprogram).	May be used with main programs only. If LINK is specified, the $SOURCE and $WORK files are required.

Figure 11 (Part 2 of 3). *PROCESS Statement Options

Option	Meaning	Remarks
UPACK-unit (Models 12 and 15 only)	The pack where user routines to be link-edited can be found. The overlay linkage editor searches this pack first when resolving EXTRNs to modules whose names do not begin with $. If the EXTRN name is not found on this pack, the program pack is searched. Unit indicates the pack to be searched first. The following codes apply: R1—removable disk, drive 1 F1—fixed disk, drive 1 R2—removable disk, drive 2 F2—fixed disk, drive 2 If the option is not coded, the program pack will be the only pack searched.	Can be used with main programs only.
EXTBUF NOEXTBUF (Program Number 5704-FO2 only)	Whether DAD files only (DAD40, DAD44, DAD45) are to have buffer space allocated external to the program space. If EXTBUF is specified with SHRBUFF, the program defaults to NOSHRBUFF.	Can be used with main programs only.

Figure 11 (Part 3 of 3). *PROCESS Statement Options

Examples:

```
*PROCESS MAP,DECK
```

In the preceding example, MAP indicates that the compiler is to print a listing, called a map, of relative addresses (and, if the link-editing step is executed, a load module storage map), and DECK indicates that a card deck of the object module is to be punched. Default values, such as SOURCE and EBCDIC, apply for *PROCESS options that are not specified.

```
*PROCESS SOURCE,GOSTMT,LINK(T,LIB(R1))
```

In the preceding example, SOURCE indicates that the compiler is to print a listing of the source program; note that SOURCE, although a default value, can be explicitly requested. GOSTMT indicates that any diagnostic trace-back messages issued at execution time are to be printed with the ISNs of related FORTRAN statements. LINK (T,LIB(R1)) indicates that the load module produced by the linkage editor is to be stored as a temporary library entry located on the R1 disk. Default values apply for *PROCESS options that are not specified.

BATCHED COMPILATION

To compile a program consisting of many program units, to be immediately followed by link-editing and load module execution, the main program must be the last program compiled. For example, assume a program consists of three program units—one main program and two subprograms. They should be submitted for compilation such that the main program follows the sub-programs in the job stream. An example showing the placement of OCL and compiler option statements for a batched compilation is illustrated in Example 4 in the following discussion.

COMBINING OCL AND COMPILER OPTION STATEMENTS

The following examples show how compiler option statements can be combined with OCL statements.

Example 1:

Assume you want to compile your program, with no link-editing step to follow.

OCL statements you need are DATE (optional), LOAD, RUN, and /*. The only compiler option statement you need is the *PROCESS statement to specify NOLINK. Statements should be submitted in this order:

```
// DATE 090174
// LOAD $FORT,R1
// RUN
*PROCESS NOLINK
FORTRAN source program
/*
```

Example 2:

Assume you want to compile, link-edit, and execute a program that reads data from a card reader and prints the results on a printer. Further assume you want all compiler default options to apply.

Necessary OCL statements are DATE, LOAD, FILE $WORK, FILE $SOURCE, RUN, and /*. The only compiler option statement you need is a READ statement to set up the input buffers for the reader used at execution time; assume that the reader at your location is the 5496 (for a Model 6 installation). You need neither a PRINT statement (because PRINT is assumed), nor a *PROCESS statement (because all *PROCESS defaults are wanted). Statements should be submitted in this order:

```
// DATE 090174
// LOAD $FORT,R1
// FILE NAME-$WORK,UNIT-R1,PACK-111111,
   TRACKS-20,RETAIN-S
// FILE NAME-$SOURCE,UNIT-R1,PACK-111111,
   TRACKS-20,RETAIN-S
// RUN
// READ DEVICE-5496
FORTRAN source program
/*
// LOAD ##MAIN,R1
// RUN
Program data
/*
```

In this example, the READ statement defines the reader to use at execution time. The compilation and link-editing operation is defined as being from the first LOAD statement through the first /* statement. The load module execution operation is defined as being from the second LOAD statement (LOAD ##MAIN) through the second /* statement.

Example 3:

Assume you wish to compile, link-edit, and execute a program that requires a card reader, a printer, and two sequential files with logical unit numbers 15 and 16. You are going to use the procedure, FORTRN, containing the appropriate LOAD and FILE statements, to call the compiler; and you want to produce all output types that the compiler is capable of generating.

OCL statements you need are DATE, CALL, RUN, and /*. Compiler option statements you need are READ (assume a Model 10 installation having the MFCU1 device), SEQ44 to define two file numbers (assume the 5444 device), and *PROCESS specifying MAP, GOSTMT, DECK, and GODECK (output-producing *PROCESSING options not assumed by default). Statements should be submitted in this order:

```
// DATE 090174
// CALL FORTRN,R2
// RUN
// READ DEVICE-MFCU1
// SEQ44 UNITNO-'15,16',BLOCKSIZE-'256,256'
*PROCESS MAP,GOSTMT,DECK,GODECK
FORTRAN source program
/*
```

Statements required by the load module execution step are not illustrated here. The SEQ44 statement requires two corresponding FILE statements at execution time. Required FILE statements are described in Chapter 15 *Load Module Execution*.

Example 4:

Assume you want to compile, link-edit, and execute a program consisting of three program units--MAIN, SUB1, and SUB2. A program consisting of more than one program unit is submitted as a batch compilation, with the subprograms compiled first and the main program compiled last.

Any compiler option statements specified, except *PROCESS and CATEGORY, must be submitted with the main program. (*PROCESS can be submitted for main programs and subprograms, CATEGORY for subprograms only.) Thus, assuming that a card reader is to read data to the program, the READ device option statement must be submitted among the compilation cards for the main program. Further assume that the procedure, FORTRN, containing the appropriate LOAD and FILE statements, is used, and that both subprograms SUB1 and SUB2 are given a category value of 10.

Statements should be submitted in this order:

```
// DATE 090174
// CALL FORTRN,R2
// RUN
// CATEGORY VALUE-10
   SUBROUTINE SUB1
/*
// CALL FORTRN,R2
// RUN
// CATEGORY VALUE-10
   SUBROUTINE SUB2
/*
// CALL FORTRN,R2
// RUN
// READ DEVICE-5471
   PROGRAM MAIN
/*
// LOAD MAIN,R2
// RUN
Data input to load module
/*
```

The first CALL FORTRN statement calls the FORTRAN compiler to compile SUB1. The /* statement indicates the end of the first compilation. The next CALL FORTRN statement calls the compiler to compile SUB2. The third CALL FORTRN statement calls the compiler to compile MAIN. The LOAD MAIN statement is the first statement of the load module execution step.

The output of the FORTRAN compiler, the object module, is not ready to be executed as a program.

During a process called link-editing, the linkage editor converts the object module into a load module that is ready for execution.

COMPILER INPUT TO THE LINKAGE EDITOR

The FORTRAN compiler calls the linkage editor automatically if any of the *PROCESS options DECK, OBJECT, LINK, or GODECK are specified. If these options are specified, the compiler must also pass to the linkage editor the object module defined in the $WORK FILE statement and a utility work file defined in the $SOURCE FILE statement.

The OBJECT option causes the linkage editor to store the object module into a library. DECK causes the linkage editor to punch a card deck of the object module. If these are the only options passed to the linkage editor, it returns control to the system after performing these functions.

LINK causes the linkage editor to perform the link-editing function. GODECK can be specified to have a card deck of the load module punched.

OCL STATEMENTS NEEDED FOR THE LINKAGE EDITOR

If the linkage editor is called by the compiler, you need supply only the FILE statements named $WORK and $SOURCE as part of the compile step OCL statements. If the linkage editor is not called automatically, you can call it directly by specifying its name, $OLINK, in a LOAD statement.

The following examples show only the order in which OCL statements should be submitted.

Example 1:

To call the linkage editor without using a procedure, you submit the following statements:

```
// LOAD $OLINK,R1
// FILE NAME-$WORK
// FILE NAME-$SOURCE
// RUN
Linkage editor input
/*
```

Example 2:

To call the linkage editor using the procedure FORTL, you submit the following statements:

```
// CALL FORTL,R1
// RUN
Linkage editor input
/*
```

LINKAGE EDITOR CONTROL STATEMENTS

The formats and uses of the linkage editor control statements are identical for all models.

Ordinarily, the compiler passes to the linkage editor all information that is necessary to link-edit an object module. However there are instances when you want to call the linkage editor facility directly, such as to group a number of subprograms together in storage. In such instances, you can load the linkage editor directly and use linkage editor control statements to specify the options you want.

Linkage editor control statements provide information about the object modules to be link-edited. Like compiler option statements, they are used exclusively in one step and are coded in the same manner as OCL statements.

Linkage editor control statements are fully described in *IBM System/3 Overlay Linkage Editor Reference Manual,* GC21-7561.

The following example shows how linkage editor control statements can be combined with OCL statements.

Example 1:

Example 1 of *Combining OCL and Compiler Option Statements* in Chapter 13, *Compilation* showed how to compile and store a program into an object library. Assume you want to retrieve that program, link-edit it, and store the load module under the name FIRST, into any library.

OCL statements you need are DATE, LOAD, FILE $WORK and FILE $SOURCE, and RUN. Linkage editor control statements you need are PHASE, to assign a name to the load module, INCLUDE, to indicate the object module to be included in the link edit, and END. Statements should be submitted in this order:

```
// DATE 100174
// LOAD $OLINK,R1
// FILE NAME-$WORK,UNIT-R1,PACK-111111,
   TRACKS-20,RETAIN-S
// FILE NAME-$SOURCE,UNIT-R1,PACK-111111,
   TRACKS-20,RETAIN-S
// RUN
// PHASE NAME-FIRST
// INCLUDE NAME-PROG1,UNIT-F1
// CATEGORY NAME-$$BTAM,VALUE-2
// END
```

In this example, the LOAD statement loads the linkage editor from its location on unit R1. The two FILE statements are required to provide work space for the linkage editor. The PHASE statement tells the linkage editor to store the load module under the name FIRST; by default the load module is stored as a temporary unit on R1 (the program pack). The INCLUDE statement specifies that the object module to be link-edited is named PROG1 and is on unit F1. The CATEGORY statement changes $$BTAM from category 3 to category 2.

LINKAGE EDITOR OVERLAY FEATURE

If main storage is too small to accommodate an entire program, the linkage editor will force the program into overlays. In addition to the automatic overlay that can be performed by the linkage editor, the programmer can explicitly request overlays through the use of linkage editor control statements.

The linkage editor attempts to fit all modules of an object program into the specified storage size without overlays. If this cannot be accomplished, the linkage editor assigns some modules to overlay segments. Modules are placed in overlays according to their size, category, and relationship to other modules in the program.

The linkage editor overlay feature is fully described in the *IBM System/3 Overlay Linkage Editor Reference Manual,* GC21-7561.

Chapter 15. Load Module Execution

The load module execution step runs a FORTRAN program that has been compiled and merged with other object modules into a load module.

Load module input consists of OCL statements defining the step and any program data to be processed by the load module. Output consists of program output and execution messages.

This section describes OCL statements needed for load module execution, program data, and combining OCL statements and data.

OCL STATEMENTS NEEDED FOR LOAD MODULE EXECUTION

The load module is called by an OCL LOAD statement specifying the load module's name. The name is either the name you assign to the source program in the PROGRAM statement, or the name ##MAIN given to an unnamed program by the compiler.

OCL statements needed for load module execution are summarized in Figure 12.

Statement	Function	When Required	Placement in Job stream	Can be Stored as Part of a Procedure

// LOAD ##MAIN
or
// LOAD name
	To call the load module	Always	First card	Yes

// FILE NAME-FTnnnnn	To define a sequential or direct-access file	When the load module processes records from a disk or tape file	After the LOAD statement	Yes
// RUN	To define the end of OCL statements in the job stream	Always	After OCL statements; before program input cards	No
/*	To define the end of input cards in the job stream	When input cards are submitted through the input stream	After the last input card	No

Figure 12. OCL Statements Needed for Load Module Execution

Examples for their use are shown below. These examples show only the order in which statements should be submitted; for examples of fully coded OCL statements see *Combining Load Module OCL Statements with Compile Step Statements* in this chapter.

Example 1:

Assume a procedure of OCL statements, FORTG, containing a LOAD statement having the name ##MAIN. To call the procedure to execute an unnamed program, submit the following cards:

```
// CALL FORTG,F1
// RUN
Program data if any
/*
```

Example 2:

To call the same procedure, only this time for a program having the name MYPROG, submit the following cards:

```
// CALL FORTG,F1
// LOAD MYPROG
// RUN
Program data if any
/*
```

The LOAD statement specifying the name MYPROG replaces the LOAD statement in the procedure.

Example 3:

To load a load module directly, without using a procedure, and assuming the use of a printer and a disk file named FT00009, submit the following cards:

```
// LOAD MYPROG
// FILE NAME-FT00009
// RUN
```

PROGRAM DATA

Program data—information to be processed by the load module—can be submitted in cards, as records on disk or tape, or from the keyboard. Disk or tape records are defined as files on FILE statements. Card records are submitted as a card deck between the OCL RUN and /* statements. The following example shows how you can define input both on cards and on disk.

Assume your program reads input from cards and from two files numbered 15 and 16, and you use the procedure FORTG to call the load module. Statements should be submitted in this order:

```
// CALL FORTG,F1
// FILE NAME-FT00015
// FILE NAME-FT00016
// RUN
Card program data
/*
```

COMBINING LOAD MODULE OCL STATEMENTS WITH COMPILE STEP STATEMENTS

The load module statements specified in the preceding examples can be combined with compile step statements to form a complete job. In Chapter 13 *Compilation*, under the heading *Combining OCL and Compiler Option Statements*, Example 3 shows compile step statements that can be merged with load module statements. Simply place the load module statements after the compile statements to form a job in the following order:

```
// DATE 090174
// CALL FORTRN,R2
// RUN
// READ DEVICE-MFCU1
// SEQ44 UNITNO-'15,16',BLOCKSIZE-'256,256'
*PROCESS MAP,GOSTMT,DECK,GODECK
FORTRAN source program
/*
// CALL FORTG,R2
// FILE NAME-FT00015,UNIT-R2,PACK-12345,
   TRACKS-5
// FILE NAME-FT00016,UNIT-R2,PACK-12345,
   TRACKS-5
// RUN
Card program data
/*
```

In this example, DATE assigns a system date to the job. The first CALL invokes the compile procedure FORTRN. READ sets up the MFCU1 as the execution-time card reader; this statement permits card input to be read during the execution step. SEQ44 sets up a system control block and buffer space for each of the execution time files named FT00015 and FT00016. *PROCESS specifies compiler options (a storage map to be printed, execution time error messages to be generated with internal statement numbers, a card deck of the object module to be punched, and a card deck of the load module to be punched).

The second CALL invokes the load module procedure. The two FILE statements define the files whose system control blocks were defined by SEQ44. Card program data is read from the device defined by the compile step READ statement.

This chapter describes job output for the FORTRAN program depicted in Figures 13 and 14. Figure 13 shows a program as it was coded. This program adds a list of 4-digit numbers, computes the average, stores the results into a direct-access file, and prints the results. Figure 14 shows the program as it was keypunched. Keypunch errors were introduced to provide examples of system diagnostic action.

Figure 15 illustrates the OCL for the job and an input statement containing a group of numbers for the source program to add and average. The CALL statement calls the compiler. The device option statements READ, PRINT, and DAD44 define devices to be used at execution time. The *PROCESS statement lists a typical selection of compiler options that produce output. If the *PROCESS statement were missing, default options SOURCE and LINK would still be in effect (SOURCE because it is the default for all programs, LINK because it is the default for main programs). The FORTRAN source program is placed between the *PROCESS statement and the /* statement which is the last of the compile step statements. The LINK option causes the linkage editor to be called automatically; no OCL statements need by supplied for the link edit step. The LOAD statement loads the load module. The FILE statement defines the direct-access file used in the program. Input data consists of 18 numbers, consecutively numbered 1 through 18, which the FORTRAN program is to read four characters at a time into the array NUMBS.

FORTRAN Coding Form

```
      PROGRAM AVG.
C     THIS PROGRAM READS AND AVERAGES NUMBERS.  THE NUMBERS ARE READ
C     IN GROUPS OF EIGHTEEN FROM THE MFCU..
      DEFINE FILE 14 ( 1, 16, E, K)
C     IT ALSO WRITES THE AVERAGE ON UNIT 14 AND READS IT BACK FOR
C     PRINTING.
      DIMENSION NUMBS ( 1ØØØ )
C     NUMBS IS THE ARRAY INTO WHICH THE NUMBERS ARE READ.  1ØØØ NUMBERS
C     IS THE MAXIMUM NUMBER OF NUMBERS THAT CAN BE AVERAGED.
      M = 1
      N = 18
   1Ø READ ( 1, 1ØØØ, END = 2Ø ) ( NUMBS(I), I = M, N )
      M = M + 18
      N = N + 18
      GO TO 1Ø
C     READING CONTINUES UNTIL END OF FILE.  M IS ONE GREATER THAN THE
C     NUMBER OF NUMBERS READ WHEN CONTROL IS TRANSFERRED TO 2Ø.
   2Ø M = M - 1
      SUM = Ø.
      DO 3Ø I = 1, M
      SUM = SUM + NUMBS(I)
   3Ø CONTINUE
      AVG = SUM / M
      WRITE ( 3, 2ØØØ ) SUM, AVG
      WRITE ( 14'1, 3ØØØ ) SUM, AVG
   4Ø READ ( 14'1, 3ØØØ ) SUM, AVG
      WRITE (3, 2ØØØ) SUM, AVG
      STOP
 1ØØØ FORMAT ( 18 I4 )
 2ØØØ FORMAT ( ' SUM OF NUMBERS = ',F8.3,' AVERAGE = ',F8.3 )
 3ØØØ FORMAT ( 2 F8.3 )
      END
```

Figure 13. Sample FORTRAN Program as Coded

```
      PROGRAM AVG
C     THIS PROGRAM READS AND AVERAGES NUMBERS. THE NUMBERS ARE READ
C     IN GROUPS OF EIGHTEEN FROM THE MFCU1.
      DEFINE FILE 14 ( 1, 16, E, K )
C     IT ALSO WRITES THE AVERAGE ON UNIT 14 AND READS IT BACK FOR
C     PRINTING.
      DIMENSION NUMBS ( 1000 )
C     NUMBS IS THE ARRAY INTO WHICH THE NUMBERS ARE READ. 1000 NUMBERS
C     IS THE MAXIMUM NUMBER OF NUMBERS THAT CAN BE AVERAGED.
      M = 1
      N = 18
   10 READ ( 1, 1000, END = 20  ( NUMBS(I), I = M, N )
      M = M + 18
      N = N + 18
      GO TO 10
C     READING CONTINUES UNTIL END OF FILE.  M IS ONE GREATER THAN THE
C     NUMBER OF NUMBERS READ WHEN CONTROL IS TRANSFERRED TO 20.
   20 M = M - 1
      SUM = 0.
      DO 30 I = 1, M
          SUM = SUM + NUMBS(H)
   30 CONTINUE
      AVG = SUM / M
      WRITE ( 3, 2000 ) SUM, AVG
      WRITE ( 14'1, 3000 ) SUM, AVG
   40 READ  ( 14'1, 3000 ) SUM, AVG
      WRITE (3,2000)SUM,AVG
      STOP
1000 FORMAT ( 18 I4 )
2000 FORMAT ( ' SUM OF NUMBERS = ',F8.3,' AVERAGE = ',F8.3 )
3000 FORMAT ( 2 F8.3 )
      END
```

Figure 14. Sample FORTRAN Program as Keypunched

```
// CALL  FORTRN,R2
// RUN
// READ  DEVICE-MFCU1
// PRINT  DEVICE-5203
// DAD44  UNITNO-14
*PROCESS  SOURCE,LINK(T,LIB(R2)),NOSHRBUFF
FORTRAN source program
/*
// LOAD AVG,R2
// FILE UNIT-F2,PACK-544400,TRACKS-20,RETAIN-S,NAME-FT00014,
// LABEL-MYFILE
// RUN
000100020003000400050006000700080009001000110012001300140015001600170018
/*
```

Figure 15. OCL Example

Each compilation produces the following output:

- An object module if no severe errors were encountered (as described in *Diagnostic Messages*), unless the *PROCESS option NOLINK is specified.

- A compiler output listing that contains informative and diagnostic messages.

OBJECT MODULE

The compiler produces an object module in the form of a series of records each 64 bytes long. There are three types of records in an object module:

- ESL (external symbol list record)

- RLD (text-relocation directory record)

- End Record

ESL Records

An ESL record defines the object module's reference name and external references to other modules that are to be link-edited with this object module. The ESL record format is illustrated in Figure 16.

An ESL record is identified by the character S in the record's first byte. An ESL record can contain up to five ESL entries, each entry being 12 bytes long. There are seven types of ESL entries:

Module Name: This entry contains the symbolic name of the module, the start address, the module length (in hexadecimal), and its category value.

Entry Point: This entry specifies this module's entry point name and its address.

EXTRN Reference: This entry specifies an external reference, that is, a reference to a module name in another module. During the link-edit step the linkage editor must resolve this reference (that is, locate the referenced module), by searching first the job stream, then the $WORK file, and finally the object library until it locates the referenced module.

Weak EXTRN Reference: This entry specifies an external reference also, but during link edit the linkage editor does not search the object library. If it cannot resolve the corresponding reference name by searching the job stream and $WORK, the linkage editor ignores the entry.

GLOBAL Entry: This entry specifies a space allocation for a global area, an area to be used by more than one load module. This area is allocated at the start of the program level and its size is the size of the largest global area encountered.

COMMON Entry: This entry specifies a space allocation for a common area, an area to be used by one load module only. This area is allocated immediately following any global area or at the load point if no global area is allocated.

EXTBUF Entry: This entry specifies the space required for external DAD buffers. The area is allocated during execution of the program and is located after the last program byte used (between the logical end of the program and the end of the partition).

RLD Records

An RLD record contains the object code and any information needed to make the text relocatable. The RLD record format is illustrated in Figure 16.

An RLD record is identified by the character T in the record's first byte.

End Record

The End record defines the end of the object module and contains the module's start address. It is identified by the character E in the record's first byte. It is illustrated in Figure 16.

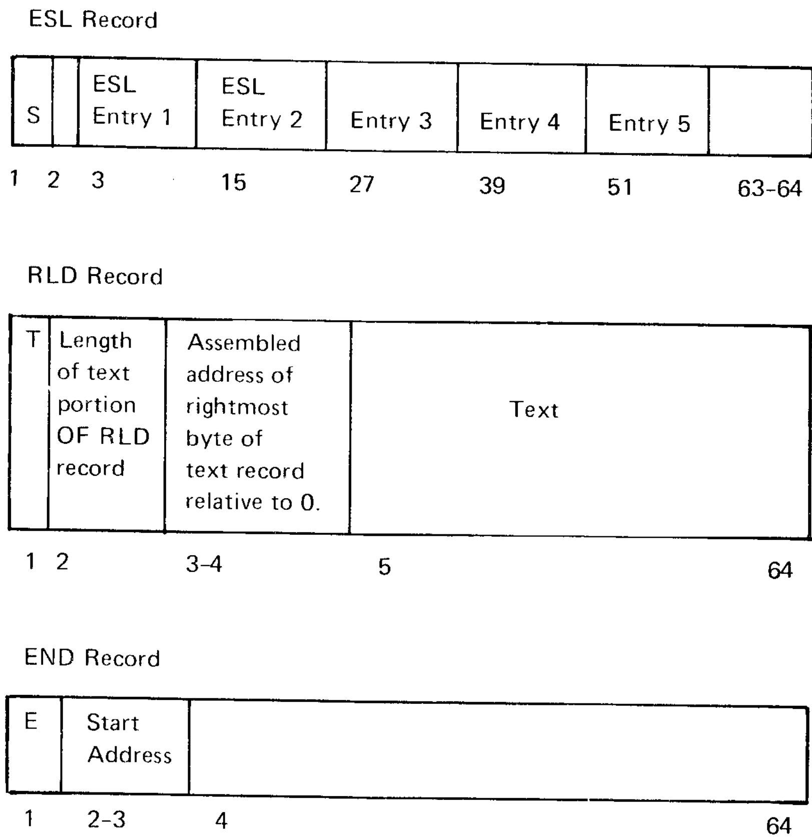


Figure 16. Object Module Record Formats

COMPILER OUTPUT LISTING

Compilation of the program shown in Figure 14 results in the output listing illustrated in Figure 17, which includes the compilation errors encountered. Figure 18 illustrates the listing for the same program after compilation errors were corrected.

Messages

Informative messages let you know the status of the compilation giving such information as the date the job was run and the compiler's version and modification level. Diagnostic messages let you know of any errors encountered during compilation.

Informative Messages

Informative messages in Figure 17 are labeled **A** and **E** the beginning and the end of the compiler output listing. Label **A** shows the first line of the listing, and a list of compiler option statements specified for the program.

Label **E** points to the end of the listing, and lists the number of compilation errors encountered, the highest severity code encountered (see *Diagnostic Messages*, following, for a discussion of severity codes), and a list of statement allocations that indicate storage addresses assigned to statement numbers used in the program. Storage address, shown in hexadecimal notation, are relative to the beginning of the program. Because the compiler moves the internal code of FORMAT statements to the beginning of the program, statement numbers of FORMAT statements, such as numbers 3000, 2000, and 1000 in this program, are allocated lower storage addresses than numbers for executable statements, such as numbers 20, 30 and 40 in this program.

Diagnostic Messages

Compiler diagnostic messages in Figure 17 are labeled **D** .
Diagnostic messages appear on an output listing under the headings:

STATEMENT NUMBER	ISN	ERROR NUMBER	SEVERITY	EXPLANATION

where

STATEMENT NUMBER is the statement number assigned to a FORTRAN source statement by the programmer. Either the statement number or the internal statement number (ISN) is printed on the diagnostic line.

ISN is the internal statement number assigned to the FORTRAN statement by the compiler. ISNs are assigned in ascending order to the statements in the source module. (ISNs appear on the extreme left side of the source listing, labeled **B**, in Figure 17.)

ERROR NUMBER is a two-digit number assigned to a particular error condition by the compiler. Each error condition is assigned a number. Error conditions, and corresponding explanations, are listed in *Appendix B. Compilation Messages.*

SEVERITY indicates the severity code assigned to the error by the compiler. Each error is assigned one of two severity codes:

4 Indicates a *possible* error. Compilation continues and the link-editing function is performed if called.

8 Indicates an error. Compilation continues but the system terminates processing after compilation. If any of the options LINK, OBJECT, DECK, or GODECK were specified, they are ignored.

EXPLANATION is a brief summary of the error condition. A full explanation of each error condition is presented in *Appendix B. Compilation Messages.*

Figure 17 illustrates three diagnostic errors, one with severity code 4, the others with severity code 8. The first error illustrated, for source statement 40, indicates that the source statement is nowhere referred to and thus does not need a statement number. Source statement 40 is:

```
40 READ(14'1,3000)SUM,AVG
```

Because this error will not affect program execution, it has a severity code of 4.

The next error illustrated, for ISN 006, indicates a syntax error. ISN 6 is the following statement:

```
10 READ(1,1000,END=20(NUMBS(I),I=M,N)
```

By scanning the statement we see that it contains three left parentheses and only two right parentheses; a right parenthesis belongs after END=20. Because this error will affect program execution, it has a severity code of 8, preventing load module execution.

The next error, for ISN 013, indicates an error in a subscript expression. ISN 13 contains only one subscript expression, NUMBS(H), where H is interpreted to be a real variable. This expression should be corrected to NUMBS(I).

Figure 18 is a listing of compiler output after these corrections were made. For purposes of easier comparison between Figures 18 and 17, areas of Figure 18 pertinent to the corrections are highlighted.

OUTPUT FROM *PROCESS OPTIONS

*PROCESS options that produce some form of output are SOURCE, MAP, DECK, GODECK, OBJECT, and LINK. If default options are in effect, SOURCE produces a listing of the source module; OBJECT produces an object module for a subprogram; LINK produces a load module for a main program; and MAP, GODECK, and DECK are suppressed. You may suppress SOURCE by explicitly specifying NOSOURCE. You can produce a storage map by specifying MAP, a card deck of the object module by specifying DECK, and a card deck of the load module by specifying GODECK.

```
     FORTRAN IV VERnn/MODnn                                                                          nn/nn/nn PAGE 001
A   // READ   DEVICE-MFCU1
    // PRINT  DEVICE-5203
    // DAD44  UNITNO-14
    *PROCESS  MAP,LINK(T,LIB(R2)),NOSHRBUFF
      1              PROGRAM AVG
             C       THIS PROGRAM READS AND AVERAGES NUMBERS.  THE NUMBERS ARE READ
             C       IN GROUPS OF EIGHTEEN FROM THE MFCU1.
      2              DEFINE FILE 14 ( 1, 16, E, K )
             C       IT ALSO WRITES THE AVERAGE ON UNIT 14 AND READS IT BACK FOR
             C       PRINTING.
      3              DIMENSION NUMBS ( 1000 )
             C       NUMBS IS THE ARRAY INTO WHICH THE NUMBERS ARE READ.  1000 NUMBERS
             C       IS THE MAXIMUM NUMBER OF NUMBERS THAT CAN BE AVERAGED.
      4              M = 1
      5              N = 18
      6           10 READ ( 1, 1000, END = 20    ( NUMBS(I), I = M, N )
      7              M = M + 18
      8              N = N + 18
      9              GO TO 10
B            C       READING CONTINUES UNTIL END OF FILE.  M IS ONE GREATER THAN THE
             C       NUMBER OF NUMBERS READ WHEN CONTROL IS TRANSFERRED TO 20.
     10           20 M = M - 1
     11              SUM = 0.
     12              DO 30 I = 1, M
     13                  SUM = SUM + NUMBS(H)
     14           30 CONTINUE
     15              AVG = SUM / M
     16              WRITE ( 3, 2000 ) SUM, AVG
     17              WRITE ( 14'1, 3000 ) SUM, AVG
     18           40 READ  ( 14'1, 3000 ) SUM, AVG
     19              WRITE (3,2000)SUM,AVG
     20              STOP
     21         1000 FORMAT ( 18 I4 )
     22         2000 FORMAT ( ' SUM OF NUMBERS = ',F8.3,' AVERAGE = ',F8.3 )
     23         3000 FORMAT ( 2 F8.3 )
     24              END

C   NAME     AT HEX1 DEC1  HEX2 DEC2  NAME    AT HEX1 DEC1   HEX2 DEC2  NAME     AT HEX1 DEC1   HEX2 DEC2
    NUMBS    I   0360 00877 130C 04876 K       I   1300 04877            M        I   1311 04881
    N        I   1315 04885            I       I   1319 04889            SUM      R   1319 04893
    AVG      R   1321 04897

                                         ERRORS FOR THIS COMPILATION

       STATEMENT                                   ERROR
        NUMBER                           ISN       NUMBER     SEVERITY     EXPLANATION

D         40                                         97          4         STATEMENT NUMBER IS UNREFERENCED
                                         006         03          8         SYNTAX ERROR
                                         013         21          8         SUBSCRIPT EXPRESSION ERR

        003 TOTAL ERRORS FOR THIS COMPILATION
        8 WAS THE HIGHEST SEVERITY
E
          STATEMENT ALLOCATIONS
    3000 =13F2  2000 =1418  1000 =142A  20    =1463  30    =1434  40    =148A
```

Figure 17. Compiler Output Listing

```
FORTRAN IV VERnnMODnn                                                          nn/nn/nn PAGE 001
// READ   DEVICE-MFCU1
// PRINT  DEVICE-5203
// DAD44  UNITNO-14
*PROCESS  MAP,LINK(T,LIB(R2)),NOSHRBUFF
    1           PROGRAM AVG
         C      THIS PROGRAM READS AND AVERAGES NUMBERS.  THE NUMBERS ARE READ
         C      IN GROUPS OF EIGHTEEN FROM THE MFCU1.
    2           DEFINE FILE 14 ( 1, 16, E, K )
         C      IT ALSO WRITES THE AVERAGE ON UNIT 14 AND READS IT BACK FOR
         C      PRINTING.
    3           DIMENSION NUMBS ( 1000 )
         C      NUMBS IS THE ARRAY INTO WHICH THE NUMBERS ARE READ.  1000 NUMBERS
         C      IS THE MAXIMUM NUMBER OF NUMBERS THAT CAN BE AVERAGED.
    4           M = 1
    5           N = 18
    6        10 READ ( 1, 1000, END = 20 ) ( NUMBS(I), I = M, N )
    7           M = M + 18
    8           N = N + 18
    9           GO TO 10
         C      READING CONTINUES UNTIL END OF FILE.  M IS ONE GREATER THAN THE
         C      NUMBER OF NUMBERS READ WHEN CONTROL IS TRANSFERRED TO 20.
   10        20 M = M - 1
   11           SUM = 0.
   12           DO 30 I = 1, M
   13             SUM = SUM + NUMBS(I)
   14        30 CONTINUE
   15           AVG = SUM / M
   16           WRITE ( 3, 2000 ) SUM, AVG
   17           WRITE ( 14'1, 3000 ) SUM, AVG
   18           READ  ( 14'1, 3000 ) SUM, AVG
   19           WRITE (3,2000)SUM,AVG
   20           STOP
   21      1000 FORMAT ( 18 I4 )
   22      2000 FORMAT ( ' SUM OF NUMBERS = ',F8.3,' AVERAGE = ',F8.3 )
   23      3000 FORMAT ( 2 F8.3 )
   24           END

NAME   AT HEX1 DEC1  HEX2 DEC2  NAME   AT HEX1 DEC1  HEX2 DEC2  NAME   AT HEX1 DEC1  HEX2 DEC2
NUMBS  I  036D 00877 130C 04876 K      I  130D 04877            M      I  1311 04881
N      I  1315 04885            I      I  1319 04889            SUM    R  131D 04893
AVG    R  1321 04897

      000 TOTAL ERRORS FOR THIS COMPILATION

        STATEMENT ALLOCATIONS
   3000 =141E  2000 =1444  1000 =1456  10    =147A  20    =14C3  30    =14FE
```

Figure 18. Compiler Output Listing After Corrections

Object Module Card Deck

An object module card deck is produced consisting of the records described in *Object Module*, and the following:

- A disk utility // COPY statement to instruct the system to insert the object module into a system library. This statement is the first card in the deck.
- A header card that contains the object module's name. This statement follows the // COPY statement.
- A disk utility // CEND statement following the records to indicate the end of the card deck.

