Burroughs

B 20 Systems FORTRAN

REFERENCE MANUAL

(Relative to Release Levels 1.3 and 2.2)

This Manual Replaces All Previous Editions of Form 1148632.

PRICED ITEM

Printed in U.S. America

February 1983

1148632

Burroughs

B 20 Systems

FORTRAN

REFERENCE MANUAL

(Relative to Release Levels 1.3 and 2.2)

This Manual Replaces All Previous Editions of Form 1148632.

PRICED ITEM

Printed in U.S. America

February 1983

1148632

Burroughs cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Correspondence regarding this publication should be forwarded using the Remarks form at the back of the manual, or may be addressed directly to Burroughs Corporation, TSG Product Support, Route 202 North, Flemington, N.J. 08822, U.S. America.

LIST OF EFFECTIVE PAGES

Page	Issue
iii	Original
iv	Blank
v thru viii	Original
1-1 and 1-2	Original
2-1 and 2-2	Original
3-1 thru 3-4	Original
4-1 thru 4-4	Original
5-1 and 5-2	Original
6-1 thru 6-9	Original
6-10	Blank
7-1	Original
7-2	Blank
8-1 thru 8-7	Original
8-8	Blank
9-1 thru 9-3	Original
9-4	Blank
10-1 thru 10-13	Original
10-14	Blank
11-1 thru 11-15	Original
11-16	Blank
12-1 thru 12-7	Original
12-8	Blank
13-1 thru 13-13	Original
13-14	Blank
14-1 thru 14-5	Original
14-6	Blank
A-1 thru A-14	Original
B-1 and B-2	Original
C-1 thru C-10	Original
D-1 thru D-4	Original
E-1 thru E-5	Original
E-6	Blank

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

SECTION 1
OVERVIEW

This is a reference manual for the Burroughs FORTRAN 77 language system. The Burroughs FORTRAN software product conforms to the Standard ANSI X3.9-1978 at the subset level. It also includes features from the full level of X3.9-1978, such as DOUBLE PRECISION.

The Burroughs FORTRAN system has also been enhanced to ease the conversion of existing FORTRAN 66 programs. For example, Hollerith constants are included and FORTRAN 66 semantics for DO loops are a compiler option.

Calls from Burroughs FORTRAN provide access to all B 20 Operating System services, such as direct (random) access to disk files, interrupt handling, and process creation, thereby supporting various kinds of system programming. Calls also extend the range of services needed by the commercial applications programmer: ISAM, Sort/Merge, and the Forms Run-Time. The Burroughs Linker allows you to combine FORTRAN object modules with those of other languages, for example, the Assembler, to facilitate writing applications that need different languages for different parts.

You should have some prior knowledge of some dialect of FORTRAN. This Manual is not a tutorial; rather, each section fully explains one part of the FORTRAN language system.

The manual is organized as follows: Section 1, 2, and 3 are general and describe the manual and basics necessary to successfully use the FORTRAN system.

Sections 4, 5, and 6 describe the data types available in the language and how a program specifies a particular data type as the type of an identifier or constant.

Section 7 deals with the DATA statement, used for initialization of memory.

Sections 8, 9, 10, and 11 define the executable parts of programs and the meanings associated with the various executable constructs. Although Section 11 describes the I/O statements, the FORMAT statement and formatted I/O are described in Section 12.

Section 13 describes the subroutine structure of a FORTRAN compilation, including parameter passing and intrinsic (system-provided) functions.

The notational conventions used in this manual are described below.

Uppercase letters and special characters: are to be written as shown in programs.

Lowercase letters and words: indicate entities for which there is a substitution in actual statements as described in the text. Once a lowercase entity is defined, it retains its meaning for the entire context of the discussion.

Example of upper- and lowercase: the format that describes editing of integers is denoted Iw, where w is a nonzero, unsigned integer constant. Thus, in an actual statement, a program might contain I3 or I44. The format that describes editing of reals is Fw.d, where d is an unsigned integer constant. In an actual statement, F7.4 or F22.0 are valid. Notice that the period, as a special character, is taken literally.

Brackets: indicate optional items. For example, A [w] indicates that either A or A12 is valid (as a means of specifying a character format).

Ellipsis(. . .): indicates that the optional item preceding the ellipsis may appear one or more times. For example, the computed GOTO statement is described by:

GOTO (s [,s] . . .) [,] i

indicating that the syntactic items denoted by s may be repeated any number of times with commas separating them.

Blanks: normally have no significance in the description of FORTRAN statements. The general rules for blanks, covered in Section 3, govern the interpretation of blanks in all contexts.

SECTION 2
USING BURROUGHS FORTRAN 77

COMPILING, LINKING, AND EXECUTING A FORTRAN PROGRAM

This Section assumes that you are familiar with the basic operation of the Burroughs Executive and Linker. (See the manuals of the same names.) The mechanics of compiling, linking, and executing a FORTRAN program are outlined here.

The Burroughs FORTRAN system consists of the FORTRAN compiler and a library of object modules that make up the FORTRAN run-time library. The first step in creating an executable FORTRAN program is compiling its (one or more) source modules. The object modules that result are then linked with the FORTRAN run-time library, producing a run file that can be invoked from the Executive.

The object modules and libraries to be linked can include the output of the Assembler or Pascal compiler, as well as the output of the FORTRAN compiler.

If Burroughs Operating System interfaces, access methods, or optional Burroughs software products such as Sort/Merge or ISAM, are called directly from FORTRAN, a special "mediator" object module, ForGen.Obj, must also be included. (See Appendix C for details about ForGen.Obj.)

Compiling

Invoke the FORTRAN compiler with the Executive's FORTRAN command. The following form appears.

FORTRAN
Source file _____
[Object file] _____
[List file] _____
[Object list file] _____

For information on filling in a form, see the section of that name in the Executive Manual.

Source file

is the name of the FORTRAN source file to be compiled.

[Object file]

is the name of the file to which to write the object code that results from the compilation. If no file is named, a default object file is chosen as follows: treat the source name as a character string, strip off any final suffix beginning with a period ".", and add the characters ".Obj". The result is the name of the file. For example, if the source file is:

[Dev] <Jones> Main

then the default object file is:

[Dev] <Jones> Main.Obj

If the source file is:

Prog.Fortran

then the default object file is:

Prog.Obj

[List file]

is the name of the file to which to write a listing of the compilation. If no file is named, a default list file is chosen, in the same manner the default object file is created except that the string added is ".Lst" instead of ".Obj".

[Object list file]

is the name of the file to which to write the listing of the generated object code. If no file is named, the default is no generation of the object list file.

To suppress the generation of an object or list file, specify " [Nul] Linking

Linking

Invoke the Linker with the Executive's Link command, as described in the Linker/Librarian Manual. Note the following special requirements:

- [Libraries] must include the library <Sys>Fortran.Lib, and
- [DS allocation?] must be Yes.

Executing

A compiled, linked FORTRAN program is executed in the same manner as any other user program; that is, it is run from the Executive or chained to by an already executing program. See the Run File command in the Executive Manual and the Chain operation in the "Task Management" section of the B 20 Operating System Manual for details.

SECTION 3
BASIC PROGRAM STRUCTURE

In the most basic sense, a FORTRAN program is a sequence of characters that, when submitted to the compiler, are interpreted in various contexts as characters, identifiers, labels, constants, lines, statements, and so on. This Section defines these entities.

CHARACTER SET

A FORTRAN source program is a sequence of characters consisting of: (1) letters - the 52 upper- and lowercase letters A through Z and a through z, (2) digits - 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, and/or (3) special characters - the remaining printable characters of the ASCII character set.

The letters and digits, treated as a single group, are called the alphanumeric characters. FORTRAN interprets lowercase letters as uppercase letters in all contexts except in character constants and Hollerith fields. Thus, the following user-defined names are all indistinguishable to the FORTRAN system.

ABCDE abcde AbCdE aBcDe

The collating sequence for the FORTRAN character set is the ASCII sequence.

LINES

A FORTRAN source program can also be considered a sequence of lines, ending with the character code OAh (produced by pressing the RETURN key while in the Editor). Only the first 72 characters in a line are treated as significant by the compiler. Any trailing characters in a line are ignored. Note that lines with fewer than 72 characters are possible and, if a line is shorter than 72 characters, the compiler treats its length as significant (for an illustration of this, see "Character" in Section 4, which describes character constants).

COLUMNS

The characters in a given line fall into columns, with the first character being in column 1, the second in column 2, and so on. The column in which a character resides is significant in FORTRAN. Columns 1 through 5 are reserved for statement labels and column 6 for continuation indicators.

BLANKS

The blank character, with the exceptions noted below, has no significance in a FORTRAN source program and may be used for improving the readability of FORTRAN programs. The exceptions are:

1. blanks within string constants are significant,
2. blanks within Hollerith fields are significant, and
3. a blank in column 6 distinguishes initial lines from continuation lines.

COMMENT LINES

A line is treated as a comment if any one of the following conditions is met: (1) a "C" (or "c") in column 1, (2) an "*" in column 1, or (3) the line contains all blanks.

Comment lines do not affect the execution of the FORTRAN program in any way. Comment lines must be followed immediately by an initial line or another comment line. They must not be followed by a continuation line. Note that extra blank lines at the end of a FORTRAN program result in a compile-time error since the system interprets them as comment lines but they are not followed by an initial line.

LABELS, INITIAL LINES, CONTINUATION LINES, AND STATEMENTS

This subsection defines a FORTRAN "statement" in terms of the input character stream. The compiler recognizes certain groups of input characters as complete statements according to the rules specified here. The remainder of this Manual further defines the specific statements and their properties. When it is necessary to refer to specific kinds of statements here, they are simple referred to by name.

Labels

A statement label is a sequence of from one to five digits. At least one digit must be nonzero. A label may be placed anywhere in columns 1 through 5 of an initial line. Blanks and leading zeros are not significant.

Initial Lines

An initial line is any line that is not a comment line or a compiler directive line and that contains a blank or a 0 in column 6. The first five columns of the line must either be all blank or contain a label. With the exception of the statement following a logical IF, FORTRAN statements begin with an initial line.

Continuation Lines

A continuation line is any line that is not a comment line or a compiler directive line and that contains any character in column 6 other than a blank or a 0. The first five columns of a continuation line must be blanks. A continuation line increases the amount of room to write a statement. If it will not fit on a single initial line, it may be extended to include up to nine continuation lines.

Statements

A FORTRAN statement consists of an initial line, followed by up to nine continuation lines. The characters of the statement are the up to 660 characters found in columns 7 through 72 of these lines. The END statement must be wholly written on an initial line and no other statement may have an initial line that appears to be an END statement.

PROGRAM UNITS

The FORTRAN language enforces a certain ordering among statements and lines that make up a FORTRAN compilation. In general, a compilation consists of, at most, one main program and none or some subprograms (see Section 13 for more information on compilation units and subroutines). The various rules for ordering statements appear below.

Main Program and Subprogram

A subprogram begins with either a SUBROUTINE or a FUNCTION statement and ends with an END statement. A main program begins with a PROGRAM statement, or any statement other than a SUBROUTINE or FUNCTION statement, and ends with an END statement. A subprogram or the main program is referred to as a program unit.

Statement Ordering

Within a program unit, whether a main program or a subprogram, statements must appear in an order consistent with the following rules.

1. A SUBROUTINE or FUNCTION statement, or PROGRAM statement if present, must appear as the first statement of the program unit.

2. FORMAT statements may appear anywhere after the SUBROUTINE or FUNCTION statement, or PROGRAM statement if present.

3. All specification statements must precede all DATA statements, statement function statements, and executable statements.

4. All DATA statements must appear after the specification statements and precede all statement function statements and executable statements.

5. All statement function statements must precede all executable statements.

6. Within the specification statements, the IMPLICIT statement must precede all other specification statements.

These rules are illustrated in Figure 3-1 below.

<table>
<tr><td rowspan="6">Comment Lines</td><td colspan="2">PROGRAM, FUNCTION, or SUBROUTINE Statement</td></tr>
<tr><td rowspan="5">FORMAT Statements</td><td>IMPLICIT Statements</td></tr>
<tr><td>Other Specification Statements</td></tr>
<tr><td>DATA statements</td></tr>
<tr><td>Statement Function Statements</td></tr>
<tr><td>Executable Statements</td></tr>
<tr><td colspan="3">END Statement</td></tr>
</table>

Figure 3-1. Order of Statements within Program Units.

Figure 3-1 is interpreted as follows.

- classes of lines or statements above or below other classes must appear in the designated order.

- classes of lines or statements may be interspersed with other classes that appear across from one another.

SECTION 4
DATA TYPES

There are five basic data types in Burroughs FORTRAN: integer, real, double precision, logical, and character. This Section describes the properties of, the range of values for, and the form of constants for each type.

INTEGER

The integer data type consists of a subset of the integers. An integer value is an exact representation of the corresponding integer. An integer variable occupies two or four bytes of memory. A 2-byte integer can contain any value in the range -32768 to 32767. A 4-byte integer can contain any value in the range -2,147,483,648 to 2,147,483,647. Integer constants consist of a sequence of one or more decimal digits preceded by an optional arithmetic sign, + or -, and must be in range. A decimal point is not allowed in an integer constant. The following are examples of integer constants.

123	+123	-123	0
00000123	32767	-32768	

An integer can be specified in Burroughs FORTRAN as INTEGER*2, INTEGER*4, or INTEGER. The first two specify, respectively, 2- and 4-byte integers. The third specifies either 2- or 4-byte integers, according to the setting of the STORAGE directive (the default is four bytes).

REAL

The real data type consists of a subset of the single-precision real numbers. A single-precision real value is normally an approximation of the desired real number. A single-precision real value occupies four bytes of memory. The range of single-precision real values is approximately:

3.0E-39 to 1.7E+38 (positive range)
-1.7E+38 to -3.0E-39 (negative range)

The precision is greater than six decimal digits.

A basic real constant consists of an optional sign followed by an integer part, a decimal point, and a fraction part. The integer and fraction parts consist of

one or more decimal digits, and the decimal point is a period, ".". Either the integer part or the fraction part may be omitted, but not both. Some sample basic real constants follow.

```
-123.456      +123.456      123.456
-123.         +123.         123.
   -.456         +.456         .456
```

An exponent part consists of the letter "E" followed by an optionally signed integer constant. An exponent indicates that the value preceding it is to be multiplied by 10 to the value of the exponent part's integer. Some sample exponent parts follow.

```
E12      E-12      E+12      E0
```

A real constant is either a basic real constant, a basic real constant followed by an exponent part, or an integer constant followed by an exponent part. For example.

```
+1.000E-2       1.E-2              1E-2
+0.01         100.0E-4           .0001E+2
```

All represent the same real number - one one-hundreth.

DOUBLE PRECISION

The double-precision real data type consists of a subset of the double-precision real numbers. A double-precision real value is normally an approximation of the desired real number. A double-precision real value occupies eight bytes of memory. The range of double-precision real values is approximately:

```
3.0D-39 to 1.7D+38 (positive range)
-1.7D+38 to -3.0D-39 (negative range)
```

The precision is greater than 15 decimal digits.

The basic double-precision real constant consists of an optional sign followed by an integer part, a decimal point, and a fraction part. The integer and fraction parts consist of one or more decimal digits, and the decimal point is a period, ".". Either the integer part or the fraction part may be omitted, but not both. Some sample basic double-precision real constants follow.

```
-123.456      +123.456      123.456
-123.         +123.         123.
   -.456         +.456         .456
```

An exponent part consists of the letter "D" followed by an optionally signed integer constant. An exponent indicates that the value preceding it is to be multiplied by 10 to the value of the exponent part's integer. Some sample exponent parts follow.

```
D12      D-12      D+12      D0
```

A double-precision real constant is either a basic double-precision real constant, a basic double-precision real constant followed by an exponent part, or an integer constant followed by an exponent part. For example.

```
+1.123456789D-2        1.D-2              1D-2
+0.000000001           100.0000005D-4     .00012345D+2
```

LOGICAL

The logical data type consists of the two logical values .TRUE. and .FALSE. A logical variable occupies two or four bytes of memory.

> NOTE
> Whether two or four is controlled by the STORAGE directive, with a default of four bytes. The significance of a logical variable is unaffected by the STORAGE directive, which is present primarily to allow compatibility with the ANSI requirement that logical, single-precision real, and integer variables are all the same size.

There are only two logical constants, .TRUE. and.FALSE., representing the two corresponding logical values. The internal representation of .FALSE. is a word of all 0's, and the representation of .TRUE. is a word of all 0's with a 1 in the least significant bit. If a logical variable contains any other bit values, its logical meaning is undefined.

CHARACTER

The character data type consists of a sequence of ASCII characters. The length of a character value is equal to the number of characters in the sequence. The length of a particular constant or variable is fixed, and must be between 1 and 255 characters. A character variable occupies one byte of memory for each character in the sequence, plus one byte if the length is odd.

Character variables are always aligned on word boundaries. The blank character is permitted in a character value and is significant.

A character constant consists of a sequence of one or more characters enclosed by a pair of apostrophes. Blank characters are permitted in character constants, and count as one character each. An apostrophe within a character constant is represented by two consecutive apostrophes with no blanks between. The length of a character constant is equal to the number of characters between the apostrophes, with doubled apostrophes counting as a single apostrophe character. Some sample character constants are:

```
'A'       ' '  'Help!'
'A very long CHARACTER constant'                    ''''
```

The last example, '''', represents a single apostrophe, '.

FORTRAN permits source lines of up to 72 columns. Shorter lines are not padded to 72 columns, but left as input. When a character constant extends across a line boundary, its value is as if the portion of the continuation line beginning with column 7 is juxtaposed immediately after the last character on the initial line. Thus, the FORTRAN source:

```
200       CH = 'ABC
         X DEF'
```

(where there are no further characters on the line after the 'ABC) is equivalent to:

```
200       CH = 'ABC DEF'
```

with the single space between the C and D being the equivalent to the space in column 7 or the continuation line. Very long character constants can be represented in this manner.

SECTION 5
FORTRAN NAMES

A FORTRAN name, or identifier, consists of an initial alphabetic character followed by a sequence of up to five alphanumeric characters. Blanks may appear within a FORTRAN name, but have no significance. A name is used to denote a user- or system-defined variable, array, function, subroutine, and so on. Any valid sequence of characters can be used for any FORTRAN name. There are no reserved names as in other languages. Sequences of alphabetic characters used as keywords are not to be confused with FORTRAN names. The compiler recognizes keywords by their context and in no way restricts the use of user-defined names. Thus, a program can have, for example, an array named IF, READ, or GOTO, with no error (as long as it conforms to the rules that all arrays must obey).

SCOPE OF FORTRAN NAMES

The scope of a name is the range of statements in which that name is known, or can be referenced, within a FORTRAN program. In general, the scope of a name is either global or local, although there are several exceptions. A name can only be used in accordance with a single definition within its scope. The same name, however, can have different definitions in distinct scopes.

A name with global scope can be used in more than one program unit (a subroutine, function, or the main program) and still refer to the same entity. Names with global scope can only be used in a single, consistent manner within the same program. All subroutine, function subprogram, and common names, as well as the program name, have global scope. Therefore, there cannot be a function subprogram that has the same name as a subroutine subprogram or a common data area. Similarly, no two function subprograms in the same program can have the same name.

A name with local scope is only visible (known) within a single program unit. A name with local scope can be used in another program unit with a different meaning, or with a similar meaning, but is not required to have similar meanings in a different scope. The names of variables, arrays, parameters, and statement functions all have local scope.

One exception to the scoping rules is the name given to common data blocks. It is possible to refer to a common name with a global scope in the same program unit that an identical name with a local scope appears. This is permitted because common names are always enclosed in slashes, such as /NAME/, and are therefore always distinguishable from ordinary names by the compiler.

Another exception to the scoping rules is made for parameters to statement functions. The scope of statement function parameters is limited to the single statement forming that statement function. Any other use of those names within that statement function is not permitted, and any other use outside that statement function is permitted.

