Fortran 77

Table of Contents

1. INTRODUCTION

This manual describes the use and operation of the FORTRAN compilers which run under the UniFLEX™ and FLEX™ operating systems (UniFLEX and FLEX are trademarks of Technical Systems Consultants, Inc.).

FORTRAN is a high-level computer programming language. This particular system runs on the Motorola 6809 series of microcomputers. This manual describes the FORTRAN programming language features and its usage with the UniFLEX and FLEX operating systems.

No attempt is made in this manual to teach programming in FORTRAN; rather, the manual describes in detail the FORTRAN language and any details specific to its use with the operating system.

Fortran 77 requires the relocating assembler and linkage editor package in order to compile and execute Fortran programs. The user should obtain and be familiar with these products before using the Fortran system.

Technical Systems Consultants' Fortran conforms to ANSI FORTRAN-77 (ANSI X3.9-1978) subset of the FORTRAN language, with the following exceptions:

- The INTRINSIC and SAVE statements are ignored.
- The EQUIVALENCE statement is not implemented.
- The BACKSPACE statement is only allowed an direct access
- The ENDFILE statement performs no useful function.
- Statement functions are not supported.
- Variable names may be of any length with 7 characters significant.
- All keywords (see appendix E) are reserved in all contexts and must not be used as identifiers.
- Direct access files are not available under FLEX.

In addition, Technical Systems Consultants' Fortran contains some features of the full FORTRAN language, most notably list-directed I/O and expanded form of the OPEN statement.

Some notation is used throughout this document which should be explained here. Uppercase words are keywords and must be used exactly as they appear, although they may be spelled in either case. Lowercase words stand for some object defined by the user, either an identifier or expression, etc. Items enclosed within the characters "<" and ">" stand for some concept described in the surrounding text. Items enclosed within square brackets, "[" and "]", indicate an optional item which may or may not be present, depending on the user's wishes. The ellipsis, "...", is used to indicate that a given item may be repeated and when it follows an optional item; that item may occur zero or more times. E.g. the description:

```
SUBROUTINE sub [(parm [, parm]... )]
```

allows any of the following statements:

```
SUBROUTINE sub1
SUBROUTINE sub2(parm1)
SUBROUTINE sub3(parm2, parm3)
SUBROUTINE sub4(parm4, parm5, parm6)
```

11-A. Running the Fortran Compiler Under UniFLEX

The purpose of running the Fortran compiler is to take a program which is written in the FORTRAN programming language and produce a program which can be executed by the 6809 processor. This process involves many steps, most of which are automatic. The steps involved are:

1. Scan the FORTRAN source program and produce an assembly language file which corresponds to that program.

2. Assemble this file, creating a relocatable binary program.

3. Use the linkage editor to link this program with the FORTRAN run-time library, creating an executable program.

Steps 2 and 3 are optional and in no case will they be executed if your program contains errors.

The Command Line

The syntax of the command to execute the Fortran compiler is:

```
++ f77 file ...  [+options]
```

The plus signs are UniFLEX's prompt, "f77' is the name of the Fortran compiler. One or more input files may be specified, given by "file...".. If more than one file is specified, the result is the same as if all the files were appended in the order given. The options may appear anywhere on the command line and are any combination of the following:

a - Save the assembly language output file. This option will inhibit the compiler from automatically calling the assembler and linkage editor.

b - Suppress the creation of an assembly file. If this option is specified, the compiler will go through all the actions necessary to compile the given programs, but will produce no executable output.

c - Suppress calling the linkage editor. The compiler will create and assemble an assembly file and the resulting relocatable output file will be preserved. This is useful if the source files do not comprise a complete program and the linking process should be postponed.

d - Instructs the compiler to place additional code in the assembly language output file for debugging purposes. Primarily, this information will provide a post-mortem dump of all active variables in the program if it aborts for any reason. This option also implies the +t and +n options.

l - List the source program. If not specified, only those lines which are in error will be listed.

n - Keep track of the exact execution line number. This option will make the compiled program somewhat larger, but if a fatal error occurs, the systen will be able to indicate exactly the last line executed in the program. This option automatically implies the +t option.

o - Specify the final output file. Normally, the Fortran compiler will create the final (executable program with a name which is derived from the first source file name (see below). If this is undesirable, the user may specify the output name directly by using +o=XXX, where XXX is the desired output file name.

s - Print smumary information at the end of each module compiled. This option lists such information as thie time and size of each variable in the program, external programs which were called by this program segment, etc.

t - Keep information about the execution of the program and print an execution trace-back if a fatal error occurs. If the +n option (above) is not on, this will simply print the subroutines and functions which are active in the current program in reverse order. If the +n option is also on, the current line being executed in each program unit will be listed.

x - Generate shared text output from the linkage editor. Normally, Fortran will generate a "no-text" output file but this option allows those users who wish to do so to generate shared text

Source file names for Fortran programs must be of the form:

XXXX.f

The ".f" is REQUIRED! This file name is used to create all internal files used by the compiler system as follows

```
Source file:        XXXX.f
Assembly file:      XXXX.a
Relocatable object: XXXX.r
Executable output:  XXXX
Literals:           XXXX.s
```

The "Literals" file is a temporary file used by the Fortran compiler. It is automatically deleted by the compiler when the compilation is complete and as such, users should refrain from naming files "XXXY.s".

If more than one input file is specified, the first file name given is used to form these file names. The user has the ability, via the +c option, to specify the file name of the executable output file, but all others are given implicitly as above.

Examples:

The simplest example is one which compiles a Fortran program which is complete in one file, producing an executable file. An example of this would be:

```
++ f77 main.f
```

This command instructs the Fortran compiler to compile the program "main.f", then call the assembler and linkage editor to produce an executable program. Nothing except error messages will be printed on the standard output. The command line:

```
++ f77 main.f +ls
```

would be the same as the previous one, except that a listing of the program along with some summary information about each module within the program would be printed. Again, this command would create an executable program called "main".

Suppose you had a program which was contained in the files "program1.f", "program2.f", "program3.f" and "program4.f". The command line:

```
++ f77 program[1-4].f +o=program
```

would compile the files "program1.f", "program2.f", "program3.f" and "program4.f" in that order and create an executable program called "program".

Programs may be broken into modules and compiled together as in the example above, or they may be compiled separately, each creating its own binary file. In this case, the linkage editor must be invoked by the user to create an executable program from the pieces. An example of this would be:

```
++ f77 program1.f +c
++ f77 program2.f +c
++ f77 program3.f +c
++ f77 program4.f +c
++ link-edit program[1-4l.r +l=F77.runlib +no=progran
```

This sequence of commands would compile each of the modules separately and create a binary file for each one. This set of binary files would then be combined using the linkage editor to create an executable program. This example produces a program which is identical to the one produced by the example above.

Fortran programs may call external routines which have been written in another language, such as assembly language. The details of this mechanism are explained fully in Appendix C. Whenever this is done, the user is responsible for linking the modules together to create the executable program. An example of this might be when the user wishes to write his own random number function. Let us assume that he has done so and that the object file for this function is in the file "rnd.r". Now assume that the user's program is the same as above except that we wish to include this random number function instead of the library function. This could be accomplished by the commands:

```
++ f77 program[1-4].f +c
++ link-edit program[1-4].r rnd.r +l=F77.runlib +to=program
```

11-B. RUNNING THE COMPILER UNDER FLEX

In order to execute the Fortran compiler under the FLEX operating system, a source file must exist on a disk. This file may then be compiled, using the Fortran compiler, into an assembly language program. This assembly language program must then be assembled and linked with the Fortran runtime library, using the relocating assembler and linkage editor, respectively. The command used to invoke the Fortran compiler is:

```
+++ F77 <source-file> [+options]
```

The <source-file> is the name of the Fortran program which is to be compiled. It is assumed that this file ends with the extension ".TXT", but this may be overidden by specifying any other extension. The Fortran compiler uses several files based on the source file name, as follows:

```
XXXX.TXT - Source file
XXXX.LIT - Temporary file
XXXX.ASM - Assembly language output file
```

The file "XXXX.LIT" is used Internally by the Fortran compiler and will be deleted after the program has been compiled. Because of this, it is advised that the user refrain from naming programs "XXXX.LIT".

The options mentioned above are the same as the options available under UniFLEX, as described in the previous section. The options "a", "c" and "x" are not available under FLEX.

Installation Procedure

The following files must be on the "system" disk when the Fortran 77 compiler is executed. These files may be copied from the master disk to your normal system disk, or alternatively, they may be copied to a separate disk used only for the Fortran system.

Needed at Compile Time	Needed at Assembly/Link Edit Time
F77.CMD	RELASMB.CMD LLOAD.CMD
SYMBOLS.F77	F77.LIB (*)
ERRORS.F77	RUNLIS.F77 (*)

The files marked with an asterisk (*) must reside on the "work" disk, not on the 'system" disk.

In order to run a Fortran program, a mutli-step process must take place. First, the program must be compiled. This step takes a Fortran source program and produces a relocatable assembly language source file as output. The second step is to use the relocating assembler to assemble this file. This produces a relocatable object module. The last step is to use the linkage-editor to link all the desired object modules together, along with the Fortran runtime package, and produce an executable program. The following statements give a schema for this process:

```
+++ F77 file
+++ RELASMB file.ASM +sluy
+++ LLOAD file.REL +l.RUNLIB.F77 +o=file.CMD +a=0
```

In the schema above, "file" represents the name of the Fortran program being compiled. This schema assures that the entire Fortran program is composed of a single source file. If this is not the case, the first two steps may be repeated as many times as necessary to compile and assemble each of the modules of the program. The last line would then be extended to:

```
+++ LLOAD file-1.REL ... file-n.REL +l=RUNLIB.F77 +o-file.REL +a=0
```

In this schema, "file-1.REL" stands for the first object module of the program, 'file-2.REL" would be the second, and so on until "file-n.REL" which would be the last module of the program.