Source Module Listing

The source module listing in Figure 17 is labeled **B**. The source module listing prints FORTRAN source statements together with their corresponding ISNs (internal statement numbers) assigned by the compiler to all FORTRAN source statements except comments statements and continuation lines.

Storage Map

The storage map in Figure 17 is labeled **C** . The storage map is a listing of each variable name used in the program, and such information as whether it describes an integer or real item, whether the item is in GLOBAL or COMMON, whether the item is undefined, and the storage address of the item, shown in both hexadecimal and decimal notation. Items in COMMON are listed first with storage addresses relative to the beginning of the COMMON area. Local variables (variables not in COMMON) are listed next, with storage addresses relative to the beginning of the local program area. The storage map appears on an output listing under the headings:

> NAME AT HEX1 DEC1 HEX2 DEC2
> (Headings appear three times per line.)

where:

NAME indicates the name of the variable.

AT indicates the attributes of the variable. Attributes can be printed using three print positions. The first print position indicates whether the variable is integer (I), real (R), halfword (H), or doubleword (D). The second print position indicates whether it is in GLOBAL (G) or COMMON (C); if the item is a local variable, this position is left blank. The third print position identifies an undefined variable (U).

HEX1 indicates the address where a variable or array begins, in hexadecimal.

DEC1 indicates the same address in decimal.

HEX2 is used with an array to indicate where the array ends; in hexadecimal. If the item is not an array this address is left blank.

DEC2 indicates the same address in decimal.

The headings are repeated three times across an output listing page, so that each line of the map can describe three data items. The first data item described, NUMBS, is of type INTEGER (I), begins at location 877 (36D in hexadecimal) and ends at location 4876 (130C in hexadecimal). NUMBS is a 4000-byte array. The data item SUM (the last item listed) is of type REAL, and is located at address 4893 (131D in hexadecimal). Because SUM is not an array, no end address is shown.

LINKAGE EDITOR OUTPUT

The link-editing step produces the following output:

- A load module
- A linkage editor listing of informative and diagnostic messages
- Output determined by options

Load Module

The load module is in the form of a series of records, each 96 bytes long. There are four types of records in a load module:

1. One header record
2. A group of text records
3. A group of relocation records
4. One end record

Header Record

A header record defines the load module's reference name and start control address. The header record is identified by the character H in the first byte of the record.

Text Records

Text records contain the object code of the load module. Text records are identified by the character T in the first byte of the record. Text records make up the bulk of load module records.

Relocation Records

Relocation records contain information needed to make the load module relocatable in the event the module cannot be loaded at the start control address specified in the header record. Relocation records are identified by the character R in the first byte of the record.

End Record

The end record defines the end of the load module. It is identified by the character E in the first byte of the record.

Linkage Editor Output Listing

Figure 19 illustrates link-editing output for the program shown in Figure 14.

Messages

Like compilation messages, linkage editor messages inform you of the status of the link edit step and whether any errors were encountered.

Informative messages in Figure 19 are labeled **B** at the end of the link edit output listing. Overlay linkage editor messages OL100, OL101, and OL104 print out, respectively, the size of the program, its beginning address (start control address), and the number of disk sectors the load module requires in the library. These messages are followed by a summary of OCL parameters that identify the load module.

Diagnostic messages are in the form of halt codes that are displayed on the display unit. No halt codes are generated for the program illustrated here. For a description of halt codes, see the appropriate halt guide listed under *Related Publications* in the Preface.

Output from Options

Linkage editor output can be produced by the compiler options GODECK and MAP. GODECK produces a card deck of the load module. To obtain the card deck, you must explicitly request GODECK at compile time, the default being NOGODECK. The MAP option causes the linkage editor to print a core usage map of the load module.

Load Module Card Deck

A load module card deck is produced consisting of the records described in *Load Module* and the following:

- A disk utility // COPY statement to instruct the system to insert the load module into a system library. This statement precedes the load module cards.

- A disk utility // CEND statement to indicate the end of a card deck.

Core Usage Map

The core usage map in Figure 19 is labeled **A** . The map appears under the headings:

START ADDRESS	CATEGORY	NAME AND ENTRY	CODE LENGTH HEXADECIMAL DECIMAL

where

START ADDRESS indicates the beginning storage address of the area or routine, in hexadecimal notation. For example, the main routine AVG begins at location 1300, and the FORTRAN library routine $FOE0 at location 2868.

CATEGORY indicates the priority value of each routine for remaining in main storage in an overlay environment (the lower the number the greater the priority).

NAME AND ENTRY indicates the name of each area or routine, and, indented from the main entry point for each routine, other entry point names within the routine. For example, in addition to its main entry point, $FOE0 contains other entry point names #MNTRY, #SNTRY, #RNTRY, and #D.

CODE LENGTH indicates the length of each area or routine, in both hexadecimal and decimal notation.

OVERLAY LINKAGE EDITOR CORE USAGE MAP nn/nn/nn

A

START ADDRESS	CATEGORY	NAME AND ENTRY	CODE LENGTH HEXADECIMAL	DECIMAL
1300	255	AVG	1568	5480
2858		#UNITB		
1600		#ERBUF		
1588		#IOBUF		
2868	0	$F0E0	0193	403
2905		#MNTRY		
299A		#SNTRY		
2909		#RNTRY		
2869		#D		
29FB	0	$F0B1	0062	98
2A2C		#DED4		
2A4D		#DEDZ0		
2A5D	0	$F0I0	00F5	245
2AF4		#ELST		
2B00		#ELST2		
2AE5		#DERR		
2AD3		#IOINT		
2ABF		#IOCOM		
2B0C		#ENDEQ		
2B13		#ERREQ		
2B24		#OUTBL		
2B1A		#INTBL		
2B30		#IO000		
2B31		#FLRP2		
2B52	0	$F0VG	000C	12
2B5D		#ADROT		
2B5B		#RLIST		
2B5E	0	$F0VH	0010	16
2B6D		#DLIST		
2B6E	2	$F0I6	0173	371
2C68		#F0I6A		
2C4B		#F0I6B		
2CE1	2	$$DAUB	0068	104
2D49	2	$$SRDA	0099	153
2DE2	2	$$SRDI	0038	56
2E01		DMSRPD		
2DFA		DMSRRD		
2E1A	2	$$SRRC	007B	123
2E95	2	$$SRRI	0029	41
2EBE	2	$$SRMD	0081	129
2F3F	2	$$SRTC	001C	28
2F3F		DMSRLD		
2F50		DMSRTC		
2F53		DMSRER		
2F5B	4	$F0E3	0028	40
2F62		#XLI		
2F68		#XST		
2F6F		#XA		
2F76		#XMLI		
2F7C		#XMST		
2F83	4	$F0E6	001C	28
2F98		#BST		

Figure 19 (Part 1 of 2). Linkage Editor Output Listing

```
START      CATEGORY  NAME AND          CODE LENGTH
ADDRESS              ENTRY      HEXADECIMAL  DECIMAL

2F8A                  #BA
2F91                  #BS
2F9F         4      $F0EA          0154        340
2FA2                  #RL
2FB3                  #RST
2FCC                  #RA0
2FCF                  #RA
2FBE                  #RS0
2FC1                  #RS
300F                  #RT0
30F3         4      $F0ED          00E1        225
31A3                  #DOEND
3177                  #DOBR
3121                  #DBGN3
31D4         4      $F0EE          0083        131
3257         4      $F0EH          0098        152
32EF         4      $F0VC          0061         97
32FE                  #FLOAT
3350         4      $F0EC          0020         32
3356                  #ROFLW
3366                  #RUFLW
3370         5      $F0B2          0134        308
3473                  #FRET
347B                  #F0B2A
33F3                  #F0B23
34B5                  #F0B2C
34A4         5      $F0C2          00CF        207
3573         5      $F0C4          01FC        508
376F         5      $F0C5          0294        660
3779                  #F0UTE
3A03         5      $F0B8          004E         78
3A51         5      $F0B9          0018         24
3A69         5      $F0BA          001D         29
3A86         5      $F0CF          00BC        188
3A8D                  #F0CF3
3B42         5      $F0CH          0120        288
3B5B                  #F0CH3
3C62         5      $F0CG          026A        618
3E11                  #DIVID
3E95                  #MULT
3ECC         6      $F0IC          0096        150
3F38                  #ERTST
3F62         6      $$LPRT         00FB        251
4050         6      $F0I3          00E4        228
4063                  #F0I3A
4114                  #F0I3
4141         6      $F0IL          005E         94
4163                  #F0ILA
419F         6      $F0VP          0019         25
41B8         6      $F0IB          009C        156
4254         6      $F0BB          006D        109
42C1         6      $F0D7          012C        300
43ED         6      $$MFRD         0145        325
```

A

B

```
OL100 I  THE TOTAL CORE USED BY AVG     IS 12850 DECIMAL
OL101 I  THE START CONTROL ADDRESS OF THIS MODULE IS 1300.
OL104 I  TOTAL NUMBER OF LIBRARY SECTORS REQUIRED IS    53
         NAME-AVG    ,PACK-R1R1R1,UNIT-R2,RETAIN-T,LIBRARY-O.
```

Figure 19 (Part 2 of 2). Linkage Editor Output Listing

LOAD MODULE OUTPUT

Output from the load module step can be the following:

- Informative and diagnostic messages
- Printed output results generated by the FORTRAN program

Figure 20 illustrates load module output for the program shown in Figure 14.

Messages

The output listing for a load module step begins with a list of OCL load module statements, labeled **A** in Figure 20.

Load module diagnostic messages are halt codes displayed on the display unit rather than on an output listing. Two halt codes are associated with FORTRAN load module processing: b̸6 to indicate that a PAUSE or STOP statement was executed, and b̸7 to indicate that an execution error occurred. On the Model 6, these halts are displayed as follows: b̸6 as 4 and b̸7 as 13.

When a b̸6 halt code occurs as a result of a PAUSE statement, you can either continue processing (by responding with a 0 option), or terminate processing (by responding with a 2 or 3 option).

When a b̸7 halt code occurs, you must terminate processing (by responding with a 2 or 3 option). Halt code b̸7 has several secondary halt codes that identify the cause of the error.

For procedures concerning responding to halts, and a description of secondary halts, see the appropriate Halt Guide--as listed in the Preface under *Related Publications.*

Figure 21 shows the options possible for PAUSE statements, STOP statements, and execution error halts.

Program Output

Output from execution of the FORTRAN program is labeled **B** in Figure 20. This output indicates the sum and the average of the numbers read. It appears in duplicate as a result of two executions of the WRITE statement, one from computations made in storage, the other from reading the results back in from direct access file number 14.

Using the FORTRAN Traceback Listing

If an execution error occurs in a program containing a number of program units, the traceback facility can be used to locate the program unit where the error occurred. To illustrate how the traceback facility can be used, refer to the program shown in Figure 22. Part 1 of this figure shows compiler output for the subroutines X and Y. Part 2 shows compiler output for the main program unit ##MAIN.

These program units perform the following operations:

- ##MAIN twice passes a value to subroutine X, in the variable A. The first time passed, A has the value 4; the second time, the value 1.
- Subroutine X accepts the value in its variable J. X then subtracts 1 from the value, calls subroutine Y, and upon return from Y again subtracts 1 and calls Y.
- Subroutine Y uses the value in J as a logical unit number in a WRITE statement to print out a message.

A

```
// LOAD  AVG,R2
//  FILE UNIT-F2,PACK-544400,TRACKS-20,RETAIN-S,NAME-FT00014,LABEL-MYFILE
// RUN
```

B

```
SUM OF NUMBERS =  171.000 AVERAGE =     9.500
SUM OF NUMBERS =  171.000 AVERAGE =     9.500
```

Figure 20. Load Module Output

Part 3 of Figure 22 shows the traceback listing resulting from an execution error in this program. The traceback listing contains the following information:

ROUTINE indicates the name of the routine. The name shown at the top of the list identifies the routine that was executing when the error occurred. The name immediately below that identifies the routine calling the top listed routine, and below that, are listed the other calling routines.

ISN specifies the internal statement number in the routine that called the routine listed. For example, subroutine Y was called by ISN 5 in subroutine X.

XR1, XR2, and ARR specify the contents of index register 1, index register 2, and the address recall register, respectively. This information is useful to the IBM customer engineer if he must service the system.

The traceback listing in Figure 22 tells a number of things:

- Subroutine Y was executing when the error occurred.
- Subroutine Y was called by ISN 5 in Subroutine X.
- Subroutine X in turn was called by ISN 3 in ##MAIN.

ISN 3 in ##MAIN is the first call to X, when A has the value 4. ISN 5 in X is the second call to Y, after X has subtracted the value of J a second time. (The first call resulted in printing the message shown before the traceback listing.) Thus, J has the value 2 when Y is called in ISN 5. Since Y uses J as a logical unit number, the error is that the number 2 has not been assigned to an output device. (No error occurred when J has the value 3 because 3 is assigned to a printer by default.)

Obtaining a Traceback Listing

To obtain a traceback listing, do the following:

1. Specify GOSTMT in the *PROCESS statement for each program unit to be traced. (If GOSTMT is not specified for a program unit, internal statement numbers in that unit are not displayed.)

2. If an error occurs, the character 7 (from the halt code b̸7) appears on the display unit. Respond with option 0 to get the secondary halt. (The error in Figure 22 would cause secondary halt code 10 to appear.) Respond with option 2. This causes FORTRAN to terminate job execution and print out the traceback listing.

	Halt			Options
	Model 6	Models 10 and 12	Model 15	
PAUSE n	4	b̸6	b̸6	0 Continue processing. 2 Have FORTRAN cancel the program.
STOP n	4	b̸6	b̸6	2 Have FORTRAN cancel the program. 3 Have system cancel the program.
Execution error	13	b̸7	b̸7	2 Have FORTRAN cancel the program. 3 Have system cancel the program.

Figure 21. Possible Options for PAUSE, STOP, and Execution Errors

```
// CALL FORTRN,F1
XX  LOAD $FORT,F1
XX  FILE NAME-$WORK,UNIT-R1,PACK-R1R1R1,TRACKS-20,RETAIN-S
XX  FILE NAME-$SOURCE,UNIT-R1,PACK-R1R1R1,TRACKS-20,RETAIN-S
// RUN
```

```
 FORTRAN IV VERnn/MODnn                                                        nn/nn/nn PAGE 001

*PROCESS MAP,GOSTMT
   1             SUBROUTINE X(J)
   2           2 J=J-1
   3           3 CALL Y(J)
   4             J=J-1
   5           5 CALL Y(J)
   6             RETURN
   7             END

NAME    AT HEX1 DEC1   HEX2 DEC2   NAME    AT HEX1 DEC1   HEX2 DEC2 NAME    AT HEX1 DEC1   HEX2 DEC2
J        1  0006 00006
```

```
// CALL FORTRN,F1
XX  LOAD $FORT,F1
XX  FILE NAME-$WORK,UNIT-R1,PACK-R1R1R1,TRACKS-20,RETAIN-S
XX  FILE NAME-$SOURCE,UNIT-R1,PACK-R1R1R1,TRACKS-20,RETAIN--S
// RUN
```

```
 FORTRAN IV VERnn/MODnn                                                        nn/nn/nn PAGE 001

*PROCESS MAP,GOSTMT
   1             SUBROUTINE Y(J)
   2           2 WRITE(J,100)
   3         100 FORMAT (' IN SUBROUTINE Y')
   4             RETURN
   5             END

NAME    AT HEX1 DEC1   HEX2 DEC2   NAME    AT HEX1 DEC1   HEX2 DEC2   NAME    AT HEX1 DEC1   HEX2 DEC2
J        1  0006 00006
```

Figure 22 (Part 1 of 3). Traceback Example

```
// CALL FORTRN,F1
XX  LOAD $FORT,F1
XX  FILE NAME-$WORK,UNIT-R1,PACK-R1R1R1,TRACKS-20,RETAIN-S
XX  FILE NAME-$SOURCE,UNIT-R1,PACK-R1R1R1,TRACKS-20,RETAIN-S
// RUN
```

```
 FORTRAN IV VERnn/MODnn                                                 nn/nn/nn PAGE 001

*PROCESS MAP,GOSTMT
   1          INTEGER A
   2          A=4
   3          CALL X(A)
   4          A=1
   5          CALL X(A)
   6          STOP
   7          END

NAME    AT HEX1 DEC1   HEX2 DEC2   NAME    AT HEX1 DEC1   HEX2 DEC2   NAME    AT HEX1 DEC1   HEX2 DEC2
A       I  0185 00389

    000 TOTAL ERRORS FOR THIS COMPILATION
```

Figure 22 (Part 2 of 3). Traceback Example

```
// LOAD ##MAIN,F1
// RUN

IN SUBROUTINE Y

  TRACEBACK FOLLOWS  ROUTINE  ISN    XR1   XR2   ARR

                     Y        00005  1504  16B7  1705

                     X        00003  1504  14CF  14E7

                     ##MAIN
```

Figure 22 (Part 3 of 3). Traceback Example

Part 3. Programming Considerations

This section contains:

- Direct-access programming considerations
- Sequential disk and tape programming considerations
- FORTRAN implementation considerations
- System considerations

Chapter 17. Direct-Access Programming Considerations

In FORTRAN, the two types of input/output (I/O) operations supported for direct-access files are formatted and unformatted.

The record length for a direct-access file is specified in the DEFINE FILE statement. The block length is specified by the BLOCKSIZE parameter in the DEVICE OPTIONS statement.

Formatted I/O

Formatted I/O has a FORMAT statement associated with the READ or WRITE statement that is used to access the data in the file. The amount of data to be transferred for a formatted READ or WRITE statement is determined by the format codes in the FORMAT statement and the number of variables in the I/O list.

The FORMAT statement used to control the reading or writing must not specify more characters (bytes) than there are in a record.

For example, to process a file described by the statement:

```
DEFINE FILE 8(10,48,L,K8)
```

the FORMAT statement used to control reading or writing could not specify more than 48 bytes of data.

The following are valid FORMAT statements:

```
FORMAT(4F12.1)
FORMAT(I10,9F4.2)
```

The following are invalid FORMAT statements:

```
FORMAT(6F10.2)
FORMAT(I10,4F12.2)
```

The I/O list for a READ or WRITE statement must always be satisfied. That is, if there are more variables in the I/O list than there are format codes, the FORMAT statement will be reused. When reused, a new record will be transferred. This will continue until the I/O list is satisfied.

If the amount of data to be written is less than the record length, the record is padded on the right with blanks.

Unformatted I/O

Unformatted I/O has no FORMAT statement associated with the READ or WRITE statement that is used to access the data in the file. The amount of data to be transferred for an unformatted READ or WRITE statement is determined by the number and type of variables in the I/O list. There is a one-to-one correspondence between internal storage locations (bytes) and external record positions.

The I/O list for a READ or WRITE statement must always be satisfied. If the amount of data specified in the I/O list exceeds the record length, more than one record will be read or written, enough to satisfy the I/O list.

If the amount of data to be written is less than the record length, the record is padded on the right with binary zeros.

SHARING THE ASSOCIATED VARIABLE BETWEEN PROGRAMS

Programs can share an associated variable as a COMMON or GLOBAL variable. The following example shows how this can be accomplished:

```
   COMMON IUAR              SUBROUTINE SUBI(A,B)
   DEFINE FILE 8(100,10,    COMMON IUAR
   L,IUAR)
          .                          .
          .                          .
          .                          .
   ITEMP=IUAR
   CALL SUBI (ANS,ARG)
 4 IF (IUAR-ITEMP) 20,16,20
          .
          .
          .
```

In this example, the program and the subprogram share the associated variable IUAR. An input/output operation that refers to logical unit 8 and is performed in the sub-routine changes the value of the associated variable. The associated variable is then tested in the main program in statement 4.

An associated variable can also be passed to a subprogram as an argument in a CALL statement. However, in a subprogram the dummy variable is not automatically updated unless it is passed to the subprogram via COMMON or GLOBAL and not through the argument list. If the variable is to be passed through the argument list, you must update it yourself with FORTRAN statements.

MINIMIZING DIRECT-ACCESS I/O TIME

You can decrease I/O time by processing a direct-access file consecutively. Consecutive processing of a direct-access file occurs when the relative record number is increased by one each time the file is accessed. This can be done by using the associated variable in the DEFINE FILE statement as the relative record number in the I/O statement.

Example:

```
     DEFINE FILE 10(100,64,L,I)
                .
                .
                .
     I=1
   5 READ(10'I)IRAY
                .
                .
                .
     GO TO 5
```

The associated variable (I) in the DEFINE FILE statement is initially set to 1 by an assignment statement (I=1). It is also used as the relative record number in the READ statement. After the READ statement is executed, the associated variable is automatically updated by FORTRAN to point to the next record. Thus, as the loop is executed a number of times, the file is processed consecutively.

Consecutive processing of a direct-access file can also be done by updating the relative record number by one each time the file is accessed.

Example:

```
     DEFINE FILE 10(100,64,L,I)
                .
     I=1
                .
     J=1
   5 READ(10'J)IRAY
                .
                .
                .
     J=J+1
     GO TO 5
```

If a direct-access file is processed consecutively, you can further decrease I/O time by specifying two buffers and a blocksize as large as possible without exceeding the amount of main storage available for buffers in the load module's environment. (Remember that the main storage needed for buffers is doubled when two buffers are specified.) See *Buffer Assignment for Direct-Access Files.*

When BLOCKSIZE and BUFFERS are used, the program must be in the NOSHRBUFF environment. This can be specified in the *PROCESS statement. (NOSHRBUFF is a default option in the *PROCESS statement.)

Note: If a file is being processed randomly, I/O time might be increased when BLOCKSIZE and BUFFERS are used.

BUFFER ASSIGNMENT FOR DIRECT-ACCESS FILES

The system accesses records from a direct-access file only in 256-byte segments to correspond with the arrangement of disk storage into 256-byte sectors. The size of the buffer allocated to a file must be an exact multiple of 256, and is determined by the blocksize specified in the DEVICE OPTIONS statement. If the BLOCKSIZE parameter is not specified, the record length from the DEFINE FILE statement is used to calculate the size of the buffer.

The following rules apply when buffer space is being allocated:

1. If the block length (or record length) is a submultiple of 256, a 256-byte buffer is allocated.

2. If the block length (or record length) is a multiple of 256, the buffer size is equal to the block length (or record length).

3. If the block length (or record length) is neither a multiple nor submultiple of 256, the buffer size is determined as: block length (or record length) + 255 raised to the next higher multiple of 256.

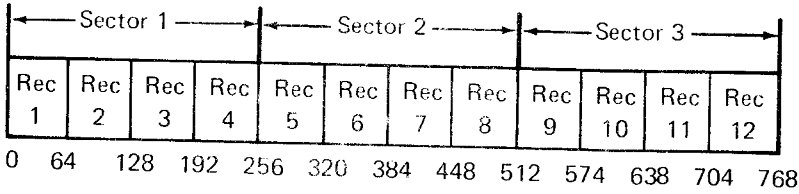
If the file's DEVICE OPTIONS statement specifies BUFFERS-2, the buffer size is doubled before it is allocated.

The following examples illustrate these rules:

Example 1:

```
  // DAD44 UNITNO-10
          .
          .
          .
     DEFINE FILE 10(100,64,L,I)
```

In this case, BLOCKSIZE is not specified, so the record length (64) from the DEFINE FILE statement is used to calculate the size of the buffer. The records are stored on disk as follows:



The record size is a submultiple of 256. Therefore, a buffer size of 256 is allocated.

Example 2:

```
// DAD44 UNITNO-12,BLOCKSIZE-768
    .
    .
    .
  DEFINE FILE 12(200.64,L,I)
    .
    .
```

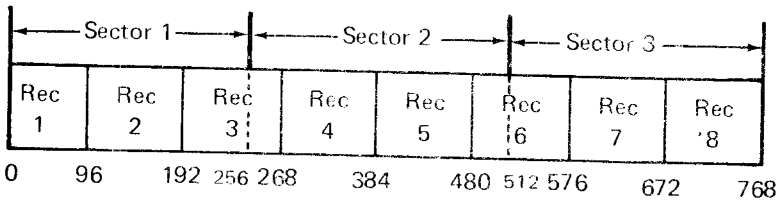
In this case, the BLOCKSIZE is specified as 768. The records are stored the same as in the previous example.

The block length is a multiple of 256. Therefore, the length allocated for the buffer is the block length. In this case, FORTRAN accesses 12 records with each physical I/O operation (if the file is being processed consecutively).

Example 3:

```
// DAD44 UNITNO-14
    .
    .
    .
  DEFINE FILE 14(50,96,L,I)
    .
    .
```

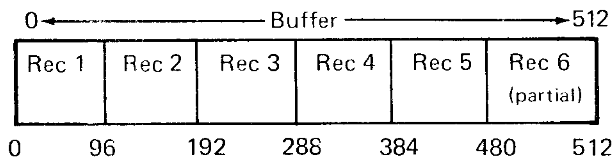
In this case, BLOCKSIZE is not specified, so the record length (96) from the DEFINE FILE statement is used to calculate the size of the buffer. The records are stored on disk as follows:



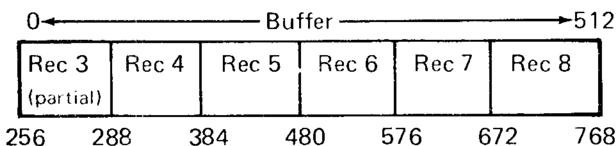
Record 3 straddles two sectors. In order to process this record, a buffer of 512 must be allocated, using the rule:

> Record length (96) +255 (=351) raised to the next multiple of 256 (512). Two sectors are loaded into each buffer.

When the buffer is loaded with sectors 1 and 2, it contains the following records:



Records 1 through 5 can be processed. When record 6 is to be processed, the second and third sectors are placed into the buffer, as follows:



Record 6 can now be processed, as can records 7 and 8. When record 9 is to be processed, the appropriate sectors are placed into the buffer.

Sharing Buffers

Direct-access files on disk drives can share a buffer. The size of the buffer is equal to the maximum block size computed. For example, if the three files shown in the preceding examples were to share a buffer, the I/O buffer size would be allocated as 768 bytes, the size of the largest buffer needed.

Buffers cannot be shared with a file that has BUFFERS-2 specified in the device option statement, or EXTBUF specified in the *PROCESS statement. For best performance, buffers should not be shared with a file that has BLOCKSIZE larger than 256 specified (or defaulted) on the associated device option statement.

File Share (Program Number 5704-FO2 Only)

Direct-access files on disk drives can be shared with another task or partition at execution time by using system control programming (SCP) statements. File share causes direct-access files with double buffers to default to single buffer status during execution. The SCP default parameter for // FILE statements is SHARE-YES. Therefore, it is necessary to specify SHARE-NO if double buffering is used during program execution.

Example:

```
// CALL FORTRN,F1
// RUN
// DAD44 UNITNO-15,BUFFERS-2
FORTRAN source program
/*
// LOAD ##MAIN,F1
// FILE NAME-FT00015,PACK-654321,UNIT-R1,
   RETAIN-T,
// RECORDS-800,SHARE-NO
// RUN
Program data
/*
```

For further information, see the appropriate System Control Programming Reference Manual listed under *Related Publications* in the Preface.

Chapter 18. Sequential Disk and Tape Programming Considerations

In FORTRAN, the two types of record formats supported for sequential disk and tape files are fixed-length and variable-length. Fixed-length records are transferred for formatted or list-directed input/ouput (I/O). Variable-length records are transferred for unformatted I/O.

The record length is defined by the BLOCKSIZE parameter on the DEVICE OPTIONS statement. This value is used as the record length in the volume table of contents (VTOC) for disk files and the header label for tape files When FORTRAN sequential files are being accessed, this value must be used in other programming languages as the record length.

FORMATTED OR LIST - DIRECTED I/O

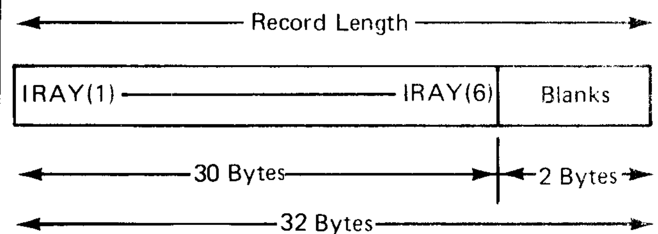
The amount of data to be transferred for a formatted READ or WRITE statement is determined by the format codes in the FORMAT statement and the number of variables in the I/O list. For list-directed I/O, the amount is determined by the type and number of variables in the I/O list.

For a READ statement, if the amount of data to be read exceeds the record length, multiple records will be read (enough to satisfy the I/O list). If the amount of data to be read is less than the record length, the remaining data in the record is skipped.

For a WRITE statement, if the amount of data to be written exceeds the record length, multiple records will be written (enough to satisfy the I/O list). Data from a variable will never span a record. If the data from a variable does not fit at the end of a record, it will be the first data placed in the next record. If the amount of data to be written is less than the record length, the record will be padded on the right with blanks.

Example:

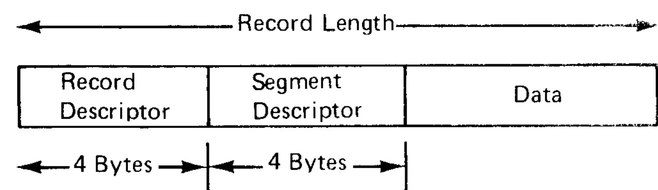
```
// SEQ44 UNIT NO-10, BLOCKSIZE-32
   .
   .
   .
DIMENSION IRAY (6)
   .
   .
WRITE (10,20) IRAY
20 FORMAT (6I5)
   .
   .
```



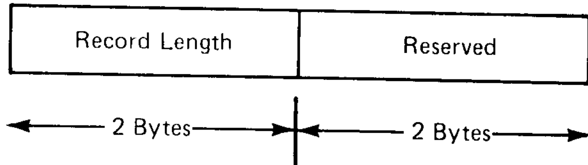
The BLOCKSIZE parameter specifies a record length of 32 bytes. The FORMAT statements specify 30 bytes (six fields, each 5 bytes long). The I/O list has six variables (six elements of array IRAY). If the amount of data to be written is 30 bytes, which is less than the record length of 32 bytes, the record is then padded with 2 bytes of blanks.

UNFORMATTED I/O

A record is made up of a 4-byte record descriptor, a 4-byte segment descriptor, and a data segment.



The format of the record descriptor is:

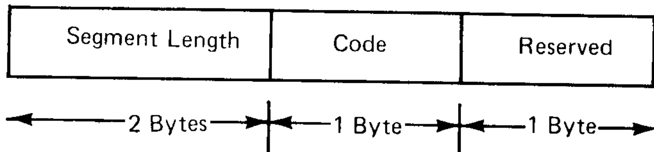


where

record length is the value given in the BLOCKSIZE parameter. The record length is stored as a binary value.

reserved is filled with binary zeroes.

The format of the segment descriptor is:



where

segment length is the length in binary of the segment of data plus the 4 bytes of the segment descriptor.

code is used to indicate the position of this segment with respect to the other segments, if any, of the FORTRAN record.

Code	Meaning
00	The only segment in the record.
01	The first segment of a multisegment record.
02	The last segment of a multisegment record.
03	Neither first nor last segment of a multisegment record.

reserved is filled with binary zeroes.

The amount of data to be transferred for each unformatted READ or WRITE statement is determined by the number and type of variables in the I/O list. The data can be recorded in one or more segments.