When a user name that has not appeared before is encountered in an executable statement, the compiler infers from the context of its use how to classify that name. If the name is used in the context of a variable, the compiler creates an entry into the symbol table for a variable of that name. Its type is inferred from the first letter of its name. Normally, variables beginning with the letters I, J, K, L, M or N are considered integers, while all others are considered reals, although these defaults can be overridden by an IMPLICIT statement (see Section 6).

If an undeclared name is used in the context of a function call, a symbol table entry is created for a function of that name, with its type being inferred in the same manner as that of a variable.

Similarly, a subroutine entry is created for a newly encountered name that is the target of a CALL statement. If an entry for such a subroutine or function name exists in the global symbol table, its attributes are coordinated with those of the newly created symbol table entry. If any inconsistencies are detected, such as a previously defined subroutine name being used as a function name, an error message is issued.

SECTION 6
SPECIFICATION STATEMENTS

Specification statements in Burroughs FORTRAN 77 are nonexecutable. They define the attributes of user-defined variable, array, and function names. There are eight kinds of specification statements.

1. IMPLICIT
2. DIMENSION
3. Type
4. COMMON
5. EXTERNAL
6. INTRINSIC
7. SAVE
8. EQUIVALENCE

Specification statements must precede all executable statements in a program unit and may appear, except IMPLICIT, in any order within their own group. IMPLICIT statements must precede all other specification statements in a program unit.

IMPLICIT STATEMENT

An IMPLICIT statement defines the default type for user-declared names. The form of an IMPLICIT statement is as follows.

IMPLICIT *type* (*a* [,*a*] . . .) ,*type* (*a* [,*a*] . . .) . . .

where:

type is one of the types shown in Table 6-1 below.

a is either a single letter or a range of letters. A range of letters is indicated by the first and last letters in the range, separated by a minus sign. For a range, the letters must be in alphabetical order.

An IMPLICIT statement defines the type and size for all user-defined names that begin with the letter or letters that appear in the specification. An IMPLICIT statement applies only to the program unit in which it appears. IMPLICIT statements do not change the type of any intrinsic functions.

Table 6-1 Memory Requirements of FORTRAN Data Types.

Type	Memory (bytes)
LOGICAL	2 or 4[1]
LOGICAL*2	2
LOGICAL*4	4
INTEGER	2 or 4[1]
INTEGER*2	2
INTEGER*4	4
CHARACTER[2]	1
CHARACTER*n[2]	n[3]
REAL[4]	4
REAL*4[4]	4
REAL*8[5]	8
DOUBLE PRECISION[5]	8

1. Either 2 or 4 bytes are used. The default is 4, but may be set explicitly to either 2 or 4 with the STORAGE directive.
2. CHARACTER and CHARACTER*1 are synonyms.
3. If n is odd, then n + 1 bytes of memory are used.
4. REAL and REAL*4 are synonyms.
5. REAL*8 and DOUBLE PRECISION are synonyms.

IMPLICIT types can be overridden or confirmed for any specific user name by the appearance of that name in a subsequent type statement. An explicit type in a FUNCTION statement also takes priority over an IMPLICIT statement. If the type in question is a character type, the length of the user name is also overridden by a later type definition.

A program unit can have more than one IMPLICIT statement, but all IMPLICIT statements must precede all other specification statements in that program unit. The same letter cannot be defined more than once in an IMPLICIT statement in the same program unit.

```
C   EXAMPLE OF IMPLICIT STATEMENT
    IMPLICIT REAL*4 (C-M)
    IMPLICIT INTEGER (A, B)
    IMPLICIT CHARACTER*10 (N)
    AGE = 10
    NAME = 'PAUL'
    COUNT = 1/2
```

DIMENSION STATEMENT

A DIMENSION statement specifies that a user name is an array. The form of a DIMENSION statement is as follows.

DIMENSION var(dim) [,var(dim)] ...

where:

var(dim) is an array declarator.

An array declarator is of the form:

name (d [,d] ...)

where:

name is the user-defined name of the array.

d is a dimension declarator.

Dimension Declarators

The number of dimensions in the array is the number of dimension declarators in the array declarator. The maximum number of dimensions is three. A dimension declarator can be:

1. an unsigned integer constant,
2. a user name corresponding to a nonarray integer formal argument, or
3. an asterisk.

A dimension declarator specifies the upper bound of the dimension. The lower bound is always one.

If a dimension declarator is an integer constant, then the array has the corresponding number of elements in that dimension. An array has a constant size if all of its dimensions are specified by integer constants.

If a dimension declarator is an integer argument, then that dimension is defined to be of a size equal to the initial value of the integer argument upon entry to the subprogram unit at execution time. In such a case, the array is called an adjustable-sized array.

If the dimension declarator is an asterisk, the array is an assumed-sized array and the upper bound of that dimension is not specified.

All adjustable- and assumed-sized arrays must also be formal arguments to the program unit in which they appear. Also, an assumed-size dimension declarator may only appear as the last dimension in an array declarator.

The order of array elements in memory is column-major order, that is, the leftmost subscript changes most rapidly in a memory sequential reference to all array elements.

Array Element Name

The form of an array element name is:

arr(sub [,sub] ...)

where:

arr	is the name of an array.
sub	is a subscript expression.

A subscript expression is an integer expression used in selecting a specific element of an array. The number of subcript expressions must match the number of dimensions in the array declarator. The value of a subscript expression must be between 1 and the upper bound for the dimension it represents.

```
C EXAMPLE OF DIMENSION STATEMENT
      DIMENSION ARRAY(10)
C DEFINE CHARACTER STRING
      CHARACTER*10 NAME
C DEFINE ARRAY OF CHARACTER STRINGS
      DIMENSION NAME(24)
```

TYPE STATEMENT

A type statement specifies the type of user-defined names. A type statement can confirm or override the implicit type of a name. A type statement can also specify dimension information.

A user name for a variable, array, external function, or statement function may appear in a type statement. Such an appearance defines the type of that name for the entire program unit. Within a program unit, a name can have its type explicitly specified by a type statement only once. A type statement may confirm the type of an intrinsic function, but it is not required. The name of a subroutine or main program cannot appear in a type statement.

The type declaration statement has the following form.

typ v [,v] ...

where:

typ	is one of the data-type specifiers.
v	is the symbolic name of a variable, array, statement function, function subprogram, or an array declarator.

The following rules apply to a type declaration statement.

- A type declaration statement must precede all executable statements.
- The data type of symbolic name can be declared only once.
- A type declaration statement cannot be labeled.
- A type declaration statement can be used to declare an array by appending an array declarator to an array name.

A symbolic name can be followed by a data-type length specifier of the form *s, where s is one of the acceptable lengths for the data type being declared. Such a specification overrides the length attribute that the statement implies, and assigns a new length to the specified item. If both a data-type length specifier and an array declarator are specified, the data-type length specifier goes last. Examples of type declaration statements are:

```
      INTEGER COUNT, MATRIX(4,4), SUM
      REAL MAN, IABS
      LOGICAL SWITCH

      INTEGER*2 Q, M12*4, IVEC(10)*4
      REAL*8 WX1, WX3*4, WX5, WX6*8

C EXAMPLE OF TYPE STATEMENT
      CHARACTER NAME*10, CITY*80, CH
      DOUBLE PRECISION DELTA
```

A COMMON statement provides a method of sharing memory between two or more program units. Such program units can share the same data without passing it as arguments. The form of the COMMON statement is:

COMMON[/[cname]/] nlist[[,] /[cname]/nlist]...

where:

cname — is a common block name. If a cname is omitted, then the blank common block is specified.

nlist — is a comma-separated list of variable names, array names, and array declarators. Formal argument names and function names cannot appear in a COMMON statement.

In each COMMON statement, all variables and arrays appearing in each nlist following a common block name cname are declared to be in that common block. If the first cname is omitted, all elements appearing in the first nlist are specified to be in the blank common block.

Any common block name can appear more than once in COMMON statements in the same program unit. All elements in all nlists for the same common block are allocated in that common memory area in the order they appear in the COMMON statement.

All elements in a single common area must be either all or none of type character. Furthermore, if two program units refer to the same named common block containing character data, the association of character variables of different length is not permitted. Two variables are said to be associated if they refer to the same actual memory.

The size of a common block is equal to the number of bytes of memory required to hold all elements in that common block. If the same named common block is referred to by several distinct program units, the common blocks must be of the same length and the blocks are juxtaposed at their lowest address. Blank common blocks, however, can have different lengths in different program units. The maximum length may occur in any program unit.

```
C EXAMPLE OF BLANK AND NAMED COMMONS
    PROGRAM MYPROG
    COMMON I, J, X, K(10)
    COMMON /MYCOM/ A(3)
    .
    .
    .
    END
    SUBROUTINE
    COMMON I, J, X, K(10)
    COMMON /MYCOM/ A(3)
```

EXTERNAL STATEMENT

An EXTERNAL statement identifies a user-defined name as an external subroutine or function. The form of an EXTERNAL statement is:

EXTERNAL name [,name]...

where:

name is the name of an external subroutine or function.

Appearance of a name in an EXTERNAL statement declares that name to be an external procedure. Statement function names cannot appear in an EXTERNAL statement. If an intrinsic function name appears in an EXTERNAL statement, then that name becomes the name of an external procedure, and the corresponding intrinsic function can no longer be called from that program unit. A user name can only appear once in an EXTERNAL statement in a given program unit.

In assembly language and Pascal, the term EXTERNAL (or EXTRN) declares that an object is defined outside the current unit of compilation or assembly. This is not necessary in FORTRAN since, in accord with the standard FORTRAN practice, any object referred to but not defined in a compilation unit is assumed to be defined externally. Therefore, in FORTRAN, EXTERNAL specifies that a particular user-defined subroutine or function is to be used as a procedural parameter.

```
C EXAMPLE OF EXTERNAL STATEMENT
      EXTERNAL MYFUNC, MYSUB
C MYFUNC AND MYSUB ARE PARAMETERS TO CALC
      CALL CALC (MYFUNC, MYSUB)
```

INTRINSIC STATEMENT

An INTRINSIC statement declares that a user name is an intrinsic function. The form of an INTRINSIC statement is:

INTRINSIC name [,name] ...

where:

name is an intrinsic function name.

Each user name may only appear once in an INTRINSIC statement. If a name appears in an INTRINSIC statement, it cannot appear in an EXTERNAL statement. All names used in an INTRINSIC statement must be system-defined INTRINSIC functions. For a list of these functions, see Section 13.

```
C EXAMPLE OF INTRINSIC STATEMENT
      INTRINSIC SIN, COSIN
C SIN AND COSIN ARE PARAMETERS TO CALC2
      X = CALC2 (SIN, COSIN)
```

SAVE STATEMENT

A SAVE statement retains the definition of a common block after the return from a procedure that defines that common block. Within a subroutine or function, a common block that was specified in a SAVE statement does not become undefined upon exit from the subroutine or function. The form of a SAVE statement is:

SAVE /name/ [,/name/] . . .

where:

name — is the name of a common block.

Note that since all common blocks are statically allocated, the SAVE statement has no effect.

```
C EXAMPLE OF SAVE STATEMENT
    SAVE /MYCOM/
```

EQUIVALENCE STATEMENT

An EQUIVALENCE statement specifies that two or more variables or arrays are to share the same memory. If the shared variables are of different types, the EQUIVALENCE does not cause any kind of automatic type conversion. The form of an EQUIVALENCE statement is:

EQUIVALENCE (nlist) , (nlist) . . .

where:

nlist — is a list of at least two variable names, array names, or array element names. Argument names may not appear in an EQUIVALENCE statement. Subscripts must be integer constants and must be within the bounds of the array they index.

An EQUIVALENCE statement specifies that the memory sequences of the elements that appear in the list nlist have the same first memory location. Two or more variables are said to be associated if they refer to the same actual memory. Thus, an EQUIVALENCE statement causes its list of variables to become associated. An element of type character can only be associated with another element of type character with the same length. If an array name appears in an EQUIVALENCE statement, it refers to the first element of the array.

Restrictions on Equivalence Statements

An EQUIVALENCE statement cannot specify that the same memory location is to appear more than once, such as:

```
REAL R,S(10)
EQUIVALENCE (R,S(1)), (R,S(5))
```

which forces the variable R to appear in two distinct memory locations. Furthermore, an EQUIVALENCE statement cannot specify that consecutive array elements are not stored in sequential order. For example.

```
REAL R(10), S(10)
EQUIVALENCE (R(1),S(1)), (R(5),S(6))
```

is not permitted.

When EQUIVALENCE statements and COMMON statements are used together, several further restrictions apply. An EQUIVALENCE statement cannot cause memory in two different common blocks to become equivalenced. An EQUIVALENCE statement can extend a common block by adding memory element following the common block, but not preceding the common block. Note that extending a named common block by an EQUIVALENCE statement must not cause its length to be different from the length of the named common in other program units. For example.

```
COMMON /ABCDE/ R(10)
REAL S(10)
EQUIVALENCE (R(1), S(10))
```

is not permitted because it extends the common block by adding memory preceding the start of the block.

```
C EXAMPLE OF EQUIVALENCE STATEMENT
      CHARACTER NAME, FIRST, MIDDLE, LAST
      DIMENSION NAME(60), FIRST(20),
     1          MIDDLE(20), LAST(2)
      EQUIVALENCE (NAME(1), FIRST(1)),
     1          (NAME(21), MIDDLE(1)),
     2          (NAME(41), LAST(1)),
```

SECTION 7
DATA STATEMENT

The DATA statement assigns initial values to variables. A DATA statement is a nonexecutable statement. If present, it must appear after all specification statements and prior to any statement function statements or executable statements. The form of a DATA statement is:

DATA nlist / clist /[[,] nlist / clist /] . . .

where:

nlist — is a list of variable, array element, or array names.

clist — is a list of constants, or constants preceded by an integer constant repeat factor and an asterisk, such as:

5*3.14159 3*Help 100*0

A repeat factor followed by a constant is the equivalent of a list of all constants of that constant's value repeated a number of times equal to the repeat constant.

There must be the same number of values in each clist as there are variables or array elements in the corresponding nlist. The appearance of an array in an nlist is the equivalent to a list of all elements in that array in memory sequence order. Array elements must be indexed only by constant subscripts.

The type of each noncharacter element in a clist must be the same as the type of the corresponding variable or array element in the accompanying nlist. Each character element in a clist must correspond to a character variable or array element in the nlist, and must have a length that is less than or equal to the length of that variable or array element. If the length of the constant is shorter, it is extended to the length of the variable by adding blank characters to the right. Note that a single character constant cannot be used to define more than one variable or even more than one array element.

Only local variables and array elements can appear in a DATA statement. Formal arguments, variables in common, and function names cannot be assigned initial values with a DATA statement.

SECTION 8
EXPRESSIONS

FORTRAN has four classes of expressions.

1. arithmetic
2. character
3. relational
4. logical

ARITHMETIC EXPRESSIONS

An arithmetic expression produces a value that is either of type integer or real. The simplest forms of arithmetic expressions are:

1. unsigned integer or real constant
2. integer or real variable reference
3. integer or real array element reference
4. integer or real function reference

The value of a variable reference or array element reference must be defined for it to appear in an arithmetic expression. Moreover, the value of an integer variable must be defined with an arithmetic value, rather than a statement label value previously set in an ASSIGN statement.

Other arithmetic expressions are built up from the above simple forms using parentheses and the arithmetic operators of Table 8-1 below.

Table 8-1. Arithmetic Operators.

Operator	Representing Operation	Precedence
**	Exponentiation	Highest
/	Division	Intermediate
*	Multiplication	Intermediate
-	Subtraction or Negation	Lowest
+	Addition or Identity	Lowest

All of the operators are binary operators, appearing between their arithmetic expression operands. The + and - may also be unary, preceding the operand.

Operations of equal precedence are left-associative, except exponentiation, which is right-associative. Thus,

```
A/B*C
```

is the same as:

```
(A/B)*C
```

and:

```
A**B**C
```

is the same as:

```
A**(B**C).
```

Arithmetic expressions can be formed in the usual mathematical sense, as in most programming languages, except that FORTRAN prohibits two operators from appearing consecutively. Thus,

```
A**-B
```

is prohibited, although:

```
A**(-B)
```

is permissible. Note that unary minus is also of lowest precedence so that:

```
-A*B
```

is interpreted as:

```
-(A*B)
```

Parentheses may be used in a program to control the associativity and the order of operator evaluation in an expression.

Integer Division

The division of two integers results in a value that is the mathematical quotient of the two values, truncated toward 0. Thus, 7/3 evaluates to 2, (-7)/3 evaluates to -2, 9/10 evaluates to 0 and 9/(-10) evaluates to 0.

Type Conversions and Result Types of Arithmetic Operators

When all operands of an arithmetic expression are of the same type, the value produced by the expression is also of that type. When the operands are of different data types, the value produced by the expression is of a data type determined by the rank shown in Table 8-2 below.

Table 8-2. Data Type Ranks.

Data Type	Rank
INTEGER*2	1 (Lowest)
INTEGER*4	2
REAL	3
DOUBLE PRECISION	4 (Highest)

When an operation has two arithmetic operands of different data types, the value of the data type produced is the data type of the highest-ranked operand. For example, an operation on an integer and a real element produces a value of data type real.

The data type of an expression is the data type of the result of the last operation performed in evaluating the expression.

The data types of operations are classified as either INTEGER*2, INTEGER*4, REAL, or DOUBLE PRECISION.

Integer operations are performed on integer operands only. A fraction resulting from division is truncated in integer arithmetic, not rounded. Thus,

1/4 + 1/4 + 1/4 + 1/4

evaluates to 0, not 1.

Note that memory for the type INTEGER (without the *2 or *4 extensions) is dependent on the usage of the STORAGE directive. See Section 14 for details.

Real operations are performed on real operands or combinations of real and integer operands only. Any integer operands are first converted to real data type by giving each a fractional part equal to 0. Real arithmetic is then used to evaluate the expression. But in this statement:

```
Y = (I/J)*X
```

integer division is performed on I and J, and a real multiplication on the result and X. An operation between a REAL*4 and a REAL*8 operand proceeds as in a REAL*8 operation, producing a REAL*8 result.

Double-precision operations are performed on real, integer, or double-precision operands. A real or integer operand is converted to double-precision by making it the most significant part of a double-precision element. The least significant part is 0. Double-precision arithmetic is then used to evaluate the expression.

The accuracy of a real operand is not increased by converting it to a double-precision operand. For example, the real number that represents the fraction 2/3:

```
0.6666667
```

is converted to:

```
0.6666667000000000D0
```

CHARACTER EXPRESSIONS

A character expression produces a value that is of type character. The forms of character expressions are:

1. character constant
2. character variable reference
3. character array element reference
4. any character expression enclosed in parentheses

There are no operators that result in character expressions.

Relational expressions compare the values of two arithmetic or two character expressions. An arithmetic value may not be compared with a character value. The result of a relational expression is of type logical.

Relational expressions can use any of the operators in Table 8-3 below to compare values.

Table 8-3. Relational Operators.

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

All of the operators are binary operators, appearing between their operands. There is no relative precedence or associativity among the relational operands since an expression of the form:

```
A .LT. B .NE. C
```

violates the type rules for operands. Relational expressions may only appear within logical expressions.

Relational expressions with arithmetic operands may have one operand of type integer and one of type real. In this case, the integer operand is converted to type real before the relational expression is evaluated.

Relational expressions with character operands compare the position of their operands in the ASCII collating sequence. An operand is less than another if it appears earlier in the collating sequence and so on. If operands of unequal length are compared, the shorter operand is considered as if it were extended to the length of the longer operand by the addition of spaces.

LOGICAL EXPRESSIONS

A logical expression produces a value that is of type logical. The simplest forms of logical expressions are:

1. logical constant
2. logical variable reference
3. logical array element reference
4. logical function reference
5. relational expression.

Other logical expressions are built up from the above simple forms using parentheses and the logical operators of Table 8-4 below.

Table 8-4. Logical Operators.

