PERQ FORTRAN

September 1983

This manual is for use with FORTRAN Version 02.1 and subsequent releases until further notice.

PERQ Systems Corporation
2600 Liberty Avenue
P. O. Box 2600
Pittsburgh, PA 15230
(412) 621-6250

PREFACE

This document describes the PERQ FORTRAN 77 language. It is a reference publication primarily intended for users with previous experience of FORTRAN programming. The document includes information on running FORTRAN 77 programs with the PERQ operating system.

The information in this publication may be accessed at random through the index. Chapter 1 is a general introductory chapter which describes the basic elements of the language, and Chapters 2 and 3 describe the various types of data, their values and how they are stored. Chapter 4 is concerned with expressions and Chapter 5 with assignment statements. Chapters 6 and 7 describe the transfer of control within and between the units of a program respectively. Chapter 8 is concerned with format specifications, which are used in conjunction with the input and output facilities described in Chapter 9. Chapter 10 describes extensions to the language as defined in ANSI X3.9-1978, and certain implementation limits. Chapter 11 explains how to use PERQ FORTRAN 77 in conjunction with PERQ Pascal programs. Chapter 12 describes how to compile and run PERQ FORTRAN 77 programs.

Appendix A gives a description of intrinsic functions.

Appendix B gives release information on the availability of the facilities described in this publication.

This document is divided into chapters, and each chapter is subdivided into sections. A section's level in the hierarchy is indicated by its number. Therefore, within Chapter n, first level section headings are numbered n.1, n.2 and so on; second level headings are numbered n.1.1, n.1.2, ..., n.2.1 and so on; third level headings are numbered n.1.1.1, n.1.1.2, ..., n.1.2.1 and so on.

The contents list and index, and cross-references in the text, all refer to section numbers.

Pages are numbered within chapters, in the form c-p, where c is the chapter number and p the page number within that chapter. Figures and tables, where they appear, are also numbered within chapters, so that Figure n.2 is the second figure in Chapter n, and Table n.2 is the second table in that chapter.

Section numbers, page numbers and figure and table numbers in appendices are preceded by the letter of the appendix.

Exponents are shown in the text as n**(e) where n is a number and e is an exponent.

Amendment lists will be issued when the manual is revised to correct errors in the manual or to enhance the manual itself. Release changes will be issued when technical changes take place. Each amendment list or release change will contain one or more numbered instructions to replace or discard existing pages or to add new pages.

The following rules apply to both amendments and release changes:

a) When a page is re-issued, significant technical changes will be indicated on the re-issued page by amendment lines (vertical lines in the margin beside the changed passages);

b) The date on the re-issued page will be the date of the amendment or release change;

c) Any lines that remain from a previous amendment or release change will be removed from re-issued pages;

d) The date on unchanged backing pages will not be updated, nor will the old amendment lines on these pages be removed.

e) New chapters will not be marked with amendment lines.

f) In completely revised chapters, only significant technical changes will be marked with amendment lines.

TABLE OF CONTENTS

Preface	iii
INTRODUCTION	Chapter 1
Character Set	1.1
Program Structure	1.2
Program Unit Structure	1.3
Lines	1.3.1
Initial line	1.3.1.1
Continuation lines	1.3.1.2
Comment lines	1.3.1.3
Statements	1.3.2
END statement	1.3.2.1
Statement labels	1.3.3
Categories of statement	1.3.4
Executable statements	1.3.4.1
Non-executable statements	1.3.4.2
Order of statements and lines	1.3.5
Names	1.4
Coding a Program	1.5

DATA Chapter 2
Data Values and Types 2.1
Constants, Variables and Arrays 2.2
Constants 2.2.1
Integer constants 2.2.1.1
Real constants 2.2.1.2
Double precision constants 2.2.1.3
Complex constants 2.2.1.4
Logical constants 2.2.1.5
Character constants 2.2.1.6
Symbolic constants 2.2.2
Variables 2.2.3
Arrays 2.2.4
Array elements 2.2.4.1
Type Specification 2.3
Predefined specification 2.3.1
The IMPLICIT statement 2.3.2
Explicit type specification statement 2.3.3
Arithmetic and logical type statements 2.3.3.1
The CHARACTER statement 2.3.3.2
The PARAMETER statement 2.3.4
STORAGE OF DATA Chapter 3
Storage Requirements 3.1
Constants 3.1.1
Variables 3.1.2

Arrays	3.1.3
Character storage	3.1.4
Character substring	3.1.5
Allocation of Storage	3.2
General considerations	3.2.1
Variables	3.2.1.1
Arrays	3.2.1.2
The DIMENSION statement	3.2.2
The COMMON statement	3.2.3
Common block names	3.2.3.1
Storage of variables	3.2.3.2
Multiple references within a program unit	3.2.3.3
Using the COMMON statement	3.2.3.4
The EQUIVALENCE statement	3.2.4
Arrays in EQUIVALENCE statement	3.2.4.1
Equivalencing items of different types or length	3.2.4.2
Equivalencing common block items	3.2.4.3
Assignment of Initial Values	3.3
The DATA statement	3.3.1
Value lists	3.3.1.1
Implied-DO in a DATA statement	3.3.1.2
Character values	3.3.1.3
Examples of initial value assignment	3.3.1.4
Block data subprogram	3.3.2

Example 3.3.2.1
EXPRESSIONS Chapter 4
Arithmetic Expressions 4.1
Arithmetic elements 4.1.1
Arithmetic operators and parentheses 4.1.2
Rules 4.1.3
Order of evaluation 4.1.4
Examples of arithmetic expressions 4.1.5
Determination of the type of an expression 4.1.6
Integer arithmetic 4.1.7
Arithmetic constant expressions 4.1.8
Integer constant expressions 4.1.9
Character Expressions 4.2
Character elements 4.2.1
Character operator and parentheses 4.2.2
Logical Expressions 4.3
Logical elements 4.3.1
Relational expressions 4.3.2
Logical operators and parentheses 4.3.3
Rules 4.3.4
Order of evaluation 4.3.5
Examples of relational and logical expressions 4.3.6

ASSIGNMENT STATEMENTS Chapter 5
Arithmetic Assignment Statements 5.1
Logical Assignment Statements 5.2
Character Assignment Statements 5.3
CONTROL STATEMENTS Chapter 6
GO TO Statements 6.1
Unconditional GO TO 6.1.1
Computed GO TO 6.1.2
Assigned GO TO and ASSIGN statements 6.1.3
IF Statements 6.2
Arithmetic IF 6.2.1
Logical IF 6.2.2
Block IF 6.2.3
The block IF statement and IF-blocks 6.2.3.1
The ELSE statement and ELSE-blocks 6.2.3.2
The ELSE IF statement and ELSE IF blocks 6.2.3.3
The END IF statement 6.2.3.4
DO Loops 6.3
DO statements 6.3.1
Terminal statements 6.3.2
Nested DO loops 6.3.3
Transfer of control in DO loop 6.3.4
CONTINUE Statements 6.4
STOP Statements 6.5
PAUSE Statements 6.6

PROGRAM UNITS AND THE TRANSFER OF CONTROL BETWEEN THEM Chapter 7

Procedures 7.1

Differences between function and subroutine subprograms 7.1.1

Functions 7.1.2

Intrinsic functions 7.1.2.1

External functions 7.1.2.2

Statement functions 7.1.2.3

Subroutines 7.1.3

External subroutines 7.1.3.1

Transfer of Control Between Program Units 7.2

Functions 7.2.1

Transfer of control to a function subprogram 7.2.1.1

Return of control from a function subprogram 7.2.1.2

Example of an external function 7.2.1.3

Subroutines 7.2.2

Transfer of control to a subroutine subprogram 7.2.2.1

Return of control from a subroutine subprogram 7.2.2.2

Example of a subroutine subprogram 7.2.2.3

Correspondence between Dummy and Actual Arguments 7.3

Use of constants and expressions 7.3.1

Use of variables 7.3.2

Use of arrays and array elements 7.3.3

Adjustable arrays 7.3.3.1

Use of functions and subroutines as arguments 7.3.4

The INTRINSIC statement	7.3.5
Transfer of Values Between Program Units	7.4
Common block items	7.4.1
Dummy and actual arguments	7.4.2
Argument reference	7.4.2.1
Multiple Entry into a Subprogram	7.5
The ENTRY statement	7.5.1
Referencing an ENTRY statement	7.5.2
Entering the subprogram	7.5.3
Exit from the subprogram	7.5.4
SAVE Statement	7.6
FORMAT SPECIFICATION	Chapter 8
Format Specifications	8.1
Field separators	8.1.1
Slash editing	8.1.1.1
Repetition of descriptors	8.1.2
Format Specification Methods	8.2
The FORMAT statement	8.2.1
Character format specification	8.2.2
Effect of FORMAT statements and character format specifications	8.2.3
Edit Descriptors	8.3
Format codes	8.3.1
The I conversion code	8.3.1.1

The F conversion code 8.3.1.2
The E and D conversion codes 8.3.1.3
The G conversion code 8.3.1.4
The scale factor 8.3.1.5
The L conversion code 8.3.1.6
The A conversion code 8.3.1.7
The H format code and character data 8.3.1.8
The X format code 8.3.1.9
The T format codes 8.3.1.10
The S format codes 8.3.1.11
The B format codes 8.3.1.12
Colon editing 8.3.1.13
Examples of Format Specification 8.4

INPUT AND OUTPUT Chapter 9
Introduction 9.1
Format of records 9.1.1
Unformatted records 9.1.1.1
Formatted records 9.1.1.2
Accessing records 9.1.2
Sequential access 9.1.2.1
Direct access 9.1.2.2
Input/Output Statements 9.2
Input/output lists 9.2.1
Correspondence between input/output lists and format codes 9.2.1.1

Implied DO loops	9.2.1.2
Sequential Access Input and Output	9.3
READ and WRITE statements	9.3.1
Formatted sequential access input and output	9.3.1.1
Unformatted sequential access input and output	9.3.1.2
File positioning input/output statements	9.3.2
Direct Access Input and Output	9.4
READ and WRITE statements	9.4.1
Formatted direct access input and output	9.4.1.1
Unformatted direct access input and output	9.4.1.2
List Directed Input and Output	9.5
The READ statement	9.5.1
Input data	9.5.2
Output statements	9.5.3
The WRITE statement	9.5.3.1
The PRINT statement	9.5.3.2
Output data	9.5.4
Internal Files	9.6
Auxiliary Input/Output Statements	9.7
Unit and file connection	9.7.1
File existence	9.7.1.1
File properties	9.7.1.2
The OPEN statement	9.7.2
Changing the properties of a connection	9.7.2.1

The CLOSE statement 9.7.3

The INQUIRE statement 9.7.4

INQUIRE by Unit 9.7.4.1

INQUIRE by File 9.7.4.2

The INQUIRE specifiers 9.7.4.3

LANGUAGE EXTENSIONS AND IMPLEMENTATION CHARACTERISTICS Chapter 10

Language Extensions 10.1

Lower case 10.1.1

Hollerith constants 10.1.2

Symbolic names 10.1.3

Calling Pascal 10.1.4

Character length 10.1.5

Integer*2 10.1.6

Implementation Characteristics 10.2

Storage of constants and variables 10.2.1

Integer 10.2.1.1

Real 10.2.1.2

Double precision 10.2.1.3

Complex 10.2.1.4

Logical 10.2.1.5

Character 10.2.1.6

Input/output record lengths 10.2.2

Code and data limitations 10.2.3

Run time file access 10.2.4

File connections 10.2.4.1
File types 10.2.4.2
Console input/output 10.2.4.3
Opening files 10.2.5
Length of program unit names 10.2.6
Block data 10.2.7
STOP and PAUSE 10.2.8

MIXED LANGUAGE PROGRAMMING Chapter 11
Introduction 11.1
Referencing Pascal from FORTRAN 77 11.2
Valid argument correspondence 11.2.1
Valid function result correspondence 11.2.2
Pascal Input and Output 11.3

RUNNING A PERQ FORTRAN 77 PROGRAM Chapter 12
FORTRAN Programming 12.1
Utilities 12.2
Compilation 12.3
Using the compiler 12.3.1
Compiler switches 12.3.2
Listings (/LIST, /CROSSREFERENCE, /SHOWCODE) 12.3.2.1
Diagnostic options (/NORANGE, /NOCHECK 12.3.2.2
Program scan (/SYNTAXCHECK) 12.3.2.3
Quiet (/QUIET) 12.3.2.4

Help (/HELP) 12.3.2.5

Using Pascal or FORTRAN externally defined routines (/EXTERNAL) 12.3.2.6

Initialization of data to zero (/ZERO) 12.3.2.7

File inclusion 12.3.2.8

Argument mismatch (/MISMATCH) 12.3.2.9

Error file (/ERRORFILE) 12.3.2.10

Compile time diagnostics 12.3.3

Consolidation 12.4

Using the consolidator 12.4.1

Partial consolidation 12.4.2

Linking 12.5

Running the Program 12.6

Run time diagnostics 12.6.1

Setting Up the Software 12.7

INTRINSIC FUNCTIONS Appendix A

PERQ OPERATING SYSTEM AND FORTRAN 77 RELEASES Appendix B

Incompatible Changes Between Version 01 and Version 02.1 of PERQ FORTRAN 77 B1.1

File types B1.1.1

Pause B1.1.2

Existing FORTRAN programs B1.2

FORTRAN 77 INPUT/OUTPUT ERRORS Appendix C

Index

CHAPTER ONE

INTRODUCTION

This chapter provides general FORTRAN information. The chapter introduces basic terminology and outlines the structure of a FORTRAN program.

Fortran is a programming language designed primarily for the mathematical or scientific user. You write a FORTRAN program as a series of statements using symbolism analogous to that used in algebra. Many of these statements are readily intelligible to a programmer with mathematical training. For example, the FORTRAN expression

```
(A+B)/C
```

resembles a line of algebra and has a similar meaning. Each statement occupies at least one line of coding and can extend onto subsequent lines if necessary. Statements can be given identifying labels.

FORTRAN provides facilities for the evaluation of common mathematical functions. The programmer need only write

```
Y=SIN (x)
```

and FORTRAN evaluates the sine function. Appendix A lists the standard procedures. The programmer can write similar procedures for himself as external functions.

A FORTRAN program normally executes in the order in which statements are written, but various control statements enable the programmer to specify that control branches to another statement with an identifying label, either unconditionally or if certain conditions are satisfied.

You can write FORTRAN programs as one or more program units and compile each program unit separately. One program unit is designated as the master program unit. This program unit controls the running of the program and passes control to other program units.

1.1 Character set

The set of 49 characters used in writing FORTRAN programs is:

alphabetic	ABCDEFGHIJKLMNOPQRSTUVWXYZ
numeric	0123456789
special characters	+-*/=,.:()'$ and the space (or blank) character

No character other than these may be used except in character constants (see section 2.2), and in comment lines (see section 1.3.1.3).

Alphabetic and numeric characters are referred to collectively as alphanumeric characters.

1.2 Program structure

A program consists of program units. A program always has at least one program unit, called the main program, and may have one or more other program units that are called subprograms. Execution of the program starts in the main program and then control is passed between the main program and subprograms or among subprograms. For further details of transfer of control between program units see Chapter 7.

There are three classes of subprogram:

1 Function subprograms

2 Subroutine subprograms

3 Block data subprograms

Function and subroutine subprograms provide a mechanism to assist the programmer in structuring programs in a meaningful way, and to allow common code to be conveniently accessed. These subprograms are described in detail in Chapter 7.

Block data subprograms are used to give initial values to variables and arrays used in more than one program unit. They differ from other subprograms in that they can contain only certain non-executable statements (see section 1.3.4.2) and in that control is never passed to them. Block data subprograms are described in detail in section 3.3.2.

1.3 Program unit structure

1.3.1 Lines

A line in a program unit consists of 72 character positions. The character positions are numbered from 1 to 72. A statement occupies positions 7 to 72 of one or more lines.

There are three classes of FORTRAN line:

1 Initial lines

2 Continuation lines

3 Comment lines

1.3.1.1 Initial lines

An initial line has the following form:

Positions 1 to 5 may contain a statement label (see section 1.3.3 below);

Position 6 contains a space or the digit 0;

Positions 7 to 72 can contain the statement.

1.3.1.2 Continuation lines

A continuation line has the following form:

Positions 1 to 5 are blank

Position 6 contains any character other than 0 or a space. It is usual to number continuation lines consecutively from 1

Positions 7 to 72 contain the continuation of a statement

1.3.1.3 Comment lines

Comment lines may be included in a program; such lines do not affect the program in any way but can be used by the programmer to include explanatory notes. The letter C or an asterisk in position 1 of a line designates that line as a comment line; a line containing only blank characters in positions 1 to 72 is also a comment line.

The comment text is written in positions 2 to 72.

1.3.2 Statements

A statement consists of an initial line and, where necessary, up to 19 continuation lines.

Except as part of a logical IF statement, no statement may begin on a line that contains any part of the previous statement.

Blank characters may appear preceding, within or following a statement without changing the interpretation of the statement, except when they appear within character constants or the H or apostrophe format codes in FORMAT statements.

1.3.2.1 END statement

An END statement marks the end of a program unit. The statement consists of the three characters E N D, in that order, in any of positions 7 to 72 of an initial line. All other positions from 1 to 72 must contain spaces. No other statement may have an initial line that appears to be an END statement.

1.3.3 Statement labels

Any statement in a FORTRAN program may be identified by preceding it by a statement label.

A statement label is an unsigned integer in the range 1 to 99999. The numbers used as labels have no sequential significance. For example, the label 7 may occur after the label 9853. Labels may appear anywhere within columns 1 to 5. Blanks and leading zeros have no significance in labels.

All statement labels within any one program unit must be unique. Labels may be referred to only in the program unit in which they occur.

1.3.4 Categories of statements

Each statement is classified as executable or non-executable.

Executable statements specify actions and form an execution sequence in a program.

Non-executable statements specify characteristics, arrangement, and initial values of data; contain format editing information; specify statement functions; classify program units; and specify entry points within subprograms. Non-executable statements are not part of the execution sequence. They may be labelled, but such statement labels must not be used to control the execution sequence.

1.3.4.1 Executable statements

The following statements are classified as executable:

1 Arithmetic, logical, statement label (ASSIGN), and character assignment statements

2 Unconditional GO TO, assigned GO TO, and computed GO TO statements

3 Arithmetic IF and logical IF statements

4 Block IF, ELSE IF, ELSE, and END IF statements

5 CONTINUE statement

6 STOP and PAUSE statements

7 DO statement

8 READ, WRITE, and PRINT statements

9 REWIND, BACKSPACE, ENDFILE, OPEN, CLOSE, and INQUIRE statements

10 CALL and RETURN statements

11 END statement

1.3.4.2 Non-executable statements

The following statements are classified as non-executable:

1 PROGRAM, FUNCTION, SUBROUTINE, ENTRY, and BLOCK DATA statements

2 DIMENSION, COMMON, EQUIVALENCE, IMPLICIT, PARAMETER, EXTERNAL, INTRINSIC and SAVE statements

3 INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, and CHARACTER type-statements

4 DATA statement

5 FORMAT statement

6 Statement function statement

1.3.5 Order of statements and lines

Table 1.1 is a diagram of the required order of statements and comment lines for a program unit. Vertical lines delineate varieties of statements that may be interspersed. For example, FORMAT statements may be interspersed with statement function statements and executable statements. Horizontal lines delineate varieties of statements that must not be interspersed. For example, statement function statements must not be interspersed with executable statements.

Table 1.1

Required order of statements and comment lines

<table>
<tr><td colspan="4">PROGRAM, FUNCTION, SUBROUTINE, or
BLOCK DATA statement</td></tr>
<tr><td rowspan="3">Comment
Lines</td><td rowspan="3">FORMAT
and
ENTRY
Statements</td><td>PARAMETER
Statements</td><td>IMPLICIT
Statements
Other
Specification
Statements</td></tr>
<tr><td>DATA
Statements</td><td>Statement
Function
Statements</td></tr>
<tr><td>Executable
Statements</td></tr>
<tr><td colspan="4">END Statement</td></tr>
</table>

Within the specification statements of a program unit, IMPLICIT statements must precede all other specification statements except PARAMETER statements. Any specifion statement that specifies the type of a symbolic constant must precede the PARAMETER statement that defines that particular symbolic constant; the PARAMETER statement must precede all other statements containing the symbolic constants that are defined in the PARAMETER statement.

ENTRY statements may appear anywhere except between a block IF statement and its corresponding END IF statement, or between a DO statement and the terminal statement of its DO-loop.

The last line of a program unit must be an END statement.

All specification statements must precede all DATA statements,

statement function statements, and executable statements.

All statement function statements must precede all executable statements.

1.4 Names

In FORTRAN various items are identified by names chosen by the programmer. A name is a string of up to six alphanumeric characters of which the first must be alphabetic. Sometimes the first character has special significance (see section 2.3.1). Spaces normally have no significance in FORTRAN programs and so, for example, the names NAME1, N AME1 and NAME 1 are identical.

A name, in general, has only one meaning within a program unit. The same name used in different program units does not in general refer to the same object except when it refers to a subprogram or common block name. There are three exceptions to these rules:

1 A common block name may also be a variable, array or statement function name

2 A function subprogram name must also be a variable name within the function subprogram (see section 7.1)

3 The name of a variable used as the DO-variable of an implied-DO in a DATA statement may have any other meaning outside the implied-DO list

Note: The term symbolic name may be used instead of name.

1.5 Coding a program

FORTRAN program statements and comments may be written on a FORTRAN coding sheet. The program can then be entered directly at the keyboard.

FORTRAN coding sheets are divided into lines with 80 positions numbered from 1 to 80. The first 72 positions correspond to the 72 character positions in a line of a FORTRAN statement. Positions 73 to 80 can be used to identify the line or to give the line a sequence number. Characters written in positions 73 to 80 will not affect the program in any way.

CHAPTER TWO

DATA

This chapter is concerned only with the organization of data in a FORTRAN program. The three permissible types of data are described, together with the possible methods of data specification. Data storage and input/output are described in Chapters 3 and 9 respectively.

2.1 Data values and types

Values in FORTRAN can be classified as follows:

1 Arithmetic values. These can be further subdivided into:

(a) Integer values, which are whole numbers. Such values are said to be of type integer and are held exactly in fixed point form in store.

(b) Real values, which are numbers expressed as decimal fractions with exponents. Such values are said to be of type real and are held approximately in floating-point form in store.

(c) Double precision values are numbers held in the same form as real values, but to a greater precision.

(d) Complex values, representing complex numbers. Such values are said to be of type complex and are held in store as a pair of real values, the first representing the real part and the second representing the imaginary part.

2 Logical values, representing the values true or false. Such values are said to be of type logical.

3 Character values, representing strings of characters. Such values are said to be of type character and their length is under the control of the programmer.

2.2 Constants, variables and arrays

Values can be made available for use in calculations in one of four ways:

1 As a constant value which can be written at the point in the program at which it is required (see section 2.2.1)

2 As a symbolic constant which has previously been named and defined with a value in a PARAMETER statement (see section 2.3.4)

3 In a variable. This is a named area of storage which can contain one item of data of a particular type, the type being determined by the variable name (see section 2.2.3) or by a type specification statement (see section 2.3)

4 In an array. This is a named area of storage which can contain a set of items of data of a particular type, the type being determined by the array name or by a type specification statement (see section 2.3). Each item of data within the set is called an array element (see section 2.2.4).

Variables, arrays and array elements may be assigned initial values by use of DATA statements (see section 3.3.1) and may be assigned new values during execution of the program.

2.2.1 Constants

There are six types of constant that can be used: integer, real, double precision, complex, logical and character. Integer, real, double precision and complex constants are grouped together as arithmetic constants.

2.2.1.1 Integer constants

An integer constant is an optionally signed whole number written as a string of digits with no decimal points or exponents. Unsigned integer constants are assumed to be positive.

2.2.1.2 Real constants

Real constants are numbers written containing a decimal point, an exponent or both. They may be signed or unsigned. If they are unsigned they are assumed to be positive. Exponents are written as the letter E

followed by a one or two digit signed or unsigned integer. The integer represents a power of ten to which the constant is to be raised.

Thus real constants may take any of the following forms:

+n.m	+n.mE+a
+n.	+n.E+a
+.m	+.mE+a
	+nE+a

where n and m are strings of digits, a is a one or two digit integer constant and + is an optional sign.

2.2.1.3 Double precision constants

Double precision constants are numbers written containing an optional decimal point and an exponent. They may be signed or unsigned. If they are unsigned they are assumed to be positive. Exponents are written as the letter D followed by a one or two digit signed or unsigned integer. The integer represents the power of ten to which the constant is to be raised.

Double precision constants take any of the following forms:

+n.mD+a

+n.D+a

+.mD+a

+nD+a

where n and m are strings of digits, a is a one or two digit integer constant and + is an optional sign.

2.2.1.4 Complex constants

Complex constants are pairs of real or integer constants; the first constant corresponds to the real part of a complex number and the second corresponds to the imaginary part.

Complex constants have the form

(a,b)

where a and b are constants and (a,b) represents the complex number a+ib.

The form -(a,b) is not a valid complex constant, and would have to be written (-a,-b).

2.2.1.5 Logical constants

There are two logical constants, representing the values true and false. They have the forms

```
.TRUE.

.FALSE.
```

2.2.1.6 Character constants

A character constant is a non-empty string of any characters, delimited by being enclosed in apostrophes.

If a string enclosed in apostrophes itself contains an apostrophe, this must be represented by two apostrophes to distinguish it from a delimiting apostrophe.

The length of a character constant is the number of characters which appear between the delimiting apostrophes, except that each pair of consecutive apostrophes counts as a single character.

The following are examples of valid character constants:

```
'C(1)='
'MUSTARD AND CRESS'
'ISN''T'
```

2.2.2 Symbolic constants

A symbolic constant is a constant value that is identified by a name (see section 1.4). The value associated with the symbolic constant is defined in a PARAMETER statement (see section 2.3.4) which must appear before any use is made of the name to represent a value. The type of a symbolic constant is determined in the same way as for a variable (see sections 2.2.3 and 2.3).

2.2.3 Variables

A variable is an item of data that is identified by a name (see section 1.4). Values can be assigned to variables during the execution of a program. The value assigned to a variable at any time is made available to the program when a reference is made to the variable name.

In general, a particular variable will be available in only one program unit. A name used for a variable in one program unit may be used for an entirely different variable in another program unit.

There are six types of variable: integer, real, double precision, complex, logical or character.

The ranges of values these types can take are the same as for the corresponding types of constant (see sections 2.2.1.1 to 2.2.1.6).

If the name chosen for a variable begins with one of the letters I to N inclusive, then the variable will be assumed to be of type integer. Otherwise it will be assumed to be of type real. However, the programmer can override this convention by specifying, in a type specification statement, the type the variable is to be (see section 2.3).

For example, variables with names such as INT, LIST, NUMBER or J322 would be assumed to be of type integer unless otherwise specified. Variables with names such as AREA, SUM or R147 would be assumed to be of type real unless otherwise specified.

2.2.4 Arrays

Sets of data items of the same type can be processed as arrays. A single name, the array name, is chosen to identify the set, and individual items are called the array elements (see section 1.4 for further details concerning names).

Arrays may have one or more dimensions. For example, the matrix A where

```
A = A(1,1) A(1,2) A(1,3) A(1,4)
    A(2,1) A(2,2) A(2,3) A(2,4)
```

could be treated as a two-dimensional array with eight elements. Arrays may have up to seven dimensions.

There are six types of array: integer, real, double precision, complex, logical and character. The type of an array is determined in the same way as the type of a variable, and each element of the array has this same type.

2.2.4.1 Array elements

In some contexts an array may be referred to as a whole by specifying the array name. In other contexts individual elements may be referred to by an array element reference which takes the form of the array name followed by a subscript list enclosed in parentheses. A subscript list is an ordered set of subscript expressions separated by commas, one subscript expression for each dimension of the array. A subscript expression may be an arithmetic expression (see Chapter 4) of type integer.

The compiler allocates storage to the array as instructed by an array declarator (see section 3.2.2). The array declarator and the subscript expressions given in the array element reference are used to calculate the position in store that is occupied by the specified element. The order in which array elements are held in store is specified in section 3.1.3.

Each subscript expression, when evaluated, must have a value within the declared bounds for that subscript.

The following are examples of valid array element references, with explanations:

TABLE(7)	Element (7) of the one dimensional array TABLE
MAT(I,I+1)	If I is an integer variable with the value 7, this reference is to element (7,8) of the two-dimensional array MAT
VECTOR(IFUN(J,3))	Where IFUN is an integer external function or statement function requiring two actual arguments and VECTOR is a one dimensional array. The function is evaluated to give the array element required.

2.3 Type specification

In FORTRAN all constants, symbolic constants, variables, arrays and functions must be identified as being of particular types so that they can be stored and processed correctly. The type of a constant is indicated by the way the constant is written.

The type of a symbolic constant, variable, array or function may be defined in any of three ways:

1 Predefined specification

2 IMPLICIT specification

3 Explicit specification statements

Explicit statements override IMPLICIT specifications, which in turn override predefined specifications.

2.3.1 Predefined specification

Any symbolic constants, variables, arrays or functions whose names are not mentioned in a type specification statement and whose initial letter is not mentioned in an IMPLICIT statement (see section 2.3.2) will be assumed to be of type integer or real according to the following rules:

1 If the name of the symbolic constant, variable, array or function begins with any of the letters I, J, K, L, M or N the compiler assumes the symbolic constant, variable, array or function to be of type integer

2 If the name begins with any other letter the quantity is assumed to be of type real

Some examples of predefined type variable names are given in section 2.2.3.

2.3.2 The IMPLICIT statement

The IMPLICIT statement provides a means of overriding the FORTRAN convention of predefined specification for the types of symbolic constants, variables, arrays and functions. This takes effect for the whole of the current program unit unless overridden by explicit type statements.

The statement takes the form:

```
IMPLICIT type1(a1,a2,...),...,typen(am,an,...)
```

where

type is one of: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL or CHARACTER *s.

a1,a2,... and am,an,... are lists of single alphabetic characters separated by commas, or a range of alphabetic characters in sequence, separated by a minus sign. The same letter may not appear singly, or within a range of characters, more than once in a subprogram.

s is the length of the character entities and is either an unsigned, non-zero integer constant, or an integer constant expression enclosed in parentheses and with a positive value. s (together with the preceding *) is optional and, if omitted, the length is one.

After this statement has been processed, all symbolic constants, variable, array or function names beginning with the characters a1,a2... are implicitly of type type1 and all symbolic constants, variable, array or function names beginning with am,an... are implicitly of type typen unless the specification is overridden by an explicit specification statement.

A program unit may contain more than one IMPLICIT statement, but IMPLICIT statements must precede all other specification statements except PARAMETER statements. For a subprogram, IMPLICIT statements can specify the type of the parameters to the subprogram, and of the function name for a function subprogram, unless their types are specified in an explicit type specification statement.

Example 1

```
IMPLICIT REAL(A-D,X,Z),LOGICAL(L)
```

This statement specifies that all variables whose names begin with A, B, C, D, X or Z that do not appear in explicit type statements are to be real. Similarly all variables whose names begin with L are assumed to be logical.

Example 2

```
COMPLEX FUNCTION BACH(THEME,FUGUE)
IMPLICIT DOUBLE PRECISION(A-H)
```

The overall effect of these two statements is that the parameter FUGUE will be of type double precision and the function BACH will be of type complex. The parameter THEME is assumed to be type real by virtue of its initial letter T.

2.3.3 Explicit type specification statements

Explicit type specification statements are used to confirm or override the predefined or implicit type specification, and optionally to give dimension information for arrays.

The appearance of the name of a symbolic constant, variable, array, external function or statement function specifies the data type for that name for all appearances in the program unit. Within a program unit a name must not have its type explicitly specified more than once. A type statement which confirms the type of an intrinsic function (listed in Appendix A) is permitted, but is not necessary. The appearance of a generic function name (see section 7.1.2.1) in a type statement does not necessarily remove the generic properties of that function.

2.3.3.1 Arithmetic and logical type statements

The statements take the form:

```
type a(k1),b(k2),...,c(kn)
```

where

type is one of: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL.

a,b,...,c are symbolic constant, variable, array, function or dummy procedure names (see section 7.1).

(k1),(k2),...,(kn) are optional and give dimension information for arrays (see section 3.2.2).

Example 1

```
REAL A,B(10),C,D
```

This statement declares

A, C and D to be real

B to be a real array with 10 elements

Example 2

```
INTEGER FRED,JIM,UNCLES(5)
```

This statement declares the variables FRED and JIM, of type integer. In addition, the integer array UNCLES is declared, which has five elements.

Example 3

```
DOUBLE PRECISION HEXNO,INTNO
```

This statement declares two real variables, HEXNO and INTNO.

Example 4

```
LOGICAL L,BOOLE
```

This statement declares two logical variables, L and BOOLE.

