Supersoft / Small Systems Services

FORTRAN IV Extended

User's Manual

(Revision 3.0)

RUN TIME LICENSE FEE

The fees for distributing the FORTRAN run time packages are as follows:

First 100 copies per year:

No charge.

101 copies or more per year:

Option 1: Per copy charge is 2% of the selling price of your product or $5.00, whichever is greater. If your package is bundled with hardware and no separate price for the software can be determined, the charge is $5.00 per copy (over 100 copies).

Option 2: $2000.00 annual license fee

Option 3: $7500.00 perpetual license fee

For additional information about these run time license fees, or to obtain agreements for FORTRAN run-time package distribution, please contact SuperSoft.

Table of Contents

1 INTRODUCTION

SSS FORTRAN was designed and implemented to make available for microcomputer systems full FORTRAN IV language and system capabilities. The language meets and exceeds the full ANSI X3.9-1966 Standard FORTRAN Language features, and is similar to FORTRAN languages available on large computers.

The system package includes:

- FORTRAN Compiler: Supports advanced features such as complex arithmetic and character variables and functions; compiles at average rates of 600 lines per minute.

- Linking Loader: Supports full linking of FORTRAN and Assembly Language modules; supports system and user libraries of mixed FORTRAN and Assembler modules.

- Disk based I/0 Package: Supports sequential and random access, binary and ASCII, block and stream mode files.

- Runtime Package: FORTRAN formatter; math library with double precision and complex routines; built-in function library; dynamic allocation and assembly language linkage library.

The operating environment necessary for using the FORTRAN system on a Z80 based microcomputer system is summarized below.

1. Secondary storage system capable of supporting the CP/M operating system.

2. Minimum of 32K bytes of RAM memory (large programs may need more space to compile), which includes the space necessary for the operation of CP/M.

2 SSS FORTRAN LANGUAGE

2.1 INTRODUCTION

This chapter is not meant to be a complete tutorial on the FORTRAN language, rather a guide to the available language features for the reader with a working knowledge of FORTRAN. There are many variants of FORTRAN; each large manufacturer of computer systems has a variant available or possibly several variations. Although some references are available for the core language (ANSI FORTRAN), many references deal with a specific variant which is most likely an extension of the core language. SSS FORTRAN implements the full ANSI X3.9-1966 FORTRAN Standard Language, including COMPLEX, and SSS FORTRAN exceeds the ANSI Standard in a number of important areas.

This chapter refers to an implementation of the FORTRAN IV language that is very similar to that of SSS FORTRAN, namely the WATFIV language. It is suggested that the reader refer to the following reference as a tutorial for SSS FORTRAN, or for a detailed description of the use of standard FORTRAN features.

FORTRAN IV with WATFOR and WATFIV,
Cress, Dirksen and Graham
Prentice-Hall Series in Automatic Computation
1970.

2.2 WATFIV COMPARISONS

2.2.1 Data Types

FORTRAN compilers which operate on a computer with a 32-bit (4-byte) word size normally support data constructs which occupy either full words (4-bytes) or double words (8-bytes), with certain constructs occupying half words (2-bytes) or quarter words (1-byte). Since most microcomputers are 8-bit (1-byte) machines, a FORTRAN compiler for such a machine would normally support data constructs which are somewhat different from those supported by a 32-bit FORTRAN compiler. This is the case with WATFIV (a 32-bit FORTRAN compiler and SSS FORTRAN (an 8-bit FORTRAN compiler). The data constructs supported by each are summarized as follows.

DATA CONSTRUCT	WATFIV	SSS FORTRAN
Short Integer	2 bytes INTEGER Range :-32768..32767	1 byte INTEGER*1 Range:-128..127
Full Integer	4 bytes INTEGER*4 or INTEGER Range:-2147483648 – 2147483647	2 bytes INTEGER*2 or INTEGER Range:-32768..32767
Real	4 bytes REAL*4 or REAL Range: approximately 10**-78 to 10**76 Precision: approximately 6.2 decimal digits	
Double Precision	8 bytes REAL*8 or DOUBLE PRECISION Range: approximately 10**78 to 10**76 Precision: approximately 16.8 decimal digits	
Complex	8 bytes COMPLEX Range and Precision: same as a pair of REALs	

Extended Complex	16 bytes COMPLEX*16 Range and Precision: same as a pair of REAL*8 s	not supported
Logical	2 or 4 bytes LOGICAL*1 or LOGICAL Value: .TRUE. or .FALSE	1 byte LOGICAL Value: .TRUE. or .FALSE.
Character	1 to 255 bytes Fixed length CHARACTER*n (0<n<256)	1 to 32767 bytes Adjustable length up to a fixed maximum CHARACTER*n (0<n<32768)

Note that in SSS FORTRAN, although a type of LOGICAL*2 is not available, INTEGER (or INTEGER*2) may be used as LOGICALs with the same results. Also, in SSS FORTRAN .FALSE. is a zero value, and .TRUE. is any non-zero value.

2.2.2 SSS FORTRAN Restrictions

WATFIV has included a number of extensions to standard FORTRAN, some of which have not been included in SSS FORTRAN. These are summarized as follows.

ENTRY, D@@LIST, NAMELIST, PRINT, PUNCH Statements

Permitted by WATFIV; not permitted by SSS FORTRAN.

Arrays

WATFIV permits up to seven subscripts. SSS FORTRAN permits up to three subscripts.

FORMAT Statement

WATFIV permits the T type specification; SSS FORTRAN does not. WATFIV permits G-format to be used with integers; SSS FORTRAN does not.

PAUSE Statement

WATFIV permits PAUSE message; SSS FORTRAN does not. (But each permits PAUSE as well as PAUSE integer.)

FORMAT-Free I/0 Statement

WATFIV permits READ and WRITE statements to the console or list device with default formatting assumed; SSS FORTRAN does not permit these forms. But SSS FORTRAN includes features for free format input, designed to allow free field input of user data to FORTRAN programs, as well as free format output of strings (See Appendix G.).

RETURN Statement

WATFIV permits variable RETURN statements of the form RETURN integer (with associated dummy parameters and n arguments); SSS FORTRAN does not.

DATA Initialization Statement

WATFIV permits initialization of data within type declarations; SSS FORTRAN permits data initialization only within DATA statements.