Refer to the Relocating Assembler and Linkage Editor manual for more details on how to run these programs. One thing to note is that in order to obtain an executable Fortran program, the "+a" option must be used with the link editor. Also, Fortran programs must be assembled using the "+u" option of RELASMB.

We have provided a test program on the Fortran master disk. The following are the commands necessary to compile and execute this test program.

```
+++ F77 svtest +l
+++ RELASMB svtest.ASM +sluy
+++ LLOAD svtest.REL +l=RUNLIB.F77 +o=svtest.CMD +a=0
+++ svtest
```

111. LANGUAGE CONSTRUCTS

This chapter describes the basic components of a FORTRAN program.

A FORTRAN program is composed of one or more program units. Each of these units may be compiled separately, using the +c option, or they may all be compiled together at one time. The advantage of separate compilation is that if only a subset of the total modules (or units - the terms are interchangable) changes, only those modules which have changed need be re-compiled. However, for simple programs, the advantage of single compilation is that the entire program can be compiled and loaded in one single command, producing an executable program. (Under FLEX, the user is responsible for assembling and linking the resulting modules together to form an executable program).

A program unit is either a MAIN program, a SUBROUTINE or a FUNCTION. The details of each of these units are described in later chapters. Note that a program must contain exactly one MAIN program and may contain one or more SUBROUTINEs or FUNCTIONs.

Each program unit is composed of records. A record In this case may be one or more lines. A line is a sequence of characters, terminated by a carriage return. If the information for a given record will not fit on one line, the record may be continued to another line. A single record may contain up to 10 lines. Each line has fields which have special meanings. Columns 1 through 5 comprise the label field. Column 6 is the continuation-marker field. Columns 7-72 are the text field.

For any given record, the first line may contain a value in the label field. This value must be a non-zero, unsigned integer. If the label field does not contain a label, this field must be completely blank.

If a record is to contain more than one line, all lines except the first must be marked as "Continuation" lines. This is done by placing a non-blank character in column 6. Any line which contains a non-blank character in column 6 is assumed to be a continuation of the previous line.

Columns 7 through 72 of all lines in every record comprise the text portion of the record and contain all program elements except record labels, as described above.

For the user's convenience, the horizontal tab character may be used to simplify the typing of FORTRAN programs. If a tab character appears, in any column of a line before column 7, the tab character is replaced by sufficient spaces so that the following character is in column 7. If the tab character appears after column 7, it is replaced by a blank. There is one exception to this rule: If the character immediately following the tab is the asterisk character '*', then the tab is really a tab to column 6 and the asterisk is placed in the continuation field of the line. The tab feature is not available under FLEX.

Any line which begins with any of the characters 'c', 'C', or '*' in column 1 is a comment line. Comment lines are totally ignored by the compiler and are only for use by humans. Also, comment lines are never part of a record and may not appear between continuation lines. A line with no characters other than the carriage return is also considered a comment line.

Programs are made up of symbols, some of which are:

- Keywords and identifiers
- Special symbols
- Character strings
- Numbers

Each of these symbols has specific syntax or construction rules. They will be explained in the sections below:

Keywords and Identifiers

These symbols are the words which are used to write a Fortran program. They must begin with a letter and may contain any number of letters or digits following the initial letter. Both upper and lower case letters are acceptable, but upper case will be implicitly mapped to lower case by the compiler. Thus the identifier "IDENT" is the same identifier is "ident". Although identifiers may be of any length in the program, only the first 7 characters are significant. Keywords must be spelled completely, with all characters significant.

Special symbols

There are some special symbols in a Fortran program. These are used in arithmetic expressions and as punctuation. They will be described in more detail in the sections which deal with these subjects. Notice that they must always be written exactly as they appear in this document. E.g. the symbol "**" is not the same as "* *".

Character strings

A character string is any sequence of characters, enclosed in single quotes. If a character string extends past column 72, then it is assumed that the line is continued and the first character of the next line in the record follows the character in column 72. It is not advisable to continue character strings in this fashion unless absolutely necessary. If it is desired to have the character string contain the single quote character itself, the quote must appear twice consecutively. Thus the string 'ab''cd' stands for the string "ab'cd".

Numbers

A number is a symbol which represents some numeric quantity. These quantities come in two varieties, integer numbers and real numbers. An integer is what is normally called a "whole" number. The numbers 1, 23, -256, are all integers. A real number is a number which may contain a fractional part. Real numbers are not the same as integers, even if they contain the same value. Some real numbers would be 3.14159. 6.02e+23 and 0.0. Notice that 0.0 is a real number but 0 is an integer. A number is an optional sign character ('-' or '+'), followed by a string of digits, optionally

followed by a decimal point followed by another string of digits, optionally followed by an exponent which is written as the letter 'e' followed by a signed integer. Any number which contains a fractional part or an exponent is considered to be a real number.

The Fortran standard defines that a FORTRAN program which has had all blanks removed (except from character strings and FORMAT statements) is equivalent to the same program with the blanks in it. This implementation uses blanks as separators and thus differs from the standard in this point. This is only significant in the fact that basic symbols (as described in the section above) may not contain blanks as would be allowed by the standard. Where there are keyword sequences which may include blanks, this implementation allows either spelling. E.g. "GO TO" is the same as "GOTO", but "ID ENT" is not the same as 'IDENT".

IV. DATA TYPES AND CONSTANTS

There are four types of data in FORTRAN programs. These are:

- Integer
- Real
- Logical
- Character

Each type is different and is used to represent different types of data.

Integers are whole numbers; 0, 1, 2, ...

Reals are numbers which may or may not contain fractional parts; 0.0, 3.14, -2.65, ...

Logical values are the values .TRUE. or .FALSE. These values indicate a "truth" value of either true or false.

Character data are strings of characters, built from the ASCII character set. Any printable character is allowed as an element of a character string.

Symbolic names (or identifiers) in a FORTRAN program have an implicit type associated with them. This type is based on the first character of the identifier. It may be overidden by the use of a "type" declaration as discussed in chapter VII. If not overidden, any identifier which begins with the character 'I' through the character 'N' is considered to be an integer. All other identifiers are considered to be reals. These implicit type definitions way also be changed by using the IMPLICIT statement as discussed in chapter VII.

FORTRAN programs may contain explicit constant values of the various types. These constants are written using the rules described in the previous chapter. The logical constants are written as ".TRUE." and ".FALSE.".

V. ARRAYS

An array is a nonempty sequence of data. An array element is one member of this sequence. An array name is the symbolic name of the entire array. An array element name is an array name qualified by a subscript.

Arrays are used to organize related data into an easily managed form. Any particular item from the set of data may be changed or used by simply indicating which item within the set it is.

An array is declared using an array declarator most any place a normal variable may be declared. For more information, see chapter VII on declarations. The form of an array declarator is:

```
     a(d [.d]...)
```

where:

a is the array name.

d is a dimension declarator which is of the form

```
          [b1:]b2
```

where b1 and b2 are integer expressions. If b1 is not present, the array will have elements which are numbered from 1 to b2, inclusive. If b1 is present, it must be less than b2 and the array will have elements which are numbered from bl up to b2, inclusive. An array declarator may contain one, two or three dimensions. If the array is being declared as a dummy array used as a parameter in a subprogram, then the array bounds may be declared using an '*' in place of a constant expression. This is because the actual bounds are not needed in the declaration of an array passed as a parameter.

An array declarator declares a collection of data items which are accessed by a common mechanism. This mechanism is known as array indexing. The general form of an array index expression (also refered to as an array subscripting expression) is:

```
     a(e [. e]...)
```

where:

a is the array name.

e is an Integer expression.

The effect of array indexing is to select a particular element of the array for use. An array element, given by a subscripted expression, is equivalent to a single simple variable.

Arrays are represented in the computer memory in an organized fashion, with the separate array elements in a specific order. For one dimensional arrays, this ordering is simple. Array element i "follows" element i-1 and "precedes" element i+l. For multiple dimensional arrays, the first dimension "varies most rapidly". That is, element (i, j) "follows" element (i-1, j) and "precedes" element (i+1, j). Array element (n, j) is "followed" by element (m, j+1), if "n" and "m" are the upper and lower bounds for the first dimension, respectively. For a three dimensional array, the process is similar.

VI. EXPRESSIONS

This chapter describes the formation, interpretation and evaluation rules for expressions. An expression may be any one of arithmetic, relational or logical expressions. Expressions are formed from operands, operators and parentheses.

Arithmetic Expressions

An arithnetic expression is made up of combinations of the following:

- Arithmetic Primary
- Arithmetic Factor
- Arithmetic Term
- Arithmetic Expression

Each of these components will be described below:

Arithmetic Primary

A primary is one of:

- Unsigned numeric constant
- Arithmetic variable reference
- Arithmetic array element reference
- Arithmetic function reference
- Arithmetic expression enclosed in parentheses

Arithmetic Factor

A factor is formed as:

- Primary
- Primary ** Factor

Arithmetic Term

A term is formed as:

- Factor
- Term / Factor
- Term * Factor

Arithmetic Expression

An arithmetic expression is formed as:

- Term
- + Term
- - Term
- Arithmetic expression + Term
- Arithmetic expression - Term

For arithmetic expressions, the operands must always be either of integer or real type. The operators described above are not defined on any other types.

The operators mentioned above have the following meanings:

Operation	Meaning
x** y	Exponentiate x to the power y
x / y	Divide x by y
x * y	Multiply x by y
x - y	Subtract y from x
- y	Negate y
x + y	Add x to y
+ y	Same as y

where: x denotes the operand to the left of the operator
y denotes the operand to the right of the operator

Evaluation of a given operator is described in the tables below:

Type and Interpretation of x + y

x \ y	Integer	Real
Integer	i = x + y	r = float(x) + y
Real	r = x + float(y)	r = x + y

Type and Interpretation for x ** y

x \ y	Integer	Real
Integer	i = x ** y	r = float(x) ** y
Real	r = x ** y	r = x ** y

In the tables above, 'i =' indicates that the result is of type integer and 'r =' indicates that the type is real.