Operator	Representing Operation	Precedence
.NOT.	Negation	Highest
.AND.	Conjunction	Intermediate
.OR.	Inclusive disjunction	Lowest

The .AND. and .OR. operators are binary operators, appearing between their logical expression operands. The .NOT. operator is unary, preceding its operand. Operations of equal precedence are left-associative so, for example,

```
A .AND. B .AND. C
```

is equivalent to:

```
(A .AND. B) .AND. C.
```

As an example of the precedence rules:

```
.NOT. A .OR. B .AND. C
```

is interpreted the same as:

```
(.NOT. A) .OR. (B .AND. C).
```

Two .NOT. operators cannot be adjacent to each other, although:

A. .AND. .NOT. B

is an example of an allowable expression with two adjacent operators.

The meaning of the logical operators is their standard mathematical semantics, with .OR. being "nonexclusive," that is,

.TRUE. .OR. .TRUE.

evaluates to the value:

.TRUE.

PRECEDENCE OF OPERATORS

When arithmetic, relational, and logical operators appear in the same expression, their relative precedence is as shown in Table 8-5 below.

Table 8-5. Relative Precedence of Operator Classes.

Operator	Precedence
Arithmetic	Highest
Relational	Intermediate
Logical	Lowest

EVALUATION RULES AND RESTRICTIONS FOR EXPRESSIONS

Any variable, array element, or function referenced in an expression must be defined at the time of the reference. Integer variables must be defined with an arithmetic value, rather than a statement label value as set by an ASSIGN statement.

Certain arithmetic operations are prohibited, if not mathematically meaningful, such as dividing by 0. Other prohibited operations are raising a 0-valued operand to a 0 or negative power and raising a negative-valued operand to a power of type real.

SECTION 9
ASSIGNMENT STATEMENT

An assignment statement assigns a value to a variable or an array element. There are two kinds of assignment statements: computational and label.

COMPUTATIONAL ASSIGNMENT STATEMENT

The form of a computational assignment statement is:

var = expr

where:

var	is a variable or array element name.
expr	is an expression.

Execution of a computational assignment statement evaluates the expression and assigns the resulting value to the variable or array element appearing on the left. The type of the variable or array element and the expression must be compatible. They must both be either numeric, logical, or character, in which case the assignment statement is called an arithmetic, logical, or character assignment statement.

If the types of the elements of an arithmetic assignment statement are not identical, automatic conversion of the value of the expression to the type of the variable is done. The cnversion rules are given in Table 9-1 below. In this table, the most significant portion is the high order and the least significant is the low order.

Table 9-1. Type Conversion for Arithmetic Assignment Statements.

Variable or Array Element (V)	Integer Expression (E)	Real Expression (E)	Double-Precision Expression (E)
Integer	Assign E to V	Truncate E to integer and assign to V	Truncate E to integer and assign to V
Real	Append fraction (.0) to E and assign to V	Assign E to V	Assign most significant portion of E to V; least significant portion of E is rounded
Double Precision	Append fraction (.0) to E and assign to most significant portion of V; least significant portion of V is 0	Assign E to most significant portion of V; least significant portion of V is 0	Assign E to V

If the length of the expression does not match the size of the variable in a character assignment statement, it is adjusted to match. If the expression is shorter, it is padded with enough blanks on the right to make the sizes equal before the assignment takes place. If the expression is longer, characters on the right are truncated to make the sizes the same.

Logical expressions of any size can be assigned to logical variables of any size without effect on the value of the expression. However, integer, real, and double-precision expressions may not be assigned to logical variables, nor may logical expressions be assigned to integer, real, or double-precision variables.

NOTE: INTEGER*2 and INTEGER*4 are treated the same, except that INTEGER*4 may be assigned in accordance with its range of values.

LABEL ASSIGNMENT STATEMENT

The label assignment statement assigns the value of a format or statement label to an integer variable. The format of the statement is:

ASSIGN label TO var

where:

label	is a format label or statement label.
var	is an integer variable.

Execution of an ASSIGN statement sets the integer variable to the value of the label. The label can be either a format or a statement label, and it must appear in the same program unit as the ASSIGN statement. When used in an assigned GOTO statement, a variable must currently have the value of a statement label. When used as a format specifier in an input/output statement, a variable must have the value of a format statement label. The ASSIGN statement is the only way to assign the value of a label to a variable.

Note that the "value" of a label is not the same as the label number. For example, the value of:

LABEL 400

is not necessarily 400.

Also note that this makes the variable undefined as an integer and it cannot be used in an arithmetic expression until it has been redefined (by an ASSIGNMENT or READ statement) as such.

SECTION 10
CONTROL STATEMENT

Control statements control the order of execution of statements in FORTRAN. This Section describes the following control statements.

1. unconditional GOTO
2. computed GOTO
3. assigned GOTO
4. arithmetic IF
5. logical IF
6. block if then else
   a. block IF
   b. ELSEIF
   c. ELSE
   d. ENDIF
7. DO
8. CONTINUE
9. STOP
10. PAUSE
11. END

Two remaining statements, CALL and RETURN, control the order of execution of statements. Both are described in Section 13.

UNCONDITIONAL GOTO

The syntax for an unconditional GOTO statement is:

GOTO s

where:

s — is a statement label of an executable statement found in the same program unit as the GOTO statement.

The GOTO statement causes the next statement executed to be the statement labeled s. Jumping into a DO, IF, ELSEIF, or ELSE block from outside the block is not permitted. The sections that follow explain the different kinds of blocks. (A special feature, extended range DO loops, permits jumping into a DO block. See Section 14 for more information.)

```
C EXAMPLE OF UNCONDITIONAL GOTO
      GOTO 4022
4022 CONTINUE
```

COMPUTER GOTO

The syntax for a computed GOTO statement is:

GOTO (s [, s] ...) [,] i

where:

s	is a statement label of an executable statement found in the same program unit as the computed GOTO statement.
i	is an integer expression.

The same statement label may appear repeatedly in the list of labels.

The effect of the computed GOTO statement is as follows. Suppose there are n labels in a list of labels. If i 1 or i n, then the computed GOTO statement acts as if it were a CONTINUE statement; otherwise the next statement executed is the one labeled by the ith label in the list of labels. Jumping into a DO, IF, ELSEIF, or ELSE block from outside the block is not permitted. The sections that follow explain the kinds of blocks. (A special feature, extended range DO loops, permits jumping into a DO block. See Section 14 for more information.)

```
C EXAMPLE OF COMPUTED GOTO
    I = 1
    GOTO (10, 20) I
10 CONTINUE
20 CONTINUE
```

ASSIGNED GOTO

The syntax for an assigned GOTO statement is:

GOTO i[[,] (s [, s] ...)]

where:

i	is an integer variable name.
s	is a statement label of an executable statement found in the same program unit as the assigned GOTO statement.

The same statement label may appear repeatedly in the list of labels. When the assigned GOTO statement is executed, i must have been assigned the label of an executable statement found in the same program unit as the assigned GOTO statement.

The assigned GOTO statement causes the next statement executed to be the statement labeled by the label last assigned to i. If the optional list of labels is present, a run-time error is generated if the label last assigned to i is not among those listed. Jumping into a DO, IF, ELSEIF, or ELSE block from outside the block is not permitted. The sections that follow explain the various kinds of blocks. (A special feature, extended range DO loops, permits jumping into a DO block. See Section 14 for more information.)

```
C EXAMPLE OF ASSIGNED GOTO
    ASSIGN 10 TO I
    GOTO I
10 CONTINUE
```

ARITHMETIC IF

The syntax for an arithmetic IF statement is:

IF (e) s1, s2, s3

where:

e	is an integer or real expression.
s1, s2, s3	are statement labels of executable statements found in the same program unit as the arithmetic IF statement.

The same statement label may appear more than once among the three labels.

The arithmetic IF statement causes evaluation of the expression and selection of a label based on the value of the expression. Label s1 is selected if the value of e is less than 0, s2 if the value of e equals 0, and s3 if the value of e exceeds 0. The next statement executed is the statement labeled by the selected label. Jumping into a DO, IF, ELSEIF, or ELSE block from outside the block is not permitted. The sections that follow explain the various kinds of blocks. (A special feature, extended range DO loops, permits jumping into a DO block. See Section 14 for more information.)

```
C EXAMPLE OF ARITHMETIC IF
    I = 0
    IF (I) 10, 20, 30
10 CONTINUE
20 CONTINUE
30 CONTINUE
```

LOGICAL IF

The syntax for a logical IF statement is:

IF (e) st

where:

e	is a logical expression.
st	is any executable statement except a DO, block IF, ELSEIF, ELSE, ENDIF, END, or another logical IF statement.

The logical IF statement causes the logical expression to be evaluated and, if the value of that expression is true, then the st statement is executed. If the expression evaluates to false, the st statement is not executed and the execution sequence continues as if a CONTINUE statement were encountered.

```
C EXAMPLE OF LOGICAL IF
    IF (I .EQ. 0) J = 2
    IF (X .GT. 2.3) GOTO 100
100 CONTINUE
```

BLOCK IF THEN ELSE

The subsections on Block IF, ELSEIF, ELSE, and ENDIF below describe the block IF statement and the various statements associated with it. These statements are new to FORTRAN 77 and improve the readability of FORTRAN programs. As an overview of these subsections, the following three code skeletons illustrate the basic concepts.

Skeleton 1 - Simple Block IF that skips a group of statements if the expression is false:

```
IF(I.LT.10) THEN

        . Some statements executed only if I.LT.10

ENDIF
```

Skeleton 2 - Block IF with a series of ELSEIF statements:

```
IF(J.GT.1000)THEN
    .
        . Some statements executed only if J.GT.1000
            .
ELSEIF(J.GT.100)THEN
    .
        . Some statements executed only if J.G.T100
            . and J.LE.1000
ELSEIF(J.GT.10)THEN
    .
        . Some statements executed only if J.GT.10
            . and J.LE.1000 and J.LE.100
ELSE
    .
        . Some statements executed only if none of
            . above conditions are true
ENDIF
```

Skeleton 3 - Illustrates that the constructs can be nested and that an ELSE statement can follow a block IF without intervening ELSEIF statements (indentation solely to enhance readability):

```
IF(I.LT.100)THEN
      .
          . Some statements executed only if I.LT.100
              .
                  IF(J.LT.10)THEN
                      .
                          . Some statements executed only if
                              . I.LT.100 and J.LT.10
                  ENDIF
      .
          . Some statements executed only if I.LT.100
              .
ELSE
      .
          . Some statements executed only if I.GE.100
              .
                  IF(J.LT.10)THEN
                      .
                          . Some statements executed only if
                              . I.GE.100 and J.LT.10
                  ENDIF
      .
          . Some statements executed only if I.GE.100
              .
ENDIF
```

To understand in detail the block IF and associated statements, the concept of an IF-level is introduced. For any statement, its IF-level is:

n1 - n2

where:

n1 is the number of block IF statements from the beginning of the program unit that the statement is in - up to and including that statement.

n2 is the number of ENDIF statements from the beginning of the program unit up to, but not including, that statement.

The IF-level of every statement must be greater than or equal to 0 and the IF-level of every block IF, ELSEIF, ELSE, and ENDIF must be greater than 0.

Finally, the IF-level of every END statement must be 0. The IF-level defines the nesting rules for the block IF and associated statements and defines the extent of IF, ELSEIF, and ELSE blocks.

Block IF

The syntax for a block IF statement is:

IF (e) THEN

where:

e	is a logical expression.

The IF block associated with this block IF statement consists of all the executable statements (there may be none) that appear following this statement up to, but not including, the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as this block IF statement (the IF-level defines the notion of "matching" ELSEIF, ELSE, or ENDIF).

The block IF statement causes the expression to be evaluated. If the expression evaluates to true and there is at least one statement in the IF block, the next statement executed is the first statement of the IF block. Following the execution of the last statement in the IF block, the next statement to be executed is the next ENDIF statement at the same IF-level as this block IF statement. If the expression in this block IF statement evaluates to true and the IF block has no executable statements, the next statement executed is the next ENDIF statement at the same IF-level as the block IF statement. If the expression evaluates to false, the next statement executed is the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as the block IF statement.

Transfer of control into an IF block from outside that block is not permitted.

ELSEIF

The syntax of an ELSEIF statement is:

ELSEIF (e) THEN

where:

e	is a logical expression.

The ELSEIF block associated with an ELSEIF statement consists of all the executable statements (there may be none) that follow the ELSEIF statement up to, but not including, the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as this ELSEIF statement.

The ELSEIF statement causes the evaluation of the expression. If its value is true and there is at least one statement in the ELSEIF block, the next statement executed is the first statement of the ELSEIF block. Following the execution of the last statement in the ELSEIF block, the next statement to be executed is the next ENDIF statement at the same IF-level as this ELSEIF statement. If the expression in this ELSEIF statement evaluates to true and the ELSEIF block has no executable statements, the next statement executed is the next ENDIF statement at the same IF-level as the ELSEIF statement. If the expression evaluates to false, the next statement executed is the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as the block IF statement.

Transfer of control into an IF block from outside that block is not permitted.

ELSE

The syntax of an ELSE statement is:

```
ELSE
```

The ELSE block associated with an ELSE statement consists of all of the executable statements (there may be none) that follow the ELSE statement up to, but not including, the next ENDIF statement that has the same IF-level as this ELSE statement. The "matching" ENDIF statement must appear before any intervening ELSE or ELSEIF statements of the same IF-level.

The execution of an ELSE statement has no effect.

Transfer of control into an ELSE block from outside that block is not permitted.

ENDIF

The syntax of an ENDIF statement is:

```
ENDIF
```

An ENDIF statement is required to "match" every block IF statement in a program unit in order to specify which statements are in a particular block IF statement.

The execution of an ENDIF statement has no effect.

DO

The syntax of a DO statement is:

DO s [,] i=e1, e2 [, e3]

where:

s	is a statement label of an executable statement.
i	is an integer variable.
e1, e2, e3	are integer expressions.

The label must follow this DO statement and be contained in the same program unit. The statement labeled by s is called the terminal statement of the DO loop. It must not be an unconditional GOTO, assigned GOTO, arithmetic IF, block IF, ELSEIF, ELSE, ENDIF, RETURN, STOP, END, or DO statement. If the terminal statement is a logical IF, it may contain any executable statement except those not permitted inside a logical IF statement.

A DO loop has a "range," beginning with the statement that follows the DO statement and ending with (and including) the terminal statement of the DO loop.

If a DO statement appears in the range of another DO loop, its range must be entirely contained within the range of the enclosing DO Loop, although the loops may share a terminal statement.

If a DO statement appears within an IF, ELSEIF, or ELSE block, the range of the associated DO loop must be entirely contained in the particular block.

If a block IF statement appears within the range of a DO loop, its associated ENDIF statement must also appear within the range of that DO loop.

The DO variable, i, may not be assigned to the statements within the range of the DO loop associated with it. Jumping into the range of a DO loop from outside its range is not permitted. (However, there is a special feature, added for compatibility with earlier versions of FORTRAN, that permits "extended range" DO loops. See Section 14 for more information.)

The execution of a DO statement causes the following steps.

1. The expressions e1, e2, and e3 are evaluated. If e3 is not present, it is as if e3 evaluated to 1 (e3 must not evaluate to 0).

2. The DO variable, i, is set to the value of e1.

> NOTE
>
> Note that the prohibition on assigning INTEGER*4 values to INTEGER*2 variables applies here. Therefore if i is INTEGER*2, e1, e2, and e3 (if it exists) must also be expressions of type INTEGER*2.

3. The iteration count for the loop is:

   MAX0(((e2-e1+e3)/e3),0)

   which may be 0 if either:

   e1 > e2 and e3 > 0

   or:

   e1 < e2 and e3 < 0

   (However if the DO66 directive is in effect, the iteration count is at least 1. See Section 14 for more information.)

4. The iteration count is tested, and if it exceeds 0, the statements in the range of the DO loop are executed.

Following the execution of the terminal statement of a DO loop, the following steps occur.

1. The value of the DO variable, i, is incremented by the value of e3 that was computed when the DO statement was executed.

2. The iteration count is decremented by 1.

3. The iteration count is tested, and if it exceed 0, the statements in the range of the DO loop are executed again.

The value of the DO variable is well-defined regardless of whether the DO loop exits because the iteration count become 0 or because of a transfer of control out of the DO loop or RETURN statement.

The following is an example of the final value of a DO variable.

```
C This program fragment displays the numbers
C 1 to 11 on the screen

      DO 200 I=1,10
200 WRITE(*,'(I5)')I
      WRITE(*,'(I5)')I

C EXAMPLE OF DO STATEMENT
C INITIALIZE A 20-ELEMENT REAL ARRAY
      DIMENSION ARRAY(20)
      DO 1 I = 1, 20
1     ARRAY(I) = 0.0
C PERFORM A FUNCTION 11 TIMES
      DO 2, I = -30, -60, -3
      J = I/3
      J = -9 - J
      ARRAY(J) = MYFUNC(I)
2     CONTINUE
```

CONTINUE

The syntax of a CONTINUE statement is:

```
CONTINUE
```

The primary use for the CONTINUE statement is a convenient statement to label, particularly as the terminal statement in a DO loop.

The execution of a CONTINUE statement has no effect.

```
C EXAMPLE OF CONTINUE STATEMENT
      DO 10, I = 1, 10
      IARRAY(I) = 0
10    CONTINUE
```

STOP

The syntax of a STOP statement is:

STOP [n]

where:

n	is either a character constants or a string of not more than five digits.

The STOP statement causes the program to terminate. The argument, n, if present, is displayed on the screen upon termination.

```
C EXAMPLE OF STOP STATEMENT
    IF (IERROR .EQ. 0) GOTO 200
    STOP 'ERROR DETECTED'
200 CONTINUE
```

PAUSE

The syntax of a PAUSE statement is:

PAUSE [n]

where:

n	is either a character constant or a string of not more than five digits.

The PAUSE statement causes the program to be suspended pending an indication from the keyboard that it is to continue. The argument, n, if present, is displayed on the screen as part of the prompt requesting input from the keyboard. To continue execution of the program, press the space bar or RETURN key. Execution resumes as if a CONTINUE statement were executed.

```
C EXAMPLE OF PAUSE STATEMENT
    IF (IWARN .EQ. 0) GOTO 300
    PAUSE 'WARNING: IWARN IS NONZERO'
300 CONTINUE
```

The syntax of an END statement is:

```
END
```

Unlike other statements, an END statement must wholly appear on an initial line and contain no continuation lines. No other FORTRAN statement, such as the ENDIF statement, may have an initial line that appears to be an END statement.

The END statement in a subprogram has the same effect as a RETURN statement. In the main program, it terminates execution of the program. The END statement must appear as the last statement in every program unit.

```
C EXAMPLE OF END STATEMENT
C END STATEMENT MUST BE LAST STATEMENT
C IN A PROGRAM
      PROGRAM MYPROG
      WRITE(*, 10H HI WORLD!)
      END
```

SECTION 11
I/O SYSTEM

Sections 11 and 12 describe the FORTRAN I/O system. Section 11 describes the basic FORTRAN I/O concepts and statements and Section 12 the FORMAT statement. The major subsections of this section are:

- Overview - Provides an overview of the FORTRAN file system. Defines the basic concepts of I/O records, I/O units, and the various kinds of file access available.
- Concepts and Limitations - Relates the definitions made in the Overview to accomplishing various tasks using he most common forms of files and I/O statements. This Section also gives a complete program illustrating these operations. There is also a general discussion of I/O system limitations.
- Statements - Presents the I/O system statements, except FORMAT.

OVERVIEW

You need to be familiar with the terms and concepts related to the structure of the FORTRAN I/O system to understand the I/O statements. However, most I/O tasks can be accomplished without a complete understanding of this material; you can skip to "Concepts and Limitations" below on first reading and use this subsection for reference.

Records

The building block of the FORTRAN file system is the record. A record is a sequence of characters or values. There are three kinds of records: formatted, unformatted, and endfile.

A formatted record is a sequence of characters terminated by the character value of the RETURN key (hexadecimal value OAh). Formatted records are interpreted on input consistently with the way the Operating System and the Editor interpret characters.

An unformatted record is a sequence of values, with no system alteration or interpretation; no physical representation exists for the end of record.