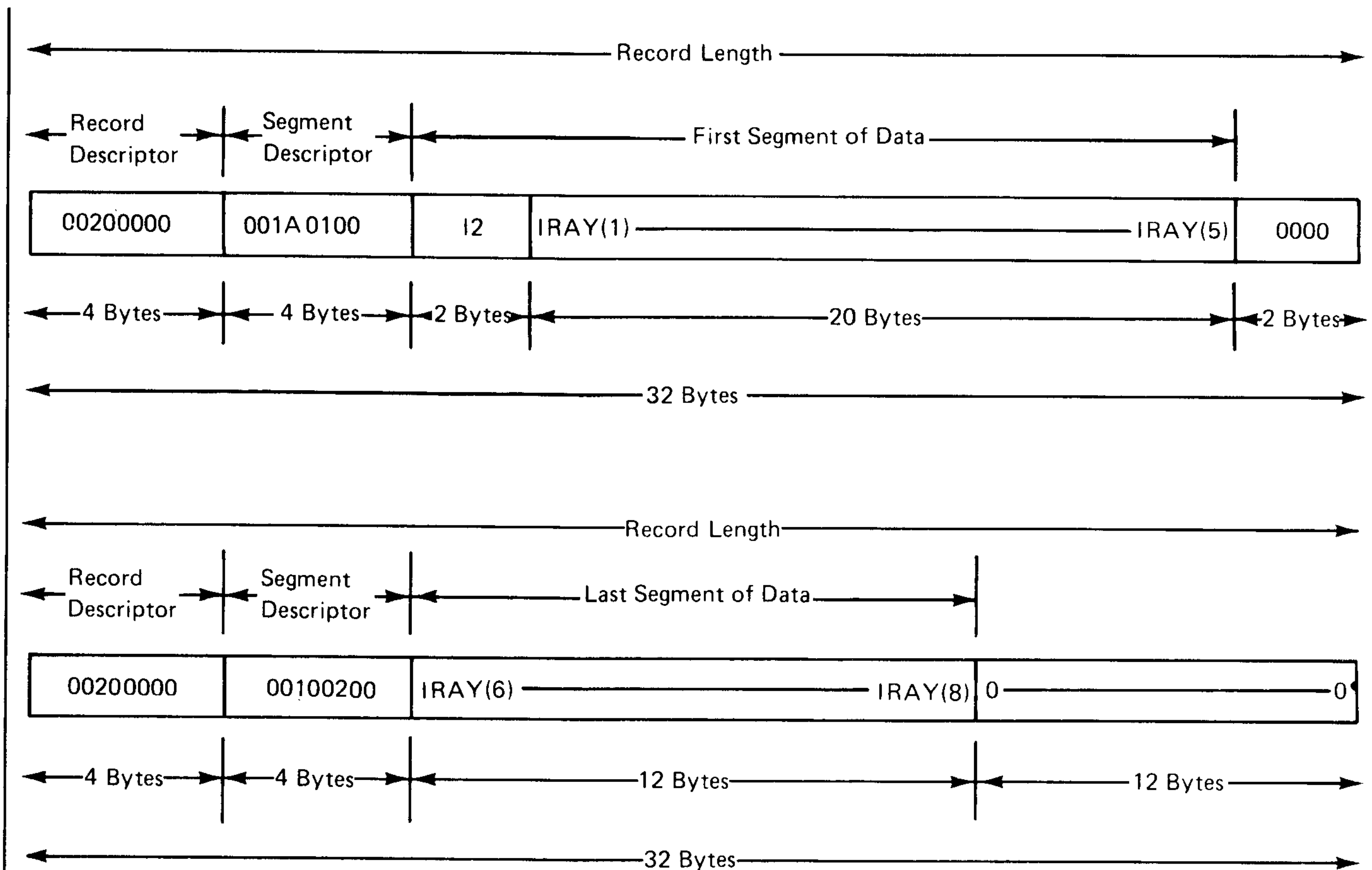
For a READ statement, if the amount of data to be read exceeds the data in a segment, multiple segments will be read. Whether the record has a single segment or multisegments, the amount of data to be read in a single READ statement must never exceed the amount of data previously written into a record. If the amount of data to be read is less than the amount previously written, the remaining data in that segment and any other segments of a multisegment record are skipped.

For a WRITE statement, if the amount of data to be written exceeds the record length minus 8 (the length of the descriptors), multiple segments will be written (enough to satisfy the I/O list). Data from a variable will never span a record. If the data from a variable does not fit at the end of a segment, it will be the first data placed in the next segment for that record.

If the amount of data to be written is less than the record length minus 8 (the length of descriptors), the segment will be padded on the right with binary zeroes.

Example:

```
// SEQ44 UNIT NO-10 BLOCKSIZE-32
      .
      .
      .
   INTEGER*2 I2
   DIMENSION IRAY(8)
      .
      .
      .
   WRITE (10) I2, IRAY
      .
      .
```



The BLOCKSIZE parameter specified a record length of 32 bytes. The amount of data to be transferred is 34 bytes: one variable 2 bytes long (I2 is integer *2) and eight variables 4 bytes long (8 elements of array IRAY). The amount of data that can be placed in a segment is 24 bytes (32 minus 8 bytes of descriptor). Thus, more than one segment must be written to satisfy the I/O list. Variable IRAY (6) will not fit in the 2 bytes at the end of the first 4 bytes of the next segment.

The record descriptor for both segments indicate a length of 32 (X'20'). The segment descriptor for the first segment indicates a length of 26 (X'1A'); (22 bytes of data plus 4 bytes of descriptor and a code of X'01' which indicates the first segment of a multisegment record). The length in the next segment descriptor is 16 (X'10'); 12 bytes of data plus 4 bytes of descriptor). The code X'02' indicates the last segment of a multisegment record.

BUFFER ASSIGNMENT FOR SEQUENTIAL FILES

The size of the buffer allocated for a tape file is either the value given on the BLOCKSIZE parameter of the TAPE DEVICE OPTIONS statement or 128, whichever is larger. The buffer will contain only one record at a time.

The size of the buffer allocated for a disk file is always 256 bytes (the size of a disk sector). The buffer can contain from 1 to 16 records, depending on the record length. To keep records from spanning sectors, the record length must be 256 or one of the following submultiples: 128, 64, 32, or 16.

To minimize I/O time and make the most efficient use of disk space, the record length should be the smallest submultiple. This value can be calculated as follows:

- *Formatted I/O:* Add the lengths given in the FORMAT statement for the number of variables in the I/O list.
- *List-Directed I/O:* Add the lengths needed for the variables in the I/O list. The length needed for each type of variable is described in Chapter 5 under *List-Directed Output Data.*
- *Unformatted I/O:* Add the lengths of the variables in the I/O list. To this total, add 8 (descriptor bytes).

RESTRICTIONS ON THE ORDER OF SEQUENTIAL I/O OPERATIONS

1. A READ request cannot follow a WRITE or END FILE request on the same file.
2. A WRITE request cannot follow an END FILE request on the same file.
3. An END FILE request cannot immediately follow another END FILE request on the same file.

MODEL 15 MULTIFILE TAPE PROCESSING

Multiple files per tape volume are supported only by Model 15 FORTRAN. The file sequence number is specified on the FILE statement at execution time. A further description can be found in the *IBM System/3 Model 15 System Control Programming Reference Manual,* GC21-5077.

TAPE PROCESSING IN PROGRAMS USING OVERLAYS

Tape data management is overlay category 3, and the FORTRAN I/O interface modules for sequential disk and tape are overlay category 2. This can cause problems for large programs that are forced into overlays when using tape I/O. In such cases, a separate link step should be performed with a linkage editor control statement changing the tape data management to category 2. The section describing *Linkage Editor Control Statements* includes the following category override statement:

```
// CATEGORY NAME-$$BTAM,VALUE-2
```

Directing Program Output to Both a Printer and a Card Punch

If program output is to be both punched and printed, use two WRITE statements with the appropriate logical unit number assignments, and include both a PUNCH and a PRINT device option statement in the job stream.

Example:

```
//     PRINT DEVICE-5471
//     PUNCH DEVICE-1442
       .
       .
       .
       WRITE(5,10)J,K
       WRITE(9,11)J,K
10     FORMAT('0',I10,5X,I5)
11     FORMAT(I10,5X,I5)
```

The example states that the variables J and K are to be printed on the device assigned logical unit number 5 (the 5471 printer/keyboard) and punched on the device assigned logical unit number 9 (the 1442 card read punch). The only difference between the two FORMAT statements is that statement 10 specifies in its first position the carriage control character needed for printing operations.

DEBUG Facility Considerations

When specifying SUBCHK or INIT to display array values, keep the following in mind:

- The INIT option displays both the array subscript and the array element value.
- The SUBCHK option displays only an invalid array subscript.
- The element number displayed is shown displaced from location 0 in the array. In other words, the first element would show beginning at location 0.
- The subscript number displayed is shown displaced from location 0 in bytes, not in words. For an array typed as length 4, the first subscript would show beginning at location 0, the second at location 4, the third at location 8, etc. Arrays typed as length 2 would be displaced by multiples of 2; elements typed as length 8, by multiples of 8.

For an example, refer to Figure 23. Part 1 of Figure 23 shows the compiler output for a program that tests certain elements of the array (-1, 0, 1, 10, and 11) using debug options SUBCHK and INIT. Part 2 of Figure 23 shows the debug output after program execution. Ignore for the moment the first four items displaying unusual subscript values in parentheses. The fifth through eighth items show the subscript as a displacement from 0. IA is an INTEGER*4 array (by default); thus, the tenth element is shown beginning at location 36 (IA(00036)=10); the eleventh element is shown beginning at location 40 (IA(00040)=11); because this element is beyond the range of the ten element array IA, it is flagged by SUBCHK.

Refer to the first four items, The unusual subscripts are displayed of an attempt to display subscripts occurring before the beginning of the array (-1 and 0). To identify the subscript that was meant to be displayed, use the following formula:

V/L+1±N=subscript value

where

V is the value shown enclosed in parentheses.

L is the length of the item being tested. L can have the value 2, 4, or 8.

N is the number 82768 if L is 2, 16384 if L is 4, or 8192 if L is 8.

Using this formula to identify the first value, SUBCHK IA(65528):

V=65528
L=4 (default length of array IA)
N=16384
65528÷4=16382+1=16383-16384=-1

Note: Because SUBCHK and INIT display only the last two bytes, a subscript value displayed may be part of a larger number; in this case, the subscript can be identified by adding, rather than subtracting, N.

```
FORTRAN IV VERnn/MODnn                    nn/nn/nn PAGE 001

// PRINT DEVICE-5203
   1          DIMENSION A(10),IA(10),B(10)
   2          I=-1
   3          IA(I)=I
   4          I=0
   5          IA(I)=I
   6          I=1
   7          IA(I)=I
   8          I=10
   9          IA(I)=I
  10          I=11
  11          IA(I)=I
  12          STOP
  13          DEBUG SUBCHK,INIT(IA)
  14          END

   000 TOTAL ERRORS FOR THIS COMPILATION
```

Figure 23 (Part 1 of 2). Debug Display

```
// LOAD ##MAIN,R2
// RUN

  -DEBUG-SUBCHK        IA(65528)

  -DEBUG-INIT        IA(65528)=          -1

  -DEBUG-SUBCHK        IA(65532)

  -DEBUG-INIT        IA(65532)=           0

  -DEBUG-INIT        IA(00000)=           1

  -DEBUG-INIT        IA(00036)=          10

  -DEBUG-SUBCHK        IA(00040)

  -DEBUG-INIT        IA(00040)=          11
```

Figure 23 (Part 2 of 2). Debug Display

Assigning Names to User-Supplied Subprograms

Care should be used in naming user-supplied subprograms. When a subprogram is compiled, it is stored in the program pack library, which also contains the FORTRAN library. User-supplied subprograms having the same name as FORTRAN-supplied subprograms cause the FORTRAN subprograms to be overlaid.

If FORTRAN subprograms are overlaid, you can restore the library by reperforming program product generation, as described in the applicable system generation reference manual. Refer to *Related Publications* for the order number.

Wherever possible, assign unique names, that is, names that do not duplicate FORTRAN subprogram names. Figure 24 lists the names of FORTRAN-supplied functions and commercial subroutines. Before assigning names to subprograms, consult this list for possible duplicate names.

If it is necessary to give a subprogram the same name as a FORTRAN-supplied subprogram, include the OBJECT compiler option to store the subprogram object module into a different library.

The following is an example:

At compilation time:

```
// LOAD $FORT,F1
// FILE NAME-$WORK, etc.
// FILE NAME-$SOURCE, etc.
// RUN
*PROCESS OBJECT(T,LIB(R2))
subprogram SQRT
    .
    .
    .
/*
```

The compiler is called from F1. Ordinarily, object modules would be stored in the library on F1; however, the OBJECT compiler option on the *PROCESS statement directs the user-supplied subprogram SQRT to the library on R2.

At load module execution time, retrieve the module from the library by using the UPACK parameter on a linkage editor OPTIONS control statement.

At execution time:

```
// LOAD $OLINK,F1
// FILE NAME-$SOURCE, etc.
// FILE NAME-$WORK, etc.
// RUN
// INCLUDE NAME-MAIN,UNIT-F1
// OPTIONS UPACK-R2
// END
// LOAD MAIN,F1
// RUN
```

Assume that the main program (MAIN) has just been compiled, with the NOLINK and OBJECT(T,LIB[F1]) options. The linkage editor is called from F1. The INCLUDE statement specifies the module to be link edited (MAIN) is located on F1. The OPTIONS statement specifies that the user-supplied subprogram (SQRT) is located on R2.

For Model 12 and 15, the location of the user supplied subprogram (SQRT) may be specified by using the UPACK option on the *PROCESS statement when compiling the main program (MAIN).

Example:

```
// LOAD $FORT,F1
// FILE NAME-$WORK,etc.
// FILE NAME-$SOURCE,etc.
// RUN
*PROCESS UPACK-R2
   PROGRAM MAIN
   .
   .
   .
/*
```

The compiler is called from F1. Ordinarily, the object modules of subprograms needed by the main program (MAIN) would be stored in the library on F1; however, the UPACK option on the *PROCESS statement directs the linkage editor to search the library on R2 first, looking for the subprogram SQRT.

A1DEC	DUMP	LCOMP	READ	STACK
ADD	DUNPK		READ42	STAK42
ALOG	DVCHK	MOD	R2501	STAK60
ALOG10		MOVE	R2560	SUB
CFTOD	EDIT	MPY		S1403
COS	EXIT		SETINQ	S3284
	EXP	NCOMP	SET0	
DATSW		NSIGN	SET1	TANH
DCOS	FCTST	NZONE	SHIFT	TYPER
DECA1	FILL		SHIFTR	
DEXP		OVERFL	SIN	UNPAC
DIV	GET		SKIP	
DLOG		PACK	SLITE	WHOLE
DLOG10	IBTST	PDUMP	SLITET	
DMOD	ICOMP	PRINT	SPACE	
DPACK	INQCHK	PUNCH	SP1403	
DSIN	I2OR4	PUT	SP3284	
DSQRT		P1403	SQRT	
DTANH	KEYBD	P1442		
		P2560		
		P3284		

Figure 24. Names of IBM System/3 FORTRAN-Supplied Functions and Commercial Subroutines

OPTIMUM ASSIGNMENT OF $WORK AND $SOURCE WORK FILES

Proper assignment of the work files, $WORK and $SOURCE, can improve the performance of the compiler and linkage editor. For the Model 6 and Model 10, work files must be on the 5444. For the Model 12, work files must be in the 5444 simulation areas. Model 15 allows work files on either the 5444, 5445, or 3340.

The work files should be located in such a way that they minimize the number and distance of disk seeks in a particular configuration. The LOCATION parameter on the // FILE statement is used to fix the location of the work files.

Assignment of Work Files on One Disk

If only one disk is available, the work files should be located close to each other:

```
// FILE NAME-$WORK,UNIT-F1,LOCATION-180,TRACKS-20,PACK-xxxxxx,RETAIN-S
// FILE NAME-$SOURCE,UNIT-F1,LOCATION-200,TRACKS-20,PACK-xxxxxx,RETAIN-S
```

Assignment of Work Files on Two Disks

When two disks are available, the work files should be located on separate disks at the same LOCATION:

```
// FILE NAME-$WORK,UNIT-F1,LOCATION-190,TRACKS-20,PACK-xxxxxx,RETAIN-S
// FILE NAME-$SOURCE,UNIT-R1,LOCATION-190,TRACKS-20,PACK-xxxxxx,RETAIN-S
```

Model 15 Assignment of Work Files on 5445 or 3340 Disk Storage

On the Model 15 only, work files can be assigned to the 5445 or 3340 disk storage:

```
// FILE NAME-$WORK,UNIT-D1,TRACKS-20,PACK-xxxxxx,RETAIN-S,LOCATION-100/1
// FILE NAME-$SOURCE,UNIT-D1,TRACKS-20,PACK-xxxxxx,RETAIN-S,LOCATION-100/2
```

LINKAGE BETWEEN MODULES PRODUCED BY SYSTEM/3 LANGUAGE TRANSLATORS

This section describes standard linkage conventions for use between modules produced by the System/3 language translators: COBOL, FORTRAN, and Assembler. Programmers using standard linkage conventions are able to code routines in the language most appropriate to the function being performed. Figure 25 illustrates the standard described on the following pages.

```
*     SAMPLE SYSTEM/3 LINKAGE
*
*     ASSEMBLER MODULE (MODA) CALLS FORTRAN MODULE (MODB)
*
            EXTRN MODB
MODA        START X'0000'
            .
            .
            .
            B      MODB          CALL FORTRAN MODULE MODB
            DC     AL2(PLIST)    PARAMETER LIST
*     CONTROL RETURNS HERE AFTER MODULE MODB HAS BEEN EXECUTED
            .
            .
            .
*     PARAMETER LIST
PLIST       EQU    *
            DC     AL2(SAVA)     ADDRESS OF SAVE AREA
            DC     AL2(PARM1)    ADDRESS OF FIRST PARAMETER
            DC     AL2(PARM2)    ADDRESS OF SECOND PARAMETER
            .
            .
            .
            DC     XL1'00'       END OF PARAMETER LIST INDICATOR
*
*     PARAMETERS
PARM1       EQU    *
            DC     CL5'FIRST'
PARM2       EQU    *
            DC     CL6'SECOND'
            .
            .
            .
*
*     SAVE AREA
SAVA        DC     XL1'BO'       INDICATOR BYTE -- CALLING PROGRAM IS ASSEMBLER
            DC     CL6'MODA'     CALLING PROGRAM'S NAME
            END    MODA
```

Figure 25 (Part 1 of 2). Standard Linkage

```
*     SAMPLE SYSTEM/3 LINKAGE
*
*     FORTRAN MODULE (MODC) CALLS ASSEMBLER MODULE (MODD)
*
XR1          EQU    1
XR2          EQU    2
ARR          EQU    8
IAR          EQU    16
*
             ENTRY MODD
MODD         START X'0000'
             ST     SAVAR1,XR1               SAVE CONTENTS OF XR1
             LA     SAVA,XR1                 XR1 IS BASE FOR SAVE AREA
             USING SAVA,XR1
             ST     SAVAR2(,XR1),XR2         SAVE CONTENTS OF XR2
             ST     SAVART(,XR1),ARR         SAVE CONTENTS OF ARR
             L      SAVART(,XR1),XR2         XR2 POINTS TO ADDRESS OF PARM LIST
             L      1(,XR2),XR2              XR2 POINTS TO PARAMETER LIST
             ALC    SAVART(,XR1),TWO(,XR1)   SET RETURN POINT 2 PAST ARR
*    BODY OF ROUTINE
             .
             .
             .
*    RETURN TO CALLING PROGRAM
             L      SAVAR2(,XR1),XR2         RESTORE XR2
             L      SAVAR1(,XR1),XR1         RESTORE XR1
             L      SAVART,IAR               RETURN
*
*    SAVE AREA
SAVA         DC     XL1'30'        INDICATOR BYTE -- CALLED PROGRAM IS ASSEMBLER
             DC     CL6'MODD'      CALLED PROGRAM'S NAME
SAVAR1       DC     XL2'00'        CONTENTS OF XR1 ON ENTRY TO THIS ROUTINE
SAVAR2       DC     XL2'00'        CONTENTS OF XR2 ON ENTRY TO THIS ROUTINE
SAVART       DC     AL2(00)        RETURN POINT
*
TWO          DC     IL2'2'
             END
```

Figure 25 (Part 2 of 2). Standard Linkage

Standards

In order to be standard, linkage must be accomplished as follows:

1. Each module must have a *save area* defined as follows:

For a subprogram:

Byte 0	Bit 0	0	Not a main program
	Bits 1-3	000	FORTRAN
		001	COBOL
		011	Assembler
	Bits 4-7	0000	Reserved
Bytes 1-6		EBCDIC name, left justified	
Bytes 7-8		Value of index register 1 (XR1) at entry	
Bytes 9-10		Value of index register 2 (XR2) at entry	
Bytes 11-12		Return point in calling program	

For a main program:

Byte 0	Bit 0	1	Main program
	Bits 1-3	000	FORTRAN
		001	COBOL
		011	Assembler
	Bits 4-7	0000	Reserved
Bytes 1-6		EBCDIC name, left-justified	

Note: Main program refers to the program with the highest level of control.

2. Each module that calls another module must have one or more *parameter lists* defined as follows:

Bytes 0-1	Address of save area in this program
Bytes 2-3	Address of first parameter
Bytes (2n)-(2n+1)	Address of nth parameter
Byte (2n+2)	XL1'00' to indicate end of parameter list

Notes:

1. The first two bytes, as well as the end-of-parameter-list indicator (XL1'00') must be present in all parameter lists. If no parameters are to be passed, the the parameter list is only 3 bytes long. In this case, byte 3 will be 0 and the called program indicates a parameter list length of 2.
2. Addresses in parameter lists refer to the first byte (byte with the lowest address of the item.

3. When control reaches a program entry point, the address recall register (ARR) must point to a 2-byte field containing the first byte of the parameter list.

   The assembler language code to call a FORTRAN subprogram would normally be as follows:

```
         EXTRN    SUBR
         B        SUBR
         DC       AL2 (PARAMS)
RETNPT   EQU      *
```

   Note that the pointer to the parameter list points to the left byte of the save area address.

4. Normal return is accomplished by placing in the hardware instruction address register (IAR) a value that is two larger than the contents of the ARR when the program was entered.

5. Index registers 1 and 2 (XR1 and XR2) must be saved upon entry in the called program's save area, and restored at exit.

6. The address recall register need not be restored, but the return address must be determined and placed in the called program's save area.

CONSOLE DISPLAY PANEL DIAL SETTINGS

To aid him in debugging a compilation error, the IBM customer engineer can set the dial setting of the console display panel to the combination CEFE. For more efficient processing, be sure that this combination of characters is *not* present on the dial settings before beginning a compilation.

MODELS 10 AND 12 DUAL PROGRAMMING CONSIDERATIONS

FORTRAN programs can run in the dual programming environment (which allows two independent programs to be run concurrently) provided that the two programs do not share the same devices. For example, if the MFCU1 is used by the FORTRAN compiler, it cannot be used by the other program.

MODEL 15 CONSIDERATIONS

Model 15 Spooled Environment and Multiprogramming

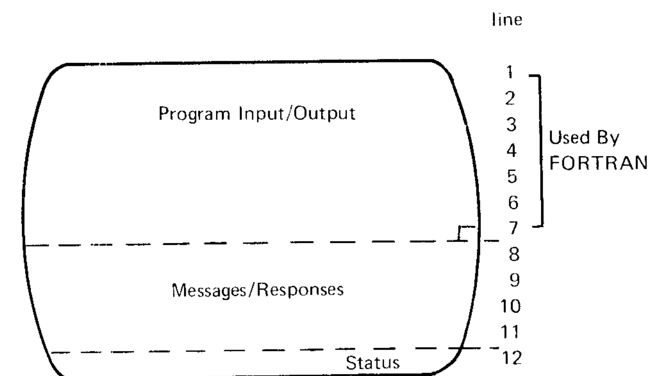
FORTRAN programs can be run in a Model 15 spooled environment or in the multiprogramming partitions. See *IBM System/3 Model 15 System Control Programming Reference Manual,* GC21-5077.

Model 15 CRT/Keyboard Support

The System/3 Model 15 CRT/keyboard is comprised of:

- An IBM 3277 Display Station Model 1—a cathode ray tube (CRT) screen.
- Feature 4632—a 78-key operator console keyboard.

Seven of the twelve 40-character lines are supported by FORTRAN for input and output operations. Because the last position is reserved for system use, a maximum of 279 positions are available to FORTRAN.



Two kinds of CRT/keyboard support are available:

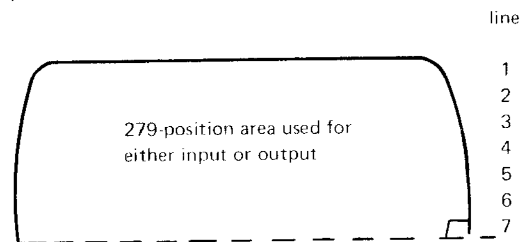
- Full screen support
- Split screen support

Full screen support is obtained by specifying 3277 on the FORTRAN READ and/or PRINT device option statements. Split screen support is obtained by specifying 3277S.

Logical unit numbers 5 and 6 are used for both split screen and full screen support. Split screen and full screen support cannot both be specified in the same program.

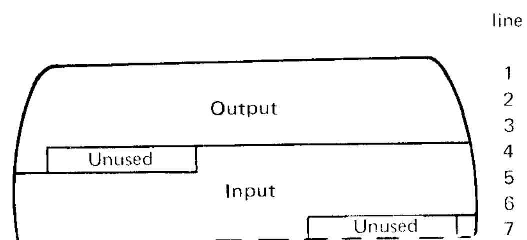
Full Screen Support

A WRITE statement first blanks all seven lines and then displays up to 279 characters. A READ statement allows the operator to key data into the 279-position area. Because the screen is not blanked by FORTRAN before a READ, a WRITE followed by a READ allows the operator to update the displayed record.



Split Screen Support

The first 125 positions are used for output, followed by 15 unused (blank) positions. The next 125 positions are used for input, followed by 14 unused (blank) positions.



The WRITE statement blanks the 125-position output area and then displays the 125 characters. A READ statement blanks the 125-position input area and then accesses data from this portion of the screen after data has been keyed and displayed on the screen.

Model 15 Double Buffering for Card Devices

Model 15 FORTRAN allows the user to double buffer the card I/O on the MFCU, MFCM, 1442, and 2501. The default of two buffers is allocated *unless* either of the following conditions exist:

1. The user specifies one buffer by adding an asterisk (*) to the device code (for example, MFCU1*).

2. Multiple operations are assigned to the same device (for example, READ, 1442; PUNCH, 1442).

The following table shows the number of buffers allocated to MFCU2 or MFCM2 for allowable combinations of operations. The asterisk after the read, punch, or print specification indicates a single buffer is requested.

Specification			Number of Buffers Allocated
Read*			1
Read			2
	Punch*		1
	Punch		2
Read	Punch		2
Read	Punch	Print	2
	Punch	Print	1
Read*	Punch*		1
Read*	Punch		2
Read	Punch*		1
	Punch*	Print	1
	Punch	Print*	1
	Punch*	Print*	1
Read*	Punch	Print	2
Read	Punch*	Print	1
Read	Punch	Print*	1
Read*	Punch*	Print	1
Read	Punch*	Print*	1
Read*	Punch	Print*	1
Read*	Punch*	Print*	1

Considerations/Restrictions

There might be a degradation in speed when the MFCU1 and the MFCU2 print are double buffered.

Model 15 (5704-FO2) 3741 Support

When using the 3741 with Model 15 FORTRAN (5704-FO2), the following considerations apply:

- The record length must be from 1 to 128 bytes.
- Each sector on the diskette contains only one logical record.
- All records in a file must have the same record length (fixed length records).
- Records are read from or written to the 3741 one at a time (unblocked).
- Double buffering is the default, single buffering may be specified.
- Records are read from or written to the 3741 sequentially.
- Reading and writing cannot be done in the same program using FORTRAN READ and WRITE statements. By using the FORTRAN Commercial Subroutines R3741 and P3741, reading and writing in the same program can be done.
- An end-of-file condition occurs for an input file when:
  - A /* statement is read.
  - A /& statement is read.
  - A /. statement is read.
  - EOD (end of data) is reached.

For a discussion on modes of operation see *IBM System/3 3741 Reference Manual*, GC21-5113.

DIFFERENCES BETWEEN 1130 AND SYSTEM/3

This section briefly summarizes some of the differences between the 1130 and System/3 which might affect FORTRAN processing.

Unit Numbers

A comparison of assignment of unit numbers to input/output devices is:

Unit Number	1130 Assignment	System/3 Models 10 and 12 Assignment	System/3 Model 6 Assignment	System/3 Model 15 Assignment
1	Console print	MFCU1 read only	5406 input	MFCU1 or MFCM1 read only
2	1442 read/punch	MFCU2 read/ print/punch	5496 input or output	MFCU2 or MFCM2 read/print/ punch
3	1132 print	5203 or 1403 print	5213 or 2222 print	1403 print
4				3284 print
5	1403 print	5471 input		Keyboard input
6	Keyboard input	5471 output		CRT output
7				
8				2501 read only
9	1442 punch	1442 read/ punch		1442 read/punch

Note: In System/3, unit numbers can be assigned to only one device; for example, if number 5 is specified for the 5471 in a program, it cannot also be assigned to a direct-access device in that program. If 5471 input is not specified, unit number 5 can be assigned to a direct-access device.

Device Options

In System/3, input/output devices are specified by device option statements, which correspond to the 1130 *IOCS statement. For example:

1130 Usage	System/3 Usage
*IOCS (1132 PRINTER, TYPEWRITER, 1403 PRINTER)	// PRINT DEVICE-'1403,5471'
*IOCS (CARD, 2501 READ,KEYBOARD)	// READ DEVICE-'MFCU1,MFCU2,1442,5471'
*IOCS (1442,PUNCH,CARD)	// PUNCH DEVICE-'MFCU2,1442'
*IOCS (DISK)	// DAD44 UNITNO-'n1,n2, . . . '
	// DAD45 UNITNO-'n1,n2, . . . '

Specifying the BCD Option

Any 1130 FORTRAN card decks that were punched using the BCD card punch (the 026 keypunch) can be read by the System/3 compiler by specifying the BCD option on the *PROCESS compiler option statement. (*PROCESS is described in Chapter 13, *Compilation.*)

Read/Punch on the Same Card

In 1130 FORTRAN the same card can be read and punched using the 1442 card read punch. System/3 FORTRAN does not permit input/output operations to be performed on the same card except for certain commercial subroutines. The MFCU2 and 1442 can be used as either, but not both, an input or output device in the same program.

Call Link

In System/3, main programs can call each other using the INVOKE statement, which corresponds to the 1130 CALL LINK statement. The PROGRAM statement must be specified as the first statement of each main program to be invoked. Further, all common blocks sharing data between main programs must be changed to GLOBAL statements. For example:

1130 Usage	System/3 Usage
COMMON A,B(10),J	PROGRAM MAIN
.	GLOBAL A,B(10),J
.	.
.	.
CALL LINK (MAIN2)	INVOKE MAIN2
	PROGRAM MAIN2
	GLOBAL C,D(10),J
	.
	.
	.

If any of the programs being invoked require more storage than the invoking program, a CORE compiler option statement should be specified explicitly stating the largest amount of storage required. (Additional information on using the CORE compiler option statement with PROGRAM and INVOKE statements can be found in Chapter 19, *FORTRAN Implementation Considerations.*)

Associated Variables in Subroutines

In System/3, if an associated variable is passed as an argument to a subroutine, it is not automatically updated when input/output operations are performed in the subroutine. To ensure that the variable is updated, pass it through COMMON. For example:

1130 Usage	System/3 Usage
	COMMON IJ
DEFINE FILE 8(r,s,f,IJ)	DEFINE FILE 8(r,s,f,IJ)
IJ=25	IJ=25
CALL WRTR(IJ)	CALL WRTR
.	.
.	.
.	.
END	END
SUBROUTINE WRTR(IVAR)	SUBROUTINE WRTR
.	COMMON IVAR
.	.
.	.
WRITE (8'IVAR)	WRITE (8'IVAR)
	.
	.
	.

Library Routines

In System/3, if calls are made to the math library (for example, SIN, SQRT), the GENERIC statements should be included to ensure that the proper library module is loaded for the argument type.

For library routines using integer arguments (for example, MIN, MAX, FLOAT), integer arguments must be typed as INTEGER*4.

Passing Arrays

In the 1130, arrays not used in a subroutine need not be dimensioned. In System/3, arrays must be dimensioned in all subroutines in which they are passed, even though they are not used in a particular subroutine. For example:

1130 Usage	System/3 Usage
DIMENSION I(20)	DIMENSION I(20)
.	.
.	.
.	.
CALL SUB1 (I)	CALL SUB1 (I)
.	.
.	.
.	.
SUBROUTINE SUB1(J)	SUBROUTINE SUB1(J)
.	DIMENSION J(20)
.	.
.	.
CALL SUB2(J)	CALL SUB2(J)
(J not used in SUB1)	(J not used in SUB1)
SUBROUTINE SUB2(K)	SUBROUTINE SUB2(K)
DIMENSION K(20)	DIMENSION K(20)
.	.
.	.
.	.