Logical Expressions

A logical expression is one which yields a truth value, either .TRUE. or .FALSE. A logical expression may be made up of relational expressions or logical variables or constants, or combinations of logical expressions. Logical expressions are combined using the logical operators:

.AND. Logical conjunction
.OR. Logical disjunction
.NOT. Logical negation

Thus a logical expression has the fortn:

<logical expr>[<logical operator> <logical expr>]...

The logical operators have the following interpretation:

.NOT. This is a unary operator. The value of .NOT. <expr> is the inverse of the value of <expr>. Thus .NOT. .FALSE. is .TRUE. and .NOT. .TRUE. is .FALSE. The operator .NOT. has the highest precedence of the logical operators.

.AND. This is a binary operator between two logical expressions. Its meaning is: if both expressions have the value .TRUE., the expression has the value .TRUE. If either expression has the value

.FALSE., the expression has the value .FALSE. The logical operator .AND. is higher in priority than the operator .OR.

.OR. This is a binary operator between two logical expressions. Its meaning is: if both expressions have the value .FALSE., then the expression is .FALSE. If either expression has the value .TRUE., then the expression has the value .TRUE.

Relational Expressions

Relational expressions are used to compare two expressions and determine a logical value based on their relation. The two expressions must be of the same type, or be coercible to the same type. The result of a relational expression is always a logical value.

Relational expressions use the relational operators:

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

A relational expression is then:

<left-expr> <relational-operator> <right-expr>

The meaning of this expression is: if the <left-expr> is in the given relation to the <right-expr> then the relational expression has the value .TRUE., otherwise it will have the value .FALSE. A relational expression may be used any place a logical value is permissible, except in a DATA statement.

Precedence of Operators and Order of Evalutation

In the sections above, the various operators within the given types of expressions were discussed. The following section describes the precedence relations between different types of expressions. These precedence relations are based on the following rules, in order.

1. Parenthesized expressions
2. Precedence of operators
3. Right-to-left interpretation of exponentiations in a factor.
4. Left-to-right interpretation of multiplications and divisions in a term.
5. left-to-right interpretation of additions and subtractions in an arithmetic expression.
6. Left-to-right interpretation of conjunctions in a logical term.
7. Left-to-right interpretation of disjunctions in a logical expression.

As indicated above, parentheses may be used to alter this evaluation ordering since they are of the absolute highest precedence.

For example, using the rules above, the expression:

```
a + b * c - d
```

is interpreted as:

```
(a + (b * c)) -  d
```

VII. SPECIFICATION STATEMENTS

There are five kinds of specification statements:

- DIMENSION
- COMMON
- INTEGER, REAL, LOGICAL, and CHARACTER type statements
- IMPLICIT
- EXTERNAL

Additionally, the specification statements:.

- INTRINSIC
- SAVE

are allowed, but are ignored. Currently, the EQUIVALENCE statement is not supported.

All specification statements are non-executable and must appear before any executable statements or DATA statements within a program unit.

DIMENSION Statement

A DIMENSION statement is used to specify the symbolic names and dimension specifications of arrays. The form of a DIMENSION statement is:

```
DIMENSION a(d) [, a(d)]...
```

where each "a(d)" is an array declarator of the form:

```
a([l]:u [,[l:]u]... )
```

where "l" specifies the lower bound for a given dimension and "u" is the upper bound for that dimension. If "l" is not present, "l" is assumed. Refer to chapter V for a complete discussion of arrays.

Each symbolic name appearing in a DIMENSION statement declares "a" to be an array in that program unit. Note that array declarators may appear in COMMON statements and type-statements as well. Only one appearance of a name in an array declarator is permitted in a given program unit.

Examples:

```
DIMENSION a(100), b(0:99)
```

COMMON Statement

The COMMON statement provides a means of associating entitles in different program units. This allows different program units to share the same data without using parameters. The form of a COMMON statement is:

```
COMMON [/[cb]/] nlist [[,]/[cb]/nlist]...
```

where:

cb is a common block name. This name has no meaning within a program unit other than to name the common block and cannot be used in any other fashion. Common block names are significant to only 6 characters.

nlist is a list of variable names, array names, and array declarators. Only one appearance of a given name may appear within all the common lists within a program unit. Variable names which are also parameters or function names must not appear within the list.

If "cb" is not present, the common block is part of the "blank common" block.

For each COMMON statement, the variables in the list "nlist" are declared to be part of the common block "cb". The common block "cb" may be referenced by more than one COMMON statement, in which case, the list of variables is simply appended to the list of variables already associated with that common block.

When two separate program units contain a COMMON statement with the same common block name, "cb", the variables in each program unit defined within that common block will share the same address space. This is not true of variables not in common, as each variable in each program unit has its own address space, regardless of its name.

Examples:

```
COMMON /com1/ a, b(10), c
COMMON // d
```

INTEGER, REAL, LOGICAL and CHARACTER Type Statements

These statements are used to declare variables and associate a "type" with them. The form of the type-statement is:

```
<type> v [, v]...
```

where:

<type> is one of INTEGER, REAL, LOGICAL or CHARACTER.

v is a variable name, array name, array declarator, function name or dummy procedure name.

Examples:

```
INTEGER x1, abcdef, x2.
REAL zz, iz
CHARACTER*10 name
LOGICAL flags(0:8190)
```

IMPLICIT Statement

The IMPLICIT statement allows the programmer to override the default implied types for variables. Normally, all variables which begin with the letters I-N are considered to be of type Integer and all other variables are considered to be of type Real. If it is desired to have these implied types be something else, the IMPLICIT statement allows the programmer to specify this.

The form of the IMPLICIT statement is:

```
IMPLICIT type (a [, a]... ) [type(a C, a]... )]...
```

where:

type is one of INTEGER, REAL, LOGICAL, or CHARACTER[*len].

a is either a single letter or a range of single letters in alphabetical order. A range is indicated by the first and last letter of the range, separated by a minus sign. This is equivalent to writing every character within the range separately.

WARNING! The IMPLICIT statement will cause all identifiers which are not explicitly declared to be implicitly declared as specified in the IMPLICIT statement. This includes built-in functions such as SQRT, ATAN, etc. The user is advised to take care when using built-in functions in programs which make use of the IMPLICIT statement.

len is the [optional] length specification for entities of type CHARACTER. If 'len' is not specified, the length is one.

The IMPLICIT statement affects all variable declarations which do not explicitly declare a type for a given item. An IMPLICIT statement only applies to the program unit which contains it.

Examples:

```
IMPLICIT INTEGER(a-z)
IMPLICIT CHARACTER*4 (c, f-i, z)
```

EXTERNAL Statement

The EXTERNAL statement is used to identify a symbolic name as the name of an external procedure or dummy procedure, and to permit such a name as an argument. The form of an EXTERNAL statement is:

```
EXTERNAL proc [, proc]...
```

VIII. DATA STATEMENT

The DATA statement is used to specify initial values for variables at compilation time. A DATA statement is not executable and must appear after any specification statements and before any executable statements in a given program unit. The form of a DATA statement is:

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

where:

nlist is a list of variable names, array names, and array element names.

clist is a list of the form:

```
a [, a]...
```

where "a" is one of the forms:

```
c
r*c
```

c is a constant

r is an unsigned integer constant. The "r*c' form is equivalent to "r" successive appearances of the constant "c".

Names of parameters, functions and entities in COMMON must not be specified in the variable list.

There must be the same number of items specified in the data list "clist" as in the variable list "nlist". There is a one-to-one correspondence between elements of "nlist" and elements of the "clist". Specifying the name of an array in the list is equivalent to specifying each element of the array, with the first dimension being varied most rapidly. Also, if any elements of an array are specified. they must be given in ascending order. That is, the array element A(10) must not appear before the array element A(1).

The type of the element in the nlist and the corresponding element in the clist must be the same.

A variable or array element must not appear more than once in a DATA statement in a given program unit.

Character constants need not have the same length as the element they are meant to initialize. In this case, they will be padded or truncated on the right as needed.

Examples:

```
INTEGER a(10)
DATA a/1, 2, 8*0/

CHARACTER*10 lines(5)
DATA lines(1)/'line 1'/, line(5)/'*Full line'/
```

IX. ASSIGNMENT STATEMENTS

There are four distinct forms of assignment in FORTRAN:

- Arithmetic Assignment
- Logical Assignment
- Statement label Assignment
- Character Assignment

Arithmetic Assignment

This statement assigns the value of an arithmetic expression to a given variable. The format of this statement is:

<variable> = <expression>

This statement is executed by first evaluating the expression and then assigning the result value to the variable which appears to the left of the "-". If the expression and variable have different types, the expression is automatically converted to the same type as the variable before the assignment takes place. The following table outlines legal combinations of variables and expressions and the conversions which take place. The rows in the table correspond to the type of the <variable> in the assignment and the columns correspond to the <expression>.

	Integer	Real
Integer	Assign	IFIX(e)
Real	FLOAT(e)	Assign

Any other combination of types is illegal.

Logical Assignment

This statement assigns the value of a logical expression to a given variable. It has the same syntax as the arithmetic assignment, except that the variable must be of type logical.

Character Assignment

This statement assigns the value of a character expression (character variable or literal string) to a given variable. The syntax of the statement is the same as the arithmetic assignment statement. If the variable and the expression have the same length, then this statement simply assigns the expression to the variable. If the variable is shorter than the given expression, the expression is truncated on the right before assignment. That is, only the leftmost "w" characters from the expression are copied into a variable which is of length "w". If the variable is longer than the expression, the expression will be padded on the right with spaces to the length of the variable.