The FORTRAN file system simulates a virtual endfile record after the last record in a file, although there is no corresponding real record.

Files

A file is a sequence of records. Files are either external or internal.

An external file is a file on a device or a device itself. An internal file is a character variable that serves as the source or destination of some I/O action. From this point on, both internal FORTRAN files and the files known to the Operating System are referred to simply as files, with context determining meaning. (The OPEN statement provides the linkage between the two notions of files and, in most cases, when the two notions coincide, the ambiguity disappears after opening a file.

File Properties

A FORTRAN file has these properties: name, position, formatted or unformatted, and sequential or direct access.

File Name. A file can have a name. If present, a name is a character string identical to the name by which it is known to the B 20 Operating System services file management system. (File naming conventions are described in the Executive Manual.)

File Position. The position property of a file is usually set by the previous I/O operation. A file has an initial point, terminal point, current record, preceding record, and next record.

It is possible to be between records in a file, in which case the next record is the successor to the previous record and there is no current record.

Opening a sequential file for writing positions the file at its beginning and discards all old data in the file. The file position after a sequential write is at the end of the file, but not beyond the endfile record. Executing the ENDFILE statement positions the file beyond the endfile record, as does a READ statement executed at the end of the file (but not beyond the endfile record). Reading an endfile record can be trapped by the user using the END= option in a READ statement.

Formatted and Unformatted Files. An external file is opened as either formatted or unformatted. All internal files are formatted. Formatted files consist entirely of formatted records and unformatted files consist entirely of unformatted records.

Sequential and Direct Access Properties. An external file is opened as either sequential or direct. Files contain records with order determined by the order in which the records were written (the normal sequential order). These files must not be read or written using the REC= option which specifies a position for direct access I/O. The Operating System attempts to extend sequential access files if a record is written beyond the old terminating file boundary; the success of this operation depends on available space on the physical device.

Direct access files can be read or written in any order (they are random access files). Recores are numbered sequentially, with the first record numbered 1. All records have the same length, specified when the file is opened, and each record has a unique record number, specified when the record is written.

It is possible to write records out of order, including, for example, writing record 9, 5, and 11 in that order without the records in between. It is not possible to delete a record once written but a record can be overwritten with a new value. An error occurs when a record is read from a direct access file that has not been written. Direct access files must reside on disk. The Operating System attempts to extend direct access files if a record is written beyond the old terminating file boundary; the success of this depends on available space on the physical device.

Internal Files

Internal files provide a mechanism for using the formatting capabilities of the I/O system to convert values to and from their external character representations, within the FORTRAN internal memory structures. That is, reading a character variable converts the character values into numeric, logical, or character values and writing a character variable allows values to be converted into their (external) character representation.

Special Properties. An internal file is a character variable or character array element. The file has exactly one record, which has the same length as the character variable or character array element. If less than the entire record is written, the remaining portion of the record is filled with blanks. The file position is always at the beginning of the file prior to I/O statement execution. Only formatted, sequential I/O is permitted to internal files and only the I/O statements READ and WRITE may specify an internal file.

Units

A unit is a means of referring to a file. A unit specified in an I/O statement is either an external unit or an internal file specifier.

An external unit specifier is either an integer expression, which evaluates to a nonnegative value, or the character *, which stands for the screen (for writing) and the keyboard (for reading). In most cases, an external unit specifier value is bound to a physical device (or files resident on that device) by name using the OPEN statement. Once this binding of value to system file name occurs, FORTRAN I/O statements specify the unit number as a means of referring to the appropriate external entity. Once opened, the external unit specifier value is uniquely associated with a particular external entity until an explicit CLOSE occurs or until the program terminates. The only exception to these binding rules is that the unit value 0 is initially associated with the keyboard for reading and the screen for writing and no explicit OPEN is necessary. The character * is interpreted by the FORTRAN file system as specifying unit 0.

An internal file specifier is a character variable or character array element that directly specifies an internal file.

FORTRAN provides a rich combination of possible file structures. However, two kinds of files suffice for most applications: * files, and explicitly opened external, sequential, formatted files.

* Files

* represents the keyboard and screen, a sequential, formatted file, also known as unit 0. This particular unit has the special properties that an entire line, terminated by the RETURN key, must be entered when reading from it, and the BACKSPACE and DELETE keys familiar to the system user serve their normal functions. Note that reading from any other unit does not have these properties, even though that unit is bound to the keyboard by an explicit OPEN statement.

Explicitly Opened External, Sequential, Formatted Files

These files are bound to a system file by name in an OPEN statement.

Example

This example program uses the two kinds of files discussed above for reading and writing. The I/O statements are explained in detail in the following subsection.

```
C Copy a file with three columns of integers,
C each 7 column wide, from a file whose name
C is input by the user to another file named
C OUT.TEXT, reversing the positions of the
C first and second columns.
      PROGRAM COLSWP
      CHARACTER*64 FNAME, MSG1
      DATA MSG1/'Done'/
C Prompt to the screen by writing to *.
      WRITE(*,900)
900   FORMAT('INPUT FILE NAME-'/)
C Read the file name from the keyboard by
C reading from *.
      READ(*,910) FNAME
910   FORMAT(A)
C use unit 3 for input; any unit number except
C 0 will do.
      OPEN(3,FILE=FNAME)
C Use unit 4 for output; any unit number except
C 0 and 3 will do.
      OPEN(4,FILE='OUT.TEXT',STATUS='NEW')
```

```
C Read and write until end of file.
100 READ(3,920,END=200)I,J,K
    WRITE(4,920)J,I,K
920 FORMAT(317)
    GOTO 100
200 WRITE(*,910)MSG1
    END
```

Less Commonly Used File Operations

The less commonly used file structures are appropriate for certain classes of applications. A very general indication of the intended usages for them follows.

If the I/O is to be random access, such as in maintaining a data base, direct access files are probably necessary, If the data is to be written by FORTRAN and reread by FORTRAN, unformatted files are more efficient in I/O overhead. The combination of direct and unformatted is ideal for a data base to be created, maintained, and accessed exclusively by FORTRAN.

If the data must be transferred without any system interpretation, especially if all 256 possible bytes are to be transferred, unformatted I/O is necessary. An example of a usage of unformatted I/O would be in the control of a device that has a single-byte, binary interface. Formatted I/O would, in this example, interpret certain characters, such as the ASCII representation for RETURN, and fail to pass them through to the program unaltered.

Internal files are not I/O in the conventional sense but rather provide certain character string operations and conversions within a standard mechanism.

A file opened in FORTRAN is either "old" or "new," but there is no concept of "opened for reading" as distinguished from "opened for writing." Therefore, you can open "old" (existing) files and write to them, with the effect of overwriting them. Similarly, you can alternately write and read to the same file (providing that you avoid reading beyond the end of the file, or reading unwritten records in a direct file).

A write to a sequential file effectively deletes any records that existed beyond the newly written record. Normally, when a device (such as the keyboard or printer) is opened as a file, it makes no difference whether it is opened as "old" or "new." With disk files, however, opening "new" creates a new file. If that file is closed or if the program terminates without doing a CLOSE on that file, a permanent file is created with the name given when the file was opened. If a previous file existed with the same name, it is overwritten.

Limitations

Direct Files/Direct Device Association. There are two kinds of devices: sequential and direct. The files associated with sequential devices are streams of characters, with no explicit motion allowed except reading and/or writing. The keyboard, screen, and printer are examples of sequential devices. Direct devices, such as disks, have the additional operation of seeking a specific location. They can be accessed either sequentially or randomly, and thus can support direct files. The FORTRAN I/O system does not allow direct files on sequential devices.

BACKSPACE/Sequential Device Association. The FORTRAN I/O system disallows backspacing a file on a sequential device (see below).

BACKSPACE/Unformatted Sequential File Association. There is no indication in an unformatted sequential file of record boundaries, therefore BACKSPACE on such files is defined as backing up by one byte. Direct files contain records of fixed, specified length, so it is possible to backspace by records on direct unformatted files.

Side Effects of Functions Called in I/O Statements. During execution of any I/O statement, evaluation of an expression may cause a function to be called. That function call must not cause any I/O statement to be executed.

I/O STATEMENTS

This subsection describes seven I/O statements: OPEN, CLOSE, READ, WRITE, BACKSPACE, ENDFILE, and REWIND.

In addition, an I/O intrinsic function, EOF (see Section 13), returns a logical value indicating whether the file associated with the unit specifier passed to it is at the end of the file. A familiarity with the FORTRAN file system is assumed.

Elements of I/O Statements

The various I/O statements take certain parameters and arguments that specify sources and destinations of data transfer, as well as other facets of the I/O operation. The abbreviations used in this subsection are the unit specifier (u), format specifier (f), and input/output list (iolist), defined below.

The unit specifier, u, can take one of the following forms in an I/O statement.

* - refers to the keyboard or screen.

integer expression - refers to an external file with a unit number equal to the value of the expression (* is unit number 0).

name of a character variable or character array element - refers to the internal file specified by the value of the variable or array element.

The format specifier, f, can take one of the following forms in an I/O statement.

statement label - refers to the FORMAT statement labeled by that label.

Integer variable name - refers to the FORTRAN label assigned to that integer variable using the ASSIGN statement.

character expression - the format specified is the current value of the character expression provided as the format specifier.

The input/output list (iolist) specifies the entities whose values are transferred by READ and WRITE statements. An iolist, a possibly empty list separated by commas, consists of input or output entities and implied DO lists.

Input Entities and Output Entities. An input entity can be specified in the iolist of a READ statement and an output entity in the iolist of a WRITE statement. The entity is either a variable name, an array element name, or an array name. An array name is a means of specifying all of the elements of the array in memory sequence order.

An output entity can also be any other expression not beginning with the character "(", to distinguish implied DO lists from expressions.

Note that the expression:

(A+B)*(C+D)

can be written as:

+(A+B)*(C+D)

to distinguish it from an implied DO list.

Implied DO Lists. Implied DO lists can be specified as items in the I/O list of READ and WRITE statements and have the form:

(iolist, i = e1, e2 [, e3])

where:

iolist	is as above (including nested implied DO lists).
i, e1, e2, e3	are as defined for the DO statement. That is, i is an integer variable, e1, e2, and e3 are integer expressions and, if i is INTEGER*2, then e1, e2, and e3 must be INTEGER*2.

In a READ statement, the DO variable i (or an associated entity) must not appear as an input list item in the embedded iolist, but may have been read in the same READ statement outside of the implied DO list. The embedded iolist is effectively repeated for each iteration of i with appropriate substitution of values for the DO variable i.

In the case of nested implied DO loops, the innermost (most deeply nested) loop is always executed fastest.

OPEN Statement

OPEN(u,FILE=fname,STATUS=st,ACCESS=ac,
FORM=fm,RECL=rl)

where:

u — is a unit specifier (see "Elements of I/O Statements" above). It is required, and must appear as the first argument. It must not be an internal unit specifier.

fname — is a character expression. This is a required parameter and must appear as the second argument.

All arguments after fname are optional and can appear in any order. The options are character constants with optional trailing blanks (except RECL=).

st — is 'OLD' (the default) or 'New'. 'OLD' is for reading or writing existing files. 'NEW' is for writing new files.

> NOTE
>
> For sequential files, STATUS='OLD' and STATUS='NEW' correspond to the B 20 Operating System services file modes mode read and mode write; for direct files, the open mode is always mode modify and STATUS has no effect.

ac — is 'SEQUENTIAL' (the default) or 'DIRECT'.

fm — is 'FORMATTED' (the default) or 'UNFORMATTED'.

rl — is the record length, an integer expression. This argument to OPEN is for DIRECT access files only, for which it is required.

The OPEN statement binds a unit number with an external device or file on an external device by specifying its file name. Binding unit 0 to a file has no effect; unit 0 is permanently connected to the keyboard and screen. If the file is to be direct, the RECL=rl option specifies the length of the records in that file.

Example program fragment 1:

```
      CHARACTER*26 MSG2
      DATA  MSG2/'Specify output File name - '/
C Prompt user for a file name.
      WRITE(*,' (A )')MSG2
C Presume that FNAME is specified to be
C CHARACTER*64. Read the file name from the keyboard.
      READ(*,'(A)') FNAME
C Open the file as formatted sequential as unit 7. Note that the
C ACCESS specified need not have appeared since it is the
C default.
      OPEN(7,FILE=FNAME,ACCESS='SEQUENTIAL',STATUS='NEW');
```

Example program fragment 2:

```
C Open an existing file created by the Editor
C called DATA3.TEXT as unit 3.
      OPEN(3,FILE='DATA3.TEXT')
```

CLOSE Statement

CLOSE(u,STATUS=st)

where:

u — is a unit specifier (see "Elements of I/O Statements" above). It is required, and must appear as the first argument. It must not be an internal unit specifier.

st — is 'KEEP' or 'DELETE', an optional argument that applies only to files opened NEW. The default is 'KEEP'. This option is a character constant.

CLOSE disconnects the unit specified and prevents subsequent I/O from being directed to that unit (unless the same unit number is reopened, possibly bound to a different file or device). Files opened NEW are temporary files and are discarded if STATUS='DELETE' is specified. Normal termination of a FORTRAN program automatically closes all open files as if CLOSE with STATUS='KEEP' was specified. CLOSE for unit 0 has no effect, since the CLOSE operation is not meaningful for keyboard and screen.

Example program fragment:

```
C Close the file opened in OPEN example,
C discarding the file.
      CLOSE(7,STATUS='DELETE')
```

READ Statement

READ(u,f,REC=rn,END=s1,ERR=s)iolist

where:

u — is a unit specifier (see "Elements of I/O Statements" above). It is required, and must appear as the first argument.

f — is required for formatted read as the second argument, and must not appear for unformatted read.

rn — is specified for direct access only, otherwise an error results. It is a positive integer expression. It positions to record number rn. If REC=rn is omitted for a direct access file, reading continues from the current position in the file.

s1 — is an optional statement label. If not present, reading the end of the file results in a run-time error. If present, encountering an end of file condition results in the transfer to the executable statement labeled s1, which must be in the same program unit as the READ statement.

s — is an optional statement label. If it is not present, I/O errors result in run-time errors. If it is present, I/O errors cause control to transfer to the executable statement labeled s.

The READ statement sets the items in iolist (assuming that no end of file or error condition occurs). If the read is internal, the specified character variable or character array element is the source of the input. Otherwise, the external unit is the source.

Example program fragment:

```
C Need a two dimensional array for the example.
      DIMENSION IA(10,20)
C Read in the bounds for the array. These
C bounds should be less than 10 and 20
C respectively. Then read in the array in
C nested implied DO lists with input format of
C 8 columns of width 5 each.
      READ(3,990)I,J,((IA(I,J),J=1,J),I=1,I,1)
990 FORMAT(2I5/,(8I5))
```

WRITE Statement

WRITE(u,f,ERR=s,REC=rn)iolist

where:

u — is a unit specifier (see "Elements of I/O Statements" above). It is required, and must appear as the first argument.

f — is required for formatted write as the second argument, and must not appear for unformatted write.

s — is an optional statement label. If it is not present, I/O errors result in run-time errors. If it is present, I/O errors cause control to transfer to the executable statement labeled s.

rn — is specified for direct access only, otherwise an error results. It is a positive integer expression. It positions to record number rn for this WRITE. If REC=rn is omitted for a direct access file, writing continues from the current position in the file.

The WRITE Statement tranfers the iolist items to the unit specified. If the write is internal, the character variable or character array element specified is the destination of the output, otherwise the external unit is the destination.

Example program fragment:

```
C Display message: "One = 1, Two = 2,
C Three = 3" on the screen, not doing things in
C the simplest way!
      WRITE(*,980) 'One=',1,1+1, 'ee=',+(1+1+1)
980   FORMAT(A,I1',Two=',1X,I1,',Thr',A,I1)
```

BACKSPACE Statement

BACKSPACE u

where

u — is an internal unit specifier.

See the "Limitations" subsection above for more information.

BACKSPACE causes the file connected to the specified unit to be positioned before the preceding record. If there is not preceding record, the file position is not changed. Note that if the preceding record is the endfile record, the file becomes psoitioned before the endfile record. If the file position is in the middle of the record, BACKSPACE positions to the start of that record.

ENDFILE Statement

ENDFILE u

where

u is an internal unit specifier.

ENDFILE "writes" and end of file record as the next record of the file connected to the specified unit. The file is then psoitioned after the end of the file record, so further sequential data transfer is prohibited until either a BACKSPACE or REWIND is executed. An ENDFILE on a direct access file makes all records written beyond the position of the new end of file disappear.

REWIND Statement

REWIND u

where:

u is an internal unit specifier.

Execution of a REWIND statement causes the file associated with the specified unit to be positioned at its initial point.

Carriage Control

The first character of every record transferred to an external device, such as the video or a printer, is not printed. Instead, it is interpreted as a carriage control character. The FORTRAN I/O system recognizes certain characters as carriage control characters. Table 11-1 below lists these characters and their effects.

Table 11-1. Carriage Control Characters.

Character	Effect
space	Advances one line.
0	Advances two lines.
1	Advances to top of next page.
+ (plus)	Does not advance (allows overprinting).

Any character than those listed in Table 11-1 is treated as a space and is deleted from the print line.

NOTE

NOTE: Note that if you accidentally omit the carriage control character, the first character of the record is not printed.

SECTION 12
FORMATTED I/O AND THE FORMAT STATEMENT

FORMAT SPECIFICATIONS AND THE FORMAT STATEMENT

If a READ or WRITE statement specifies a format, it is considered a formatted, rather than an unformatted, I/O statement. Such a format can be specified in one of three ways, as explained in Section 11. Two ways refer to FORMAT statements and one is an immediate format in the form of a character expression containing the format itself. The following are all valid and equivalent means of specifying a format.

```
      WRITE (*,990) I,J,K
990   FORMAT(2I5,I3)

      ASSIGN 990 to IFMT
990   FORMAT(2I5,I3)
      WRITE(*,IFMT) I,J,K

      WRITE(*,'(2I5,I3)') I,J,K

      CHARACTER*8 FMTCH
      FMTCH = '(2I5,I3)
      WRITE(*,FMTCH) I,J,K
```

The format specification itself must begin with "(", possibly following initial blank characters, and ending with a matching ")". Characters beyond the matching ")" are ignored.

FORMAT statements must be labeled, and like all nonexecutable statements, may not be the target of a branching operation.

Between the initial "(" and terminating ")" is a list of items, separated by commas, each of which is one of:

[r]ed	repeatable edit descriptors.
ned	nonrepeatable edit descriptors.
[r]fs	a nested format specification. At most three levels of nested parentheses are permitted within the outermost level.

where:

[r]	is an optionally present, nonzero, unsigned, integer constant called a repeat specification.

The comma separating two list items may be omitted if the resulting format specification is still unambiguous, such as after a P edit descriptor or before or after the / edit descriptor.

The repeatable edit descriptors, described below, are:

Iw
Fw.d
Ew.d
Ew.dEe
Lw
A
Aw

where:

I, F, E, L, A	indicate the manner of editing.
w, e	are nonzero, unsigned, integer constants.
d	is an unsigned integer constant.

The nonrepeatable edit descriptors, described below, are:

'xxxx'	character constants of any length (see special rules below).
nHxxxx	another means of specifying character constants (see rules below).
nX	
/	
\	
kP	
BN	
BZ	

where:

', H, X, /, \ , P, BN, BZ	indicate the manner of editing.
x	is any ASCII character.
n	is a nonzero, unsigned, integer constant.
k	is an optionally signed integer constant.

INTERACTION BETWEEN FORMAT SPECIFICATION AND INPUT/OUTPUT LIST (iolist)

If an iolist contains at least one item, at least one repeatable edit descriptor must exist in the format specification. In particular, the empty edit specification, ( ), can be used only if no items are specified in the iolist (in which case the only action caused by the I/O statement is the implicit record-skipping action associated with formats). Each item in the iolist is associated with a repeatable edit descriptor during the I/O statement execution in turn. In contrast, the remaining format control items interact directly with the record and do not become associated with items in the iolist.