2.2.3 WATFIV/SSS FORTRAN Differences

There are some features of FORTRAN which are implemented differently in SSS FORTRAN than they are in WATFIV. These are summarized as follows.

Hexadecimal Constants

WATFIV requires that hexadecimal constants be preceded by the letter Z; SSS FORTRAN requires that they be enclosed in exclamation marks (e.g. !6520!). Note that in SSS FORTRAN, all hexadecimal constants are typed as numeric constants depending on the number of hex digits specified.

NUMBER OF HEX DIGITS	TYPE GIVEN TO CONSTANT
1 or 2	INTEGER*1
3 or 4	INTEGER*2
5 to 8	REAL*4
9 or more	REAL*8

If a hexadecimal constant is shorter than the data type it will be given, then padding with zeroes is done on the right. If more than 16 hex digits are specified, truncation on the right occurs.

FORMAT Statement

In WATFIV, using the Zn (hex) specification on output, if the data item has more than n digits, then the rightmost digits are output; in SSS FORTRAN, the leftmost digits are output.

IMPLICIT Statement

WATFIV permits only one IMPLICIT statement per (sub)program. SSS FORTRAN permits more than one, but requires that within any one IMPLICIT statement, only one implicit type be declared. For example:

```
IMPLICIT REAL (A-V), INTEGER (X,Y)
```

in WATFIV, must, in SSS FORTRAN, be written:

```
IMPLICIT REAL (A-V)
IMPLICIT INTEGER (X,Y)
```

READ Statement

WATFIV uses END=Label and ERR=Label for transferring control on end-of-file or I/0 errors. SSS FORTRAN uses ENDFILE=Label and ERREXIT=Label to do the same thing.

Length Specifications

WATFIV permits length specifications on individual items in a declaration; SSS FORTRAN permits length specification on only the explicit type specification. For example:

```
CHARACTER*2 A(30), B*4, C*5(3)
```

in WATFIV, must, in SSS FORTRAN be written:

```
CHARACTER*2 A(30)
CHARACTER*4 B
CHARACTER*5 C(3)
```

Direct-Access I/0

WATFIV employs the DEFINE FILE statement as well as the FIND statement. SSS FORTRAN uses system subroutine calls to implement these features.

Call-by-Reference

WATFIV uses call-by-value for parameter passing when possible, and call-by-reference may be specified by enclosing a dummy parameter in slashes. SSS FORTRAN uses the more standard system of call-by-reference as default (address of items are passed), and call-by-value may be specified by enclosing an argument in parentheses. For example:

```
CALL SUB (A, (2.0) )
```

In SSS FORTRAN, A is passed by reference (its value may be modified by the subroutine), while 2.0 is passed by value (a copy is made and passed to the subroutine). In this case the constant 2.0 cannot be modified by the called program. If the parentheses are removed, and call-by-location is used, it would be possible to modify the constant in the called routine.

Core-to-Core I/0

This feature allows an SSS FORTRAN CHARACTER item to be used in lieu of a unit specification in a formatted READ or WRITE statement. Thus, data can be read from or written to a CHARACTER item, instead of an I/0 unit, giving a memory-to-memory or core-to-core transfer. For example:

```
      CHARACTER*15 A
      DATA A/ 1  2  3 /
      READ(A,100) I,J,K
100   FORMAT (3I3)
```

I, J, and K are read from the CHARACTER string, A, with the FORMAT given. Core-to-core I/0 in SSS FORTRAN is compatible with the WATFIV core-to-core feature with the following exceptions:

1. SSS FORTRAN allows the core-to-core feature in formatted I/0 only;

2. SSS FORTRAN allows only CHARACTER variables or constants as unit specifications (CHARACTER array names or array elements are not allowed).

In SSS FORTRAN, any error messages referring to core-to-core I/0 will indicate a unit number of 64.

2.2.4 SSS FORTRAN Extensions

In addition, SSS FORTRAN implements some extensions which are not permitted in WATFIV. These are summarized as follows.

Literal Constants

A string of characters may be delimited in any one of three ways:

1. The string can be enclosed in apostrophes (single quotes,).

2. The string can be enclosed in double quotes ().

3. The string can be preceded by wH where w is the number of characters in the string (Hollerith specification).

Type 1 and Type 3 literal constants are ANSI style, as normally seen in FORTRAN. That is, they are strings with a fixed length, with no extra information stored about the string. Type 2 string constants are compatible with the CHARACTER variable type in SSS FORTRAN. Current length, maximum length, and pointer information is stored separately from the string. Note that any of the above style literal constants can be used in DATA or FORMAT statements.

As a further extension, SSS FORTRAN allows 1-byte hexadecimal specifications within Type 1 or Type 2 strings. Hexadecimal 1-byte values in strings are delimited by left and right brackets ([]). If a left bracket is to be included in a Type l or a Type 2 string, then two consecutive left brackets must be used. For example:

```
INTEGER*1 A(10)
DATA A / [0D]MESSAGE[0D][[ /
```

The array A is initialized with 10 bytes of data

```
cr M E S S A G E cr
```

where cr represents an ASCII carriage return.

String Comparisons

Both CHARACTER variables and Type 2 (double quoted) strings may be compared using any of the arithmetic comparison operators. Comparison is ASCII DICTIONARY ORDER, with .LT. A. Two strings are .EQ. if they have the same length and their characters match. For example,

A < AB < B

Short INTEGER Constants

An INTEGER constant in the range -128..127 may be written with a trailing sharp sign (as, for example, 15#). Such a constant occupies only one byte, instead of the normal two bytes.

Computed GOTO Statement

The index of a computed GOTO statement may be an arithmetic expression.

DO Statement

The initial value, test value, and increment of a DO statement may be arbitrary arithmetic expressions. Moreover, the DO index may be either a variable name or an array element. The initial, test, increment, and index items must be of the same Data Type. SSS FORTRAN also allows DO loops with negative increment values. The DO index may be REAL. For example:

```
DO 10 A(I) = 50.0, 20.EO, -1.0
```

Assuming that A is an array of Type REAL, this DO statement is legal in SSS FORTRAN.

Output Lists

Arbitrary expressions are permitted in output lists. However, parenthesized SIMPLE LISTS, such as

```
WRITE (1) (B,C)
```

are not permitted, since they are sometimes indistinguishable from complex constants and parenthesized expressions. Also, if any output-list expression involves a function call, that function must not attempt any input/output.