ASSIGN Statement and GOTOs:

The fourth form of assignment is the operation of assigning a label to a variable. This variable may then be used in a GOTO statement. This statement has the form:

ASSIGN <label> TO <variable>

This <variable> may later be used in a GOTO statement, as follows:

GOTO <variable>

The meaning of this statement is to "GOTO" the <label> which was specified in the most recently executed "ASSIGN" statement involving the given variable. Notice that this assignment is only valid within the current SUBROUTINE or FUNCTION and the variable should not be a

parameter or passed to any other routine.

X. CONTROL STATEMENTS

Control statements are used to control the execution of a program. There are sixteen control statements, as follows:

- Unconditional GO TO
- Computed GO TO
- Assigned GO TO
- Arithmetic IF
- Logical IF
- Block IF
- ELSE IF
- ELSE
- END IF
- DO
- CONTINUE
- STOP
- PAUSE
- END
- CALL
- RETURN

The CALL and RETURN statements are described in chapter XIV. The Assigned GOTO statement is found in chapter IX.

Unconditional GO TO

the form of an unconditional GOTO statnent is:

```
GO TO s
```

where "s" is the statement label of an executable statement which appears in the same program unit as this statement. Execution of the unconditional GOTO statement causes the statement whose label is "s" to become the next statement to be executed. The keywords "GO TO" may be written as a single word, "GOTO".

Computed GO TO

The form of a computed GO TO statement is:

```
GO TO (s [,' s]...) [,] i
```

where:

i is an Integer expression.

5 is the statement label of an executable statement within the same program unit as the computed GO TO statement. The sare statement label may appear more than once in the same computed GO TO statement.

Execution of a computed GO TO statement causes the expression "i" to be evaluated. If "i" has a value between 1 and "n", where "n" is the number of statement labels "s" appearing in the statement, then control is passed to the "i"'th statement label in the list. If "i" is outside this range, then control continues with the statement following the computed GO TO statement.

Arithmetic IF

The form of an arithmetic IF statement is:

```
IF (e) s1, s2, s3
```

where:

e is an integer or real expression

s1, s2, and s3 are each the statement label of a statement within the same program unit as the IF statement. The same statement label may appear more than once in the same IF statement.

Execution of an arithmetic IF statement causes the expression 'e" to be evaluated and if the result is less than 0, control is passed to the statement labeled "sl". If the result is "0", execution continues with the statement labeled "s2". If the result is greater than 0, control continues at "s3".

Logical IF

The form of a logical IF statement is:

```
IF (e) st
```

where:

e is a logical expression

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

Execution of a logical IF statement causes the expression "e" to be evaluated and if the result is ".TRUE.", the statement "st" is executed. If the result is ".FALSE.", execution continues with the following statement.

Block IF, ELSE IF, ELSE, and END IF

These statements are always used together to form an IF-block. The general form of the IF-block is:

```
IF(e) THEN
 <statement> ...
[ELSE
 <statement> ...
]
END IF
```

The block IF statement and END IF statements must always appear in matched pairs, with an optional ELSE statement between them. Execution of this statement causes the logical expression "e" to be evaluated and if it has the value ".TRUE.", the statements following the block IF statement are executed, up to, but not including the ELSE statement if present. All statements between the ELSE and the END IF statement are skipped if the expression is ".TRUE.". If the expression has the value ".FALSE.", then execution continues with the first statement following the ELSE statement if present, otherwise execution continues with the statement following the END IF statement.

If the last statement executed within a IF-block statement does not cause control to continue outside the block IF statement, then control automatically continues witli the statment which follows the END IF statement. Transfer of control into an IF-block from outside the IF-block is prohibited.

IF-block statements may be nested if desired. Also the construct:

```
     IF(e) THEN
       <statement> ...
     ELSE
       IF(e) THEN
           <statement> ...
       END IF
     END IF
```

may be abreviated as:

```
    IF(e) THEN
       <statement> ...
    ELSE IF (e) THEN
       <statememt> ...
    END IF
```

The keywords "ELSE IF' and "END IF" may be written as "ELSEIF" and "ENDIF" respectively if desired.

DO

A DO statement is used to specify a loop. called a DO-loop. The form of a DO statement is:

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

where

s is the statement label of an executable statement. This statement is known as the terminal statement of the DO-loop and must follow the DO statement in the sequence of statements within the same program unit as the DO statement.

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

e1, e2, and e3 are integer expressions.

The terminal statement of a DO-loop must not be an unconditional GO TO, assigned GO TO, arithmetic IF, logical IF, block IF, ELSE IF, ELSE, END IF, RETURN, STOP, END, or DO statement. Execution of a DO statement causes the expressions e1, e2 and e3 to be evaluated. The value of e1 is then assigned to the variable "i". The value of "i" is then compared to the value of "e2". If "i" is less than or equal to "e2" and "e3" is positive then the statements which follow the DO statement up through and including the terminal statement are executed. The value of "i" is then incremented by "e3" and control continues with the test previously described. Thus, the execution of a DO-loop will cause the statements between the DO statement and the terminal statement to be executed for all integer values of "i" between "e1" and "e2" in steps of "e3". If "e3" is not specified, a default value of "+1" is implied. If the value of "e3" is negative, the DO-loop continues execution as long as the value of "i" is greater than or equal to "e2". Thus the execution of a DO-loop may be considered similar to the following schemata:

```
          i = e1
     11   if((e3 .GE. 0) .AND. (i .GT. e2)) GOTO 12
          if((e3 .LT. 0) .AND. (i .LT. e2)) GOTO 12
            <statements within the DO-loop, including terminal statement>
          i = i + e3
          GOTO 11
     12   ---
```

Transfer of control out of the DO-loop is acceptable, but transfer of control into a DO-loop must never be done. Also, the DO-loop and any IF-blocks must not overlap in any manner. If a DO statement appears within another DO-loop, the range of the DO-loop specified by that DO statement must be completely within the range of the outer DO-loop. Nested GO-loops may share a common terminal statement.

CONTINUE

The form of a CONTINUE statement is:

```
     CONTINUE
```

Execution of a CONTINUE statement has no effect unless the statement is the terminal statement of a DO-loop.

STOP

The form of a STOP statement is:

```
     STOP [n]
```

where the constant "n" is an integer or a character string. Execution of a STOP statement causes the string "n" or the value "n" to be displayed at the user's console and the program terminated.

PAUSE

The form of the PAUSE statement is:

```
     PAUSE En]
```

where "n" is an integer constant or a character string. Execution of a PAUSE statement causes the given value "n" to be displayed on the user's terninal and the program suspends execution until the user enters a carriage return on the terminal. Execution then continues with the statement following the PAUSE statement.

END

The END statement indicates the end of a program unit. The END statement must be present for each program unit and has the effect, if executed, of a STOP statement within the main program, or a RETURN statement within a FUNCTION or SUBROUTINE. The form of the END statement is:

```
     END
```

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

XI. INPUT/OUTPUT STATEMENTS

There are several Input/Output statements in FORTRAN. These provide the user with mechanisms to read and write formatted or free format data. Also, there is a mechanism for doing "direct" access to a data file in a more or less random fashion. Each of these alternatives will be explored in the following sections.

List-directed Input/Output

These statements allow the FORTRAN programmer to perform sequential input or output from a file of character data. These data are then interpreted (on input) to represent numbers or characters as necessary. On output, the data are converted to a form readable by humans as well as being usable by other programs. The general form of these statements is:

```
READ *, <io-list>
PRINT *, <io-list>
```

An <io-list> is any non-empty list of <io-element>s, separated by commas. An <io-element> is a variable name or expression, including character strings, or an implied do-loop construct described later. An example would be:

```
CHARACTER*6 name
INTEGER age
READ *, name. age
```

This statement would cause the program to read a 6 character string from the standard input followed by an integer value representing an age. The rules for the form of this input data follows later.

In the above example, simple variables were being read into. The READ statement requires that all the <io-element>s be variable names or variable expressions, such as array element references. Complete arrays may also be used as an <io-element>. In this case, the entire array will be read into, with the first dimension being varied most rapidly. If the user waited to read more than one value into an array, but did not want to read every element in the array, he might choose to use the Implied do-loop construction given below:

```
(exp-1, exp-2, ..., exp-n, i=1,n)
```

The meaning of this expression is to evaluate "exp-1" for the value of the variable '1" equal to 1, then evaluate "exp-2" with "i" equal to 1, etc. Then evaluate "exp-1" with 'i" equal to 2, and so on.

An example of using an implied do-loop construct would be:

```
INTEGER a(100)
READ *, (a(i), i=1,10)
```

This statement would read 10 integer values into the first 10 elements of the array "a".

The READ statement for list-directed input interprets the input data differently based on the type of item being read. The rules for this interpretation are as follows:

Character

Character strings must be enclosed in single quotes, the "'" character. Any characters precedeing the opening quote are ignored. The character variable is then filled with characters until full or the closing quote character is found. If the closing quote appears before the variable is full, it will be padded on the right with spaces until full. If the variable becomes full before the closing quote appears, all characters past the point at which the variable became full up to the closing quote are ignored. The closing quote character must always be followed by some termination character, such as a space or a carriage return. This terminator will also be consumed so that the next element to be read will start with the second character which follows the closing quote. If it is desired to include the quote character in thh string being read, it is necessary to place two quote characters continuously in the string, just like in a program text. These two characters will then be read as a single quote character. Carriage returns are totally ignored during the reading of character data.

Integer

An integer value is represented by any number of leading spaces and carriage returns, including zero, followed by an optional sign character ('-' or '+') followed by a string of digits '0' through '9'. This string is interpreted to be a decimal number with the sign indicated, no sign meaning a positive value.

Logical

A logical value is represented by any number of leading spaces or carriage returns, followed by the character "T" or "t" indicating .true., or "F" or "f " indicating .false. This value may be preceded by a period, ".", character and followed by a string of characters up to a separator which must be a space or a carriage return. Thus the string ".TRUE." is an acceptable input for a logical input value.