When arrays are passed as arguments to subroutines, System/3 passes the address of the array in the calling programs data area.

Length Specification of Variables

In System/3, INTEGER*2 and REAL*8 variables can be specified using the IMPLICIT statement, which corresponds to the 1130 *ONE WORD INTEGERS and *EXTENDED PRECISION statements. For example:

1130 Usage	System/3 Usage
*ONE WORD INTEGERS	IMPLICIT INTEGER*2 (I-N)
*EXTENDED PRECISION	IMPLICIT REAL*8(A-H,O-$)

System/3 FORTRAN requires some variables to be INTEGER*4, such as: variables and arguments to INTEGER intrinsic functions, arguments to commercial subroutines, and logical unit numbers.

Use of COMMON, EQUIVALENCE, and DEFINE FILE

In System/3, the relative size of double precision variables to single precision variables (REAL*8 to REAL*4) is 2:1. In 1130 the relative size of EXTENDED PRECISION variables to single precision variables is 3:2. Thus, it may be necessary to change COMMON and EQUIVALENCE statements and to increase the size of records in DEFINE FILE statements.

Rounding

1130 FORTRAN does not round when converting data items on input; System/3 FORTRAN does. Thus, if any FORTRAN rounding code is present in 1130 FORTRAN programs, remove it before converting to System/3.

Passing Scalar Arguments to Subroutines

When scalar arguments are passed to a subroutine, System/3 copies the arguments into the subroutine's data area, uses them in the subroutine, then copies the values back to the calling program when the subroutine returns control. With the 1130, the values in the calling program are used directly; they are neither copied to the subroutine nor copied back to the calling program.

Consider the following code:

```
J=1              SUBROUTINE SUM(L,M,N)
CALL SUM(J,J,5)  L=M+N
                 RETURN
                 END
```

Before control is returned to the calling program, L has the value 6, M the value 1, and N the value 5. However, under System/3, when control returns, the value of J is still 1 because the variable M (value 1) is copied back to J after the variable L.

To avoid this problem, before converting an 1130 program, recode the parameter list in a CALL statement so that any value being tested is not specified more than once, for example, CALL SUM(K,J,5).

Forms Control

The 1130 has a carriage control tape to sense channel 12 or the overflow line and channel 1, which is the beginning of a page. This provides for a limited amount of page formatting. Because System/3 has no carriage control tape, all forms control must be incorporated into the program by the programmer.

The 1130 also can print a program name on each page of a program listing by the use of an ** card. This is an invalid card for System/3.

Commercial Subroutines

The System/3 FORTRAN Commercial Subroutines Package includes three 1442 card read punch routines. P1442 is used for card punching, READ42 for card reading, and STAK42 for selecting alternate stackers. The corresponding 1130 routines were PUNCH or P1442, READ, and STACK. The System/3 commercial subroutines support allows the 1442 to be used as a combined file.

Decimal Data Format

In System/3, D1 format corresponds to the standard System/360 zoned decimal format: one digit per eight-bit byte, or two digits per INTEGER*2 array element. The digit is carried in the low-order four bits of the byte, with the high-order four bits set to 1's (X'F'). However, the high-order four bits of the low-order byte are used to carry the sign of the number: X'C' (binary 1100) or X'F' (binary 1111) for positive; X'D' (binary 1101) for negative.

In 1130, D1 format consists of one digit per word, right justified. The decimal field is stored in an array, one digit per element. The sign of the digit is carried with the rightmost digit. If the number is negative, a negative one (–1) is added to the rightmost digit. This must be done because the 1130 cannot represent a negative zero.

A1 Data Format

The A1 format is the same in System/3 and 1130, except for the sign of a numeric field. In System/3, the sign of a numeric field in A1 format is assumed to be carried as the zoned portion of the rightmost character: X'C' (binary 1100) or X'F' (binary 1111) for positive; X'D' (binary 1101) or X'60' (minus) over the units position for negative. A negative zero is represented by a X'D0' or X'60' (minus).

In 1130, the sign of a numeric field in A1 format is a multiple punch over the rightmost character:

11 punch for negative and 12 punch for positive.
A negative zero is represented by a X'60' (minus).

Negative Zero

In System/3, a negative zero is represented by an X'D0'. In 1130, a negative zero is represented by an X'60'. Commercial subroutines require a minus (-) over the units position for a negative value. If the units position is a zero, the 1130 cannot recognize the value. Therefore, the circumvention of punching a minus (X'60') in the units position instead of the zero was implemented for 1130.

Number of Record Fields in the DEFINE FILE Statement

System/3 does not use this number as a check for accessing records outside the range of the DEFINE FILE statement. The range or size of the file is set by the FILE statement TRACKS and RECORDS parameters.

Part 4. Reference

This section contains:

- FORTRAN statement reference
- System/3 FORTRAN intrinsic and external library functions
- FORTRAN service subprograms

This section lists the FORTRAN statements in alphabetical order and summarizes each of them using the following format:

- Statement name
- General form of the statement
- Examples of statement use

Arithmetic Assignment Statement

General Form: a=b

where

a is a variable or array element

b is a variable, array element, or arithmetic expression

Examples:

```
A=B         A=A+B
A=B(1)      A=SIN(A**I)
A=B(I)      A(1)=FUNC(B,C,D)
A=I         I=A
A=6         I=I+6
A=6.4
```

Arithmetic IF Statement

General Form: IF (a)n_1,n_2,n_3

where

a is an arithmetic expression

n_1,n_2, and n_3 are statement numbers of executable statements in the program unit containing the IF statement.

Examples:

```
      IF (A-B)10,4,30
40    D=C**2
      .
      .
      .
4     D=B+C
      .
      .
      .
30    C=D**2
      .
      .
      .
10    E=(F*B)/D+1
```

The IF statement compares the value of the difference between A and B. If the value is negative, a branch is made to statement number 10; if zero, a branch is made to statement number 4; and if positive, a branch is made to statement number 30.

AT Statement

General Form: AT n

where
n is an executable statement number in the program or subprogram to be debugged.

Examples:

```
200   X=Y+Z
        .
        .
        .
      DEBUG TRACE
      AT 200
      TRACE ON
        .
        .
        .
      END
```

BACKSPACE Statement

General Form: BACKSPACE i

where
i is an unsigned integer constant or INTEGER*4 variable that is the logical unit number of a sequential file located on a magnetic tape or disk unit.

Examples:

```
BACKSPACE 10
BACKSPACE L
```

CALL Statement

General Form: CALL name(a_1,a_2,a_3, . . . ,a_n)

where

name is the name of a SUBROUTINE subprogram.

a is an actual argument that is being supplied to the SUBROUTINE subprogram. It can be a variable, array name, array element, arithmetic expression, or subprogram name.

Examples:

```
CALL OUT
CALL MATMPY(X,5,40,Y,7,Z)
CALL SUB1(X+Y*5,ABDF,SINE)
CALL SUB(P,Q,R,I)
```

COMMON Statement

General Form: COMMON a_1(k_1),a_2(k_2), . . . ,a_n(k_n)

where

a is a variable or array name that is not a dummy argument.

k is optional and is composed of from one to three unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array.

Examples:

Calling Program

```
COMMON A,B,C,R(100)
REAL A,B,C
INTEGER R
     .
     .
     .
CALL MAPMY
     .
     .
     .
```

Subprogram

```
SUBROUTINE MAPMY

COMMON X,Y,Z,S(100)
REAL X,Y,Z
INTEGER S
     .
     .
     .
```

Computed GO TO Statement

General Form: GO TO $(n_1, n_2, \ldots, n_n)$, i

where

n is the number of an executable statement in the program unit containing the GO TO statement.

i is an integer variable whose value is in the range $1 \leqslant i \leqslant n$.

Examples:

```
      GO TO (25,10,7,10),ITEM
 3    C=7.02
       .
       .
       .
 7    C=E**2+A
       .
       .
       .
 25   L=C
       .
       .
       .
 10   B=21.3E02
```

A branch is made to a statement number based on the value of the integer variable, ITEM. If ITEM has a value of 1, the branch is to the first statement enclosed in parentheses in the GO TO statement (statement number 25); if ITEM has a value of 2, the branch is to the second statement (statement number 10), etc.

CONTINUE Statement

General Form: CONTINUE

Examples:

```
      DO 30 I=1,20
       .
       .
       .
      GO TO 7
 30   CONTINUE
```

DATA Statement

General Form: DATA $k_1/i_1*d_1/, k_2/i_2*d_2/, \ldots, k_n/i_n*d_n/$

where

k is a list containing variables, array elements (in which case the subscript quantities must be unsigned integer constants), or array names.

d is a list of constants, (integer, real, hexadecimal or literal).

i* is optional and is an unsigned integer constant appearing before d, indicating that d is to be specified i times.

Examples:

```
DIMENSION D(5,10)
DATA A,B,C/5.0,6.1,7.3/,D,E/25*1.0,25*2.0,5.1/

DIMENSION A(5),B(3,3)
DATA A/5*1.0/,B/9*2.0/,C/'FOUR'/
```

DEBUG Statement

General Form: DEBUG option, . . . ,option

where

option may be one of the following, specified in any order:

SUBCHK$(n_1, n_2, \ldots, n_n)$
where n is an array name and is optional.

TRACE

INIT$(m_1, m_2, \ldots, m_n)$

where m is the name of a variable or an array and is optional.

SUBTRACE

Examples:

```
PROGRAM FIRST
  .
  .
  .
DEBUG SUBCHK(SUB),TRACE,INIT(SUM,MULT),
SUBTRACE
```

DEFINE FILE Statement

General Form: DEFINE FILE u_1 (r_1,s_1,f_1,v_1),u_2 (r_2,s_2,f_2,v_2), . . . u_n (r_n,s_n,f_n,v_n)

where

u is an unsigned integer constant that is the logical unit number.

r is an unsigned integer constant that specifies the number of records in the file associated with u.

s is an unsigned integer constant that specifies the maximum size (in characters, bytes, or words) of each record associated with u.

f specifies whether data is to be read or written with or without format control. The f code may be one of the characters L, to indicate that a file may be processed (read or written) either with or without format control (s is interpreted to specify bytes); E, to indicate that a file is processed with format control (s specifies bytes); or U, to indicate that a file is processed without format control (s is interpreted to specify words).

v is an integer variable called the associated variable.

Examples:

```
DEFINE FILE 8(50,100,L,I2),9(100,50,L,J3)
```

DIMENSION Statement

General Form: DIMENSION a_1 (k_1),a_2 (k_2), . . . ,$a_n(k_n)$

where

a is an array name.

k is composed of from one to three unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array.

Examples:

```
DIMENSION A(10),ARRAY(5,5,5),LIST(10,100)
DIMENSION B(25,25),TABLE(5,10,15)
```

Direct-Access READ/WRITE Statement

See *READ Statement (Direct-Access)*.

DO Statement

General Form: DO n i = m_1,m_2,m_3

where

n is the statement number of an executable statement appearing after the DO statement and in the same program unit.

i is a non-subscripted integer variable called the DO variable.

m_1,m_2, and m_3 are either unsigned integer constants greater than 0 or non-subscripted integer variables. The value of m_1 should not exceed that of m_2. The value of m_2 cannot exceed $2^{31}-2$. m_3 is optional and, if omitted, is assumed to be 1. If m_3 is omitted, the preceding comma must also be omitted.

The statements that physically follow the DO statement up to and including the statement numbered n are called the *range* of the DO. The value of m_1 is called the *initial value.* The value of m_2 is called the *test value.* The value of m_3 is called the *increment.*

Example 1:

```
   .
   .
   .
   K=0
   L=10
   DO 5 JOB=1,L,2
   K=K+1
5  M(JOB)=N(JOB)-K*JOB
   .
   .
   .
```

Example 2:

```
   .
   .
   .
   15 DO 25 J=1,1000
   25 INV(J)=INV(J)-IOUT(J)
   35
```

END Statement

General Form: END

Examples:

```
PROGRAM FIRST     SUBROUTINE SECOND
     .                  .
     .                  .
     .                  .
CALL SECOND       RETURN
     .            END
     .
     .
STOP
END
```

END FILE Statement

General Form: END FILE i

where
i is an unsigned integer constant or INTEGER*4 variable that is the logical unit number of a sequential file on a magnetic tape or disk unit.

Examples:

```
END FILE 10
END FILE L
```

EQUIVALENCE Statement

General Form: EQUIVALENCE (a_{11},a_{12},a_{13}, . . .), (a_{21},a_{22},a_{23}, . . .), . . .

where
a is a variable or array element.

Examples:

```
DIMENSION C(100,100),A(50,50),B(100)
EQUIVALENCE (C(1),A(1)),(C(2501),B(1))

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

Explicit Specification Statement

See *INTEGER Statement/REAL Statement*.

EXTERNAL Statement

General Form: EXTERNAL a_1,a_2,a_3, . . . a_n

where
a is the name of a subprogram that is passed as an argument to other subprograms.

Examples:

```
EXTERNAL MULT
  .
  .
  .
CALL SUB(J,MULT,C)
```

FIND Statement

General Form: FIND (u'r)

where

u is an unsigned integer constant or INTEGER*4 variable that represents a logical unit number of a direct-access file.

r is an integer expression that represents the relative position of a record within the file associated with u.

Examples:

```
      DEFINE FILE 8(1000,80,L,IVAR)
10    FIND(8'50)
        .
        .
        .
15    READ(8'50)A,B
```

FORMAT Statement

General Form: xxxxx FORMAT ($c_1 s_1 c_2 s_2 \ldots c_n$)

where

xxxxx is a one-to-five digit statement number.

c is a format code that describes integer data (code I), real data (codes D, E, F), character data (A), literal data (H, or data enclosed in apostrophes), fields to be skipped (X), or a position in a FORTRAN record where data transfer is to begin (T), and the number of characters in a field.

s is a separator, which may be either a comma or any number of slashes. Slashes indicate the beginning of a new record.

Examples:

```
   10 FORMAT (E10.5,D16.10,I7,E7.2,D12.7)
    2 FORMAT (3F9.2,2D15.10/8E10.5)
30000 FORMAT (' THE FOLLOWING IS A LIST OF
        PRIMES'/(I5))
```

Function Definition Statement

General Form: name($a_1, a_2, a_3, \ldots, a_n$)=expression

where

name is the statement function name.

a is a dummy argument.

expression is any arithmetic or relational expression that does not contain array elements.

Examples:

```
FUNC(A,B)=3.*A+B**2.+X+Y+Z
SUM(A,B,C,D)=A+B+C+D
```

FUNCTION Statement

General Form: type FUNCTION name*s($a_1, a_2, a_3, \ldots a_n$)

where

type is optional and can be INTEGER or REAL

name is the name of the FUNCTION

s is optional when type is specified and represents one of the length specifications for its associated type (2 or 4 for INTEGER, 4 or 8 for REAL).

a is a dummy argument.

Examples:

```
FUNCTION CALC(A,B,J)
INTEGER FUNCTION CALC*2(I,J,K)
```

GENERIC Statement

General Form: GENERIC

Examples:

```
GENERIC
REAL*8 A,B,C,D
C=COS(A)
D=DCOS(B)
```

GLOBAL Statement

General Form: GLOBAL $a_1(k_1), a_2(k_2), \ldots, a_n(k_n)$

where

a is the name of a variable or an array.

k is optional, and is a subscript composed of one through three unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array.

Examples:

```
PROGRAM FIRST          PROGRAM SECOND
GLOBAL A,B,C(5,5),     GLOBAL,X,Y,Z(25),
D(10,100)              DATA(10,100)
```

GO TO Statement

(See also *Computed GO TO Statement.*)

General Form: GO TO n

where
n is the number of an executable statement in the program unit containing the GO TO statement.

Examples:

```
        .
        .
        .
        GO TO 6
  12    X=Y-Z
  6     A=2.*B
```

IF Statement

See *Arithmetic IF Statement/Relational IF Statement.*

IMPLICIT Statement

General Form: IMPLICIT type*s_1 (a_{11},a_{12}, . . .), . . . , type*s_n (a_{n1},a_{n2}, . . .)

where

type is either INTEGER or REAL.

s is optional and represents one of the length specifications for its associated type (2 or 4 for INTEGER, 4 or 8 for REAL).

a is a single alphabetic character or a range of characters in the set A,B, . . . ,Z,$ in that order. The range is denoted by the first and last characters of the desired range separated by a minus sign (for example (A-D)).

Examples:

```
IMPLICIT INTEGER*2(A-H),REAL*8(I-K)
IMPLICIT REAL(A-H,O-Z),INTEGER(I-N)
```

INTEGER Statement

General Form: INTEGER*s a_1 (k_1),a_2 (k_2), . . . ,a_n(k_n)

where

*s is optional and represents one of the INTEGER length specifications, 2 or 4.

a is a variable, array, or function name.

k is optional and gives dimension information for arrays. Each k is composed of one through three unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array.

Examples:

```
INTEGER*2 ITEM,VALUE
INTEGER*4 LIST,VAL2
INTEGER B(100)
```

INVOKE Statement

General Form: INVOKE name

where
name is the name of a main program specified in a PROGRAM statement.

Examples:

```
PROGRAM FIRST          PROGRAM SECOND
      .                      .
      .                      .
      .                      .
INVOKE SECOND
```

PAUSE Statement

General Form: PAUSE n

where
n is an optional integer constant that is printed with the PAUSE statement for identification.

Examples:

```
PAUSE
PAUSE 50
PAUSE 00002
```

PROGRAM Statement

General Form: PROGRAM name

where
name is the name assigned to the main program.

Examples:

```
PROGRAM SECOND
    .
    .
    .
```

READ Statement (Direct-Access)

General Form: READ(u'r,f,ERR=s) list

where

u is an unsigned integer constant or INTEGER*4 variable that represents a logical unit number of a direct-access file.

r is an integer expression that represents the relative position of a record within the file associated with u.

f is optional and, if specified, is the statement number of a FORMAT statement that describes the data being read.

ERR=s is optional and specifies the statement number (s) in the same program unit to which to transfer control if an error occurs during data transfer.

list is an I/O list.

Examples:

```
DEFINE FILE 8(500,100,L,ID1),9(100,28,L,ID2)
9  READ (8'16,10)(M(K),K=1,10)
    .
    .
    .
13 READ(9'ID2+5)A,B,C,D,E,F,G
```

READ Statement (Sequential)

General Form: READ (u,f,END=s,ERR=t) list

where

u is an unsigned integer constant or INTEGER*4 variable that represents the logical unit number of the device to be read from.

f is optional and can be the statement number of a FORMAT statement or an asterisk (*).

END=s is optional and specifies the statement number (s) in the same program unit to which to transfer control if an end of file condition is encountered.

ERR=t is optional and specifies the statement number (t) in the same program unit to which to transfer control if an error occurs during data transfer. ERR is ignored if the file is not a disk or tape file.

list is an I/O list (optional if f is specified).

Examples:

```
READ (1,98)A,B,(C(I,K),I=1,10)
READ (J)A,B,C
READ (I,*,END=200) (ARRAY(I),I=1,25),B(2),C(6))
```

REAL Statement

General Form: REAL*s $a_1(k_1),a_2(k_2), \ldots ,a_n(k_n)$

where

*s is optional and represents one of the REAL length specifications, 4 or 8.

a is a variable, array, or function name.

k is optional and gives dimension information for arrays. Each k is composed of one through three unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array.

Examples:

```
REAL ITEM(5,5),B(100)
REAL*8 MULT
REAL*4 JMULT
```

Relational IF Statement

General Form: IF (a)s

where

a is a relational expression.

s is an executable statement except a DO statement or another relational IF statement.

A relational expression is formed by combining two arithmetic expressions with one of the six relational operators: .GT., .LT., .EQ., .NE., .GE., or .LE.; which stand for greater than, less than, equal to, not equal to, greater than or equal to, and less than or equal to, respectively. The periods must precede and follow the relational operators, as shown.

Examples:

The two following IF statements have the same effect:

```
      IF (A.LT.B)A=B
  200 .
      .
      .

      IF (A-B)100,200,200
  100 A=B
  200 .
      .
      .
```

RETURN Statement

General Form: RETURN

Examples:

```
SUBROUTINE COPY (A,B,N)
      .
      .
      .
RETURN
END

FUNCTION CALC (A,B,J)
      .
      .
      .
RETURN
END
```

REWIND Statement

General Form: REWIND i

where

i is an unsigned integer constant or INTEGER*4 variable, that is the logical unit number of a sequential file on a magnetic tape or disk unit.

Examples:

```
REWIND 10
REWIND L
```

Sequential READ/WRITE Statements

See *READ Statement (Sequential)/WRITE Statement (Sequential).*

STOP Statement

General Form: STOP n

where

n is an optional integer constant that is printed with the STOP statement for identification.

Examples:

```
STOP
STOP 25
```

SUBROUTINE Statement

General Form: SUBROUTINE name(a_1,a_2,a_3, . . . a_n)

where

name is the SUBROUTINE name.

a is a dummy argument that can be a variable name, array name, or the dummy name of another SUBROUTINE or FUNCTION subprogram.

Examples:

```
SUBROUTINE COPY(A,B,N)
SUBROUTINE NULL
```

TRACE OFF Statement

General Form: TRACE OFF

Examples:

```
200  X=Y+2
       .
       .
       .
210  X=Y-2
       .
       .
       .
     DEBUG TRACE
     AT 200
     TRACE ON
       .
       .
       .
     AT 210
     TRACE OFF
     END
```

TRACE ON Statement

General Form: TRACE ON

Examples:

```
200  X=Y+2
       .
       .
       .
     DEBUG TRACE
     AT 200
     TRACE ON
       .
       .
       .
     END
```

WRITE Statement (Direct-Access)

General Form: WRITE (u'r,f) list

where

u is an unsigned integer constant or INTEGER*4 variable that is the logical unit number of a direct-access file

r is an integer expression that represents the relative position of a record within the file associated with u.

f is optional and, if specified, is the statement number of the FORMAT statement that describes the data being written.

list is an I/O list (optional if f is specified).

Examples:

```
      DEFINE FILE 8(500,100,L,ID1),9(100,28,L,ID2)
        .
        .
        .
8     WRITE (8'16,10)(M(K),K=1,10)
        .
        .
        .
11    WRITE (9'ID2+5)A,B,C,D,E,F,G
```

WRITE Statement (Sequential)

General Form: WRITE (u,f) list

where

u is an unsigned integer constant or INTEGER*4 variable that is the logical unit number of a device to be written to.

f is optional and can be the statement number of a FORMAT statement or an asterisk (*) for list-directed I/O.

list is an I/O list (optional if f is specified).

Examples:

```
WRITE (3,75) A,(B,(I,3),I=1,10,2),C
WRITE (4) ARRAY
WRITE (3,*) I,N(I)
```

Many commonly used mathematical functions or calculations are provided by the System/3 FORTRAN IV language. Mathematical functions are called in two ways: *explicitly,* when you include the function name in a source language statement; and *implicitly,* when a certain notation (such as converting a real number to an integer number) appears within a source language statement. Explicitly called functions are known as *external* functions, because the programmer calls them. Implicitly called functions are *intrinsic,* because the compiler generates the calls to them.

To a programmer using the System/3 FORTRAN IV mathematical functions, it is of no importance whether a specific function is intrinsic or external unless you wish to detach the function name by specifying external.

System/3 FORTRAN IV intrinsic functions include routines for determining maximum, minimum, and absolute values, converting from real to integer and from integer to real, and increasing or decreasing the precision of a number. External functions include sine, cosine, logarithm, and square root routines.

Figure 27 summarizes the intrinsic functions; Figure 28, the external functions. Figure 29 lists the accuracy of the external functions.

The following discussion describes the algorithms used in the mathematical subroutines.

ALGORITHMS

This section contains information about the method used to compute each function. The information for explicitly called subprograms is arranged alphabetically according to the specific function of each subprogram (that is, exponentiation, logarithmic, etc).

Information for the implicity called subprograms is arranged alphabetically according to function, and alphabetically by entry name within that function.

The presentation of each algorithm is divided into its major computational steps; the formulas necessary for each step are supplied. For the sake of brevity, the needed constants are normally given only symbolically. (The actual values can be found in the assembly listing of the subprograms.) Some of the formulas are widely known; those that are not so widely known are derived from more common formulas. The process leading from the common formula to the computational formula is sketched in enough detail so that the derivation can be reconstructed by anyone who has an understanding of college mathematics and access to the common texts on numerical analysis. [1] All approximations were derived by the so-called "minimax" methods. The approximation sought by these methods can be characterized as follows. Given a function $f(x)$, an interval I, the form of the approximation (such as the rational form with specified degrees), and the type of error to be minimized (such as the relative error), there is normally a unique approximation to $f(x)$ whose maximum error over I is the smallest among all possible approximations of the given form. Details of the theory and the various methods of deriving such approximation are provided in the reference. [1] The accuracy figures cited take round-off errors into account. Minor programming techniques used to minimize round-off errors are not described here.

[1] Any of modern numerical analysis texts can be used as a reference. Such texts are A. Ralston's *A First Course in Numerical Analysis* (McGraw-Hill Book Company, Inc., New York, 1965), and C. T. Fike's *Computer Evaluation of Mathematical Functions* (Prentice-Hall, Inc., Englewood Cliffs, New Jersey).

General Function	Entry Name	Definition	Arguments			Function Value Returned	
			No.	Type	Range (1)	Type	Range (1)
Absolute value	IABS	$y = \|x\|$	1	INTEGER *4	Any INTEGER argument	INTEGER *4	
	ABS		1	REAL *4	Any REAL argument	REAL *4	
	DABS		1	REAL *8		REAL *8	
Maximum and minimum values	MAX(2)	$y = \max(x_1, \ldots, x_n)$	≥2	INTEGER *4	Any INTEGER argument	INTEGER *4	
	MAX0		≥2	INTEGER *4		INTEGER *4	
	AMAX0		≥2	INTEGER *4		REAL *4	
	MAX1		≥2	REAL *4	Any REAL argument	INTEGER *4	
	AMAX1		≥2	REAL *4		REAL *4	
	DMAX1		≥2	REAL *8		REAL *8	
	MIN(2)	$y = \min(x_1, \ldots, x_n)$	≥2	INTEGER *4	Any INTEGER argument	INTEGER *4	
	MIN0		≥2	INTEGER *4		INTEGER *4	
	AMIN0		≥2	INTEGER *4		REAL *4	
	MIN1		≥2	REAL *4	Any REAL argument	INTEGER *4	
	AMIN1		≥2	REAL *4		REAL *4	
	DMIN1		≥2	REAL *8		REAL *8	
Truncation	AINT	$y = (\text{sign } x) \cdot n$ where n is the largest integer $\leq \|x\|$	1	REAL *4	Any	REAL *4	
	INT		1	REAL *4		INTEGER *4	
	IDINT		1	REAL *8		INTEGER *4	
Modulo arithmetic	MOD	y = remainder $\left(\frac{x_1}{x_2}\right)$, that is, $y = x_1 \text{ (modulo } x_2)$	2	INTEGER *4	$x_2 \neq 0$	INTEGER *4	
	AMOD		2	REAL *4		REAL *4	
	DMOD		2	REAL *8		REAL *8	
Float	FLOAT	Convert from INTEGER to REAL	1	INTEGER *4	Any INTEGER argument	REAL *4	
	DFLOAT		1	INTEGER *4		REAL *8	
Fix	IFIX	Convert from REAL to INTEGER	1	REAL *4	Any REAL argument	INTEGER *4	
Transfer of sign	ISIGN	$y = (\text{sign } x_2) \cdot x_1$ $x_1 \neq 0$	2	INTEGER *4	Any INTEGER argument	INTEGER *4	
	SIGN		2	REAL *4	Any REAL argument	REAL *4	
	DSIGN		2	REAL *8		REAL *8	
Positive difference	IDIM	$y = x_1 - \min(x_1, x_2)$	2	INTEGER *4	Any INTEGER argument	INTEGER *4	
	DIM		2	REAL *4	Any REAL argument	REAL *4	
Obtain most significant part of a REAL argument	SNGL		1	REAL *8	Any REAL argument	REAL *4	
Precision increase	DBLE		1	REAL *4	Any REAL argument	REAL *8	

(1) $\gamma = 16^{63} \cdot (1\text{-}16^{-6})$ for single precision and $16^{63} \cdot (1\text{-}16^{-14})$ for double precision.
(2) Alias. This name can be used in place of the REAL * 4 or INTEGER * 4 function name.

Figure 27. Intrinsic Mathematical Functions

General Function	Name	Definition	Arguments			Function Value Returned	
			No.	Type	Range ①	Type	Range ①
Natural and common logarithm	LOG ②	$y = \log_e x$ or $y = \ln x$	1	REAL *4	$x > 0$	REAL *4	$-180.218 \leq y \leq 174.673$
	ALOG		1	REAL *4		REAL *4	
	DLOG		1	REAL *8		REAL *8	
	LOG10 ②	$y = \log_{10} x$	1	REAL *4	$x > 0$	REAL *4	$-78.268 \leq y \leq 75.859$
	ALOG10		1	REAL *4		REAL *4	
	DLOG10		1	REAL *8		REAL *8	
Exponential	EXP	$y = e^x$	1	REAL *4	$-180.218 \leq x \leq 174.673$	REAL *4	$0 \leq y \leq \gamma$
	DEXP		1	REAL *8		REAL *8	
Square root	SQRT	$y = \sqrt{x}$ or $y = x^{1/2}$	1	REAL *4	$x \geq 0$	REAL *4	$0 \leq y \leq \gamma^{1/2}$
	DSQRT		1	REAL *8		REAL *8	
Arctangent	ATAN	$y = \arctan x$	1	REAL *4	Any REAL Argument	REAL *4 (in radians)	$-\frac{\pi}{2} \leq y \leq \frac{\pi}{2}$
	DATAN		1	REAL *8		REAL *8 (in radians)	
Sine and cosine	SIN	$y = \sin x$	1	REAL *4 (in radians)	$\lvert x \rvert < (2^{18} \cdot \pi)$	REAL *4	$-1 \leq y \leq 1$
	DSIN		1	REAL *8 (in radians)	$\lvert x \rvert < (2^{50} \cdot \pi)$	REAL *8	
	COS	$y = \cos x$	1	REAL *4 (in radians)	$\lvert x \rvert < (2^{18} \cdot \pi)$	REAL *4	$-1 \leq y \leq 1$
	DCOS		1	REAL *8 (in radians)	$\lvert x \rvert < (2^{50} \cdot \pi)$	REAL *8	
Hyperbolic tangent	TANH	$y = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$	1	REAL *4	Any REAL argument	REAL *4	$-1 \leq y \leq 1$
	DTANH		1	REAL *8		REAL *8	

Notes:

① $\gamma = 16^{63} \cdot (1-16^{-6})$ for single precision and $16^{63} \cdot (1-16^{-14})$ for double precision.

② Alias. This name may be used in place of the REAL *4 function name.

Figure 28. External Mathematical Functions

Function Name	Sample Type ①	Argument Range	Error		Absolute/Relative
			Maximum	Root Mean Square	
ALOG	U	$0 \leq x \leq 0.5$	5.41×10^{-7}	2.40×10^{-7}	Absolute
	U	$0.5 \leq x \leq 1.5$	9.20×10^{-8}	2.89×10^{-8}	
	U	$1.5 \leq x \leq 5.0$	5.45×10^{-7}	2.27×10^{-7}	
ALOG10	U	$0 \leq x \leq 0.5$	5.54×10^{-7}	1.22×10^{-7}	Absolute
	U	$0.5 \leq x \leq 1.5$	8.53×10^{-8}	3.10×10^{-8}	
	U	$1.5 \leq x \leq 5.0$	9.15×10^{-8}	3.20×10^{-8}	
ATAN	U②	$0 \leq x \leq \tan 1.0$	1.08×10^{-7}	2.87×10^{-8}	Absolute
		$\tan 1.0 \leq x \leq 100$	9.53×10^{-7}	4.57×10^{-7}	
COS	U	$0 \leq x \leq \pi$	1.02×10^{-7}	2.80×10^{-8}	Absolute
	U	$\pi \leq x \leq 100$	1.13×10^{-5}	3.02×10^{-6}	
DATAN	U②	$0 \leq x \leq \tan 1.0$	2.23×10^{-17}	6.97×10^{-18}	Absolute
		$\tan 1.0 \leq x \leq 100$	2.13×10^{-16}	7.72×10^{-17}	
DCOS	U	$0 \leq x \leq \pi$	2.44×10^{-17}	7.73×10^{-18}	Absolute
	U	$\pi \leq x \leq 100$	2.97×10^{-15}	8.57×10^{-16}	
DEXP	U	$\lvert x \rvert \leq 1$	1.51×10^{-16}	3.06×10^{-17}	Relative
	U	$\lvert x \rvert \leq 170$	1.09×10^{-16}	2.65×10^{-17}	
DLOG	U	$0.177 \leq x \leq 0.5$	1.34×10^{-16}	5.13×10^{-17}	Absolute
	U	$0.5 \leq x \leq 1.5$	2.61×10^{-17}	9.04×10^{-18}	
	U	$1.5 \leq x \leq 5.0$	1.36×10^{-16}	5.37×10^{-17}	
DLOG10	U	$0.177 \leq x \leq 0.5$	1.94×10^{-17}	6.38×10^{-18}	Absolute
	U	$0.5 \leq x \leq 1.5$	1.45×10^{-17}	5.11×10^{-18}	
	U	$1.5 \leq x \leq 5.6$	2.01×10^{-17}	6.37×10^{-18}	
DSIN	U	$0 \leq x \leq \pi$	2.69×10^{-17}	7.54×10^{-18}	Absolute
	U	$\pi \leq x \leq 100$	2.96×10^{-15}	8.61×10^{-16}	
DSQRT	E	$16^{-65} \leq x \leq 16^{63}$	1.11×10^{-16}	2.72×10^{-17}	Relative
DTANH	U	$0 \leq x \leq 0.55$	1.44×10^{-17}	4.16×10^{-18}	Absolute
	U	$0.55 \leq x \leq 21.0$	1.22×10^{-17}	4.14×10^{-18}	
	U	$21.0 \leq x \leq 32.0$	1.14×10^{-18}	1.66×10^{-19}	
EXP	U	$\lvert x \rvert \leq 1$	4.63×10^{-7}	1.29×10^{-7}	Relative
	U	$\lvert x \rvert \leq 170$	4.73×10^{-7}	1.16×10^{-7}	
SIN	U	$0 \leq x \leq \pi$	1.06×10^{-7}	3.02×10^{-8}	Absolute
	U	$\pi \leq x \leq 100$	1.13×10^{-5}	3.04×10^{-6}	
SQRT	E	$16^{-65} \leq x \leq 16^{63}$	4.77×10^{-7}	1.17×10^{-7}	Relative
TANH	U	$0 \leq x \leq 0.55$	5.40×10^{-8}	1.72×10^{-8}	Absolute
	U	$0.55 \leq x \leq 9.0625$	5.77×10^{-8}	1.85×10^{-8}	
	U	$9.0625 \leq x \leq 16.0$	2.68×10^{-8}	4.94×10^{-9}	

① E = exponentially distributed argument sample
U = uniformly distributed argument sample
② Sample arguments are distributed so that arctangents are uniformly distributed between arctan 1 and arctan 100.

Figure 29. Accuracy of External Mathematical Functions

The accuracy of an answer produced by these algorithms is influenced by two factors: the performance of the subprogram and the accuracy of the argument. The effect of an argument error upon the accuracy of an answer depends upon the mathematical function involved and not upon the particular coding used in the subprogram.

CONTROL OF PROGRAM EXCEPTIONS IN MATHEMATICAL FUNCTIONS

The FORTRAN mathematical functions were coded with careful control of error situations. A result is provided whenever the answer is within the range representable in the floating-point form. In order to be consistent with FORTRAN control of exponent overflow/underflow exceptions, the following types of conditions are recognized and handled separately.

When the magnitude of the function value is too large to be represented in the floating-point form, the condition is called a terminal overflow: when the magnitude is too small to be represented, a terminal underflow. On the other hand, if the function value is representable, but if execution of the chosen algorithm causes an overflow or underflow in the process, this condition is called an intermediate overflow or underflow.

Function subroutines in the FORTRAN library are coded to observe the following rules for these conditions:

1. Those arguments for which the answer can overflow are excluded from the permitted range of the subroutine.

2. When the magnitude of the answer is less than 16^{-65} zero is given as the answer.

3. Algorithms which can cause an intermediate overflow have been avoided. Therefore an intermediate overflow should not occur during the execution of a function subroutine of the library.

4. Intermediate underflows are detected and not allowed to give an indication. In other words, spurious underflow signals are not allowed. Computation of the function value is successfully carried out.

5. Terminal overflow conditions are screened out by the subroutine. The argument is considered out of range for computation and an error diagnostic is given. Terminal overflow conditions are handled by forcing a floating-point overflow exception. This provides for the detection of overflow in the same manner as for an arithmetic statement. Terminal overflows can occur in the function subroutines EXP and DEXP.

6. Terminal underflow conditions are handled by forcing a floating-point underflow exception. This provides for the detection of underflow in the same manner as for an arithmetic statement. Terminal underflows can occur in the function subroutines EXP and DEXP.

For implicit arithmetic subroutines, these rules do not apply. In this case, both terminal overflows and terminal underflows cause respective floating-point exceptions.

EXPONENTIAL FUNCTIONS

EXP, REAL*4 Exponential Function (Module Names EXP,$FOMM,$FOMC)

Argument Reduction

Acceptable Range: Any argument that results in a function value that can be represented in REAL*4 format is valid. This range is approximately:
-180,218 ≤ argument ≤ 174.673

Error Conditions: If the argument is too large, the overflow indicator is set on, and the result is the largest positive REAL*4 number. If the argument is too small, the underflow indicator is set on, and the result is zero.

Reduction: The argument is multiplied by $\log_2$ (e) and separated into integer and fractional parts for calling the module $FOMC, which computes REAL*4 values of 2^X. ($FOMC is described in *Implicitly Invoked Exponentiation Subprograms* in this chapter.)

Computational Method

The result is computed by the 2^x routine, with $x = \log_2(e) \cdot \text{argument}$. This method depends on the identify:

$$e^{\text{argument}} = 2^{\log_2(e) \cdot \text{argument}}$$

DEXP, REAL*8 Exponential Function (Module Names DEXP,$FOMN,$FOMD)

Argument Reduction

Acceptable Range: Any argument that results in a function value that can be represented in REAL*8 format is valid. This range is approximately:
-180.218 ≤ argument ≤ 174.673

Error Conditions: If the argument is loo large, the overflow indicator is set on, and the result is the largest positive REAL*8 number. If the argument is too small, the underflow indicator is set on, and the result is zero.

Reduction: The argument is multiplied by $\log_2$ (e) and separated into integer and fractional parts for calling the module $FOMD, which computes REAL*8 values of 2^X. ($FOMD is described in *Implicitly Invoked Exponentiation Subprograms* in this chapter.)

Computational Method

The result is computed by the 2^X routine with $x = \log_2(e) \cdot \text{argument}$. This method depends on the identify:

$$e^{\text{argument}} \equiv 2^{\log_2(e) \cdot \text{argument}}$$

LOGARITHMIC FUNCTIONS

ALOG, REAL*4 Natural Logarithm (Module Names ALOG, $FOM5, $FOMI)

Argument Reduction

The range testing and argument reduction are done in module $FOM5, which also computes the $\log_2$ of the reduced argument.

Acceptable Range: The argument must be a positive normalized number. Zero is not acceptable because the logarithm of zero is undefined. Negative numbers are not acceptable because the logarithm of a negative number is a complex number with a nonzero imaginary part.

Error Conditions: If the argument is zero, the result is set to the negative number with the largest possible magnitude, and bit 7 of the FTEST byte is set to 1. If the argument is negative, the logarithm of its magnitude is computed, and bit 7 of the FTEST byte is set to 1.

Reduction: The argument is separated into two parts, c' and m', These are computed from the characteristic and the mantissa of the argument, c and m respectively, as follows:

$$c' = 4(c-64) - a - \tfrac{1}{2}$$

$$m' = |m| \cdot 2^{a}$$

where a is the number of leading zeros in m. This produces c' and m' such that:

$$-259\tfrac{1}{2} \leq c' \leq 251\tfrac{1}{2}$$

$$\tfrac{1}{2} \leq m' < 1$$

The argument (representing the number $16^{(c-64)} \cdot |m|$) can also be represented as:

$$2^{c'} \cdot \sqrt{2} \cdot m'$$

Computational Method

The $\log_2$ of $\sqrt{2} \cdot m'$ is computed using the polynomial approximation:

$$\log_2(\sqrt{2} \cdot m') \cong \frac{2(m' - \sqrt{2}/2)}{m'/2 + \sqrt{2}/4} \cdot \left\{ p_0 + p_1 \left(\frac{2(m' - \sqrt{2}/2)}{m'/2 + \sqrt{2}/4}\right)^2 + p_2 \left(\frac{2(m' - \sqrt{2}/2)}{m'/2 + \sqrt{2}/4}\right)^4 + p_3 \left(\frac{2(m' - \sqrt{2}/2)}{m'/2 + \sqrt{2}/4}\right)^6 \right\}$$

The $\log_2$ of the argument is then $c' + \log_2(\sqrt{2} \cdot m')$ so the result can be computed as:

$$\log_e(\text{argument}) = \log_e(2) \cdot \log_2(\text{argument})$$

$$= \log_e(2) \cdot (c' + \log_2(\sqrt{2} \cdot m'))$$

ALOG10,REAL*4 Base 10 Logarithm (Module Names ALOG10,$FOMK,$FOM5)

Argument Reduction

Identical to ALOG, including the acceptable range and error conditions.

Computational Method

Identical to ALOG except for the last step, which is:

$\log_{10}(\text{argument}) = \log_{10}(2) \cdot (c' + \log_2(\sqrt{2} \cdot m'))$

The only difference from the computation of ALOG is that $\log_{10}(2)$ is used instead of $\log_e(2)$.

DLOG,REAL*8 Natural Logarithm (Module Names DLOG,$FOMJ,$FOM6)

Argument Reduction

The range testing and argument reduction are done in module $FOM6, which also computes the $\log_2$ of the reduced argument. The acceptable range, error conditions, and reduction are identical to those for ALOG, except that the reduced mantissa occupies 7 bytes instead of 3.

Computational Method

The $\log_2$ of $\sqrt{2} \cdot m'$ is computed by the same method as for ALOG, using a seventh order polynomial instead of a third order, as in ALOG, to achieve the accuracy required. The $\log_e(\text{argument})$ is then computed as:

$\log_e(\text{argument}) = \log_e(2) \cdot (c' + \log_2(\sqrt{2} \cdot m'))$

DLOG10,REAL*8 Base 10 Logarithm (Module Names DLOG10,$FOML,$FOM6)

Argument Reduction

Identical to DLOG, including the acceptable range and error conditions.

Computational Method

Identical to DLOG except for the last step, which is:

$\log_{10}(\text{argument}) = \log_{10}(2) \cdot (c' + \log_2(\sqrt{2} \cdot m'))$

The only difference from the computation of DLOG is that $\log_{10}(2)$ is used instead of $\log_e(2)$.

TRIGONOMETRIC FUNCTIONS

SIN/COS, REAL*4 Sine/Cosine (Module Names SIN,COS,$FOM1,$FOM3)

Argument Reduction

Acceptable Range: The sine/cosine routine accepts normalized arguments in the range

$$|\text{argument}| < \pi \cdot 2^{18} \cong 8 \cdot 10^5$$

and zero. Arguments outside this range are not accepted because they are not precise enough to yield a meaningful result after the argument is reduced to the principal range. This range limit is not due to a limitation of the algorithm used. Rather, it is a consequence of the periodicity of the trigonometric functions and the representation used for REAL*4 numbers.

Error Conditions: If an argument is out-of-range, a zero result is returned, and bit 6 in the FTEST byte is set to 1.

Reduction: The argument is separated into two parts after multiplying its absolute value by $4/\pi$. Let q be the integer part of the result, and r be the fractional part of the result. If x is the argument,

$$\left\{\begin{matrix}\sin\\ \cos\end{matrix}\right\}(|x|) = \left\{\begin{matrix}\sin\\ \cos\end{matrix}\right\}\left(\frac{\pi}{4}\left(|x| \cdot \left(\frac{4}{\pi}\right)\right)\right) = \left\{\begin{matrix}\sin\\ \cos\end{matrix}\right\}\left(\frac{\pi}{4}(q+r)\right)$$

Negative arguments are treated by adding 4 to q if the function desired is SIN, since $\sin(-x) = \sin(x+\pi) = \sin(x + 4(\frac{\pi}{4}))$. No change is required for COS, since $\cos(-x) = \cos(x)$.

The computation of COS is transformed into the computation of SIN by adding 2 to q, since $\cos(x) = \sin(x + \frac{\pi}{2}) = \sin(x + 2(\frac{\pi}{4}))$

Let $q_0 \equiv q \bmod 8$.

Then, for $q_0 = 0$, $\sin(x) = \sin\left(\frac{\pi}{4} \cdot r\right)$,

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

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

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

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

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

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

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

These formulas reduce each case to the computation of either $\sin\left(\frac{\pi}{4} \cdot r_1\right)$ or $\cos\left(\frac{\pi}{4} \cdot r_1\right)$; where r_1 is either r or $(1 - r)$, and is within the range, $0 \leqslant r_1 \leqslant 1$.

Computational Method

$\sin\left(\frac{\pi}{4}\cdot r_1\right)$ is computed by a polynomial of the form:

$$\sin\left(\frac{\pi}{4}\cdot r_1\right) \cong r_1\,(a_0 + a_1 r_1^{\,2} + a_2 r_1^{\,4} + a_3 r_1^{\,6}).$$

$\cos\left(\frac{\pi}{4}\cdot r_1\right)$ is computed by a polynomial of the form:

$$\cos\left(\frac{\pi}{4}\cdot r_1\right) \cong 1 + b_1 r_1^{\,2} + b_2 r_1^{\,4} + b_3 r_1^{\,6} + b_4 r_1^{\,8}$$

These polynomials are computed by calling module $FOM3.

ATAN, REAL*4 Arctangent (Module Names ATAN,$FOME)

Argument Reduction

Acceptable Range: Any argument in the range representable in REAL*4 format is valid.

Error Conditions: None.

Reduction: The magnitude of the argument x is used to determine two numbers, A and Z, as follows:

$$\text{for } \tan(0) \leqslant |x| < \tan\left(\frac{\pi}{16}\right),\ A = 0,\ z = |x| - \tan(0) = |x|$$

$$\text{for } \tan\left(\frac{\pi}{16}\right) \leqslant |x| < \tan\left(\frac{3\pi}{16}\right),\ A = \pi/8,\ z = |x| - \tan\left(\frac{\pi}{8}\right)$$

$$\text{for } \tan\left(\frac{3\pi}{16}\right) \leqslant |x| < \tan\left(\frac{5\pi}{16}\right),\ A = \pi/4,\ z = |x| - \tan\left(\frac{\pi}{4}\right)$$

$$\text{for } \tan\left(\frac{5\pi}{16}\right) \leqslant |x| < \tan\left(\frac{7\pi}{16}\right),\ A = 3\,\pi/8,\ z = |x| - \tan\left(\frac{3\pi}{8}\right)$$

$$\text{for } \tan\left(\frac{7\pi}{16}\right) \leqslant |x| < \tan\left(\frac{\pi}{2}\right),\ A = \pi/2,\ z = -\frac{1}{|x|}$$

Computational Method

Let $y = \tan^{-1}(|x|)$. Then, using the formula for the tangent of the difference of two angles, the following identity can be derived:

$$y \equiv A + \tan^{-1}\left(\frac{\tan(A) - \tan(y)}{1 + \tan(A)\cdot\tan(y)}\right) = A + \tan^{-1}\left(\frac{z}{\sec^2(A) + z\cdot\tan(A)}\right)$$

This formula is used for $A = 0, \frac{\pi}{8}, \frac{\pi}{4}$, or $\frac{3\pi}{8}$, with the arctangent being approximated by the polynomial

$$\tan^{-1}(R) \cong R\,(p_0 + p_1\cdot R^2 + p_2\cdot R^4 + p_3\cdot R^6)$$

For $A = \left(\frac{\pi}{2}\right)$, since $\tan^{-1}(|x|) \equiv \left(\frac{\pi}{2}\right) + \tan^{-1}\left(-\frac{1}{|x|}\right)$, the formula is

$$y = A + \tan^{-1}(z)$$

Finally, if the argument was negative, the result is made negative, because $\tan^{-1}(-x) = -\tan^{-1}(x)$

DSIN/DCOS, REAL*8 Sine/Cosine (Module Names DSIN, DCOS,$FOM2,$FOM4)

Argument Reduction

Acceptable Range: The precision available in REAL*8 variables allows an acceptable range of arguments of normalized values in the range

$|\text{argument}| < \pi \cdot 2^{50} \cong 3 \cdot 10^{15}$

and zero. See the description of SIN/COS for a discussion of the reasons for this range limit.

Error Conditions: Same as SIN/COS.

Reduction: Same as SIN/COS except that the reduction is carried out in REAL*8 arithmetic.

Computational Method

$\sin\left(\frac{\pi}{4} \cdot r_1\right)$ is computed by a polynomial of the form:

$$\sin\left(\frac{\pi}{4} \cdot r_1\right) = r_1 \left(a_1 r_1{}^2 + a_2 r_1{}^4 + a_3 r_1{}^6 + a_4 r_1{}^8 + a_5 r_1{}^{10} + a_6 r_1{}^{12}\right)$$

$\cos\left(\frac{\pi}{4} \cdot r_1\right)$ is computed by a polynomial of the form:

$$\cos\left(\frac{\pi}{4} \cdot r_1\right) = 1 + b_1 r_1{}^2 + b_2 r_1{}^4 + b_3 r_1{}^6 + b_4 r_1{}^8 + b_5 r_1{}^{10} + b_6 r_1{}^{12} + b_7 r_1{}^{14}$$

These polynomials are evaluated by calling module $FOM4.

DATAN, REAL*8 Arctangent (Module Names DATAN,$FOMF)

Argument Reduction

Acceptable Range: Any argument in the range representable in REAL*8 format is valid.

Error Conditions: None.

Reduction: The magnitude of the argument, x, is used to determine two numbers, A and Z, as follows:

for $\tan(0) \leqslant |x| < \tan\left(\frac{\pi}{20}\right)$, $A = 0$, $z = |x| - \tan(0) = |x|$

for $\tan\left(\frac{\pi}{20}\right) \leqslant |x| < \tan\left(\frac{3\pi}{20}\right)$, $A = \frac{\pi}{10}$, $z = |x| - \tan\left(\frac{\pi}{10}\right)$

for $\tan\left(\frac{3\pi}{20}\right) \leqslant |x| < \tan\left(\frac{\pi}{4}\right)$, $A = \frac{\pi}{5}$, $z = |x| - \tan\left(\frac{\pi}{5}\right)$

for $\tan\left(\frac{\pi}{4}\right) \leqslant |x| < \tan\left(\frac{7\pi}{20}\right)$, $A = \frac{3\pi}{10}$, $z = |x| - \tan\left(\frac{3\pi}{10}\right)$

for $\tan\left(\frac{7\pi}{20}\right) \leqslant |x| < \tan\left(\frac{9\pi}{20}\right)$, $A = \frac{2\pi}{5}$, $z = |x| - \tan\left(\frac{2\pi}{5}\right)$

for $\tan\left(\frac{9\pi}{20}\right) \leqslant |x| < \tan\left(\frac{\pi}{2}\right)$, $A = \frac{\pi}{2}$, $z = -\frac{1}{|x|}$

Computational Method

The computation is by the same method as for ATAN, except that the polynomial approximation for $\tan^{-1}$ is

$$\tan^{-1}(R) \cong R(p_0 + p_1 \cdot R^2 + p_2 \cdot R^4 + p_3 \cdot R^6 + p_4 \cdot R^8 + p_5 \cdot R^{10} + p_6 \cdot R^{12} + p_7 \cdot R^{14})$$

SQUARE ROOT FUNCTIONS

SQRT, REAL*4 Square Root (Module Names SQRT and $FOMG)

Argument Reduction

Acceptable Range: The square root routine successfully computes the square root of any nonnegative number representable in REAL*4 format. Negative numbers are out-of-range since the square root of a negative number is imaginary.

Error Conditions: If the argument is negative, bit 5 in the FTEST byte is set to 1, and the square root of the argument's magnitude is taken.

Reduction: If the argument is zero, a zero result is returned immediately. Otherwise, the characteristic of the argument is tested. If it is even, no change is made to the mantissa. If it is odd, the mantissa is divided by 16. A divide instruction is not used to perform this division; the operand is shifted by additions.

Computational Method

The characteristic, c, of the result is computed as:
c(result) = c(argument)/2 for c(argument) even
c(result) = (1 + c(argument))/2 for c(argument) odd

The mantissa, m, of the result is the square root of the mantissa of the reduced argument. Because:

$1/256 \leq m(\text{reduced argument}) < 1,$

$1/16 \leq m(\text{result}) < 1,$

and the result is therefore a normalized REAL*4 number.

This method reflects the identities:

$$\left[16^{c(\text{argument})} \cdot m(\text{argument})\right]^{1/2} \equiv \left[16^{c(\text{argument})}\right] \cdot \left[m(\text{argument})\right]^{1/2} \quad \text{for } c(\text{argument}) \text{ even}$$

$$\left[16^{c(\text{argument})} \cdot m(\text{argument})\right]^{1/2} \equiv \left[16^{1 + c(\text{argument})}\right]^{1/2} \cdot \left[m(\text{argument})/16\right]^{1/2} \quad \text{for } c(\text{argument}) \text{ odd.}$$

The square root of the reduced mantissa is computed by a series of successive subtractions, with each set of subtractions determining one hexadecimal digit of the result from two hexadecimal digits of the argument's reduced mantissa and remainder (if any) and the results from the previous sets of subtractions. This method is similar to the normal method for the hand extraction of square roots. At the completion of the subtractions, at least 21 binary digits of the result are correct. A correction to the last binary digits is computed and applied if necessary.

Accuracy

The method produces the correct result in the characteristic and the correct result, rounded to 6 hexadecimal digits, for the mantissa.

DSQRT, REAL*8 Square Root (Module Names DSQRT and $FOMH)

Argument Reduction

Identical to that for SQRT except that the argument is in REAL*8 format. Acceptable range and out-of-range action are the same.

Computational Method

Identical to SQRT except that 14 hexadecimal digits are developed in the result mantissa, and the subtractions are carried out only to the precision necessary at each stage in the development of the result mantissa.

Accuracy

The method produces the correct result in the characteristic, and the correct result, rounded to 14 hexadecimal digits, for the mantissa.

HYPERBOLIC TANGENT FUNCTIONS

TANH, REAL*4 Hyperbolic Tangent (Module Names TANH,$FOMO,$FOMM)

Argument Reduction

Acceptable Range: Any number representable in REAL*4 format is acceptable.

Error Conditions: None.

Reduction: None.

Computational Method

In the range $|x| \leqslant .55$, tanh x is computed with the approximation:

$$\tanh(x) \cong x \cdot [1 - x^2 \cdot (c_1 \frac{x^2}{c_2 + c_3 \cdot x^2})]$$

In the range $.55 < |x| < 9.0625$ the approximation is:

$$\tanh(x) \cong \text{sign}(x) \left[.5 - .5 \left(\frac{z}{.75 + .25_z} - 1 \right) \right]$$

where $z = e^{2(1/2 \ln 3 - x)}$

The REAL*4 exponential module, $FOMM, is used to compute $e^{2|x|}$

For $|x| \leqslant 9.0625$, $\tanh(x) \cong \text{sign}(x) \cdot 1.0$

DTANH, REAL*8 Hyperbolic Tangent (Module Names DTANH,$FOMP,$FOMN)

Argument Reduction

Acceptable Range: Any number representable in REAL*8 format is acceptable.

Error Conditions: None.

Reduction: None.

Computational Method

In the range $|x| \leqslant .55$, tanh x is computed with the approximation:

$$\tanh(x) \cong x \cdot \left[1 - x^2 \cdot \left\{ \frac{c_0 + x^2 (c_1 + c_2 x^2)}{d_0 + x^2 (d_1 + x^2 (d_2 + x^2))} \right\} \right]$$

In the range $.55 < |x| < 21.0$, the approximation is:

$$\tanh(x) \cong \text{sign}(x) \left[.5 - .5 \left(\frac{z}{.75 + .25_z} - 1 \right) \right]$$

where $z = e^{2(1/2 \ln 3 - |x|)}$

The REAL*8 exponentiation module, $FOMN, is used to compute $e^{2|x|}$

For $|x| > 21.0$, $\tanh(x) = \text{sign}(x) \cdot 1.0$.

IMPLICITLY INVOKED EXPONENTIATION SUBPROGRAMS

$FOM7, Subprogram for I**J (Module Name $FOM7, Secondary Entry Point #FOM7)

Argument Reduction

Acceptable Range: Any pair of arguments for which the result does not exceed the integer range is valid, except that I = 0 and $J \leq 0$ are invalid.

Error Conditions: If I = 0 and $J \leq 0$, bit 3 in the FTEST byte is set to 1, and a result of zero is returned. No indication is given if the result exceeds the integer range. This range is:

$-2^{31} \leq \text{Result} \leq 2^{31} - 1$ for Integer *4
$-2^{15} \leq \text{Result} \leq 2^{15} - 1$ for Integer *2

Reduction: No reduction is performed other than the check for invalid arguments.

Computational Method

Five special cases are checked first:
If I = 0,J > 0, the result is zero.
If $I \neq 0$, J = 0, the result is 1.
If I = 1, the result is 1 for any J.
If I = -1, the result is -1 for J odd, + 1 for J even.
If $I \neq 1$, J < 0, the result is zero.
If none of these special cases exists, J is positive and the result is computed as follows: Set a partial result equal to I, then scan J for its leftmost bit having the value 1. For each bit position in J to the right of the leftmost 1 bit, square the partial result, and if the bit is a 1, multiply the partial result by I. This method is based on the identity:

$$I^J = I^{(j_{30} \cdot 2^{30} + j_{29} \cdot 2^{29} + \ldots + j_0 \cdot 2^0)}$$

$$= (^{j_{30}})^{2^{30}} \cdot (I^{j_{29}})^{2^{29}} \cdot \ldots \cdot (I^{j_0})^{2^0}$$

$$= ((\ldots(I^{j_{30}})^2 \cdot I^{j_{29}})^2 \cdot \ldots)^2 \cdot I^{j_0}$$

where J indicates the bits of J, with J_0 being the rightmost bit. The partial result becomes the final result when all the bits of J are processed. The method is the same for INTEGER*2 operands, except that the operands contain fewer bits, and a secondary entry point to the module is used. The compiler uses the entry point #FOM7 when J is INTEGER*2, regardless of the length of I. I is always in INTEGER*4 form for this module as a result of the automatic extension of INTEGER*2 to INTEGER*4 when the register is loaded before this module is invoked.

Accuracy

The result is exact, provided that the result does not exceed the allowable range.

$FOM8, Subprogram for A**J (Module Name $FOM8)

Argument Reduction

Acceptable Range: Any pair of arguments that produces a result representable in REAL*4 format is acceptable, except that argument pairs for which J is negative and the result's magnitude is in the range $[16^{-65}, (16^{63}(1-16^{-6}))^{-1}]$ are not acceptable.

Error Conditions: If A = 0.0 and $J \leq 0$, bit 2 of the FTEST byte is set to 1, and the result is set to 0.0. Other out-of-range conditions are handled as follows:

	$\|A\| < 1.0$	$\|A\| > 1.0$
$J > 0$	Underflow Indicator On $\|Result\| = 0.0$	Overflow Indicator On $\|Result\| = 16^{63}(1-16^{6})$
$J < 0$	Underflow Indicator On Divide Check Indicator On (1) $\|Result\| = 1.0$	Overflow indicator On $\|Result\| = [16^{63}(1-16^{-6})]^{-1}$

(1) The result of 1.0 is due to a division of 1.0 by the zero resulting from the underflow when $A^{|J|}$ is computed.

Reduction: No reduction is performed on the arguments after the test for A = 0.0 and $J \leq 0$.

Computational Method

The arguments are tested for these special cases:

If A = 0.0 and $J > 0$, the result is 0.0

If J = 0, the result is 1.0

Otherwise, A**|J| is computed using the same method as for integers in $FOM7, except that REAL*4 arithmetic is used for the multiplications. Then, if J were negative, the reciprocal of the result is taken. If J is INTEGER*2, it is extended to INTEGER*4 by the instruction that loads J before this module is invoked.

$FOM9, Subprogram for D**J (Module name $FOM9)

Identical to $FOM8 for A**J except that the arithmetic and results are REAL*8 and the range limit of the result is $[16^{63}(1-16^{44})]^{-1}$ when $J < 0$ and $|D| < 1.D0$.

$FOMA, Subprogram for AB (Module Names $FOMA,$FOM5,$FOMC)**

If A is not REAL*4, the compiler inserts instructions to convert A to REAL*4 format before $FOMA is invoked. Thus, $FOMA itself computes only (REAL*4)**(REAL*4).

Argument Reduction

Acceptable Range: For A > 0, the value of B must produce a result representable in REAL*4 format. For A = 0,B must be greater than zero. A < 0 is invalid.

Error Conditions: If A > 0 and the result is too large, the overflow indicator is turned on and the result is set to the largest positive REAL*4 number. If A > 0 and the result is too small, the underflow indicator is turned on and the result is set to zero. If A = 0 and B is negative, bit 1 in the FTEST byte is set to 1 and the result is set to zero. If A < 0, bit 7 in the FTEST byte is set to 1 and the absolute value of A is used. In this case, underflow or overflow can also occur if the result is too small or too large, respectively; the results are the same as for A < 0.

Reduction: No reduction is performed in this module.

Computational Method

The method is based on the identity
$$A^b = 2^{b \cdot \log_2 a}$$
The REAL*4 $\log_2$ module, $FOM5, is called with A as the argument. The result is multiplied by B and passed as the argument to the REAL*4 2^x module, $FOMC. This result is then returned as the result of the exponentiation.

$FOMB, Subprogram for AB (Module Names $FOMB,$FOM6,$FOMD)**

If A is REAL*4, or if B is INTEGER*2, INTEGER*4, or REAL*4, the compiler inserts instructions to convert them to REAL*8 format before $FOMB is invoked. Thus, $FOMB itself computes only (REAL*8)**(REAL*8).

Argument Reduction

The range, error conditions, and argument reduction are identical to those for the REAL*4A**B module, $FOMA, except that:

1. The result can be slightly greater without overflow since the largest REAL*8 number is larger than the largest REAL*4 number.

2. The largest positive REAL*8 number is the result in case of overflow.

Computational Method

The method is identical to that for the REAL*4A**B module, $FOMA, except that the REAL*8 modules for 2^x and $\log_2$, $FOMD and $FOM6, are used.

$FOMC, REAL*4 Subprogram to Compute 2^x (Module Name $FOMC)

Argument Reduction

Acceptable Range: Any argument that results in a function value that can be represented in REAL*4 format is valid. This range is approximately:

$-260 \leq x < 252$

Error Condition: If x is too large, the overflow indicator is turned on and the result is the largest positive REAL*4 number. If x is too small, the underflow indicator is turned on and the result is zero.

Reduction: The argument is received in two parts, an integer part, i, and a fractional part, f. The fractional part is brought into the range $[-\frac{1}{2},\frac{1}{2}]$ by computing i' and f' as follows:

if $f > \frac{1}{2}$, $f' = f-1$ and $i' = i + 1$

if $f < -\frac{1}{2}$, $f' = f + 1$ and $i' = i + 1$

otherwise $f' = f$ and $i' = i$

Then i' is separated into two parts, i'_1, and i'_2, which satisfy the relation

$i = 4 \cdot i'_1, + i'_2$ with $-4 \leq i'_2 \leq -1$

The desired result can be written:

$2^x = 2^{i+f} = 2^{i'+f'} = 2^{4i'_1 + i'_2 + f'} = 16^{i'_1} \cdot 2^{i_2} \cdot 2^{f'}$

Computational Method

Compute $2f'$ using the approximation:

$$2^f \cong 1 + 8 \cdot \left(\frac{(2f') \cdot P((2f')^2)}{Q((2f')^2) - 4 \, [\, (2f') \cdot P((2f')^2) \,]} \right)$$

Where

$P((2f')^2) = p_0 + p_1 \cdot ((2f')^2)$

$Q((2f')^2) = q_0 + q_1 \cdot ((2f')^2)$

Multiply the result by $2^{i'_2}$, and multiply that result by $16^{i'_1}$ by adding i_1' to its characteristic.

$FOMD, REAL*8 Subprogram to Compute 2^X (Module Name $FOMD)

Argument Reduction

The argument reduction, range, and error conditions are identical to those for the REAL*4 2^X subprogram, $FOMC, except:

1. The upper limit of the range is slightly greater, since the largest REAL*8 number is greater than the largest REAL*4 number.

2. The result for an argument too large is the largest REAL*8 number.

Computational Method

The method is the same as for the REAL*4 2^X subprogram, except that one more term is used in the polynomials P and Q:

$$P((2f)^2) = p_0 + p_1((2f)^2) + p_2((2f)^2)^2$$
$$Q((2f)^2) = q_0 + q_1((2f)^2) + p_2((2f)^2)^2$$

Chapter 23. FORTRAN Service Subprograms

MACHINE INDICATOR TEST SUBPROGRAMS

The machine indicator subprograms test the status of pseudo-indicators and return a value to the calling program. When the indicator is zero, it is off; when the indicator is other than zero, it is on. In the following descriptions of the subprograms, *i* represents an integer expression and *j* represents an integer variable.

Pseudosense Light Subprogram (Entry Names: SLITE/SLITET)

This subprogram is used to alter, test, and/or record the status of pseudosense lights. Either of two entry names (SLITE or SLITET) is used to call the subprogram. The particular entry name used in the CALL statement depends upon the operation to be performed.

If the *four* sense lights are to be turned off or *one* sense light is to be turned on, entry name SLITE is used. The source language statement is:

```
CALL SLITE (i)
```

where i has a value of 0, 1, 2, 3, or 4.

If the value of i is 0, the four sense lights are turned off; if the value of i is 1, 2, 3, or 4, the corresponding sense light is turned on. If the value of i is not 0, 1, 2, 3, or 4, halt code 14 is issued and execution of this module or phase is terminated.

If a sense light is to be tested and its status recorded, entry name SLITET is used. Regardless of its status before the test, after a sense light is tested, it is always set off. The source language statement is:

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

where

i has a value of 1, 2, 3, or 4, and indicates which sense light to test.

Note: If the value of i is not 1, 2, 3, or 4, halt code 15 is issued and execution of this module or phase is terminated.

j has a value returned by the subprogram. 1 indicates the sense light was on; 2 indicates the sense light was off.

Divide Check Subprogram (Entry Name: DVCHK)

This program tests for a divide-check exception (divisor = zero) and returns a value that indicates the existing condition. After testing, the divide-check indicator is turned off. This subprogram is called by using entry name DVCHK in a CALL statement. The source language statement is:

```
CALL DVCHK (j)
```

where j is set to 1 if the divide-check indicator was on, or to 2 if the indicator was off.

Overflow Indicator Subprogram (Entry Name: OVERFL)

This subprogram tests for an exponent overflow or underflow exception and returns a value that indicates the existing condition. After testing, the overflow indicator is turned off. This subprogram is called by using the entry name OVERFL in a CALL statement. The source language statement is:

```
CALL OVERFL (j)
```

where j is returned by the subprogram to indicate the following:

1 = floating-point overflow condition occurred last.
2 = no overflow condition occurred.
3 = a floating-point underflow condition occurred last.

UTILITY SUBPROGRAMS

The utility subprograms perform the following operations for the FORTRAN programmer:

- Terminate execution (EXIT)
- Dump a specified area of storage (DUMP/PDUMP)
- Test the bits of a setting of the address/data switch on the System/3 console (DATSW)
- Test an error indicator (FCTST),
- Test for an inquiry request (INQCHK/SETINQ)
- Retrieve the system date and time-of-day (CFTOD).

End Execution Subprogram (Entry Name: EXIT)

The end execution subprogram terminates execution and returns control to the operating system. (EXIT performs a function identical to that performed by an unnumbered STOP statement.) This subprogram is called by using the entry name EXIT in a CALL statement. The source language statement is:

```
CALL EXIT
```

Storage Dump Subprogram (Entry Names: DUMP/PDUMP)

This subprogram dumps a specified area of storage. Either of two entry names (DUMP or PDUMP) can be used to call the subprogram. The entry name is followed by the limits of the area to be dumped and the format specification. The entry name used in the CALL statement depends upon whether you want to resume execution of the FORTRAN program after the dump.

If execution of the load module or phase is to be terminated after the dump is taken, entry name DUMP is used. The source language statement is:

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

where

a and b are variables whose names indicate the limits of storage to be dumped (either a or b can be the name of the upper or lower limit of storage), which must be in the same program or subprogram or in common:

f indicates the dump format and can be one of the following:

0 = hexadecimal
3 = INTEGER*2
4 = INTEGER*4
5 = REAL*4
6 = REAL*8

Any positive number other than the preceding results in a hexadecimal dump, provided the number is less than 256. (The low-order byte of this argument is all that is used. If the number is negative or greater than 255, you must determine the bit pattern in the low-order byte to determine the dump format.)

When hexadecimal format is used, the dump program assumes that the upper limit of the dump is a variable 8 bytes long. If it is actually 2 or 4 bytes long, the last 6 or 4 bytes of the dump (respectively) are not part of the upper-limit variable.

If execution is to resume after the dump is taken, entry name PDUMP is used. The source language statement is:

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

where a, b, and f have the same meanings as for DUMP.

DUMP/PDUMP output is directed to the FORTRAN error logging device. See *PRINT and NOPRINTER Device Option Statements* for more information.

DUMP/PDUMP Programming Considerations

A load module or phase can occupy a different area of storage each time it is executed. To ensure that the appropriate areas of storage are dumped, the following conventions should be followed.

Transaction Logging Subprogram (SUBR81) 5704-FO2 only

This subprogram is used with $TRLOG to log transaction oriented data to tape. The source language statement is:

```
CALL SUBR81(i,j)
```

where

i is an integer *2 variable or array element and must contain the length of the data to be logged. The length must not exceed 2,040 bytes.

j is a variable, an array element, or an array containing the data to be logged.

For additional information about transaction logging ($TRLOG), see *IBM System/3 Model 15 System Control Programing Concepts and Reference Manual,* GC21-5162.

Example:

```
INTEGER*2 LGTH
INTEGER*4 LOGDAT(32)
DATA LGTH/128/
   .
   .
   .
CALL SUBR81 (LGTH,LOGDAT)
   .
   .
   .
CALL SUBR81 (LGTH,LOGDAT)
```

If an array and a variable are to be dumped at the same time, a separate set of arguments should be used for the array and for the variable. The specification of limits for the array should be from the first element in the array to the last element. For example, assume that B is a REAL number, and TABLE is an array of 20 elements. The following call to the storage dump subprogram could be used to dump TABLE and B in the hexadecimal format and teminate execution after the dump is taken:

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

If an area of storage in common is to be dumped at the same time as an area of storage not in common, the arguments for the area in common should be given separately. For example, assuming A is in common and B is not, the following call to the storage dump subprogram should be used to dump the variables A and B in REAL*8 format without terminating execution:

```
CALL PDUMP(A,A,6,B,B,6)
```

If variables not in common are to be dumped, each variable must be listed separately in the argument list. For example, if R, P, and Q are not in common, the statement:

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

should be used to dump the three variables If the statement

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

is used, all main storage between R and Q is dumped, which might or might not include P, and might include other variables.

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

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

and the call to SUBI within the source module is:

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

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

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

If the statement

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

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

Address/Data Switch Subprogram (Entry Name: DATSW)

This subprogram tests the setting of the address/data switches. The four address/data switches correspond to 16 binary switches, numbered 0 to 15 from left to right, as shown in Figure 29. The binary switches 0-15 are either off (0) or on (1) depending upon the hexadecimal setting of the address/data switches.

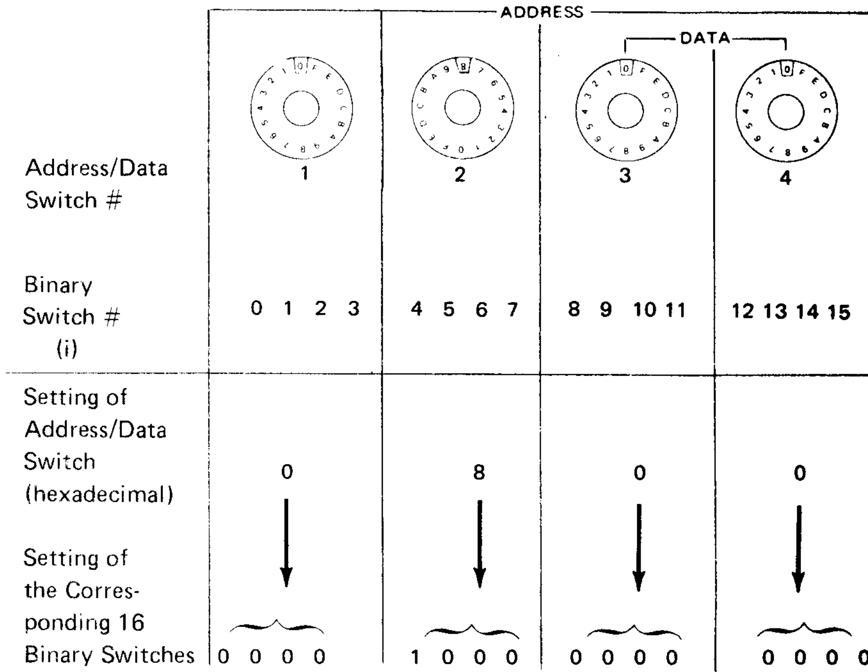


Figure 29. Address Data Switch Settings

In Figure 29, the address/data switches are set to:

0800

Therefore, only binary switch 4 is on (1), and the other binary switches are off (0).

The DATSW subprogram is called by the FORTRAN statement:

```
CALL DATSW(i,j)
```

where

i is an INTEGER*4 constant, expression, or variable in the range 0 to 15, indicating the binary switch to be tested.

Note: If i is outside the permitted range, halt code RFƀ7, subhalt 16 is issued.

j is an INTEGER*4 variable, which is set to 1 if the value of the tested binary switch was 1, or is set to 2 if the value of the binary switch was 0.

Example:

Setting of address/data switches	4	A	7	0
Setting of the corresponding binary switches	0100	1010	0111	0000

Here, binary switches 1, 4, 6, 9, 10, and 11 are on (are 1). Therefore,

```
CALL DATSW(4,J)
```

would set J to 1, and

```
CALL DATSW(5,J)
```

would set J to 2.

CAUTION
Binary switches 12-15 should be used with care on the Model 10 because the rightmost console dial must be used in replies to system halts, and might be changed during the execution of a program.

Example:

```
PAUSE 55
CALL DATSW(13,J)
```

The PAUSE statement requires a replay of 0 to continue processing, causing binary switches 12, 13, 14, and 15 to be set off (to 0).

Library Function Error Subprogram (Entry Name: FCTST)

This subprogram is used to test an indicator word for an error in a library function subprogram. After testing, all conditions are turned off. The subprogram is called by writing the FORTRAN statement:

```
CALL FCTST(j,k)
```

where

j is set to 1 if any bit in the indicator word is on (1), or to 2 if all the bits in the indicator word are off (0).

k is the indicator word. The error indicator is returned as the rightmost byte of this word (bits 24-31). The other bytes are zero.

The errors that can be detected are shown in Figure 30.

The value of the error word can be 1, 2, 4, 8, 16, 32, 64, or 128, or the sum of 2 or more of these numbers if more than one error condition occurred. In such a case, the value returned is the sum of the integers associated with the individual conditions. For example, if an error was detected in DSIN (value 2), and an error was detected in SQRT (value 4), the value of k is 6.

Value (decimal)	Routine	Error	Result
1	ALOG/DLOG	argument = 0	Largest negative value
1	ALOG/DLOG	argument < 0	Absolute value of argument used for evaluation
1	Real to Real exponentiation	negative base and nonzero exponent	Absolute value of argument used for evaluation
1	SIN/COS	argument magnitude exceeds $\pi \cdot 2^{18}$	Zero
2	DSIN/DCOS	argument magnitude exceeds $\pi \cdot 2^{50}$	Zero
4	SQRT/DSQRT	argument < 0	Absolute value of argument used for evaluation
8	Real to Integer conversion	overflow on conversion	Maximum value of appropriate sign
16	Integer to Integer exponentiation	base = 0 and exponent ≤ 0	Zero
32	Real to Integer exponentiation	base = 0 and exponent ≤ 0	Zero
64	Real to Real exponentiation	base = 0 and exponent ≤ 0	Zero
128		Invalid input character to a conversion routine	Character treated as zero

Note: For exponentiation routines, the error can occur for any length item, that is, for I*2, I*4, R*4, and R*8, where applicable.

Figure 30. Error Conditions that can be Tested in FCTST

ROLLOUT Support Subprogram (Entry Name: INQCHK)

System/3 Models 6, 10, 12, and Model 15 with Program Number 5704-FO1 allow programs to be interrupted (ROLLOUT/ROLLIN) while they are being processed; Model 15 with Program Number 5704-FO2 does not.

To request an interruption:

Model 6	Set the INQUIRY REQUEST switch to ON.
Models 10 and 12	Press the REQUEST key on the printer keyboard.
Model 15 with Program Number 5704-FO1	Press the PF10 key, issue ROLLOUT via the keyboard, and press ENTER.

Programs are normally interrupted to permit another program to run. After this program is run, control is then given back to the original program (the one that was interrupted).

For more information about the ROLLOUT/ROLLIN routines, see the appropriate operator's guide listed under *Related Publications* in the Preface.

The subprogram INQCHK allows the user to interrupt the executing FORTRAN program in order to execute another program. Calls to INQCHK should be placed at appropriate points in the interruptable program, so that the interrupt request is serviced within a reasonable interval. The INQCHK subprogram has no parameters. The format of the source language statement is:

```
CALL INQCHK
```

If no inquiry is pending, the subprogram immediately returns control to the interruptable program. If an interrupt request has been made:

1. A system control program ROLLOUT routine moves the interrupted primary program from main storage to disk.
2. The desired secondary program is processed; this program cannot be interrupted.
3. After the secondary program has been executed, the interrupted program moves back into main storage using a ROLLIN routine. The interrupted program resumes execution at the point beyond the call to INQCHK and terminates in a normal manner.

Notes:
1. In the dual programming mode, only program level 1 or partition 1 programs can be interrupted and moved out of main storage by a ROLLOUT routine.
2. The Model 15 with Program Number 5704-FO2 processes programs containing the CALL INQCHK statement but the ROLLOUT/ROLLIN routine is replaced by a simple return-to-caller statement in subroutine INQCHK.

Inquiry Support Subprogram (Entry Name: SETINQ)

A FORTRAN program can be made an INQUIRY (I-TYPE) program by inserting a CALL SETINQ statement in the source program. The CALL SETINQ statement should be the first executable statement in the program. The SETINQ subprogram has no parameters.

Inquiry programs are usually used on systems with multiple program levels; however, they can be used on systems with only one program level. An inquiry program is in main storage and is executed when an inquiry request is made by one of the following procedures:

Model 6	Set the INQUIRY REQUEST switch to ON.
Models 10 and 12	Press the REQUEST key on the printer keyboard.
Model 15	Press the PA1 key.

For further information, see the appropriate operator's guide under *Related Publications* in the Preface.

Date/Time-of-Day Subprogram (Entry Name: CFTOD)

Note: The interval timer is available as a feature only on the Model 15. On the Model 10, the interval timer is available as a hardware RPQ (RPQ WE0922); and the CFTOD subprogram is available as a programming RPQ (RPQ 5799-WGY).

The CFTOD subprogram makes available the date and time-of-day. It is called by using the entry name CFTOD in a CALL statement. The source language statement is:

CALL CFTOD (date,time[,1])

where

date must be the name of a one-dimensional, three element, INTEGER*2 array

time must be (1) the name of a one-dimensional, three element, INTEGER*2 array if the third operand is omitted, or (2) the name of an INTEGER*4 variable if the third operand is included

1 is the constant 1. When this optional operand is coded, the time returned is in timer units. When it is omitted, the time returned is in hours, minutes, and seconds.

The date is incremented by one day at midnight. The date returned in the three-element array is the system date that was set at IPL time—either day, month, year (DDMMYY), or month, day, year (MMDDYY), depending upon the option chosen at system generation.

Examples are:

- For DDMMYY format March 14, 1974 would be returned as 140374.
- For MMDDYY format March 14, 1974 would be returned as 031474.

The time (when the third operand is omitted) is returned in hours, minutes, and seconds (HHMMSS) where the first element of the array contains the hours, the second element contains the minutes, and the last element contains the seconds. The values in the elements have the following range:

Element	From	To
Hours	00	23
Minutes	00	59
Seconds	00	59

Unless the third operand (,1) is coded, both the date and time are returned in the A2 format (two numeric characters per element).

If the third operand (,1) is coded, the time is returned as a four-byte unsigned binary number where each timer unit represents 3.33 milliseconds, or the low-order bit represents 3.33 milliseconds. An example is:

$(0000002A)_2$ represents $(42)_{10}$ timer units or (42 x 3.33) milliseconds.

##MAIN: The name given to a main program by the compiler if the main program was not given a name by the programmer.

alphabetic character: A character of the set A, B, C, . . ., Z, $.

alphameric character: A character of the set that includes the alphabetic characters and the numeric characters.

argument: A parameter passed between a calling program and a subprogram or statement function.

arithmetic expression: A combination of arithmetic operators and arithmetic primaries.

arithmetic operator: One of the symbols +, -, *, /, **, used to denote, respectively, addition, subtraction, multiplication, division, and exponentiation.

array: An ordered set of data items identified by a single name.

array element: A data item in an array, identified by the array name followed by a subscript indicating its position in the array.

array name: The name of an ordered set of data items.

assignment statement: An arithmetic variable or array element, followed by an equal sign (=), followed by an arithmetic expression.

basic real constant: A string of decimal digits containing a decimal point.

batched compilation: Sequential compilation of more than one program.

common area: A storage area that may be referred to by a calling program and one or more subprograms.

compilation time: The time during which a source program is compiled, that is, translated from a high level language to a machine language program.

compile: To prepare a machine language program from a computer program written in a higher level programming language.

compiler option statements: A set of statements used to provide control information regarding a compilation by the FORTRAN IV compiler.

constant: A fixed and unvarying quantity. The three classes of constants specify numbers (numeric constants), literal data (literal constants), and hexadecimal data (hexadecimal constants).

control program: A program that supervises the performance of a computing system; an example is the System/3 System Control Program.

control statement: Any of the several forms of GO TO, IF and DO statements, or the PAUSE, CONTINUE, and STOP statements, used to alter the normally sequential execution of FORTRAN statements, or to terminate the execution of the FORTRAN program.

core usage map: A linkage editor listing that shows the names and storage locations of routines that make up the load module.

data item: A constant, variable, or array element.

data type: The mathematical properties and internal representation of data and functions. The two types are integer and real.

device option statements: A group of statements, in the set of compiler option statements, used to define input/output devices.

direct-access file: A file from which records may be retrieved, or to which records may be written, in a nonsequential manner.

DO loop: Repetitive execution of the same statement or statements using a DO statement.

DO variable: A variable, specified in a DO statement, that is initialized or incremented before each execution of the statement or statements within a DO loop. It controls the number of times the statements within the DO loop are executed.

dummy argument: A variable in a FUNCTION or SUBROUTINE statement, or statement function definition, with which actual arguments from the calling program or function reference are associated.

executable program: A program that can be used as a self-contained procedure. It consists of a main program and, optionally, one or more subprograms or non-FORTRAN-defined external procedures or both.

executable statement: A statement that specifies action to be taken by the program; e.g., causes calculations to be performed, conditions to be tested, flow of control to be altered.

extended range of a DO statement: Those statements that are executed between the transfer out of the innermost DO of a completely nested set of DO statements and the transfer back into the range of this innermost DO.

external function: A function whose definition is not included in program unit that refers to it.

external procedure: A procedure subprogram or a procedure defined by means other than FORTRAN statements.

file: An ordered collection of one or more records.

formatted record: A record that is transmitted with the use of a FORMAT statement.

FUNCTION subprogram: An external function defined by FORTRAN statements, the first of which is a FUNCTION statement. It returns a value to the calling program unit at the point of reference.

global area: A data area at the load point of a main storage, which is not overlaid when different programs are loaded, thereby permitting communication between two or more programs.

halt code: A number assigned to a programmed halt; a machine instruction that stops the execution of a program.

hexadecimal constant: The character Z followed by a hexadecimal number, formed from the set 0 through 9 and A through F.

hierarchy of operations: Relative priority assigned to arithmetic or logical operations to be performed.

I/O list: A list of variables in an I/O statement, specifying the storage locations into which data is to be read or from which data is to be written.

implied DO: The use of an indexing specification similar to a DO statement (but without specifying the word DO and with a list of data elements, rather than a set of statements, as its range).

integer constant: A string of decimal digits containing no decimal point.

internal statement number: A number assigned to a source statement by the compiler for identification.

intrinsic function: A FORTRAN library function used by the compiler when certain operations are specified in a FORTRAN source statement.

ISN: Internal statement number.

job stream: The sequence of operation control statements and data submitted to the system on an input device.

length specification: An indication, by the use of the form *s, of the number of bytes to be occupied by a variable or array element.

link-editing: The combining, by the linkage editor, of a number of object or load modules for execution as one program.

linkage: Coding that connects two separately coded routines.

linkage editor: A program that combines separately produced object or load modules, resolves cross references between them, and generates overlay structures on request. The output of a linkage editor is called a load module, suitable for loading into main storage for execution.

linkage editor control statements: A set of statements used to provide control information regarding the link-editing process.

list-directed input/output: Input or output records that are transmitted without the use of a FORMAT statement; an asterisk is placed in the READ or WRITE statement where the FORMAT statement number would ordinarily be.

literal constant: A string of alphameric and/or special characters enclosed in apostrophes.

load module: An executable program that is the output of a linkage editor.

load module execution time: The time during which a load module is executed.

logical unit number: A constant or variable in an input/output statement, that specifies the file that is to be read from or written to.

looping: Repetitive execution of the same statement or statements, usually controlled by a DO statement.

main program: A program not containing a FUNCTION or SUBROUTINE statement and containing at least one executable statement. A main program is required for program execution.

map, compiler: A compiler listing that shows the names and storage locations of variables and statement numbers in the object module.

name: A string of from one through six alphameric characters, the first of which must be alphabetic, used to identify a variable, an array, a function, or a subroutine.

nested DO: A DO loop whose range is entirely contained by the range of another DO loop.

nonexecutable object program: An object module.

nonexecutable statement: A statement that describes the use or extent of the program unit, the characteristics of the operands, editing information, statement functions, or data arrangement.

numeric character: Any one of the set of characters 0, 1, 2, . . ., 9.

numeric constant: An integer or real constant.

object library: An area on disk that contains object modules and load modules.

object module: A nonexecutable module that is the output of a language translator such as the System/3 FORTRAN IV Compiler.

object program: A group of object modules that have been link edited together into an executable program; same as load module.

OCL: Operation control language.

predefined convention: The FORTRAN-defined type and length of a variable, based on the initial character of the variable name in the absence of any specification to the contrary. The characters I-N are INTEGER*4; the characters A-H, O-Z and $ are REAL*4.

procedure: A set of operation control statements in a source library that can be retrieved by specifying its name in an OCL CALL statement.

procedure subprogram: A FUNCTION or SUBROUTINE subprogram.

program unit: A main program or a subprogram.

range of a DO statement: Those statements that physically follow a DO statement, up to and including the statement specified by the DO statement as being the last to be executed in the DO loop.

real constant: A string of decimal digits that must have either a decimal point or an exponent, and can have both.

relational expression: An arithmetic expression, followed by a relational operator, followed by an arithmetic expression. The expression has the value true or false.

relational operator: Any of the set of operators that express an arithmetic condition that can be either true or false. The operators are: .GT., .GE., .LT., .LE., .EQ., .NE., and are defined as greater than, greater than or equal to, less than, less than or equal to, equal to, and not equal to, respectively.

scale factor: A specification in a FORMAT statement whereby the location of the decimal point in a real number (and, if there is no exponent, the magnitude of the number) can be changed.

sequential file: A file from which records are retrieved, or to which records are written, solely on the basis of their order.

severity code: A code assigned to a compilation error.

source library: An area on disk that contains procedures and source program statements.

source module listing: A compiler-generated listing that shows the source statements in a program unit and their corresponding internal statement numbers.

source program: A computer program written in a source language, such as a program written in FORTRAN.

specification statement: One of the set of statements that provide the compiler with information about the data used in the source program. In addition, the statement supplies information required to allocate storage for this data.

statement: The basic unit of a FORTRAN program, composed of a line or lines containing some combination of names, operators, constants, or words whose meaning is predefined to the FORTRAN compiler. Statements fall into two broad classes: executable and nonexecutable.

statement function: A function defined by a function definition within the program unit in which it is referred to.

statement function definition: A name, followed by a list of dummy arguments, followed by an equal sign (=), followed by an arithmetic expression.

statement function reference: A reference in an arithmetic expression to a previously defined statement function.

statement number: A number of from one through five decimal digits placed within columns 1 through 5 of the initial line of a statement. It is used to identify a statement uniquely, for the purpose of transferring control, defining a DO loop range, or referring to a FORMAT statement.

storage map: A compiler listing that shows the names and storage locations of variables and statement numbers in the object module.

subprogram: A program unit headed by a FUNCTION or SUBROUTINE statement.

SUBROUTINE subprogram: A subroutine consisting of FORTRAN statements, the first of which is a SUBROUTINE statement. It optionally returns one or more parameters to the calling program unit.

subscript: A subscript quantity or set of subscript quantities, enclosed in parentheses used in conjunction with an array name to identify a particular array element.

subscript quantity: A component of a subscript: a positive integer constant, integer variable, or expression that evaluates to a positive integer constant. If there is more than one subscript quantity in a subscript, the quantities must be separated by commas.

system input device: A device, such as a card reader or a printer/keyboard, used to read the job stream.

type declaration: The explicit specification of the type and, optionally, length of a variable or function by use of an explicit specification statement.

unformatted record: A record for which no FORMAT statement exists, and which is transmitted with a one-to-one correspondence between internal storage locations (bytes) and external positions in the record.

variable: A data item, identified by a symbolic name, that is not an array or array element.

FORTRAN SAMPLE PROGRAM

Figure 31 contains a sample System/3 FORTRAN program and the output listing resulting from its compilation, link editing, and execution. The sample program can be used to accomplish post-installation checkout of the compiler.

After program product installation, if the system uses 5444, this program can be run by placing the distribution disk cartridge on R1 (for systems using 3340, place the program product distribution data module on D1). Then, generate a call to the procedure FRTSMP using the following OCL statements:

```
// DATE 00/00/00
// CALL FRTSMP,R1
// RUN
```

You must have 10 tracks available for $SOURCE and 10 tracks available for $WORK on F1. The procedure assumes that F1 is named F1F1F1.

This test program consists of three groups of statements: a FORTRAN source module of a subprogram named KBINCO (used to compute entries in an array), a FORTRAN source module of a main program named SAMPLE (used to print out the array), and a single data statement that is used as input to the load module resulting from the compilation of the two FORTRAN source modules.

The sample program is written to be run in the minimum system configuration.

Figure 31 shows the output as it appears on the printer. Each page heading will display the current version number, modification number, modification number, date, and page number. Compiler output consists of:

1. The source module listing **A**
2. The compiler storage map **B**
3. Informative messages **C**

```
// CALL FRTSMP,R1
XX CALL PART01,R1
XX LOAD $FORT,F1
XX COMPILE SOURCE-KBINCO,UNIT-R1
XX FILE NAME-$WORK,UNIT-F1,PACK-F1F1F1,TRACKS-10,RETAIN-S
XX FILE NAME-$SOURCE,UNIT-F1,PACK-F1F1F1,TRACKS-10,RETAIN-S
XX RUN
// RUN
```

```
   FORTRAN IV VERXX/MODXX                                                      XX/XX/XX PAGE 001

*PROCESS  MAP
              C                                                                 01210000
              C      ***********************************************            01220000
              C      * KBINCO COMPUTES THE BINOMIAL COEFFICIENT.       *        01230000
              C      * C(N,K) = (N*(N-1)***(N-K+1))/(K*(K-1)***1).     *        01240000
              C      * WHERE N AND K ARE THE INTEGER ARGUMENTS TO      *        01242000
              C      * THE FUNCTION.  INTERMEDIATE CALCULATIONS ARE    *        01244000
              C      * PERFORMED IN REAL ARITHMETIC.  IN THE CASE      *        01246000
              C      * WHERE K .GT. N, A VALUE OF ZERO IS RETURNED.    *        01248000
              C      * THE VALUES OF N AND K ARE LEFT UNCHANGED.       *        01248400
              C      * THE FUNCTION HAS BEEN CHECKED FOR ALL COM-      *        01248800
              C      * BINATIONS OF N=1,2,...,20 AND K=1,2,...,10.     *        01249200
              C      ***********************************************            01249600
              C                                                                 01249700
      1              FUNCTION      KBINCO(N,K)                                  01250000
              CHECK FOR TRIVIAL CASES                                           01260000
      2              IF(       K .GT.N)   GOTO 50                               01300000
      3              IF(       K .EQ. 0)  GOTO 60                               01350000
      4              IF(       K .EQ. N)  GOTO 60                               01400000
      5              IF(       K .EQ. 1)  GOTO 70                               01450000
      6              IF(       N-K .EQ.1)      GOTO 70                          01500000
              CONVERT TO REAL FOR INT. CALCULATIONS                             01510000
      7              P    =    N                                                01550000
      8              Q    =    K                                                01600000
A             CHECK FOR LOWER 'DENOMINATOR'                                     01610000
      9              IF(       P-Q   .LT. Q)    Q     =     P-Q                 01650000
              CALCULATE DENOMINATOR                                             01660000
     10              MAX       =     Q                                          01700000
     11              BOT       =     1.0                                        01750000
     12              DO 30     I=2,MAX                                          01800000
     13              BOT       =     I      *     BOT                           01850000
     14        30    CONTINUE                                                   01900000
              COMPUTE NUMERATOR                                                 01910000
     15              MAX       =     P                                          01950000
     16              MIN       =     P      -     Q     +     1.0               02000000
     17              TOP       =     1.0                                        02050000
     18              DO 40     I=MIN,MAX                                        02100000
     19              TOP       =     I      *     TOP                           02110000
     20        40    CONTINUE                                                   02120000
              CALCULATE AND ROUND BIN. COEFF.                                   02130000
     21              KBINCO    =     TOP/BOT      +     0.5                     02150000
     22              RETURN                                                     02200000
     23        50    KBINCO    =     0                                          02250000
     24              RETURN                                                     02300000
     25        60    KBINCO    =     1                                          02350000
     26              RETURN                                                     02400000
     27        70    KBINCO    =     N                                          02450000
     28              RETURN                                                     02500000
     29              END                                                        02550000

 NAME     AT HEX1 DEC1   HEX2 DEC2   NAME     AT HEX1 DEC1   HEX2 DEC2   NAME     AT HEX1 DEC1   HEX2 DEC2
 KBINCO I    0006 00006              N        I  000A 00010              K        I  000E 00014
B P     R    0012 00018              Q        R  0016 00022              MAX      I  001A 00026
 BOT    R    001E 00030              I        I  0022 00034              MIN      I  0026 00038
 TOP    R    002A 00042

     000 TOTAL ERRORS FOR THIS COMPILATION
```

Figure 31 (Part 1 of 6). FORTRAN Sample Program

C

```
          STATEMENT ALLOCATIONS
   30   =0141  40   =0185  50   =0190  60   =01A9  70   =0185
0L105 I   THE CODE LENGTH OF KBINCO IS    449 DECIMAL.
0L103 I   TOTAL NUMBER OF LIBRARY SECTORS REQUIRED IS    5
          NAME-KBINCO.PACK-F1F1F1.UNIT-F1.RETAIN-T.LIBRARY-R.CATEGORY-020


XX CALL PART02,R1
XX LOAD $FORT,F1
XX COMPILE SOURCE-SAMPLE,UNIT-R1
XX FILE NAME-$WORK,UNIT-F1,PACK-F1F1F1,TRACKS-10,RETAIN-S
XX FILE NAME-$SOURCE,UNIT-F1,PACK-F1F1F1,TRACKS-10,RETAIN-S
XX RUN



*PROCESS  MAP
   1            PROGRAM  SAMPLE
          C    ******************************************************************00040000
          C    *                                                                *00040400
          C    * THIS PROGRAM IS A TEST CASE DESIGNED TO VERIFY THAT THE S/3    *00041200
          C    * FORTRAN IV COMPILER AND LIBRARY HAVE BEEN PROPERLY INSTALLED IN *00041600
          C    * YOUR SYSTEM.  THE REQUIRED OCL STATEMENTS ARE INCLUDED WITHIN  *00041700
          C    * THE CALLED PROCEEDURES.                                        *00041800
          C    *                                                                *00041900
          C    * THE PROGRAM GENERATES A TABLE OF BINOMIAL COEFFICIENTS WHICH IS *00056400
          C    * THEN PRINTED ON THE PRINTER.  ALL DATA IS PROGRAM GENERATED.   *00066400
          C    *                                                                *00070400
          C *                                                                   *00071000
          C *    THE OUTPUT SHOULD BE ---                                       *00081000
          C *                                                                   *00083000
          C *        I-----------------------------------------------------I   *00085000
          C *        I                          K                          I   *00085400
          C *   I----I-----------------------------------------------------I   *00090200
          C *   I  N I 1     2     3     4     5     6     7      8      9     10 I *00092200
          C *   I----I-----------------------------------------------------I   *00094200
          C *   I  1 I  1    0     0     0     0     0     0      0      0      0 I *00094600
          C *   I  2 I  2    1     0     0     0     0     0      0      0      0 I *00095000
          C *   I  3 I  3    3     1     0     0     0     0      0      0      0 I *00095100
          C *   I  4 I  4    6     4     1     0     0     0      0      0      0 I *00108800
          C *   I  5 I  5   10    10     5     1     0     0      0      0      0 I *00118800
          C *   I  6 I  6   15    20    15     6     1     0      0      0      0 I *00120800
          C *   I  7 I  7   21    35    35    21     7     1      0      0      0 I *00121200
          C *   I  8 I  8   28    56    70    56    28     8      1      0      0 I *00121600
          C *   I  9 I  9   36    84   126   126    84    36      9      1      0 I *00122000
          C *   I 10 I 10   45   120   210   252   210   120     45     10      1 I *00122400
          C *   I 11 I 11   55   165   330   462   462   330    165     55     11 I *00122500
          C *   I 12 I 12   66   220   495   792   924   792    495    220     66 I *00127100
          C *   I 13 I 13   78   286   715  1287  1716  1716   1287    715    286 I *00129100
          C *   I 14 I 14   91   364  1001  2002  3003  3432   3003   2002   1001 I *00131100
          C *   I 15 I 15  105   455  1365  3003  5005  6435   6435   5005   3003 I *00131500
          C *   I 16 I 16  120   560  1820  4368  8008 11440  12870  11440   8008 I *00131600
          C *   I 17 I 17  136   680  2380  6188 12376 19448  24310  24310  19448 I *00136200
          C *   I 18 I 18  153   816  3060  8568 18564 31824  43758  48620  43758 I *00138200
          C *   I 19 I 19  171   969  3876 11628 27132 50388  75582  92378  92378 I *00140200
          C *   I 20 I 20  190  1140  4845 15504 38760 77520 125970 167960 184756 I *00140600
          C *   I----I-----------------------------------------------------I   *00140700
          C *                                                                   *00152500
          C    ******************************************************************00162500
```

A

Figure 31 (Part 2 of 6). FORTRAN Sample Program

A