2.3.3.2 The CHARACTER type statement

This statement is written:

```
CHARACTER *s, a(k1)*s1,b(k2)*s2,...,c(kn)*sn
```

where

s,s1,s2,...,sn are length specifications (numbers of characters) of a character variable, character array element, character symbolic constant, or character function. Each s is one of the following:

1 An unsigned, non-zero, integer constant

2 An integer constant expression enclosed in parentheses and with a positive value

3 An asterisk in parentheses.

The value of s may be up to 32,767.

An s immediately following the word CHARACTER is the length specification for each entity in the specification not having one of its own. A length specification immediately following an entity applies only to that entity: for an array the length specification is for each element of that array. If a length is not specified for an entity, its length is one. If a length is specified for an entity declared in the statement, the length specification must be a positive non-zero integer constant expression, unless the entity is an external function, a dummy argument of an external subprogram or a character symbolic constant.

If a dummy argument has a length (*) declared, the dummy argument assumes the length of the associated actual argument for each reference of the subprogram. If the associated actual argument is an array name, the length assumed by the dummy argument is the length of an array element in the associated actual argument array.

If an external function has a length (*) declared in a function subprogram, the function name must appear as the name of a function in a FUNCTION or ENTRY statement in the same subprogram. When a reference to such a function is executed, the function assumes the length specified in the referencing program unit.

The length specified for a character function in the program unit that references the function must be an integer constant expression and must agree with the length specified in the subprogram that specifies the function. There is always agreement of length if (*) is specified in the subprogram that specifies the function.

If a character symbolic constant has a length (*) declared, the symbolic constant assumes the length of its corresponding constant expression in a PARAMETER statement.

The length specified for a character statement function or statement function dummy argument of type character must be an integer constant expression.

a,b,c are variable, symbolic constant, function or dummy procedure names.

(k1),(k2),...,(kn) are optional and give dimension information for arrays.

Example

```
CHARACTER CHAR,BUFF*80
```

This statement declares two character variables BUFF and CHAR. CHAR occupies one character (the default length) and BUFF occupies 80 characters.

2.3.4 The PARAMETER statement

A PARAMETER statement is used to define the value of a symbolic constant. The statement has the form:

```
PARAMETER (a=e,a=e,...)
```

where

a is a symbolic constant name.
e is a constant expression.

If a is of type integer, real, double precision or complex, the corresponding e must be an arithmetic constant expression. If a is of type character or logical, the corrresponding e must be a character constant expression or a logical constant expression respectively.

Each a is the name of a symbolic constant that is defined by the value of its corresponding e in accordance with the rules for assignment statements (see Chapter 5). No a may be defined more than once in any program unit.

If any a is not to have the type specified implicitly then its type must be specified by a type-statement (see section 2.3.3) or an IMPLICIT statement (see section 2.3.2) before its appearance in a PARAMETER statement. If the length specified for a symbolic constant of type character is not the default length of one, then its length must first be given in an IMPLICIT or type statement. Its length cannot be changed subsequently.

Once a symbolic constant has been defined it may be used in any subsequent statement in the same program unit as an element of an expression or in a DATA statement, but not as part of a format specification or as part of another constant.

CHAPTER THREE

STORAGE OF DATA

This chapter deals with the storage of data. It describes how quantities are held in store according to their type and then describes the various non-executable statements concerned with allocating storage and assigning initial values to variables.

Specification of type is described in Chapter 2 and the order in which the non-executable statements described in this chapter must occur is given in section 1.3.5.

3.1 Storage requirements

The standard unit of storage is a byte, which consists of 8 binary digits.

The amounts of storage required by the various types of data are defined below.

3.1.1 Constants

The number of bytes reserved for each type of constant is defined in sections 10.2.1.1 to 10.2.1.6.

3.1.2 Variables

The number of bytes reserved for each type of variable is defined in sections 10.2.1.1 to 10.2.1.6.

3.1.3 Arrays

Arrays can take the same types as variables. Each of the array elements has the same type as the array and is allocated the standard amount of storage for a variable of that type. The type of the array depends on its name (see section 2.3) unless otherwise specified.

The number of dimensions of each array and their sizes must be specified once, and only once, in a DIMENSION statement, COMMON statement (see sections 3.2.2 and 3.2.3) or an explicit type statement (see section 2.3.3).

Individual array elements may in general be referred to by giving the array name and a list of subscript expressions, one subscript expression for each dimension (see section 2.2.4.1).

However, it is sometimes necessary for the programmer to know how the elements of an array are arranged in store. They are stored consecutively so that the left-hand subscript varies most rapidly and successive subscripts vary less rapidly.

For example, the array A declared as DIMENSION A(3,2) would be stored as follows:

A(1,1) A(2,1) A(3,1) A(1,2) A(2,2) A(3,2)

and array B, declared as DIMENSION B(4,3,2) would be stored:

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

3.1.4 Character storage

Characters may be held in a variable or array element of type character. Character values may be given to variables or array elements in four ways:

1 By assigning a character value in a DATA statement (see section 3.3.1)

2 By specifying a character constant as an actual argument of a subroutine call or of a function reference

3 By using an A format description in conjunction with a READ statement (see section 8.3.1.7)

4 By specifying a character constant or expression on the right-hand side of a character assignment statement

3.1.5 Character substring

A character substring is an unbroken portion of a character scalar or array element and is a variable of type character. It may be assigned values and referenced, and is identified by a substring name of the form:

```
c(p1:p2)
a(k,k,...)(p1:p2)
```

where

c is a character variable name.

a(k,k,...) is a character array element name

p1 and p2 are integer expressions and are known as substring expressions.

The value p1 specifies the leftmost character position of the substring, and p2 specifies the right most character position. The values of p1 and p2 must be such that

```
1 <= p1 <= p2 <=s
```

where s is the length of the character variable c or the array element a(k,k...).

If p1 is omitted then the value of 1 is assumed, and if p2 is omitted then the value of s is assumed: both p1 and p2 may be omitted.

3.2 Allocation of storage

3.2.1 General considerations

This section discusses various statements used for storage allocation.

COMMON statements (see section 3.2.3) allow the same area of storage to be accessed by a number of different program units. This allows values assigned in one program unit to be used in other units.

EQUIVALENCE statements (see section 3.2.4) allow the same storage space to be used for more than one variable or array within one program unit.

DIMENSION statements (see section 3.2.2) are used for declaring arrays (see also section 3.1.3).

3.2.1.1 Variables

It is not necessary to specify every variable name in a non-executable statement in order that storage will be reserved for it. Variable names may be mentioned in various non-executable statements but this will always be for some purpose other than merely informing the compiler of the existence of the variable. If a variable name is not mentioned in a non-executable statement, when the name is first encountered in an executable statement, the compiler will assume the variable to be of the type given by its initial letter (see section 2.3.1) and will implicitly allocate the amount of storage accordingly.

3.2.1.2 Arrays

Arrays must be specifically defined so that the compiler is informed of the number of dimensions and the size of each dimension. The definition is given by means of an array declarator, whose form is specified in section 3.2.2. Array declarators may be given in type specification statements, COMMON statements and DIMENSION statements, but a particular array declarator may only be given once in each program unit. An array name on its own does not constitute an array declarator and thus, for example, the array name on its own could occur in a COMMON statement and a type specification statement if the declarator were given in a DIMENSION statement.

3.2.2 The DIMENSION statement

The DIMENSION statement is used to declare names as being array names and to specify the number of dimensions of the array and the size of each dimension, so that the appropriate amount of storage can be allocated to the array.

The statement has the form

```
DIMENSION arrayname1(k1),arrayname2(k2),...,arraynamen(kn)
```

where

arrayname1,arrayname2,...,arraynamen are array names.

k1,k2,...,kn are parameters that give the sizes of each dimension of the corresponding array.

k takes the form

```
l1:u1, l2:u2,... lz:uz
```

where l and u are the lower and upper dimension bounds respectively. The dimension bounds are arithmetic expressions in which all constants, symbolic constants, and variables are of type integer. Integer variables may appear in dimension bound expressions only for adjustable array specifications. A dimension bound expression must not contain a function or array element reference. The upper dimension bound of the last dimension in an assumed size specification may be an asterisk.

The value of either bound may be positive, negative or zero provided that the upper bound is not less than the lower bound. If the lower bound is not specified it is assumed to have the value one. An upper bound specified as an asterisk always has a value greater than or equal to the lower bound.

z is a number between 1 and 7 inclusive.

Each arrayname(k) is an array declarator.

A declarator for each array used in a program unit must be given once and only once in that program unit, in either a DIMENSION statement, a type specification statement or a COMMON statement. A declarator appearing in a COMMON statement may not contain dimension sizes specified by integer variable names or an asterisk.

Example

The statement

```
DIMENSION A(10),B(1:500,0:9)
```

declares A as a one-dimensional array with ten elements and B as a two-dimensional array, one dimension having 500 elements and the other dimension having ten elements.

3.2.3 The COMMON statement

The COMMON statement enables areas of storage, known as common blocks, to be used in more than one program unit of a program. Thus, values obtained in one program unit can be used in other program units (see section 3.2.3.4).

The statement has the form:

```
COMMON/cbname1/a(k1),b(k2),...,/cbname2/c(k3),d(k4),...
```

where

/cbname/ is an optional parameter giving a common block name (see section 3.2.3.1 below).

a,b,... are variable or array names. In a subroutine or function these must not be dummy arguments (see section 7.1).

If any variable or array is of type character then all variables and arrays in that common block must also be of type character.

(k1),(k2),... are optional parameters giving dimension information for arrays.

An optional comma may appear after a list of variable or array names and before the slash (/) prefacing another common block name.

3.2.3.1 Common block names

Any number of common blocks may be used in a program. If desired, one common block may be used without a name: such a block is referred to as the blank common block. All other blocks must be given a block name chosen to obey the rules for names, given in section 1.4. No intrinsic function name (see Appendix A) may be used as a common block name nor may the name of a constant or of any program unit of the program be used.

There are no other restrictions on common block names (and thus duplication of variable and other names by common block names is permissible).

If a blank common block is used, two consecutive slashes (//) must precede the names of variables and arrays in the block unless the blank common block is the first block given in the COMMON statement, in which case the two slashes may be omitted.

The same block name may appear more than once in a COMMON statement or in a subprogram; see section 3.2.3.3 below.

Example 1

```
COMMON /BLOCK 1/VALUE,ARRAY/BLOCK 2/X,Y,Z
```

The variable VALUE and the array ARRAY are held in a common block named BLOCK 1, and variables X,Y,Z are held in a common block named BLOCK 2.

The dimensions of the array ARRAY must be declared in a type statement or a DIMENSION statement, since they have not been given in the COMMON

statement.

Example 2

```
COMMON COUNT,TABLE,RESULT/AREA/SUB,ITEM
COMMON /AREA/SUB,ITEM//COUNT,TABLE,RESULT
```

These two statements have the same effect; COUNT, TABLE, RESULT are held in the blank common block and SUB and ITEM are held in common block AREA.

Example 3

```
COMMON A,B(20)/COMM/D(40),E,I//X,Y,Z
```

This statement declares the variables A,B,X,Y,Z to be in the blank common block, and D,E,I to be in the block COMM.

If later, the statement

```
COMMON /COMM/A1,A2,A3(17)/ON/BA,BC
```

occurs, then the variables A1, A2, A3 are added to the end of block COMM and a new block ON is created with variables BA, BC (see section 3.2.3.3).

3.2.3.2 Storage of variables

The size of a common block is the sum of the storage required for each element within it. However, program units which are to be executed together must specify the same sizes for named common blocks which they share.

Variables are allocated consecutive units of storage in the order in which they occur in the program.

3.2.3.3 Multiple references within a program unit

It is permissible for a common block name (or a reference to the blank common block) to occur more than once in a COMMON statement or in more than one COMMON statement in the same program unit.

For example, the statement

```
COMMON X,Y,CHECK/RESULT/A(10)//SUM,AREA/RESULT/B(10),C
```

is equivalent to the following two statements occurring in the same program unit:

```
COMMON X,Y,CHECK/RESULT/A(10)
COMMON SUM,AREA/RESULT/B(10),C
```

Either of these examples will have the same effect as the statement

```
COMMON X,Y,CHECK,SUM,AREA/RESULT/A(10),B(10),C
```

That is, variable and array names are allocated to the common block indicated in the order in which they occur in the program unit.

3.2.3.4 Using the COMMON statement

The common block is an area of storage that will be used in more than one program unit of the program. The name of the block must appear in a COMMON statement in each program unit in which the block is to be used. The names of the variables and arrays within the block may be the same in different program units, but need not be.

Thus if the two statements

```
COMMON X,Y,CHECK/RESULT/A(10)
COMMON XCOORD,YCOORD,CH/RESULT/ARRAY(10)
```

occurred in two different program units, the variables X,Y,CHECK used in the first program unit would occupy the same storage area as XCOORD, YCOORD and CH used in the second program unit. The storage area is called the blank common block. Similarly A(10) in the first program unit and ARRAY(10) in the second program unit will share the storage area of common block RESULTS.

Usually, the variable and array names included in a given common block in one program unit will correspond in number and type to those used in all other program units in which the block is referenced.

For the initialization of named common blocks, see section 3.3. For the equivalencing of common items, see section 3.2.4.3.

3.2.4 The EQUIVALENCE statement

The EQUIVALENCE statement enables variables and array elements of the same or different types used in one program unit to share storage space. Thus, the amount of storage used by a program unit can be reduced. There is no mathematical equivalence, simply a sharing or saving of space.

The statement has the form

```
EQUIVALENCE (a1,a2,a3,...),...,(n1,n2,n3,...)
```

where

a1,a2,a3,...,n1,... are variables, array elements, arrays or character substrings. They must not be function names or dummy arguments of functions or subroutines.

This statement causes all the elements of each list of variables, a1 to am and n1 to nm to share the same storage area. That is a1,a2,a3,...,am will share one area and n1,n2,n3,...,nm will share another.

The subscripts of an array element used in this statement must be integer constant expressions and must correspond in number to the dimensions of the array.

3.2.4.1 Arrays in EQUIVALENCE statement

Since array elements are stored in a fixed order, an equivalence between two elements of different arrays effectively defines an equivalence between other elements. It is important not to contradict these equivalences by further EQUIVALENCE statements.

3.2.4.2 Equivalencing items of different types or length

Variables, arrays and array elements except of type character may be equivalenced with other such items of different types, but wherever one of the equivalenced items of type type1 is assigned a value of type type1 then the value of another item of type type2 that is equivalenced to the first item becomes undefined and should not be referred to in an expression until it has been assigned a value of type type2.

If equivalenced items occupy different amounts of store (for example if a real variable and a complex variable are equivalenced), the starts of the items are aligned. For example, the statement

```
EQUIVALENCE(A,B)
```

where A is of type complex and B is of type real will cause B to share the first four bytes of the storage allocated to A.

An entity of type character may only be equivalenced to another entity of type character. The lengths of the equivalenced entities do not need to be the same.

3.2.4.3 Equivalencing common block items

If one of the items given in the EQUIVALENCE statement appears in a common block then none of the other items given in the same list in the EQUIVALENCE statement may appear in the same or any other common block in the program unit.

Since an array element may be equivalenced to a variable in common, the implicit equivalencing of the rest of the array may extend the size of the common area. Such lengthening of a common block can take place only beyond the last entry in the block and not before the first entry.

For example, the following series of statements:

```
COMMON/BLOCK/A(10),B,C,I,J

DIMENSION TABLE(3,4),ARRAY(4,4)

EQUIVALENCE(A(1),TABLE(1,1))
```

would result in the 12 real elements of TABLE sharing the common storage area with the 10 real elements of array A and the two real variables B and C. If the EQUIVALENCE statement were replaced by

```
EQUIVALENCE(A(1),ARRAY(1,1))
```

then ARRAY would share storage with A,B,C,I and J, and the common block would be extended to hold the remaining elements of ARRAY. The EQUIVALENCE statement could not validly be replaced by

```
EQUIVALENCE(ARRAY(2,2),A(5))
```

since if this were implemented it would result in the common block BLOCK being extended before the first item in the block. Element ARRAY(2,2) is the sixth item of array ARRAY; if it were to be equivalenced with A(5) then ARRAY(2,1) would by implication be equivalenced with A(1) and the block would have to be extended before A(1) to give storage to ARRAY (1,1). This therefore is an invalid EQUIVALENCE statement.

3.3 Assignment of initial values

Initial values may be assigned to variables, arrays, array elements and substrings by using DATA statements.

Initial values may not be assigned to the dummy arguments of function or subroutine subprograms nor to a variable in a function subprogram whose name is also the name of the subprogram or an entry to the subprogram. If initial values are to be assigned to variables or arrays that form part of a common block (or are equivalenced to items in a common block), this must be done in the block data subprogram (see section 3.3.2) and not in any of the other program units in which the commmon block is used. Initial values may only be assigned to named common blocks, and not to the blank common block.

Initialization is carried out when a program unit is loaded, and not when control enters the program unit. Thus when a subroutine or external function is to be entered several times it cannot be assumed that the initial values apply on each occasion of entry.

In the case of character and logical values the type of a value must be the same as that of the variable or array element to which it is being assigned. For arithmetic values, the type of a value should be the same as that of the variable or array element to which it is to be assigned. However, the compiler will perform a conversion between integer and real values if necessary.

3.3.1 The DATA statement

The DATA statement has the form

```
DATA namelist1/valuelist1/,namelist2/valuelist2/
,...,namelistn/valuelistn/
```

where

namelist1,namelist2,... are lists of names of variables, arrays, array elements or substrings, and implied DO lists, separated by commas. The comma after a slash (/) and before a list is optional.

valuelist1,valuelist2,... are value lists as described in section 3.3.1.1 below.

The values given in valuelist are assigned in order to the items given in the corresponding namelist. When an array name appears in namelist it is treated as a list of all the elements of the array, in the order given in section 3.1.3.

The number of items listed in each namelist (counting each array element of any arrays named as one item) must be the same as the number of initial values given in the associated valuelist.

3.3.1.1 Value lists

Values are given in the form of lists of items separated by commas: these items may take either of the forms

```
j
r*j
```

where j is a constant or the symbolic name of a constant and r a non-zero unsigned integer constant or the symbolic name of such a constant. The latter form is equivalent to specifying r copies of the constant j. Thus the list

```
1,3,4.975,.TRUE.,4*8,3E-2
```

will have the same effect as the list

```
1,3,4.975,.TRUE.,8,8,8,8,3E-2
```

3.3.1.2 Implied-DO in a DATA statement

The implied-DO list in a DATA statement has the form:

```
(dlist, i = p1,p2,p3)
```

where

dlist is a list of array element names and implied-DO lists.

i is the name of an integer variable, known as the implied-DO-variable.

p1, p2 and p3 are integer constant expressions, except that the expressions may contain implied-DO-variables of other implied-DO lists that have this implied-DO list within their ranges.

The range of an implied-DO list is the list dlist. An iteration count and the values of the implied-DO-variable are found from p1,p2 and p3 as for a DO-loop (see section 6.3.1), except that the iteration count must be positive. p3 together with the preceding comma may be omitted.

When an implied-DO list appears in a DATA statement, the list items of dlist are specified once for each iteration of the implied-DO list with

the appropriate substitution of values for any occurrence of the implied-DO- variable i. The appearance of an implied-DO-variable in a DATA statement does not affect the definition of a variable of the same name elsewhere in the same program unit.

Each subscript expression in the list dlist must be an integer constant expression, except that the expression may contain implied-DO-variables of implied-DO lists that have the subscript expression within their ranges.

Example

```
DATA ((A(J,I), I=1,J), J=1,5) / 15*0. /
```

3.3.1.3 Character values

Initialization with character values is restricted to variables and array elements of type character and character substrings. Character values are held as strings of eight bit characters. One byte of storage will hold one character.

If a character constant contains fewer characters than the number required to fill the variable, array element or substring to which it is being assigned as an initial value, space characters will be added to the right-hand end of the string to make the number of characters equal to the length of the variable, array element or substring.

If a character constant contains more characters than the number required to fill the variable, array element or substring to which it is being assigned as an initial value, the surplus rightmost characters in the constant are ignored.

For example, the character constant

```
'HEAD'
```

could be assigned as an initial value to a character variable of length 4, and would fill it exactly. If the character constant

```
'ARRAY#ELEMENTS'
```

were assigned as an initial value to an element of a character*8 array, the element would hold ARRAY#EL.

3.3.1.4 Examples of initial value assignment

The effect of the statement

```
DATA I/1/,J/1,3*0,1,3*2,1/
```

where J is a 3 x 3 array, is to assign to I the initial value 1, and to assign to J the values:

```
1  0  2
0  1  2
0  2  1
```

The effect of the statement

```
DATA A,B,C,D/4*0.0/,HEAD/'VALUES'/
```

is to assign the initial value 0.0 to each of real variables A,B,C and D, and to insert the characters VALUES## in the character*8 variable HEAD.

3.3.2 Block data subprogram

Block data subprograms are used to give initial values to items in named common blocks by means of DATA statements. A block data subprogram must start with a BLOCK DATA statement and end with an END statement and may only contain the following kinds of statement:

IMPLICIT statements

PARAMETER statements

Type specification statements

DIMENSION statements

COMMON statements

EQUIVALENCE statements

DATA statements

A block data subprogram is never executed.

If variables and arrays from a common block are named in a COMMON statement in a block data subprogram, then the total storage area in the common block must be specified completely. For example, an array must have its dimension information specified in the COMMON statement

or in a DIMENSION or a type statement in the subprogram, even if it is not named in DATA statements.

If any part of a common block is being given an initial data value then a complete set of specification statements for the whole block must be included before any part is initialized. This means that DIMENSION, COMMON, EQUIVALENCE and type specification statements must come before DATA statements for each common block.

A FORTRAN program may contain more than one block data subprogram but any one common block can be referred to in only one block data subprogram. Initial data values may be entered into more than one common block in a single block data subprogram. Items in the blank common block (see section 3.2.3.1) cannot be given initial values.

The block data subprogram may be given an optional name, in which case the name must not be the same as any local name in the subprogram, nor the same as the name of any external procedure, main program, common block or other block data subprogram in the same executable program. There must not be more than one unnamed block data subprogram in an executable program.

3.3.2.1 Example

The following block data subprogram gives initial values to some items of the common blocks ERC and RCC. All the items in each block are specified completely.

```
BLOCK DATA
REAL B(4),Z(3)
DOUBLE PRECISION Z (3)
COMPLEX C
COMMON/ERC/C,A,B/RCC/Z,Y
DATA B,Z,C/1.0,1.2,2*1.3,3*7.654321D0,(2.4,3.76)/
END
```

CHAPTER FOUR

EXPRESSIONS

In FORTRAN, expressions may be used in many different statements in a variety of contexts. There are three kinds of expression: arithmetic expressions, logical expressions and character expressions. Arithmetic expressions have numerical values; logical expressions have logical values; and character expressions have character values: this chapter gives the rules for forming and evaluating these kinds of expression.

4.1 Arithmetic expressions

An arithmetic expression is a sequence of arithmetic elements of type integer, real, double precision or complex, combined by arithmetic operators and parentheses. The type of an arithmetic expression depends upon those of its constituents; see section 4.1.6.

4.1.1 Arithmetic elements

An arithmetic element can be a numerical constant, an arithmetic symbolic constant name, a variable name, an array element reference or a function reference (see section 7.2.1.1). For example, the following are valid arithmetic elements:

```
7E23   VARI   A(1,3)   SIN(X)
```

The simplest arithmetic expression is one that consists of only one arithmetic element: the expression is then of the same type as the element. The term expression is used in this manual to include elements as well as more complex expressions.

4.1.2 Arithmetic operators and parentheses

Arithmetic operators are used to combine arithmetic elements or other arithmetic expressions to give more complex arithmetic expressions. The arithmetic operators are:

Operator	Meaning
+	Addition
-	Subtraction or negation
*	Multiplication
/	Division
**	Exponentiation

Some simple examples of arithmetic expressions using only one operator are:

```
-A
A+B
A*B    equivalent to the algebraic expression
           a x b or ab
A**B   equivalent to the algebraic expression a**(b)
```

Parentheses are used to enclose arithmetic expressions which form part of a more complex arithmetic expression. The parenthesized expressions are evaluated as separate entities; this usage of parentheses is therefore equivalent to normal mathematical usage.

4.1.3 Rules

When writing arithmetic expressions, the following rules must be observed:

1 Arithmetic elements must be separated by an arithmetic operator

2 No two operators may be adjacent

3 The operators + and - must be followed by an element: the other operators must be both preceded and followed by elements

Thus the following are not valid arithmetic expressions:

```
A.B      (Rule 1: A multiplied by B must be written as A*B)
A**-B    (Rule 2: A to the power -B must be written as A**(-B))
*B       (Rule 3: this on its own is meaningless, while -B is
             valid)
```

4.1.4 Order of evaluation

Arithmetic expressions are evaluated in general from the innermost set of parentheses outwards. Within each set of parentheses or each unparenthesized expression, the order of evaluation is from left to right, except when the precedence of operators dictates otherwise. This precedence is:

1 Function references

2 Exponentiations

3 Multiplications and divisions

4 Additions and subtractions

This order of precedence determines the sequence of operations in the evaluation of an expression. The first two operators are compared and, if the first takes precedence over or is equal to the second, then the first operation is performed. If the second takes precedence over the first, the third operator is compared with the second and so on. When the end of the expression is reached, any remaining operations are performed, reading from right to left.

For example, in the expression

```
A*B+C*D**I
```

the operations are performed as follows:

```
1    A*B   = E1     (intermediate result) (E1+C*D**I)

2    D**I  = E2     (intermediate result) (E1+C*E2)

3    C*E2  = E3     (intermediate result) (E1+E3)

4    E1+E3          (final operation)
```

If one exponentiation operator follows immediately after another, the evaluation is from right to left. Thus

A**B**C

is evaluated as follows:

1 B**C = E1 (intermediate result)

2 A**E1 (final operation)

A series of multiplications and divisions is evaluated from left to right. Under some circumstances this could lead to results that are inaccurate owing to rounding errors or to a lack of precision in the values of the elements in use. If such errors are possible, the programmer may use parentheses to control the order of evaluation so as to produce the most precise result.

Where part of an expression is contained within parentheses, that part is evaluated first, and the result obtained is used in evaluation of the expression as a whole. Where nested parentheses occur, that part of the expression contained within the innermost set is evaluated first.

The sign of a signed quantity takes the same precedence as the addition or subtraction sign. Thus

A = -B is treated as A = 0 - B

A = -B*C is treated as A =-(B*C)

A = -B+C is treated as A =(-B) + C

4.1.5 Examples of arithmetic expressions

The expression

ARRAY(2,10)-COS(Z)/(2*PI)

is evaluated as follows:

COS(Z)	= E1	Function reference (see section 7.2.1.1)
(2*PI)	= E2	parenthesized expression
E1/E2	= E3	Division
ARRAY(2,10)-E3	= final result	Subtraction

The expression

```
A+B*C/D*(P-1)-3.0**(Q+R)+2.0/X**2
```

is evaluated as follows:

```
B*C       = E1
E1/D      = E2
P-1       = E3
E2*E3     = E4
A+E4      = E5
Q+R       = E6
3.0**E6   = E7
E5-E7     = E8
X**2      = E9
2.0/E9    = E10
E8+E10    = final result
```

Although the order in which expressions are evaluated in these examples may not be exactly that described, it will be mathematically equivalent. However, the order of evaluation implied by the presence of parentheses will be followed.

4.1.6 Determination of the type of an expression

The value of an arithmetic expression can be of any of the types integer, real, double precision or complex. The evaluation of an expression is carried out in simple steps (as in the example in section 4.1.5). The types of the elements involved in each step determine the type of the value produced by that step. The type of the final expression can be found by following through the steps of the evaluation, noting the types of the intermediate values at each stage.

Table 4.1 gives the type of an expression composed of two simpler expressions of type A and type B.

4.1.7 Integer arithmetic

The following special considerations apply when both the arguments of an arithmetic operation are of type integer:

1 A result of type integer is that integer obtained by truncating the mathematical result towards zero:

```
 15/4      =    3        ( 3.75)
-15/4      =   -3        (-3.75)
4**(-1)    =    0        ( 0.25)
```

2 A series of multiplication and division operations on integer quantities will always proceed from left to right

4.1.8 Arithmetic constant expressions

An arithmetic constant expression may contain only arithmetic constants and arithmetic symbolic constants.

The exponentiation operator is not permitted unless the exponent is of type integer.

Arithmetic constant expressions may be used in PARAMETER or DATA statements.

4.1.9 Integer constant expressions

An integer constant expression is an arithmetic constant expression in which each constant or symbolic constant is of type integer.

Integer constant expressions may be used in PARAMETER statements, as a character length or array bound specifier in a specification statement or as an array element subscript or character substring position expression in EQUIVALENCE or DATA statements.

Table 4.1

Expression types

Type of A	Integer	Real	Double precision	Complex
Type of B				
Integer	Integer	Real	Double precision	Complex
Real	Real	Real	Double precision	Complex
Double precision	Double precision	Double precision	Double precision	Prohibited
Complex	Complex	Complex	Prohibited	Complex

4.2 Character expressions

A character expression is a sequence of one or more character elements separated by character operators.

4.2.1 Character elements

A character element can be a character constant, a character symbolic constant name, a character variable name, a character array element reference, a character substring reference or a character function reference. For example, the following are valid character elements:

```
'SOME TEXT'    CVAR    NAME(I)
```

provided that CVAR has been specified as of type character and that NAME has been specified as a one-dimensional array of type character, or is a function of type character.

4.2.2 Character operator and parentheses

The concatenation operator, //, is a character operator that is used to concatenate two character elements to produce a character string of type character whose length is the sum of the lengths of the two elements. Except in a character assignment statement, a character expression must not concatenate a character element whose length specification is an asterisk in parentheses unless the element is the symbolic name of a constant. Parentheses may have a cosmetic effect on a character expression but they do not affect the value found.

Example

```
'AN'//('AESTHETIC'//'ALLY')
```

is the same as

```
('AN'//'AESTHETIC')//'ALLY'
```

each producing

```
'ANAESTHETICALLY'
```

4.3 Logical expressions

A logical expression is a sequence of logical elements and relational expressions combined by logical operators and parentheses. The value of a logical expression is always either .TRUE. or .FALSE..

4.3.1 Logical elements

A logical element is a constant, a symbolic constant, a variable, an array element or a function reference of type logical (see section 2.1). The value of a logical element must be either .TRUE. or .FALSE. For example, the following are valid logical elements:

```
.TRUE.    LVAR    STATUS(1,3)    OK(B)
```

provided that LVAR has been specified as of type logical, that STATUS has been specified as a two dimensional array of type logical, and that OK is a function of type logical. The simplest logical expression is one that consists of only one logical element.

4.3.2 Relational expressions

A relational expression has the form

e1 r e2

where

The operands e1 and e2 are both arithmetic expressions or are both character expressions.

r is one of the following relational operators:

Operator	Meaning
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

The periods are essential.

A complex operand is only permitted when the relational operator is .EQ. or .NE.

If e1 and e2 are character expressions of different lengths then the shorter operand is considered as if it were extended with blanks on the right to the length of the longer operand.

If the relation indicated by the relational operator between the two arithmetic expressions is true, then the value of the relational expression is .TRUE.. Similarly, if the relation is false, the value of the expression is .FALSE..

A relational expression is equivalent to a logical element for the purpose of constructing further logical expressions; it need not be enclosed in parentheses.

4.3.3 Logical operators and parentheses

Logical operators are used to combine logical elements, relational expressions or other logical expressions to give more complex logical expressions. The logical operators are defined as follows, where a and b are logical expressions:

Operator	Meaning
NOT.a	This expression has the value .TRUE. if a has the value .FALSE. and has the value .FALSE. if a has the value .TRUE.
a.AND.b	This expression has the value .TRUE. if both a and b have the value .TRUE.. It has the value .FALSE. if either a or b or both have the value .FALSE.
a.OR.b	This expression has the value .TRUE. if either a or b or both are .TRUE.. It has the value .FALSE. if both a and b are .FALSE.
a.EQV.b	This expression has the value .TRUE. if a and b both have the same value .TRUE. or .FALSE.. It has the value .FALSE. if a and b have different truth values
a.NEQV.b	This expression has the value .TRUE. if a and b do not both have the same value .TRUE. or FALSE. It has the value .FALSE. if a and b have the same truth value.
	The periods are essential.

Parentheses may be used to enclose logical expressions which form part of more complex logical expressions. Their usage here is analogous to their usage in arithmetic expressions. An expression enclosed in parentheses must satisfy the rules given below.

4.3.4 Rules

When writing logical expressions, the following rules must be observed:

1 If arithmetic expressions appear, they must be in pairs separated by relational operators

2 Logical elements (and relational expressions) must be separated by logical operators

3 No two logical operators may be adjacent unless the first is one of .AND., .OR., .EQV. or .NEQV. and the second is .NOT.