DO-Implied Specification

The initial value, test value, and increment of a DO-implied specification may be arbitrary arithmetic expressions. But in this case, unlike the DO statement, although the index may be non-INTEGER, it may not be an array element.

Multiple Statements

More than one statement may be entered on a line by separating with a semicolon (;). If the succeeding statement is to have a statement number, then it is simply keyed at the beginning of that statement. There is effectively no column 6 continuation ON SUCH SUCCEEDING statements. One exception is that there can be no additional statements following a FORMAT statement on a given line. The following is a legal input line:

```
X=Y;123 CONTINUE;4567 PAUSE; GOT0 33;22 STOP
```

Subscripts

SSS FORTRAN permits arbitrary arithmetic expressions to be used as subscripts.

Continuation Lines

There is no limit to the number of continuation lines permitted in SSS FORTRAN, as long as assignment statements are recognizable as such within three lines.

INCREMENT and DECREMENT Statements

If I is a variable name, SSS FORTRAN permits

```
I = I + l
```

to be written as

```
INCREMENT I
```

and

```
I = I - l
```

as

```
DECREMENT I
```

thus permitting the compiler to generate much more efficient code. Note that the variable used in an INCREMENT or DECREMENT statement must be a simple INTEGER (or INTEGER*2) variable, and that it may not be a subprogram parameter.

LOGICAL Expressions

The logical operation of Exclusive Or (.XOR.) is permitted in SSS FORTRAN, and has the same precedence level as Inclusive Or (.OR.). Exclusive Or has the following values:

X	Y	X .XOR. Y
True	True	False
True	False	True
False	True	True
False	False	False

Masking Expressions

One or two byte quantities (INTEGER*1s, INTEGER*2s, one to four digit hexadecimal constants, LOGICAL values) may, in SSS FORTRAN, be combined in masking expressions.

.AND.	bitwise And
.OR.	bitwise Inclusive Or
.NOT.	bitwise Complement
.XOR.	bitwise Exclusive Or

The precedence of the operators is the same as with logical expressions. For example, the following masking assignment statement exchanges the rightmost two bits of an integer, I:

```
I = (I.AND..1)*2 .OR. (I.AND.2#)@ .OR. I.AND.!FFFC!
```

FORMAT Statement

A formatted WRITE normally terminates with a carriage return/line feed (CR/LF) sequence sent to the output device. Replacing the rightmost right parenthesis of the FORMAT statement by a dollar sign ($), will suppress the transmission of the final CR/LF. This is particularly useful for interactive programs. For example, the normal sequence:

```
      WRITE (TTY,lOO)
100   FORMAT (  TYPE l OR 2, PLEASE:  )
      READ (TTY,200) ANSR
200   FORMAT (Il)
```

produces the following output at the terminal:

```
TYPE l OR 2, PLEASE: [CR/LF]
[user responds here]
```

whereas changing FORMAT 100 to

```
100   FORMAT (  TYPE 1 OR 2, PLEASE:  $
```

produces

```
TYPE l OR 2, PLEASE:  [user responds here]
```

COMPLEX Constants

In WATFIV, COMPLEX constants must be specified as two REAL values. In SSS FORTRAN, although a COMPLEX constant is always stored as two REAL values, the constant may be specified using either REAL, DOUBLE PRECISION, or INTEGER numbers. For example, the following COMPLEX constants are all equivalent in SSS FORTRAN.

```
(1,2)  (1.0,2.0)  (lD0,2DO)  (1,2DO)
```

LOGICAL IF Statement

Since INTEGER*1 and INTEGER*2 values may, in SSS FORTRAN, be used to hold LOGICAL values, either of these Data Types may be used in a LOGICAL IF statement. Moreover, if the type of the branch expression is not INTEGER or INTEGER*1, it will be converted if possible. Thus, in SSS FORTRAN, the following LOGICAL IFs are all valid.

```
INTEGER*2 I
REAL R
IF (I) GOTO 10
IF (I+3 .AND. 7) GOTO 10
IF (R+1.0) GOTO 10
```

2.3 SUMMARY OF STATEMENT TYPES

2.3.1 Executable Statements

Arithmetic Assignment Statement

```
V = E
```

where V is a variable name or an element of any Type except CHARACTER. E is an arithmetic expression or an ANSI style literal string (a string enclosed in single quotes, or a Hollerith string).

Note: In case V and E are of different types, a conversion is performed. Standard FORTRAN rules are used for the conversion process. That is, in the case of an assignment operator, E is made to be of the Type of V before the assignment is done. In arithmetic expressions for operators other than assignment, if two items are of different Types, standard UPWARD conversion is done. This upward conversion ordering is:

```
INTEGER*1 < INTEGER*2 < REAL*4 < REAL*8 < COMPLEX
```

Note that ANSI style literals (single quote strings and Hollerith strings) can be used wherever general expressions are allowed. In these cases, COERCION is done, with the string taking the Type of the .(item directly to its left, and with truncation or padding with blanks if necessary. For example,

```
REAL A
A = 123
```

Since A is REAL (@ bytes), the string is made to be 4 bytes (123) before the assignment is performed.

```
IF (I .EQ.  1234  ) GOTO l
```

Since I is INTEGER (2 bytes), the string is converted to 2 bytes (12) before the comparison is performed.

ANSI style literals are passed intact when used as arguments to subprograms (i.e. no coercion is done). For example:

```
CALL XYZ ( 123456789 , 5HABCDE)
```

the strings are not changed in any way. The addresses of the items 123456789 and ABCDE are passed to the subroutine.

CHARACTER Assignment Statement

```
V = E
```

where V is a variable name or an array element of Type CHARACTER*n. E is of type CHARACTER*n and is either a constant (.a double quote string), or a variable name or an array element or a function invocation.

Note: No conversions need be done in this statement type, since all items are of the same type.

Masking Assignment Statement

```
V = E
```

where V is a variable name or an array element of any type except CHARACTER. E is a masking expression.

Note: Conversion is from INTEGER to the type of V as in an arithmetic assignment statement.

INCREMENT Statement

```
INCREMENT V
```

where V is an INTEGER variable name which is not a subprogram parameter.

DECREMENT Statement

```
DECREMENT V
```

where V is the same as for INCREMENT.