Real

A Real value is represented by an optional sign, followed by a string of digits, optionally containing a decimal point, optionally followed by an exponent of the form 'e' followed by an optionally signed integer. On input, the value may be preceded by any number of spaces or carriage return combinations and must be terminated by a space or carriage return. On output, the value will be formatted in the most convenient form, using decimal notation when possible and scientific notation when necessary.

Note that all values read must be terminated, either by a space or by a carriage return. After the last item in the <io-list> has been read, the rest of that record in the input file is discarded and the next READ statement will begin at the start of the next record in the file.

The PRINT statement performs the inverse of the READ statement, following much the same rules, except that character strings are not delimited by quote characters. For example.

```
INTEGER age
age = 96
PRINT *. 'The old man is ', age, ' years old.'
```

would print the line:

```
The old man is 96 years old.
```

on the standard output file. Note that the integer value "96" was printed with a number of leading spaces. This is because PRINT will always use the same number of space s to print an integer value, regardless of its magnitude. Logical and Real values also have specific field widths which are always used. Character strings only print the minimum number of characters required to print the string.

Formatted Input/Output:

The list-directed statements described above are fine for simple things, but they have their drawbacks. One of the most important of these is the fact that the user has little control over the format of the resulting output when using the PRINT statement. This limitation can be overcome by using FORMATted operations. The general form of these statements is:

```
READ f, <io-list>
PRINT f, <io-list>
READ(u, f [, END=n] C, ERR-n]) <io-list>
WRITE(u, f [, ERR-n]) [<io-list>]
```

where:

f Stands for a FORMAT number. This must be the label of a FORMAT statement or the name of a character string which contains a FORMAT string. The details of FORMAT are given in the following chapter.

u Stands for a unit number. This must be an integer expression or the special character "*" or a character variable name. In the case that this is the character "*". the unit is a predefined unit, normally used for the desired operation. E.g. "*" in a READ statement is the same as specifying the unit number 5, the standard input. "*" in a WRITE statement stands for the unit number 6, the standard output file. If the unit is specified as a character variable, no actual Input/Output will take place, but the system will treat the given character variable as the source for input on a READ statement and the destination for output from a WRITE statement. This allows FORMATted conversions using only memory-to-memory operations. This mechanism replaces the old "ENCODE"/"DECODE" routines of FORTRAN-66.

END-N When this option is present in a READ statement, "n' must be a statement label. If the end of file is encountered on the given unit before all <io-element>s have been read, control is transferred to the given label. The values of all the

<io-element>s in the list are undefined if this action takes place. If this option is not present and the end of file is encountered, the program will be aborted.

ERR=n This option allows the program to regain control if any errors occur during the processing of the I/O statement. Again, all values in the <io-list> are undefined on input if this occurs. If this option is not present, the program will be aborted.

These statements work much the same as the list-directed statements except that the way the character data is interpreted/produced depends on the FORMAT statement provided. The details of FORMAT specifications may be found in the following chapter. As these statements are being processed, the FORMAT specification is scanned and there must be a one-to-one correspondence between <io-element>s and FORMAT items. The FORMAT item then defines how the transfer to/from the <io-element> is to take place. An example of this is:

```
    WRITE(*, 100)'Hi there', 90
100 FORMAT(//, A, ', today is Jan 1, 19', I2, '.')
```

would print the line:

```
Hi there, today is Jan 1, 1990.
```

preceded by two blank lines. Refer to the following chapter for the details of FORMAT specifications.

OPEN statement:

The OPEN statement allows the FORTRAN programmer to create an association between a unit number used in Input/Output operations and an external file. This association must be established before any operations may be performed on any unit, except the standard units. The association for these standard units has already been defined before the program begins execution, as follows:

0 - Standard Error (user's terminal)
5 - Standard Input
6 - Standard Output

The definition of what "Standard Input/Output" means depends upon the operating system. For UniFLEX, the Standard Input and Output files are defined using the shell. Under FLEX, all these files are associated with the user's terminal. All other files must be "opened" explicitly before use. The general form of the OPEN statement is:

```
OPEN(u [, FILE='name'])
```

Execution of this statement will open unit number "u', associating that unit with the file "'name"'. If the "FILE=" parameter is not given, a default name will be used. This default name is "fortnn.dat", where "nn" is the unit number being opened. If the unit is open when the OPEN statement is executed, it will be closed and then processing continues as normal

Direct-Access Input/Output:

The READ and WRITE statements may also be used to transfer varies between memory and files without conversion to/from external form. This is known as direct access Input/Output. The general mechanism is much the same as before, except that as values are transferred, they are not interpreted in any fashion. Thus the user may write a Real number to a file and actually get the 8 bytes which are the value written to the file, not an approximate value which has been formatted so that humans

can read it. To put it another way, this mechanism allows the FORTRAN programmer to perform I/O operations using the "natural" form of values, not their "human" counterparts. This can be very useful for keeping files of data which are to be manipulated. The values kept on the file will always be as accurate as possible, since no conversions are taking place.

The method by which the FORTRAN programmer performs this kind of operation is simple - just leave out the FORMAT specification in the I/O statement. I.e.

```
READ(u [, END=nn][, ERR=nn])<io-list>
WRITE(u [,ERR=nn])<io-list>
```

Units which are to be accessed directly by this mechanism must be opened as "direct access" files. This is accomplished through the use of the OPEN statement, as follows:

```
OPEN(u [, FILE='name'], ACCESS='direct', RECL=nn)
```

The function of this statement is to identify unit ":j" as a direct access file, optionally named by the user, with a record length of "nn". Notice that for direct access files, a record length must be present. The record length is used by the system and all operations on this file will use a multiple of this length bytes.

Direct access files may be accessed sequentially, as above, or specific records of the file may be accessed directly using the form of READ and WRITE given below:

```
READ(u, REC=nn  END=m][, ERR=m])<io-list>
WRITE(u, REC=nn ERR='m')<io-list>
```

The effect of these statements is to position to record number "nn" before performing the given operation. Records are numbered 0, 1, ... The READ statement would then read record "nn" and the WRITE statement would then write to record "nn". If the <io-list> requires more than one record, multiple records will be transferred, but always a multiple of the record length bytes will be transferred.

Auxiliary Input/Output Statements:

There are three other statements associated with Input/Output operations. These are:

```
REWIND u
BACKSPACE u
ENDFILE u
```

The purpose of each of these statements will be described below.

The REWIND statement repositions the given unit at the beginning of the data. The file may then be read or written, using the READ and WRITE statements.

The BACKSPACE statement repositions the file so that the next operation begins with the preceding record. This operation is only available for direct access files. If the file is already positioned at the beginning of the data, the BACKSPACE statement has no effect.

The ENDFILE statement is accepted by the Fortran system, but performs no function. This statement normally indicates that an "End of File" mark is to be placed at the current position in the file.

XII. FORMAT SPECIFICATION

The FORMAT string is used during FORMATted Input/Output, as described in the last chapter. This chapter outlines the rules for formation of FORMAT strings and how the specific FORMAT items are interpreted.

The general form of a FORMAT string is:

(<format-item-list>)

where <format-item-list> is a list of one or store <format-item>s, separated by commas. Each <format-item> may be any of the following:

nA[w]
nlw
nfw.d
new.d
nlw
nX
/
$
BN
BZ
kP
nhaaaaaaaa (n characters)
'aaaaaaaa'
n(<format-item-list>)

In the above list, "n" indicates a positive non-signed integer or an empty string which indicates how many times the following FORMAT item is to be used. For example, the FORMAT item "5I3" is the same as the FORMAT items "I3,I3,I3,I3,I3". The FORMAT code in a FORMAT item may be written in either upper or lower case. That is the item "i3" is equivalent to "I3". The meaning of each of these FORMAT items is given below:

A[wl

This FORMAT item is used to transfer character (alphanumeric) data. If the "w" value is present, it must be a non-signed integer value and exactly that many characters will be transferred. If the "w" value is not present, only the number of characters necessary will be transferred.

On input, if "w" is not present, enough characters will be transferred to fill the given variable. If "w" is present and "w" is greater than the size of the character variable, the variable will be filled and the regaining characters discarded. If "w" is present and is less than the size of the character variable, "w" characters will be transferred with the variable being padded on the right with spaces.

On output, if "w" is not present, enough characters will be transferred to output the entire expression. If "w" is present and is greater than the size of the expression, enough spaces will be output preceding the value so that the entire field is "w" characters. If "w" is present and less than the expression size, only the left-most "w" characters from the expression will be

transferred.

Iw

This FORMAT item specifies an integer value, Occupying exactly "w" characters. These characters will be an optional sign ("-"), followed by a string of digits '0' to '9' which represent the value. The field may contain any number of leading spaces necessary to complete the total "w" characters. On Input, if there are any trailing spaces, i.e. spaces which occur after the last digit character, but within the "w" characters, these are interpreted according to the current blank interpretation mode. See the BN and BZ FORMAT item description for details.

Lw

This FORMAT item indicates that a Logical value should be interpreted using "w" characters. This string of characters may contain leading spaces, an optional period character, followed by the character "T" or "t" indicating .true. or "F" or "f" indicating .false.. On output, the "T" or "F" will be right justified in the field with "w-1" leading spaces.

Fw.d

This FORMAT item represents a floating point value which occupies "w" total characters, exactly "d" of them being to the right of the decimal point. The value may be preceded by zero or more spaces, followed by an optional sign, followed by a string of digits, followed by a decimal point, ".", followed by "d' digits.

Ew.d

This FORMAT item indicates that a value is to be formatted using scientific notation. On input, the FORMAT item is exactly the same as the "Fw.d" item. On output, this item indicates that "w" characters should be used to form the field, with cxactly "d" digits in the mantissa. The remaining character positions are used for the sign and exponent values. Because of this, "w" should be at least "d"+6.

X

This FORMAT item indicates that the next character position is to be ignored. On input, this means that the next input character is simply skipped. On output, this item indicates that a blank character should be inserted at the current character position.