4 The logical operator .NOT. must be followed by, but must not be preceded by, a logical element. The logical operators .AND., .OR., .EQV. and .NEQV. must be preceded by a logical element and must be followed either by a logical element or by the operator .NOT.

Thus the following are not valid logical expressions:

A.AND.7.0 (Rule 1: A.AND.7.0.GT.B is valid if B is an arithmetic element and A is of type logical)

A+C (Rule 2: A.AND.C is valid if A and C are of the type logical. A+C is, of course, a valid arithmetic expression if A and C are arithmetic elements)

A.AND..OR.C (Rule 3: A.AND..NOT.C is valid)

A.NOT.C (Rule 4: .NOT.C.AND.A.OR..NOT.D illustrates the various possible valid uses of operators)

4.3.5 Order of evaluation

Logical expressions are in general evaluated starting at the innermost set of parentheses and working outwards. Within one set of parentheses or within one expression the order of evaluation is as follows:

1 Evaluation of functions

2 Exponentiation

3 Multiplication and division

4 Addition and subtraction

5 Relational expressions

6 .NOT. operations

7 .AND. operations

8 .OR. operations

9 .EQV. or .NEQV. operations

Rounding errors (see section 4.1.4) cannot occur with logical expressions since these may only take the values .TRUE. or .FALSE..

No more code than necessary is actually obeyed when evaluating an expression. Some parts of a logical expression may not be evaluated every time the expression occurs. In the following example

```
A.OR.LGF(.TRUE.)
```

the LGF function need not be called to evaluate the expression when A has the value .TRUE..

4.3.6 Examples of relational and logical expressions

The expression

```
B**2.GT.4.*A*C
```

is a relational expression. The arithmetic expressions are evaluated as described in section 4.1.4. Then the relational operation is performed. The resulting value will be .TRUE. or .FALSE..

The expression

```
JOB.EQ.3.AND.AGE.LT.18
```

is evaluated as follows:

```
JOB.EQ.3  = E1                Relational expressions

AGE.LT.18 = E2

E1.AND.E2 = final result      Logical expression
```

The expression

```
A.OR.B.OR.C.AND.(P.OR.Q).AND.(I.LT.1.OR.J.EQ.0)
```

is effectively evaluated as follows:

```
A.OR.B     = E1

P.OR.Q     = E2

C.AND.E2   = E3

I.LT.1     = E4

J.EQ.0     = E5

E4.OR.E5   = E6

E3.AND.E6  = E7

E1.OR.E7   = final result
```

However, certain stages are sometimes unnecessary. For example in the above expression

If A is .TRUE. then the result is .TRUE.

If A is .FALSE., B is .FALSE. and C is .FALSE.
the result is .FALSE.

The order of evaluation in these examples may not be exactly as shown, but will be logically equivalent.

CHAPTER FIVE

ASSIGNMENT STATEMENTS

Assignment statements are used in FORTRAN to assign new values to variables or array elements, replacing any existing values. Assignments are executable statements. There are three types of assignment statements: arithmetic assignment, logical assignment and character assignment. The rules for forming assignment statements are given in this chapter.

5.1 Arithmetic assignment statements

An arithmetic assignment statement is used to assign a new value to a variable or an array element of type integer, real, double precision or complex. The statement takes the form:

name = expression

where

name is the name of the variable or array element of type integer, real, double precision or complex.

expression is an arithmetic expression (see section 4.1).

When an assignment statement is encountered in a program the expression part is evaluated and the resulting value is assigned to the variable or array element with the name name. The variable or array element then retains this value until it is assigned a new value. Any previous value of the variable or array element is lost.

The variable or array element name need not be of the same type as the expression. The expression is evaluated according to the rules given in section 4.1 and the resulting value is assigned to the variable or array element after any necessary type transformations as indicated in the table below.

Type of expression / Type of name	Integer	Real or Double precision	Complex
Integer	Assign	Fix and assign	Fix and assign real part
Real or Double precision	Float and assign	Assign	Assign real part
Complex	Float and assign real part, set imaginary par to zero	Assign real part, set imaginary part to zero	Assign

Notes:

1 Assign means assign the unchanged result to the variable or array element name.

2 Float means convert the result to type real.

3 Fix means truncate towards zero any fractional part of the result and convert the remaining value to type integer. An overflow error condition will occur if the result is outside the range of integer values.

4 Where real and double precision numbers of different lengths are involved, as much precision is preserved as possible. If a double precision value is assigned to a real variable or array element name, then the value is truncated as necessary. If a real value is assigned to a double precision variable or array element name, then the mantissa is extended with zeros.

Some examples of arithmetic assignment statements follow:

```
VALUE = 38.7654
COUNT = COUNT1
NEXT = ITEM(3,1)
MATRIX(7) = MATRIX(4*I) + MATRIX(3)
M = (2*M-1)/Q-M
```

5.2 Logical assignment statements

A logical assignment statement assigns a new value to a variable or array element of type logical.

The statement takes the form:

```
name = expression
```

where

name is the name of a logical variable or array element.

expression is a logical expression (see section 4.3).

When a logical assignment statement is encountered in a program, the logical expression is evaluated and the resulting value is assigned to the variable or array element with the name name. The variable or array element then retains this value until it is assigned a new value. Any previous value of the variable or array element is lost.

Some examples of logical assignment statements follow:

```
COURSE = .TRUE.
QUAD = B**2.GT.4*A*C
STATUS = PUPIL.AND.AGE.LT.21
RES = A.AND.(B.OR.(C.AND..NOT.D)
```

5.3 Character assignment statements

A character assignment statement assigns a new value to a variable, substring or array element of type character. The statement takes the form:

```
name = expression
```

where

name is the name of a character variable, substring or array element.

expression is a character expression (see section 4.2).

Note that no character position in name may be referenced in expression.

The variable, substring or array element need not be of the same length as the character expression. If the expression is shorter, then spaces are added to the right on assignment; but if the expression is longer,

then truncation will occur on the right. An example of a character assignment statement is

```
CH = 'TEST DATA'
```

CHAPTER SIX

CONTROL STATEMENTS

Execution of a FORTRAN program begins at the first executable statement of the main program. Subsequent statements are executed in the order in which they occur until a control statement is encountered. Control statements are used to transfer control from one part of a program to another. This chapter describes those statements used for transferring control within a program unit. Statements used to transfer control from one program unit to another are described in Chapter 7.

6.1 GO TO statements

A GO TO statement is an executable statement that is used to transfer control to another executable statement in the same program unit. There are three types of GO TO statements: unconditional GO TO, computed GO TO and assigned GO TO.

6.1.1 Unconditional GO TO

An unconditional GO TO statement has the form:

```
GO TO label
```

where label is the label of the executable statement to which control is to be transferred (see section 1.3.3).

Each time a GO TO statement of this form is encountered, control is transferred to the statement with the label label. This statement must be in the same program unit as the GO TO statement. The first executable statement after the GO TO statement should be labelled unless it is an ELSE IF, ELSE or END IF statement, otherwise control can never reach it.

The following is an example of an unconditional GO TO statement:

```
GO TO 3
```

6.1.2 Computed GO TO

A computed GO TO statement transfers control to one of a list of statements, depending upon the computed value of an expression. The statement has the form:

```
GO TO (label1,label2,...,labeln),i
```

where

each label is the label of an executable statement in the same program unit as the GO TO statement.

i is an integer expression: the preceding comma is optional.

When a computed GO TO statement is encountered i should have a value in the range 1 to n. If i has the value j, then control is passed to the executable statement with label labelj.

The same label may appear more than once in the list.

If i is not in the required range, then the next statement in sequence will be executed, as the GO TO statement has no effect.

An example of the use of a computed GO TO statement follows:

```
COUNT = 3
.
.
.
GO TO(14,21,20,15,20),COUNT
```

Control is transferred to the statement with label 20, the third label in the list.

6.1.3 Assigned GO TO and ASSIGN statements

An assigned GO TO statement transfers control to one of a list of labelled executable statements depending on the value assigned, by an ASSIGN statement, to an integer variable. The statement has one of the forms:

```
GO TO i,(label1,label2,...,labeln)
GO TO i
```

where

i is an integer variable.

each label is the label of an executable statement in the same program unit as the assigned GO TO statement. If present, the bracketed list of labels must contain all those labels that may be assigned to the variable.

The ASSIGN statement has the form:

```
ASSIGN labelj TO i
```

where

labelj is the label of an executable statement in the same program unit as the ASSIGN statement:

i is the integer variable to be used in an assigned GO TO statement.

Each time an assigned GO TO statement is encountered, control is transferred to the statement with label labelj where labelj is the value last assigned to the variable i in an ASSIGN statement. If, when the statement is encountered, the variable i has not been assigned a value by an ASSIGN statement in the same program unit, then the effect of the GO TO statement is unpredictable.

A variable may have, at different times, a statement label value assigned by an ASSIGN statement or an integer value assigned in any other way. An attempt to use a variable which has a statement label value when an integer value is required, or vice versa, will have an unpredictable effect. The label list when present enables the compiler potentially to check that an appropriate label has been assigned.

For example, the statements

```
ASSIGN 57 TO MEANS
.
.
.
GO TO MEANS,(92,3,9999,57)
```

will result in a transfer of control from the GO TO statement to the statement with label 57. If the GO TO statement were written

```
GO TO MEANS
```

the same result would be achieved.

6.2 IF statements

An IF statement allows the program to take different actions depending on a particular condition. Thus an IF statement may have one result the first time it is executed in a program and a different result on a subsequent execution if the relevant condition has altered. There are three types of IF statement: arithmetic IF, logical IF and block IF.

6.2.1 Arithmetic IF

An arithmetic IF statement transfers control to one of three statements depending on the value of an arithmetic expression. It has the form:

```
IF(expression)label1,label2,label3
```

where

expression is an arithmetic expression of type integer, real or double precision.

label1,label2,label3 are the labels of executable statements in the same program unit as the IF statement. The same label may be used more than once.

The statement causes control to be transferred to the statement with label label1, label2 or label3 depending on whether the value of the expression is less than, equal to, or greater than zero respectively.

For example, the statement

```
IF(B*B-4.0*A*C) 100,101,102
```

would have the following effects:

If B**(2) - 4AC<0, control is transferred to the statement with label 100.

If B**(2) = 4AC, control is transferred to the statement with label 101.

If B**(2) - 4AC>0, control is transferred to the statement with label 102.

Note: Real expressions rarely evaluate exactly to zero.

6.2.2 Logical IF

A logical IF statement tests whether a logical expression has the value .TRUE. or .FALSE.; if it has the value .TRUE. then a particular executable statement included in the IF statement is executed. The statement has the form:

```
IF (expression) statement
```

where

expression is a logical expression.

Statement is any executable statement except a DO, block IF, ELSE, ELSE IF, END IF, END statement or another logical IF statement.

If expression, when evaluated, has the value .TRUE., statement is executed; if expression, when evaluated, has the value .FALSE., statement is not executed.

For example, if the statement

```
IF(B**B.LT.4.0*A*C) GO TO 100
```

is executed when B**(2) < 4AC, control is transferred to the statement with label 100. Otherwise the GO TO statement is ignored.

Some other examples of logical IF statements follow:

```
IF(SUM+TERM.GE.9E7.OR.TERM.LT.1E-2)CALL CHECK
IF(COUNT.EQ.60) IF(COUNT1-14)26,27,28
IF(I.LT.0)I = -1
```

6.2.3 Block IF

A block IF statement is used with the END IF statement and optionally with the ELSE and ELSE IF statements to control the execution of a block of consecutive executable statements. The block of statements is divided into IF-blocks, ELSE-blocks and ELSE-IF blocks. The block IF statement and IF-blocks, the ELSE statement and ELSE-blocks, and the ELSE IF statement and ELSE IF blocks are described in sections 6.2.3.1, 6.2.3.2 and 6.2.3.3 respectively. The END IF statement is described in section 6.2.3.4.

Every statement in an IF-block, an ELSE block or an ELSE-IF block has an IF- level. The IF-level of a statement s is n1,-n2, where n1, is the number of block IF statements from the beginning of the program unit down to and including s, and n2 is the number of END IF statements in

the program unit down to but not including s.

The IF-level of every statement must be non-negative, the IF-level of each block IF, ELSE, ELSE IF and END IF statement must be positive: the IF-level of the END statement must be zero.

Example

```
                                                          IF-level
.                                                             0
IF (expression) THEN        )                                 1
.                           )                                 1
.                           )   IF-block                      1
.                           )                                 1
ELSE                        )                                 1
.                                                             1

IF (expression) THEN        )                                 2
.                           )                                 2
.                           )   IF-block                      2
.                           )                                 2

ELSE IF (expression) THEN   )                 ELSE-block      2
.                           )                                 2
.                           )   ELSE IF-block                 2
.                           )                                 2
END IF                      )                                 2
.                                                             1
END IF                                                        1
.                                                             0
```

6.2.3.1 The block IF statement and IF-blocks

The block IF statement has the form:

IF (expression) THEN

where expression is a logical expression.

An IF-block consists of all the executable statements following the block IF statement down to but not including the next ELSE, ELSE IF or END IF statement that has the same IF-level as the block IF statement. An IF-block may be empty.

When the block IF statement is executed, expression is evaluated and if it has the value .TRUE. then the IF-block is executed. If the IF-block

is empty and the value of expression is .TRUE. then control passes to the next END IF statement at the same IF-level as the block IF statement. If the value of expression is .FALSE. then cont ELSE IF or END IF statement that has the same IF-level as the block IF statement.

Control cannot be passed into an IF-block. If the last statement in an IF-block does not pass control elsewhere then control passes to the next END IF statement that has the same IF-level as the block IF statement preceding the IF-block.

6.2.3.2 The ELSE statement and ELSE-blocks

The ELSE statement has the form:

```
ELSE
```

An ELSE-block consists of all the executable statements following the ELSE statement down to but not including the next END IF statement that has the same IF-level as the ELSE statement. An ELSE-block may be empty.

An END IF statement of the same IF-level as the ELSE statement must be included before an ELSE IF or another ELSE statement of the same IF-level.

Control cannot be passed into an ELSE-block. No reference may be made to the statement label, if any, of the ELSE statement.

6.2.3.3 The ELSE IF statement and ELSE IF blocks

The ELSE IF statement has the form:

```
ELSE IF (expression) THEN
```

where expression is a logical expression.

An ELSE IF-block consists of all the executable statements following the ELSE IF statement down to but not including the next ELSE, ELSE IF or END IF statement that has the same IF-level as the ELSE IF statement. An ELSE IF-block may be empty.

When the ELSE IF statement is executed, expression is evaluated and if it has the value .TRUE. the ELSE IF-block is executed. If the ELSE IF-block is empty and expression has the value .TRUE. then control passes to the next END IF statement that has the same IF-level as the ELSE IF statement. If the value of expression is .FALSE. then control passes to the next ELSE, ELSE IF or END IF statement that has the same

IF-level as the ELSE IF statement.

Control cannot be passed into an ELSE IF-block. No reference may be made to the statement label, if any, of the ELSE IF statement.

6.2.3.4 The END IF statement

The END IF statement has the form:

```
END IF
```

Each block IF statement must be matched by a separate END IF statement.

6.3 DO loops

A DO loop is a series of statements that is to be executed several times. It is headed by a DO statement which specifies the number of times the loop is to be executed and also specifies the last statement included in the loop. The range of a DO loop is the series of statements from the statement after the DO statement down to and including the terminal statement.

If a DO statement appears within an IF-block, ELSE-block or ELSE IF-block, then the range of the DO-loop must be wholly within that block.

6.3.1 DO statements

The DO statement has the form:

```
DO label i = p1,p2,p3
```

where

label is the label of the terminal statement (see section 6.3.2 below for further details)

i is the DO-variable.
p1 is the initial parameter.
p2 is the terminal parameter.
p3 is the incrementation parameter and may be omitted, together with the preceding comma.

The terminal statement must be in the same program unit as the DO statement and must occur later in the program unit than the DO statement. The DO-variable must be an integer, real or double precision

variable. The three parameters must all be integer, real or double precision expressions. The incrementation parameter must not be zero at the time of execution of the DO loop. If the incrementation parameter is omitted, it is assumed to have a value of 1.

When a DO statement is encountered in a program, the values of the three parameters are calculated and, if necessary, converted according to the rules given in section 5.1 to be of the same type as the DO-variable.

The value of p1 is assigned to the DO-variable and the iteration count is found thus:

```
MAX(INT((p2 - p1 + p3)/p3), 0)
```

If this count is non-zero then execution of the first statement in the range of the DO-loop begins. When the terminal statement is reached the DO-variable is incremented by p3. The iteration count is decremented by one. If it is non-zero, control then returns to the first statement in the range of the DO-loop. If it is zero, the DO-statement is satisfied: the DO-variable retains its current value and control passes to the next executable statement following the terminal statement.

Variables and array elements used in the expressions for the parameters p1, p2 and p3, and the DO-variable i, may be referenced within a DO loop, but the value of i must not be altered by any statement within the range of the DO statement.

For example, in the series of statements

```
SUMSQ = 0.0
SUM = 0.0
DO 27 J = 1,MAX,2
SUM = SUM + PART(J+1)
27 SUMSQ = SUMSQ + PART(J)*PART(J)
```

the range of the DO statement is the two statements following it. The effect of the sequence is to set SUM equal to PART(2)+PART(4)+...+PART(x+1), where x is the greatest odd integer less than or equal to MAX, and to set SUMSQ equal to the sum of the squares of PART(1),PART(3),...,PART(x).

Control may be transferred out of a DO loop before the DO statement is satisfied (for example, by use of a GO TO statement): in this case the control variable retains its current value (see section 6.3.4). Transfer into the range of a DO loop from outside the range is forbidden.

Execution of a function reference or a CALL statement that appears

within the range of a DO-loop is permitted. If control is returned from the subprogram by means of an extended form of the RETURN statement (see section 7.2.2.2) to a statement not in the range of the DO-loop, then control cannot be transferred into the range of the DO-loop.

6.3.2 Terminal statements

The terminal statement must not be any of the following statements:

1 Unconditional GO TO

2 Assigned GO TO

3 RETURN

4 STOP

5 DO

6 Arithmetic IF

7 Block IF

8 ELSE

9 ELSE IF

10 END IF

11 END

12 Logical IF containing any one of the following:

(a) DO

(b) Block IF

(c) ELSE

(d) ELSE IF

(e) END IF

(f) END

(g) Another logical IF

A labelled CONTINUE statement may be used as the terminal statement to overcome this restriction (see section 6.4).

6.3.3 Nested DO loops

The statements included in the range of a DO statement may include other DO statements; the DO loops are then said to be nested. In a system of nested DO loops, the range of any inner DO statement must be completely contained in the range of any outer DO statements. However, DO statements may share a terminal statement. If two or more DO statements share the same terminal statement then the DO-variable for any outer DO is not increased and tested until all inner DOs are satisfied.

For example the sequence of statements

```
      DIMENSION A(10,10),B(10,10),C(10,10)
      .
      DO 20 J = 1,10
      DO 20 I = 1,10
   20 C(I,J) = A(I,J)+B(I,J)
```

forms the elements of array C by adding together the corresponding elements of arrays A and B. When the first DO statement is encountered J is set equal to 1. Then the second DO statement is encountered, and the inner DO loop is performed with J equal to 1 and I varying from 1 up to 10. Only after that is J increased to 2.

6.3.4 Transfer of control in DO loops

Any transfers of control may occur within the range of a DO statement except that, if a statement is the terminal statement for more than one DO statement, then control can be transferred to it only from the range of the innermost DO having that terminal statement. A CONTINUE statement (see section 6.4) may be used to overcome this restriction.

Any transfers of control from inside to outside a DO loop are allowed. The DO- variable retains its current value.

For example, the sequence of statements

```
      SUM = 0.0
      DO 25 J = 1,100
      IF (A(J).GT.50)GO TO 80
   25 SUM = SUM + A(J)
      ...
   80 statement
```

forms the sum of the elements of an array, except that if any element is greater than 50, control passes to statement 80 outside the DO loop.

6.4 CONTINUE statements

The CONTINUE statement is a dummy statement and causes no action. It has the form

```
CONTINUE
```

This statement is most often used to give distinct terminal statements to a nest of DO loops. It is also useful for avoiding the statements forbidden in section 6.3.2.

For example, in the sequence of statements

```
      X = 0
      DO 200 I = 5, 100, 5
      Y = A(I,1)
      IF(Y.LT.0.0)GO TO 200
      X = X + Y
      DO 300 J = 2, 150
  300 X = X + A(I,J)
  200 CONTINUE
```

the CONTINUE statement is necessary to allow the IF statement to transfer control to the terminal statement of the outer loop without also transferring control to the terminal statement of the inner loop.

6.5 STOP statements

A STOP statement terminates the execution of the program. It has one of the following forms:

```
STOP
STOP n
STOP 'message'
```

where

n is a string of one to five digits

'message' is a literal constant enclosed in apostrophes.

When a STOP statement is encountered in a program, no further statements are
executed, the run is terminated and a message of one of the following forms:

```
STOP
STOP n
STOP message
```

is reported to the user.

6.6 PAUSE statements

A PAUSE statement causes the program to output a message, and then continue.

The statement has one of the following forms:

```
PAUSE
PAUSE n
PAUSE 'message'
```

where

n is a string of 1 to 5 decimal digits.

'message' is a literal constant enclosed in apostrophes.

Execution of this statement causes a message of one of the following forms to be reported to the user:

```
PAUSE
PAUSE n
PAUSE message
```

Execution continues.

CHAPTER SEVEN

PROGRAM UNITS AND TRANSFER OF CONTROL BETWEEN THEM

As described in Chapter 1 a program is made up of program units: one main program and, possibly, other program units called subprograms. The block data subprogram, described in section 3.3.2, contains only non-executable statements and control never enters it. The remaining subprograms, called procedures, are described below in section 7.1. Section 7.2 describes the transfer of control between program units, involving the use of CALL and RETURN statements and function references. The following sections describe the transfer of values between program units, the correspondence between dummy and actual arguments of procedures (see also section 7.1) and the facility of multiple entry into a subprogram.

7.1 Procedures

Procedures normally contain sequences of statements that carry out a process that is likely to be repeated in the execution of the program. It is convenient for the programmer to write out such a sequence only once, so he writes a procedure using dummy arguments and declares it in the program once in that form.

A dummy argument is a name that is used in a procedure at the declaration stage. Dummy arguments represent the values that will be associated with the procedure when it is actually called later in the program. At each call of the procedure, the values required in that particular call are substituted for the dummy arguments; these substituted values are called the actual arguments.

Besides avoiding the need to write out repeated processes each time, procedures also form logical subdivisions of the program. These subdivisions may be written and tested quite separately from the main body of the program if deired.

The location of the procedure declaration, the scope of the validity of the use of the procedure and how to call the procedure depend on the kind of procedure. There are four kinds of procedure:

1 Intrinsic function

2 External function

3 Statement function

4 External subroutine

Each of these is described in detail in sections 7.1.2 and 7.1.3.

7.1.1 Differences between function and subroutine subprograms

A function subprogram is used to evaluate a specific function and to substitute a value for the function reference in the calling program unit. This class of subprogram would not normally be used to change the value of any variables or array elements in the referencing program unit though facilities are available for the programmer to do so if desired.

A subroutine subprogram is not specifically used for the calculation of a single value but may perform any series of operations. From this basic difference some others follow:

1 Function subprograms are entered by function references; subroutine subprograms are entered by CALL statements.

2 Function subprograms have a type; subroutine subprograms do not.

3 Results from a function subprogram are returned principally via the function value; results from subroutine subprograms are returned only via dummy arguments and common blocks.

7.1.2 Functions

7.1.2.1 Intrinsic functions

The Fortran compiler provides a number of standard functions. These are called intrinsic functions and include certain standard mathematical functions, such as sine and cosine, and type conversion functions.

Use of a function name in an EXTERNAL statement (see section 7.3.4), will mean that the name will not be recognized as an intrinsic function name.

Intrinsic function names are either generic or specific. General function names provide an automatic function selection facility. This

facility allows the programmer to use a single generic name when requesting a Fortran-supplied function which has several specific names, depending on argument type. The proper function is selected by the Fortran Compiler, based on the type of the arguments of the function. With this facility the programmer can, for example, use the generic name SIN to refer to any sine routine, rather than explicitly calling SIN for REAL arguments, DSIN for DOUBLE PRECISION arguments or CSIN for COMPLEX arguments.

Generic function names may have specific function names associated with them. If a specific name is used, then the arguments must be of the correct type otherwise a compile time fault will be reported. Some functions do not have a generic name. The specific names that identify the intrinsic functions, their generames, function definitions, types of arguments and types of results are given in Appendix A.

A specific name of an intrinsic function that appears in an INTRINSIC statement (see section 7.3.5) may be passed as an actual argument to an external procedure with the exception of intrinsic functions for type conversion, lexical comparison, and maximum/minimum functions.

An intrinsic function can be called at any point in any program by means of a function reference (see section 7.2.1.1), in which the function name is given and the dummy arguments are replaced by actual arguments.

Actual arguments can be any expressions of the correct type and therefore may contain function references to other function subprograms (see Example 4 below).

Examples

```
1    SIN(ANGLEA)
2    A + SQRT(B)
3    A**2 + 2.0* COS(BETA + PI/2.0)
4    A + SIN(ALOG(A+B+C)**3 + SQRT(Z+SQRT(Y))
```

7.1.2.2 External functions

Any functions required in a particular program that are not intrinsic functions can be written as external functions for that program. External functions are independently written subprograms that are executed whenever a function reference (see section 7.2.1.1) to their name is encountered in any program unit.

An external function is identified as such by the first statement being a function declaration statement. This statement has the form:

```
type FUNCTION functionname(x1,x2,...,xn)
```

where

type is an optional parameter and is one of INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL or CHARACTER (optionally followed by a length specification, which can be any of the forms specified in section 2.3.3.2). In its absence the function will be given its type by the predefined convention (see section 2.3.1).

functionname is the name of the function being declared; the name by which it will be called elsewhere in the program by a function reference.

x1,x2,...,xn is a dummy argument list which may be empty, although the enclosing brackets must be specified. The items represent variable, array or dummy procedure names.

The FUNCTION statement is followed by the statements that make up the required process. The function declaration itself finishes with an END statement (see section 1.3.2.1).

A function can be invoked as a variable anywhere in the program by a function reference (see section 7.2.1.1) giving its name, functionname, with actual arguments (a1,a2,...,an).

Within the function subprogram, the function name can be used as a variable (of the specified type) and must be assigned a value at least once before a RETURN or END statement is executed.

A function must not reference itself: recursion, either direct or indirect, is not allowed.

Example

```
      LOGICAL FUNCTION TMEAN(A,D)
      REAL MEAN,A(10, 10)
      MEAN = 0
      DO 4 I = 1, 10
      DO 4 J = 1, 10
    4 MEAN = MEAN + A(I,J)
      MEAN = MEAN/100
      TMEAN = MEAN.GT.D
      END
```

7.1.2.3 Statement functions

If a mathematical function can be written in one statement then it may be written as a statement function. The statement function is declared by a statement function statement which takes the form:

```
sfname(x1,x2,...xn) = expression
```

where

sfname is the name given to the function.

x1,x2,...xn is a dummy argument list which may be empty, although the enclosing brackets must be specified.

The names used as dummy arguments in this list may be used elsewhere in the same subprogram as variables of the same type.

expression is an arithmetic, logical or character expression. It may contain references to external functions or previously defined statement functions. It may be a logical expression only if sfname is defined as type logical and a character expression only if sfname is defined as type character.

All statement function declarations must precede the first executable statement of the program unit and the statement function can be invoked only within that program unit.

A statement function is invoked by using sfname, with actual arguments replacing the dummy arguments in an expression. The actual arguments must correspond in number and type to the dummy arguments.

Example

FORTRAN program	Explanation
`VOL(R,H) = 3.14*R**2*H`	Declaration of statement function VOL(R,H)
`TOTAL = 0.`	First executable statement
`DO 11 I = 1,15`	
`READ(S,12)D,X`	Other executable statements
`12 FORMAT(2F10.3)`	
`.`	
`.`	
`11 TOTAL = TOTAL + VOL(0.5*D,X)`	Reference to statement function VOL with actual argument 0.5*D replacing dummy argument R and X replacing H

7.1.3 Subroutines

7.1.3.1 External subroutines

A subroutine subprogram fulfils a similar purpose to a function subprogram but returns any results in a different way.

A subroutine is declared by a SUBROUTINE statement, which takes the form:

```
SUBROUTINE subroutinename(x1,x2,...xn)
```

where

subroutinename is the name of the subroutine.

x1,x2,...xn is a dummy argument list. The list may be empty, in which case enclosing brackets may be omitted. The list items can represent variable, array or dummy procedure names or can take the form *. The character * represents a label in the calling program unit (see section 7.2.2.2).

The SUBROUTINE statement is followed by the statements that carry out the desired processes and the subroutine declaration is finished by an END statement (see section 1.3.2.1).

m another program unit by use of the CALL statement (see section 7.2.2.1). Control is returned to this position when the RETURN or END statement of the subroutine is encountered.

A subroutine must not contain a call to itself; recursion, either direct or indirect, is not allowed.

7.2 Transfer of control between program units

Every executable Fortran program contains at least one program unit, the main program. The first statement of the main program may be a program statement. This statement has the form:

```
PROGRAM programname
```

where programname is the name of the program and must be of the form specified in section 1.4.

Execution of the program begins at the first executable statement of the main program. Control stays in the main program until either a

function reference or a CALL statement is encountered, when control will be transferred to another program unit: a function from a function reference or a subroutine from a CALL statement. Control will then be passed between the main program and the other program units and between the program units themselves until either a STOP statement is executed or the END statement of the main program is reached.

The ways of entering and leaving function and subroutine subprograms are described in the sections below.

7.2.1 Functions

7.2.1.1 Transfer of control to a function subprogram

Control is passed to a function subprogram by a function reference. This function reference takes the form:

```
name(a1,a2,...,an)
```

where

name is the name of an external function, an intrinsic function or a statement function.

a1,a2,...,an is a list of actual arguments that replace the dummy arguments x1,x2,...,xn given in the function declaration. They must agree in order, number and type with the dummy arguments. If the referenced function has no parameters, the reference to it must still include a pair of brackets.

When a function reference to an intrinsic function or a statement function is encountered in a program unit, the function is evaluated using the actual arguments supplied by the function reference. This value is then substituted where the function reference occurs and execution of the program unit continues.

When a function reference to an external function is encountered in a program unit, control enters the function at the first executable statement of the function subprogram. Within the body of a function, the function name must be assigned a value at least once.

External function names must have an associated type. The type of the external function may be declared in the FUNCTION statement; otherwise its type is defined by the initial letter of its name, as described in section 2.3.1. In any program unit in which the external function is referenced, the type must be declared in a type specification statement unless the predefined type is correct. The type assumed by the referencing program unit must agree with that defined in the external

function. An external function name must not be assigned an initial value in a DATA statement.

7.2.1.2 Return of control from a function subprogram

In the case of a statement function or an intrinsic function, control is returned automatically to the position of the function reference in the calling program unit. In the case of external functions, control is returned from the external function to the calling statement when control reaches a RETURN or the END statement and the function value is returned to the calling program unit.

The RETURN statement takes the form:

```
RETURN
```

An extended form of this statement is available for use in a subroutine subprogram only (see section 7.2.2.2).

7.2.1.3 Example of an external function

The following is an example of an external function showing how the function is referenced in the calling program unit and how control is returned to the statement containing the function reference.

The function TMEAN has the value .TRUE. if the mean of the elements of a 10 x 10 array is greater than D.

```
      LOGICAL FUNCTION TMEAN(A,D)
      REAL MEAN,AN = 0
      DO 4 I = 1, 10
      DO 4 J = 1, 10
4     MEAN = MEAN + A(I,J)
      MEAN = MEAN/100
      TMEAN = MEAN.GT.D
      RETURN
      END
```

TMEAN could be referenced by any statement which may contain a logical expression. For example:

```
IF(TMEAN(ARR,50.0))GO TO 48
```

The effect of this statement is to test whether the mean of the 10 x 10 array ARR is greater than 50. If it is, control passes to the statement labelled 48.

7.2.2 Subroutines

7.2.2.1 Transfer of control to a subroutine subprogram

Subroutine subprograms are called from another program unit by a CALL statement.

The CALL statement has the form:

```
CALL subroutinename (a1,a2,...,an)
```

where

subroutinename is the name of the subroutine being called.

a1,a2,...,an is a list of actual arguments. These must agree in order, number and type with the dummy arguments given in the subroutine declaration for subroutinename. If the referenced subroutine has no arguments the CALL to it may omit the brackets.

To correspond with a dummy argument of the form*, an actual argument must take the form

```
*n
```

where n is the label of an executable statement in the calling program unit (see section 7.2.2.2).

When a CALL statement referring to a subroutine is executed, control is transferred to that subroutine, which is entered at its first executable statement.