GO TO Statement

```
GO TO Label
```

where Label is the statement label of an executable statement.

Computed GO TO Statement

```
GO TO (List), E
```

where List is a comma-separated list of statement labels of executable statements. E is an arithmetic expression.

Note: E is evaluated and converted to type INTEGER*2. If the resulting value is less than zero or greater than the number of statements in the list, then execution continues with the statement following the Computed GO TO.

ASSIGN Statement

```
ASSIGN Label TO V
```

where Label is the statement label of an executable statement. V is a variable name or array element of type INTEGER or INTEGER*2.

Assigned GO TO Statement

```
GO TO V, (List)
```

where V is a variable name or array element of Type INTEGER. List is a comma-separated list of statement labels of executable statements, one of which has been previously ASSIGNed to V.

Note: List must consist of statement labels defined elsewhere in the program, but no check is made to determine that the value of V matches any of the labels in List.

Arithmetic IF Statement

```
IF (E) Label-1, Label-2, Label-3
```

where E is an arithmetic expression. Label-1, Label-2, and Label-3 are statement labels of executable statements.

Logical IF Statement

```
IF (E) S
```

where E is any arithmetic expression. S is any unlabeled executable statement except DO and Logical IF. The result of the expression evaluation will be converted to an integer value, and if the result is 0 (LOGICAL .FALSE.), S will not be executed. If the result is non-zero (LOGICAL .TRUE.), S will be executed.

DO Statement

```
DO Label V = El, E2, E3
```

or

```
D0 Label V = El, E2
```

where Label is the statement label of the executable statement which is the TARGET or END OF RANGE STATEMENT OF THE DO. V IS A VARIABLE OR array element, called the control variable. EL, E2, AND E3 ARE arithmetic expressions; E1 is called the INITIAL EXPRESSION, E2 THE TEST expression, and E3 the INCREMENT expression.

Note: El, E2, and E3 may be of any arithmetic type, while V may be of any arithmetic type except COMPLEX. If any of E1, E2, or E3 do not match V in type, then they are converted to match. For example:

```
REAL R
INTEGER I
DO 10 R = l, 4DO, I
```

is equivalent to

```
DO 10 R = 1.0, 4.0, FLOAT(I)
```

CONTINUE Statement

```
CONTINUE
```

PAUSE Statement

```
PAUSE
```

or

```
PAUSE integer
```

STOP Statement

```
STOP
```

or

```
STOP integer
```

RETURN Statement

```
RETURN
```

READ Statement

```
READ (U, F, ENDFILE@...abel-1, ERREXIT=Label-2) List
```

where U is an integer or variable name (simple INTEGER variable or simple CHARACTER variable), which specifies the unit number of the input device or a CHARACTER variable name for core-to-core input (see Appendix F). F is either a statement label of the FORMAT statement which describes the data on a formatted input device, or is an array name or CHARACTER variable which contains the FORMAT specification which describes the data on a formatted input device. Label-1 and Label-2 are statement labels of executable statements. List is an input/output list.

Note: F and ENDFILE=Label-1 and ERREXIT@...abel-2 and List are all optional.

WRITE Statement

```
WRITE @, F, ENDFILE@...abel-1, ERREXIT@..abel-2) List
```

where U is an integer or variable name (simple INTEGER variable or simple CHARACTER variable), which specifies the unit number of the output device or a CHARACTER variable name for core-to-core output (see Appendix F). F is either a statement label of the FORMAT statement which describes the data on a formatted output device, or is an array name or CHARACTER variable which contains the FORMAT specification which describes the data on a formatted output device. Label-1 and Label-2 are statement labels of executable statements. List is an input/output list.

Note: F and ENDFILE@..abel-1 and ERREXIT=Label-2 and List are all optional.

Direct-Access READ Statement

```
READ @ /R, F, ENDFILE@..abel-1, ERREXIT@...abel-2) List
```

where U, F, Label-1, Label-2, and List are as in the READ statement. R is an expression which specifies which record is to be read.

Direct-Access WRITE Statement

```
WRITE (U/R, F, ENDFILE...abel-1, ERREXIT=Label-2) List
```

where U, F, Label-1, Label-2, and List are as in the WRITE statement. R is an expression which specifies which record is to be written.

REWIND Statement

```
REWIND U
```

where U is an integer or a variable name which specifies the unit number of the device.

BACKSPACE Statement

```
BACKSPACE U
```

where U is an integer or a variable name which specifies the unit number of the device.

ENDFILE Statement

```
ENDFILE U
```

where U is an integer or a variable name which specifies the unit number of the device.

CALL Statement

```
CALL N
```

or

```
CALL N list
```

where N is the symbolic name of the subroutine being called. List is a comma-separated list of expressions, array names, subroutine names, and function names which are the ARGUMENTS or ACTUAL PARAMETERS for the subroutine being called.

2.3.2 Non-Executable Statements

Explicit Type Declaration Statement

type List

where *type* is one of the following ALLOWABLE TYPES:

```
INTEGER*1
INTEGER*2
INTEGER
REAL*4
REAL
REAL*8
DOUBLE PRECISION
COMPLEX
LOGICAL
CHARACTER
CHARACTER*n
CHARACTER*32767
```

List is a comma-separated list of variable names, function names, array names, and array declarations.

Note: An array which has its dimensions specified by a DIMENSION or COMMON statement may be explicitly Typed, but may not be re-dimensioned in a Type Declaration, i.e., it may appear as an array name, but not as an array declaration.

IMPLICIT Type Declaration Statement

```
IMPLICIT type (List)
```

where Type is one of the allowable types. List is a comma-separated list of single alphabetic letters, and triples of letter-minus-sign-letter (e.g. P-W) representing the first and last characters of a range.

Note: For IMPLICIT CHARACTER*n declarations, the maximum value for n is 255. This does not restrict the maximum length of CHARACTER strings in general, but does restrict the maximum length of a CHARACTER variable that is typed via an IMPLICIT declaration.

DIMENSION Statement

```
DIMENSION List
```

where List is a comma-separated list of array declarations.

COMMON Statement

```
COMMON List-1 /Name-2/List-2 ... /Name-n/List-n
```

or

```
COMMON /Name-1/List-1 /Name-2/List-2 ... /Name-n/List-n
```

where List-1, List-2, etc. are comma-separated lists of variable names, array names, and array declarations. Name-1, Name-2, etc. are either names of LABELED COMMON BLOCKS or are blank (designating the BLANK COMMON block).