/

This FORMAT item indicates that the next operation should continue on the next record. On input, this means to discard the remainder of the current input record and continue processing with the start of the next input record. On output, the next operation will begin on a new line.

'aaaa'

This FORMAT item is only allowed in an output FORMAT string and indicates that the character string within the quotes should be output at the current position in the outfit record. This is useful for placing text information within other formatted data

items.

nHaaaa
This FORMAT item is equivalent to 'aaaa', where there must be exactly "n" characters in the string which follow the "H". Care must be used with this form of "Hollerith" output because if the count is not correct, disasterous things will occur. The 'aaaa' FORMAT item Is much recommended.

BZ and BN
These FORMAT items control how blanks are interpreted during input using I, F and E FORMAT items. If the "Blank mode" is "BZ", all trailing blanks are treated as if they were zeroes. If the "Blank mode" is "BN", trailing blanks are ignored. "BZ" is the default mode.

kP
This FORMAT item controls the scaling of input and output values using the "F" or "E" FORMAT items. "k" must be an optionally signed integer value. On input, if there is no exponent in the field, the input value is scaled (multiplied) by a value of 10**k. On output, the value is first scaled (divided) by a value of 10**k. The default value for "k" is 0.

$ This FORMAT item indicates that when the current output operation completes, the next operation should continue on the same line. This is useful for interactive input/output from a terminal when a response is desired on the same line as a prompt. E.g.

```
          CHARACTER*40 name
          WRITE(*, 100)
      100 FORMAT('Enter your name: ', $)
          READ(*, 101) name
      101 FORMAT(A40)
```

would produce the following exchange at the terminal (user input underlined).

```
     Enter your name: :ul San Jones
```

n(<format-item-list>)
This FORMAT item indicates that the entire <format-item-list> within the parentheses should be used "n' times. This is a more general form of "nZ" described above for any arbitrary FORMAT item.

When processing an Input/Output statement, there is a correspondence between FORMAT items and elements in the <io-element-list>. As the statement is processed, a new <io-element> is determined. This item is then transferred either from a unit to memory during a READ statement or to a unit during a WRITE operation. A corresponding FORMAT item must be found for this element. This is done by scanning, from left to right, until a FORMAT item is found which can be used. If any of the items XPH/$' are encountered, they are processed but do not correspond to any data element. The FORMAT item is then used to direct the Input/Output transfer, as described above. When the entire <io-element-list> has been processed, the rest of the current record is discarded on input and on output a new record will be started.

If the FORMAT string is exhausted before all transfers take place, it will be reused as follows: The FORMAT will be restarted at the left parenthesis which corresponds to the rightmost right parenthesis before the parenthesis which closes the FORMAT list, or the leftmost parenthesis if there is none. The FORMAT is restarted after going to a new record, as if a "/" FORMAT item had been executed just before the restart point. An example may make this more clear. The statement:

```
        WRITE(*, 100)(i, i=1,10)
    100 FORMAT('The first 10 integers are',/,(3i5))
```

would result in the lines:

```
    The first 10 integers are
        1    2    3
        4    5    6
        7    8    9
       10
```

FORMAT Statement

The FORMAT statement is used to give a FORMAT specification a label so that it may be used by the Input/Output statements of the previous chapter. As seen in the examples above, the form of this statement is:

```
    <label> FORMAT<format-string>
```

FORMAT statements must have a label which is unique within the program unit. This label is then used to reference the FORMAT string.

XIII. MAIN PROGRAM

Every complete FORTRAN program must contain exactly one "main" routine. This routine is indicated by not being a FUNCTION or SUBROUTINE definition. The general form of the "main" program is then:

```
     [PROGRAM name [(argument [, argument]...)]]
     <declaration statements>
     <executable statements>
     END
```

If a PROGRAM statement is present, it must contain a name to be given to the program. This name is effectively ignored. If any arguments are present, they should conform to the argument mechanism of the operating system. Examples will be given below for the various systems. These arguments provide a mechanism for the user to communicate with the program when it is envoked.

UniFLEX Example:

Consider the following program:

```
     PROGRAM prog1(count, arg0, arg1, arg2)
     INTEGER count
     CHARACTER*32 arg0, arg1, arg2
     CHARACTER*32 file1, file2
     IF(count .NE. 3)STOP 'Wrong number of arguments!'
C  Arguments must be copied before use
     file1 = arg1
     file2 = arg2
     call copy(file1, file2, ier)
```

In this example, the variables "arg0", "arg1" and "arg2" correspond to the exec arguments used when this program was initiated. If the program was run via the shell, these arguments correspond to the comand line arguments. In this case, "arg0" corresponds to the program name , with "arg1" and "arg2" being additional parameters. This program would be stalled via a command such as:

```
      ++ prog fnl fn2
```

FLEX Example:

Under the FLEX operating system, a single argument is available to the program. This argument, which is an array of 80 characters, is a blank-filled image of the command line which invoked the program, starting with the character following the program name. Thus a program running under FLEX might look something like:

```
      PROGRAM. main(cmd)
      CHARACTER*1 cmd(80)
       ...
      END
```

Any statement may be written in a "main" program except the "RETURN" statement.

XIV. FUNCTIONS AND SUBROUTINES

SUBROUTINES and FUNCTIONS are the FORTRAN mechanism for breaking programs into small, easily used and understood modules. These routines contain pieces of the program which may be "called" from some other part of the program to perform some particular task. They are particularly useful when the same operation must be performed repeatedly, perhaps by different portions of the program. Routines cone in two varieties, SUBROUTINEs which simply perform some operations and FUNCTIONs which perform similar operations but also explicitly return a value.

Before any routine can be used, it must be described through the SUBROUTINE or FUNCTION declaration. The general form of these declarations is:

```
     SUBROUTINE name [(parm1 [, parm2]...)]
     <declaration statements>
     <executable statements>
     END
or
     [<type>] FUNCTION name(parm1 [, parm2]...)
     <declaration statements>
     <executable statements>
     END
```

This declaration serves many purposes:

1. It gives the routine a name. This is the name which must be used to access the routine from some other point in the program. A given routine must not, either directly or indirectly, call itself.

2. It provides the declaration of the parameters to the routine. These parameters are the mechanism by which this routine communicates with the routine which called it. Notice that FUNCTIONs must have at least one parameter but SUBROUTINEs may have zero. Parameters are discussed in more detail later.

3. It outlines the statements which are to be executed when the routine is invoked (called).

SUBROUTINEs and FUNCTIONs must be "called" for the statements within them to be executed. This is accomplished by using the "CALL" statement for SUBROUTINEs and by using the FUNCTION identifier in an expression. The form of these are:

```
     CALL sub [(arg1 [, arg2]...)]
or
     var = fun(arg1 [, arg2]...)
```

The CALL statement works like this: When a CALL statement is executed, the expressions which represent the arguments, "arg1" etc., are evaluated. Then control is passed to the routine named "sub". In this routine, a correspondence between the arguments which were just evaluated and the parameters listed on the SUBROUTINE statement is established. Execution then continues with the first executable statement within the routine. When the last statement of the routine

has been executed and the next statement to be executed would be the "END" statement, control returns to the statement following the original CALL statement. Control may also be returned by using the "RETURN" statement anywhere within the routine. A FUNCTION activation works much the same way except that at some point in the FUNCTION itself, a value must be assigned to the FUNCTION identifier. When the FUNCTION returns control, this value is used in the place of the FUNCTION call in the calling expression.

Note that FUNCTIONs return a value but SUBROUTINEs do not. Some examples should make this mechanism more clear.

Consider the declaration:

```
     SUBROUTINE add(x, y, sum)
     INTEGER x, y, sum
     sum = x + y
     RETURN
     END
```

The purpose of this SUBROUTINE is to add the value of the first two parameters and return the value in the third argument. This routine might be called using the following statement:

```
     CALL add(1, 2, i)
```

where "i" is a variable of type INTEGER. When the CALL statement is encountered, the values of the arguments are computed. In this case, this is easy since they are constants, but they could be arbitrarily complex expressions. Then the SUBROUTINE is entered, establishing the correspondence between arguments and parameters. In this case, the parameter "x" stands for the value "1" and the parameter "y" stands for the value "2". The parameter "sum" stands for the variable "i" in the calling program. This means that anywhere inside the SUBROUTINE that "sum" is referenced, the routine is really referencing "i" from the calling routine. This is especially important when we consider what the statement

```
     sum = x + y
```

really means. The assignment will actually change the value of "i" in the routine where the CALL statement was that called "add"! Notice that since a SUBROUTINE can change the value of a variable in the calling routine, care must be taken to insure that arguments which correspond to parameters which are going to be modified must actually be variables and not expressions. This means that the statement:

```
     CALL add(l, 2, 7)
```

doesn't make any sense because "add" will try to store the value of the sum in the constant 7. Since there is no way to enforce that "add" is called with a variable as the third argument, the result of such a call is undefined.

Similar results may be obtained by using a FUNCTION declaration if it is desired to return exactly one result as in the above example. The program segment below shows this:

```
     INTEGER FUNCTION add(x, y)
     INTEGER x, y
     add = x + y
     RETURN
     END
```

This routine would be called by in same sort of INTEGER expression, e.g.

```
     i = add(1, 2)
```
again, assuming that "i" was an INTEGER variable. The one problem with this approach is that implicitly the type of the external function "add" would be REAL, unless the IMPLICIT statement had been used to change this. There is a way to indicate that "add" is really an external FUNCTION and that it is of type INTEGER. This is accomplished by the declaration
```
     EXTERNAL add
     INTEGER add
```
appearing somewhere in the calling program. Note that this was not necessary for "add" when it was declared as a SUBROUTINE, since its type was not important. The EXTERNAL statement indicates that "add" is really a FUNCTION, not an array which is accessed using the same syntax, and the INTEGER statement indicates its type. These two statements may occur in either order, but must be present if the implicit type of a function is not correct.