Subroutine names do not have an associated type.

7.2.2.2 Return of control from a subroutine subprogram

Control is returned from a subroutine subprogram to the calling program unit when control reaches a RETURN or the END statement within the subroutine. There may be any number of RETURN statements in a subroutine but only one of these will be executed in any one execution of the subroutine.

The RETURN statement has two possible forms:

```
RETURN
RETURN e
```

where e is an integer expression.

The simple form, RETURN, has the effect of returning control to the statement following the CALL statement in the calling program unit.

The extended form, RETURN e, provides a means of returning to any labelled statement in the calling program unit. The expression e has a value, say n, and this value denotes that return is to be made to the nth statement label in the argument list. If e is less than one or greater than the number of statement labels in the argument list then control is returned to the statement following the CALL statement in the calling program unit.

For example, if a CALL statement of the form

```
CALL SUB(X,Y,*3,*10,2)
```

is made to a program unit whose first statement is

```
SUBROUTINE SUB(A,B,*,*,C)
```

and this subroutine contains the following RETURN statements:

```
RETURN 2
RETURN
RETURN 1
```

then RETURN 2 will return control to the statement labelled 10 in the calling program unit.

RETURN 1 will return control to the statement labelled 3

RETURN will return control to the statement following the CALL statement in the calling program unit.

7.2.2.3 Example of a subroutine subprogram

The following is an example of a subroutine subprogram showing how it is called and how control is returned to the calling program unit by means of a simple RETURN statement.

Subroutine ADD is required to add the elements of matrix I.

```
      SUBROUTINE ADD(I,J)
      DIMENSION I(10)
      J = 0
      DO  2 K = 1,10
    2 J = J + I(K)
      RETURN
      END
```

If this subroutine is to be entered, then the CALL statement must give the subroutine name ADD. For example, if it is required to add together the elements of an array IARR and to hold the result in N, the calling program unit would contain the following statements:

```
      .
      .
      .
      DIMENSION IARR(10)
      .
      .
      .
      CALL ADD(IARR,N)
```

7.3 Correspondence between dummy and actual arguments

A dummy argument (see section 7.1) must be one of the following:

1 A dummy variable name

2 A dummy array name

3 A dummy function or subroutine name

Actual arguments may be any of the following:

1 An expression except a character expression involving concatenation of a character element whose length specification is an asterisk in parentheses (unless it is a symbolic constant). Note that an expression may be a constant or a symbolic constant

2 An array name

3 An intrinsic function name

4 An external procedure name

5 A dummy procedure name

6 A statement label

Examples of actual arguments

FA	(where FA is the name of an intrinsic or external function)
A	(an array name)
A(2,3)	(an array element, see below)
Z	(a variable)
A(4,5)	(an array element)
SIN(Z)	(an expression comprising a function reference)
3.16	(a constant)
X+4*Y+3/Z	(an expression)

However, several rules apply for correct correspondence between dummy and actual arguments. These rules are as follows:

1 Actual and dummy arguments must correspond in number, order and type

2 If the dummy argument is a function, the actual argument that replaces it must be the name of an intrinsic or external function

3 If the dummy argument is a subroutine, the actual argument that replaces it must be the name of a subroutine subprogram (see section 7.4.2)

4 If the dummy argument is an array, the actual argument that replaces it must be an array or an array element

5 If the dummy argument is a variable, the actual argument that replaces it may be a constant, a variable, an array element or any other expression

The transfer of values between program units by means of dummy and

actual arrays is described in section 7.4.

7.3.1 Use of constants and expressions

If a dummy argument is assigned a value within the function or subroutine subprogram and this is used to return a result, then the corresponding actual argument may not be a constant or an expression.

7.3.2 Use of variables

Dummy variables must not occur in COMMON, DATA, EQUIVALENCE, PARAMETER or INTRINSIC statements; they may occur in type specification or DIMENSION statements (as bounds of dummy arrays).

7.3.3 Use of arrays and array elements

If a dummy argument is an array then the array must be declared in the subroutine or function subprogram in which the dummy array is used (see section 3.2.2). The size of each dimension may be given as an integer or as a variable of type integer. Dummy array names must not occur in COMMON, DATA, EQUIVALENCE, PARAMETER or INTRINSIC statements.

If the actual argument is an array name, it is made available to the called subprogram starting at its first element. That is, if the dummy array has n elements then the first n elements of the actual argument will be used as the n elements of the dummy array.

An array element used as an actual argument may replace a dummy argument that is either a variable or an array. If an array element replaces a variable, only that one element is made available to the called program unit. If an array element replaces an array, the specified actual element is used as the first element of the dummy array, and subsequent elements of the actual array form the remaining elements of the dummy array.

Thus if the actual argument is an array element, say A(x), and the dummy array is specified as having n elements then the n elements of array A from A(x) to A(x+n-1) are used as the n elements of the dummy array.

The actual argument specified must at least be large enough to cover the dummy array completely. That is, if the actual argument is an array, it must have at least as many elements as the dummy array. If the actual argument is an array element, that part of the actual array from and including the element given as the actual argument to the last element must contain at least as many elements as the dummy array.

Care must be taken when arrays having more than one dimension are used because of the order in which array elements are stored (see section 3.1.3).

An example of the use of arrays and array elements as actual arguments follows:

```
      FUNCTION FUN1(A,B)
      REAL A(500),B(10)
      DO 4 I=1, 500
      .
      .
      .
    4 CONTINUE
      DO 5 J=1, 10
      .
      .
      .
    5 CONTINUE
      FUN1= ...
      RETURN
      END
```

A reference to this external function could be included in an expression as follows:

```
      REAL LIST(540)
      .
      .
      .
      X=4*FUN1(LIST,LIST(531))
```

In this case, the first 500 elements of array LIST will be used as array A in the function and the elements LIST(531) to LIST(540) will be used as array B.

7.3.3.1 Adjustable arrays

A dummy array declared using one or more integer variables is an adjustable array. This method of declaration is only permissible for dummy arrays, but it allows the dummy array to have dimensions of different size each time the subprogram is executed, though the number of dimensions must remain constant. Each integer variable which specifies a dimension of an adjustable array must appear either in a common block or as a dummy argument in every dummy argument list which contains the array name.

For example, the subroutine ADD given in section 7.2.2.3 could be rewritten to add together the elements of an array of variable size.

```
      SUBROUTINE ADD(I,J,L)
      DIMENSION I(L)
      J=0
      DO 2 K=1,L
    2 J=J+I(K)
      RETURN
      END
```

When the subroutine is called, the CALL statement will specify the size of the actual array to be used. For example

```
CALL ADD(IARR,N,20)
```

would add the elements of a one dimensional array IARR with twenty elements.

7.3.4 Use of functions and subroutines as arguments

If a dummy argument is used as a subroutine or function name, the corresponding actual argument must be the name of a subroutine subprogram if the dummy argument appears in a CALL statement, or the name of an intrinsic or external function if the dummy argument appears in a function reference.

Any subroutine or external function name used as an actual argument in a CALL statement or function reference must be given in an EXTERNAL statement in the program unit in which the name is used as an actual argument.

The EXTERNAL statement has the form:

```
EXTERNAL name1,name2,...,namen
```

where each name is the name of a subroutine or an external function that is used as an actual argument in the program unit containing the EXTERNAL statement.

For example, if the subroutine

```
      SUBROUTINE GREEN(FUN,X,Y,Z)
      X=FUN(Y/Z)
      RETURN
      END
```

is to be called, giving an external function name as actual argument to

replace the dummy argument FUN, the calling program unit could contain the following statements:

```
EXTERNAL FUN1,FUN2
.
.
.
CALL GREEN(FUN2,A,B,C)
.
.
.
CALL GREEN(FUN1,A1,F1,EXP(C))
```

7.3.5 The INTRINSIC statement

An INTRINSIC statement is used to identify a name as representing an intrinsic function (see section 7.1.2.1). It also permits a name representing a specific intrinsic function to be used as an actual argument. The INTRINSIC statement has the form:

```
INTRINSIC name1,name2,...,namen
```

where each name is an intrinsic function name.

The use of a name in an INTRINSIC statement declares that name to be an intrinsic function name. If a specific name of an intrinsic function is used as an actual argument in a program unit, it must appear in an INTRINSIC statement in that program unit. The names of intrinsic functions for type conversion (INT, IFIX, IDINT, FLOAT, SNGL, REAL, DBLE, CMPLX, ICHAR AND CHAR), lexical relationship (LGE, LGT, LLE and LLT) and for choosing the largest or smallest value (MAX, MAXO, AMAX1, DMAX1, AMAXO, MAX1, MIN, MINO, AMIN1, DMIN1, AMINO and MIN1) must not be used as actual arguments.

The use of a generic name in an INTRINSIC statement does not cause that name to lose its generic property.

A name must not be used in more than one INTRINSIC statement in a program unit. Note that a name must not be used in both an EXTERNAL and an INTRINSIC statement in a program unit.

7.4 Transfer of values between program units

Values can be transferred between program units by use of:

1 Function values (see section 7.1.2.2)

2 Common block items

3 Dummy and actual arguments

If either of the latter two methods is used for returning results from external functions, care must be used if the actual arguments also appear elsewhere in the calling statement. For example in the statement

```
VAL = A**2/FUN(A,B)*B**2
```

the function FUN must not alter the values of its arguments, otherwise the results obtained will be unpredictable.

The second and third methods of transferring values are described in the sections below.

7.4.1 Common block items

A common block is an area of storage that may be referred to in any program unit which mentions the name of the block in a COMMON statement. This facility is discusssed fully in section 3.2.3. If in one program unit a value is assigned to an item which forms part of a common block and control is then transferred to another program unit which also refers to that common block, the value assigned in the first program unit becomes the value of the item occupying the same area of storage in the second program.

Items in named common blocks may be given initial values in block data subprograms by means of the DATA statement. This is described in section 3.3.2.

7.4.2 Dummy and actual arguments

When a transfer of control is made to a subroutine or a function subprogram, actual arguments are supplied in the CALL statement or the function reference, and the actual arguments are substituted for the dummy arguments in the function or subroutine on entry to the subprogram.

The actual arguments given in the CALL statement or the function reference must agree in number, order and type with the dummy arguments that they replace in the subroutine or function subprogram. The correspondence rules for dummy and actual arguments are described more fully in section 7.3.

The form of the dummy argument affects the way the corresponding actual

argument is referenced.

7.4.2.1 Argument reference

If the dummy argument has the form name, where name is the name of a variable or a subprogram, the corresponding actual argument will be referenced by value. This means that the dummy argument is assigned a storage location in the subprogram, to which the value of the actual argument is brought from the calling program unit at execution time. During execution all intermediate values are also stored in this location. On return to the calling program unit the final value is transferred from the dummy argument to the actual argument. Final values in the storage location corresponding to expressions or constants as actual arguments are lost.

If a dummy argument is an array name, the corresponding actual argument will be referenced by location.

7.5 Multiple entry into a subprogram

It is possible to enter a function or a subroutine subprogram at a statement other than the first executable statement. This is done by using a CALL statement or function reference that references an ENTRY statement within the subprogram. Control will enter the subprogram at the first executable statement following the ENTRY statement.

7.5.1 The ENTRY statement

The ENTRY statement has the form:

```
ENTRY name (x1,x2,...,xn)
```

where

name is the name of the entry point.

x1,x2,...xn is a list of dummy arguments corresponding to the actual arguments given in the CALL statement or function reference.

The names used as dummy arguments in the list may be used elsewhere in the same subprogram as variables of the same type.

The ENTRY statement is non-executable and does not affect control sequencing during the execution of the subprogram.

The appearance of an ENTRY statement does not affect the rule that

statement functions in a subprogram must precede the first executable statement of that subprogram.

7.5.2 Referencing an ENTRY statement

The ENTRY statement is referenced by a CALL statement if it is in a subroutine subprogram or by a function reference if it is in a function subprogram.

A subprogram must not reference itself directly or through any of its own entry points.

The actual arguments in the CALL statement or function reference must agree in order, number and type with the dummy arguments in the ENTRY statement being referenced. However, the dummy arguments of the ENTRY statement need not agree in order, type or number with the dummy arguments in the SUBROUTINE or FUNCTION statement or in any other ENTRY statement in the subprogram.

In a function subprogram the types of the function name and entry point name are determined by the FUNCTION and ENTRY statements. If an entry name in a function subprogram is of type character then each entry name and the name of the function subprogram must be of type character. If not of type character the types of the function name and entry point names can be different; whether they are or not, they are treated as variables equivalenced by means of an EQUIVALENCE statement (see section 3.2.4). After one of these variables is assigned a value in the subprogram, the others become undefined.

If information for an array is passed in the reference to an ENTRY statement, the array name and all its dimension parameters (except any that are in a common area or are constant) must appear in the dummy argument list of the ENTRY statement.

7.5.3 Entering the subprogram

Entry into a subprogram assigns new values to the dummy arguments of the referenced ENTRY statement. Thus, all appearances of these arguments in the whole subprogram are affected.

Reference to an ENTRY statement will not transmit new values for the arguments not listed in that ENTRY statement.

Entry cannot be made into an IF-block or the range of a DO statement.

7.5.4 Exit from the subprogram

On exit from the subprogram, the value returned to the calling program is the last value assigned in the subprogram to the entry point name before control is returned. A value may be returned via a name other than the one used to enter the subprogram. If this is done the two names must be of the same type, otherwise the value returned will be undefined.

7.6 SAVE statement

Values assigned to local items or items held in a common block normally retain any values held in them between execution of a RETURN or END statement and a subsequent re-entry to the subprogram, except in the case of common items which are assigned a new value elsewhere. However, the FORTRAN 77 standard does not require all implementations to retain values automatically in such circumstances.

The SAVE statement is used to specify which items are to retain their values after the execution of a RETURN or END statement and should be used in any program which is required to be run using other FORTRAN 77 implementations. The SAVE statement has the form:

SAVE name1, name2,...,namen

where each name is a named common block name preceded and followed by an oblique, a variable name, or an array name. A name may not occur more than once in a SAVE statement within a particular program unit.

Dummy argument names, procedure names and names of items in a common block must not appear in a SAVE statement.

A SAVE statement without a list is treated as though it contained the names of all allowable items in that program unit.

The appearance of a common block name preceded and followed by an oblique in a SAVE statement has the effect of specifying all of the items in that common block.

If a particular common block name is specified by a SAVE statement in a subprogram it must also be specified by a SAVE statement in every other subprogram in which that common block appears. Note however, that if a named common block is used in the main program unit it is effectively saved for all subprograms referenced by the main program.

The execution of a RETURN or END statement within a subprogram may not retain the values of items other than the following in some implementations of FORTRAN 77.

1 Items specified by SAVE statements;

2 Items in blank common;

3 Items defined with an initial value and not assigned a new value;

4 Items in a named common block that appears in the subprogram and in at least one other program unit which is referencing, either directly or indirectly, that subprogram.

CHAPTER EIGHT

FORMAT SPECIFICATION

In FORTRAN 77 all input and output data are handled in the form of records.

Records can be output to:

1 The display

2 A printer

3 Files on magnetic media

4 Internal files consisting of character variables or arrays

Records can be input from:

1 The keyboard

2 Files on magnetic media

3 Internal files

Two kinds of records are recognized: formatted records and unformatted records.

A formatted record is a string of characters which might, for instance, be input on one punched card or output as a line of print. Each record can be regarded as being split into fields, where each field contains one or more characters and normally represents the value of one variable or array element.

An unformatted record is essentially in internal machine form and is normally used for re-input to the computer rather than for examination by the programmer. Unformatted records are most appropriately used on magnetic media.

If formatted records are being used the input or output of records may be controlled by a format specification. The format specification can be given either in a FORMAT statement (see section 8.2.1) or as values of character arrays, character variables, or other character expressions.

The format specification defines the form of one or more external or internal records and specifies the conversion of values between the

internal form and the fields of the record. Formatted input records are read in the format given in the format specification and are converted from character codes on the input medium to internal machine form. Formatted output data is converted from internal machine form to records in the character code used by the output medium and are output in the format given in the format specification.

The individual fields within a record and the conversions to be applied to them are specified within a format specification by means of edit descriptors, which take the forms described in section 8.3. Thus, for example, the edit descriptor I4 describes an integer field four character positions wide. The way in which edit descriptors can be combined to form complete format specifications are described below, and the use of edit descriptors in FORMAT statements and arrays is discussed in sections 8.2.1 and 8.2.2.

Input normally involves assigning the value represented in each field to a variable or array element in store, output normally involves placing the value of a variable or array element in store into the appropriate field. Some descriptors have other effects such as allowing for spacing between fields. The edit descriptors of a format specification are associated with items in an input or output list in a READ or WRITE statement as described in section 8.2.3. The first character in a record to be output to a line printer is taken as a print control character (see section 8.3.1.8).

8.1 Format specifications

A format specification consists of a series of edit descriptors (see section 8.3) surrounded by parentheses. The specification describes one or more records which are to be input or output. Apart from the edit descriptors, the following may appear in the specification:

commas
parentheses
repeat counts and group repeat counts

8.1.1 Field separators

Consecutive edit descriptors and groups of edit descriptors enclosed in parentheses (see below) must be separated by a comma, except

1 Between an edit descriptor containing a P and an immediately following edit containing an F,E,D or G descriptor

2 Before or after a slash

3 Before or after a colon

Other slashes may appear before or after a series of edit descriptors. A comma marks the end of a field.

8.1.1.1 Slash editing

A slash marks the end of a field and the end of a record. Thus, when one slash is present, the edit descriptor following initiates a transfer to the next new record, and if a series of n slashes is present, n - 1 records will be omitted on input or n - 1 blank records will be created on output, and the next edit descriptor will initiate a transfer to the next new record.

For example, the format specification

(F1,F2/F3,///F4)

where each F is an edit descriptor, describes a series of records in the following order:

1 A record containing two fields corresponding to F1 and F2

2 A record containing one field corresponding to F3

3 Two blank records (or two records to be omitted on input)

4 A record containing one field corresponding to F4

The parentheses at the beginning and end of the format specification may be considered to initiate a new record and terminate a record respectively. For example, the specification

(///F///)

where F is an edit descriptor, describes a series of records in the following order:

1 Three blank records

2 A record containing one field corresponding to F

3 Three blank records

8.1.2 Repetition of descriptors

An edit descriptor or a group of edit descriptors (that is a series of descriptors enclosed in parentheses) may be repeated by preceding it with an integer r. The effect will be as if the descriptor or group was repeated r times. Non-repeatable edit descriptors (see section 8.3) may be repeated only if they are enclosed in parentheses.

The repeat count or group repeat count r must be a positive (and non-zero) integer. If a group enclosed in parentheses is not preceded by a group repeat count, a count of one is assumed.

Any group of descriptors enclosed in parentheses may have among its items other groups of descriptors enclosed in parentheses but these must not be nested more than seven deep.

For example, the specification

```
(F1,3F2/5F3,6(/F4,F5,),2F6)
```

where each F is an edit descriptor, describes a series of records in the following order:

1 A record containing one field described by F1 and three described by F2

2 A record containing five fields described by F3

3 Five records each containing one field described by F4 and one by F5

4 A record containing one field described by F4, one by F5 and two by F6

8.2 Format specification methods

Format specifications as described in the earlier part of this chapter may be either specified in FORMAT statements or as values of character arrays, character variables, or other character expressions. When a format specification is to be used to control the input or output of data, either a reference is made to the label of the FORMAT specification or the name of the character array or character variable, or a character expression is used. Information on the use of format statements or character format specifications for input and output is given in Chapter 9.

8.2.1 The FORMAT statement

A FORMAT statement is a non-executable statement and has the form:

label FORMAT specification

where

label is a statement label; every FORMAT statement must be labelled.

specification is a format specification as described earlier in the chapter.

The following are examples of FORMAT statements, where the Fi are edit descriptors as described in section 8.3. Complete examples of FORMAT statements including the use of edit descriptors are given in section 8.4.

```
16   FORMAT (F1)

3049 FORMAT (F2,3F3/F4)

4    FORMAT (F5,F6,6(/F7,3F8)/F9)
```

8.2.2 Character format specification

A format specification may be held in a character array, or within a character variable, or may be specified as a character expression. The specification must include the parentheses at the beginning and end. It may have been read into a character array or character variable by means of an A conversion code (see section 8.3.1.7) or may have been set by an initialization or assignment statement. Character data may follow the right parenthesis that ends the format specification and will be ignored.

For example, the format specifications given in the examples in the previous section could be held in arrays instead of being given in FORMAT statements. If the specification given in the statement labelled 3049 were held in a character*4 array named ARR, the first element of the array would hold the left parenthesis followed by some or all of the characters that make up the edit descriptor F2. The succeeding character positions would hold the comma, the figure three, the characters making up the field descriptor F3, the slash, the figure three, the characters making up the edit descriptor F4 and, finally, the right parenthesis, using as much of the remainder of the array ARR as is required.

A character format specification can be used for input or output in the

same way as one given in a FORMAT statement. For example, a reference to the array ARR mentioned in the previous paragraph would have the same effect as a reference to the statement labelled 3049 in the previous section.

Formats may be varied at run-time either by assigning new values to, or using A format codes (see section 8.3.1.7) to read values into character array elements or character variables.

8.2.3 Effect of FORMAT statements and character format specifications

A READ or WRITE statement (see sections 9.3.1 and 9.4.1) referencing a FORMAT statement or character format specification normally contains a list of variable names and array elements known as an input or output list. These are associated in order with the descriptors in the format specification, except that non-repeatable edit descriptors are not associated with variables or array elements. Thus if a list of names in a READ statement was

```
Y,Z
```

and the FORMAT specification was

```
(F1,W,F2)
```

where F1 and F2 are repeatable edit descriptors and W is a non-repeatable edit descriptor, then the variable Y will be associated with edit descriptor F1 and Z with edit descriptor F2.

Each action performed during the execution of a formatted READ or WRITE statement is determined by the next descriptor in the format specification and the next item, if any, in the input or output list. If the descriptor is a non- repeatable edit descriptor, then it is acted upon and the next descriptor examined; the process is repeated until a repeatable edit descriptor is encountered.

The descriptor must be one which is permitted with a variable or array element of the type under examination (see section 8.3 below). On input the value represented in the field is converted according to the edit descriptor and is assigned to the variable or array element; on output the value of the variable or array element is output to the field in the format specified. The next item from the list in the READ or WRITE statement and the next edit descriptor are then selected, and the process repeated.

When the last named variable or array element has been operated upon, the next descriptor is examined. If it is a non-repeatable edit descriptor it will be acted upon and the next descriptor will be

examined in the same way. This process is repeated until a repeatable edit descriptor is encountered.

When the last edit descriptor has been acted upon or when an edit descriptor not of the types given above is encountered, execution of the statement ceases.

A special case of a formatted READ or WRITE statement is one that does not contain a list of variable and array element names; the first or only descriptor in the corresponding format specification must be a non-repeatable edit descriptor otherwise the corresponding record is skipped. If the READ or WRITE statement does contain a list of names, the corresponding format specification must contain at least one repeatable edit descriptor.

If, when the format specification has been completely scanned, there are still items left in the list of names, a new record will be started and the format specification will be re-scanned as follows:

1 If there are no internal parentheses, scanning will be repeated from the beginning of the specification

2 If there are internal parentheses, scanning will be repeated from the left parenthesis corresponding to the right-most internal right parenthesis. If this left parenthesis is preceded by a group repeat count, the repeat count is taken into account

When rescanning is completed by the closing parenthesis of the format specification being reached again, the rescanning as described above is repeated if further items still remain.

The arrows in the following examples show where scanning would be restarted:

```
FORMAT(.....)
       ^
FORMAT(...,(...))
           ^
FORMAT(...,(...(...)...)...)
           ^
FORMAT(...(...)...3(.(.).(.))..)
                   ^
```

Examples of complete FORMAT statements and their effects are given in section 8.4.

8.3 Edit descriptors

The edit descriptors are used to specify the external format of fields in a record, and are classified into two types:

1 Repeatable edit descriptors

2 Non-repeatable edit descriptors

Repeatable edit descriptors may be preceded by a repeat count which is an unsigned, non-zero, integer constant and which specifies the number of times the edit descriptor is to be repeated. If the repeat count is omitted a value of one is assumed. A repeatable edit descriptor is one containing one of the following format codes:

A,D,E,F,G,I,L

A repeatable edit descriptor indicates the manner in which a variable or array element is to be edited.

Non-repeatable edit descriptors, slashes and colons must not be preceded by a repeat specification, and they operate independently of any items in the input or output list. A non-repeatable edit descriptor is one containing one of the following format codes:

H,literal,T,TL,TR,X,S,SP,SS,P,BN,BZ

A non-repeatable edit descriptor indicates the manner in which a field is to be edited. The use of the colon to terminate format control if there are no more items in the input or output list is described in section 8.3.1.13.

The different edit descriptors, the types of internal variables with which they correspond, and their actions, are listed below. A reference is given to the section in which each descriptor is discussed.

Edit descriptor	Internal data type	Action	Section
Iw	Integer	Numeric conversion	8.3.1.1
Iw.m			
Fw.d	Real or part complex	Numeric conversion	8.3.1.2
Dw.d			8.3.1.3
Ew.d			8.3.1.3
Ew.dEe			8.3.1.3
Gw.d			8.3.1.4
Gw.dEe			8.3.1.4
kP		Scaling real numbers	8.3.1.5
Lw	Logical	Logical value conversion	8.3.1.6
A	Character	Character conversion	8.3.1.7
Aw			8.3.1.7
nHliteral		Write text	8.3.1.8
'literal'			
nX		Alter position in record	8.3.1.9
Tc		where transfer of data	8.3.1.10
TLc		should begin	8.3.1.10
TRc			8.3.1.10
S		Control of optional plus	8.3.1.11
SP		characters in numeric	8.3.1.11
SS		output fields	8.3.1.11
BN		Control of the	8.3.1.12
BZ		interpretation of blanks in numeric input fields	8.3.1.12

In the above list of edit descriptors, the capital letters I, F, E, D, etc., are called format codes and it is these format codes that indicate in what way the record is to be converted either from external format to internal machine form on input, or from internal machine form to external format on output. w is the total width of the field in characters in the external format. d is the number of character positions in the fractional part of a number. m is the number of significant digits in the field. e is the number of digits in the exponent. k is an optionally signed integer constant. w, e, n and c are non-zero, unsigned, integer constants.

Since complex values can be considered as two real values for the purposes of input and output, they are transferred by means of two D,

E, F, or G format codes.

8.3.1 Format codes

The following sections describe the various format codes and their effects on data on input and output. The term conversion code is used interchangeably with format code for those codes which are directly concerned with converting data between its external format and its internal machine form.

8.3.1.1 The I conversion code

The I conversion code is used to transfer integer data. The edit descriptor has one of the following forms:

Iw

Iw.m

where

w is an unsigned positive integer that gives the external width of the field in characters.

m is an unsigned integer that gives the minimum number of digits to be output.

Input

The Iw.m edit descriptor is treated identically to the Iw edit descriptor. When this conversion code is used for input, the edit descriptor causes the next w characters in the current record to be read as an integer and the converted value to be assigned to the relevant item in store.

The characters in the external field may be a signed or an unsigned integer; unsigned numbers will be assumed to be positive. Spaces before the first digit are ignored but must be included in the character count w, as must the sign, if any. All other spaces are treated as zeros or are ignored as determined by a combination of any BLANK= specifier that is currently in effect for the unit (see Chapter 9), and any BN or BZ edit descriptors (see section 8.3.1.12). Unless specified otherwise spaces, other than leading spaces, are treated as zeros. A field of all spaces is treated as a field of zeros. The field must not contain a decimal point or an exponent.

The following table gives some examples of the effects of the I code on input. The symbol # represents a space.

Edit descriptor	External number	Internal number
I5	#+376	+376
I6.2	####-2	-2
I6	####-2	-2
I4	34##	+3400
I3	###	0

Note: If a number that has been read in is too large or too small for the associated variable, the maximum or minimum value respectively for that type of variable is substituted, namely 2,147,483,647 or -2,147,483,648 for integer.

Output

When this conversion is used for output, the edit descriptor will cause the value of the relevant item in store to be output as an integer occupying w character positions in the current record; the number will be right justified. The effect of using the Iw.m edit descriptor is the same as Iw except that the unsigned integer constant consists of at least m digits and if necessary, has leading zeros.

The value of m must not exceed the value of w. If m is zero and the value of the output item is zero then the output field consists only of blank characters regardless of the sign control that is currently in effect. Negative numbers will be preceded by a minus sign which occupies one of the w character positions specified; positive numbers will be unsigned. The field of w characters will be space filled on the left if necessary. If the integer to be output, including any minus sign, exceeds w characters, the output field is filled with asterisks.

Some examples follow of the effect of the I code on output:

Edit descriptor	Internal number	External number
I5	+3659	#3659
I6	-987	##-987
I3	+3659	***
I6.4	-123	#-0123
I5.3	24	##024

Example

The following:

```
      I = 20
      J = -236
      K = 9872
      WRITE(6,100)I,J,K
  100 FORMAT('0',I4.3,2I8)
      .
      .
      .
```

causes the following line to be printed:

```
#020####-236####9872
```

8.3.1.2 The F conversion code

The F conversion code is used to transfer basic real numbers, that is those numbers written without an exponent, namely real or one part of a complex value. The edit descriptor has the form:

Fw.d

where

w is an integer giving the width in characters of the external field.

d is the number of digits in the fractional part of the number. w must always be greater than or equal to d.

Input

When this conversion code is used for input, the edit descriptor causes the next w characters in the current record to be read as a real number and the converted value to be assigned to the relevant item in store. If the item is of type complex then two edit descriptors are required.

The external field must contain w characters including:

1 A sign (optional)

2 A string of digits which may contain a decimal point

The external field may also contain an exponent.

Unsigned numbers are assumed to be positive. Spaces occurring before the first digit are ignored. All other spaces are treated as for I editing. All spaces must be included in the character count. A field of all spaces is treated as zero.

If the field does not contain a decimal point, the number is treated as though a point occurred before the last d digits of the string. This is the number that any scale factor can be considered to operate on (see section 8.3.1.5). If the external field contains a decimal point, this will override the decimal point implied by the value d in the descriptor.

Some examples follow of the effects of the F code on input. The symbol # represents a space.

Edit descriptor	External number	Internal number
F6.2	##1234	12.34
F6.2	1.2300	1.23
F6.2	#-2345	-23.45
F6.2	#123E1	12.30

Output

When this conversion code is used for output, the edit descriptor causes the value of the relevant item in store to be output as a decimal fraction, rounded to d decimal places and made up by trailing zeros if necessary. The number is right justified and if it is negative it is preceded by a minus sign. The field will be space filled on the

left to make up the w characters. If the number has no integral part and if the field width specified is large enough, the decimal point will be preceded by a zero. If the item is of type complex, two descriptors are required to output it.

The number of characters to be output should not exceed the field width. If it does, the output field is filled with asterisks.

The decimal point and minus sign must be included in the character count w.

Some examples follow of the effects of the F code on output. The symbol # represents a space.

Edit descriptor	Internal number	External number
F10.4	+5227.3278	#5227.3278
F10.4	-345.6789	#-345.6789
F10.4	+12.3	###12.3000
F10.4	-3.21989623	###-3.2199

Example

The following statements:

```
      X=-3.7690
      Y=1.55
      Z=12345.69
      WRITE(6,100)X,Y,Z
  100 FORMAT(1H#,F10.4,F8.6,F8.3)
      .
      .
      .
```

would produce the following line:

```
###-3.76901.550000********
```

The value for Z requires five positions before the decimal point but since only four are available the value is represented by ********, that is asterisks in all eight positions of the field.

8.3.1.3 The E and D conversion codes

The E and D conversion codes are used to transfer real numbers.

The edit descriptors have the following forms:

```
Ew.d
Dw.d
Ew.dEe
```

where

w is an integer giving the width of the external field in characters.

d is an integer giving the number of digits in the fractional part of the number. w must always be greater than or equal to d.

e is an integer giving the number of digits in the exponent: e must be greater than zero. This has no effect on input.

Input

When the E or D conversion code is used for input, the edit descriptor causes the next w characters in the current record to be read as a real number and the converted value to be assigned to the relevant item in store. If the item is of type complex then two edit descriptors are required.

The external field must contain w characters including:

1 A sign (optional)

2 A string of digits which may contain a decimal point

3 An exponent (optional)

The exponent may have one of the following forms;

1 A signed integer constant

2 E or D followed by a signed integer constant

3 E or D followed by an unsigned integer constant

Unsigned numbers and exponents are assumed to be positive. Spaces occurring before the first digit are ignored. All other spaces are treated as for I editing. All spaces must be included in the character count. A field of all spaces is treated as zero.

If the field does not contain a decimal point, the number is treated as though a point occurred before the last d digits of the string. This is the number that any exponent or scale factor can be considered to operate on. If the external field contains a decimal point, this will override the decimal point implied by the value d in the descriptor.