Note 1: There may be one, two, or many COMMON block lists per COMMON statement.

Note 2: An array which has its dimensions specified by a DIMENSION or COMMON statement may be explicitly typed, but may not be re-dimensioned in a type declaration, i.e., it may appear in a type declaration statement as an array name, but not as an array declaration.

Note 3: An item of type CHARACTER*n is stored in COMMON without length information, i.e. simply as an array of characters (see Section II.D.8).

EQUIVALENCE Statement

```
EQUIVALENCE (List-1), (List-2), ... (List-n)
```

where List-1, List-2, etc. (called EQUIVALENCE LISTS) are comma-separated lists of two or more variable names, array names, or array elements.

Note: There may be one, two, or many equivalence lists per EQUIVALENCE statement.

EXTERNAL Statement

```
EXTERNAL List
```

where List is a comma-separated list of subprogram names.

DATA Statement

```
DATA Vlist-1/Dlist-1/, Vlist-2/Dlist-2/, ... Vlist-n/Dlist-n/
```

where Vlist-1, Vlist-2, etc. (called VARIABLE LISTS) are comma-separated lists of variable names, array names, or array elements to be initialized. Dlist-1, Dlist-2, etc. (called INITIALIZATION LISTS) are comma-separated lists or initialization constants; each constant may be preceded by a REPETITION FACTOR of the form R* where R is an integer REPETITION COUNT.

Note: SSS FORTRAN has two modes of initialization in DATA statements. Mode l is the standard ANSI mode. Mode 2 is an extension that applies to array initialization only. Each of these modes is explained in the following.

Mode 1: This is the normal ANSI mode of data initialization. In this mode, only simple variables or array elements may be initialized, i.e. array names without subscripts may not be used. In this mode, the following rules apply.

a. One constant must appear for each variable to be initialized.

b. If the constant and corresponding variable are of the same length, no changes to the constant occur.

c. If the lengths differ, truncation or padding (whichever is needed) is done on the right. Strings are padded with blanks and other constants (with one exception) are padded with zeroes.

d. The exception is the following. If a 2-byte variable is initialized with a 1-byte integer constant, then the constant is not padded with a zero, but rather the equivalent 2-byte representation of the constant is used.

e. All three string types (single quote, double quote, Hollerith) are equivalent in Initialization Lists.

f. For CHARACTER*n variables in the Variable List, the current length is set to the length of the corresponding data item in the Initialization List prior to any truncation or padding (see example below).

Examples for Mode 1 initialization.

```
REAL A,B,C
DATA A,B,C / 1.0, 2.0, 3.0 /
```

No truncation or ps@ding is necessary in this case since all items are REAL (4 bytes).

```
REAL A,B,C
DATA A,B,C / !01020304051,  ABC , 1# /
```

Since A, B, and C are all REAL (4 bytes), all of the constants in the Initialization List must be made 4 bytes long; the hex string is truncated to !01020304!, the string is padded to ABC[0], and the INTEGER*1 constant is padded to the right with zeroes so that its hex representation is !01000000!.@

```
CHARACTER*8 A,B,C
DATA A,B,C / 12345,  12345 , 5H12345 /
```

A, B, and C are all initialized to the equivalent of 8H12345 Note that since A, B, and C are CHARACTER*n variables, the current length of each is set by the DATA statement. Since the length of each constant in the initialization list is 5 before padding, the current length of each of A, B, and C is set to 5.

```
INTEGER A
LOGICAL B
DATA A,B / .TRUE., .FALSE. /
```

Since B is LOGICAL, as is its corresponding initialization constant, no changes are necessary. A is INTEGER (2 bytes) while its corresponding initialization constant is a 1-byte LOGICAL. In this case, Rule d above applies, and since .TRUE. is represented as all l s, A is initialized to hex !FFFF!.

Mode 2: This is an extended mode of initialization that applies to array initialization only. In this mode, array names without a subscript may be used in the Variable List. This mode thus allows entire arrays to be initialized without specifying each element separately in the Variable list. The following rules apply to Mode 2 DATA statements.

a. Sufficiently many constants must be specified in the Initialization List to fill the entire array.

b. Constants are padded on the right to a multiple of the element length of the array being initialized. With one exception, no truncation occurs.

c. The exception is that if the array involved is of Type INTEGER*1 or LOGICAL, any INTEGER*2 constants are truncated on the left to 1 byte. If the array is of Type INTEGER*2 or INTEGER, any INTEGER*1 or LOGICAL constants are converted to its 2-byte equivalent.

d. As with Mode l, all string styles are equivalent.

e. For arrays of Type CHARACTER*n, one constant must be specified in the Initialization List for each element of the array being initialized. The current length of each element of the array is set to the length of its corresponding data item prior to any padding.

Examples of Mode 2 Initialization.

```
REAL A(3)
DATA A / 1.0, 2.0, 3.0 /
```

No conversion is required. A is 12 bytes, and 12 bytes worth of constants are specified.

```
INTEGER A(9)
DATA A /  12345 , 678, 8HABCDEFGH /
```

Since the array A is of Type INTEGER (2 bytes per element), each data item is padded to a multiple of 2 bytes: Thus, A is 18 bytes, and is initialized to the equivalent of 18H12345 678 ABCDEFGH.

```
INTEGER A(9)
DATA A / 6*0, 2*2, -1# /
```

The only conversion is that the 1-byte constant is converted to its 2-byte equivalent, viz. hex !FFFF!.

Note 1: In addition Mode l and Mode 2 may occur together in a single DATA statement. For example,

```
INTEGER A(2),B
DATA A,B / l, 2, 3 /
```

Note 2: In accordance with ANSI requirements, multiply dimensioned arrays are filled in column major order, i.e. with the first subscript varying most rapidly, and the last subscript varying least rapidly. For example,

```
INTEGER A(2,3)
DATA A / l, 2, 3, 4, 5, 6 /
```

has the initialization effect of

```
A(1,1)=1; A(2,1)=2; A(1,2)=3; A(2,2)=4; A(1,3)=5; A(2,3)=6
```

The above rules may at first seem overly complex. But since SSS FORTRAN has been extended to include the CHARACTER*n Data Type, 1-byte constants, and Mode 2 array initialization, Standard FORTRAN rules had to be extended in a reasonable way, so that logical results would occur. The point to remember is that standard results occur when standard features are used. The advanced features may take some getting used to but are handy features to have.