When the statement containing "add(1, 2)" is executed, "add" is recognized as an external function and the process for calling external routines is started. The arguments are evaluated and the routine is called, just like a SUBROUTINE. The only difference is that some place in the FUNCTION body, there must be an assignment to the FUNCTION name itself. This value then is used at the point of call, replacing the function call by its value. Thus writing "i = add(1, 2)" is equivalent (at runtime) to the statement "i = 3". Of course, the FUNCTION body can do much more than simply return a value as in this example, but it must do at least this much. FUNCTIONs cannot be called by any other mechanism than by using them in an expression. In particular, they may not be called using the CALL statement.

Arguments to a FUNCTION or SUBROUTINE may be:
```
     Arithmetic expression
     Variable name
     Array element reference
     Array name
```
In the first three cases, there must be a simple variable parameter in the called routine which corresponds to the argument. In the case of an array name, the corresponding parameter in the SUBROUTINE or FUNCTION must be an array, with the same number of dimensions. Other FORTRAN implementations have allowed, and in some cases encouraged, the number of dimensions to be different, but this implementation holds that they must be the same. The bounds on the array need not be the same, as the bounds that were given where the variable was originally declared are used. E.g.
```
     DIMENSION X(10, 50)
      ...
     CALL sub(X)
      ...
```
must have declarations similar to:
```
     SUBROUTINE sub(parm)
     DIMENSION parm(10, 50)
```
or
```
     DIMENSION parm(1, 1)
```
In this case, any reference to "parm" within "sub" really stands for a reference to "x" in the calling routine.

FORTRAN allows external functions to be called without being explicitly declared, by leaving out the EXTERNAL statment discussed above. Thus care must be taken when writing a FORTRAN program, as mispellings will probably end up being interpreted as external function references. E.g.

```
INTEGER array(10, 10)
i = arry(j, k)
```

In this example, the programmer has mistyped the name of the array, typing "arry" instead of "array". FORTRAN has no was of knowing that this is really a mistake and will assume that "arry" is really an external function of type REAL. This will only become evident when the program is linked and "_arry" comes up as an unsatisfied external reference.

A. COMPILER ERROR MESSAGES

This section describes the possible error messages which can arise from the compilation of a FORTRAN program. Error messages which come from execution of a program are discussed in the following chapter.

The Fortran compiler attempts to find and inform the user of all illegal or ill-defined constructs in the source program. Whenever an error is found, the source line which contains the error, along with a description of the error, will be printed. All error messages are given as text, not as some encripted number which must be looked up. For this reason, some error messages may be used for more than one error, whenever the errors are similar enough to warrant such overloading.

All errors detected by the compiler are considered fatal. If any errors are found during compilation, the assembly and linking loader phases are omitted. Some errors may come from the linking loader and the user is referred to the manual for that product for more information.

The following is a list of possible error messages and a description of their causes.

Number syntax
: The input text contains a number which has an illegal digit or which has a value larger than 32767 and does not contain a decimal point or an exponent.

String syntax
: A character string was encountered which contained more than 80 characters or which did not terminate properly. This will occur if the end of record occurs before the closing quote mark (') is found.

Unrecognizable statement
: The current statement could not be identified as a legal FORTRAN statement.

Too many continuation lines
: The current input record contains more than 10 continued lines.

Illegal label
: A value was found in the label field (cols 1-5) which was not numeric or had a value of 0.

Label on continuation line
: An input line was encountered which was non-blank in column 6 and which had something in the label field (cols 1-5).

Bad logical operator
: An input symbol was encountered which began with a period '.' character. This symbol was not a legal LOGICAL operator or constant, which is the only legal use of the period outside of a quoted string.

Expecting XXX
    This is not exactly a possible error message. In this case, the "XXX" is replaced by some item, which is spelled out. The meaning of this message is that the particular item could not be located on the current input line, but that one was expected. E.g. Expecting ")" normally indicates t hat a right parenthesis is missing on the current line.

Expression syntax
    This error will occur whenever a poorly formed expression is encountered. Refer to chapter VI for help.

XXX statement syntax
    Again, this is not an exact error message. The "XXX' will be replaced by the name of a particular type of statement, such as GOTO. This message indicates that the current statement is not syntactically correct, which may be caused by any number of reasons. E.g. "GOTO syntax error" indicates that the current line is a GOTO statement, but there is something wrong in the construction of it. Refer to chapters VIII-XIV for the details of statement syntaxes.

Part of line ignored
    This error will occur if there is any information on a line which does not belong there. Often, this is a result of some syntax error, but this is not always the case. E.g. the statement

```
GOTO x 1
```

    would cause this error since the "1" is not part of the GOTO statement. The error implies that this value was not examined when the GOTO statement was parsed, and thus the line is in error.

Missing iteration limit
    This error implies a syntax error in a DO statement. It specifically states that the second limit value of the iteration could not be located properly.

Bad character range
    A character range was specified in an IMPLICIT statement which didn't make sense. E.g. the statement

```
IMPLICIT INTEGER(Z-A)
```

    would give this error since the range "Z-A" is not ascending. Note that "A-Z" is quite legal.

Illegal constant expression
    A constant value was expected, but the expression given could not be evaluated to an [INTEGER] value. E.g. the declaration

```
INTEGER A(1+'a')
```

    would give this message.

Illegal symbol type
: A symbol was used on the current line which was not of an appropriate type for the given statement. There are many cases where this error applies, but the most common is where the wrong type variable (or expression) is used where a specific data type is required. E.g. the DO statement requires that the iteration limits be integer constants or variables. If a DO statement is written with one of these limits with type REAL this error will occur.

Multiply defined label
: This error will occur if the current line contains a label which has previously occurred in the current program segment.

Misplaced statement
: This message will occur if the current statement is a declaration statement and executable statements have already been processed. It will also occur if a SUBROUTINE or FUNCTION statement appears after the first statement of a program unit.

Statement not allowed here
: This error occurs if a RETURN statement appears in a main program.

Unmatched or Displaced ENDIF
: This error occurs whenever an ENDIF statement is encountered for which there is no corresponding IF-THEN statement.

Unmatched or misplaced ELSE or ELSEIF
: This error occurs whenever an ELSE or ELSE IF statement is encountered for which there is no preceding IF-THEN statement.

Duplicate declaration
: The current input line contains a variable declaration for a variable which has already been defined.

Too many labels in GOTO statement
: This error will occur as the result of a computed GOTO statement where the number of labels exceeds the maximum number allowable, currently 10.

Too many labels in program segment
: This error occurs if the current program segment contains too many statement labels. The current maximum number of labels in any given segment is 50.

Parameter error
: This error occurs whenever a SUBROUTINE or FUNCTION statement is being processed and the parameters are not variable natives.

Wrong number of subscripts
This error occurs whenever an array is used in an expression, but the number of indices does not match with the number of dimensions for the array. E.g. given the declaration:

```
     DIMENSION X(100, 100)
```

the expression

```
     X(I) = 1.0
```

would produce this error message.

Array has too many dimensions
This error indicates that the current array declaration contains an array declaration which has more than 3 dimensions.

Dummy array bound not allowed here
This error is generated by the use of a dummy array indicator (either '*' or a variable name) and the array is not a parameter. All arrays which are not parameters must have explicit, constant bounds.

Illegal DO destination label
The destination label in a DO statement has already been defined in some other context and is not allowed to be the termination of the current DO statement.

DO statements nested too deep
This error will occur if the current DO statement is included inside another DO statement, which Is inside yet another DO statement, ... If the DO statements are nested more than 5 levels deep, this error is produced.

DO statement nesting error
This error occurs if the range of one DO statement overlaps the range of another DO statement. E.g. the statements

```
          DO 100 I = 1, 10
               DO 110 J = 1, 10
     100 CONTINUE
     110 CONTINUE
```

would produce this error since the range of the second DO Statement overlaps the range of the first DO statement.

IF statements nested too deep
This error ressage will occur if IF-THEN statements are nested Inside one another for more than 5 levels.

IF-THEN-ELSE terminates outside DO loop
This error message is produced if a DO loop starts inside of an IF-THEN-ELSE construct, but terminates somewhere outside of this construct. The range of the DO loop must be completely contained within the IF-THEN-ELSE construct.

Illegal DO termination
This error message occurs if the terminal statement of a DO loop is not legal. DO loops must not terminate with any of the following statements:

```
     GOTO, IF, ELSE, ELSE IF, END, END IF, RETURN, STOP, DO
```

DO control variable reassigned
This message occurs if there is an assignment to the control variable for a DO statement which occurs within the range of the DO statement. It can also occur if nested DO statements attempt to use the same DO control variable.

Not an assignable variable
An assignment statement contains an expression on the left of the "=" which cannot be assigned to.

Illegal DO control variable
The control variable for a DO loop was not a simple INTEGER variable.

Implied DO loop syntax
An implied DO loop, which occurs within I/O statements, is not syntactically correct.

Variable not allowed in DATA statement
A variable which was declared in COMMON, or as a parameter or the name of a function, may not be initialized in a DATA statement.

Variable already initialized
A variable may appear in only one DATA statement.

Array elements out of order
When separate array elements are being initialized in a DATA statement, they must appear in "linear" order. This is the order in which they exist in memory, with the first subscript varying most rapidly. See the discussion of arrays in chapter V for more details.

Program too complicated - compiler abort error near line #nnnn
This error actually comes from the compiler system itself. The line number printed means nothing to the user's program and should in most cases be ignored. The error indicates that the user's program is too complex for the Fortran compiler to handle. The size of a program which can be handled depends on a number of factors, all of which are under the programmer's control.

- The number of unique identifiers in the program.
- The complexity of expressions within the program. Simpler expressions can be handled more easily.
- Where identifiers are declared. If the programmer follows good programming practice and declares all variables before they are used, including external procedures and functions, then the compiler will be able to handle much larger programs.

B. RUNTIME ERROR MESSAGES

The following is a list of the possible error messages which can occur during the execution of a Fortran program, along with a short explanation of what causes the error.