The items in a format specification are interpreted from left to right. Repeatable edit descriptors act as if they were present r times (if omitted, r is treated as a repeat factor of 1). Similarly, a nested format specification is treated as if its items appeared r times.

The formatted I/O process proceeds as follows. The "format controller" scans the format items in the order indicated above. When a repeatable edit descriptor is encountered either:

- a corresponding item appears in the iolist, in which case the item and the edit descriptor are associated, and I/O of that item proceeds under format control of the edit descriptor, or
- no corresponding item appears in the iolist, in which case the "format controller" terminates I/O.

If the format controller encounters the matching final ")" of the format specification and there are no further items in the iolist, the "format controller" terminates I/O. If, however, there are further items in the iolist, the file is positioned at the beginning of the next record and the "format controller" continues by rescanning the format starting at the beginning of the format specification terminated by the last preceding right parenthesis.

If there is no such preceding right parenthesis, the "format controller" rescans the format from the beginning. Within the portion of the format rescanned, there must be at least one repeatable edit descriptor.

If the rescan of the format specification begins with a repeated nested format specification, the repeat factor indicates the number of times to repeat that nested format specification. The rescan does not change the previously set scale factor or BN or BZ blank control in effect. When the "format controller" terminates, the remaining characters or an input record are skipped or an end of record is written on output, except as noted under the edit descriptor below.

EDIT DESCRIPTORS

Nonrepeatable

'xxxx' (Apostrophe Editing). The apostrophe edit descriptor has the form of a character constant. Embedded blanks are significant and double ' ' are interpreted as a single '. Apostrophe editing cannot be used for input (READ) as it causes the character constant to be transmitted to the output unit. For an example, see "H (Hollerith Editing)" below.

H (Hollerith Editing). The nH edit descriptor causes the following n characters, with blanks counted as significant, to be transmitted to the output unit. Hollerith editing cannot be used for input (READ).

Examples of Apostrophe and Hollerith editing:

```
C Each write outputs characters between the
C slashes: /ABC'DEF/
      WRITE (*,970)
970 FORMAT ('ABC' 'DEF')
      WRITE (*,'("ABC'' ''DEF")')
      WRITE (*,'(7HABC' 'DEF)')
      WRITE (*,960)
960 FORMAT (7HABC'DEF)
```

X (Positional Editing). On input (READ), the n^X edit descriptor causes the file position to advance n characters thus the next n characters are skipped. On output (WRITE), the nX edit descriptor causes n blanks to be written, providing that further writing to the record occurs; otherwise, the nX descriptor results in no operation.

/ (Slash Editing). The slash indicates the end of data transfer on the current record. On input, the file is positioned to the beginning of the next record. On output, an end of record is written and the file is positioned to write on the beginning of the next record.

\(Backslash Editing). Normally when the "format controller" terminates, the end of data transmission on the current record occurs. If the last edit descriptor encountered by the "format controller" is , this automatic end of record is inhibited. This allows subsequent I/O statements to continue reading (or writing) out of (or into) the same record. The most common use for this mechanism is to prompt to the screen and read a response off the same line as in:

```
WRITE (*,'(A \ )') 'Input an integer    '
READ (*,'(BN,I6)') I
```

The \ edit descriptor does not inhibit the automatic end of record generated when reading from the * unit. Input from the keyboard must always be terminated by the RETURN key. This permits the backspace character and the DELETE key to function properly.

P (Scale Factor Editing). The kP edit descriptor sets the scale factor for subsequent F and E edit descriptors until another kP edit descriptor is encountered. At the start of each I/O statement, the scale factor is initialized to 0. The scale factor affects format editing in the following ways.

- On input, with F and E editing, providing that no explicit exponent exists in the field, and F output editing, the externally represented number equals the internally represented number multiplied by 10**k.
- On input, with F and E editing, the scale factor has no effect if there is an explicit exponent in the input field.
- On output, with E editing, the real part of the quantity is output multiplied by 10**k and the exponent is reduced by k (effectively altering the column position of the decimal point but not the value output).

BN and BZ (Blank Interpretation). These edit descriptors specify the interpretation of blanks in numeric input fields. The default, BZ, is set at the start of each I/O statement. This makes blanks, other than leading blanks, identical to zeros. If a BN edit descriptor is processed by the "format controller," blanks in subsequent input fields are ignored unless, and until, a BZ edit descriptor is processed. The effect of ignoring blanks is to take all the nonblank characters in the input field, and treat them as if they were right-justified in the field with the number of leading blanks equal to the number of ignored blanks. For instance, the following READ statement accepts the characters shown between the slashes as the value 123 (where <lf> indicates pressing the RETURN key).

```
    READ(*,100) I
100 FORMAT (BN,I6)

    /123 <lf> /,
    /123 456 <lf> /,
    / 123 <lf> /.
```

The BN edit descriptor, in conjunction with the infinite blank padding at the end of formatted records, makes interactive input very convenient.

Repeatable

I, F, and E (Numeric Editing). The I, F, and E edit descriptors are used for I/O of integer and real data. The following general rules apply to all three of them.

- On input, leading blanks are not significant. Other blanks are interpreted differently depending on the BN or BZ flag in effect, but all blank fields always become the value 0. Plus signs are optional.
- On input, with F and E editing, an explicit decimal point appearing in the input field overrides the edit descriptor specification of the decimal point position.
- On output, the characters generated are right-justified in the field with padding by leading blanks if necessary.
- On output, if the number of characters produced exceeds the fields width or the exponent exceeds its specified width, the entire field is filled with asterisks.

I (Integer Editing). The edit descriptor Iw must be associated with iolist item of type integer. The field is w characters wide. On input, an optional sign may appear in the field.

F (Real Editing). The edit descriptor Fw.d must be associated with an iolist item of type real. The field is w characters wide, with a fractional part d digits wide. The input field begins with an optional sign followed by a string of digits optionally containing a decimal point. If the decimal point is present, it overrides the d specified in the edit descriptor; otherwise the rightmost d digits of the string are interpreted as following the decimal point (with leading blanks converted to zeros if necessary). Following this is an optional exponent which is either:

- + (plus) or - (minus) followed by an integer, or
- E or D followed by zero or more blanks followed by an optional sign followed by an integer.

The output fields occupies w digits, d of which fall beyond the decimal point and the value output is controlled both by the iolist item and the current scale factor. The output value is rounded rather than truncated.

E (Real Editing). An E edit descriptor takes either the form Ew.d or Ew.dEe. In either case the field is w characters wide. The e has no effect on input. The input field for an E edit descriptor is identical to that described by an F edit descriptor with the same w and d. The form of the output field depends on the scale factor (set by the P edit descriptor) in effect. For a scale factor of 0, the output field is a minus sign (if necessary), followed by a decimal point, followed by a string of digits, followed by an exponent field for exponent, exp, of one of the following forms.

Ew.d	-99 < = exp < = 99	E followed by plus or minus followed by the two-digit exponent.
Ew.d	-999 < = exp < = 999	Plus or minus followed by the three-digit exponent.
Ew.dEe	-((10**e) - 1) < = exp < = (10**e) -1	E followed by plus or minus followed by e digits which are the exponent with possible leading zeros.

The form Ew.d must not be used if the absolute value of the exponent to be printed exceeds 999.

The scale factor controls the decimal normalization of the printed E field. If the scale factor, k, is in the range -d < k < = 0, then the output field contains exactly -k leading zeros after the decimal point and d + k significant digits after this. If 0 < k < d + 2, then the output fields contains exactly k significant digits to the left of the decimal point and d - k - 1 places after the decimal point. Other values of k are errors.

L (Logical Editing). The edit descriptor is Lw, indicating that the field is w characters wide. The iolist element associated with an L edit descriptor must be of type logical. On input, the field consists of optional blanks, followed by an optional decimal point, followed by T (for .TRUE.) or F (for .FALSE.). Any further characters in the field are ignored, but accepted on input, so that .TRUE. and .FALSE. are valid inputs. On output, w - 1 blanks are followed by either T or F as appropriate.

A (Character Editing). The forms of the edit descriptor are A or Aw, in which the former acquires an implied field width, w, from the number of characters in the iolist item with which it is associated. The iolist item must be of type character if it is to be associated with an A or Aw edit descriptor. On input, if w exceeds or equals the number of characters in the iolist element, the rightmost characters of the input field are used as the input characters; otherwise the input characters are left-justified in the input iolist item and trailing blanks are provided. On output, if w exceeds the characters produced by the iolist item, leading blanks are provided; otherwise, the leftmost w characters of the iolist item are output.

SECTION 13
PROGRAMS, SUBROUTINES, AND FUNCTIONS

This Section describes the format of program units. A program unit is either a main program, a subroutine, or a function program unit. Procedure refers to either a function or a subroutine. This Section also describes the CALL and RETURN statements, and function calls.

MAIN PROGRAM

A main program is any program unit that does not have a FUNCTION or SUBROUTINE statement as its first statement. In addition, it may have a PROGRAM statement as its first statement. The execution of a program always begins with the first executable statement in the main program. Consequently, there must be precisely one main program in every executable program.

The form of a PROGRAM statement is:

PROGRAM pname

where:

pname is a user-defined name that is the name of the main program.

pname is a global name. Therefore, it cannot be the same as that of another external procedure or common block. (It is also a local name to the main program, and must not conflict with any local name in the main program.) The PROGRAM statement may only appear as the first statement of a main program.

SUBROUTINES

A subroutine is a program unit that can be called from other program units by a CALL statement. When invoked, it performs the set of actions defined by its executable statements, and then returns control to the statement immediately following the statement that called it. A subroutine does not directly return a value, although values can be passed back to the calling program unit via parameters or common variables.

SUBROUTINE Statement

A subroutine begins with a SUBROUTINE statement and ends with the first following END statement. It can contain any kind of statement other than a PROGRAM statement, SUBROUTINE statement, or a FUNCTION statement.

The form of a SUBROUTINE statement is:

SUBROUTINE sname [([farg [, farg] ...])]

where:

sname	is the user-defined name of the subroutine.
farg	is the user-defined name of a formal argument.

sname is a global name (and it is also local to the subroutine it names). The list of argument names defines the number and, with any subsequent IMPLICIT, type, or DIMENSION statements, the type of arguments to that subroutine. Argument names cannot appear in COMMON, DATA, EQUIVALENCE, or INTRINSIC statements.

CALL Statement

A subroutine is executed by executing a CALL statement in another program unit that references that subroutine.

The form of a CALL statement is:

CALL sname [([arg [, arg] ...])]

where:

sname	is the user-defined name of a subroutine.
arg	is an actual argument.

An actual argument may be either an expression or the name of an array. The actual arguments in the CALL statement must agree in type and number with the corresponding formal arguments specified in the SUBROUTINE statement of the referenced subroutine. If there are no arguments in the SUBROUTINE statement, then a CALL statement referencing that subroutine must not have

any actual arguments, but may optionally have a pair of parentheses following the name of the subroutine. Note that a formal argument can be used as an actual argument in another subprogram call.

Execution of a CALL statement proceeds as follows. All arguments that are expressions are evaluated. All actual arguments are associated with their corresponding formal arguments, and the body of the specified subroutine is executed. Upon exiting the subroutine, control is returned to the statement following the CALL statement by executing either a RETURN statement or an END statement in that subroutine.

A subroutine can be called from any program unit. Recursive subroutine calls, however, are not normally permitted in FORTRAN. That is, a subroutine cannot call itself directly, nor can it call another subroutine that results in that subroutine being called again before it returns control to its caller. However, the DYNAMIC directive of Burroughs FORTRAN gives you the optional capability of writing recursive subroutines. See Section 14 for details.

```
C EXAMPLE OF CALL STATEMENT
C CALL ERROR TO REPORT AN ERROR
      IF (IERR .NE. 0) CALL ERROR(IERR)
      END
C
      SUBROUTINE ERROR(IERRNO)
      WRITE (*, 200) IERRNO
200   FORMAT(1X, 'ERROR', I5, 'DETECTED')
      END
```

FUNCTIONS

A function is referred to in an expression and returns a value that is used in the computation of that expression. There are three kinds of functions: external, intrinisic, and statement.

A function reference may appear in an arithmetic expression. Execution of a function reference causes the function to be evaluated, and the resulting value is used as an operand in the containing expression. The form of a function reference is:

fname ([arg [, arg] ...])

where:

fname	is the user-defined name of an external, intrinsic, or statement function.
arg	is an actual argument.

An actual argument can be an arithmetic expression or an array. The number of actual arguments must be the same as in the definition of the function, and the corresponding types must agree.

External Functions

An external function is specified by a function program unit. It begins with a FUNCTION statement and ends with an END statement. It may contain any kind of statement other than a PROGRAM statement, FUNCTION statement, or a SUBROUTINE statement.

The form of a FUNCTION statement is:

[type] FUNCTION fname ([farg [, farg] ...])

where:

type	is INTEGER, REAL, or LOGICAL.
fname	is the user-defined name of the function.
farg	is a formal argument name.

fname is a global name, and it is also local to the function it names. If no type is present in the FUNCTION statement, the function's type is determined by default and by any subsequent IMPLICIT or type statements that determine the type of an ordinary variable. If a type is present, then the function name cannot appear in any additional type statements.

In any event, an external function cannot be of type CHARACTER. The list of argument names defines the number and, with any subsequent IMPLICIT, type, or DIMENSION statements, the type of arguments to that subroutine. Neither argument names nor fname can appear in COMMON, DATA, EQUIVALENCE, or INTRINSIC statements.

The function name must appear as a variable in the program unit defining the function. Every execution of that function must assign a value to that variable. The final value of this variable, upon execution of a RETURN or an END statement, defines the value of the function. After being defined, the value of this variable can be referenced in an expression, exactly as any other variable. An external function may return values in addition to the value of the function by assignment to one or more of its formal arguments.

A function can be called from any program unit. Recursive function calls, however, are not normally permitted in FORTRAN. That is, a function cannot call itself directly, nor can it call another function that results in that function being called again before it returns control to its caller. However, the DYNAMIC directive of Burroughs FORTRAN gives you the optional capability of writing recursive functions. See Section 14 for details.

```
C EXAMPLE OF A FUNCTION REFERENCE
C GETCH IS A FUNCTION THAT READS A
C CHARACTER FROM A FILE
      I = 2
      IF (GETCH(I) .EQ. 'Y') GOTO 10
      GOTO 100
      END
C
      CHARACTER GETCH(UNITNO)
      CHARACTER CH
      READ(UNITNO, (A1)') CH
      GETCH = CH
      RETURN
      END
```

Intrinsic Functions

An intrinsic function is predefined by the FORTRAN compiler and available for use in a FORTRAN program. Table 13-1 below gives the name, definition, number of parameters, and type of the intrinsic functions available in Burroughs FORTRAN 77. An IMPLICIT statement does not alter the type of an intrinsic function. For those intrinsic functions that allow several types of arguments, all arguments in a single reference must be of the same type.

Only those intrinsic functions listed in Table 13-1 can appear in an INTRINSIC statement. An intrinsic function name also can appear in a type statement, but only if the type is the same as the standard type of that intrinsic function.

Arguments to certain intrinsic functions are limited by the definition of the function being computed. For example, the logarithm of a negative number is mathematically undefined, and therefore not permitted.

In Table 13-1 all angles are expressed in radians. All arguments in an intrinsic function reference must be of the same type. X and Y are real, I and J integer, and C, C1, and C2 character values.

Appendix D contains additional intrinsic functions. The functions in this appendix are altered by an IMPLICIT statement.

Table 13-1. Intrinsic Functions. (Page 1 of 4)

Name	Definition	Type of Argument	Type of Function
Type Conversion			
INT(X)	Conversion to integer[1]	Real	Integer
IFIX(X)	Conversion to integer[1]	Real	Integer
REAL(X)	Conversion to real[2]	Integer	Real
FLOAT(I)	Conversion to real[2]	Integer	Real
ICHAR(C)	Conversion to integer[3]	Character	Integer
CHAR(X)	Conversion to character	Integer	Character
SNGL(X)	Conversion to real	Double	Real
DBLE(X)	Conversion to double	Real	Double
Truncation			
AINT(X)	Truncation to real[1]	Real	Real
IDINT(X)	Truncation to integer	Double	Integer
Nearest Whole Number			
ANINT(X)	Rounding to real[1]	Real	Real
Nearest Integer			
NINT(X)	Rounding to integer	Real	Integer
Absolute Value			
IABS(I)	Integer absolute	Integer	Integer
ABS(X)	Real absolute	Real	Real
DABS(X)	Double precision absolute	Double	Double
Remaindering			
MOD(I,J)	Integer remainder[1]	Integer	Integer
AMOD(X,Y)	Real remainder[1]	Real	Real
DMOD(X,Y)	Double-precision remainder	Double	Double
Transfer of Integer Sign			
ISIGN(I,J)	Integer transfer	Integer	Integer
SIGN(X,Y)	Real transfer	Real	Real
DSIGN(X,Y)	Double-precision transfer	Double	Double

X and Y are real, I and J integer, and C, C1, and C2 character values.

Table 13-1. Intrinsic Functions. (Page 2 of 4)

Name	Definition	Type of Argument	Type of Function
Positive Difference[4]			
IDIM(I,J)	Integer difference	Integer	Integer
DIM(X,Y)	Real difference	Real	Real
DDIM(X,Y)	Double-precision difference	Double	Double
Choosing Largest Value			
MAXO(I,J,. . .)	Integer maximum	Integer	Integer
AMAX1(X,Y,. . .)	Real maximum	Real	Real
AMAXO(I,J,. . .)	Real maximum	Integer	Real
MAX1(X,Y,. . .)	Integer maximum	Real	Integer
DMAX1(X,Y,. . .)	Double maximum	Double	Double
Choosing Smallest Value			
MINO(I,J,. . .)	Integer minimum	Integer	Integer
AMIN1(X,Y,. . .)	Real minimum	Real	Real
AMINO(I,J,. . .)	Real minimum	Integer	Real
MIN1(X,Y,. . .)	Integer minimum	Real	Integer
DMIN1(X,Y,. . .)	Double minimum	Double	Double
Square Root			
SQRT	Square Root	Real	Real
DSQRT	Double-precision square root	Double	Double
Exponential			
EXP(X)	Real e raised to power	Real	Real
DEXP(X)	Double-precision e raised to power	Double	Double
Natural Logarithm			
ALOG(X)	Natural logarithm of real argument	Real	Real
DLOG(X)	Natural logarithm of double-precision argument	Double	Double

X and Y are real, I and J integer, and C, C1, and C2 character values.

Table 13-1. Intrinsic Functions. (Page 3 of 4)

Name	Definition	Type of Argument	Type of Function
Common Logarithm			
ALOG10(x)	Common logarithm of real argument	Real	Real
DLOG10(X)	Common logarithm of double-precision argument	Double	Double
Sine			
SIN(X)	Real sine	Real	Real
DSIN(X)	Double-precision sine	Double	Double
Cosine			
COS (X)	Real cosine	Real	Real
DCOS (X)	Double-precision cosine	Double	Double
Tangent			
TAN(X)	Real Tangent	Real	Real
DTAN(X)	Double-precision tangent	Double	Double
Arc Sine			
ASIN(X)	Real arc sine	Real	Real
DASIN(X)	Double-precision arc sine	Double	Double
Arc Cosine			
ACOS(X)	Real arc cosine	Real	Real
DACOS(X)	Double-precision arc cosine	Double	Double
Arc Tangent			
ATAN(X)	Real arc tangent	Real	Real
DATAN(X)	Double-precision arc tangent	Double	Double
ATAN2(X/Y)	Real arc tangent of X/Y	Real	Real
DATAN2(X/Y)	Double-precision arc tangent of X/Y	Double	Double
Hyperbolic Sine			
SINH(X)	Real hyperbolic sine	Real	Real
DSINH(X)	Double-precision hyperbolic sine	Double	Double

X and Y are real, I and J integer, and C, C1, and C2 character values.