Note: If the modulus of the external real number to be input is greater than the maximum real (the largest possible real value that can be assigned to the variable) then the maximum real is assigned to the variable together with the appropriate sign.

Some examples follow of the effects of the E code on input. The symbol # represents a space.

Edit descriptor	External number	Internal number
E7.3	7654321	+7654.321
E7.3	#+137-3	+.000137
E7.3E2	#+137-3	+.000137
E7.3	1.234E2	+123.4
E7.3	-123E02	-12.3

Output

When the E or D conversion code is used for output, the edit descriptor causes the value of the relevant item in store to be output as a decimal fraction with an exponent. If the exponent field e exists then the exponent will be output with e digits.

The fractional part, f, of the external number will be in the range

$$0.1 \leq f < 1$$

and will be rounded down to d digits and will be output preceded by a minus sign (if the number is negative), a zero (if the field width specified is wide enough) and a decimal point. If the item is of type complex then two edit descriptors are required.

The exponent will have one of the forms:

1 E+d1d2 or E-d1d2 if Ew.d or Dw.d is used and |exp|<=99

2	+d1d2d3	or -d1d2d3 if Ew.d or Dw.d is used and 99<|exp|<=999
3	E+d1d2 ... dn	or E-d1d2 ... dn if Ew.dEe is used

where d1 d2...dn are digits. The number will be right justified.

The fractional part f will be signed only if it is negative. The exponent part will always be signed. The scale factor can be used to alter the range of the fractional part f of the external number from the limits defined above.

If necessary, the field will be space filled on the left to w characters. The number of characters, including the minus sign if any, should not exceed the field width. If it does, the output field is filled with asterisks.

Some examples follow of the effects of the E code on output. The symbol # represents a space.

Edit descriptor	Internal number	External number
E14.5	+12345678	###0.12346E+08
E14.5	-1.23	##-0.12300E+01
E14.5	+.000123	###0.12300E-03
E14.5	-.003	##-0.30000E-02
E14.5E4	-.003	-0.30000E-0002

Example

The following:

```
      A=4764.732
      B=-21.5E-4
      C = .003210
      D = -99.9E3
      WRITE(6,100)A,B,C,D
  100 FORMAT('0',E15.8E3,E13.6,E12.4,E9.4)
      .
      .
      .
```

causes this line to be printed out:

```
0.47647320E+004-0.215000E-02##0.3210E-02*********
```

The value for D requires at least ten positions (-.9990E+05) and as only nine are specified, the field is set to *********

The following:

```
      DOUBLE PRECISION X,Y,Z
      X=-3.66D2
      Y=123456.12345
      Z=155.151
      WRITE(6,100)X,Y,Z
100   FORMAT('0',D10.3,D16.8,D18.7)
      .
      .
      .
```

causes this line to be printed out:

```
-0.366E+03##0.12345612E+06#####0.1551510E+03
```

The edit descriptor for Y specifies only eight significant figures; in this case rounding occurs.

8.3.1.4 The G conversion code

The G conversion code is a code that can be used to transfer real values.

Edit descriptors using the G conversion code have the format:

```
Gw.d
Gw.dEe
```

where

w is an integer giving the width, in characters, of the external field.

d is an integer giving the number of digits in the fractional part of the number: w must always be greater than or equal to d.

e is an integer giving the number of digits in the exponent. This has no effect on input.

Input

For input the G conversion code has the same effect as if it were Ew.d,

Dw.d or Fw.d (see sections 8.3.1.3 and 8.3.1.2).

Output

When this conversion code is used for output the edit descriptor causes the value of the relevant item in store to be output either in fixed point form (without an exponent) or in floating point form (with an exponent).

The magnitude of the value determines the form in which it is output as follows:

1 If the number, for example x, is outside the range 0.1<=x<10**d, then the number is output with an exponent in the same manner as the E edit descriptor (see section 8.3.1.3)

2 If the number is inside the above range, then the d most significant digits of the number are output as a decimal fraction without a decimal exponent and will be justified towards the left by a fixed number of spaces

If the Gw.dEe conversion code is used then e+2 spaces will be produced at the right of the field; four spaces will be produced if the Gw.d conversion code is used. The field width w must allow for these additional characters.

If a scale factor (see section 8.3.1.5) is operating, it will have no effect unless the value being output is outside the range 0.1<=x<10**d. If the value is outside this range, then the effect of the scale factor will be as for the E conversion code (see section 8.3.1.3).

Some examples follow of the effects of the G code on output. The symbol # represents a space.

Edit descriptor	Internal number	External number
G11.4	+10.3456	##10.35####
G11.4E4	+10.3456	10.35######
G11.4	-0.000367	-0.3670E-03
G11.4E1	-0.000367	#-0.3670E-3
G11.4	+4958.67	##4958.####

G11.4	+49586.7	#0.4959E+05
2PG11.4	+10.3456	##10.35####
2PG11.4	-.00036	#-36.00E-05

Example

The following:

```
      REAL A,R,S,T
      COMPLEX C
      READ(5,4) A,C,R,S,T
4     FORMAT(2G8.3,G6.2,G11.8E2,G4.0,G15.12)
      WRITE(6,41) A,C,R,S,T
41    FORMAT(1H ,G10.3,G11.4,2G14.7E3,G15.8,G7.1)
```

and a data card of this form:

```
#33854##2000.E-4-12775##-96612E-8768#+###105########
```

would produce a printed output line as follows:

```
#0.339E+04#0.2000####-127.7500#####-.9661200E-011##7680.
                                              0000####0.1E-01
```

8.3.1.5 The scale factor

The scale factor is used to change the position of the decimal point in real numbers. It has the form:

kP

where k is an integer, optionally preceded by a minus sign.

A scale factor of zero is assumed in any format specification until a scale factor is specified. Once a scale factor is specified it operates on all real or complex values converted in that FORMAT statement by F, E, D or G edit descriptors (see sections 8.3.1.2, 8.3.1.3 and 8.3.1.4) until a new scale factor is encountered. A scale factor of the form

0P

cancels the operation of any previous scale factor.

Effects on input

A scale factor affects only real numbers without an exponent. The scale factor is ignored for any other type of number.

The effect of the scale factor on a real number input is that the number will be divided by the kth power of 10, as it is converted from an external value to the internal value.

That is:

1 If the input data is in the form ab.cde and it is required to use this data internally in the form .abcde, the edit descriptor necessary would be 2PF6.3

2 If the input data is in the form ab.cde and it is required to use this data internally in the form abcd.e, the edit descriptor would be -2PF6.3

Effects on output

The scale factor can be used to modify the effect of edit descriptors containing F, D, E or G conversion codes. It has no effect on descriptors other than these.

Its effects are as follows:

1 F CONVERSION CODE The internal number is multiplied by the kth power of 10 as it is output

2 E or D CONVERSION CODES The internal number is multiplied by the kth power of 10 as it is output but the exponent is adjusted to compensate. Therefore, the number is changed in form but not in value. Note that in this instance the scale factor k must be restricted to the range -d<k<d+2, where d is an integer giving the number of digits in the fractional part of the number

3 G CONVERSION CODE If the number is output without an exponent, the scale factor has no effect. Otherwise, the effect is the same as for the E or D descriptors

Examples

The following table shows the effect of a scale factor on field descriptors used for input:

Edit descriptor	External number	Internal value
-3PF6.3	99.99	99990.0
3PF6.3	99.99	.09999
2PF12.2	4120.0	41.2
0PF5.2	21.2	21.2
2PE7.1	8642.0	86.42
2PE7.1	86.42E2	8642.0

The next table shows the effect of a scale factor in format codes used for output:

Edit descriptior	Internal value	External number
2PF11.0	12345.0	###1234500.
-3PE11.5	12345.0	0.00012E+08
2PE11.3	12345.0	##12.34E+03
4PG11.3	12345.0	##1234.E+01
-1PG11.3	12345.0	##0.012E+06

8.3.1.6 The L conversion code

The L conversion code is used to transfer logical values. The edit descriptor has the form

Lw

where

w is an integer giving the width in characters of the external field.

Input

When this conversion code is used for input, the edit descriptor will cause a field of w characters to be read and converted to the internal representation of .TRUE. or .FALSE.; the converted value is assigned to the corresponding item in the input list. The external field consists of w characters as follows:

1 Optional spaces, optionally followed by a decimal point, followed by T (representing the value .TRUE), optionally followed by any other characters

2 Optional spaces, optionally followed by a decimal point, followed by F (representing the value .FALSE.), optionally followed by any other characters

Output

When this conversion code is used for output it will cause w-1 spaces to be output followed by the character T if the relevant item has the value .TRUE. or by the character F if the item has the value .FALSE.

Example

The following:

```
      LOGICAL X,Y
      X = .TRUE.
      Y = .FALSE.
      N = 250
      A = 27.4
      WRITE(6,4)N,X,A,Y
    4 FORMAT('0',I5,L6,F6.2,L3)
      .
      .
      .
```

will cause the following line to be output:

```
##250#####T#27.40##F
```

8.3.1.7 The A conversion code

The A conversion code is used to transfer character fields. The edit descriptors have the form:

Aw

A

where w gives the width, in characters, of the external field. If no width is specified then the number of characters in the external field is the length of the item in the input/output list.

The number of characters transferred depends partly on the length of the corresponding variable in the input/output list.

Input

If the field width w is greater than or equal to the length of the item (len), then the rightmost len characters will be taken from the input field. If the field width is less than len then w characters will be left justified in the field with len-w padding spaces to the right.

Example

The following:

```
      CHARACTER*4 A,B,C,X,Y,Z
      READ(5,3)A,B,C,X,Y,Z
    3 FORMAT(A4,A,A,A5,A2,A5)
```

and this data card:

```
HOT*AND+COLDHOT*AND+COLD
```

would cause the character strings to be assigned to the variables, A,B,C,X,Y,Z as follows:

Variable	String
A	HOT*
B	AND+
C	COLD
X	OT*A
Y	ND##
Z	COLD

Output

When this conversion code is used for output, the edit descriptor will cause w characters to be output. If w is less than or equal to len, then the w leftmost characters are output and the rest are ignored. If w is greater than len then w-len blank characters will be output

followed by the characters of the list item.

8.3.1.8 The H format code and character data

Edit descriptors using the H format code are used to transfer character strings between the format specification and the current record. They do not involve program variables. They have the form:

nHstring

'string'

where

n is the width of the character string.

string is the character string to be transferred.

If character data within apostrophes contains an apostrophe, that apostrophe must be represented by two apostrophes, for example, 'DON"T' and 5HDON'T are equivalent.

Both forms can be used in format specifications.

For example, the following two formats are equivalent:

```
100 FORMAT(' ANNUAL TOTAL')
100 FORMAT(13H ANNUAL TOTAL)
```

Input

Edit descriptors using apostrophes or the H format code may not be used on input.

Output

When used for output, this edit descriptor will cause the n characters of string to be output as part of the current record. For example, either of the examples quoted above would cause

ANNUAL TOTAL

to be written to the output stream.

A formatted record that is to be printed must begin with a print control character of the form

1Hx or 'x'

where x can be one of the characters listed below with the effects each has on the printing of the record.

Print control character	Effect
Space	Feed one line before printing
0	Feed two lines before printing
1	Feed to head of a new page
+	No line feed (this facility is not available on all line printers)
Any other character	As for space

The print control character is never printed, although it is held on the record. Thus an output line of 132 characters actually occupies 133 character positions on the record.

8.3.1.9 The X format code

The X format code is used to omit characters in the input or output record. The edit descriptor has the form:

nX

where n is an integer giving the number of characters to be skipped. The X code is not concerned with transfer of data to or from a variable or array element in store.

Input

When this format code is used for input, the edit descriptor will cause n characters on the external record to be omitted.

Output

When this format code is used for output the edit descriptor will cause the next character that is to be transmitted to the record, to be

written at a position n characters forward from the current position. Any unfilled positions will be filled with spaces.

8.3.1.10 The T format codes

The T format codes are used to specify the position in the record where the transfer of data is to begin. Their use may result in the overwriting of data already in the record.

The edit descriptor takes one of the forms:

Tc

TLc

TRc

where c specifies the character position at which the transfer should begin.

The Tc edit descriptor indicates that the next character is to be transferred to or from the cth character position within the record, counting the first character position in the record as one.

The TLc edit descriptor indicates that the next character to or from the current record is to be c character positions backward from the current position. If the current position is less than or equal to position c, then the transmission of the next character to or from the record would occur at position one.

The TRc edit descriptor indicates that the next character to or from the current record is to be c character positions forward from the current position.

Note that on output, T format codes do not in themselves cause any characters to be transferred and therefore do not affect the length of the record. However if characters are subsequently written beyond any unfilled positions, then those positions will be filled with spaces.

Example 1

The format specification:

```
100 FORMAT (T16,'OF',T1'#THE',T19,'FILEX',T6,9HBEGINNING)
```

causes the following line to be written to the output stream:

```
THE BEGINNING OF FILEX
```

Example 2

The format specification:

```
200 FORMAT(T19,'ING',TL8,'#EDIT',T6,'SM',TL8,'#TRAN',TR2,'ISSION')
```

causes the following line to be written to the output stream

```
TRANSMISSION EDITING
```

8.3.1.11 The S format codes

The S format codes control the outputting of plus characters in numeric output fields.

The edit descriptors take one of the forms

```
S
SP
SS
```

If an SP edit descriptor occurs in a format specification then a plus sign will be produced in any subsequent position which normally contains an optional plus.

If an SS edit descriptor occurs in a format specification then a plus sign will not be produced in any subsequent position which normally contains an optional plus.

If an S edit descriptor occurs then the option is restored to the compiler default, which in 2900 FORTRAN 77 is the same as for the SS edit descriptor.

8.3.1.12 The B format codes

The B format codes control the interpretation of spaces other than leading spaces in numeric input fields.

The edit descriptors take one of the following forms

```
BN
BZ
```

If a BN edit descriptor occurs in a format specification all spaces in succeeding numeric input fields are ignored. The effect of this is to

treat the input field as if the spaces had been removed and the field right justified. A field of all spaces has the value zero.

The following example

```
      READ(5,10) I,J,K
10 FORMAT(BN,2I6)
```

with data ##2#34#1#2#3######

would cause the following values to be assigned:

I to be assigned the value 234

J to be assigned the value 123

K to be assigned the value 0

If a BZ edit descriptor occurs in a format specification all such space characters in succeeding numeric input fields are treated as zero. In the example above with BZ specified:

I would be assigned 2034

J would be assigned 10203

K would be assigned 0

BN and BZ edit descriptors affect only I, F, E, D and G editing during execution of an input statement. They have no effect during execution of an output statement.

8.3.1.13 Colon editing

When a colon is encountered in a format specification and there are no more items in the input/output list then format control is terminated. If there are more items in the input/output list then the colon has no effect.

8.4 Examples of format specification

1 A READ statement (see Chapter 9) could refer to the format statement

```
16 FORMAT(E7.3)
```

If the first seven characters of the next record on

the file specified in the READ statement contain the characters

```
1.234E2
```

and a real variable X is specified in the READ statement, X will be assigned the value +123.4

If this record also contains other fields, the information given in these subsequent fields will not be read.

If the READ statement is executed again, X will be assigned the value corresponding to the first seven characters of another record

2 The CHARACTER*8 array AX(4) holds the following characters:

```
element 1   ( 3 P G 1 1 . 4
element 2   , 0 P E 7 . 3 ,
element 3   3 ( / E 7 . 3 )
element 4   ) X 3 / ) ) . ,
```

The first three elements and the first character of the fourth element form a format specification; the final seven characters of the fourth element are irrelevant. A reference in a READ statement to the array name AX is equivalent to a reference to statement label 101 where statement 101 reads as follows:

```
101 FORMAT(3PG11.4,0PE7.3,3(/E7.3))
```

If the READ statement contains a list giving the following names:

```
A,B,C,D,E
```

in that order, where all the names are those of real variables, the first eleven characters of the next record (say the nth card of a punched card file) will be read according to the G conversion code with a scale factor of 3, and the value assigned to variable A. The next seven characters of the card will be read according to the E conversion code with a scale factor of zero and the value assigned to variable B. The first seven characters of each of the (n+1)th, (n+2)th and (n+3)th cards will be read according to the E conversion code with a scale factor of zero and the values assigned to variables C, D and E respectively

3 The FORMAT statement

11 FORMAT(G11.4,2(E7.3,E8.5))

is used in conjunction with a WRITE statement which lists the following real variables:

A,B,C,D,E,F,G,H

in that order. The output records contains the values of the following variables:

Record	Characters	Variable	Format
1	1 to 11	A	G11.4
	12 to 18	B	E7.3
	19 to 26	C	E8.5
	27 to 33	D	E7.3
	34 to 41	E	E8.5
2	1 to 7	F	E7.3
	8 to 15	G	E8.5
	16 to 22	H	E7.3

As the output list is not exhausted when the format specification has been completely scanned, rescanning takes place as described in section 8.2.3.

CHAPTER NINE

INPUT AND OUTPUT

This chapter describes the input and output facilities available in FORTRAN 77.

9.1 Introduction

Reading data into and writing data out from main store is controlled by input/output statements.

The general form of these statements is discussed in section 9.2 where references to details of each statement can be found.

Each external file is identified in an input/output statement by a unique number, a unit number, which takes the form of an unsigned integer.

The identification of internal files is described in sections 9.3.1 and 9.6 below.

Items of data for input or output are grouped into records which can be either formatted or unformatted.

The records within an external file are either all unformatted or they are all formatted. These two types are described in more detail in sections 9.1.1.1 and 9.1.1.2 below and the formats of input/output statements for each are referenced in those sections.

Each input/output operation begins at the start of a record. The method of access to a file may be either serial or direct depending on the type of input/output device used. These methods of access are described in sections 9.1.2.1 and 9.1.2.2 respectively and the input/output statements available for each method are described in sections 9.3 and 9.4 respectively.

9.1.1 Format of records

9.1.1.1 Unformatted records

Unformatted records are input and output under the control of a READ or WRITE statement with no associated format specification. The records are representations of the internal machine form. Unformatted records will normally be output by the computer and used subsequently for

re-input rather than for examination by the programmer.

Details of input/output statements for unformatted records are found, for sequential access, in section 9.3.1.2, and, for direct access, in section 9.4.1.2.

9.1.1.2 Formatted records

Formatted records are input and output under the control of a READ or WRITE statement in conjunction with a format specification. The records are character representations of the internal values.

Some special considerations apply when punched cards are used to hold formatted records, and when records are output to a line printer. These are described in the following sections.

Details of input/output statements for formatted records are found, for sequential access, in section 9.3.1.1 and, for direct access, in section 9.4.1.1.

Punched cards

A formatted record on punched cards is one 80 column card.

Line printer

A formatted record output to the line printer is a line of characters. The maximum length of the record depends on the type of printer being used. Typical maximum line lengths are 120 and 132 characters. A record that is to be printed must begin with a print control character (see section 8.3.1.8).

9.1.2 Accessing records

9.1.2.1 Sequential access

In sequential access, each record is read or written in sequence starting with the first record on the file.

Records on any type of input/output device may be read or written sequentially. Records on some types of device must be accessed sequentially, for example, records on a printer. However, on certain types of input/output device such as disk it is possible to space backward past one or more records and to position at the first record on the device. Input and output using sequential access is described in section 9.3.

9.1.2.2 Direct access

In direct access, any record can be accessed directly in an order chosen by the user.

It is only possible to use direct access on certain types of input/output devices known as direct access devices. A typical direct access device is a magnetic disk. Input and output using direct access is described in section 9.4.

9.2 Input/output statements

The most important input/output statements are the READ and WRITE statements.

The READ statement has the effect of making values from external data records available to the program by assigning them to specified variables and array elements. The WRITE statement has the effect of forming external records from the values of specified variables and array elements.

The READ and WRITE statements can take various forms depending on the kind of record to be handled, the kind of file on which it is held, and the facilities used to control the handling of the record.

The most general forms of the READ and WRITE statements are:

```
READ(parameters)list
WRITE(parameters)list
```

where

parameters is a list of parameters which varies according to the kind of record being handled and the file being used.

list is an input/output list, which specifies the names of the items to have their values input or output (see section 9.2.1). A list parameter may appear for formatted or unformatted records (see section 9.2.1.1 for details of the correspondence between list items and the format specification).

The effects of the various READ and WRITE statements are described in sections 9.3 to 9.5.

There are three input/output statements, known as auxiliary input/output statements which may be used to describe, terminate, and

inspect a connection between a unit number and an external file. The statements are:

OPEN	see section 9.7.2
CLOSE	see section 9.7.3
INQUIRE	see section 9.7.4

Other input/output statements, known as file positioning input/output statements, are available for limited forms of input/output device. The following statements are available for certain sequential access input/output devices:

ENDFILE
REWIND
BACKSPACE

They are described in detail in section 9.3.2.

Other statements, called list directed statements (see section 9.5) are available. In these statements the format of the input or output data need not be specified.

9.2.1 Input/output lists

An input/output list is normally required in a READ or WRITE statement. The list has the form:

item1,item2,...,itemn

where each item can be the name of a variable, array, array element, character substring or an implied DO loop (see section 9.2.1.2). Additionally in a WRITE statement, an item may be any other expression that is not a character expression which involves concatenation of a character element whose length specification is of assumed size unless the character element is a symbolic constant. Any item or series of items may be enclosed in parentheses. An array name in an input/output list represents the whole array and thus corresponds to n separate items in the input/output list, where n is the total number of elements in the array taken in their order of storage (see section 3.1.3). Note that the name of an assumed-size dummy array must not appear in an input/output list.

An input/output list is normally required with all READ and WRITE statements, whether the data is to be transferred formatted or unformatted and whether the access method is sequential or direct.

9.2.1.1 Correspondence between input/output lists and format codes

When a READ or WRITE statement is executed, successive items in the input/output list are transmitted according to successive format codes. Where the format code is of a specific type, for example, the I code for integer values and the L code for logical values, then the corresponding item in the input/output list must be of the same type.

If, in an input/output statement, there are more items than format codes, then a new record is started and control is transferred to one of the following:

1 If there is a group format specification: to the group format specification repeat count that is terminated by the penultimate right hand parenthesis of the FORMAT statement

2 If there are no group format specifications: to the beginning of the FORMAT statement.

The same series of format codes is then used for the next items in the input/output list.

9.2.1.2 Implied DO loops

An implied DO loop is a series of list elements, usually array elements, that is to be repeated for different values of a DO-variable. An implied DO loop is used to simplify the specification of array elements required in input/output operations.

It takes the form:

(e1,e2,...,en,i=m1,m2,m3)

where

each e is a list element as defined in section 9.2.1. e may be another implied DO loop.

i is the DO-variable.

m1 is the initial parameter.

m2 is the terminal parameter.

m3 is the incrementation parameter, which may be omitted in which case it is assumed to have the value 1.

The DO-variable and parameters are analogous to those of a DO statement (see section 6.3.1). As with DO statements, implied DO loops may be nested.

i may be the name of an integer, real or double precision variable; m1, m2 and m3 may be any integer, real or double precision expression except that any functions referenced must not themselves carry out input or output operations.

The effect of the implied DO loop is the same as if the list e1,...,en had been written down once for each iteration of the implied DO loop with appropriate substitution of values for any occurrence of the DO-variable i.

For input lists, i, m1, m2 and m3 must not appear within the implied DO loops except in subscripts to array names.

If one e in an implied DO loop is a variable rather than an array then, on output, the same value will be output several times, and on input, several values will be assigned successively to the same variable, each value overwriting the previous value.

Example 1

A simple implied DO loop of the form

```
(A(I),I = -1,10,1)
```

would have the same effect as the input/output list

```
A(-1),A(0),A(1),...,A(10)
```

Example 2

An implied DO loop of the form

```
N,(A(I),B(I),I = 1,N),ALPHA(2)
```

transfers the data in the following order:

```
N,A(1),B(1),A(2),B(2),...,A(N),B(N),ALPHA(2)
```

Note that, in this example, N appeared in the same input/output list as an implied DO loop using it for indexing information. It also shows a specific array element, ALPHA(2), appearing in the input/output list

Example 3

The input/output list

```
A,M,MOD,((CAB(J,L),B(L), L = 1,N),J = 1,35,2)
```

causes the variables to be accessed in the following order:

A,M,MOD,CAB(1,1),B(1),CAB(1,2),B(2),...,CAB(1,N),B(N),CAB(3,1),
B(1),...,CAB(35,N-1),B(N-1),CAB(35,N),B(N)

Note that because of the position of array B in the nested implied DO loops, every element of B is accessed a total of 18 times

Example 4

```
(I,I = 1,10)
```

If used in a WRITE statement this implied DO loop would output integer numbers 1,2,...,10. However, this list would be invalid in a READ statement.

9.3 Sequential access input and output

Reading from and writing to sequential access input/output devices are carried out by READ and WRITE statements. The form of these statements is described below and the use of these statements for formatted and unformatted data is described in sections 9.3.1.1 and 9.3.1.2 respectively. Other sequential input/output statements are described in section 9.3.2.

9.3.1 READ and WRITE statements

The basic forms of these statements for sequential access are as follows:

```
READ(k,l)list
WRITE(k,l)list
```

where

k is either an integer variable or integer expression giving the unit number of the external file to be used in the input/output operation, or it is the name of a character variable, character array, character array element or character substring to be used in an internal file operation (see section 9.6).

The unit identifier may also be an asterisk which identifies an installation defined primary input or output channel that is the keyboard or the display. This form of unit identifier may only be used to read or write formatted records in a sequential manner.

l is normally a FORMAT statement label (see section 8.2.1) or a character variable or array name (see section 8.2.2). Other permissible forms are described in section 9.3.1.1.

If the records to be input or output are formatted the READ or WRITE statement must contain an l parameter, and if they are unformatted the statement must not contain an l parameter.

list is an input/output list as described in section 9.2.1.

9.3.1.1 Formatted sequential access input and output

Input

The appropriate form of the READ statement for formatted sequential input is

```
READ(UNIT=k,FMT=l,END=x1,ERR=x2,IOSTAT=m)list
```

where

UNIT=k is an unsigned integer constant, variable or expression that represents the unit number of the input/output file involved; or it may be an asterisk signifying input/output on an installation defined unit (see section 9.3.1). The characters UNIT= may be omitted, in which case the unit identifier must be the first item.

FMT=l identifies a format specification. A format identifier may be one of the following:

1 The statement label of a FORMAT statement

2 An integer variable that has been assigned the statement label of a FORMAT statement that appears in the same subprogram as the READ statement

3 A character array name

4 Any character expression that does not involve the concatenation of a character element which has a length specification of assumed size unless the character element is a symbolic constant

5 An asterisk, signifying list directed formatting

The characters FMT= may be omitted, but only if the format identifier is the second item in the list and if the first item is the unit identifier without the optional characters UNIT=.

END=x1 is optional, and x1 is the statement label to which control is transferred if an attempt is made to read data beyond the end of the file on unit number k.

ERR=x2 is optional, and x2 is the statement label to which control may be transferred when an error condition is detected.

IOSTAT=m is optional, and m is an integer variable or array element which is known as the input/output status specifier. Once the input/output statement has been completed it is assigned a value which indicates the existence of any abnormal condition encountered as follows:

1 If an end of file condition is encountered, m is assigned a value of -1

2 If an error condition is encountered, m is assigned a positive value which identifies the corresponding error message.

3 If no end of file condition or error condition exists, m is assigned a value of zero

If the input/output status specifier is omitted, program execution will terminate if either an end of file condition is encountered and the END= specifier is omitted, or if an error condition is encountered and the ERR= specifier is omitted.

list is optional and is an input/output list.

This READ statement reads in the items listed in list, under the

control of the format specification identified by l, from the file with unit number k.

An example of a formatted sequential READ statement is

```
READ(5,12)A,B,(C(I),I=1,10),J
```

In this example, data is read from a file with unit number 5 under control of the format specification in the format statement labelled 12. The variables A,B,C(1),C(2),...,C(10),J are given values in that order.

An alternative form of the READ statement is:

```
READ l,list
```

where

l identifies a format specification as in the READ statement above; it may not be preceded by the optional characters FMT=.

list is optional, and is an input/output list. If the input/output list is omitted then the preceding comma must also be omitted.

This form of the READ statement specifies an installation defined unit which is the same as the unit identified by an asterisk in the READ statement above.

Output

The appropriate form of the WRITE statement for formatted sequential output is

```
WRITE(UNIT=k,FMT=l,ERR=x2,IOSTAT=m) list
```

where k,l,x2,m and list are as for the READ statement above.

The WRITE statement outputs to the file with unit number k the items listed in list under the control of the format specification identified by l.

An example of a formatted sequential WRITE statement is

```
WRITE(6,101)X,((Y(I,J),I=1,10),J=1,5)
```

Data is written to the file with unit number 6, under the control of the FORMAT specification labelled 101, in the order X,Y(1,1),Y(2,1),..., Y(10,1),Y(1,2),...,Y(10,5).

An alternative form of the WRITE statement is:

```
PRINT 1,list
```

where

1 identifies a format specification as in the WRITE statement above.

list is optional, and is an input/output list. The preceding comma must be omitted if the input/output list is not specified.

The unit identified by the PRINT statement is installation defined, and is the same as the unit identified by an asterisk in the WRITE statement above.

9.3.1.2 Unformatted sequential access input and output

Input

The appropriate form of the READ statement for unformatted sequential input is

```
READ(UNIT=k,END=x1,ERR=x2,IOSTAT=m)list
```

where

UNIT=k is either an integer constant, integer variable or integer expression whose value is either zero or positive, and it represents the unit number of the external file involved; k may not be an asterisk.

The characters UNIT= are optional, and if they are omitted the unit specifier k must be the first item.

END=x1 is optional, and x1 is the statement label to which control is transferred if any attempt is made to read data beyond the end of the file on unit number k.

ERR=x2 is optional, and x2 is the statement label to which control may be transferred when an error condition is detected.

IOSTAT=m is optional, and specifies an input/output status specifier m where m is an integer variable or array element. After the input/output statement has been executed it may be examined to determine whether any abnormal condition was encountered as follows:

1 value of -1 indicates that an end of file condition

was encountered

2 A positive value indicates that an error condition was encountered and the value corresponds to an appropriate error message identifier

3 a value of zero indicates that no end of file condition nor error condition was encountered

If an end of file condition is detected while performing the READ statement and no end of file specifier (END=) nor input/output status specifier (IOSTAT=) is defined, then program execution will terminate. Similarly if an error condition exists and no error specifier (ERR=) nor input/output status specifier is defined, then program execution will also terminate.

list is optional, and is an input/output list.

When this READ statement is executed, the next record will be read and the values will be assigned in order to the variables listed in the input/output list. The number of items in the input/output list may be equal to or less than the number of values in the external record but it must not be greater than this number. If there is no input/output list, one external record is skipped.

An example of an unformatted sequential READ statement is as follows:

```
READ(8)X,(Y(2,K),K=1,5)
```

This statement causes data to be transferred from an input file to X,Y(2,1),Y(2,2),...,Y(2,5) in turn.

Output

The appropriate form of the WRITE statement for unformatted sequential output is

```
WRITE(UNIT=k,ERR=x2,IOSTAT=m)list
```

where

k, x2, m and list are as for the READ statement above.

When an unformatted sequential WRITE statement is executed, the values of the items listed in list will be output to the file associated with the unit k in the order in which they occur, in internal machine form. Each unformatted WRITE statement will cause one, and only one, new record to be created.

The WRITE statement

```
WRITE(9)A,B,C
```

causes variables A,B,C to be written, in that order, to the file with unit number 9.

9.3.2 File positioning input/output statements

These statements are for use with magnetic input/output devices only.

The ENDFILE statement

This statement has the following forms

```
ENDFILEi

ENDFILE (UNIT=i,ERR=el,IOSTAT=s)
```

where

UNIT=i is an integer constant or expression, or an integer variable whose value must be zero or positive and represents a unit number. The unit specifier may not be an asterisk.

The characters UNIT= are optional, and if they are omitted then the unit specifier must be the first item in the list.

ERR=el is optional, and el is the statement label to which control is transferred if an error condition is detected. el is known as the error specifier.

IOSTAT=s is optional, and s is an integer variable or array element which becomes defined with a zero value if no error condition exists. If an error condition does exist it becomes defined with a positive value which identifies the corresponding error message. s is known as the input/output status specifier.

If an error condition is detected, and if both the error specifier and the input/output status specifier are omitted, then program execution terminates. The ENDFILE statement defines the end of the data on a particular output file, with unit number i, by outputting an end-of-file record to that file.

After an ENDFILE statement has been executed, the file must be repositioned using either a REWIND or BACKSPACE statement prior to

executing a subsequent READ or WRITE on that file.

The REWIND statement

This statement has the following forms:

```
REWINDi

REWIND (UNIT=i,ERR=el,IOSTAT=s)
```

where i,el and s are as for the ENDFILE statement above.