I/O error #nn while xxxx
: This error indicates that a system error occurred during some form of an input-output operation. The error code "nn" is a decimal value which can be found in the operating systems manual. The string "xxxx" stands for a piece of text which indicates what type of operation was going on at the time. E.g. "xxxx" may actually be "while reading on file #mm", which indicates that the I/O error occured while the Fortran runtime system was attempting to perform a "read" operation for unit number "mm".

Internal file Overflow/Underflow
: This error occurs If an attempt is made to read more characters than possible or write more characters than possible on an internal file. The amount of data which can be processed via an internal file operation, see chapter XI for details, is limited to the size of the character array used.

Unable to open file "zzz" on unit # nn
: This error occurs during an OPEN statement if there is some problem trying to open the specified file. The error message will be followed by an "I/O" error, discussed above.

Too many files open to open unit #nn.
: This error indicates that the program has already opened all the files permissible by the Fortran system. Currently this limit is 10 files, including the standard files on units 0, 5 and 6.

FORMAT too complicated
: A FORMAT string was being processed which contained more than 6 nesting levels of parentheses.

Unrecognizable FORMAT item
: During the processing of a FORMAT string, a FORMAT item was encountered which was not one of the characters ADELXH'$BP/.

Illegal Blank mode
: During the processing of a FORMAT string, a blank mode FORMAT item was encountered which was not either BZ or BN.

"." expected in F or E item
: During the processing of a FORMAT string, either an F or an E item was found which was not followed by w.d. Refer to chapter XII for details.

"ZZ" FORMAT item expected
    This error occurs during the processing of a FORMAT string and a particular FORMAT item, represented by "ZZ", was expected at that point in the processing. E.g. when reading an Integer value, the current FORMAT item must be "I" and if it is not, this error would occur.

Illegal field width
    During the processing of a FORMAT string, a FORMAT item of the form Zw or Zw.d was expected and the value of "w" or "d" did not make sense or was not present. E.g. in the "Iw" FORMAT item, the value of "w" must be positive and non-zero.

End of File on unit #nn
    An end of file was encountered while reading from the specified unit and the current READ statement did not include an END= specification.

Program aborted because of I/O error
    This message will follow many of the Input-Output related errors above if the current I/O statement does not include an ERR= processing option. See chapter XI for details.

Unit #nn not in correct mode for operation
    This error occurs if READ and WRITE operations are intermixed on a sequential file.

Illegal numeric Input
    During the processing of reading Integer, Logical or Real data, some illegal character or construct was encountered.

Array index error
    This error occurs if an array reference is made with an index for one of the dimensions which has a value outside of the permissible range for that array.

Integer Overflow/Underflow
    An Integer operation (multiplication or division) was attempted for which the result would not fit in 16 bits.

Division by 0
    Integer division by 0 was attempted.

Floating point Overflow/Underflow
    A floating point (Real) operation was attempted for which the result could not be represented. The range of Real values is roughly 1e+38 to 1e-38.

Floating to Integer conversion error
    An attempt was made to convert a Real value to an Integer value for which the value could not fit in 16 bits.

Floating point division by 0
    Real division by 0 was attempted.

Argument too large for exp(x)
    The argument to exp(x) was so large that the value would cause overflow. This error can also be the result of an exponentiation, x**y, since this operation is implemented in terms of "exp'.

Negative or zero argument to "zzz"
    The function "zzz" does not allow either negative or zero values to be used as arguments. E.g. "log" cannot be used with an argument less than or equal to zero.

User program ABORT
    This is not really an error, but is printed by the library routine "abort". The purpose of this routine is to abort the program and thereby produce any post-mortem information such as a trace or variable dump.

C. INTERFACING WITH EXTERNAL ROUTINES

Routines written in Fortran may be interfaced with routines which have been written in other languages, most notably assembly language and C. The following section describes how to interface assembly language routines with ones written in Fortran. See the documentation on C for details of interfacing with that language.

To illustrate how Fortran routines interface with other routines, you must first understand how information is passed between calling routines and those that are called. In Fortran, all values are passed between routines by reference. That is, the address of the value or variable is passed to the called routine rather than the actual value. Consider the Fortran statement:

```
     z = fun1(a, b, c)
```

This statement would be compiled into the following code:

```
     leas -8,s leave space for Real result
     ldx #c
     pshs x
     ldx #b
     pshs x
     ldx #a
     pshs x
     jsr _fun1
     leas 6,s
     <value for 'z' on top of stack>
```

From this example, a few rules should become obvious.

1. Parameters are pushed in reverse order.
2. All parameters are addresses, not values.
3. The calling routine is responsible for stack management.
4. All routine names have the character '_' prepended to them.
5. Functions leave their result on the stack, above the parameters. If a simple subroutine is being called which returns no explicit values, no space need be reserved.

By following the rules above, interfacing with external routines becomes simple.

Data representation:

The data manipulated by Fortran programs differs based on its type. In particular, Integers and Logicals are represented by 2 byte quantities, Reals occupy 8 bytes each and Character strings use exactly their length bytes (e.g. a Character*6 variable would use 6 bytes). Arrays have their own specific, internal representation, called a "dope vector". The form of a dope vector is important to anyone who wishes to access Fortran arrays from assembly language. The basic form of the dope vector for the array declared as:

```
     real x(0:10, 1:25)
```

would be:

```
     x fdb __a1
      fdb 2  # of dimensions
      fdb 8  # size of each element (real)
      fdb 0,10 lower and upper bound for 1st dimension
      fdb 1,25 lower and upper bound for 2nd dimension
     __al rmb (25-1+1)*(10-0+1)*8
```

If in-array is passed as a parameter, the address of the dope vector is what must be actually passed.

Of course there is an internal routine used to access array elements which would be of use to the assembly language programmer. Assume that we need to calculate the address of the element in the above array, "x(1, 6)". The following code would be used to perform this calculation:

```
      ldd #6
      pshs d
      ldd #1
      pshs d
      ldx #x  X has dope vector address
      jsr index
      fdb 2 number of dimensions in array
```

This routine calculates the appropriate address and returns it in the X register. The index values which are placed on the stack prior to the call are removed before the routine returns. If any of them are out of range, an error will be detected and the program aborted.

Program trace facilities:

Another feature of the Fortran system which may be of interest to the assembly language programmer is the "trace" feature. This is enabled whenever a routine is compiled with any of the "+dnt" options. This feature allows the runtime system to keep track of which routines are currently active and possibly the current line number within that routine. There are three entry points to this feature, described below:

```
     ** Enter a new routine
      jsr PAenter
      fcc /routine name/  exactly 8 characters!
      fdb n    current line #

     ** Exit the current routine
      jsr PAexit

     ** Update the most current line #
      jsr PAn1
      fdb n              new line #
```

If this feature is used and the program aborts, the system will be able to print a list of the active routines in the reverse order of how they were called, along with the line number in that routine. This can be very useful to the assembly language programmer.

D. STANDARD FUNCTIONS

Several standard functions are available to the FORTRAN programmer. These include the basic mathematical functions as well as some type conversion functions. In the following table which summarizes these functions, "a1" is the first argument to the function and "a2" is the second.

FORTRAN Intrinsic Functions

Name	Type of Arguments	Type of Result	Definition
ifix	Real	Integer	Convert Real to Integer
int	Real	Integer	Same as ifix
float	Integer	Real	Convert Integer to Real
ichar	Character	Integer	Convert Character to Integer
aint	Real	Real	Truncate a Real
anint	Real	Real	Nearest whole number int(a+.5) if a > 0.0 int(a-.5) if a < 0.0
nint	Real	Integer	Nearest Integer
iabs	Integer	Integer	Absolute value
abs	Real	Real	
mod	Integer	Integer	Remainder a1-int(a1/a2)*a2
amod	Real	Real	
isign	Integer	Integer	Sign transfer abs(a1) if a2 > 0.0 -abs(a1) if a2 < 0.0
sign	Real	Real	
idim	Integer	Integer	Positive difference a1-a2 if a1 >= a2 0 if a1 < a2
dim	Real	Real	

FORTRAN Intrinsic Functions

Name	Type of Arguments	Type of Result	Definition
max0 max1 amax0 amax1	Integer Real Integer Real	Integer Integer Real Real	Largest value a1 if a1 >= a2 a2 otherwise
min0 min1 amin0 amin1	Integer Real Integer Real	Integer Integer Real Real	Minimum value a1 if a1 <= a2 a2 otherwise
sqrt	Real	Real	Square root
exp alog alog10	Real Real Real	Real Real Real	e**x Natural log(x) Common log(x)
sin cos tan asin acos atan atan2 sinh cosh tanh	Real Real Real Real Real Real Real Real Real Real	Real Real Real Real Real Real Real Real Real Real	Sine Cosine Tangent ArcSine ArcCosine ArcTangent Arctangent(a1/a2) Hyperbolic Sine Hyperbolic Cosine Hyperbolic Tangent

Note:
The standard functions listed above meet the FORTRAN language subset standard except for the "min" and "max" functions. These are only defined to work with pairs of values, not an arbitrary list of values as defined in the standard.

E. RESERVED WORDS

The following words and symbols are reserved in all contexts in a FORTRAN program and must not be used as identifiers.

.AND.	ERR
.EQ.	EXTERNAL
.FALSE.	FILE
.GE.	FORMAT
.GT.	FUNCTION
.LE.	GO
.LT.	GOTO
.NE.	IF
.NOT.	IMPLICIT
.OR.	INTEGER
.TRUE.	INTRINSIC
ACCESS	LOGICAL
ASSIGN	OPEN
BACKSPACE	PAUSE
CALL	PRINT
CHARACTER	PROGRAM
COMMON	READ
CONTINUE	REAL
DATA	RECL
DIMENSION	RETURN
DO	REWIND
ELSE	SAVE
ELSEIF	STOP
END	SUBROUTINE
ENDFILE	THEN
ENDIF	TO
EQUIVALENCE	WRITE