Table 13-1. Intrinsic Functions. (Page 4 of 4)

Name	Definition	Type of Argument	Type of Function
Hyperbolic Cosine COSH(X) DCOSH(X)	Real hyperbolic cosine Double-precision hyperbolic cosine	Real Double	Real Double
Hyperbolic Tangent TANH(X) DTANH(X)	Real hyperbolic tangent Double-precision hyperbolic tangent	Real Double	Real Double
Lexically Greater Than or Equal LGE(C1,C2)	First argument greater than or equal to second[5]	Character	Logical
Lexically Greater Than LGT(C1,C2)	First argument greater than second[5]	Character	Logical
Lexically Greater Than or Equal LLE(C1,C2)	First argument less than second[5]	Character	Logical
End of File EOF(X)	Integer and of file[6]	Integer	Logical

X and Y are real, I and J integer, and C, C1, and C2 character values.

1. For X of type real, if X > = 0, then INT(X) is the largest integer not greater than X, and if X< 0, then INT(X) is the most negative integer not less than X. IFIX(X) is the same as INT(X).

2. For X of type integer, REAL(X) is as much precision of the significant part of X as a real value can contain. FLOAT(X) is the same as REAL(X).

3. ICHAR converts a character value into an integer value. The integer value of a character is the ASCII internal representation of that character, and is in the range 0 to 255. For any two characters, c1 and c2, (c1 .LE. c1) is true if and only if (ICHAR(c1) .LE. ICHAR(c2)) is true.

4. Positive difference is defined as the actual difference if that number is positive and 0 otherwise.

5. LGE(X,Y) returns true if X = Y or if X follows Y in the ASCII collating sequence; otherwise it returns false.

   LGT(X,Y) returns true if X follows Y in the ASCII collating sequence; otherwise it returns false.

   LLE(X,Y) returns true if X = Y or if X precedes Y in the ASCII collating sequence; otherwise it returns false.

   LLT(X,Y) returns true if X precedes Y in the ASCII collating sequence; otherwise it returns false.

   The operands of LGE, LGT, LLE, and LLT must be of the same length.

6. EOF(a) returns the value true if the unit specified by its argument is at or past the end of file record; otherwise it returns false. The value of a must correspond to an open file, or to 0 which indicates the screen or keyboard device.

Statement Functions

A statement function is defined by a single statement. It is similar in form to an assignment statement. A statement function statement can only appear after the specification statements and before any executable statements in the program unit in which it appears.

A statement function is not an executable statement, since it is not executed in order as the first statement in its particular program unit. Rather, the body of a statement function serves to define the meaning of the statement function. It is executed, as any other function, by the execution of a function reference.

The form of a statement function is:

fname ([arg [, arg] ...]) = expr

where:

fname	is the user-defined name of the statement function.
arg	is a formal argument name.
expr	is an expression.

The type of the expr must be assignment compatible with the type of the statement function name. The list of formal argument names serves to define the number and type of arguments to the statement function. The scope of formal argument names is the statement function. Therefore, formal argument names can be used as other user-defined names in the rest of the program unit enclosing the statement function definition.

The name of the statement function, however, is local to the enclosing program unit, and must not be used otherwise (except as the name of a common block, or as the name of a formal argument to another statement function). The type of all such uses, however, must be the same. If a formal argument name is the same as another local name, then a reference to that name within the statement function defining it always refers to the formal argument, never to the other usage.

Within the expression expr, references to variables, formal arguments, other functions, array elements, and constants are permitted. Statement function references, however, must refer to statement functions defined prior to the statement function in which they appear. Statement functions cannot be called recursively, either directly or indirectly.

A statement function can only be referenced in the program unit in which it is defined. The name of a statement function cannot appear in any specification statement, except in a type statement which may not define that name as an array, and in a COMMON statement as the name of a common block. A statement function cannot be of type character.

```
C EXAMPLE OF STATEMENT FUNCTION STATEMENT
      DIMENSION X(10)
      ADD(A,B) = A + B
C
      DO 1, I=1, 10
      X(I) = ADD(Y, Z)
1     CONTINUE
```

RETURN STATEMENT

A RETURN statement causes return of control to the calling program unit. t can only appear in a function or subroutine.

The form of a RETURN statement is:

```
RETURN
```

Execution of a RETURN statement terminates the execution of the enclosing subroutine or function. If the RETURN statement is in a function, then the value of that function is equal to the current value of the variable with the same name as the function. Execution of an END statement in a function or subroutine is equivalent to execution of a RETURN statement.

```
C EXAMPLE OF RETURN STATEMENT
C THIS SUBROUTINE LOOPS UNTIL THE USER
C TYPES 'Y' TO THE KEYBOARD
      SUBROUTINE LOOP
      CHARACTER IN
C
10    READ(*, '(A1)') IN
      IF (IN .EQ. 'Y') RETURN
      GOTO 10
      RETURN
      END
```

PARAMETERS

This subsection discusses the relationship between formal and actual arguments in a function or subroutine call. A formal argument is the name by which the argument is known within the function or subroutine, and an actual argument is the specific variable, expression, array, etc., passed to the procedure in question at any specific calling location.

Arguments pass values into and out of procedures. The number of actual arguments must be the same as formal arguments, and the corresponding types must agree.

Upon entry to a subroutine or function, the actual arguments are associated with the formal arguments, much as an EQUIVALENCE statement associates two or more arrays or variables, and COMMON statements in two or more program units associate lists of variables. This association remains in effect until execution of the subroutine or function is terminated.

Thus, assigning a value to a formal argument during execution of a subroutine or function may alter the value of the corresponding actual argument. If an actual argument is a constant, function reference, or an expression other than a simple variable, assigning a value to the corresponding formal argument is not permitted, and can have some strange side effects. In particular, assigning a value to a formal argument of type character, when the actual argument is a literal, can produce anomalous behavior.

If an actual argument is an expression, it is evaluated immediately prior to the association of formal and actual arguments. If an actual argument is an array element, its subscript expression is evaluated just prior to the association, and remains constant throughout the execution of the procedure, even if it contains variables that are redefined during the execution of the procedure.

A formal argument that is a variable can be associated with an actual argument that is a varible, an array element, or an expression.

A formal argument that is an array can be associated with an actual argument that is an array or an array element. The number and size of dimensions in a formal argument may be different than those of the actual argument, but any reference to the formal array must be within the limits of the memory sequence in the actual array. While a reference to an element outside these bounds is not detected as an error in a running FORTRAN program, the results are unpredictable.

A formal argument can also be the name of an external procedure, function, or intrinsic function. the actual argument must appear in an EXTERNAL or INTRINSIC statement in the program unit in which the procedure or function reference is made.

SECTION 14
COMPILER DIRECTIVES

OVERVIEW

This Section describes how compiler directives direct the FORTRAN compiler to process FORTRAN source text in particular ways. Compiler Directives may be intermixed with FORTRAN source text within a FORTRAN source program; however, they are not part of the FORTRAN language. Any line of input to the FORTRAN compiler that begins with a $ in column 1 is interpreted as a compiler directive and must conform to one of the formats below. A compiler directive must fit on a single source line; continuation lines are not permitted. Also, blanks are not permitted in directive lines.

DEBUG DIRECTIVE

The DEBUG directive directs that all subsequent arithmetic operations are tested for overflow and division by 0. A run-time error is generated if such a condition is detected. Also, range checking is performed on array indexes to assure valid index values.

It has the format:

```
$DEBUG
```

The directive can appear anywhere in a program.

The default value of the DEBUG/NODEBUG pair of directives is NODEBUG.

DO66 DIRECTIVE

The DO66 directive directs that DO statements have FORTRAN 66 semantics. It has the format:

```
$DO66
```

DO66 must precede the first declaration or executable statement of the source file in which it occurs.

The FORTRAN 66 semantics are as follows. First, all DO statements are executed at least once. Second, extended range is permitted; that is, control may transfer in and out of the syntactic body of a DO statement. The range of the DO statement is thereby extended to logically include any statement that

may be executed between a DO statement and its terminal statement. However, the transfer of control into the range of a DO statement prior to the execution of the DO statement or following the final execution of its terminal statement is invalid.

If a program contains no DO66 directive, the default is to FORTRAN 77 semantics, as follows. First, DO statements may be executed 0 times, if the initial control variable value exceeds the final control variable value (or the corresponding condition for a DO statement with negative increment). Second, extended range is invalid, that is, control may not transfer in and out of the syntactic body of a DO statement.

DYNAMIC DIRECTIVE

The DYNAMIC directive directs that local variables of functions and subroutines are allocated on the stack. The values of these variables are therefore not retained from one invocation of the containing function or subprogram to the next. However, it is valid for functions and subprograms under the control of DYNAMIC to be called recursively. Typically, DYNAMIC reduces the amount of memory required by a program.

It has the format:

```
$DYNAMIC
```

DYNAMIC must precede the first declaration or executable statement of the source file in which it occurs.

If a program contains no DYNAMIC directive, the default directs that local variables of functions and subroutines are allocated in static memory, not on the stack. The values of these variables are therefore retained from one invocation of the containing function or subprogram to the next. However, it is invalid for functions and subprograms under the control of this default condition to be called recursively.

INCLUDE DIRECTIVE

The INCLUDE directive directs the compiler to proceed as though the specified file were textually inserted at the point of INCLUDE. At the end of the included file, the compiler resumes processing the original source file at the line following INCLUDE.

It has the format:

$INCLUDE: 'filespec'

where:

filespec is a valid Burroughs file specification, as described in the Executive Manual.

INCLUDE directives can be nested up to eight levels. INCLUDE directives are particuarly useful in guaranteeing that several modules use the same declaration for a COMMON block.

LINESIZE DIRECTIVE

The LINESIZE directive directs subsequent pages of the listing to be formatted n pages wide.

It has the format:

$LINESIZE: n

where:

n is any positive integer.

If a program contains no LINESIZE directive, a default line size of 132 characters is assumed.

NODEBUG DIRECTIVE

The NODEBUG directive turns off DEBUG checks for array indexes and arithmetic operations.

It has the format:

$NODEBUG

The directive can appear anywhere in a program.

The default value of the DEBUG/NODEBUG pair of directives is NODEBUG.

PAGE DIRECTIVE

The PAGE directive directs a new page of the listing to be started. If the first character of a line of source text is the ASCII form feed character (hexadecimal code OCh), this is treated as equivalent to the occurrence of a PAGE directive at that point.

It has the format:

$PAGE

PAGESIZE DIRECTIVE

The PAGESIZE directive directs subsequent pages of the listing to be formatted n lines high.

It has the format:

$PAGESIZE: n

where:

n	is any positive integer.

If a program contains no PAGESIZE directive, a default page size of 66 lines is assumed.

STORAGE DIRECTIVE

The STORAGE directive directs that all variables declared in the source file as INTEGER or LOGICAL are allocated n bytes of memory. STORAGE does not affect the allocation of memory for variables declared with an explicit length specification, for example, as INTEGER*n or LOGICAL*n. If several files of a source program are compiled separately and linked together, you should be particularly careful that they are consistent in their allocation of memory for variables (such as actual and formal parameters) referred to in more than one module.

It has the format:

$STORAGE: n

where:

n	is either 2 or 4.

This directive must precede the first declaration or executable statement of the source file in which it occurs.

If a program contains no STORAGE directive, a default allocation of 4 bytes is used. Note that this default results in INTEGER, LOGICAL, and single precision REAL variables being allocated the same amount of memory, as specified by the FORTRAN standard, ANSI X3.9-1978.

The TITLE directive directs subsequent pages of the listing to be headed with the specified title, until overridden by another TITLE directive.

It has the format:

$TITLE: 'title'

where:

title is any valid character constant.

If a program contains no TITLE directive, the null string is used as a title.

APPENDIX A
ERROR MESSAGES

Compile-Time Error Messages: Pass 1

The FORTRAN Compiler is a two-pass compiler. These error messages occur during the first pass of of the FORTRAN Compiler (FORTRAN, RUN).

Decimal Value	Meaning
1	Fatal error reading source block
2	Nonnumeric characters in label field
3	Too many continuation lines
4	Fatal end of file encountered
5	Labeled continuation line
6	Missing field on $ compiler directive line
7	Unable to open listing file specified on $ compiler directive line
8	Unrecognizable $ compiler directive
9	Input source file not valid textfile format
10	Maximum depth of include file nesting exceeded
11	Integer constant overflow
12	Error in real constant
13	Too many digits in constant
14	Identifier too long
15	Character constant extends to end of line
16	Zero length character constant
17	Illegal character in input
18	Integer constant expected
19	Label expected
20	Error in label
21	Type name expected (INTEGER, REAL, LOGICAL, CHARACTER, or DOUBLE PRECISION).
22	Integer constant expected
23	Extra characters at end of statement
24	"(" expected
25	Letter IMPLICITed more than once
26	")" expected
27	Letter expected
28	Identifier expected
29	Dimension(s) required in DIMENSION statement
30	Array dimensioned more than once
31	Maximum of three dimensions in an array
32	Incompatible arguments to EQUIVALENCE
33	Variable appears more than once in a type specification statement
34	This identifier has already been declared
35	This intrinsic function cannot be passed as an argument
36	Identifier must be a variable

Decimal Value	Meaning
37	Identifier must be a variable or the current FUNCTION
38	"/" expected
39	Named COMMON block already saved
40	Variable already appears in a COMMON block
41	Variables in two different COMMON blocks cannot be equivalenced
42	Number of subscripts in EQUIVALENCE statement does not agree with variable declaration
43	EQUIVALENCE subscript out of range
44	Two distinct cells EQUIVALENCEd to the same location in a common block
45	EQUIVALENCE statement extends a COMMON block in the negative direction
46	EQUIVALENCE statement forces a variable to two distinct locations, not in a COMMON block
47	Statement number expected
48	Mixed CHARACTER and numeric items not allowed in same COMMON block
49	CHARACTER items cannot be EQUIVALENCEd with noncharacter items
50	Illegal symbol in expression
51	Can't use SUBROUTINE name in an expression
52	Type of argument must be INTEGER or REAL
53	Type of argument must be INTEGER, REAL, or CHARACTER
54	Types of comparisons must be compatible
55	Type of expression must be LOGICAL
56	Too many subscripts
57	Too few subscripts
58	Variable expected
59	"=" expected
60	Size of EQUIVALENCEd CHARACTER items must be the same
61	Illegal assignment - types do not match
62	Can only call SUBROUTINES
63	Dummy parameters cannot appear in COMMON statements
64	Dummy parameters cannot appear in EQUIVALENCE statements
65	Assumed-size array declarations can only be used for dummy arrays
66	Adjustable-size array declarations can only be used for dummy arrays
67	Assumed-size array dimension specifier must be last dimension

Decimal Value	Meaning
68	Adjustable bound must be either parameter or in COMMON prior to appearance
69	Adjustable bound must be simple integer variable
70	Cannot have more than one main program
71	The size of a named COMMON must be the same in all procedures
72	Dummy arguments cannot appear in DATA statements
73	COMMON variables cannot appear in DATA statements
74	SUBROUTINE names, FUNCTION names, INTRINSIC names, etc. cannot appear in DATA statements
75	Subscript out of range in DATA statement
76	Repeat count must be =1
77	Constant expected
78	Type conflict in DATA statement
79	Number of variables does not match number of values in DATA statement list
80	Statement cannot have label
81	No such INTRINSIC function
82	Type declaration for INTRINSIC function does not match actual type of INTRINSIC function
83	Letter expected
84	Type of FUNCTION does not agree with a previous call
85	This procedure has already appeared in this compilation
87	Error in type of argument to an INTRINSIC FUNCTION
88	SUBROUTINE/FUNCTION was previously used as a FUNCTION/SUBROUTINE
89	Unrecognizable statement
90	Functions cannot be of type CHARACTER
91	Missing END statement
93	Fewer actual arguments than formal arguments in FUNCTION/SUBROUTINE call
94	More actual arguments than formal arguments in FUNCTION/SUBROUTINE call
95	Type of actual argument does not agree with type of format argument
96	The following procedures were called but not defined:
98	Maximum size of type CHARACTER is 255, minimum is 1
100	Statement out of order
101	Unrecognizable statement

Decimal Value	Meaning
102	Illegal jump into block
103	Label already used for FORMAT
104	Label already defined
105	Can't jump to format label
106	DO statement forbidden in this context
107	DO label must follow DO statement
108	ENDIF forbidden in this context
109	No matching IF for this ENDIF
110	Improperly nested DO block in IF block
111	ELSEIF forbidden in this context
112	No matching IF for ELSEIF
113	Improperly nested DO or ELSE block
114	"(" expected
115	")" expected
116	THEN expected
117	Logical expression expected
118	ELSE statement forbidden in this context
119	No matching IF for ELSE
120	Unconditional GOTO forbidden in this context
121	Assigned GOTO forbidden in this context
122	Block IF statement forbidden in this context
123	Logical IF statement forbidden in this context
124	Arithmetic IF statement forbidden in this context
125	, expected
126	Expression of wrong type
127	RETURN forbidden in this context
128	STOP forbidden in this context
129	END forbidden in this context
131	Label referenced but not defined
132	DO or IF block not terminated
133	FORMAT statement not permitted in this context
134	FORMAT label already referenced
135	FORMAT must be labeled
136	Identifier expected
137	Integer variable expected
138	TO expected
139	Integer expression expected
140	Assigned GOTO but no ASSIGN statements
141	Unrecognizable character constant as option
142	Character constant expected as option
143	Integer expression expected for unit designation
144	STATUS option expected after "," in CLOSE statement

Decimal Value	Meaning
145	Character expression as file name in OPEN
146	FILE= option must be present in OPEN statement
147	RECL= option specified twice in OPEN statement
148	Integer expression expected for RECL= option in OPEN statement
149	Unrecognizable option in OPEN statement
150	Direct access files must specify RECL= in OPEN statement
151	Adjustable arrays not allowed as I/O list elements
152	End of statement encountered in implied DO, expression beginning with "(" not allowed as I/O list elements
153	Variable required as control for implied DO
154	Expressions not allowed in I/O list for READ statement
155	REC= option appears twice in statement
156	REC= expects integer expression
157	END= option only allowed in READ statement
158	END= option appears twice in statement
159	Unrecognizable I/O unit
160	Unrecognizable format in I/O statement
161	Options expected after "," in I/O statement
162	Unrecognizable I/O list element
163	Label used as format but not defined in format statment
164	Integer variable used as assigned format but no ASSIGN statements
165	Label of an executable statement used as a format
166	Integer variable expected for assigned format
167	Label defined more than once as format
203	FUNCTIONS cannot return values of type CHARACTER
406	Cannot open unit 0 as DIRECT or UNFORMATTED file.
407	"Err=" clause appears twice in a statement.
408	Too many labels for arithmetic IF.
409	Byte length incompatible with type.
410	Keyword PRECISION expected.
411	Incompatible integer type.
412	Incompatible logical type.
420	Illegal function call
421	Illegal intrinsic function used as procedural parameter. (See section on Intrinsic functions in this release notice.)
500	Real number constant overflow.
501	Incorrect syntax for compiler directive.
502	Blanks not allowed in compiler directive.
503	Incorrect placement of compiler directive.
504	Duplicate definition of compiler directice.
1274	Illegal Integer*4 constant on input.
1275	Integer*4 error on output.

COMPILER-TIME ERROR MESSAGES: PASS 2

Compiler Errors - Optimizer and Code Generator

The optimizer and code generators perform a large amount of internal consistency checking, in order to verify that icode (intermediate code) trees are of a form that is expected or can be handled; that register allocation and usage is correct; etc. When one of these checks encounters an unexpected condition, the result is an internal error generated by the module where the inconsistency was discovered.

Such errors should generally not occur. When they do occur, we request that they be reported promptly to Burroughs. Since it may be quite difficult to analyze such reports unless they include the complete source code involved, please include the complete source code in a machine readable form.

All optimizer/code generator internal errors are numbered according to the module in which they occur. This error number is then passed to a single internal error routine (PROCEDURE OPT=ERR in module SUBR) which will print the error number, and the last source line number encountered by the optimizer on both the object code output listing (if any) and the terminal screen. The optimizer is then exited via a call to the PASCAL run-time routine EMSEQQ.