END Statement

```
END
```

BLOCK DATA Statement

```
BLOCK DATA
```

SUBROUTINE Statement

```
SUBROUTINE N
```

or

```
SUBROUTINE N list)
```

where N is the symbolic name of the subroutine. List is a comma-separated list of symbolic names which are the DUMMY parameters for the subroutine.

FUNCTION Statement

```
UNCTION N (List)
```

or

```
Type Function N (List)
```

where N is the symbolic name of the function. List is a comma-separated list of symbol names which are the DUMMY parameters for the function. Type is one of the allowable types.

Statement Function Statement

```
N (List) = E
```

where N is the symbolic name of the function. List is a comma-separated list of symbol names which are the DUMMY parameters for the function. E is an expression.

FORMAT Statement

```
Label FORMAT (List)
```

or

```
Label FORMAT (List)
```

where Label is a statement label for the FORMAT statement. List is a comma-separated list of FSPECs, and FSPEC is either an Edit Descriptor:

```
rIw         (INTEGER FORMAT Code)
rFw.d       (REAL FORMAT Code)
rDw.d       (DOUBLE PRECISION FORMAT Code)
rEw.d       (REAL FORMAT Code)
rLw         (LOGICAL FORMAT Code)
rAw         (CHARACTER FORMAT Code)
rZw         (Hexadecimal FORMAT Code)
rGw.d       (REAL FORMAT Code)
wHXX...X    (Hollerith Literal Character FORMAT Code)
wX          (Spacing FORMAT Code)
```

```
              (End-of-record FORMAT Code)
 nP           (Scale Factor)
```

or is a group FORMAT Specification:

```
r (List)
```

where r (Repeat Specification) is optional, and is an unsigned integer constant.

```
w (Width Specification) is an unsigned integer constant.

d (Fractional Decimal Places) is an unsigned integer constant.

n (Scale Factor) is an optionally signed integer constant.
```

Note 1: The comma may be omitted before or after a slash Edit Descriptor (/).

Note 2: The commas must be omitted after the np Edit Descriptor, and nP must be immediately followed by a D, E, F, or G Edit Descriptor.

Note 3: See Appendix G for free-format I/0 and for additional information concerning formatter operation.

2.4 NOTES

The following notes describe ODDS AND ENDS that pertain to the SSS FORTRAN language.

1. Only CAPITAL letters are permitted in SSS FORTRAN commands and variable names. Any symbol may be used in a string literal except a carriage return, which is in all cases a reserved symbol.

2. Input source lines may be of any length, terminated by a carriage return/line feed. As a consequence of this, columns 72-80 are not used for sequence information.

3. SSS FORTRAN code is re-entrant and recursive. But, no automatic storage capabilities are available except through dynamic allocation by the user. All FORTRAN variables are static.

4. SSS FORTRAN has the following statement ordering restrictions:

A. IMPLICIT statements should precede all other statements in a main program, and should directly follow the FUNCTION or SUBROUTINE statement in a subprogram. IMPLICIT statements are applied in the order that they appear in the program. Any conflicts in IMPLICIT specifications are resolved by their ordering.

B. DATA statements for a given variable must follow any type or DIMENSION statements for the variable.

C. Statement function definitions must follow all other non-executable statements.

5. Function return types must match the type given to the function in the calling program, or undefined results will occur. The only exception to this rule is that INTEGER*1, INTEGER*2, and LOGICAL types are considered equivalent for function returns.

6. Sometimes it is desirable to pass LOGICAL or INTEGER*1 values to subprograms that expect INTEGER*2 or INTEGER arguments. This is possible by using the call by value feature. For example:

```
      LOGICAL Q
10    CALL SUB((10#),(Q))

      SUBROUTINE SUB(A,B)
      INTEGER*2 A,B
```

will give expected results whereas, deletion of the extra parentheses in statement 10 above will not, since call by reference will be used.

7. The EXTERNAL statement can be used to define variables as external, if they are not parameters. In the case of CHARACTER strings or arrays, the dope vector of the item is made to be external, and has the same name as the item (see APPENDIX D for details on dope vectors). A variable that has been declared EXTERNAL cannot be EQUIVALENCEd or initialized with a DATA statement.

8. CHARACTER variables or string constants used in a COMMON block have their values put in COMMON, not their current or maximum lengths. Consider the following example.

```
      CHARACTER*12 STRING
      COMMON STRING
      DATA STRING /ABCDE/
      CALL XXX
      STOP
      END
      SUBROUTINE XXX
      CHARACTER*12 STRING
      COMMON STRING
      WRITE(1,100) STRING
100   FORMAT(lX,AO)
      RETURN
      END
```

This program prints the null string. In order to achieve what was probably intended, the following program should be used.

```
      CHARACTER*12 STRING
      COMMON STRING, LENGTH
      DATA STRING / ABCDE/
      LENGTH=KLEN(STRING)
      CALL XXX
      STOP
      END
      SUBROUTINE XXX
      CHARACTER*12 STRING
      COMMON STRING, LENGTH
      CALL SETLEN(STRING,LENGTH)
      WRITE(1,100) STRING
100   FORMAT(lX,AO)
      RETURN
      END
```

This implementation allows true overlaying of strings in COMMON. For example, a string of length 100 may be overlayed on top of two strings of length 50 as follows:

```
      CHARACTER*50 STRI, STR2
      COMMON STRI, STR2

      SUBROUTINE XXX
      CHARACTER*100 STRING
      COMMON STRING
      CALL SETLEN(STRING,IOO)
```

3 USING THE COMPILER

3.1 GENERAL USAGE

This chapter assumes a working knowledge of the CP/M Operating System.

The compiler is a module in executable form with the name FOR.COM. To compile a program, two files are necessary, an input file and an output file. The input file should contain the FORTRAN source code to be compiled, while the output file will be produced by the compiler and will contain the object code representation of the given source code. The compiler produced object file is normally in binary form. The compiler also produces a source listing and map on the console device, or, using the ;DISK switch, on a disk file.