When a REWIND statement is executed, the effect is that the next READ or WRITE statement referencing the same file will operate on the first record of the file.

Note that no input or output is performed. The file is positioned at its initial point.

The BACKSPACE statement

This statement has the following forms:

```
BACKSPACEi

BACKSPACE (UNIT=i,ERR=el,IOSTAT=s)
```

where i, el and s are as for the ENDFILE statement above.

When a BACKSPACE statement is executed, the effect is that the next READ or WRITE statement referencing the same file will operate on the previous record of the file. If the file was positioned at its first record before a BACKSPACE statement is encountered, then the statement will have no effect.

If the BACKSPACE statement occurs immediately after an ENDFILE statement, it has the effect of back-spacing over the end-of-file marker.

Backspacing over records which have been written using list-directed formatting (see section 9.5) is prohibited. Also prohibited is backspacing a file that is connected (see section 9.7.1) but does not exist.

9.4 Direct access input and output

Reading from and writing to a direct access input/output device is carried out by READ and WRITE statements of the form described below. At any time, any record may be read or written; there is no requirement to start at the first record. Writing to an output file alters only each record written, without destroying any record before or after it. Data items are not written across record boundaries, nor are they read from across record boundaries.

Data can be accessed directly only on direct access devices, usually magnetic disks.

Each record in a direct access file is assigned a number, called its record number, by which it can be referenced. The first record of a file is numbered one and the rest are numbered consecutively in steps of one. Record numbers appear in direct access input/output statements.

Note that all the records of a direct access file have the same length.

The READ and WRITE statements that are used for reading from and writing to direct access files are described in section 9.4.1 below.

9.4.1 READ and WRITE statements

The basic forms of these statements for direct access are as follows:

```
READ(k,REC=r)list
WRITE(k,REC=r)list
```

where

k is an integer variable, or integer constant or integer expression and gives the unit number to be used in the input/output operation. The unit number must be zero or positive.

r is an integer expression whose value is positive. It specifies the number of the first record that is to be read or written.

list is optional and is an input/output list as described in section 9.2.1.

9.4.1.1 Formatted direct access input and output

Input

The appropriate form of the READ statement for formatted direct access input is

```
READ(UNIT=k,FMT=f,REC=r,ERR=x2,IOSTAT=s)list
```

where

UNIT=k is an integer expression whose value is zero or positive, and it identifies a unit number. The characters UNIT= may be omitted provided that the unit identifier is the first item.

FMT=f identifies a format specification. A format identifier may be one of the following:

1 The statement label of a FORMAT statement

2 An integer variable that has been assigned the statement label of a FORMAT statement that appears in the same subprogram as the READ statement

3 A character array name

4 Any character expression that does not involve the concatenation of a character element operand which has a length specification of assumed size unless the character element is a symbolic constant

The characters FMT= may be omitted, but only if the format identifier is the second item in the list and if the unit identifier is the first item without the optional characters UNIT=.

REC=r is an integer expression whose value must be positive. It represents the record number of the first record which is to be read.

ERR=x2 is optional and x2 is the label of the statement to which control may be transferred when an error condition is encountered while executing the input/output statement.

IOSTAT=s is optional, and specifies an input/output status specifier. s is an integer variable or array element which becomes defined with a zero or positive value when the input/output statement has been executed. If no error condition exists then a value of zero is assigned, otherwise the value assigned is the number of the error message which corresponds to the error detected.

If the input/output status specifier and the error specifier are both omitted, program execution will terminate when an error condition is encountered.

list is optional and is an input/output list as defined in section 9.2.1.

This READ statement causes data to be transferred from a file on a direct access device into internal storage. The file from which data is being read must be a direct access file.

Output

The appropriate form of the WRITE statement for formatted direct access output is

```
WRITE(UNIT=k,FMT=f,REC=r,ERR=x2,IOSTAT=s)list
```

where k,f,r,x2,s and list are as for the READ statement above.

This WRITE statement causes data to be transferred from internal storage to a direct access device. The writing of the data starts at record r of the direct access file. The file to which data is written must be a direct access file.

If the input/output list is omitted the only data that will be written will be any character data that may appear at the beginning of the format specification. Each record to be written is completed with blanks if necessary.

9.4.1.2 Unformatted direct access input and output

Input

The appropriate form of the READ statement for unformatted direct access input is:

```
READ(UNIT=k,REC=r,ERR=x2,IOSTAT=s)list
```

where

UNIT=k identifies the unit number of the file involved, and must be an integer expression whose value is either zero or positive. The characters UNIT= are optional, and if they are omitted the unit specifier must be the first item in the list.

REC=r is an integer expression that represents the relative position of

a record within the file: its value must be greater than zero.

ERR=x2 is optional, and x2 is the label of the statement to which control is passed when an error condition is detected during data transfer to storage.

IOSTAT=s is optional, and defines an input/output status specifier where s is the name of an integer variable or array element. When the READ statement has been completed s may be examined to determine whether any abnormal condition exists. If no error condition exists then it has the value zero. Otherwise it is defined with a positive value which identifies an error message that corresponds to the fault detected. Program execution will terminate with appropriate diagnostics if an error is detected while performing the READ statement and both the input/output status specifier and the error specifier are omitted.

list is optional and is an input/output list as defined in section 9.2.1.

This statement causes the items of data in list to be transferred from a direct access device on unit number k into internal storage; only one record is read and so the input/output list must specify more values than can be contained in one record. The file from which the data is being transferred must be a direct access file.

Output

The appropriate form of the WRITE statement for unformatted direct access output is:

```
WRITE(UNIT=k,REC=r,ERR=x2,IOSTAT=s)list
```

where k,r,x2,s and list are as for the READ statement above.

This statement causes the data items of list to be transferred from internal storage to a direct access file on unit number k. Writing commences at the rth record within the file.

The input/output list must not specify more values than can fit into a single record. If the values specified do not fill the record, the remainder of the record becomes undefined. If the input/output list is omitted then the entire output record becomes undefined.

An example of an unformatted direct access WRITE statement is

```
WRITE(ERR=999,UNIT=30,REC=I+J) IARRAY,(A(I,K),K=4,8)
```

This statement will write a record to the file with unit number 30. The value of the expression I+J identifies the particular record within the

file to which the variables IARRAY, A(I,4), A(I,5), A(I,6), A(I,7), A(I,8) are to be written. Control will be transferred to the statement labelled 999 should an error condition occur (for example if the record specifier (REC=) has a negative value).

9.5 List directed input and output

The use of list directed input/output statements allows data to be read or written without the restrictions imposed by format specifications.

9.5.1 The READ statement

The list directed READ statement may take one of the following forms:

```
READ(UNIT=k,FMT=*,END=x1,ERR=x2,IOSTAT=s)list READ*,list
```

where

UNIT=k gives the unit number of the file to be used in the input/output operation. It takes the form of an integer expression whose value is zero or positive. The characters UNIT= are optional, and if they are omitted the unit specifier must be the first item in the list.

FMT=* specifies that list directed formatting is to be used. The characters FMT= may be omitted, but only if the asterisk is the second item in the list and the first item is the unit specifier without the optional characters UNIT=.

END=x1 is optional, and x1 is the statement label to which control is to be transferred if an attempt is made to read beyond the end of the file on unit number k.

ERR=x2 is optional, and x2 is the statement label to which control is to be transferred if an error condition is detected.

IOSTAT=s is optional, and s is an integer variable or integer array element which becomes defined with a value that specifies the existence of some abnormal condition as follows:

1 If no abnormal condition exists, s is assigned the value zero

2 If an end of file condition exists, s is assigned the value of -1

3 If an error condition exists, s is assigned a positive value which identifies an appropriate error message that

describes the fault

s is known as the input/output status specifier.

list is an input/output list as defined in section 9.2.1.

If an abnormal condition exists after the completion of the READ statement program execution is not terminated if an input/output status specifier is defined. Nor will program execution be terminated if an end of file condition exists and an end of file specifier is defined, or if an error condition exists and an error specifier is defined.

Execution of the READ statement causes values to be read from external records and given to the items of the input list in order. In the case of an array name the elements are given values in order of storage (that is, with the leftmost subscript expression varying most rapidly). A value is terminated by a value separator which may be one of the following:

1 A comma optionally preceded by one or more spaces and optionally followed by one or more spaces

2 A slash optionally preceded by one or more spaces and optionally followed by one or more spaces

3 One or more spaces between two values or following the last value

The input operation is terminated by the satisfaction of the input list or by the reading of a slash.

Items of the input list are not redefined if null data items (see below) are encountered or if a slash is encountered before their values are read.

The type of each item from the input list must correspond with the form of data from the external medium.

Each READ statement starts with a new record, and reads as many records as are necessary to provide data to fill the input/output list.

9.5.2 Input data

When a list directed READ statement is executed, reading begins at the start of the next unaccessed record on the input file and continues until either each item in the input list has been given a value or a slash (/) is encountered in the input. Any data in the last record

accessed by a READ statement which follows a slash or is not required for input cannot be accessed. Any input list items not given a value before a slash is reached retain their current value (or remain undefined).

The input stream consists of a series of data items which are associated in their order of occurrence with the items of the input list. Data items are separated by one or more spaces or by a single comma optionally preceded and optionally followed by spaces. Note that the end of a record has the effect of a space, except when it appears within a character constant (see below). An item may be:

1 A NUMERIC CONSTANT This may take any of the forms listed in section 2.2.1 except Hollerith constants. Numeric constants may not contain any embedded spaces except between the parts of a complex constant, in which case any number of spaces is permissible. The end of a record may not appear within a constant unless the constant is a complex value, in which case the end of record may occur between the real part and the comma or between the comma and the imaginary part.

The type of the constant must be the same as that of the corresponding list item, but there need be no correspondence of length

2 A LOGICAL CONSTANT If the corresponding item is of type logical, the data item may be any value acceptable to L editing (see section 8.3.1.6). However commas or slashes are not permitted as optional characters

3 A CHARACTER CONSTANT A data item may be a non-empty string of characters enclosed within apostrophes. Note that the form nH... is not permitted. Each apostrophe that is part of the character value must be represented by two consecutive apostrophes. The constant may be continued on as many records as needed and an end of record does not cause a space or any other character to become a part of the value.

The corresponding input list item need not be of type character, and there need be no correspondence of length. Note that the constant is assigned in the same manner as if the constant appeared in a character assignment statement (see section 5.3)

4 A NULL ITEM A null item may take one of the following forms:

(a) No characters appearing between two successive value separators

(b) No characters preceding the first value separator in

the first record input by a READ statement

The value (or undefined status) of the corresponding list item is left unchanged.

5 REPEATED ITEMS Any of the above items may be preceded by a basic positive integer constant and an asterisk (n*). n* must not contain any embedded spaces and may not extend over a record. A repeated null item occurs if the next character after * is a value separator.

The effect of a repeated item is that the next n items from the input list have the same value read into them; they must all have the same type as the value. If a null item is repeated, the next n items from the input list are left unchanged

A constant is terminated by the first space, end of record, or comma after its syntactic completion (that is, after a closing bracket for a complex constant or the closing apostrophe for a character constant).

Note that spaces are never used as zeros, and all spaces are considered to be part of some value separator except in the following circumstances:

1 Spaces embedded within a character constant

2 Spaces which precede or follow the real or imaginary part of a complex constant

3 Leading spaces in the first record input by a READ statement unless they are immediately followed by a comma or a slash

As an example, if the next records on an input medium with record length 40 characters are

```
1056#198765,#####,####,####50*##########
50*(4E1,5E1)###T,'ONE''S'#'TWO',/##456##
```

and the file is accessed by the statement

```
READ*,I,J,K,L,(A(M),M=1,50),(B(M),M=1,50),P,X,Y,N
```

I is given the value 1056, J the value 198765, K, L and A are unchanged, the first 50 elements of B are each given the complex value 40+50i, P the value true, X the value ONE'S and Y the value TWO. N remains unchanged and the value 456 is not accessed since the slash intervenes.

9.5.3 Output statements

Execution of the list directed form of the WRITE or PRINT statements causes the values of the items of the output list to be output in order to external data sets, for example, line printer, disk and magnetic tape data sets.

9.5.3.1 The WRITE statement

This statement has the form:

```
WRITE(UNIT=k,FMT=*,ERR=x2,IOSTAT=s)list
```

where k,x2,s and list are as for the READ statement above.

The WRITE statement causes data to be written to an external data set identified by the integer expression k.

9.5.3.2 The PRINT statement

This statement has the form:

```
PRINT*,list
```

where list is an input/output list as defined in section 9.2.1.

The PRINT statement causes data to be shown on the display. The unit will be the same as if an asterisk had been specified in a list directed WRITE statement. The following is an example of a list directed PRINT statement

```
PRINT*,I,J,K,(A(I),I = 1,100)
```

9.5.4 Output data

When a list directed WRITE or PRINT statement is executed, the values of all elements of each list item are output in order. Each record starts with a single space and contains at least one space between each value output and no embedded spaces within items (other than spaces within character values). A record may end with no spaces or with one or more spaces.

The forms of output are as follows:

1 For integer values, all digits are output except for leading

zeros. If negative, the value is preceded by a minus sign. No exponent is used

2 For real values, all significant digits are output. If the value to be output contains d significant digits, and the value is greater than or equal to 0.1 and less than 10**d, the number is output in a form which is similar to the effect of using an F edit descriptor (see section 8.3.1.2) with a zero scale factor, that is, without an exponent; otherwise, the number is output with an exponent in a form that is similar to the effect of using an E edit descriptor (see section 8.3.1.3) with a scale factor of 1. The value is preceded by a minus sign if it is negative

3 For complex values, an opening parenthesis output followed by the value of the real part, followed by a comma, followed by the imaginary part, followed by a closing parenthesis. The real and imaginary parts are output as for real values

4 For logical values, the single character T or F is output

5 For character values, all the characters are output without spaces preceding or following the characters other than any spaces that may be part of the character value. Character values that are output are not delimited by apostrophes

As many records as are necessary will be written but the end of a record will not occur within a value, apart from a complex or character value. The end of a record may appear within a complex value between the comma and the imaginary part only if the entire constant is as long as, or longer than an entire record. Character values will be extended across as many records as required and each such new record will have a space character inserted at the beginning for carriage control.

Note that slashes, as value separators, and null items are not produced by list directed formatting.

9.6 Internal files

If the first parameter to a sequential READ or WRITE statement (see section 9.3.1) is the name of a character variable, character array element, or character array, or if it is a character substring, then the input/output operation is to be carried out on an internal file consisting of that variable, array element, array, or substring.

An internal file has the following properties:

1 A record of an internal file is a character variable or character array element or character substring

2 If the file is a character variable, character array element, or character substring it consists of a single record whose length is that of the variable, array element, or substring, respectively. If the file is a character array it is treated as a sequence of character array elements, each of which is a record of the file. Each record of the file has the same length, namely the defined array element length

3 If the number of characters being written to the file is less than the length of the record, the remaining portion of the record is filled with blanks

4 An internal file is always positioned before the first record at the start of a READ or WRITE statement accessing that file

5 Reading and writing records may only be performed using sequential access formatted input/output statements that do not specify list-directed formatting

The character variable, character array element, or character array being used as an internal file must not appear in the input/output list nor contain the format being used when accessing that file.

Example

In the following code

```
      CHARACTER*16 TEXT
      .
      .
      .
      WRITE(TEXT,10)I
   10 FORMAT('VALUE OF I =',I4)
```

if I contains the value of 136 when the WRITE statement is obeyed the effect will be equivalent to assigning the character string 'VALUE OF I = 136' to the character variable TEXT.

9.7 Auxiliary input/output statements

The input/output statements above decribe the manner in which data may be transferred between internal storage and external media, and between internal storage and internal files. They also describe file

positioning statements. Auxiliary input/output statements may be used to manipulate the external medium, or to interrogate or describe the manner in which an external medium may be accessed, that is they operate upon the properties of the connection between a given unit and an external file.

9.7.1 Unit and file connection

The physical association of a unit to an external file is known as a connection. Prior to program execution a connection may either be defined externally by some job control statement, or be predefined by the system. This is known as preconnection. For example, the list directed input statement

```
READ *,list
```

makes use of the preconnection to the primary input medium which is installation defined.

Internal to a program a connection may be established by means of the OPEN statement (see section 9.7.2)

A connection is between a unit and a file. No unit may be connected to more than one file at the same time and similarly no file may be connected to more than one unit at the same time. However, means are available to terminate a connection (see section 9.7.3), and to connect a unit to a different file. A READ, WRITE, or PRINT statement cannot be executed without a connection to the specified unit.

The properties of a connection include the following:

1 The type of access, either direct access or sequential access

2 The kind of records, either formatted or unformatted

3 The length of the records if the file is to be accessed with direct access input/output statements

File existence is totally independent of a connection, that is a file may be connected but not exist (see section 9.7.1.1). An example is a preconnected new file.

9.7.1.1 File existence

A file is said to exist for a program if it may transfer data either to or from the file, provided that it does not have to be created first. For example, for security reasons a program may be denied access to a

file that physically exists; such a file is said not to exist for the program. Note that a connection to a file does not imply that the file exists.

A new file may be created by executing an OPEN statement (see section 9.7.2) or by writing a record to the file when the file is preconnected.

Such a file is then said to exist.

9.7.1.2 File properties

A file property is a characteristic of an external file which exists for the life-span of the file. Taken together a file's properties describe the permissible methods that may be used to access the file.

Within the context of FORTRAN 77 a file is attributed the following properties:

1 A file may exist, or it may not exist. If it does not exist, then it has no other property

2 Its records may either be all formatted, or all unformatted. A file may not contain both types of record

3 It may be accessed with direct access input/output statements, or it may be accessed with sequential access input/output statements. Some files may be accessed with either type of statement, but note that a given connection is only for a single type of access

4 A file may have a name. If it has no name then it is a temporary file which will cease to exist after program termination

When a connection is defined (see section 9.7.2) between a unit and a file, the properties of the connection must be compatible with the properties of the file. For example, it is not valid to define a connection for direct access when the file to be connected is a line printer.

9.7.2 The OPEN statement

The OPEN statement provides a means of accessing files that are not preconnected. If a file does not exist then it will be created. The OPEN statement may also be used to create a new file that is preconnected, or to alter certain properties of a connection between a

file and a unit.

Once the OPEN statement has established a correspondence between a given unit number and a specified file then both the unit and the file are said to be connected, and hence a READ or a WRITE statement can be executed on the unit and hence the file. Without a connection (or preconnection) a READ or WRITE statement cannot be executed.

The general form of the OPEN statement is:

```
OPEN(UNIT=u,IOSTAT=ios,ERR=x2,FILE=fn,STATUS=sta,ACCESS=acc,
FORM=fm,RECL=rl,BLANK=blk)
```

where

UNIT=u identifies a unit number where u is an integer expression whose value is either zero or positive. The characters UNIT= are optional and if they are omitted the unit specifier must be the first item in the list; otherwise its position in the list is not fixed.

All other specifiers may be omitted, and if they are specified they may appear anywhere within the list. The specifiers are described below together with any assumed value that may be used if a specifier is not defined.

IOSTAT=ios defines an input/output status specifier which may be the name of either an integer variable or integer array element. It will become defined with either a positive value or a value of zero. If an error condition exists the input/output status specifier is assigned the identifier of an error message which corresponds to the error; if no error condition is detected it is assigned the value zero.

Program execution will terminate if an error condition is detected and neither an input/output status specifier nor an error specifier (see below) is defined.

ERR=x2 is an error specifier and defines a statement label to which control is transferred if an error condition exists. If the error specifier is omitted, and also the input/output status specifier (see above), then program execution will terminate when an error condition is detected.

FILE=fn specifies the name of a file to be connected to the defined unit. fn is a character expression whose value must represent a valid filename once any trailing spaces have been removed. If the file specifier is omitted then the value assumed by fn depends whether the specified unit is connected. Should the unit be connected, then the name of the file to which it is connected is assumed; otherwise the default value used for the file specifier is dependent upon the value

of the status specifier (see below).

STATUS=sta is a status specifier sta is a character expression whose value may be one of the following:

OLD
NEW
SCRATCH
UNKNOWN

Any trailing spaces in the value are ignored. The status specifier defines the existence of the file to be connected. If OLD is specified the named file must exist. Conversely if NEW is specified the named file must not exist but it will be created by the OPEN statement provided no error occurs. The values OLD or NEW may only be used if a file specifier is defined; while the value SCRATCH may only be used if no file specifier is defined. SCRATCH causes the specified unit to be connected to a temporary file which exists only until either that unit is closed (see section 9.7.3) or the program terminates.

If the status specifier is omitted, the default value is UNKNOWN. UNKNOWN assumes the status of the named file if a file specifier has been defined (that is either NEW or OLD) or the status of the file to which the unit is connected if no file specifier has been defined. If there exists no connection and the file specifier is omitted then UNKNOWN assumes the value SCRATCH.

Once the OPEN statement has successfully established a connection, the status of the connected file becomes OLD unless the file is a temporary file.

ACCESS=acc defines the manner in which the connection is to access the file. acc is a character expression whose value may be either SEQUENTIAL or DIRECT; any trailing spaces in the value will be ignored. The value SEQUENTIAL specifies sequential input/output and the value DIRECT specifies direct access input/output. When a new file is created the specified access method becomes a property of the file, that is the file is created as a sequential or direct access file; while for an existing file the specified access method must be among the properties of the file. A value of SEQUENTIAL will be assumed if the access specifier is omitted.

FORM=fm specifies whether the file is to be accessed with either formatted or unformatted input/output statements. fm is a character expression whose value when trailing spaces have been removed is either FORMATTED or UNFORMATTED. If FORMATTED is specified the connected file may contain no unformatted records, and if UNFORMATTED is specified the connected file may contain no formatted records. Note that the type of the records is a file property. If the form specifier is omitted, a

value of UNFORMATTED is assumed if the connection specified is for direct access, while a value of FORMATTED is assumed if the connection is for sequential access.

RECL=rl is a record length specifier. rl is an integer expression whose value must be positive. It specifies in units of bytes the length of each record in a file being connected for direct access. For an existing file the specified length must not be greater than the actual record length of the file. For a new file, the OPEN statement creates the file with a record length of rl. If the connection defined is for sequential access, the record length specifier must be omitted; otherwise it must be specified.

BLANK=blk may only be specified for a connection which is to be used for formatted input/output, and defines the interpretation to be applied to space characters within numeric input fields. blk is a character expression whose value when any trailing spaces have been removed is either ZERO or NULL. If ZERO is specified then all spaces in numeric input fields read from the specified unit are treated as zeros apart from leading spaces. If NULL is specified all spaces are ignored. A field which consists entirely of spaces always has the value zero. NULL is the assumed value if the blank specifier is omitted.

Example 1

```
OPEN(UNIT=273,FILE='FIL001')
```

defines a connection between unit 273 and the file FIL001. In the absence of other specifiers the connection is specified for sequential formatted input/output and any spaces which are read in numeric fields are to be ignored. As the status specifier has not been defined a value of UNKNOWN is assumed and the file will be created if it does not exist.

Example 2

```
OPEN(ACCESS='DIRECT',RECL=160,NREC=25,UNIT=10,
   FILE='DFX',STATUS='OLD')
```

connects the file DFX to unit 10 for direct access. The file is specified as having 25 records each of which is unformatted and 160 bytes long. The file is also assumed to exist.

Example 3

```
OPEN(1,STATUS='UNKNOWN',BLANK='ZERO')
```

either refers to an existing connection to unit 1 and changes the interpretation of spaces within formatted numeric input fields to ZERO; or if no connection exists a temporary file is created and connected to unit 1 for formatted input/output.

Example 4

```
OPEN(22,ACCESS='DIRECT',BLANK='NULL')
```

is not a valid statement as the record length specifier is not defined. In addition because the form specifier is omitted, the assumed access form is unformatted input/output which is incompatible with the specification of the blank specifier.

9.7.2.1 Changing the properties of a connection

When a unit becomes connected to a file, the same unit may appear in an OPEN statement to either define a new connection or to change certain properties of the current connection.

A new connection is defined when the filename specified in an OPEN statement is not the same file as the file to which the specified unit is already connected. The effect is as if an implicit CLOSE statement (see section 9.7.3) without a status specifier is executed immediately prior to the OPEN statement.

When the file specifier is the same as the name of the connected file then the current connection is specified. If the file is a file that is preconnected and does not exist, the values specified by the OPEN statement become a part of the connection and the file is also created. Otherwise, should the connected file exist then only the BLANK= specifier may have a value that is different from the one currently in force. Note that if the file specifier is omitted then the assumed filename is the name of the connected file unless STATUS=SCRATCH was specified.

A file which is already connected to a unit may not be connected to another unit unless its current connection is first terminated by a CLOSE statement (see section 9.7.3).

Example 1

The sequence

```
OPEN(73,FILE='DATA3',...
.
OPEN(73,FILE='DATA4',...
```

will first connect unit 73 to the file DATA3. At the second OPEN statement, unit 73 becomes connected to the file DATA4 after first terminating the original connection.

Example 2

The sequence

```
OPEN(2000,FILE='RESULTS')
.
OPEN(2000,BLANK='ZERO')
```

will perform the connection of file RESULTS to unit 2000. The connection will be defined for sequential formatted input/output with any spaces in numeric fields being ignored apart from such fields which consist entirely of spaces. The effect of the second OPEN statement is to change the interpretation of the spaces. No other property of the connection is affected.

Example 3

The sequence

```
OPEN(10,FILE='OUTPUT',...
.
OPEN(44,FILE='OUTPUT',...
```

is not permitted as it attempts the simultaneous connection of the file OUTPUT to two units.

9.7.3 The CLOSE statement

The CLOSE statement terminates the connection of a unit, and hence terminates the connection of the file to which it is connected. After a CLOSE statement has been executed no input/output statement may refer to the given unit unless the unit is connected again by an OPEN statement.

The CLOSE statement may be used to optionally destroy the associated file, that is, to cause it not to exist after the statement has been executed. Note that once a file has been disconnected it will be free to be connected again (provided that it still exists), either to the same unit or to another unit.

The general form of the CLOSE statement is:

```
CLOSE(UNIT=u,IOSTAT=ios,ERR=x2,STATUS=sta)
```

where

UNIT=u is an integer expression that identifies the unit to be disconnected. Its value must be zero or positive. The characters UNIT= may be omitted but in this case the unit specifier must be the first item in the list, otherwise the position of the specifier is not fixed.

Note: the remaining specifiers are optional, and if they are defined they may appear anywhere in the list. The specifiers are described below together with any assumed value or action that may be taken if a specifier is not defined:

IOSTAT=ios is an input/output status specifier that defines an integer variable or integer array element which becomes assigned with a positive or zero value. When an error condition exists the input/output status specifier is assigned a positive value which corresponds to an error message which describes the error. When no error exists it is set to zero.

ERR=x2 defines a statement label to which control is transferred if an error condition is detected. If both the error specifier and the input/output status specifier are omitted then program execution will terminate when an error occurs.

STATUS=sta is a status specifier. sta is a character expression whose value may be either KEEP or DELETE; any trailing spaces are ignored. If DELETE is specified the connected file will cease to exist; DELETE is the only value that may be specified for a file which is a temporary file, that is one whose status was SCRATCH before the CLOSE statement. If KEEP is specified for a preconnected file which does not exist, the file will still not exist after the CLOSE; otherwise if it is specified

for an existing file then the file will continue to exist.

If the status specifier is omitted the assumed value is KEEP, unless the file status prior to the CLOSE statement was SCRATCH in which case the assumed value is DELETE.

It is quite permissible in a CLOSE statement to specify a unit which is not connected. It has no effect on the unit and it affects no file.

When program execution terminates, each unit that is still connected is closed with STATUS=KEEP, unless the file to which it is connected is a temporary file, that is one which was created with STATUS=SCRATCH, in which case it is closed with STATUS=DELETE. Note that the effect is the same as if the unit was specified in a CLOSE statement with no status specifier defined.

9.7.4 The INQUIRE statement

The INQUIRE statement is used to interrogate the properties of a particular file or the properties of the connection to a particular unit. The given file or unit need not be connected.

There are two forms of the INQUIRE statement:

```
INQUIRE by unit
INQUIRE by file
```

An INQUIRE by unit statement interrogates a specified unit and the file to which it is connected if any; while the INQUIRE by file statement interrogates the properties of a specified file which may or may not exist.

9.7.4.1 INQUIRE by unit

The general form of an INQUIRE by unit statement is:

```
INQUIRE(UNIT=u,slist)
```

where

UNIT=u is the unit specifier which defines an integer expression. The value of the integer expression must be either zero or positive and specifies the unit number being inspected. The characters UNIT= may be omitted in which case the unit specifier must occur in the position indicated above; otherwise it may appear anywhere within slist.

slist is a set of INQUIRE specifiers (see section 9.7.4.3) which may be

an empty list. Each specifier which is not omitted inquires about a particular property of the specified unit or file to which it is connected.

9.7.4.2 INQUIRE by file

The general form of an INQUIRE by file statement is:

```
INQUIRE(FILE=fn,slist)
```

where

FILE=fn specifies the name of the file being interrogated and may appear anywhere within slist. fn is a character expression whose value represents a valid file name; any trailing spaces are ignored. The file may or may not either exist or be connected.

slist is a list of INQUIRE specifiers (see section 9.7.4.3) which may be empty. Those specifiers which are defined inquire about a particular property of the file.

9.7.4.3 The INQUIRE specifiers

Any of the specifiers defined below may be used with either form of the INQUIRE statement. Each specifier is optional, and may appear anywhere within the list of INQUIRE specifiers. A description of the specifiers follows:

IOSTAT=ios

where ios is an integer variable or array element which becomes defined with a zero value if no error condition exists or with a positive value if an error condition was detected. ios is known as an input/output status specifier. If it becomes defined with a positive value then the value identifies an error message which describes the error encountered.

ERR=x2

where x2 is an error specifier that defines a statement label to which control is transferred if an error condition exists. If no error specifier nor input/output status specifier is defined then program execution will terminate when an error occurs.

EXIST=ex

where ex is a logical variable or logical array element. For an INQUIRE

by file statement it is assigned the value .TRUE. if the specified file exists, otherwise it is assigned the value .FALSE. If the inquiry relates to a unit, ex is always assigned the value .TRUE., that is, any unit number that is zero or positive may be used for input/output. Note that some implementations may provide a more restricted set of unit numbers.

OPENED=op

where op is a logical variable or logical array element that is always assigned a value when this specifier is defined. In an INQUIRE by file statement it is assigned the value .TRUE. if the specified file is connected to a unit and the value .FALSE. if it is not connected to any unit. Similarly, in an INQUIRE by unit statement it is assigned the value .TRUE. if the specified unit is connected to a file, and the value .FALSE. if it is not currently connected.

NUMBER=num

where num is an integer variable or integer array element which only becomes defined with a value if the specified unit or file is currently connected, in which case it is assigned the number of the connected unit. If there is no connection num becomes undefined.

NAMED=nmd

where nmd is a logical variable or logical array element. For an INQUIRE by unit statement nmd becomes undefined only if the unit is not connected to a file, while for an INQUIRE by file statement it becomes undefined only if the specified file does not exist. When nmd does become defined, it is assigned the value .TRUE. if the file exists and has a name; otherwise it is assigned the value .FALSE.

NAME=nm

where nm is a character variable or character array element which is assigned the name of the file only if the NAMED= specifier may be assigned the value .TRUE.; otherwise nm becomes undefined. Note that the value assigned to nm may not necessarily be the same as that defined by the FILE= specifier in an INQUIRE by file statement; however it will always represent an acceptable value for the file specifier in an OPEN statement. For example the file name may be qualified by a user name.

ACCESS=acc

where acc is a character variable or character array element which only becomes defined if a connection exists. acc is assigned the value SEQUENTIAL if the connection is for sequential input/output, or it is

assigned the value DIRECT if the connection is for direct access input/output.

SEQUENTIAL=seq

where seq is a character variable or character array element. It is assigned the value YES if sequential access is one of the permitted forms of access method for the file. A value NO is assigned if the file may not be accessed sequentially. If the access property of the file cannot be determined, for example a new preconnected file, a value of UNKNOWN is assigned.

DIRECT=dir

where dir is a character variable or character array element which is assigned the value YES if direct access is one of the permitted forms of access method for the file. If the file property precludes direct access, for example the primary card input source, dir will be assigned the value NO. A value of UNKNOWN is assigned if for some reason the access property of the file cannot be determined.

FORM=fm

where fm is a character variable or character array element which describes the form of the current connection. It is assigned the value FORMATTED if the connection is for formatted input/output statements, and the value UNFORMATTED if the connection is for unformatted input/output statements. fm becomes undefined if there is no connection.

FORMATTED=fmt

where fmt is a character variable or character array element. It describes whether formatted input/output statements may be used to access the file. If the file consists of formatted records, fmt is assigned the value YES, while if the file consists of unformatted records it is assigned the value NO. In some circumstances it may not be possible to determine the permitted form and fmt will be assigned the value UNKNOWN.