The internal error numbering is as follows:

0 - 99	OPTIM
100 - 199	GEN6
299 - 299	SUBR
300 - 399	FOLD
400 - 499	CHKLEN
500 - 599	CTL6
600 - 699	DUMP86
700 - 799	DUMP

The format of an optimizer error message is as shown below:

*** Internal Error <error number>
Near Line <source line number>
Contact Technical Support

where <error number> is one of the optimizer error numbers listed below, and <source line number> is the last source line number seen by the optimizer. The error may not have occurred exactly at this line (due to the way the optimizer gets line numbers) but it is likely to be within a few lines following this line. The <source line number> corresponds to the line numbers on the listing generated by the front end.

Errors are listed by module, with the following format for each error messages:

<err num>: Error explanation (routine in which errror occurs)

Value	Meaning
0	Garbage in Icode stream (READICODE).
1	Bad Icode file format (PRSDEC10).
2	Bad Symbols file format, can't find function return variable (READ_SYMTAB).
3	Multiple symbol file entries for symbol which is not procedure or function (READ_SYMTAB).
4	Forward reference to an Icode number (XLATE).
5	Icode reference to a missing symbol (XLATE_SYM).
6	Duplicate Icode numbers in same block (ENTER_XLATE).
7	Illegal or unexpected operand for ADDR Icode (PHASE1).
8	Illegal addressing mode for ADDr Icode (PHASE1).
9	Illegal or unexpected operand for DRRR Icode (PHASE1).
10	Illegal or unexpected operand for DRFR Icode (PHASE1).
11	Illegal symbol type for UPPR Icode (PHASE1).
12	Illegal addressing mode for ASMS/ASVS (PHASE1).
13	Bad tree format, assignment target tree does not have a SYMR node as its leftmost leaf (DEL_TARGET).
14	Unknown Icode value (SUBEX).
15	Bad statement list returned from SPLITTREE (OPTIM - main program).
16	Bad statement list returned from PHASE1 (OPTIM).
17	Bad statement list returned from CHECK_LENGTH (OPTIM).
18	Bad statement list returned from PHASE2 (OPTIM).
19	Bad statement list rturned from PHASE3 (OPTIM).
20	Bad statement list returned from MD_XFORM (OPTIM).
21	Bad statement list returned from SUBEX (OPTIM).
22	Unexpected operand for OFSR icode (PHASE1).
23	Subexpression lists not cleared (SUBEX).
24	Subexpression lists not cleared (SUBEX).
100	Static nesting level 0 (NESTLEV).
101	Illegal or unexpected oeprand for OFFR Icode (MD_XFORM).
102	Illegal flag values for CONR (MD_XFORM).
103	Illegal or unexpected operand for UPPR Icode (MD_XFORM).
104	Illegal symbol type for UPPR operand (MD_XFORM).
105	Too many levels of indirection for UPPR operand (MD_XFORM).
106	Illegal addressing mode for VALP Icode (MD_XFORM).
107	Illegal or unexpected oprand for LVAP Icode (MD_XFORM).
108	Multiple definition of an internal label (GENDONE).
109	Can't load long constant value with a length >4 (CASELONR).
110	Illegal offset value for OFSR Icode (CASEOFSR).
111	Register table entry or use count for OFSR Icode is bad (CASEOFSR).
112	Illegal nesting for procedure/function call (CALLPF).
113	Illegal function return length (CASECALP).
114	Bad use count for SFRT Icode operands (CASESFRT).
115	Symbol type is illegal, must be a variable (CLASS).
116	Operand use count is already zero (COUNTUSE).
117	User label must begin a basic block (DEF_ULAB).
118	Duplicate definition of user label (DEF_ULAB).

Value	Meaning
119	Address flag missing for LONR Icode (EMITIMM).
120	Address flag missing for SYMR Icode (EMITIMM).
121	Variable must be static (EMITIMM).
122	Symbol type must be variable (EMITIMM).
123	Illegal Icode type (EMITIMM).
124	Can't save a multi-byte value (EMPTYREG).
125	Illegal register contents (EMPTYREG).
126	Symbol type must be variable (GENREF).
127	Missing address flag for long constant reference (GENREF).
128	Illegal Icode type (GENREF).
129	Value must be in an index register (GENREF).
130	Value must be in an index register (GENREFI).
131	No registers available for allocation (GETREG).
132	Register BX already in use (INBXES).
133	Register must be SI or DI (INDREF).
134	Symbol must be variable (INDREF).
135	missing address bit for long constant reference (LOADR).
136	Symbol must be a variable (LOADR).
137	Illegal Icode type (LOADR).
138	Symbol type must be a label (LONGGOTO).
139	Register residence flags do not match register table conten (MOVER).
140	Value must be in some register (REGN).
141	Illegal operand register specified by template (REGSPEC).
142	Operands register residence flag does not match the specifi register (REGSPEC).
143	Operand must be in a register (X_BINOP).
144	Unexpected opcode value (X_BINOP).
145	Contents of BX do not match operand (X_CHKBXES).
146	Illegal Icode oerator (X_COMPI).
147	Illegal Icode operand (X_CMPI).
148	Illegal variable kind (must be static) or address bit missing (CMPI).
149	Illegal Icode operator, must have 2 operands (X)COMOPR).
150	Desired register already in use (XCOMP).
151	Desired register already in use (X_DONE).
152	Index must already be in a register (X_DONEA).
153	Illegal register contents (X_DONEA).
154	Symbol must be a variable (X_DONEA).
155	Illegal Icode operand (X_DONEA).
156	Illegal condition code for IF template (X_IFCOND).
157	Illegal condition code for IFOPR template (X_IFOCOND).
158	Register BX contents are wrong (X_INREGS).
159	Source register is empty (X_MOVREG).
160	Register residence flag for operand is bad (X_MOVREG).
161	Illegal register designated, can't access high half of registe (X-SELFH).
163	Illegal Icode for assignment target (X_STOR).
164	Illegal Icode operator, must have two oeprands (X_REVOPR).

Value	Meaning
165	Illegal opcode value (X_UNIOP).
166	Illegal register specification (X_XCHG).
167	Can't exchange registers containing part of a multi-register value (X_XCHG).
168	Register residence flag does not match register table contents (X_XCHG).
169	Register residence flaf does not match register table contents (X_CHG).
170	Register table contents do not match their associated register residence flags (INTERPRET).
171	Operand must be a CONR node (INTERPRET).
172	No match for this oerand class in the templates for this Icode (SCANCLASS).
173	Register BX is already in use (INTERPRET).
174	Use count was not decremented properly (INTERPRET).
175	Use count was not decremented properly (INTERPRET).
176	Error in template processing (INTERPRET).
177	Illegal register specificatin, can't access high/low half of the register (INTERPRET).
178	Illegal or unexpected template operator (INTERPRET).
179	Illegal length for OFFR Icode, must be length 1, 2, or 4 (GEN SUBTREE).
180	Symbol table entry for RTPP Icode does not match the current procedure/function (GEN_SUBTREE).
181	Symbol table entry for RTPP Icode must be procedure or function (GEN_SUBTREE).
182	Illegal or unexpected Icode value (GEN_SUBTREE).
183	Long value parameter length must be even (GENSUBTREE).
184	Operand must be in a register (XBINOPM).
185	Unexpected opcode value (XBINOPM).
186	Incorrect value for CONR node (GENSUBTREE).

Module SUBR

200	Value too large to convert to WORD type, BOOT compiler only (WRDTOINT).
201	Missing address bit for assignment target (TARGCHECK).
202	Illegal Icode for assignment target (TARGCHECK).
203	Unexpected opcode value, BOOT compiler only (GET OPCFLAGS).
204	Illegal opcode flag value (GETTYP).
205	Illegal opcode flag value (GETTYP).
206	Unexpected ocode sequence (TARGCHECK).

Module FOLD

300	Illegal operand count, must have two operands (FOLD_CONS).
301	Illegal constant values for operands to the NOTB Icode (FOLD-CONS).

Value	Meaning
Module CHKLEN	
400	Operand length cannot be zero (CHECKLEN).
401	Length of operands must match if both are greater than zero (CHECKLEN).
402	Operand length must be -1, 1, or 2 (CHECKLEN).
403	Operand length must be -1, 1, or 2 (MUST10R2).
404	New length must be 1 or 2 (COERCE).
405	Assignment target must be variable or function (TARG_LEN).
406	Illegal Icode for assignment target (TARG_LEN).
407	Illegal symbol type for SYMR Icode (CHECK_LENGTH).
408	Assignment target length must be 4 for AS4B (CHECK LENGTH).
409	Illegal addressing fo VAXP operand (CHECK_LENGTH).
410	Unexpected Icode value (CHECK_LENGTH).
Module CTL6	
500	Code Generator computed code size does not match the code size computed during link text emission (FINBIN).
501	Illegal class override, CS_DTYP record (BINPS2).
502	Illegal symbol type, CS_SYM record (BINPS2).
503	Internal label reference to an undefined label, CS__CJMP record (BINPS2).
504	Internal label reference to an undefined label, CS_ILAB record (BINPS2).
505	Internal label location does not match current location counter, CS_DILB record (BINPS2).
506	User label reference to an undefined label, CS_ULAB record (BINPS2).
507	User label location does not match current location counter, CS_DULB record (BINPS2).
508	P-code procedure/function entry address does not match current location counter, CS__PFBEG/CS__PROB record (BINPS2).
509	Procedure/functin entry address does nto match current location counter, CS_PFBEG/CS_PROB record (BINPS2).
510	Unknown Binary Interpass File record type (BINPS2).

Value	Meaning

Module DUBMP86 - 8086 Machine Code

600	Unexpected interpass record type (GETBYTE).
601	Unexpected end of data (GETDATA).
602	Illegal data size (GETDATA).
603	Illegal data size (GETDATA).
604	Unexpected end of data (GETDISP).
605	Unexpected interpass record type (GETDISP).
606	Unexpected end of data (GETLABEL).
607	Illegal label type, must be short lable (GETLABEL).
608	Unexpected interpass record type (GETLABEL).
609	Illegal opcode (WRITEOP).
610	Illegal opcode, no PUSH CS opcode exists (PUSHPOPSEG).
611	Can;'t do sign extension on operands for logical operators AND, OR, XOR (BINARYOPS).
612	Illegal mode value (LOADPTR).
613	Illegal mode value (SHIFTOPS).
614	Unused opcode (GROUPC).
615	Unused opcode (DUMP86).
616	Unused opcode (DUMP86).
617	Unused opcode (DUMP86).
618	Unused opcode (DUMP86).
619	Unused opcode (DUMP86).
620	Unused opcode (DUMP86).
621	Unused opcode (DUMP86).
622	Unused opcode (DUMP86).
623	Unused opcode (DUMP86).
624	Unused opcode (DUMP86).
625	Unused opcode (DUMP86).
626	Unused opcode (DUMP86).
627	Illegal data size (GETDATA).
628	Expected csalof record (INLINEDATA).

Module DUMP

700	Illegal opcode value (OPNAME).
701	Unknown varkind value (DMP1ID).
702	Unexpected symbol type (DMP1ID).
703	Illegal operator mode value (WRIMOD).
704	Unexpected Icode value (DMPNOD).
705	Unexpected interpass record type (DMPBREC).

If a run-time error occurs with a code not listed below, it may be a B 20 Operating System services status code. See Appendix A of the B 20 Operating System Manual.

1000-1100

These status codes are always issued in conjunction with a B 20 Operating System services status code.

Decimal Value	Meaning
1200	Format missing final ")"
1201	Sign not expected in input
1202	Sign not followed by digit in input
1203	Digit expected in input
1204	Missing N or Z after B in format
1205	Unexpected character in format
1206	Zero repetition factor in format not allowed
1207	Integer expected for w field in format
1208	Positive integer required for w field in format
1209	"." expected in format
1210	Integer expected for d field in format
1211	Integer expected for e field in format
1212	Positive integer required for e field in format
1213	Positive integer required for w field in A format
1214	Hollerith field in format must not appear for reading
1215	Hollerith field in format requires repetition factor
1216	X field in format requires repetition factor
1217	P field in format requires repetition factor
1218	Integer appears before + or - in format
1219	Integer expected after + or - in format
1220	P format expected after signed repetition factor in format
1221	Maximum nesting level for formats exceeded
1222	")" has repetition factor in format
1223	Integer followed by "," illegal in format
1224	"." is illegal format control character
1225	Character constant must not appear in format for reading
1226	Character constant in format must not be repeated

Decimal Value	Meaning
1227	"/" in format must not be repeated
1228	" " in format must not be repeated
1229	BN or BZ format control must not be repeated
1230	Attempt to perform I/O on unknown unit number
1231	Formatted I/O attempted on file opened as unformatted
1232	Format fails to begin with "("
1233	I format expected for integer read
1234	F or E format expected for real read
1235	Two "." characters in formatted real read
1236	Digit expected in formatted real read
1237	L format expected for logical read
1239	T or F expected in logical read
1240	A format expected for character read
1241	I format expected for integer write
1242	w field in F format not greater than d field + 1
1243	Scale factor out of range of d field in E format
1244	E or F format expected for real write
1245	L format expected for logical write
1246	A format expected for character write
1247	Attempt to do unformatted I/O to a unit opened as formatted
1251	Integer overflow on input
1252	Too many bytes read from input record
1253	Too many bytes written to direct access unit record
1255	Attempt to do external I/O on a unit beyond end of file record
1256	Attempt to position a unit for direct access on a nonpositive record number
1257	Attempt to do direct access to a unit opened as sequential
1260	Attempt to backspace unit connected to unblocked device
1264	Attempt to do unformatted I/O to internal unit
1265	Attempt to put more than one record into internal unit
1266	Attempt to write more characters to internal unit than its length
1267	EOF called on unknown unit
1268	Dynamic file allocation limit exceeded
1269	File name too long

Decimal Value	Meaning
1270	Video I/O error encountered while attempting to write lf to video display
1271	EOF function called on terminal device
1272	File operation attempted after error encountered on previous operation
1273	Keyboard buffer overflow: too many bytes written to keyboard input record (must be less than 132)
1297	Integer variable not currently assigned a format label
1298	End of file encountered on read with no END= option
1299	Integer variable not ASSIGNed a label used in assigned GOTO

APPENDIX B

DIFFERENCES BETWEEN BURROUGHS FORTRAN 77 AND ANSI STANDARD SUBSET FORTRAN 77

This Appendix describes how Burroughs FORTRAN 77 differs from the standard subset language. The standard defines two levels, full FORTRAN and subset FORTRAN. Burroughs FORTRAN is a superset of the latter. The differences between Burroughs FORTRAN and the standard subset FORTRAN fall into two general categories: full-language features, and extensions to standard.

Full-Language Features

Several features from the full language are included in this implementation. In all cases, a program written to comply with the subset restrictions compiles and executes properly, since the full language properly includes the subset constructs.

Subscript Expressions

The subset does not allow function calls or array element references in subscript expressions, but the full language and this implementation do.

DO Variable Expressions

The subset restricts expressions that define the limits of a DO statement, but the full language does not. Burroughs FORTRAN also allows full integer expressions in DO statement limit computations. Similarly, arbitrary integer expressions are allowed in implied DO loops associated with READ and WRITE statements.

Unit I/O Number

Burroughs FORTRAN allows an I/O unit to be specified by an integer expression, as does the full language.

Expressions in Input/Output List (iolist)

The subset does not allow expressions to appear in an I/O list whereas the full language does allow expressions in the I/O list of WRITE statements. Burroughs FORTRAN allows expressions in the I/O list of a WRITE statement providing that they do not begin with an initial left parenthesis.

NOTE: The expression (A+B)*(C+D) can be specified in an output list as +(A+B)*(C+D). Doing so does not generate any extra code to evaluate the leading +.

Expression in Computer GOTO

Burroughs FORTRAN allows an expression for the value of a computed GOTO, consistent with the full, rather than the subset, language.

Generalized I/O

Burroughs FORTRAN allows both sequential and direct access files to be either formatted or unformatted. The subset language restricts direct access files to be unformatted and sequential to be formatted. Burroughs FORTRAN also contains an augmented OPEN statement that takes additional parameters not included in the subset. There is also a form of the CLOSE STATEMENT, WHICH IS NOT INCLUDED IN THE SUBSET. I/O is described in more detail in Section 11 and 12. The READ and WRITE statements allow the optional ERR parameter.

Extensions to Standard

The implemented language has several minor extensions to the full-language standard. These are described below.

Compiler Directives

Compiler directives were added to allow the programmer to communicate certain information to the compiler. An additional kind of line, called a compiler directive line, has been added. It is characterized by a dollar sign, $, appearing in column 1. A compiler directive line is used to convey certain compiler-time information to the FORTRAN system about the nature of the current compilation. A compiler directive line may appear any place that a comment line can appear, although certain directives are restricted to appear in certain places. The set of directives is described in Section 14 above.

Backslash Edit Control

The edit control character can be used in formats to inhibit the normal advancement to the next record associated with the completion of a READ or a WRITE statement. This is particularly useful when prompting to an interactive device, such as the screen, so that a response can be on the same line as the prompt.

End of File Intrinsic Function

An intrinsic function, EOF, is provided. The function accepts a unit specifier as an argument and returns a logical value that indicates whether the specified unit is at its end of file.

Lowercase Input

Upper- and lowercase source input is allowed. In most contexts, lowercase characters are treated as indistinguishable from their uppercase counterparts. Lowercase is significant in character constants and Hollerith fields.

APPENDIX C
CALLING NON-FORTRAN PROCEDURES

Burroughs FORTRAN can call non-FORTRAN procedures that were compiled or assembled into standard object module format. Hence, you can access procedures written in Pascal or assembly language. Using this facility, you can call the B 20 Operating System, Burroughs software products such as Forms, ISAM, and Sort/Merge, and your own non-FORTRAN procedures.

Calls to non-FORTRAN procedures are mediated by the FORTRAN run-time system. The mediator converts parameter addresses on the stack into the form expected by the called procedure.

You call a non-FORTRAN procedure by its procedure name synonym, a FORTRAN name associated with the procedure. Synonyms permit FORTRAN to call procedures that do not have valid FORTRAN names (for example, names longer than six characters). Synonyms are defined outside of FORTRAN in an assembly language module name ForGen.Asm.

To invoke a procedure that does not return a value, use the CALL statement. To invoke a procedure that does return a value (a function) use the procedure name synonym in any arithmetic expression. See Chapter 12 for a description of invocation syntax.

To call non-FORTRAN procedures, you must link your application to ForGen.Obj, an assembly language module that defines the non-FORTRAN procedures. Creating ForGen.Obj is described in "Configuring FORTRAN" below.

SAMPLE CALLS TO NON-FORTRAN PROCEDURES

The following FORTRAN statements demonstrate calls to non-FORTRAN procedures. The first example calls Initialize and passes the value of i. It then calls NoOp, passing no parameters. The synonym Init is used.

```
CALL Init(i)
CALL NoOp
```

The second example calls the B 20 Operating System CloseFile operation, passing the value of iFh. The synonym FMCLOS is used. The returned status code is stored in iErc.

```
iErc = FMCLOS(iFh)
```

In the third example, shown below, a call is made to the B 20 Operating System services OpenFile operation, and the returned status code is stored in iErc. The synonym FMOPEN is used.

The OpenFile operation stores a file handle into the first parameter, which is passed by reference.

The second parameter, passed by reference, is a string of bytes naming the file to be opened. The third parameter, passed by value, is the length of the previous string.

The next two parameters describe a password and are identical in format to the previous two.

The final parameter, passed by value, is two bytes describing the file mode.

```
CHARACTER*8 filnam
CHARACTER*5 paswrd
CHARCTER*2 mode

DATA filnam/'testfile'/
DATA paswrd/'xyzzy'/
DATA mode/'mm'/

iErc = FMOPEN(iFh, filnam, 8, paswrd, 5, mode)
```

PARAMETER PASSING

Burroughs FORTRAN passes parameters either by reference or value, depending on the interface of the called non-FORTRAN procedure. (Parameters are passed on the stack. To pass a parameter by reference means to put a pointer to the parameter value on the stack; to pass by value means to put the value itself on the stack.) Only bytes, words, and doublewords can be passed by value. Note that in FORTRAN-to-FORTRAN calls, parameters are passed only by reference.