The basic form for compiling a source module named, say, INP.FOR, is

```
A>FOR INP.FOR OUT.REL
```

The output file in this case would then be named OUT.REL. If no extension for the input file is given, a default of .FOR is assumed. The default output extension is .REL.

```
A>FOR INP OUT
```

The above would suffice if the file INP had an extension of .FOR, and the output file would be named OUT.REL. If a file with the same name as the output file already exists when a compile command is issued, the existing file is deleted, and a new file with the same name is opened.

If no output file is given, the input file name is used, with an extension of .REL, to form the output file name. For example,

```
A>FOR INP
```

compiles INP.FOR to INP.REL. Standard unit specifications as used in CP/M may also be used.

```
A>B:FOR B:INP
```

assumes that the compiler and the input file reside on unit B:, while the output file will be named INP.REL and will reside on the logged-in unit.

3.2 COMPILER SWITCH OPTIONS

Several switches are available for controlling compiler execution. They are:

`;LISTOFF`

Turns off the source listing and map, so that all information (error messages and identifier names) is sent to the console device. The default is LIST ON, in which case all information goes to the list device.

`;MAP`

Turns off the statement and variable map which is produced by default. Used in conjunction with the ;LISTOFF switch, this switch turns on the map. Thus, ;LISTOFF;MAP produces no source listing but does produce a map.

`;OUTOFF`

Specifies that no object file be created for this current compilation. The default is OUTON. This switch is useful for syntax checking of programs.

;PAGE

Normally any information sent to the list device during compilation is broken into numbered pages of 66 lines each, with no special attempt at alignment of information with page boundaries. This switch specifies that each subprogram and its map be forced to a new page, with the last page padded with blank lines to total 66 lines. This switch is useful for those who have line printers and wish segmented output.

;HEXOUT

This flag specifies that the compiler produce the object module for the current compilation in HEX-ASCII format. This switch may be useful for storing object modules on paper tape, but its main use is to aid in maintenance of the compiler (see NOTES, Section III.C). The default form of output is binary.

;DISK

This flag specifies that the compiler list output be placed on a disk file (with extension .LST) rather than on the list device.

The compiler switches as given above can be used in the command line, and should be specified after any file information given. Ordering of the switches is not important, and only the first letter after the ; is significant.

Examples:

```
A>FOR TEST ;L ;M
```

specifies no listing be produced, but that the map be produced.

```
A>FOR TEST ;0 ;M ;D
```

specifies an input file of TEST.FOR with no output object file or map produced by the compilation, and with the list output written to file TEST.LST.

3.3 NOTES

The following notes pertain to the compiler and its usage.

1. TABS (control-I) may be included in the input stream of the compiler. The first tab positions to column 7 unless it is followed by a digit from l to 9. If followed by a digit, the current line is assumed to be a continuation line, with the digit being the continuation mark. Succeeding tabs move to multiples of column 8, starting with column 16, (16,24,32,...). All tabs are expanded in the source listing.

Example: (indicates a tab character)

```
C2345678901234567890
10 GOTO 20
```

yields:

```
C2345678901234567890
10    GOTO     20
```

2. Multiple program segments are permitted in a single compilation. That is, many subprograms, or a main program and many subprograms may be compiled in one execution of the compiler.

3. Blank lines are allowed in SSS FORTRAN.

4. A @ character in column 1 of a statement is treated as a C, (comment line), except that the compiler list flag is toggled, (see Section III B on Compiler Switch Options).

5. If the list flag is set, source listing on, error messages are intermixed with the source listing and sent to the list device. A character is included in the source listing to indicate the approximate position of the error. If the list flag is reset, source listing off, all information is sent to the console device.

6. During compilation the following trap characters are recognized from the console device.

Control-S Stop execution; wait for Control-Q
Control-Q Resume execution after Control-S
Control-C Terminate execution; return to CP/M

7. To achieve maximum compilation speed, the following should be noted.

A.. Disk head movement is minimized if the input source file and output object files are on separate units.

B. The source listing takes time. Cure: use the LISTOFF feature whenever possible.

C. Using the OBJECT-OFF flag for syntax checking speeds up compilation since much of the waiting for disk time is avoided.

8. Any suspected errors that may result from using the compiler should be documented with the following information.

A. Compiler output listing with full listing and map produced. Version number of the compiler should be present on the listing.

B. A listing of the object module produced by the compilation, in hex form (;H flag used).

C. Any loader output that is significant.

D. Any other information that can be supplied by the user that may be significant.

9. In the symbol table statement map, statement numbers and their offsets in the appropriate area are listed. For each ERROR 200 (undefined label), the undefined statement number(s) will be marked, with a zero offset.

10. In the symbol table variable map, the following information is presented:

Type Flags

B LOGICAL or INTEGER*1
I INTEGER*2
R REAL*4
D REAL*8 or DOUBLE PRECISION
C COMPLEX
S STRING

Attribute Flags:

A variable is an array
C variable is in COMMON
P variable is a parameter
E variable is EXTERNAL
V value of variable has been used
D variable has been assigned a value

Program and data area lengths are also given, in hex, for each segment.

11. If errors are detected by the compiler, the file $$$.SUB is, if present, erased.

12. The compiler supports a library insert feature. This feature allows the insertion of source text to the compiler from library files. Insertion is triggered by a # in column 1 of a source file being compiled. The # character should be followed by a valid CP/M filename, for example:

```
#     B: FILE.EXT
```

however, the filename may contain only letters and digits (not special symbols). When triggered, the compiler will read from the library file until and end-of-file is encountered. At this point the compiler reverts to the original file for source input. Note that this feature is restricted to a ONE-LEVEL insert only, i.e. the library file may not have any # s in column 1.

Blanks and tabs are ignored in the insert specification statement. If no unit is given, the logged-in unit is assumed. If no extension is given, a default of FOR is assumed. If the specified file is not found, or if more than one level of insertion is attempted, an error number 186 is issued.

4 LOADER

4.1 GENERAL USAGE

LOADER is the SSS FORTRAN Linking Loader, a system utility that merges independently compiled or assembled modules into an executable binary image. Its function is to relocate all addresses in each module and resolve external references, searching certain files for necessary modules.