UNFORMATTED=unf

where unf is a character variable or character array element that is assigned the value YES if unformatted input/output statements may be used to access the file. Otherwise it is assigned the value NO when only formatted input/output statements may be used. In circumstances when the permitted form cannot be determined unf becomes defined with the value UNKNOWN.

RECL=rl

where rl is an integer variable or integer array element which becomes defined only if there is a connection and the connection is for direct access, otherwise it becomes undefined. rl is assigned the value of the record length of the file in units of characters (bytes).

NEXTREC=nr

where nr is an integer variable or integer array element. If the file is connected and the connection is for direct access, nr is assigned the record identifier of the record which follows the last record that was written or read. Otherwise nr becomes undefined. Note that if no record has been accessed since the file was connected a value of 1 is assigned.

BLANK=blk

where blk is a character variable or character array element and applies only to a connection which is for formatted input/output. If any spaces within a numeric input field are to be ignored then blk is assigned the value NULL. If spaces other than leading spaces are interpreted as zeros then blk is assigned the value ZERO. blk becomes undefined if there is no connection or if the connection is for unformatted input/output.

When using an INQUIRE by unit statement or an INQUIRE by file statement, the following points should be noted:

1 Unless an error condition exists, the opened specifier (OPENED=) and the exist specifier (EXIST=) always become defined. If an error condition does exist then all the INQUIRE specifier values become undefined.

2 No variable or array element may be defined more than once in the list of INQUIRE specifiers

3 If the exist specifier and the opened specifier become defined with the value .TRUE. within an INQUIRE by unit statement, then all the other specifiers become defined

4 Within an INQUIRE by file statement, if the opened specifier is assigned the value .FALSE., that is if the file is not connected, then the number, access, form, record length, next record, and blank specifiers become undefined. Note that if the file is connected the record length, next record, and blank specifiers need not become defined

5 Within an INQUIRE by file statement, if the exist specifier

is assigned the value .TRUE., then the named, name, sequential, direct, formatted, and unformatted specifiers become defined; otherwise they become undefined.

Example

Assume that the statement

```
OPEN(FILE='DATAFILE',ACCESS='DIRECT',RECL=80,UNIT=730)
```

has been successfully executed and has connected an existing file DATAFILE to unit 730. The properties of the connection are defined as direct access input/output with unformatted records whose length is 80 bytes. Also assume that no other connection or preconnection exists.

Then

```
INQUIRE(UNIT=9,EXIST=L1,OPENED=L2,DIRECT=C1)
```

will assign the value .FALSE. to L2 as unit 9 is not connected and consequently C1 becomes undefined. L1 will be assigned the value .TRUE. as the specified unit may be used for input/output.

```
INQUIRE(FILE='INPUTS',OPENED=L1,NUMBER=I1)
```

will assign the value .FALSE. to L1 as the file INPUTS is not connected to a unit and I1 will therefore become undefined. Note that the file's existence has no effect on the values set.

```
INQUIRE(BLANKS=C1,NEXTREC=I1,ACCESS=C2,FORMATTED=C3,
    FILE='DATAFILE',OPENED=L1,EXIST=L2)
```

interrogates the connection defined above and consequently L1 and L2 are assigned the value .TRUE., and C2 the value DIRECT. C3 is assigned the value NO which causes C1, the blank specifier, to become undefined as the connection is for unformatted input/output. The value assigned to I1 defines the current position within the file. If no records have been accessed since the connection was defined I1 is assigned the value 1, otherwise it is assigned the record identifier of the record which follows sequentially after the last record accessed.

CHAPTER TEN

LANGUAGE EXTENSIONS AND IMPLEMENTATION CHARACTERISTICS

10.1 Language extensions

PERQ FORTRAN 77 adheres closely to the ANSI FORTRAN 77 standard as defined in ANSI X3.9-1978. However, a few extensions to the language, as described in the following subsections, are provided as a transition aid from other FORTRAN dialects, and to allow PERQ-specific features to be exploited.

Such extensions are allowable within the ANSI77 standard since they do not conflict with the standard definition, but they should not be used in programs that are intended to be portable to other implementations of FORTRAN 77. The use of an extension is flagged with a warning by the compiler.

10.1.1 Lower case

Lower case may be used in a FORTRAN source program, but is treated as equivalent to upper case everywhere except in character literals. (Note that at run time, lower and upper case characters are not treated as equivalent.)

10.1.2 Hollerith constants

Hollerith constants may be used for data initialization in DATA statements and in the argument list of a call statement. In data statements, variables and array elements may be initialized by Hollerith constants and each constant must have a length which is less than or equal to the length of the item. If the constant is shorter than the item, it is extended on the right with blanks.

When initialized with Hollerith constants, an integer, real, double precision, complex, or logical array may be used as a format specification.

An Aw edit descriptor may be used with Hollerith data when the input/output list item is a type integer, real, complex, or logical.

An actual argument in a subroutine reference may be a Hollerith constant. The corresponding dummy argument must be of type integer, real, double precision, or logical. If the length of the constant is one to four bytes then a four byte argument is passed (blank characters

being added to the right if necessary). If the length of the constant is five to eight bytes then an eight byte constant is passed.

10.1.3 Symbolic names

Symbolic names may include up to 32 alphanumeric characters all of which are significant.

10.1.4 Calling Pascal

A PERQ FORTRAN 77 program can reference Pascal procedures and functions. The names of the procedures and functions must be declared in a special form of the EXTERNAL statement:

```
EXTERNAL/PASCAL/name1,name2,...,namen
```

where name1,name2,...,namen is the list of Pascal procedures and functions to be referenced from FORTRAN. Once the procedures and functions have been so declared, they may be referenced in FORTRAN as if they were FORTRAN procedures and functions, but they may not be passed as arguments (see Chapter 11 and sections 12.1 and 12.3.2.6).

10.1.5 Character length

The length of character variables, array elements, or constants may be from 1 to 32767 characters.

10.1.6 Integer*2

Integer*2 may be used as a type specificaiton in IMPLICIT, FUNCTION and type statements to define integer items which occupy two bytes of storage.

Example:

```
INTEGER *2   I2,I2ARRAY(10,10)
```

Integer*2 variables or array elements may not be used as:

1) A DO-variable in a DO statement or implied DO
2) An integer variable name in an ASSIGN statement
3) Any specifier in an input/output statement

10.2 Implementation characteristics

10.2.1 Storage of constants and variables

10.2.1.1 Integer

An integer constant or variable occupies four bytes. An integer value is held in twos complement form, and may range from -2**(31) to +2**(31-1), that is, from -2147483648 to +2147483647.

Some examples of valid integer constants are:

0	5678	2147483647
-2147483648	-255	-0
+0	+5678	+2147483647

10.2.1.2 Real

A real constant or variable occupies four bytes. A real value is held as a normalized floating point number in accordance with the IEEE floating point format (see An Implementation Guide to a Proposed Standard for Floating Point, Computer, January 1980), and may range from +2**(-126) to approximately +2**(+128), that is, approximately, from +1.1754945x10**(-38) to +3.402823x10**(+38).

Some examples of valid real constants are:

1.23	-1.23	+1.23
0.	.0001234	667744.
1.23E2	+1.23E2	-1.23E2
123456.E5	1.23E30	1.23E+33
1.23E+3	0.E0	1.23E0
1.23E+03		
-1.23E30	-1.23E10	+123E10
123456789E-34	123E-10	0E0

Also see section 2.2.1.2.

10.2.1.3 Double precision

A double precision constant or variable occupies eight bytes. A double precision value is held as a normalized floating point number in accordance with the IEEE floating point format and may range from +2**(-1022) to +2**(1024), that is, approximately from +2.225073858507200x10**(-308) to +1.797693134862315x10**(308).

10.2.1.4 Complex

A complex number consists of a real part and an imaginary part. (The word real, in the term real part, is not used in the sense of section 10.2.1.2.)

A complex constant, or variable, consists of either a pair of real (in the sense of section 10.2.1.2) constants, or variables, or a pair of integer constants, or variables; the first of the pair is the real part and the second is the imaginary part.

Some examples of valid complex constants are:

```
(3.75,-2.100)
(0.,0.)
(-2.75E+2,7.1E-2)
```

10.2.1.5 Logical

A logical constant or variable occupies 4 bytes.

10.2.1.6 Character

Each character in the character constant or variable occupies one byte.

A character constant or variable may comprise from 1 to 32767 characters.

10.2.2 Input/output record lengths

The maximum length of formated and unformatted records is (32K-5) bytes.

When RECL is specified in an OPEN or INQUIRE statement, the record length must be specified in bytes.

10.2.3 Code and data limitations

Item	Limit
Character arrays	128 Kbyte
Non-character arrays	See note
Character COMMON blocks	128 Kbyte
Non-character common blocks	See note
Code per compilation unit	64 Kbyte
Code per program unit	32 Kbyte
Maximum number of COMMON blocks	256

Note: The maximum size of non-character arrays and non-character COMMON blocks is determined by system limitations, usually the space available for on disk swap files. If COMMON blocks or arrays greater than a segment (128 Kbytes) are used, double precision items must be aligned so that they do not cross a segment boundary.

10.2.4 Run time file access

10.2.4.1 File connections

The default unit input file is pre-connected to the keyboard and the default unit output file is pre-connected to the display before the FORTRAN code is entered.

All other file connections are established after the FORTRAN code is entered:

1 By the OPEN statement (see section 9.7.2)

2 By a run time routine if no OPEN statement is executed before the unit is accessed. You are prompted for the file name and properties on the display

When specifying the filename it is recommended that you give the full name including any extension.

Note that the filename in the OPEN statement may be a string variable and thus the actual file may be chosen dynamically if required.

10.2.4.2 File types

Files used only for formatted sequential access may be compatible with Pascal and FORTRAN source files, and hence with PERQ Operating System utilties such as the editor. In this case no special file extension is necessary.

If a formatted sequential file is to be printed (that is, the first character in each record is to be interpreted as a FORTRAN format control character), it must have the extension .PRI (see section B1.1).

Files used for direct access or unformatted sequential access are held in a format only accessible through a FORTRAN program. These files are allocated a special file type in the PERQ Operating System with extension .DTA, and the first block of each such file is used to hold FORTRAN related administration information.

Note that .DTA files may also be used for formatted sequential access in order that a single file may be accessed in different modes at different times.

10.2.4.3 Console input/output

The console may be used directly for formatted input or output by giving a filename of CONSOLE: in the OPEN statement or in response to the prompt for a name. When CONSOLE: is used for output, the first character of the line is treated as a format control character. If this character is new page, the user is prompted before the screen is cleared.

10.2.5 Opening files

When a file is opened the FORTRAN system positions at the first record.

10.2.6 Length of program unit names

Program unit names (that is, the names of main programs, subroutines, and fuctions) should be restricted to 8 characters.

10.2.7 Block data

A block data subprogram, if required, must be included in or before the first compilation unit entered at run-time which contains a

reference to any common blocks declared in the block data.

A block data subprogram must be compiled in a unit containing at least one executable program unit.

10.2.8 STOP and PAUSE

Execution of a FORTRAN STOP or PAUSE statement causes a message to be output to the screen and the program to halt. Following a PAUSE statement the user may resume the program by typing CONTROL Q.

CHAPTER ELEVEN

MIXED LANGUAGE PROGRAMMING

11.1 Introduction

A PERQ FORTRAN 77 program can reference Pascal procedures and functions provided that:

1 The names of the procedures and functions are declared in a special form of the EXTERNAL statement (see section 10.1.4)

2 The names of the files containing the compiled Pascal procedures and functions are supplied by use of the /EXTERNAL switch and its associated file (see sections 12.1 and 12.3.2.6)

It is not possible to access directly any data exported by a Pascal module. If such access is essential, a Pascal interface routine must be written.

11.2 Referencing Pascal from FORTRAN 77

To reference a Pascal function or procedure from a FORTRAN program unit, it is necessary to ensure that the arguments and, in the case of functions, function results correspond to suitable items in the program. In FORTRAN terms the correspondence must be in terms of type and actual length.

Valid correspondences for arguments are given first in section 11.2.1 while correspondences for function results follow in section 11.2.2. Invalid correspondence of arguments or results produces unpredictable results.

Note that it is not possible to reference FORTRAN from Pascal.

11.2.1 Valid argument correspondence

FORTRAN	Pascal
INTEGER	Long integer
REAL	Real
CHARACTER*n	String of length n
INTEGER arrays	Long integer arrays
REAL arrays	Real arrays
INTEGER*2	Integer
INTEGER*2 arrays	Integer arrays

Only one-dimensional arrays can be passed since FORTRAN and Pascal store elements in different orders.

Function and subroutine names cannot be passed as arguments.

Arguments are passed by reference, thus if their values are changed by the Pascal routine the FORTRAN argument is similarly changed. This also implies that arguments in Pascal routines called from FORTRAN must be declared as var.

11.2.2 Valid function result correspondence

FORTRAN	Pascal
INTEGER	Long integer
REAL	Real
INTEGER*2	Integer

11.3 Pascal input and output

The default Pascal input and output channels may not be used in Pascal routines called from FORTRAN since their use requires a Pascal main program to be present.

CHAPTER TWELVE

RUNNING A PERQ FORTRAN 77 PROGRAM

This chapter explains how a PERQ FORTRAN 77 source program is made into a runnable program and describes the utilities required in this process.

12.1 FORTRAN Programming

The FORTRAN compiler, given a source file, produces a segment (.SEG) file. Segment files for programs can be linked to produce a run (.RUN) file. The program is invoked by giving the name of the run file.

Since FORTRAN supports separate compilation, references may be made to routines residing in other files. Where Pascal provides a mechanism for specifying information about externally defined routines at compile time, FORTRAN does not. A different method of resolving references must be used in FORTRAN. There are two ways of doing this:

1. At compile time, the /EXTERNAL switch can be used. Basically, a list of existing segment files is specified. These segment files can be Pascal or FORTRAN generated. They may contain unreferenced routines as well, but they may not collectively define a routine more than once. This method is the ONLY way to resolve references to Pascal routines. Note that no distinction is made between exported and private Pascal routines; all of these are available. (See sections 10.1.4 and 11.1 for a further discussion of Pascal routines.)

   If existing segment files are used at compile time, all references can be resolved in this manner and no further action is necessary. The resultant segment file will be complete and ready as input to the linker.

   It is possible to arrange FORTRAN routines and references in a way that prohibits some necessary segment files from existing at compile time. If the FORTRAN compiler cannot resolve all routine references at compile time, it will produce a pre-segment (.PSG) file instead of a segment file. In this case, the second method (described below) must be used in place of or in addition to the /EXTERNAL switch.

2. After compile time, the Consolidate utility can be used. Basically, a list of segment and pre-segment files is specified. These files contain externally referenced

routines not specified with the /EXTERNAL switch at compile time. The collection of routines defined in the files must satisfy remaining references in all pre-segment files specified. They may contain unreferenced routines as well, but they may not collectively define a routine more than once. All files must be FORTRAN generated. References to Pascal routines must be resolved with the /EXTERNAL switch at compile time. Consolidate produces one complete segment file for each pre-segment file. The segment files are ready for input to the linker.

A combination of the above methods may be used--none, one, or both may be necessary.

Note that when a FORTRAN unit is recompiled it must be reconsolidated with the .PSG (not .SEG) files for all units that reference the recompiled unit; otherwise errors may occur. .PSG files exist unless they have been deleted either explicitly or by use of the /NOSAVE switch during a previous consolidation.

Figure 12.1 illustrates the process for the program Calc.FOR, assuming that it does not explicitly or implicitly reference any Pascal module or any other independently compiled FORTRAN section. This means that the compiler can resolve all the references and can therefore produce a segment file directly.

Figure 12.2 illustrates the conversion of Plot.FOR, assuming that it references the Pascal module OtherP, which must already have been compiled. OtherP could also be a FORTRAN generated segment file.

Figure 12.3 illustrates the conversion of Stress.FOR, assuming that it references the independently compiled FORTRAN unit OtherF. When OtherF.FOR is itself compiled it will yield OtherF.SEG (or OtherF.PSG if it too contains unresolved references). When consolidating Stress.PSG, OtherF.FOR must have been compiled.

12.2 Utilities

The utilities used to prepare and run a PERQ FORTRAN 77 program are:

1 EDITOR This is used initially to input, and later to amend, the source program. The EDITOR is described in the PERQ Editor User's Guide.

2 FORTRAN This is the compiler. It takes a FORTRAN source program (.FOR file) and produces a pre-segment or segment file.

3 CONSOLIDATE This takes one or more pre-segment and segment

files and produces one or more .SEG files.

4 LINK This takes the necessary segment file (.SEG files) and produces the run file (.RUN file) which, together with the segment files, enables the program to be loaded.

5 RUN/RERUN Once a program has been successfully linked, it can be run in several ways, including by use of the RUN or RERUN utilities (see section 12.5)

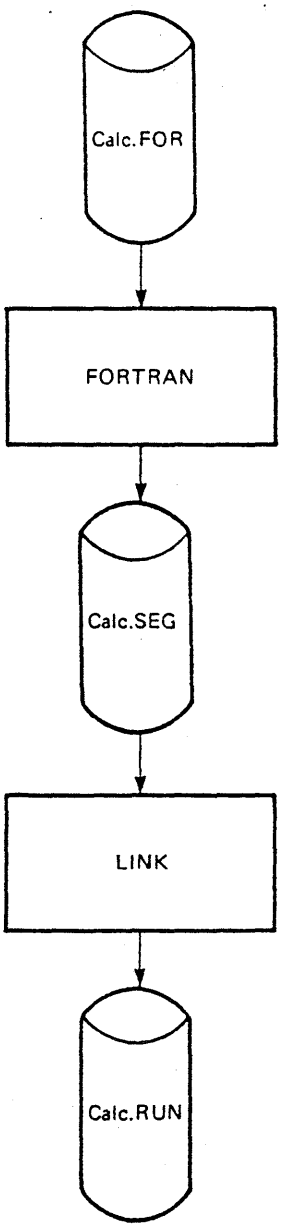

Figure 12.1 A Self-Contained FORTRAN Program

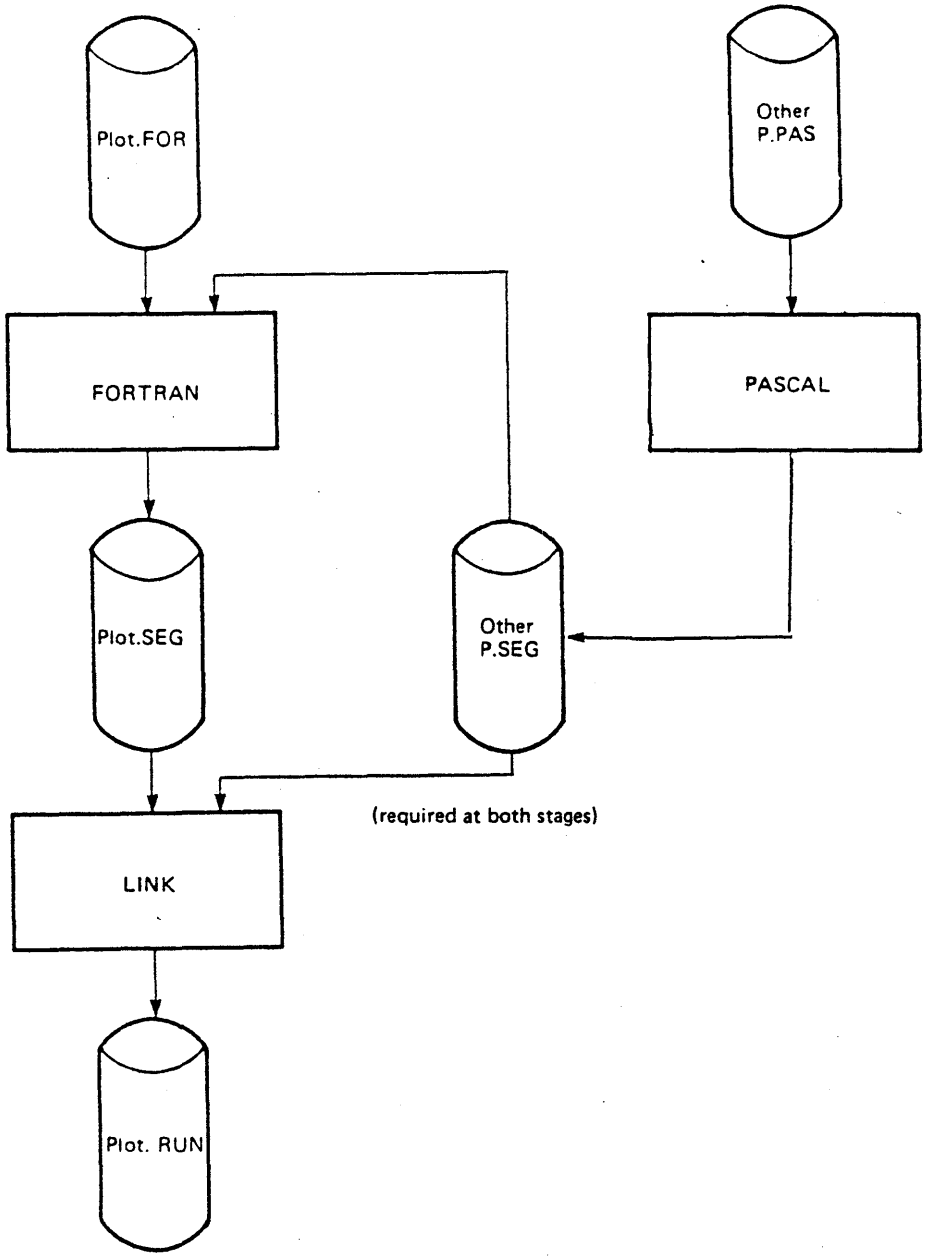

Figure 12.2 A FORTRAN Program Which References a Pascal Module

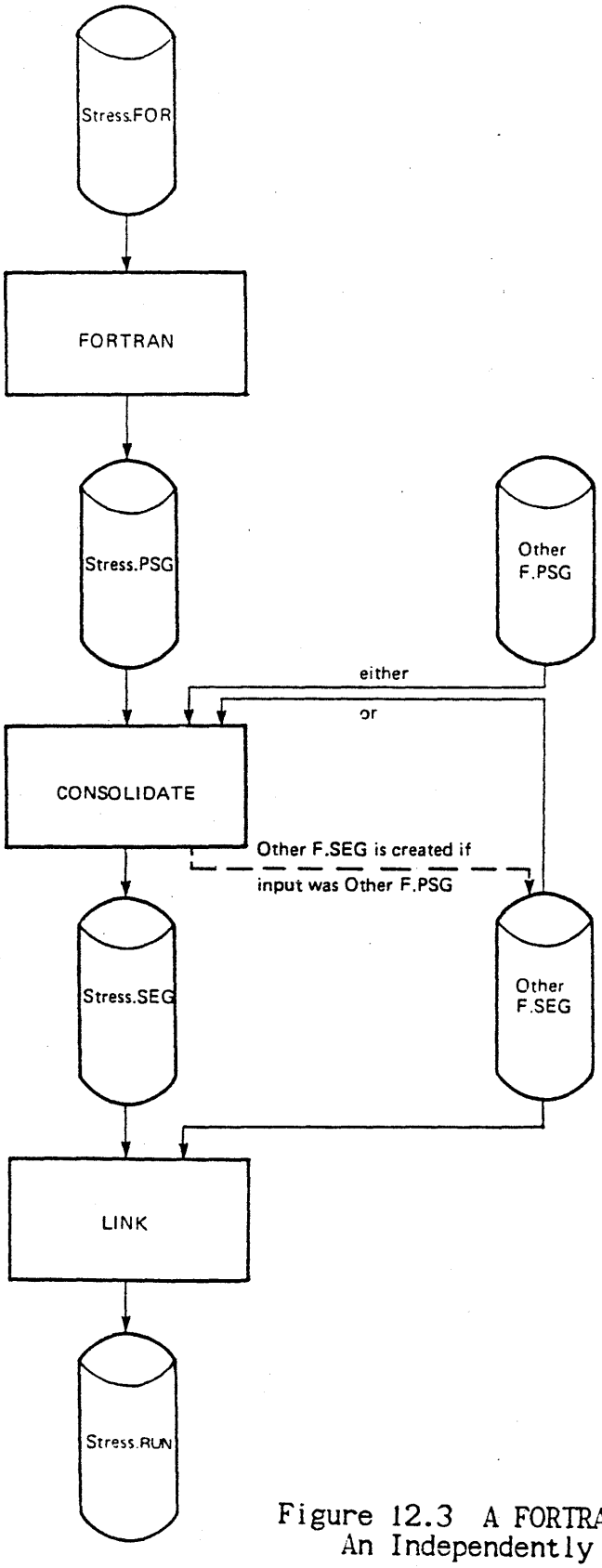

Figure 12.3 A FORTRAN Program Which Referenc
An Independently Compiled FORTRAN Unit

12.3 Compilation

The utility FORTRAN compiles a FORTRAN 77 source program (.FOR file) into either a pre-segment file (.PSG file) from which it can be consolidated, or into a segment file (.SEG file).

12.3.1 Using the compiler

The compiler is run by issuing the command line:

FORTRAN[<InFile>][<OutFile>]{<switch>}

where

<InFile> is the source file to be compiled. If InFile is absent, the current filename is used. If the file does not exist, .FOR is appended to the filename and the search is repeated.

<OutFile> is the name of the file to contain the object code. The extension .PSG or .SEG is appended as required. If <OutFile> is absent, the compiler uses the source filename. If at the start of the compilation the file already exists, it is deleted (whether it is a .PSG or a .SEG file). If any errors are reported during the compilation (as distinct from comments or warnings) no pre-segment or segment file is produced.

Any number of switches may follow, as defined in section 12.3.2.

Example 1

```
EDIT Stress.FOR
FORTRAN
```

This compiles Stress.FOR into Stress.PSG

Example 2

```
FORTRAN Calc.FOR/LIST/CROSSREFERENCE/SHOWCODE
```

This compiles Calc.FOR into Calc.SEG and produces a listing, with cross references and object code, in the file Calc.LST.

Example 3

```
FORTRAN/HELP
```

This presents general information on the use of the compiler, and takes no other actions.

12.3.2 Compiler switches

12.3.2.1 Listings (/LIST, /CROSSREFERENCE, /SHOWCODE)

By default, the compiler does not produce a listing. This saves compilation time, and furthermore, a simple listing is very similar to the source program, which can easily be examined instead by using EDITOR or the TYPEFILE utility. One advantage of a listing is that the source program lines are numbered. A listing is obtained with the switch:

/LIST[=<filename>]

If <filename> is present, the listing is sent to a file with name <filename>.LST. If =<filename> is absent, the listing is sent to a file of the same name as the source program, but with .LST as the extension.

If a /LIST switch is present, two other listing options can be switched on:

1 Addition of the /CROSSREFERENCE switch causes the generation of a cross-reference and attribute listing, which records the names of all identifiers in the program, their properties, and the line numbers at which they are referenced

2 Addition of the /SHOWCODE switch causes the generation of an object code listing.

3 Addition of the /MAP swtich causes the generation of a listing giving information about the object program structure.

If /CROSS REFERENCE, /SHOWCODE or /MAP is used without /LIST, a warning is given and the listing is directed to the default list file.

12.3.2.2 Diagnostic options (/NORANGE, /NOCHECK)

By default, the compiler will generate code to check that, at run time, each array subscript and character substring position is within range. The generation of such checks may be inhibited by use of the /NORANGE switch.

By default, the compiler will generate code to check that variables are

not accessed before being assigned a value. The generation of such a check may be inhibited by use of the /NOCHECK switch.

12.3.2.3 Program scan (/SYNTAXCHECK)

By default, the compiler produces a pre-segment or segment file (.PSG or .SEG file). The use of the switch /SYNTAXCHECK suppresses the production of the file, but the source program is still checked for syntactic and semantic validity. This saves compilation time when errors are expected, for example, initial compilations of large programs.

12.3.2.4 Quiet (/QUIET)

By default, the compiler displays the name of each section as the compilation proceeds. These displays may be inhibited by use of the /QUIET switch.

12.3.2.5 Help (/HELP)

Inclusion of the /HELP switch causes any other information on the command line to be ignored; instead general information on the use of the compiler is presented.

12.3.2.6 Using Pascal or FORTRAN externally defined routines (/EXTERNAL)

This switch specifies a file which contains a list of filenames (one per line) of segment files required by the compilation (see section 12.1). The switch must be supplied if the FORTRAN program references any Pascal modules. The format of the switch is:

```
/EXTERNAL=[<filename>]
```

For example, if Plot.FOR references an item in the module OtherP, the switch /EXTERNAL=Plot.EXT or the switch /EXTERNAL could be supplied, where Plot.EXT contains the line:

```
OtherP.SEG
```

or just:

```
OtherP
```

since the compiler automatically appends .SEG if it is missing. All the

.SEG files mentioned in the list must exist.

12.3.2.7 Initialization of data to zero (/ZERO)

This switch causes the initialization of all data to zero prior to execution of the program. It may only be used when checks are not required and hence the /NOCHECK switch is also used (see section 12.3.2.2). Its use is not generally recommended, as programs relying on data initialization are not portable under the ANSI77 standard.

12.3.2.8 File inclusion

The compiler may be directed to include the contents of secondary source files in the compilation. The effect of using the file inclusion mechanism is identical to having the text of the secondary file or files present in the file being compiled.

The directive to include a file is a special form of FORTRAN comment line:

```
%$INCLUDE filename
```

where %$INCLUDE must start in column 1 and must not contain any spaces.

If the specified filename does not exist, .FOR will be appended and the search repeated.

12.3.2.9 Argument mismatch (/MISMATCH)

In standard FORTRAN77 the type of dummy and actual arguments must match. This is checked:

1) at compile time, if the reference is in the same compilation unit as the reference;

2) during consolidation, if the referenced procedure is not in the same compilation unit as the reference.

These checks may be inhibited in PERQ FORTRAN 77 for 4 byte data items of type integer, real and logical by use of the /MISMATCH switch. Use of /MISMATCH causes the type to be taken from the dummy argument. A /MISMATCH switch is provided for both the compiler and consolidator.

12.3.2.10 Error file (/ERRORFILE)

By default when the compiler detects an error, a message and, if appropriate, the line in error are displayed on the screen. However, if the /ERRORFILE switch is specified the information is instead written to a file. The format of the switch is:

/ERRORFILE [=<filename>]

where <filename> is the name of the file to be written. The extension .ERR is appended to the name if not already present. If <filename> is omitted, the listing is sent to a file of the same name as the source program with the extension of .ERR.

If the error file already exists, it is overwritten by the new file.

12.3.3 Compile time diagnostics

The following categories of message are produced compile time:

1 COMMENTS For example, constant has too great a precision and has been truncated

2 WARNINGS For example, use of a non-standard feature

3 ERRORS For example, syntax error

The messages are output to the listing file, if used (see section 12.3.2.1), following the line which causes the error at the end of that file if the error is detected after the first pass of the source.

Error messages only are also output to the display, preceded by the relevant source line, if appropriate.

At the end of the compilation, a summary of the number of errors, warnings, and comments is output to the display and to the listing file, if used.

12.4 Consolidation

The utility CONSOLIDATE converts one or more pre-segment files (.PSG files) into one or more segment files (.SEG files).

12.4.1 Using the consolidator

The consolidator is run by issuing the command line:

```
CONSOLIDATE <InFile1>,<InFile2>{,<InFile>}
   [/LIBRARY=<filename>][/NOSAVE][/MISMATCH]
```

where

<InFile1> is the name of a pre-segment file for consolidation, or of a segment file containing information required to resolve references in the pre-segment files. You are recommended to supply filenames without extensions, in which case the consolidator initially appends .PSG to each filename, and changes this to .SEG if it cannot find such a file.

The library switch must specify a filename (a file which contains a list of additional files, one per line, to be used as input to the consolidation). The format of the names is as defined for <Infile> and the effect is equivalent to all the library files being specified as input files to CONSOLIDATE. A library file must have the extension .LIB although this may be omitted when specifying <filename>.

A successful run of CONSOLIDATE converts all .PSG files to .SEG files. By default the .PSG files will remain after consolidation; they may be deleted by the use of the /NOSAVE switch. As the .PSG files may be required if it is necessary to reconsolidate, the use of the /NOSAVE switch is not normally recommended.

The /MISMATCH switch inhibits argument checks during consolidation (see section 12.3.2.9).

Examples

Consider again the example illustrated in Figure 12.3. The command to the consolidator should be:

```
CONSOLIDATE Stress,OtherF
```

If OtherF could only be compiled into a .PSG file (because it references sections in Stress), the consolidator would read Stress.PSG and OtherF.PSG and output Stress.SEG and OtherF.SEG.

If OtherF could be compiled directly into a .SEG file, the consolidator would read Stress.PSG and OtherF.SEG, and output Stress.SEG.