```
  2             DIMENSION     NBYK(20,10)                                          00164600
  3             INTEGER OUT                                                        00176400
  4             DATA      OUT/3/                                                          6
  5             DO 10     K=1,10                                                   00200000
  6             NBYK(1,K)      =      K                                            00250000
  7      10     CONTINUE                                                           00300000
  8             WRITE(OUT,2)                                                       00400006
  9             WRITE(OUT,3) (NBYK(1,K),K=1,10)                                    00450000
 10             DO 30     N=1,20                                                   00500000
 11             DO 20     K=1,10                                                   00550000
 12             NBYK(N,K)      =      KBINCO(N,K)                                  00600000
 13      20     CONTINUE                                                           00650000
 14             WRITE(OUT,4) N,(NBYK(N,K),K=1,10)                                  00700000
 15      30     CONTINUE                                                           00750000
```

```
FORTRAN IV VER05/MOD00                                                     09/11/76 PAGE 002
 16             WRITE(OUT,5)                                                       00800000
 17             STOP                                                               00850000
 18       2     FORMAT('1   S/3 FORTRAN IV SAMPLE TEST CASE'///////1X,'TEST INPUT,00900006
               1FOR S/3 SAMPLE PROGRAM'/////)                                     00910006
 19       3     FORMAT(T7,'I',57('-'),'I'/T7,'I',T35,'K',T65,'I'/T2,'I----I',     00950000
              *   57('-'),'I'/T2,'I  N I',I2,I4,2I5,3I6,3I7,'  I'/T2,            01000000
              *   'I----I',57('-'),'I')                                           01050000
 20       4     FORMAT(' I',I3,' I',I3,I4,2I5,3I6,3I7,' I')                        01100000
 21       5     FORMAT(' I----I',57('-'),'I')                                      01150000
 22             END                                                                01200000
```

B

```
NAME    AT HEX1 DEC1  HEX2 DEC2  NAME    AT HEX1 DEC1  HEX2 DEC2  NAME    AT HEX1 DEC1  HEX2 DEC2
OUT     I  0185 00389            NBYK    I  0189 00393 04A8 01192 K       I  04A9 01193
N       I  04AD 01197

   000 TOTAL ERRORS FOR THIS COMPILATION
```

C

```
     STATEMENT ALLOCATIONS
 5    =0561  4    =057A  3    =05BB  2    =0666  10   =06B0  20   =072C  30   =0764
```

Figure 31 (Part 3 of 6). FORTRAN Sample Program

```
                OVERLAY LINKAGE EDITOR CORE USAGE MAP

 START     CATEGORY   NAME AND        CODE LENGTH
ADDRESS               ENTRY      HEXADECIMAL  DECIMAL

 1800        128      SAMPLE         0781      1921
 1F79                   #UNITB
 1800                   #ERBUF
 1900                   #IOBUF
 1F81         0       $FOFO          0193       403
 20FF                   #MNTRY
 20B3                   #SNTRY
 20F2                   #RNTRY
 1F82                   #D
 2114         0       $FCF5          0023        35
 2125                   #RETRN
 2137         0       $FOB1          0062        98
 2168                   #DED4
 2189                   #DED70
 2199         0       $FOIO          00FD       253
 2230                   #FLST
 223C                   #FLST2
 2221                   #DERR
 220F                   #IOINT
 21FB                   #IOCOM
 2248                   #ENDEQ
 224F                   #ERREQ
 2260                   #OUTBL
 2256                   #INTBL
 226C                   #IO@@@
 226D                   #FLRP2
 2296         0       $FOVG          000C        12
 22A1                   #ADROT
 229F                   #RLIST
 22A2         0       $FOVH          0010        16
 22B1                   #DLIST
 22B2         4       $FCE3          0028        40
 22B9                   #XLT
 22BF                   #XST
 22C6                   #XA
 22CD                   #XMLT
 22D3                   #XMST
 22DA         4       $FOF6          001C        28
 22FF                   #BST
 22F1                   #BA
 22F8                   #BS
 22F6         4       $FOFD          00E1       225
 23A6                   #DOEND
 237A                   #DOBR
 2324                   #DBGN3
 23D7         4       $FOF2          0046        70
 241D         4       $FOEA          0154       340
 2420                   #RL
 2431                   #RST
 244A                   #RAO
 244D                   #RA
 243C                   #RSO
 243F                   #RS
```

Figure 31 (Part 4 of 6). FORTRAN Sample Program

```
START       CATEGORY   NAME  AND          CODE  LENGTH
ADDRESS                  ENTRY       HEXADECIMAL  DECIMAL

 2550                    #RTO
 2571          4       $FOLF              0083      131
 25E4          4       $FOEF              0084      132
 2678          4       $FOFH              0098      152
 2710          4       $FOVC              0061       97
 271F                    #FLOAT
 2771          4       $FOVA              006D      109
 27DE          4       $FOEC              0020       32
 27F4                    #ROFLW
 27F4                    #RUFLW
 27FF          5       $FOB2              0134      308
 2901                    #FRET
 2909                    #FOB2A
 2881                    #FOB2B
 2913                    #FOB2C
 2932          5       $FOC3              00AD      173
 29DF          5       $FOB8              004E       78
 2A2D          5       $FOB9              0018       24
 2A45          5       $FOBA              001D       29
 2A62          5       $FOCA              002B       43
 2A8D          6       $FO1C              00C2      194
 2B23                    #FRIST
 2B4F          6       $FO1B              006A      106
 2BB9          6       $$LPRT             00FB      251
 2CB4          6       $FO13              00E4      228
 2CBA                    #FO13A
 2D6B                    #FO13
 2D98          6       $FOVP              0019       25
 2DB1          6       $FOBB              006D      109
 2E1E          6       $FOD7              012C      300
 2F4A         20       KBINCO             01C1      449

OL100 I   THE TOTAL CORE USED BY SAMPLE IS  6411 DECIMAL.
OL101 I   THE START CONTROL ADDRESS OF THIS MODULE IS 1800.
OL104 I   TOTAL NUMBER OF LIBRARY SECTORS REQUIRED IS   27
          NAME-SAMPLE,PACK-F1F1F1,UNIT-F1,RETAIN-T,LIBRARY-0
```

Figure 31 (Part 5 of 6). FORTRAN Sample Program