When FORTRAN calls a non-FORTRAN procedure using a synonym, control passes to a part of the FORTRAN run-time called the mediator. The mediator converts parameter addresses on the stack into the form expected by the interface of the called procedure. The mediator then passes control to the procedure associated with the synonym.

The FORTRAN mediator gets information about procedure interfaces from a module named ForGen.Obj. Creating ForGen.Obj is described in "Configuring FORTRAN" below.

Word-Aligned Data

Some non-FORTRAN procedures, such as OpenRsFile, require word alignment for buffers. Burroughs FORTRAN guarantees word alignment of data items, hence parameters passed by references are always word aligned.

> NOTE
>
> The procedures discussed in this section require certain files released only with the B 20 CUSTOMIZER. The B 20 FORTRAN compiler does not include these files.

To configure a FORTRAN in which non-FORTRAN procedures can be called, create a run file (FORTRAN.Run) that contains the application, the module defining the non-FORTRAN procedures (ForGen.Obj), and the actual non-FORTRAN procedures. This process is described below.

1. If you are simply configuring Burroughs software into your FORTRAN, skip this step.

   If you are configuring your own non-FORTRAN procedures into FORTRAN, invoke the Editor to modify ForGen.Asm, the assembly language module that defines rgProcedures, the lookup table of mediated procedures.

   Add an entry to ForGen.Asm for each of your non-FORTRAN procedures. Comments within ForGen.Asm explain how to add an entry.

2. Assembly ForGen.Asm to produce ForGen.Obj (see the Assembly Language Manual for details on invoking the assembler). During assembly, the assembler asks questions of this type.

   Are you calling Forms (y or n)?

   Are you calling Sort/Merge (y or n)?

   Are you calling ISAM (y or n)?

   Press RETURN after each response. If you answer y (for yes) to a question, the assembler creates an entry in rgProcedures for each procedure in the corresponding Burroughs package. This enables FORTRAN to call non-FORTRAN procedures.

3. Link ForGen.Obj, the FORTRAN application object modules, and the object modules for all non-FORTAN procedures to produce a run file. (See the Linker/Librarian Manual for details on invoking the Linker.)

Your FORTRAN application can now call non-FORTRAN procedures.

Unmediated Calls to Non-FORTRAN Procedures

You can call non-FORTRAN procedures directly without invoking the run-time mediator. However, the called procedure must meet the following requirements.

1. The name of the procedure must be a valid FORTRAN name.

2. Parameters of the procedure must be addresses only.

Mediated Calls to FORTRAN Procedures

When you use the FORTRAN CALL facility to call a FORTRAN procedure, the run-time mediator is not invoked. However, it is occasionally desirable to mediate FORTRAN-to-FORTRAN calls.

For example, suppose you want to allocate the largest possible short-lived memory segment (using the B 20 Operating System services AllocAllMemorySL operation; see the B 20 Operating System Manual) and pass it to a FORTRAN procedure. Because the memory segment is not named by a FORTRAN variable, FORTRAN cannot pass it as a parameter. However, you can make this work with the mediator.

Make an entry in ForGen.Asm for the called FORTRAN procedure. Specify that the parameter is an address, passed by value. Within the calling procedure, invoke AllocateAllMemorySL to allocate the memory segment and store its address in a FORTRAN variable. Then invoke the target procedure (with its synonym), passing the FORTRAN variable containing the address of the work area. The mediator adjust the stack, replacing the address of the FORTRAN variable with its value, the address of the memory segment. The mediator then passes control to the target procedure.

Consider the example below.

```
INTEGER cPar
CHARACTER*4 pPar
iErc = MMAASL(cPar,pPar)
CALL Target(pPar)
```

where:

pPar — is a 4-byte variable containing the address of the allocated memory segment.

MMAASL — is the synonym of AllocateAllMemorySL.

Target — is the synonym of the called FORTRAN procedure.

In calling Target, the mediator adjusts the stack, replacing the address of pPar with the value of pPar. Hence the allocated memory segment is passed to the called procedure.

STANDARD PROCEDURE NAME SYNONYMS

The list below defines FORTRAN procedure name synonyms for portions of the B 20 Operating System, and for DAM, Forms, ISAM, RSAM, SAM, and Sort/Merge. ForGen.Asm also defines these synonyms. These procedures must be declared INTEGER*2.

The first two letters of a synonym designate the software package containing the procedure. For example, all synonyms for Sort/Merge procedures begin with the letters SM.

Operation Name	Synonym
B 20 Operating System File Management - FM	
ChangeFileLength	FMCGFL
CheckReadAsync	FMCKRA
CheckWriteAsync	FMCKWA
ClearPath	FMCLRP
CloseAllFiles	FMCLAF
CloseAllFilesLL	FMCLAL
CreateDir	FMCRTD
CreateFile	FMCRTF
DeleteDir	FMDELD
DeleteFile	FMDELF
GetFhLongevity	FMGTFL
GetFileStatus	FMGTFS
GetUCB	FMGTCB
QueryWsNum	FMQYWS
QuietIO	FMQTIO
Read	FMREAD
ReadAsync	FMRDAS
ReadDirSector	FMRDDS
RenameFile	FMRNMF
SetFhLongevity	FMSTFL
SetFileStatus	FMSTFS
SetPath	FMSTPA
SetPrefix	FMSTPX
Write	FMWRIT
WriteAsync	FMWRAS

Operation	Synonym
B 20 Operating System Keyboard Management - KM	
Beep	KMBEEP
CheckpointSysIn	KMCHKP
DisableActionFinish	KMDSAF
QueryKbdLeds	KMQYKL
QueryKbdState	KMQYKS
ReadActionCode	KMRDAC
ReadKbd	KMRDKB
ReadKbdDirect	KMRDKD
SetKbdLed	KMSTKL
SetKbdUnencodedMode	KMSTUM
SetSysInMode	KMSYSI
B 20 Operating System Memory Management - MM	
AllocAllMemorySL	MMAASL
AllocMemoryLL	MMAMLL
AllocMemorySL	MMAMSL
DeallocMemoryLL	MMDMLL
DeallocMemorySL	MMDMSL
ResetMemoryLL	MMRSLL
QueryMemAvail	MMQYMA
B 20 Operating System OpenFile/CloseFile - FM	
CloseFile	FMCLOS
OpenFile	FMOPEN
B 20 Operating System Task Management - TM	
Chain	TMCHAI
ErrorExit	TMEREX
Exit	TMEXIT
LoadTask	TMLTSK
B 20 Operating System Timer Management - CM	
CloseRTClock	CMCLOS
CompactDateTime	CMCPDT
Delay	CMDLAY
ExpandDateTime	CMEXDT
GetDateTime	CMGTDT
OpenRTClock	CMOPEN
ResetTimerInt	CMRSTI
SetDateTime	CMSTDT
SetTimerInt	CMSTTI

Operation	Synonym
B 20 Operating System Video Access Method - VA	
PosFrameCursor	VAPSFC
PutFrameAttrs	VAPTFA
PutFrameChars	VAPTFC
QueryFrameChar	VAQYFC
ResetFrame	VARSTF
ScrollFrame	VASCRL
B 20 Operating System Video Display Manager - VD	
InitCharMap	VDINCH
InitVidFrame	VDINVF
LoadCursorRam	VDLDCR
LoadFontRam	VDLDFR
LoadStyleRam	VDLDSR
QueryVidHdw	VDQYVH
ResetVideo	VDRSTV
SetScreenVidAttr	VDSTSA
Direct Access Method - DA	
CloseDaFile	DACLOS
DeleteDaRecord	DADLRC
OpenDaFile	DAOPEN
QueryDaLastRecord	DAQYLR
QueryDaRecordStatus	DAQYST
ReadDaFragment	DARDRF
ReadDaRecord	DARDRC
SetDaBufferMode	DASTMD
TruncateDaFile	DATRNC
WriteDaFragment	DAWRRF
WriteDaRecord	DAWRRC
Forms - FO	
DefaultField	FODFFD
DefaultForm	FODFFO
DisplayForm	FODISP
GetFieldInfo	FOGTFI
LockKbd	FOLKKB
OpenForm	FOOPEN
ReadField	FORDFD
SetFieldAttrs	FOSTFA
UndisplayForm	FOUNDS
UserFillField	FOUFFD
WriteField	FOWRFD

Operation	Synonym
Indexed Sequential Access Method - IS	
CloseISAM	ISCLOS
CreateISAM	ISCRTF
DeleteISAM	ISDELF
DeleteISAMRecord	ISDLRC
GetISAMRecords	ISGTRS
InstallISAM	ISNSTL
LockISAM	ISLOCK
ModifyISAMRecord	ISMDRC
OpenISAM	ISOPEN
ReadISAMRecordByUri	ISRDUR
ReadNextISAMRecord	ISRDNX
ReadUniqueISAMRecord	ISRDUQ
RenameISAM	ISRENM
SetISAMProtection	ISSTPR
SetupISAMIterationKey	ISSTKY
SetupISAMIterationPrefix	ISSTPX
SetupISAMIterationRange	ISSTRG
StoreISAMRecord	ISSRRC
UnLockISAM	ISULCK
Record Sequential Access Method - RS	
CheckpointFile	RSCHKP
CloseRsFile	RSCLOS
GetRsLfa	RSGTFA
GetSTAMFileHeader	STGTHD
OpenRsFile	RSOPEN
ReadRsRecord	RSRDRC
ReleaseRsFile	RSRELS
ScanToGoodRsRecord	RSSCAN
WriteRsRecord	RSWRRC
Sequential Access Method - BS	
CheckpointBs	BSCHKP
CloseByteStream	BSCLOS
GetBsLfa	BSGTFA
InitCommLine	BSINCL
InitCommOptions	BSINCO
OpenByteStream	BSOPEN
PutBackByte	BSPBBT
QueryVidBs	BSQVID
ReadBsRecord	BSRDRC
ReadByte	BSRDBT
ReadBytes	BSRDBS
ReleaseByteStream	BSRELS
SetBsLfa	BSSTFA
WriteBsRecord	BSWRRC
WriteByte	BSWRBT

APPENDIX D
ADDITIONAL BUILT-IN FEATURES

The procedures below are contained in the FORTRAN library. Note that the functions beginning with the letter I are Integer*2 functions. Therefore the identifier must be explicitly declared an Integer*2.

1. BLDPTR

Syntax:

```
BLDPTR(iSa, iRa)
```

where
iSa is an Integer*2 expression
iRa is an Integer*2 expression

Action:

BLDPTR returns the memory address whose segment address is iSa and relative address is iRa.

Example:

To build a pointer to the element of the System Common Address Table that contains a pointer to the Video Control Block (VCB):

```
real ppVCB

ppVCB = BLDPTR(0,580)
```

2. GETPTR

Syntax:

```
DETPTR(arg)
```

where
arg is any FORTRAN variable

Action:

GETPTR returns the memory addresses of arg.

Example:

To get the memory address of i:

```
real p
integer i

p = GETPTR(i)
```

3. ICLRER

Syntax:

ICLRER(u)

where

u is any Integer*2 expression

Action:

ICLRER clears the error flags associated with the external file connected to unit u. This function allows further I/O to a file that has previously encountered an error. Note, however, that some runtime errors leave the file in an undefined state. Therefore the results of subsequent I/O to a file that has encountered an error may be indeterminate.

Example:

To clear all the error flags associated with the file connected to unit 10:

```
Integer*2 ICLRER
CALL ICLRER(10)
```

4. IGETER

Syntax

IGETER(u)

where

u is any Integer*2 expression

Action:

IGETER returns the error code associated with the external file connected to unit u.

Example:

To get the error code for file connected to unit 10:

```
Integer*2 IGETER, iError

iError = IGETER(10)
```

5. IGETRA

Syntax:

IGETRA(p)

where

p is any expression that evaluates to a memory address

Action:

IGETRA returns the relative address portion of p.

Example:

To get the relative address of i:

```
Integer*2 IGETRA, iError, iRa

iRa - IGETRA(GETPTR(i))
```

6. IGETSA

Syntax:

IGETSA(p)

where

p is any expression that evaluates to a memory address

Action:

IGETSA returns the segment address portion of p.

Example:

To get the segment address of i:

```
integer*2 IGETSA, i, iSa

iSa + IGETSA(GETPTR(i))
```

7. IPEEKB and IPEEKW

Syntax:

```
IPEEKB(p)

IPEEKW(p)
```

where
p is any expression that evaluates to a memory address

Action:

IPEEKB and IPEEKW return, respectively, the byte or word at memory address p.

Example:

To get the first byte and word of x:

```
real x
integer*2 IPEEKB, IPEEKW, iByte, iWord

iByte = IPEEKB(GETPTR(x))
iWord = IPEEKW(GETPTR(x))
```

8. POKEB and POKEW

Syntax:

```
POKEB(p, bData)

POKEW(p, wData)
```

where
p is any expression that evaluates to a memory address
bData is any integer expression
wData is any integer expression

Action:
POKEB and POKEW store, respectively, a byte or word at memory address p. In the case of POKEB, the value (bData MOD 256) is stored.

Example:

To store 0 into ch using a pointer:

```
character ch
real p

p = GETPTR(ch)

call POKEB(p, 0)
```

To store 0 into the word at memory address q:

```
call POKEW(q, 0)
```

APPENDIX E
GUIDE TO TECHNICAL DOCUMENTATION

This Manual is one of a set that documents the B 20 family of information processing systems. The set can be grouped as follows:

Introductory and Planning

Burroughs B 20
Your B 20 Installation Planning Guide

Hardware Installation

B 20 Hardware Installation Instructions
AP 1300 Printer, Installation, Operation, and Maintenance Guide
B 9251-1 Printer, Installation, Operation, and Maintenance Guide
B 9252 Printer, Installation, Operation, and Maintenance Guide
B 20 Cluster Work-Station, Installation and Operations Guide
B 20 Mass-Storage Unit, Installation Instructions for Qualified Service Personnel

Operations Training

B 20 Operations, Part 1 - Learning To Use the Hardware
B 20 Operations, Part 2 - Learning To Use the Systems Software
B 20 Operations, Quick Reference Guide

BASIC Language

B 20 BASIC Programming Language - An Overview
Introduction to B 20 BASIC Learning Guide
B 20 BASIC Language Reference Manual
B 20 BASIC Quick Reference Guide

Reference Manuals

B 20 System Software Operation Guide
B 20 COBOL II Reference Manual
B 20 Pascal Reference Manual
B 20 FORTRAN Reference Manual
B 20 English Version
B 20 Operating System Reference Manual
B 20 Debugger Reference Manual
B 20 Editor Reference Manual
B 20 Linker/Librarian Reference Manual
B 20 System Programmers Guide/Assembler
B 20 Font Reference Manual

Reference Manuals (continued)

B 20 Forms Reference Manual
B 20 ISAM Reference Manual
B 20 2780/3780 RJE Reference Manual
B 20 3270 Reference Manual
B 20 ATE Reference Manual
B 20 Sort/Merge Reference Manual
B 20 Batch Reference Manual

Following is a brief description of each B 20 manual:

INTRODUCTION AND PLANNING

Burroughs B 20 provides a general description of the B 20 system.

Your B 20 Installation Planning Guide offers suggestions to the new B 20 system owners about how to prepare for B 20 installation.

HARDWARE INSTALLATION

B 20 Hardware Installation Instructions provide step-by-step procedures on unpacking and installing the B 20 system.

AP1300 Printer, Installation, Operation, and Maintenance Guide provides instructions for unpacking, assembling, and using the B 20 AP1300 Printer.

B9251-1 Printer, Installation, Operation, and Maintenance Guide provides instructions for unpacking, assembling, and using the B 20 B9251-1 Printer.

B9252 Printer, Installation, Operation, and Maintenance Guide provides instructions for unpacking, assembling, and using the B 20 B9252 Printer.

B 20 Cluster Work Station, Installation and Operations Guide describes how to install and operate the B 20 cluster workstation.

B 20 Mass Storage Unit, Installation Instructions for Qualified Service Personnel provides complete instructions for unpacking and assembling the B 20 mass storage unit.

OPERATIONS TRAINING

B 20 Operations, Part 1 - Learning To Use the Hardware provides comprehensive, step-by-step guidance in learning to use the hardware.

B 20 Operations, Part 2 - Leaning To Use the Software provides comprehensive, step-by-step guidance in learning to use the B 20 software.

B 20 Operations, Quick Reference Guide provides quick, easy reference to questions that come up during B 20 operation.

BASIC LANGUAGE

B 20 BASIC Language - An Overview describes BASIC programming language and its uses on the B 20 system.

Introduction to B 20 BASIC Learning Guide provides comprehensive, step-by-step guidance in learning to use BASIC programming language on the B 20 system.

B 20 BASIC Language Reference Manual describes operating instructions for the B 20's BASIC programming language as well as a description of the language, itself.

B 20 BASIC Quick Reference Guide provides a summary and examples of all BASIC commands and functions.

The B 20 System Software Operation Guide describes the B 20 Executive, the program that first interacts with the user when the system is turned on. It specifies commands for managing files and invoking other programs such as the Editor, the programming language compilers, and communications interfaces.

The B 20 COBOL, FORTRAN, BASIC, and Pascal Language Reference Manuals describe the system's programming languages. Each manual specifies both the language itself and also operating instructions for that language.

The Pascal Manual is supplemented by a popular text, Pascal User Manual and Report.

The B 20 Operating System Reference Manual describes the Operating System. It specifies services for managing processes, messages, memory, exchanges, tasks, video, disk, keyboard, printer, timer, communications, and files. In particular, it specifies the standard file access methods: SAM, the Sequential Access Method; RSAM, the Record Sequential Access Method; and DAM, the Direct Access Method.

The B 20 Debugger Reference Manual describes the Debugger, which is designed for use at the symbolic instruction level. Together with appropriate interlistings, it can be used for debugging FORTRAN, Pascal, and Assembly language programs. (COBOL and BASIC, in contrast, are more conveniently debugged using special facilities described in their respective manuals.)

The B 20 Editor Reference Manual describes the text editor.

The B 20 Linker/Librarian Reference Manual describes the Linker, which links together separately compiled object files, and the Librarian, which builds and manages libraries of object modules.

The B 20 System Programmer's Guide/Assembler addresses the needs of the system programmer or system manager for detailed information on Operating system structure and system operation. It describes (1) cluster architecture and operation, (2) procedures for building a customized Operating System, and (3) diagnostics. It also describes the Assembly programming language.

The B 20 Font Reference Manual describes the interactive utility for designing new fonts (character sets) for the video display.

The B 20 Forms Reference Manual describes the Forms facility that includes (1) the Forms Editor, which is used to interactively design and edit forms, and (2) the Forms run time, which is called from an application program to display forms and accept user input.

The B 20 ISAM Reference Manual describes the multikey Indexed Sequential Access Method. It specifies the procedural interfaces and shows how these interfaces are called from the various langauges.

The B 20 2780/3780 RJE Reference Manual describes the 2780/3780 emulator package.

The B 20 3270 Reference Manual describes the 3270 emulator package.

The B 20 ATE Reference Manual describes the asynchronous terminal emulator.

The B 20 Sort/Merge Reference Manual describes (1) the Sort and Merge utilities that run as a subsystem invoked at the Executive command level, and (2) the Sort/Merge object modules that can be called from an application program.

The B 20 Batch Reference Manual describes the format of JCL files for invoking programs via the B 20 Batch Manager.

Documentation Evaluation Form

Title: B 20 Systems FORTRAN Reference Manual

Form No: 1148632

Date: February 1983

Burroughs Corporation is interested in receiving your comments and suggestions regarding this manual. Comments will be utilized in ensuing revisions to improve this manual.

Please check type of Suggestion:

☐ Addition ☐ Deletion ☐ Revision ☐ Error

Comments:

From:

Name _____

Title _____

Company _____

Address _____

_____

Phone Number _____ Date _____

Remove form and mail to:

Burroughs Corporation
TSG Product Support
Route 202 North
Flemington, N.J. 08822
U.S. America