12.4.2 Partial consolidation

It is not essential to consolidate all parts of a program at the same time. For example, if you are writing a program which will comprise A.SEG, B.SEG, and C.SEG, in which A references B, which references C, and C has no external references, you can either:

```
CONSOLIDATE A,B,C
```

or

```
CONSOLIDATE B,C /NOSAVE
```

which converts B.PSG into B.SEG while reading C.SEG, then deletes B.PSG, and later:

```
CONSOLIDATE A,B
```

which converts A.PSG into A.SEG, while reading B.SEG.

Partial consolidation can be useful in the development of large programs when a set of routines form a self-contained unit. However, it is recommended that the .PSG files are only deleted (as in the example) when the unit is well tested, as they may be required if it is necessary to repeat the partial consolidation.

12.5 Linking

Running a single program often requires pieces of several segment files (.SEG files). It is the job of the LINK utility to produce a run file (.RUN) containing information about the segment files.

To produce the run file, LINK must read the segment file for a main program. Usually, the LINK utility can be issued without an input file parameter, in which case it reads the segment file with the current filename, by default. Wherever LINK finds an external reference, it searches for the segment file referred to.

For example, if, as illustrated in Figure 12.3, you have just compiled and consolidated Stress.FOR (so that Stress is the current filename), the command:

```
LINK
```

or

LINK Stress

reads Stress.SEG, OtherF.SEG, and any other .SEG files which they reference, and generates Stress.RUN, containing the table. Note that references are often implicit, for example, FORTRAN input or output statements automatically reference PERQ Operating System procedures.

For a detailed description of LINK, see the PERQ Utility Programs manual.

12.6 Running the program

A program can be run in one of three ways:

1 Issue its name as a command, for example:

Myprog

2 Issue the RUN command, with the program's name as the first parameter, for example:

RUN Myprog

The sole advantage of using RUN is that, by default, it uses the current filename as the name of the program to be run. Thus if you are repeatedly editing, compiling, consolidating and running one program during its testing phase, you need only mention its name once if you use RUN

3 Issue the command RERUN, with new parameters, after a previous run, either of type 1 or 2 above, for example

RERUN para1,para2

Details of RUN and RERUN are given in the PERQ Utility Programs manual.

12.6.1 Run time diagnostics

The following categories of error can occur at run time:

1 HARDWARE DETECTED For example, floating point overflow

2 MATHEMATICAL LIBRARY

3 INPUT/OUTPUT

4 SOFTWARE DETECTED (see section 12.3.2.2)

On detection of an error, an error message is output, execution is halted and the FORTRAN 77 debugger is entered. This outputs a trace of each currently active FORTRAN or PERQ Operating System procedure, and the byte offset within the procedure.

You may then select more detailed diagnostics by typing one of the following and pressing RETURN:

1	?	This provides help on the debugger.
2	f	This requests that the diagnostics are output to the file instead of to the display. You are prompted for the filename.
3	l	This outputs local scalars for active FORTRAN procedures.
4	c	This outputs local and common scalars for all active FORTRAN procedures.
5	a	This outputs local and common scalars and arrays for all active FORTRAN procedures. You are prompted for the maximum number of array elements to be output. You may respond either with a number, or by typing a (or all) to output every element.
6	q	This quits the diagnostics.

After selecting any of the options 1 to 5 you may make another selection.

All the diagnostics are expressed in source language terms.

12.7 Setting up the software

PERQ FORTRAN 77 is supplied on one or more floppy disks, with details of how to set up the software.

APPENDIX A

INTRINSIC FUNCTIONS

Instrinsic function	Definition	Generic Name	Specific Name	No. of Arguments	Type of Argument	Type of Function
Type conversion	Conversion to integer	INT	INT	1	Real	Integer
			IFIX	1	Real	Integer
			IDINT	1	Double	Integer
				1	Complex	Integer
	Conversion to real	REAL	REAL	1	Integer	Real
			FLOAT	1	Integer	Real
				1	Real	Real
			SNGL	1	Double	Real
				1	Complex	Real
	Convert to double	DBLE		1	Integer	Double
				1	Real	Double
				1	Double	Double
				1	Complex	Double
	Conversion to complex	CMPLX		1 or 2	Integer	Complex
				1 or 2	Real	Complex
				1 or 2	Double	Complex
				1 or 2	Complex	Complex
	Conversion to character		CHAR	1	Integer	Character
	Conversion to integer		ICHAR	1	Character	Integer
Truncation	See Note 1	AINT	AINT	1	Real	Real
			DINT	1	Double	Double

Nearest whole number	See Note 2	ANINT	ANINT	1	Real	Real
			DNINT	1	Double	Double
Nearest integer	See Note 2	NINT	NINT	1	Real	Integer
			IDNINT	1	Double	Integer
Absolute value	x	ABS	IABS	1	Integer	Integer
			ABS	1	Real	Real
			DABS	1	Double	Double
			CABS	1	Complex	Real
Remainder	See Note 3	MOD	MOD	2	Integer	Integer
			AMOD	2	Real	Real
			DMOD	2	Double	Double
Transfer of sign	See Note 4	SIGN	ISIGN	2	Integer	Integer
			SIGN	2	Real	Real
			DSIGN	2	Double	Double
Positive difference	See Note 5	DIM	IDIM	2	Integer	Integer
			DIM	2	Real	Real
			DDIM	2	Double	Double
Double length product	x1 * x2		DPROD	2	Real	Double
Largest value	max(x1,x2,...)	MAX	MAXO	>=2	Integer	Integer
			AMAX1	>=2	Real	Real
			DMAX1	>=2	Double	Double
			AMAXO	>=2	Integer	Real
			MAX1	>=1	Real	Integer

Smallest value	min(x1,x2,...)	MIN	MINO	>=2	Integer	Integer
			AMIN1	>=2	Real	Real
			DMIN1	>=2	Double	Double
			AMINO	>=2	Integer	Real
			MIN1	>=2	Real	Integer
Length	Length of character entity		LEN	1	Character	Integer
Index of a substring	Location of substring a2 in string a1		INDEX	2	Character	Integer
Imaginary part of complex argument			AIMAG	1	Complex	Real
Conjugate of a complex argument	(x - iy)		CONJG	1	Complex	Complex
Square root	x	SQRT	SQRT	1	Real	Real
			DSQRT	1	Double	Double
			CSQRT	1	Complex	Complex
Exponential	ex	EXP	EXP	1	Real	Real
			DEXP	1	Double	Double
			CEXP	1	Complex	Complex

Natural Logarithm	loge(x)	LOG	ALOG	1	Real	Real
			DLOG	1	Double	Double
			CLOG	1	Complex	Complex
Common Logarithm	log10(x)	LOG10	ALOG10	1	Real	Real
			DLOG10	1	Double	Double
Cosine	cos(x)	COS	COS	1	Real	Real
			DCOS	1	Double	Double
			CCOS	1	Complex	Complex
Sine	sin(x)	SIN	SIN	1	Real	Real
			DSIN	1	Double	Double
			CSIN	1	Complex	Complex
Tangent	tan(x)	TAN	TAN	1	Real	Real
			DTAN	1	Double	Double
Arccosine	arcos(x)	ACOS	ACOS	1	Real	Real
			DACOS	1	Double	Double
Arcsine	arcsin(x)	ASIN	ASIN	1	Real	Real
			DASIN	1	Double	Double
Arctangent	arctan(x)	ATAN	ATAN	1	Real	Real
			DATAN	1	Double	Double
	arctan(x1/x2)	ATAN2	ATAN2	2	Real	Real
			DATAN2	2	Double	Double
Hyperbolic cosine	cosh(x)	COSH	COSH	1	Real	Real
			DCOSH	1	Double	Double

Hyperbolic sine	sinh(x)	SINH	SINH DSINH	1 1	Real Double	Real Double
Hyperbolic tangent	tanh(x)	TANH	TANH DTANH	1 1	Real Double	Real Double
Lexically greater than or equal	See Note 6		LGE	2	Character	Logical
Lexically greater than	See Note 6		LGT	2	Character	Logical
Lexically less than or equal	See Note 6		LLE	2	Character	Logical
Lexically less than	See Note 6		LLT	2	Character	Logical

Notes:

1 int(x)

2 int(x + 0.5) if x > 0

int(x - 0.5) if x < 0

3 x1=int(x1/x2)*x2

4 |x1| if x2 > 0

-|x1| if x2 < 0

5 x1 - x2 if x1 > x2

0 if x1 < x2

6 The value .TRUE. is returned if the condition is satisfied according to the ASCII collating sequence, otherwise the value .FALSE. is returned. If the operands are of unequal length the shorter operand is considered as if it were extended on the right with blanks to the length of the longer operand.

APPENDIX B

PERQ OPERATING SYSTEM AND FORTRAN 77 RELEASES

All the facilities described in this publication are available in the PERQ operating system versions G.3 and later, with version 02.1 of PERQ FORTRAN 77.

B1.1 Incompatible Changes Between Version 01 and Version 02.1 of PERQ FORTRAN 77

B1.1.1 File types (see section 10.2.4.2)

In version 01 of PERQ FORTRAN 77, formatted sequential files not explicitly given a .DTA extension were assumed to contain a FORTRAN format control character as the first character in the record. In version 02.1, this is only applicable to .PRI files. Therefore any formatted sequential files created under version 01 must be renamed to include a .PRI extension if they are to be read under version 02.1

B1.1.2 Pause (see section 10.2.8)

In Version 02.1, Pause causes the program to halt until the user types CONTROL Q, unlike version 01 in which the program continued after the message had been output to the screen.

B1.2 Existing FORTRAN programs

It is recommended that any existing FORTRAN programs be recompiled under version 02.1 to take advantage of the performance improvements offered by this release.

APPENDIX C

FORTRAN 77 INPUT/OUTPUT ERRORS

Error number	Error text	Explanation	Statement type
118	File already connected	An attempt was made to OPEN a file on one unit while it was still connected to another	Open
119	ACCESS conflict	When a file is to be connected to a unit to which it is already connected, then only the BLANK specifier may be re-defined. An attempt has been made to re-define the ACCESS specifier. This message is also used if an attempt is made to use a direct-access I/O statement on a unit which is connected for sequential I/O or a sequential I/O statement on a unit connected for direct-access I/O.	Open Positional Read Write
120	RECL conflict	When a file is to be connected to a unit to which it is already connected, then only the BLANK specifier may be re-defined. An attempt has been made to re-define the RECL specifier.	Open
121	FORM conflict	When a file is to be connected to a unit to which it is already connected, then only the BLANK specifier may be re-defined. An attempt has been made to re-define the FORM specifier.	Open
122	STATUS conflict	When a file is to be connected to a unit to which it is already connected, then only the BLANK specifier may be re-defined. An attempt has been made to re-define the STATUS specifier.	Open
123	Invalid STATUS	STATUS=KEEP has been specified in	Close

		a CLOSE statement for a unit which is connected to a scratch file.	
124	FORM not suitable	Either a file with unformatted records has been opened for formatted I/O or a file with formatted records has been opened for unformatted I/O.	Open
125	Specifier not recognized	A specifier value defined by the user has not been recognized.	Open
126	Specifiers inconsistent	Within an OPEN statement one of the following invalid combinations of specifiers was defined by the user: 1 FILENAME= was specified when STATUS=SCRATCH 2 RECL= was specified when ACCESS=SEQUENTIAL. 3 BLANK= was specified when FORM=UNFORMATTED.	Open
127	Illegal specifier value	The value of the RECL specifier was not a positive integer.	Open
128	Invalid filename	The name of the file in an Inquire by file statement is not a valid filename.	Inquire
129	No filename specified	In an OPEN statement, the STATUS specifier was not SCRATCH and no filename was defined	Open
130	Record length not specified	The RECL specifier was not defined although ACCESS=DIRECT was specified.	Open
132	Value separator missing	A complex or literal constant in the input stream was not terminated by a delimiter (that is, by a space, a comma or a record boundary).	List-Directed Read
133	No digits specified	The current input field contained a real number in which either the exponent part or the fixed point part contained no significant digits.	Read with format List-Directed Read

134	Invalid scaling	If d represents the decimal field of a format description and k represents the current scale factor, then the ANSI Standard requires that the relationship -d< k<d+2 is true when an E or D format is used with a WRITE statement. This requirement has been violated.	Write with format
135	Invalid logical value	A logical value in the input stream was syntactically incorrect.	List-Directed Read
136	Invalid character value	A literal constant with the value ; was found in the input stream; this is prohibited by the ANSI Standard. This message is also used if a character constant did not begin with a quote.	List-Directed Read
137	Value not recognized	An item in the input stream was not recognized.	List-Directed Read
138	Invalid repetition value	The value of a repetition factor found in the input stram is not a positive integer constant.	List-Directed Read
139	Illegal repetition factor	A repetition factor in the input stream was immediately followed by another repetition factor.	List-Directed Read
140	Invalid integer	The current input field contained a real number when an integer was expected.	Read with format List-Directed Read
141	Invalid real	The current input field contained a real number which was syntactically incorrect.	Read with format List-Directed Read
143	Invalid complex constant	The current input field contained a complex number which was syntactically incorrect.	List-Directed Read
148	Invalid character	A character has been found in the current input stream which cannot syntactically be part of the entity	Read with format

		being assembled.	
150	Literal not terminated	A literal constant in the input file was not terminated by a closing quote before the end of the file.	List-Directed Read
151	Channel not defined	An I/O request was made on a unit for which no definition has been supplied.	Read Write Positional
152	File does not exist	An attempt has been made to open a file which does not exist with STATUS=OLD.	Open
153	Input file ended	All the data in the associated internal or external file has been read.	Read
154	Wrong length record	The record length as defined by a FORMAT statement, or implied by an unformatted READ or WRITE, exceeds the defined maximum for the current input or output file.	Read Write
155	Incompatible format descriptor	A format description was found to be incompatible with the corresponding item in the I/O list.	Read with format Write with format
156	Read after Write	An attempt has been made to read a record from a sequential file after a WRITE statement.	Read
157	Write after Endfile	An attempt has been made to write a record to a sequential file after an ENDFILE statement.	Write
158	Record number out of range	The record number in a direct-access I/O statement is not a positive value or is greater than 32767.	Direct-Access Read Direct-Access Write
159	No format descriptor for data item	No corresponding format code exists in a FORMAT statement for an item in the I/O list of a READ or WRITE statement.	Read with format Write with format

160	Read after Endfile	An attempt has been made to read a record from a sequential file which is positioned at ENDFILE.	Read
164	Invalid channel number	The unit specified in an Auxiliary I/O statement is a negative value.	Auxiliary
168	File already exists	An attempt has been made to OPEN an existing file with STATUS=NEW.	Open
169	Output file capacity exceeded	An attempt has been made to write to an internal or external file beyond its maximum capacity.	Read Write
171	Invalid operation on file	An I/O request was not consistent with the file definition; for example, attempting a BACKSPACE on a unit that is connected to the screen.	Positional Read Write
184	Format text too large	An array or character variable which is longer than 2048 characters has been specified as a run-time format.	Read with run-time format Write with run-time format
188	RECL too large	The value of the RECL specifier in an OPEN statement is greater than the record length of the direct access file opened or is greater than the maximum allowed record length.	Open
190	File not suitable	Either a file has been opened for direct-access but it does not contain fixed length records; or an attempt has been made to OPEN a file for either unformatted or direct-access I/O but the specified filename suffix is not .DTA.	Open
191	Workspace exhausted	Workspace for internal tables has been exhausted.	Open
192	Record too large	The record length of the file to be opened is greater than the maximum permitted size.	Open

193	Not connected for unformatted I/O	An attempt has been made to access a formatted file with an unformatted I/O statement.	Unformatted Read Unformatted Write
194	Not connected for formatted I/O	An attempt has been made to access an unformatted file with a formatted I/O statement.	Formatted Read Formatted Write
195	Backspace not allowed	An attempt was made to BACKSPACE a file which contains records written by a List-Directed output statement; this is prohibited by the ANSI Standard.	Backspace
197	System Open Failure	Either a failure occurred to physically open a nominated file, or the file to be opened was specified as a DTA type file but the internal file header was either found not to exist or has been corrupted.	Open
198	Constant not repeatable	This message applies only when the current unit is connected to the screen and when a repeated constant has been specified on more than one line. The message is given if the value of the repetition factor is greater than the number of elements associated with the current I/O item. This is an implementation restriction.	List-Directed Read
199	Field too large	An item in the input stream was found to be more than 1024 characters long (this does not apply to literal constants).	List-Directed Read

Other errors reported by I/O statements:

101 to 116	Errors in run-time formats
101	Missing left bracket
102	Missing right bracket
103	Negative sign incorrect

104	Invalid format
105	Decimal field too wide
106	Format width zero invalid
107	Repetition factor invalid
108	Null literal invalid
109	Integer field too large
110	No width field allowed
111	Literal in input format
112	Minimum digits too large
114	Non-repeatable edit disk
115	Comma required
116	Decimal point not allowed
401	Either unassigned variable in an output list or unassigned specifier in an I/O statement

Notes:

1. The I/O statements OPEN, CLOSE, and INQUIRE are classified as Auxiliary I/O statements.

   The I/O statements REWIND, ENDFILE, and BACKSPACE are classified as Positional I/O statements.

2. Any of the errors listed above will force a jump to the label given by the ERR= specifier (if defined).

   Any of the errors documented above will cause any defined IOSTAT variable to become defined with the corresponding error number.

INDEX

Accessing records	9.1.2
Actual arguments	7.1 7.4.2 10.1.2
correspondence with dummy arguments	7.3
reference	7.4.2.1
Adjustable dummy arrays	7.3.3.1
Alignment of variables in store	3.2.4.2
Allocation of storage	3.2
Arguments	7.1 7.4.2
actual	10.1.2
correspondence between dummy and actual functions and subroutines as	7.3 7.3.4
mismatch	12.3.2.9 12.4.1
valid correspondence between FORTRAN and Pascal	11.2.1
Arithmetic	
assignment statement	5.1
constant expressions	4.1.8
elements	4.1.1
expressions	4.1
IF statement	6.2.1
integer	4.1.7
operators	4.1.2

values 2.1

Array 2.2.4

and format specification 8.2

declarator 2.2.4.1
3.2.1.2
3.2.2

dimensions 2.2.4
10.2.3

elements 2.2.4.1
7.3.3

in EQUIVALENCE statement 3.2.4.1

names 2.2.4

storage 3.1.3
3.2.1.2

ASSIGN statement 6.1.3
10.1.6

Assigned GO TO statement 6.1.3

Assignment of initial values 3.3

Assignment statements Ch. 5

arithmetic 5.1

character 5.3

logical 5.2

BACKSPACE statement 9.3.2

Blank

common block 3.2.3.1

lines 1.3.1.3

Block

common 3.2.3
10.2.3

data subprogram 1.2

1.3.4.2
3.3.2
10.2.7

IF 6.2.3
6.2.3.1

CALL statement 7.2.2.1
10.1.2

Calling Pascal 10.1.4

Character

array limits 10.2.3

constants 2.2.1.6
3.3.1.3
10.2.1.6

elements 4.2.1

fields 8.3.1.7

length 10.1.5

storage 3.1.4
10.2.1.6

strings 8.3.1.8

substring 3.1.5

values 2.1
3.3.1.3
8.3.1.8

CHARACTER statement 2.3.3.2

CLOSE statement 9.7.3

Code and data limitations 10.2.3

Colon editing 8.3.1.13

Comment lines 1.3.1.3

Common block 3.2.3
3.2.3.2

	10.2.3
blank	3.2.3.1
correspondence of items	3.2.3.4
equivalencing of items	3.2.4.3
items	7.4.1
names	3.2.3
COMMON statement	3.2.3
Compilers	Ch. 1 12.3
Complex	
arrays	2.2.4
constants	2.2.1.4 10.2.1.4
constants, storage of	3.1.1 10.2.1
values	2.1 8.3.1.4
variables	2.2.3 10.2.1.4
Computed GO TO statement	6.1.2
Console input/output	10.2.4.3
Consolidation	12.2 12.4
Constants	2.2.1
arithmetic	2.2.1
character	2.2.1.6 3.3.1.3 10.2.1.6
complex	2.2.1.4

	10.2.1.4
double precision	2.2.1.3 10.2.1.3
Hollerith	10.1.2
integer	2.2.1.1 10.2.1.1
logical	2.2.1.5 10.2.1.5
real	2.2.1.2 10.2.1.2
storage of	3.1.1 10.2.1
symbolic	2.2.2
use of	7.3.1
Continuation lines	1.3.1.2
CONTINUE statement	6.4
Control statements	Ch. 6
Conversion code	8.3.1
Correspondence of common items	3.2.3.4
Data	Ch. 2
character	8.3.1.8
storage of	Ch. 3
DATA statement	3.3.1 10.1.2
Data values	2.1
initial	3.3 10.1.2
limitations	10.2.3

Declarator, array	2.2.4.1 3.2.1.2 3.2.2
Diagnostics	
compile time	12.3.3
options	12.3.2.2
run time	12.6.1
DIMENSION statement	3.2.2
Dimensioning of arrays	2.2.4 2.3.3 3.2.2
Direct access	9.1.2.2 9.4
formatted READ and WRITE statements	9.4.1.1
unformatted READ and WRITE statements	9.4.1.2
DO	
statement	10.1.6
variable	10.1.6
DO loops	6.3
implied	3.3.1.2 9.2.1.2
nested	6.3.3
terminal statement	6.3.2
transfer of control in	6.3.4
variable	6.3.1
DO statements	6.3.1
Double precision numbers	2.1 8.3.1.3 10.2.1.3

DOUBLE PRECISION statement	2.3.3.1
Dummy arguments	7.1 7.4.2
Edit descriptors	8.3 10.1.2
Elements	
arithmetic	4.1.1
array	10.1.5 10.1.6
character	4.2.1
logical	4.3.1
ELSE	6.2.3.2
ELSE IF	6.2.3.3
END statement	1.3.2.1
ENDFILE statement	9.3.2
END IF	6.2.3.4
ENTRY statement	7.5.1
referencing an	7.5.2
EQUIVALENCE statement	3.2.4
Equivalencing items	
in common block	3.2.4.3
of different type of length	3.2.4.2 10.1.5
Error file	12.3.2.10
Executable statements	1.3.4.1
Existing FORTRAN programs	App. B

Explicit type specification statement	2.3.3
Expressions	Ch.4
arithmetic	4.1
arithmetic constant	4.1.8
character	4.2
integer constant	4.1.9
logical	4.3 4.3.5
relational	4.3.2 4.3.5
use of	7.3.1
External	
functions	7.1.2.2 7.2.1.3
statement	7.3.4 10.1.4 11.1
subroutines	7.1.3.1
.FALSE. value	2.2.1.5 4.3.1
Field separators	8.1.1
File connections	10.2.4.1
File inclusions	12.3.2.8
File positioning statements	9.3.2
File types	10.2.4.2 App. B

Format codes	8.3.1
A conversion code	8.3.1.7
B format code	8.3.1.12
D conversion code	8.3.1.3
E conversion code	8.3.1.3
F conversion code	8.2.1.2
G conversion code	8.3.1.4
H format code	8.3.1.8
I conversion code	8.3.1.1
L conversion code	8.3.1.6
S format code	8.3.1.11
T format code	8.3.1.10
X format code	8.3.1.9
Format specifications	Ch. 8 10.1.2
FORMAT statement	8.2.1
Formatted	
direct access input and output	9.4.1.1 10.2.4.3
records	Ch. 8 9.1.1.2 10.2.2
sequential access input and output	9.3.1.1 10.2.4.2 App. B

FORTRAN	Ch. 1
basic elements	Ch. 1
program conversion	12.1
program units	1.2
routines	12.3.2.6
Function	
declaration statement	7.1.2.2
reference	7.2.1.1
result correspondence between FORTRAN and Pascal statement	10.1.6 11.2.2
Functions	7.1.1 7.1.2
external	7.1.2.2 7.2.1.3
generic	7.1.2.1 App. A
intrinsic	7.1.2.1
specific	7.1.2.1 App. A
statement	7.1.2.3
transfer of control between	7.2.1
used as arguments	7.3.4
Generic function names	7.1.2.1 App. A
GO TO statements	6.1
Group repeat count	8.1.2
Help	12.3.2.5
Hollerith constants	10.1.2

IF level	6.2.3
IF statements	6.2
Implementation characteristics	10.2
IMPLICIT statement	2.3.2 10.1.6
Implied DO loop	3.3.1.2 9.2.1.2 10.1.6
Incompatible changes between Versions 01 and 02	App. B
Incrementation parameter	6.3.1
Initial	
lines	1.3.1.1
parameter	6.3.1
values	3.3
Initialization of data to zero	12.3.2.7
Input data	
list directed	9.5.2
Input/output	Ch. 9
console	10.2.4.3
errors	App. C
list	9.2.1
list directed	9.5
Pascal	11.3
record lengths	10.2.2
statements	9.2
INQUIRE statement	9.7.4

Integer

*2 10.1.6

arithmetic 4.1.7

arrays 2.2.4

constants 2.2.1.1
10.2.1.1

constants, storage of 3.1.1
10.2.1

values 2.1
8.3.1.1
8.3.1.4

variables 2.2.3
10.2.1.1

Internal files 9.6

Intrinsic functions 7.1.2.1

INTRINSIC statement 7.3.5

Labels, statement 1.3.3

Language extensions 10.1

Length of data in store 3.1

Length of input/output records 10.2.2

Length of program unit names 10.2.6

Line printer 8.3.1.8
9.1.1.2

Lines, types of 1.3.1

Linking 12.2
12.5

List directed input 9.5

data 9.5.2

statement	9.5.1
List directed output	9.5
data	9.5.4
statements	9.5.3
Listings, compiler	12.3.2.1
Logical	
arrays	2.2.4
assignment statements	5.2
constants	2.2.1.5 10.2.1.5
constants, storage of	3.1.1 10.2.1
elements	4.3.1
expressions	4.3
IF statement	6.2.2
values	2.1 8.3.1.5 8.3.1.6
variables	2.2.3 10.2.1.5
Lower case	10.1.1
Mismatch, argument	12.3.2.9 12.4.1
Mixed language programming	Chap. 11
Multiple	
entry to a subprogram	7.5
references in a program unit	3.2.3.3

Names	1.4 10.1.1 10.1.3
Nested DO loops	6.3.3
Non-executable statements	1.3.4.2
Null data items	9.5.2
Omission of characters	8.3.1.9
OPEN statement	9.7.2 10.2.4.1 10.2.4.3
Opening files	10.2.5
Operators	
arithmetic	4.1.2
character	4.2.2
logical	4.3.3
relational	4.3.2
Order of evaluation	
arithmetic expressions	4.1.4
logical expressions	4.3.5
Order of statements and lines	1.3.5
Ouput data	
list directed	9.5.4
Output statements	9.2
list directed	9.5.3
PARAMETER statement	2.3.4
Parentheses	

arithmetic expressions	4.1.2
character expressions	4.2.2
logical expressions	4.3.3
Partial consolidation	12.4.2
Pascal	10.1.4 Chap. 11 12.1 12.3.2.6
PAUSE statement	6.6 10.2.8 App. B
Predefined specification	2.3.1
Print control character	8.3.1.8 9.1.1.2
PRINT statement	9.3.1.1
list directed	9.5.3.2
Procedures	7.1
Program conversion	12.1
Program scan	12.3.2.3
PROGRAM statement	7.2
Program units	1.2 Ch. 7
multiple references in	3.2.3.3
structure	1.3
transfer of control between	7.2
transfer of values between	7.4
Punched cards	9.1.1.2
Quiet	12.3.2.4

READ statement	9.2 9.3.1 9.4.1
formatted direct access	9.4.1.1
formatted sequential access	9.3.1.1
list directed	9.5.1
unformatted direct access	9.4.1.2
unformatted sequential access	9.3.1.2
Real	
arrays	2.2.4
constants	2.2.1.2 10.2.1.2
constants, storage of	3.1.1 10.2.1
values	2.1 8.3.1.2 8.3.1.3 8.3.1.4
variables	2.2.3 10.2.1.2
RECL	10.2.2
Record number	9.4.1
Referencing Pascal	11.2
Relational expressions	4.3.2 4.3.5
Repeat count	8.1.2
Rescanning	8.2.3
RETURN statement	
functions	7.2.1.2

subroutines	7.2.2.2
REWIND statement	9.3.2
Rounding errors	4.1.4
Run time	
diagnostics	12.6.1
file access	10.2.4
Running the program	12.6
SAVE statement	7.6
Scale factor	8.3.1.5
Sequential access	
formatted READ and WRITE statements	9.3.1.1
unformatted READ and WRITE statements	9.3.1.2
Slash editing	8.1.1.1
Specific function name	7.1.2.1 App. A
Specification, predefined	2.3.1
Standard functions	7.1.2.1 App. A
Statement	
functions	7.1.2.3
labels	1.3.3
Statements	
arithmetic IF	6.2.1
ASSIGN	6.1.3 10.1.6
assigned GO TO	6.1.3

Statements (cont.)

assignment Ch. 5

BACKSPACE 9.3.2

BLOCK DATA 3.3.2

block IF 6.2.3.1

CALL 7.2.2.1
10.1.2

categories of 1.3.4

CHARACTER 2.3.3.2

CLOSE 9.7.3

COMMON 3.2.3

COMPLEX 2.3.3.1

computed GO TO 6.1.2

CONTINUE 6.4

control Ch. 6

DATA 3.3.1
10.1.2

DIMENSION 3.2.2

DO 6.3.1
10.1.6

DOUBLE PRECISION 2.3.2
2.3.3.1

ELSE 6.2.3.2

ELSE IF 6.2.3.3

Statements (cont.)

END 1.3.2.1

END IF 6.2.3.4

ENDFILE 9.3.2

ENTRY 7.5.1

EQUIVALENCE 3.2.4

executable 1.3.4.1

explicit type specification 2.3.3

EXTERNAL 7.3.4
10.1.4

file positioning 9.3.2

FORMAT 8.2.1

FUNCTION 7.1.2.2
10.1.6

function declaration 7.1.2.2

IF 6.2

IMPLICIT 2.3.2
10.1.6

implied DO 3.3.1.2
9.2.1.2
10.1.6
10.2.2

input/output 9.2

INQUIRE 9.7.4
10.2.2

INTEGER 2.3.3.1

INTRINSIC 7.3.5

Statements (cont.)

label 1.3.3

LOGICAL 2.3.3.1

non-executable 1.3.4.2

OPEN 9.7.2
10.2.4.1
10.2.4.3

order of 1.3.5

PARAMETER 2.3.4

PAUSE 6.6
10.2.8
App. B

PRINT 9.3.1.1

PROGRAM 7.2

READ 9.2
9.3.1
9.4.1
9.5.1

REAL 2.3.3.1

RETURN 7.2.1.2
7.2.2.2

REWIND 9.3.2

SAVE 7.6

Statement function 7.1.2.3

STOP 6.5
10.2.8

SUBROUTINE 7.1.3.1

Statements (con'd)

terminal 6.3, 6.3.2

type specification 2.3, 10.1.6

unconditional GO TO 6.1.1

WRITE 9.2, 9.3.1, 9.4.1, 9.5.3.1

STOP statement 6.5, 10.2.8

Storage Ch. 3

allocation 3.2

arrays 3.1.3, 3.2.1.2

constants 3.1.1, 10.2.1

character data 3.1.4

requirements 3.1

variables 3.1.2, 3.2.1.1, 3.2.3.2, 10.2.1

Subprograms 1.2, Ch. 7

block data 1.2, 1.3.4.2, 3.3.2

function 1.2, 7.1.2

multiple entry to 7.5

subroutine	1.2 7.1.3 7.2.2
SUBROUTINE statement	7.1.3.1
Subroutines	7.1.1 7.1.3 7.2.2
used as arguments	7.3.4
Subscript expressions	2.2.4.1
Symbolic constants	2.2.2
Symbol names	10.1.3
Terminal	
parameter	6.3.1
statements	6.3 6.3.2
Transfer of control between program units	7.2
functions	7.2.1
subroutines	7.2.2
Transfer of values between program units	7.4
.TRUE. value	2.2.1.5 4.3.1
Truncation	4.1.7 5.1
Type specification	2.3 10.1.2 10.1.6
Unconditional GO TO statement	6.1.1
Unformatted	
direct access input and output	9.4.1.2

records	Ch. 8 9.1.1.1
sequential access input and output	9.3.1.2
Unit number	9.1
Using the consolidator	12.4.1
Using Pascal routines	12.3.2.6
Utilities	12.2
Valid argument correspondence	11.2.1
Valid function correspondence	11.2.2
Value lists	3.3.1.1
Values, types of	2.1
Variables	2.2.3
storage of	3.1.2 3.2.1.1 3.2.3.2 10.2.1
use of	7.3.2
WRITE statement	9.2
formatted direct access	9.4.1.1
formatted sequential access	9.3.1.1
list directed	9.5.3.1
unformatted direct access	9.4.1.2
unformatted sequential access	9.3.1.2