```
XX CALL PART03.R1
XX LOAD SAMPLE.F1
XX RUN
```

S/3 FORTRAN IV SAMPLE TEST CASE

TEST INPUT.FOR S/3 SAMPLE PROGRAM

N	K 1	2	3	4	5	6	7	8	9	10
1	1	0	0	0	0	0	0	0	0	0
2	2	1	0	0	0	0	0	0	0	0
3	3	3	1	0	0	0	0	0	0	0
4	4	6	4	1	0	0	0	0	0	0
5	5	10	10	5	1	0	0	0	0	0
6	6	15	20	15	6	1	0	0	0	0
7	7	21	35	35	21	7	1	0	0	0
8	8	28	56	70	56	28	8	1	0	0
9	9	36	84	126	126	84	36	9	1	0
10	10	45	120	210	252	210	120	45	10	1
11	11	55	165	330	462	462	330	165	55	11
12	12	66	220	495	792	924	792	495	220	66
13	13	78	286	715	1287	1716	1716	1287	715	286
14	14	91	364	1001	2002	3003	3432	3003	2002	1001
15	15	105	455	1365	3003	5005	6435	6435	5005	3003
16	16	120	560	1820	4368	8008	11440	12870	11440	8008
17	17	136	680	2380	6188	12376	19448	24310	24310	19448
18	18	153	816	3060	8568	18564	31824	43758	48620	43758
19	19	171	969	3876	11628	27132	50388	75582	92378	92378
20	20	190	1140	4845	15504	38760	77520	125970	167960	184756

Figure 31 (Part 6 of 6). FORTRAN Sample Program

Compiler-generated messages indicate conditions encountered during program compilation. Usually they describe invalid uses of FORTRAN syntax, but can also relate to violations of System/3 requirements.

Error messages are listed here in order by message number and are described using the format:

1. Message number and text.

2. Explanation, which summarizes the condition(s) causing the message to be generated.

3. Severity code, which indicates the action taken by the compiler. Two severity codes are possible:

   - Severity 4—warning message. Compilation continues; the link-editing step and load module execution step are still permitted.

   - Severity 8—serious error. Compilation continues, but processing terminates at the end of compilation (the LINK, OBJECT, DECK, and GODECK options are ignored). The phrase *output text not generated* is used as a reminder that processing is terminated at the end of compilation.

   If the compiler makes an assumption or takes a special action because of one of the error messages, that assumption or action is described in the severity code section.

4. Programmer response, suggesting appropriate action which you should take. Corrective action should solve the problem; however, if the problem recurs, have the source program and associated source listing available before calling IBM for programming support.

01 NON-NUMERIC STMT NUMBER.

Explanation: The statement number contains a character that is not a decimal digit.

Severity: 8; output text not generated.

Programmer Response: Respecify the statement number using only decimal digits.

02 CONTINUATION ERROR.

Explanation: More than 19 continuation cards are specified, or a continuation card is out of sequence.

Severity: 8; output text not generated.

Programmer Response: Check for proper delimiters; respecify the statement so that it does not extend over more than 19 continuation cards.

03 SYNTAX ERROR

Explanation: A statement is found that does not conform to the rules for writing FORTRAN statements.

Severity: 8; output text not generated.

Programmer Response: Correct the statement.

04 INVALID STATEMENT.

Explanation: The statement is undeterminable, misspelled, or incorrectly formed.

Severity: 8; output text not generated.

Programmer Response: Correct the statement.

05 SEQUENCE ERROR.

Explanation: The statement is not in the proper order of statements for a FORTRAN program.

Severity: 8; output text not generated.

Programmer Response: Place the statement into proper sequence. For example, if the FUNCTION, SUBROUTINE, or PROGRAM statement appears, it must be the first statement in a program; the IMPLICIT statement must precede all other statements; specification statements must precede executable statements.

06 MISSING STMT NUMBER.

Explanation: A statement following a STOP, RETURN, GOTO, or arithmetic IF statement does not have a statement number.

Severity: 8; output text not generated.

Programmer Response: Supply a statement number where required.

07 INVALID NAME.

Explanation: A statement contains a variable or subprogram name longer than six characters or one that does not begin with an alphabetic character; or in a FUNCTION or SUBROUTINE statement, the name is missing.

Severity: 8; output text not generated.

Programmer Response: Supply or respecify the name, making sure it begins with an alphabetic character and is no more than six characters long.

08 SUBSCRIPT ERROR.

Explanation: A subscript is missing or incorrectly specified within dimensioned information.

Severity: 8; output text not generated.

Programmer Response: Correct the subscript.

09 DUPLICATE STMT NUMBER.

Explanation: The statement number is previously defined.

Severity: 8; output text not generated.

Programmer Response: Assign a different statement number.

10 DUPLICATE COMMON/GLOBAL.

Explanation: A variable in a COMMON or GLOBAL statement is previously defined in a COMMON or GLOBAL statement, or an attempt is made to initialize a COMMON or GLOBAL item.

Severity: 8; output text not generated.

Programmer Response: Assign a different variable name.

11 SUBPROGRAM IN COMMON.

Explanation: A subprogram name or dummy parameter appears in a COMMON or GLOBAL statement.

Severity: 8; output text not generated.

Programmer Response: Remove the subprogram from COMMON; respecify COMMON or GLOBAL statements without dummy arguments.

12 DUPLICATE SUBPROGRAM VAR.

Explanation: A variable name appears more than once in a SUBROUTINE or FUNCTION statement.

Severity: 8; output text not generated.

Programmer Response: Assign a different variable name.

13 INVALID FORMAT SYNTAX.

Explanation: The FORMAT statement contains a slash followed by a comma, or a comma followed by a slash.

Severity: 4; warning.

Programmer Response: Correct the statement if the program is to be compiled again.

14 SUBPROGRAM NAME ERROR.

Explanation: An attempt was made to dimension a subprogram name or type a SUBROUTINE name.

Severity: 8; output text not generated.

Programmer Response: Remove dimension information from subprogram names. If a name is to be typed, make the subprogram a function rather than a subroutine.

15 INVALID SPECIFICATION.

Explanation: A name is dimensioned more than once or explicitly typed more than once.

Severity: 8; output text not generated.

Programmer Response: Make sure type and dimension information is specified only once for a name; assign different names to different variables and arrays.

16 INVALID EXTERNAL NAME.

Explanation: A name specified in an EXTERNAL statement is a subprogram name, a dummy parameter, a COMMON or GLOBAL variable, a PROGRAM name, or is dimensioned.

Severity: 8; output text not generated.

Programmer Response: Correct any duplicate names; assign undimensioned names.

17 INVALID REAL CONSTANT.

Explanation: An invalid or incorrectly specified real constant is detected.

Severity: 8; output text not generated.

Programmer Response: Respecify as a valid real constant; that is, a number of decimal digits written with a decimal point optionally followed by a D or E and one or two digits for an exponent.

18 INVALID INTEGER CONSTANT.

Explanation: An invalid or incorrectly specified integer constant is detected.

Severity: 8; output text not generated.

Programmer Response: Respecify as a valid integer constant; that is, a number of decimal digits written without a decimal point that does not exceed the value 2147483647.

19 DUMMY ARGUMENT ERROR.

Explanation: More than 15 dummy arguments, or duplicate dummy arguments, appear in a statement function argument list, or more than 25 dummy arguments appear in a FUNCTION or SUBROUTINE statement.

Severity: 8; output text not generated.

Programmer Response: Correct any duplicate names; make sure the number of dummy arguments does not exceed 15 or 25.

20 SUBSCRIPTED VAR IN ASF

Explanation: A subscripted variable is specified in a statement function.

Severity: 8; output text not generated.

Programmer Response: Remove the subscript; if necessary, restructure the program to execute the proper variable.

21 SUBSCRIPT EXPRESSION ERR.

Explanation: An incorrectly formed subscript expression or undefined variable appears in a subscript expression.

Severity: 8; output text not generated.

Programmer Response: Correct the subscript expression; define all variables.

22 SUBSCRIPT DIMENSION ERR.

Explanation: The number of subscripts in a subscript expression, or the evaluated result of the subscript(s) does not agree with the dimension information, or the evaluated result of the subscript exceeds 32767.

Severity: 8; output text not generated.

Programmer Response: Respecify the expression or the dimensioned definition so that they are consistent.

23 ARITHMETIC STMT ERROR.

Explanation: An invalid arithmetic statement or variable is specified; or in a FUNCTION subprogram, the left side of an arithmetic statement is a dummy argument or is in COMMON.

Severity: 8; output text not generated.

Programmer Response: Correct the arithmetic statement.

24 INVALID IF EXPRESSION.

Explanation: An invalid or incorrectly formed IF expression is specified.

Severity: 8; output text not generated.

Programmer Response: Correct the IF expression.

25 INVALID SIMPLE ARGUMENT.

Explanation: An invalid simple argument appears in a CALL statement.

Severity: 8; output text not generated.

Programmer Response: Correct the CALL statement.

26 INVALID CALL EXPRESSION.

Explanation: An invalid expression appears in a CALL statement.

Severity: 8; output text not generated.

Programmer Response: Correct the CALL statement.

27 INVALID ASF LEFT SIDE.

Explanation: An invalid expression appears to the left of an equal sign in a statement function.

Severity: 8; output text not generated.

Programmer Response: Correct the statement function.

28 INVALID ASF RIGHT SIDE.

Explanation: An invalid expression appears to the right of an equal sign in a statement function.

Severity: 8; output text not generated.

Programmer Response: Correct the statement function.

29 INVALID STMT NUMBER.

Explanation: In an arithmetic IF, GOTO, or AT, or on the END= or ERR= parameter of an I/O statement, a statement number is missing, invalid, incorrectly placed, or is the number of a non-executable statement.

Severity: 8; output text not generated.

Programmer Response: Respecify the statement with a valid statement number.

30 FORMAT STMT NUMBER ERROR.

Explanation: A FORMAT statement number is missing or is incorrectly specified in a READ or WRITE statement, or a READ/WRITE statement containing an I/O list references a FORMAT statement not having a D, E, F, I, or A format specification.

Severity: 8; output text not generated.

Programmer Response: Respecify the statement with a valid statement number.

31 INPUT/OUTPUT LIST ERROR.

Explanation: A syntax error or an invalid list element appears in an input/output list; or in a FUNCTION subprogram, the input list element is a dummy argument or is in COMMON.

Severity: 8; output text not generated.

Programmer Response: Correct the input/output list.

32 COMPUTED GO INDEX ERROR.

Explanation: The index of a computed GO is missing, invalid, or not preceded by a comma.

Severity: 8; output text not generated.

Programmer Response: Correct the computed GO.

33 DO LOOP ERROR.

Explanation: The DO statement is incorrectly nested; or the terminal statement of the associated DO statement is a GOTO arithmetic IF, RETURN, FORMAT, STOP, PAUSE, or another DO.

Severity: 8; output text not generated.

Programmer Response: Supply a CONTINUE statement if possible; otherwise, restructure the DO loop.

34 TOO MANY NESTED DOS.

Explanation: There are more than 25 nested DO statements.

Severity: 8; output text not generated.

Programmer Response: Restructure the program wherever possible to contain fewer DO statements in a loop.

35 INVALID DO VALUE.

Explanation: The initial, test, or increment value in a DO statement or implied DO in a READ or WRITE I/O list is of the invalid type or the initial value is negative or zero.

Severity: 8; output text not generated.

Programmer Response: Respecify the DO statement initial value as an unsigned integer constant or variable greater than zero.

36 INVALID DO INDEX.

Explanation: The index of a DO or an implied DO is of invalid type, or, in a FUNCTION subprogram, the index of a DO is a dummy argument or is in COMMON or GLOBAL.

Severity: 8; output text not generated.

Programmer Response: Correct the DO statement.

37 INVALID LAST STATEMENT.

Explanation: The last executable statement before the END statement is not a STOP, GOTO, arithmetic IF, or RETURN statement.

Severity: 4; warning.

Programmer Response: Supply a STOP or other valid statement before the END statement.

38 NO END STATEMENT.

Explanation: The END statement is missing.

Severity: 4; warning.

Programmer Response: Supply an END statement if the program is to be compiled again.

39 STATEMENT TOO LONG.

Explanation: A statement is too long to be scanned because of compiler expansion of subscript expressions or addition of generated temporary storage locations.

Severity: 8; output text not generated.

Programmer Response: Restructure the statement into smaller logical units and specify as separate statements.

41 COMMON/GLOBAL EXTENSION.

Explanation: A variable is made equivalent to an element of an array in such a manner as to cause the array to extend beyond the origin of the COMMON or GLOBAL area.

Severity: 8; output text not generated. The compiler will restructure the EQUIVALENCE nest so that COMMON or GLOBAL is not extended. Note that this restructuring may cause apparent errors in other nests, generating error 42.

Programmer Response: Restructure equivalence wherever possible; include dummy arrays if necessary to make the COMMON area larger.

42 EQUIVALENCE ERROR.

Explanation: Two variables or array elements in COMMON or GLOBAL are equated, or the relative locations of two variables or array elements are assigned more than once.

Severity: 8; output text not generated.

Programmer Response: Restructure COMMON or GLOBAL variables and arrays, or correct EQUIVALENCE statements to eliminate double definition.

43 INVALID EQUIVALENCE NAME.

Explanation: A name that may not appear in an EQUIVALENCE list was detected in the list.

Severity: 8; output text not generated.

Programmer Response: Make sure that only *data* variable names are used in EQUIVALENCE lists.

44 RETURN ERROR.

Explanation: A RETURN statement is present in a main program.

Severity: 8; output text not generated.

Programmer Response: Remove the RETURN statement and supply a STOP statement.

45 MISSING DEFINE FILE.

Explanation: A DEFINE FILE statement is missing in a main program that contains a direct access READ, WRITE, or FIND statement.

Severity: 8; output text not generated.

Programmer Response: Supply a DEFINE FILE statement.

46 INVALID DEFINE FILE.

Explanation: A DEFINE FILE statement is present in a subprogram or is specified more than once for a file.

Severity: 8; output text not generated.

Programmer Response: Remove any extraneous DEFINE FILE statement; make sure duplicate DEFINE FILE statements are not specified.

47 TOO MANY ASFS.

Explanation: The number of arithmetic statement functions has exceeded the compiler's maximum of 20.

Severity: 8; output text not generated.

Programmer Response: Reduce the number of arithmetic statement functions, or subdivide the program into FUNCTION subprograms.

48 MISMATCH IN DATA.

Explanation: Names and constants in a DATA statement are not in a one-to-one correspondence.

Severity: 8; output text not generated.

Programmer Response: Respecify the DATA statement, deleting extraneous names or constants.

49 MIXED MODE IN DATA.

Explanation: A constant and variable in a DATA statement do not have the same type.

Severity: 8; output text not generated.

Programmer Response: Respecify the DATA statement so that typing is consistent.

50 INVALID H CONSTANT.

Explanation: An invalid literal constant is specified in a DATA statement.

Severity: 8; output text not generated.

Programmer Response: Respecify the DATA statement; check for valid delimiters.

51 INVALID HEX CONSTANT.

Explanation: An invalid hexadecimal constant is specified in a DATA statement.

Severity: 8; output text not generated.

Programmer Response: Respecify the DATA statement using only the characters 0 through 9, A through F, and blank, for hexadecimal characters.

52 SUBPROGRAM VAR IN DATA.

Explanation: A subprogram name or dummy variable is specified in DATA statement.

Severity: 8; output text not generated.

Programmer Response: Respecify the DATA statement, removing subprogram names or dummy variables.

53 COMMON NAME IN DATA.

Explanation: A COMMON or GLOBAL variable is loaded with a DATA specification.

Severity: 8; output text not generated.

Programmer Response: Respecify the DATA statement, deleting COMMON or GLOBAL variables.

54 OPTIONS ERROR.

Explanation: There was an options error in one or more compiler option statements. (Compiler option statements include device option statements, the CORE statement, the CATEGORY statement, and the *PROCESS statement.) Each statement in error is followed by the message **CONFLICTING OPTIONS OR INVALID SYNTAX**.

Severity: 8; output text not generated.

Programmer Response: Correct the statement(s).

55 MISSING FORMAT DELIMITER.

Explanation: A literal, X, or H type group in a FORMAT statement is not followed by a comma, slash, or right parenthesis.

Severity: 4; warning.

Programmer Response: Correct the statement.

56 RELATIONAL IF ERROR.

Explanation: A relational IF statement contains a DO statement, another relational IF statement, or a non executable statement.

Severity: 8; output text not generated.

Programmer Response: Correct the statement.

57 INVALID DEBUG VARIABLE.

Explanation: A variable name is misspelled, not defined, an EXTERNAL name, the name of a subprogram (not this function), a dummy argument, or an attempt is made to SUBCHK a scalar.

Severity: 4; warning.

Programmer Response: Correct the DEBUG statement.

58 INVALID DEBUG OPTION.

Explanation: A DEBUG option is specified incorrectly; or the NOTRACE option is specified with AT and/or TRACE statements in the DEBUG packet.

Severity: 8; output text not generated.

Programmer Response: Correct the DEBUG packet as necessary.

59 INVALID RECORD NUMBER.

Explanation: The record number expression in a READ, WRITE, or FIND statement does not evaluate to an *integer* value.

Severity: 8; output text not generated.

Programmer Response: Respecify the expression so that it is an integer value greater than zero.

60 PROGRAM NAME MISSING.

Explanation: No name is specified in the PROGRAM statement.

Severity: 8; output text not generated.

Programmer Response: Supply a program name.

61 INVALID DEBUG PACKET.

Explanation: The DEBUG packet contains statements other than AT and TRACE.

Severity: 8; output text not generated.

Programmer Response: Remove all but AT and TRACE statements from the packet.

62 NOT ENOUGH CORE.

Explanation: Total constant, variable, buffer, and IOB allocations exceed the indicated number of bytes:

Models 6, 10, and 12	62,720 bytes
Model 15 with Program Number 5704-FO1	49,152 bytes
Model 15 with Program Number 5704-FO2	57,344 bytes

For program 5704-FO2 only: The total external storage requirements (external buffers) exceeds 64K.

Severity: 8; output text not generated.

Programmer Response:
(one of the following):

- Segment the program into smaller subprograms that can be overlaid in an overlay environment.
- Separate the program into a number of smaller programs performing independent functions.
- Use either the SHRBUFF or EXTBUF (Model 15 with Program Number 5704-FO2 only) options for DAD files.

63 DATA NAME DOUBLY DEFINED.

Explanation: A variable in a DATA statement is previously defined.

Severity: 4; warning. The last definition is used.

Programmer Response: Respecify variables using different names; or remove previously defined variables from the DATA statement.

64 MISSING RIGHT PAREN.

Explanation: A required right parenthesis is missing.

Severity: 8; output text not generated.

Programmer Response: Supply required right parenthesis; or remove the extraneous left parenthesis.

65 COMPUTED GO TO ERROR.

Explanation: More than 60 statement labels are specified in the COMPUTED GO TO statement.

Severity: 8; output text not generated.

Programmer Response: Reduce the COMPUTED GO TO statement into two or more statements.

66 MISSING STMT NUMBER.

Explanation: The FORMAT statement does not have a statement number.

Severity: 8; output text not generated.

Programmer Response: Respecify the statement with a statement number.

67 MISSING LEFT PAREN.

Explanation: The FORMAT statement does not begin with a left parenthesis.

Severity: 8; output text not generated.

Programmer Response: Correct the statement.

68 INVALID SCALE FACTOR.

Explanation: The FORMAT statement specifies a scale factor on a data type other than D, E, or F, or no number or an invalid number precedes the scale factor, or the scale factor is greater than 127 or less than -127.

Severity: 8; output text not generated.

Programmer Response: Correct the statement.

69 FIELD TOO LARGE.

Explanation: The FORMAT statement contains a literal longer than 255 characters or a numeric specification greater than 255.

Severity: 8; output text not generated.

Programmer Response: Restructure the statement; segment large numeric specifications wherever possible.

70 UNBALANCED PARENTHESES.

Explanation: The FORMAT statement has a nesting level greater than 2, or an enclosed literal, or more code after the last right parenthesis, or too large an H specification, or the number of right parentheses is less than the number of left parentheses.

Severity: 8; output text not generated.

Programmer Response: Correct the statement.

71 INVALID FIELD WIDTH.

Explanation: The FORMAT statement contains a field width of zero, or in a D, E, or F specification, W is specified less than D, or a literal of length 0 is specified.

Severity: 8; output text not generated.

Programmer Response: Correct the statement.

72 MEANINGLESS NUMBER.

Explanation: The FORMAT statement contains a meaningless number (for example, a field or group count of zero), or a field count precedes a literal or T format specification.

Severity: 8; output text not generated.

Programmer Response: Correct the statement.

73 INVALID DELIMITER.

Explanation: The FORMAT statement has a delimiter other than a comma or slash, or there are two consecutive data items without an intervening delimiter, or two consecutive delimiters.

Severity: 8; output text not generated.

Programmer Response: Correct the statement.

74 MORE THAN 1 GENERIC STMT.

Explanation: The program contains more than one GENERIC statement.

Severity: 4; warning.

Programmer Response: Remove extraneous GENERIC statements.

75 INVALID CHARACTER.

Explanation: The FORMAT statement contains a character that is not a valid format character.

Severity: 8; output text not generated.

Programmer Response: Correct the statement.

76 INVALID $ NAME.

Explanation: A name beginning with $ is specified for a FUNCTION or SUBROUTINE subprogram.

Severity: 8; output text not generated.

Programmer Response: Rename the subprogram.

77 MISSING DEVICE OPTIONS.

Explanation: A sequential or direct access I/O statement appears without a matching device options statement specified, or a DEFINE FILE statement appears with no corresponding direct access device options statement.

Severity: 8; output text not generated.

Programmer Response: Specify the proper device options statements.

78 STATEMENTS FOLLOWING END.

Explanation: The END statement is not immediately followed by a /*.

Severity: 4; warning.

Programmer Response: Remove extraneous statements. Place the END statement after all statements in program execution.

79 DUPLICATE UNIT NUMBER.

Explanation: A logical unit number is used more than once.

Severity: 8; output text not generated.

Programmer Response: Redefine files so that each logical unit number defines one file only.

80 INVALID UNIT NUMBER.

Explanation: The logical unit number in a READ, WRITE, BACKSPACE, REWIND, END FILE, or FIND statement is of the wrong type or is zero.

Severity: 8; output text not generated.

Programmer Response: Make sure the logical unit number is valid.

81 TOO MANY NESTED CALLS.

Explanation: The program contains a statement which has more than 20 levels of nesting for functions.

Severity: 8, output text not generated.

Programmer Response: Correct the statement.

82 INCORRECT NUMBER OF ARGS.

Explanation: An incorrect number of arguments was specified for a function in a program containing a GENERIC statement, or more than 25 arguments are present in a CALL statement or function reference.

Severity: 8; output text not generated.

Programmer Response: Respecify the number of arguments to agree with the number required by the function, (for example, specify only one argument for SIN); or reduce the length of the argument list to 25.

83 PROGRAM SELF REFERENCE.

Explanation: The program name in the main program or subprogram is used as a subroutine name in a CALL statement or in a function reference.

Severity: 8; output text not generated.

Programmer Response: Respecify either the program name or parameter name to make each unique.

84 INVALID ARGUMENT TYPE.

Explanation: A subprogram contains:

1. An argument of the wrong type.
2. More than one argument not all of the same type.
3. An argument which was an array name.
4. An argument which was a name that appeared in an EXTERNAL statement.

Severity: 8; output text not generated.

Programmer Response: Correct the argument type.

85 INVALID SUBPGM REFERENCE.

Explanation: The subroutine name in a CALL statement or a function reference is in a COMMON, GLOBAL, or DATA statement or is dimensioned, or the subroutine name in a CALL statement is explicitly typed or is missing.

Severity: 8; output text not generated.

Programmer Response: Correct the subprogram.

86 NAME USED AS SUBPR, FUNCT.

Explanation: A variable name is used as both a subroutine and function name.

Severity: 8; output text not generated.

Programmer Response: Assign different names to functions and subroutines.

87 INCONSISTENT NO. OF ARGS.

Explanation: The number of arguments specified for a function in a program containing no GENERIC statement was inconsistent with FORTRAN rules.

Severity: 4; warning.

Programmer Response: If the function is meant to be user-supplied, assign a function name that does not duplicate a FORTRAN-supplied function name. If the function is meant to be FORTRAN-supplied, specify the correct number of arguments.

88 INCORRECT ARGUMENT.

Explanation: Without the GENERIC statement present, a non-intrinsic processor function contains:

1. An argument of the wrong type.
2. More than one argument not all of the same type.
3. An argument which was an array name.
4. An argument which was a name that appeared in an EXTERNAL statement.

Severity: 4; warning. The function is assumed to be user-supplied.

Programmer Response: Correct the argument.

89 DATA STMT ERROR IN ARRAY.

Explanation: A redefinition of a subsection of an array is attempted.

Severity: 4; warning.

Programmer Response: Review the DATA statement and correct redefinitions.

90 INCONSISTENT FUNCT REF.

Explanation: A non-intrinsic processor function that is either non-generic (or the GENERIC statement is not present) has a parameter list that does not agree with the original reference.

Severity: 8; output text not generated.

Programmer Response: Correct the parameter list or specify the proper function.

91 DIRECT ACCESS DEV OPTNS.

Explanation: Two or more conflicting options have been specified:

A. Two buffers were specified for a direct access file. SHRBUFF was indicated in the option statement; NOSHRBUFF is assumed and two buffers are used.

B. For Model 15 with Program Number 5704-FO2 only: SHRBUFF and EXTBUF have both been specified for a direct access file. NOSHRBUFF and EXTBUF are assumed.

Severity: 4; warning.

Programmer Response:

A. If NOSHRBUFF is desired, change *PROCESS statement to indicate NOSHRBUFF. If SHRBUFF is desired, change the DEVICE option statement to indicate one buffer.

B. If NOSHRBUFF is desired, change the *PROCESS statement to indicate NOSHRBUFF. If SHRBUFF is desired, change the *PROCESS statement to indicate NOEXTBUF.

92 SEQ DISK DEVICE OPTIONS.

Explanation: BLOCKSIZE was not 256 or a valid submultiple of 256 (128, 64, 32, 16). 256 is assumed as the BLOCKSIZE.

Severity: 4; warning.

Programmer Response: Specify a valid BLOCKSIZE.

93 DAD/DEFINE FILE MISMATCH.

Explanation: The number of define file statements does not equal the number of direct access device option statements, or the logical unit number(s) does not match.

Severity: 4; warning.

Programmer Response: Add the missing statement(s) or correct the statement(s).

97 STATEMENT NUMBER IS UNREFERENCED.

Explanation: The statement is not referred to anywhere in the program.

Severity: 4; warning.

Programmer Response: Remove extraneous statement numbers; correct statement numbers that may have been specified incorrectly.

98 UNDEF VAR NAME, INDICATED BY 'U' IN MAP.

Explanation: One or more variables are not defined. That is, the variable was not assigned a value in the program unit. The variable did not appear on the left of an equal sign (=), in a DATA statement, as a dummy argument in a SUBROUTINE or FUNCTION statement, or in the input/output list in a READ statement. The variable(s) are identified by a 'U' under the AT column of the storage map.

Severity: 4; warning.

Programmer Response: Remove extraneous variables; correct variable names that may have been specified incorrectly.

99 INTERNAL TEXT FORMAT EXCEEDS CORE SIZE.

Explanation: The compiled program is too large to be contained in working storage, and compilation cannot continue.

Severity: 8; compilation is terminated *immediately.*

Programmer Response: Reduce the size of the program by subdividing it into subprograms, or by eliminating unnecessary parts of the program (for example, reducing the number of CALL arguments by using COMMON where possible), or by increasing the partition size for the program, or by specifying SHRBUFF on the *PROCESS statement.

Index

$FOM7 exponentiation subprogram 184
$FOM8 exponentiation subprogram 185
$FOM9 exponentiation subprogram 185
$FOMA exponentiation subprogram 186
$FOMB exponentiation subprogram 186
$FOMC exponentiation subprogram 187
$FOMD exponentiation subprogram 188
$FORT compiler name, specified on LOAD statement 86
$OLINK, linkage editor name 105, 106
$SOURCE file 105, 106
  assigning storage location for 145
$WORK file 105, 106
  assigning storage location for 145
*, to define list-directed data 42
*EXTENDED PRECISION 1130 statement 154
*IOCS 1130 statement 152
*ONE WORD INTEGERS 1130 statement 154
*PROCESS compiler option statement 98
/, use in FORMAT statement 29
##MAIN load module name 107
  definition 197

A, use in FORMAT statement 34
ABS generic function name 67
  summary of function 170
absolute value functions, summarized 170
access methods, input/output 41
addition, operation symbol for 9
address recall register (ARR) 148
address/data switch subprogram (DATSW) 192, 193
AINT generic function name 67
  summarized 170
algorithms, of library functions 169
aliases, for function names 67
ALOG, ALOG10 functions
  accuracy of 172
  algorithm for 175
  error conditions tested 194
  summarized 171
alphabetic character, definition 197
alphameric
  character, definition 197
  data 34, 35
AMAX0, AMAX1 functions 170
AMIN0, AMIN1 functions 170
AMOD function 170
appendix 201

apostrophe
  used in device option statements 94
  used in direct-access input/output statements 46, 48
  used in literal data definition 12, 35
arctangent functions
  accuracy of 172
  algorithms for 178, 180
  summarized 171
arguments
  definition 197
  dummy 64
  in COMMON 53
  passed to subroutines, differences between System/3 and 1130 154
arithmetic assignment statements 15
  mode of 17
  summarized 159
  type and length of result 16
arithmetic expressions 15
  definition 197
  hierarchy of operations (order of operations) 17
  parentheses in 17
  programming considerations for 139
  rules for constructing 15
  type and length of result 16
arithmetic IF statement 21
  summarized 159
arithmetic operator, defined 197
ARR (address recall register) 148
arrays 13
  arrangement in storage 14
  debug facility to check subscripts 73
  defining in specification statements 52
  differences between System/3 and 1130 154
  equivalencing 56
  programming considerations for 139
assembler language for standard linkage between modules 146
assignment statements, arithmetic 15
  mode of 17
  summarized 159
  type and length of result 16
associated variable 45
  differences between System/3 and 1130 153
  sharing of 131
AT debug statement 72
  summarized 160
ATAN function
  accuracy of 172
  algorithm for 178
  summarized 171
automatic function selection 66
  GENERIC statement 66

BACKSPACE statement 43
  summarized 160
basic real constant 10
  definition 197
batched compilation 102
  definition 197
  example of 103
  order of program units 102
BCD compiler option 99
  differences between System/3 and 1130 152
blank character in FORTRAN program 7
  affect on list-directed data 40
  affect on numeric data 32
blank fields in a record 36
block length of files, specifying in device option statements 95
BLOCKSIZE parameter in device option statements 95
buffers
  direct-access 132
  double buffering (Model 15) 150
  sequential 137
  sharing of 134

C, to define comments in FORTRAN 7
calculations
  floating point 9
  integer 9
CALL LINK 1130 statement 153
CALL operation control statement
  examples of
    in compilation 103
    in link editing 105
CALL statement (FORTRAN) 63
  programming considerations for 140
  summarized 160
card devices, double buffering (Model 15) 150
carriage control characters 39
CATEGORY compiler options statement 93, 98
CFTOD subprogram 196
CLEAR parameter (Model 15) 96
coding form, FORTRAN 6
comma, separator in list-directed data 40
comments, in FORTRAN statements 7
commercial subroutines, differences between System/3 and 1130 155
common block
  (see also COMMON statement)
  arguments in 53
  conserving space with 140
  definition 197
  identified on compiler storage map 109
  relationship to EQUIVALENCE statement 57
COMMON entry in object module records 113
COMMON statement 53
  (see also common block)
  contrasted with GLOBAL statement 54
  relationship of EQUIVALENCE statement 57
  summarized 160
  use in converting 1130 programs 154
compilation
  batched 102
  defining FORTRAN files for 89
  description of compile step 93
  messages 115
  relationship of FORTRAN processing 3, 83
compilation time, definition 197
compile step 93
  compiler option statements 93
  job stream 103
  output from 111
compiled sample program 201
compiler
  calling with $FORT name 86
  files for 88
  output from 111
    compiler option output 117
    messages 113, 209
    object module 83, 113
    storage map 88, 119
  post installation checkout of 201
    sample program output 202
compiler map 117
  definition 197
compiler messages 209
  described 115
compiler option statements
  *PROCESS 98
  CATEGORY 98
  combining with load module OCL statements 108
  combining with operation control statements 102
  CORE 97
  device option statements (READ, PRINT, NOPRINTER, PUNCH, DAD40, DAD44, DAD45, SEQ40, SEQ44, SEQ45, TAPE) 94
  placement in job stream 93
compiler options 98
  output from
    in compile step 117
    in link edit step 119
compiler output
  compiler option output 117
  messages 115, 209
  object module 113
  sample of 117
  storage map 88, 117
  summarized 88
computed GO TO statement 19
  summarized 161
consecutive processing of direct-access file 132
console display panel dial setting 148
  Model 6 algorithm 25

constant 9
definition 197
hexadecimal 11
definition 198
in arithmetic expression 15
integer 10
definition 198
literal 12
definition 199
numeric, definition 199
real 10
definition 199
summarized 9
CONTINUE statement 23
summarized 157
continuing FORTRAN statements 7
control characters, carriage 39
control program 83
definition 197
control statements, FORTRAN 19
definition 197
programming considerations with 139
control statements, linkage editor 106
conversion codes (see format codes)
conversion functions, error conditions tested 194
CORE compiler option statement 97
to specify storage for more than one main program 140
core usage map
definition 197
described 121
sample 122
COS function
accuracy of 172
algorithm for 177
as a generic name 67
error conditions tested 194
summarized 171
cosine and sine functions
accuracy of 172
algorithm for 177
summarized 171
CRT/keyboard (3277 display station) 149

D, use in format statement 29, 32
DABS function, summarized 170
DAD40, DAD44, DAD45 device option statements 95
data item, definition 197
DATA statement 58
in the order of a FORTRAN program 8
summarized 161
data type
definition 197
equivalencing 56
DATAN function
accuracy of 172
algorithm for 180
summary of 171
date/time-of-day subprogram (CFTOD) 196
DATSW subprogram 192
DBLE function, summarized 170
DCOS function
accuracy of 172
algorithm for 179
error conditions tested 194
summarized 171
debug facility 73, 141
AT statement 72
debug packet, definition of 71
DEBUG statement 71
examples of 73
separating output from other program output 140
TRACE OFF statement 73
TRACE ON statement 72
DEBUG statement 71
summarized 161
DECK compiler option 99, 105
object module from 113, 118
DEFINE FILE statement 45
buffer assignment of files 132
differences between System/3 1130 153
example of use 45
summarized 162
use in converting 1130 program 153
defining files 88
at compilation time 88
at execution time 89
for load module 88
use of logical unit numbers when 91
device option statements 94
compared with 1130 *IOCS statement 152
DAD40, DAD44, DAD45 95
definition 197
NOPRINTER 94
placement in job stream 93
PRINT 94
PUNCH 95
READ 94
SEQ40, SEQ44, SEQ45 96
TAPE 97
device, input/output
differences between System/3 and 1130 151
logical unit numbers for 90
1403 printer 90, 94
1442 card read punch 90, 94
2222 printer 90, 94
2501 card reader 92, 94
2560 MFCM 92, 94
3277 display station 92, 94, 149
3284 printer 92, 94
3410/3411 magnetic tape subsystem 90
5203 printer 90, 91, 94
5213 printer 90, 94
5406 console keyboard 90, 94
5424 MFCU 90, 94
5444, 5445 disk storage drive 90
5471 printer/keyboard 90, 91, 94
5496 data recorder 90, 94

DEXP function
accuracy of 172
algorithm for 174
summarized 171
diagnostic messages
compiler 115, 209
linkage editor 119
dial setting, console display panel 148
Model 6 algorithm 25
DIM function 170
as a generic name 67
DIMENSION statement 53
summarized 162
direct-access file 41
associated variable, sharing of 131
buffer assignment 132
consecutive processing 132
definition 197
minimizing I/O time 132
programming considerations for 131
specifying on device option statements 95
direct-access I/O statements 45
DEFINE FILE 45
summarized 162
FIND 48
summarized 163
general example of 49
programming considerations for 131
READ 46
summarized 166
WRITE 47
summarized 168
direct-access programming considerations 131
associated variable, sharing of 131
buffer assignment 132
consecutive processing 132
minimizing input/output time 132
disk drive (5444 and 5445) 90, 92
disk file
definition
at compilation time 88
at execution time 89
logical unit assignment for 90
specifying on a device option statement 96
display panel, console 148
divide check subprogram (DVCHK) 189
division, operation symbol for 9
DLOG, DLOG10 function
accuracy of 172
algorithm for 176
error condition tested for 194
summarized 171
DMAX1 function 170
DMIN1 function 170
DMOD function 170
DO loop, programming considerations for 139
DO statement 21
examples of 23
extended range of 22
looping with 21
nested DOs 22
rules governing use of 22
summarized 162
DO, nested 22
definition 195
DO-type notation, in I/O list 38
double precision
data 9
format code 31, 32
scale factor in 34
DSIN function
accuracy of 172
algorithm for 179
error condition tested 194
summarized 171
DSQRT function
accuracy of 171
algorithm for 182
error condition tested 194
summarized 171
DTANH function
accuracy of 172
algorithm for 183
summarized 171
dual programming environment 149
dummy argument 64
definition 198
specifying
in function definition statement 61
in SUBROUTINE statement 167
dump subprograms (DUMP/PDUMP) 190
output from 94
separating from other program output 140

E
use in DEFINE FILE statement 45
use in FORMAT statement 29, 32
EBCDIC compiler option 99
end execution subprogram (EXIT) 190
END FILE statement 43
sequence of 137
summarized 163
END parameter, in READ statement 42
end record
in load module 119
in object module 114
END statement (FORTRAN) 27
summarized 163
end-of-file, END parameter to control 42
EQ relational operator 20

EQUIVALENCE statement 56
  relationship of COMMON and GLOBAL statements 56
  summarized 163
  use in converting 1130 programs 154
ERR parameter in READ statement 42, 46
error conditions, tested in FCTST subprogram 194
error messages
  compiler 115, 209
  halt codes for 120
  linkage editor 119
  load module 120
error subprogram (FCTST) 193
error traceback 124
  directed to a particular printer 95
ESL records, in object module 113
exceptions program, in mathematical functions 173
executable program, definition 198
executable statements 5
  definition 198
  order in a FORTRAN program 8
execution time 107
  (see also load module execution step)
execution, load module 107
EXIT subprogram 190
EXP function
  accuracy of 172
  algorithm for 174
  summarized 171
explicit specification statements 52
  summarized 163
exponential functions
  accuracy of 172
  algorithms for 174
  summarized 171
exponentiation
  operation symbol for 9
exponentiation functions
  error conditions tested 194
  implicitly invoked 184
expression, relational, definition 199
expressions, arithmetic
  definition 197
  programming considerations for 139
EXTENDED PRECISION 1130 statement 154
extended range of DO statement 22
  definition 198
external functions, mathematical
  accuracy of 172
  algorithms for 169
  definition 198
  summarized 171
EXTERNAL statement 65
  relationship of GENERIC statement 66
  specifying library functions with 169
  summarized 163
EXTRN reference
  in ESL records 113

F, use in FORMAT statement 29, 32
FCTST subprogram 193
file
  $SOURCE 105, 145
  $WORK 105, 145
  buffer assignment for 132
  defining a 88
  definition 198
  direct-access 41, 131
  linkage editor 88
  load module 88
  name of 89
  relationship of file name and logical unit number 89
  sequential 40
  specifying on device option statements 97
FILE operation control statement
  for multifile tape volumes (Model 15 only) 98
  in load module execution 89
FIND statement 48
  summarized 163
fixed arithmetic functions
  summarized 170
fixed-length records 135
FLOAT function 170
floating point
  (see also real)
  calculations 9
format codes
  for alphameric data 34
  for numeric data 31
format control codes
  in DEFINE FILE statement 45
format of input/output 29
FORMAT statement 29
  delimiters in 30
  summarized 164
  use of parentheses in 30
  use of slashes in 30
formatted record
  definition 198
  examples of 42
  relationship of FORMAT statement 41
forms control
  differences between System/3 and 1130 154
FORTB procedure 87
FORTG procedure 87
FORTL procedure 87
  example of 105
FORTN procedure 87
FORTRAN
  coding form 6
  compiler
    messages 209
    post-installation, checkout of 201
  file
    definition 88
  language
    list of elements of 8
    summarized 5
  library
    functions 169
    subprograms 177

FORTRAN (continued)
operation symbols, list of 9
processing
overview of 83
traceback listing 124
program
called by assembler language module 146
calling an assembler language module 147
order of statements in 8
sample of 75
statements
classes of 5
coding of 6
functions of 5
order of 8
use of blanks in 7
steps in using 84
traceback listing 124
FTnnnnn to name FORTRAN files 89
function definition statement 60
summary of 164
function error subprogram (FCTST) 193
FUNCTION statement 61
(see also statement function)
declaration of type 59
relationship of statement functions 59
summarized 164
FUNCTION subprogram 61
definition 198
dummy arguments in 64
RETURN statement in 64

GE relational operator 20
generic function names 67
GENERIC statement 66
summarized 164
global area
(see also GLOBAL statement)
conserving space with 140
definition 198
identified on storage map 119
relationship of EQUIVALENCE statement 56
GLOBAL entry in object module records 113
GLOBAL statement 70
(see also global area)
contrasted with COMMON statement 53
relationship of EQUIVALENCE statement 56
summarized 164
GO TO statements
computed GO TO 19
summarized 161
restrictions with DO statement 22
unconditional GO TO 19
summarized 165
GODECK compiler option 99
linkage editor processing of 105
output from 120
GOSTMT compiler option 99
GT relational operator 20

H, use in FORMAT statement 35
halt code
conversion algorithm 25
definition 198
displaying for PAUSE and STOP statements 24
for linkage editor messages 120
for load module messages 124
header record in load module 119
hexadecimal constant 11
definition 198
hierarchy of operations 17
hyperbolic tangent functions
accuracy of 172
algorithm for 183
summary of 171

I/O list, definition 198
I, use in FORMAT statement 29, 31
IABS function 170
IAR (instruction address register) 148
IDIM function 170
IDINT function 170
IF statement
arithmetic IF 21
summarized 159
relational IF 20
programming considerations for 139
summarized 167
restrictions with DO statement 22
use in looping 23
IFIX function 170
IMPLICIT statement 51
compared to 1130 ONE WORD INTEGERS and EXTENDED PRECISION statements 154
restrictions in SUBROUTINE subprogram 62
summarized 165
implicitly invoked exponentiation subprograms 184
implied DO notation 38
definition 198
example of 38
increment, in DO statement 21, 162
index registers (XR registers) 148
indexing parameters
in DO statement 21, 162
in input/output lists 37
indicator test subprograms 189
informative messages
compiler 115
linkage editor 119
INIT debug option 71
example of output 141
initial value, in DO statement 21, 162
input, mixing BCD and EBCDIC 99
input error, ERR parameter to control 42

input/output statements 41
direct-access 45
FORMAT 29
list-directed 40
lists in 37
relationship to files 41
sequential 42
tape 43
INQCHK subroutine 195
instruction address register (IAR) 148
INT function 170
integer
calculations 9
constants 10
data 29
typing in specification statements 52
INTEGER statement 52
rules for 52
summarized 165
internal statement number (ISN) 99, 116
definition 198
interprogram communication 69
GLOBAL statement 70
INVOKE statement 70
PROGRAM statement 69
intrinsic functions
algorithms for 169
definition 198
generic names for 67
summarized 170
INVOKE statement 70
example of use in job stream 140
relationship of 1130 CALL LINK statement 153
summarized 165
IOCS 1130 statement 152
ISIGN function, summarized 170
ISN (internal statement number) 99, 116
definition 198

job
described 83
output
compiler 111
linkage editor 119
load module 124
job stream, example of 86
example of in combining operation control statements and compiler option statements 102
examples of when using PROGRAM and INVOKE statements 140

L, use in DEFINE FILE statement 45
language elements, summarized 8
LE relational operator 20
length specification of variables 51
differences between System/3 and 1130 154
library
FORTRAN
compiler options to store modules into 86
description of
functions 169
service subprograms 189
utility subprograms 190
differences between System/3 and 1130 153
priority value of 93
system 88
library function error subprogram (FCTST) 193
LINK compiler option 100, 105
link edit step
compiler input to 105
files needed for 105
output from 119
relationship of FORTRAN processing 84
linkage between modules 148
linkage editor
calling with FORTL procedure 105
calling with operation control statements 105
definition 198
description of link edit step 105
output 86, 119
overlay feature 106
linkage editor control statements, examples of 106
list-directed data 40
definition 199
field width 40
relationship of FORMAT statement 42
specifying in input/output statements 42
lists in input/output statements 37
indexing (implied DO notation) in 38
specifying in READ statement 42
literal
constant 12
data 35
definition 199
load module
compared to object program 83
contents of 119
definition 199
GODECK option to punch 98
LINK option to store in object library 86
name 107
size of 97
load module execution step 107
definition 199
files needed for 107
operation control statements for 107
combining with compile step statements 108
output from 124
program data in 108
relationship to FORTRAN processing 84

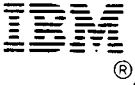
LOG, LOG10 functions
    accuracy of   171
    algorithms for   175
    summarized   171
logarithmic functions
    accuracy of   171
    algorithms for   175
    summarized   171
logical unit number   89
    definition   199
    differences between System/3 and 1130   151
    relationship to device   41
    relationship to device option statement   88
    use in FORTRAN input/output statements   42
looping   21
    definition   199
LT relational operator   20

machine indicator test subprograms   189
magnetic tape
    device option statement for   93, 96
    input/output statements
        BACKSPACE   43, 160
        END FILE   43, 137, 163
        REWIND   44, 167
magnitude
    of double precision constants   10
    of integer constants   10
    of single precision constants   10
main program, definition   199
map
    compiler   88, 117
    core usage   121
    definition   199
    MAP compiler option to specify   99
MAP compiler option   99
    compiler map generated   116, 118
    core usage map generated   121
mathematical functions
    external   171, 174
    intrinsic   170
    program exceptions, control of   173
    rules for coding   173
MAX functions   170
maximum value functions, summarized   170
messages
    compiler   115, 209
    linkage editor   119
    load module   124
MFCM device (2560 multi-function card machine)
    logical unit number for   92
    specifying
        on PRINT device option statement   94
        on PUNCH device option statement   95
        on READ device option statement   94
MFCU device (5424 multi-function card unit)
    differences between System/3 and 1130   151
    logical unit number for   90
    specifying
        on PRINT device option statement   94
        on PUNCH device option statement   95
        on READ device option statement   94
MIN, MIN0, MIN1 functions   170
minimax, method of during mathematical approximations   1[illegible]
minimum value functions   170
minus character, use in expression   9
MOD function   170
    as a generic name   67
module (see load module; object module)
modulo arithmetic function, summarized   170
multi-function card unit device (see MFCU)
multidimensional arrays, example of   13
multifile tape processing (Model 15 only)   137
    FILE statement for   97
multiplication, operation symbol for   9
multiprogramming partitions   145

names in program
    definition   199
    variable   12
natural logarithmic functions
    accuracy of   171
    algorithms for   174
    summarized   171
NE relational operator   20
negative zero, representation of, in System/3 and 1130   155
nested DO   22
    definition   199
NODECK compiler option   99
NOGODECK compiler option   99
NOGOSTMT compiler option   99
NOLINK compiler option   100
NOMAP compiler option   99
non-executable object program
    (see also object module)
    definition   199
non-executable statements   5
    definition   199
NOOBJECT compiler option   100
NOPRINTER device option statement   94
normal exit of DO loop   22
NOSHRBUFF compiler option   99
NOSOURCE compiler option   99
null item, in list-directed data   40
numeric character   199
numeric constant   199
numeric data
    (see also integer data; real data)
    FORMAT codes for   31

OBJECT compiler option 100
  example of 142
  linkage editor processing of 105
object library
  contents of 88
  definition 199
  LINK compiler option to store load modules 100
  OBJECT compiler option to store object modules 100
object module
  card deck of 118
  contents of
    end record 114
    ESL records 113
    RLD records 113
  definition 199
  options
    DECK to punch 99
    OBJECT to store into library 100
object program
  (see also load module)
  definition 199
OCL (see operation control language)
ONE WORD INTEGERS 1130 statement 154
operation control language (OCL)
  combining with compiler option statements 102
  combining with linkage editor control statements 106
  definition 199
  in job processing
    in compile step 102
    in link edit step 105
    in load module execution step 107
  relationship to FORTRAN processing 86
operation symbols, list of 9
operator
  arithmetic 197
  relational 20, 199
options, compiler 97
order of a FORTRAN program 8
order of operations, in arithmetic expressions 17
output
  compiler 113
  directing to both printer and punch 141
  linkage editor 88, 119
  load module 124
  separating debug or dump output 140
OVERFL subprogram 189
overflow exceptions 173
overflow indicator subprogram (OVERFL) 189
overlay, linkage editor 106
P, scale factor code (in FORMAT statement) 29, 34
parameter list, for standard linkage 148
parentheses
  use in EQUIVALENCE statement 56
  use in expressions 17
  use in FORMAT statement 30
  use in subprograms 64
passing arrays, differences between System/3 and 1130 154
PAUSE algorithm 25
PAUSE statement 24
  halt code for 24, 124
    conversion from Models 10 and 15 to Model 6 25
  response to 24
  summarized 165
PDUMP/DUMP subprograms 190
  separating from other program output 140
plus character, use in expressions 9
positive difference functions 170
precision increase function 170
predefined specification of variables 13, 199
PRINT device option statement 94
  directing output to a card punch 141
printer, input output 90, 94
  carriage control characters for 39
procedure subprogram 199
procedures, FORTRAN-supplied
  definition 199
  list of 87
  use of 86
PROCESS compiler option statement 98
  function of 93
  placement in job stream 93
processing, FORTRAN 3
  overview of 83
program data 108
program exception in mathematical functions 173
program output (see output)
program processing 83
PROGRAM statement 69
  example of use in job stream 140
  relationship to 1130 CALL LINK statement 152
  summarized 166
  use of CORE compiler option 140
program unit, definition 199
program, FORTRAN
  order of statements in 8
  samples of 75, 201
programming considerations
  direct-access 131
  sequential disk and tape 135
pseudo sense light subprograms (SLITE/SLITET) 189
publications, related v
PUNCH device option statement 95
  to obtain printed cards 141
punching and reading the same card 152

range of DO loop 22
definition 199
READ device option statement 94
READ statement 46
direct-access 46
summarized 166
lists in 37
sequential 42, 137
summarized 166
reading and punching the same card 152
real (floating point)
arguments 170
calculations 9
constants 10
data
format codes for 31
scale factor in 34
specification of type
in FORMAT statement 31
in specification statements 51
REAL statement 52
rules for 52
summarized 166
record
fixed length 135
format codes for 35
formatted 41
length of, assigning buffer for 132
list-directed 41
load module 119
object module 113
size of, in direct-access file 45
unformatted 41
variable spanned 135
related publications v
relational expression, definition 199
relational IF statement 20
programming considerations for 139
summarized 167
relational operators, in relational IF statement 20
relative record number, in direct-access statements 46
relocation record in load module 119
repeat factor, in input/output statements 40
RETURN statement 64
summarized 167
REWIND statement 42
summarized 167
RLD records, in object module 113
rounding, differences between System/3 and 1130 154
routine (see object module)
sample FORTRAN programs
examples of coding 75
post installation checkout 201
save area 148
scale factor 34
definition 199
sense light subprograms 189
sequential files
assigning buffers for 137
definition 200
multi-file tape processing (Model 15 only) 137
programming considerations 135
sequential input/output statements 41
order of 137
SEQ40, SEQ44, SEQ45 device option statements 96
service subprograms 189
SENTINQ subroutine 195
severity code 88, 116
definition 200
SHRBUFF compiler option 99
sharing buffers 134
SIGN function 170
sign transfer functions 170
SIN function
accuracy of 172
algorithm for 177
as a generic name 67
error condition tested 194
summarized 171
sine and cosine functions
accuracy of 172
algorithms for 177
summarized 171
single precision
constant 10
conversion code 32
data 9
scale factor in 34
SIZE parameter in CORE compiler option statement 97
SLITE/SLITET subprogram 189
SNGL function 170
SOURCE compiler option 98
listing generated from, sample 117
source library 88
definition 200
source module listing
definition 200
sample 117
source program, definition 200
specification statements 51
COMMON 53, 160
(see also common block)
definition 200
DIMENSION 53, 162
EQUIVALENCE 56, 163
explicit specification statement 52, 163
IMPLICIT 51, 165
split screen support (CRT/keyboard) 149
spooled environment 149



Technical Newsletter

This Newsletter No. SN21-5634
Date 29 September 1978

Base Publication No. SC28-6874-3
File No. S3-25

Previous Newsletters SN21-5568

IBM System/3
FORTRAN IV
Reference Manual

©IBM Corp. 1972, 1974, 1976

This technical newsletter applies to version 03, modification 00 of IBM System/3 Model 15 FORTRAN (Program 5704-FO2) and also applies to the current versions and modifications of the System/3 programs listed in the edition notice. It provides replacement pages for the subject publication. These replacement pages remain in effect for subsequent versions and modifications unless specifically altered. Pages to be inserted and/or removed are:

Cover, Edition Notice
v through x
13, 14
67, 68
97, 98
131, 132
135, 136
137, 138
191, 192
211, 212
219, 220

Changes to text and illustrations are indicated by a vertical line at the left of the change.

Summary of Amendments

- Partial rewrite of Chapter 18. Sequential Disk and Tape Programming Considerations
- Adds support of Transaction Logging Subprogram (SUBR81) 5704-FO2 only
- Miscellaneous technical changes

Note: Please file this cover letter at the back of the manual to provide a record of changes.

IBM Corporation, Publications, Department 245, Rochester, Minnesota 55901

©IBM Corp. 1978

Printed in U.S.A.

U, use in DEFINE FILE statement 45
unconditional GO TO statement 19
underflow exceptions 173
unformatted record 41
    definition 200
unit numbers, logical 89
    differences between System/3 and 1130 151
unit record devices (see card devices)
UNITNO parameter in device option statement 97
UPACK compiler option 101
utility subprograms 190

VALUE parameter in CATEGORY compiler option statement 98
variable 12
    definition 200
    names 12
    type specification of
        explicit specification 13
        implicit specification 13
        predefined convention 13
variable-spanned records 135

weak EXTRN reference, in ESL records 113
work files 105, 145
WRITE statement
    direct-access 47
        summarized 168
    sequential 43, 137
        summarized 168

X, use in FORMAT statement 36
XR registers (index registers) 148

Z, used to define hexidecimal constants 11
zero, in FORTRAN statement numbers 7

1130 differences with System/3 151
1403 printer
    logical unit number for 91
    specifying on PRINT device option statement 94
1442 card read punch
    differences between System/3 and 1130 152
    logical unit number for 91
    specifying on PUNCH device option statement 95
    specifying on READ device option statement 94
2222 printer
    logical unit number for 90
    specifying on PRINT device option statement 94
2501 card reader
    logical unit number for 92
    specifying on PRINT device option statement 94
2560 MFCM
    logical unit number for 92
    specifying on PRINT device option statement 94
    specifying on PUNCH device option statement 95
    specifying on READ device option statement 94
3277 display station (CRT/keyboard)
    logical unit number for 92
    programming considerations for 149
    specifying on PRINT device option statement 94
    specifying on READ device option statement 94
3284 printer
    logical unit number for 92
    specifying on PRINT device option statement 94
3401/3411 magnetic tape subsystem, logical unit number for 90
5203 printer
    logical unit number for 90
    specifying on PRINT device option statement 94
5213 printer
    logical unit number for 90
    specifying on PRINT device option statement 94
5406 console keyboard
    logical unit number for 90
    specifying on READ device option statement 94
5424 MFCU, logical unit number for 90
5444 disk storage drive
    logical unit number for 90, 92
    sharing buffers on 134
5445 disk storage drive
    logical unit number assignment 90, 92
    restriction on sharing buffers 134
5471 printer/keyboard
    logical unit number for 90
    specifying on PRINT device option statement 94
    specifying on READ device option statement 94
5496 data recorder
    logical unit number for 90
    specifying on PUNCH device option statement 95
    specifying on READ device option statement 94

Technical Newsletter

This Newsletter No. SN21-5568

Date 25 November 1977

Base Publication No. SC28-6874-3

File No. S3-25

Previous Newsletters None

IBM System/3
FORTRAN IV
Reference Manual

© IBM Corp. 1972, 1974, 1976

This technical newsletter, a part of version 02, modification 00 of IBM System/3 Model 15 FORTRAN (Program Product Number 5704-FO2), also applies to IBM System/3 Model 15 FORTRAN (Program Product Number 5704-FO1). This technical newsletter provides replacement pages for the subject publication. These replacement pages remain in effect for subsequent versions and modifications unless specifically altered. Pages to be inserted and/or removed are:

v, vi	137, 138
91 through 96	149, 150
96.1, 96.2 (added to accommodate moved information)	189, 190
	201 through 204
105, 106	

Changes to text and illustrations are indicated by a vertical line at the left of the change.

Summary of Amendments

- Adds support for 3741
- Tape processing in programs using overlays
- Miscellaneous technical changes

Note: Please file this cover letter at the back of the manual to provide a record of changes.

IBM Corporation, Publications, Department 245, Rochester, Minnesota 55901

© IBM Corp 1977

Printed in U.S.A.

SC28-6874-3

IBM S/3 FORTRAN IV Reference (File No. S3-25) Printed in U.S.A. SC28-6874-3

International Business Machines Corporation
General Systems Division
5775D Glenridge Drive N.E.
Atlanta, Georgia 30301
(USA Only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

IBM

Technical Newsletter

This Newsletter No. SN21-5711
Date 21 December 1979

Base Publication No. SC28-6874-3
File No. S3-25

Previous Newsletters SN21-5568
SN21-5634

IBM System/3
FORTRAN IV
Reference Manual

©IBM Corp. 1972, 1974, 1976

This technical newsletter applies to the current versions and modifications of the applicable System/3 programs listed in the edition notice and provides replacement pages for the subject publication. These replacement pages remain in effect for subsequent versions and modifications unless specifically altered. Pages to be inserted and/or removed are:

69, 70
87, 88

Changes to text and illustrations are indicated by a vertical line at the left of the change.

Summary of Amendments

Miscellaneous technical changes

Note: Please file this cover letter at the back of the manual to provide a record of changes.

IBM Corporation, Publications, Department 245, Rochester, Minnesota 55901

©IBM Corp. 1979

Printed in U.S.A.