Input to LOADER is a number of files that are the concatenated output of an assembler (producing TDL or Xitan compatible REL files), and the SSS FORTRAN Compiler, Version 2.0 or later. LOADER outputs the executable image of loaded modules, and also a map of module locations and @ symbol table if specified.

4.2 COMMAND STRING FORMAT

LOADER is executed under the CP/M Operating System with a command string of the form:

```
A>LOADER [outfile=] infile [,infile]... [/switch]... [,libfil]...
```

where outfile, infile, and libfile are times. If the unit is not specified in a file specification, then the logged-in unit is assumed. The extension has default of REL for input files, and COM for the output file. If the output file specification is not present, the first input file name is used. For example, a LOADER command string to load TEST.REL and PROG.REL outputting to TEST.COM would be:

```
A>LOADER TEST,PROG
```

If, for the input files given above, the output file was to be TESTY.COM, the following command could be used:

```
A>LOADER TESTY=.TEST,PROG
```

If a file cannot be found or part of the command is mistyped so that it cannot be interpreted, an error message is issued and the remainder of the command may be retyped. For example:

```
A>LOADER TEST,UDPATE,PROG
A:UDPATE.REL FILE NOT FOUND!
>UPDATE,PROG
```

4.3 OUTPUT FORMAT

The output from LOADER is a core image (a COM file), with an initialization routine located at 100H. The block contains a stack pointer initialization, a debug patch area, a pointer to the first free byte, and

a jump to the global symbol .MAIN. . The user should define .MAIN. entry to any main program. The initialization routine has the following format and global symbols.

```
  100H:
.SYSB.: LSPD            6       .,STACK INIT
.DBUG.: NOP                     ;DEBUG PATCH LOCATIONS
        NOP
        NOP
        JMP             .MAIN.   START
.SYMB.: .WORD.          DBGDT   ;DEBUG IDENT LIST
.FREE.: .WORD.          PRGEND  ;FIRST FREE BYTE ADDRESS
```

The first module is relocated to 110H. Subsequent modules are loaded at sequentially higher addresses. The LOADER utilizes two relocation counters, one for data and one for code. The output from the loader may also be standard checksum format, if the /HEX switch is used. The user may also load and execute a program without outputting the image by including the /GO switch.

A map of the loaded modules and their relocation addresses is output to the list device if a /MAP switch is present in the command. Similarly, a /SYM will list the global symbols in order of definition. Symbols are preceded by a ? if a duplicate symbol was present in the input stream. If a duplicate identifier name is noted, it is skipped. This is useful for loading updates before the balance of a program. The updated modules are loaded! and older idents with the same names are skipped. Appropriate warnings are typed out when duplicates are encountered.

```
? DUPLICATE IDENT name
? DUPLICATE SYMBOL name
```

4.4 LIBRARIES

After the user's modules are loaded the system subroutine library, LIBRARYS.REL (which must reside on the logged-in device), is searched for any modules that contain entries referenced by the loaded modules. If a necessary entry is found, that module is loaded. The user may build his own libraries (see Section VI), and search them by using the /LIB command switch. All files following a /LIB are loaded in library search mode; only previously referenced modules are taken from the library and included.

If there exist undefined symbols after LIBRARYS.REL has been searched, they are typed and the user may define them in hexadecimal notation, or he may load other files in an attempt to define the symbols. For example:

```
UNDEFINED SYMBOLS
INPORT>A
LSQFIT>;
>LSQPAK
```

INPORT is defined as OOOAH, and then command mode is re-entered when the symbol LSQFIT is shown as undefined, and the file LSQPAK is then loaded.

4.5 LOADER ERRORS

LOADER is a one pass linking loader utilizing all of core for its 4K program, the symbol table, and storage for the program being loaded. The message INSUFFICIENT MEMORY IS SELF EXPLANATORY. THE LODFRE SYMBOL WHICH is listed on any ident map produced is an indicator of the unused amount of memory during the loading process.

Although it is unlikely, the message INVALID INPUT is typed when some input module has an invalid format, a symbol containing invalid characters, etc.

An error may be encountered when LOADER shuffles the core image into the proper locations for binary output, or a load-and-go. If this occurs, an appropriate message is typed and the user may reconstitute the core image by outputting a hex file (with the /HEX switch), and running the LOAD.COM program. For example:

```
LOADER TEST

? OVERWRITE USE /HEX! ?

LOADER TEST/H
LOAD TEST
```

The OVERWRITE error will most likely be encountered when absolute modules located at high addresses are loaded before re-locatable modules. The problem is alleviated by loading absolute modules in their location order, after re-locatable modules are loaded. Note that a special LOAD program is supplied with the SSS FORTRAN system. This LOAD program should be used in lieu of the CP/M LOAD utility, when you have generated .HEX files with the /H switch using LOADER.

4.6 SWITCH OPTIONS

The LOADER switches may be truncated to one letter and must be followed by a delimiter. The switches are:

/GO or /G load and execute, no output is produced

/HEX or /H output a checksum format file with a hex extension

/LIB or /L all following files are searched in library search mode

/MAP or /M output a module or ident map

/SYM or /S output a map of the global symbols

4.7 NOTES

1. The LOADER responds to Control-S, Control-Q, and Control-C, as does the compiler.

2. If a load error occurs, then the file $$$.SUB is erased, if present.

5 I/0 PACKAGE

5.1 GENERAL USAGE

The FORTRAN I/0 package supports all CP/M interfaced devices, and supplies the necessary linkages to support any user interfaced device. The I/0 package uses logical units 1 through 4 to reference the console, punch, reader, and list devices, and units 5 through 20 to reference the disk files. While the console, punch, reader and list devices are always open and assumed online, the disk units must be explicitly opened and closed. The disk files may be accessed sequentially, either in Stream or Blocked Mode, or may be accessed randomly in Stream Mode.

A Stream Mode file is a sequential file without record information added. The Blocked Mode files contain record lengths at the start of each record so that they resemble magnetic tape devices. The random access files are Stream Mode files with which the user controls the READ or WRITE address by means of the record number. There are different library entries for each different type of access, so only the necessary routines are loaded.

5.2 CALLABLE ROUTINES

All file access routines are FORTRAN functions returning .TRUE. on errors. The disk I/0 routine explanations follow.

DEFINITIONS

UNIT = Integer between 5 and 20

MODE = 0 binary Stream Mode
1 binary Blocked Mode
2 ASCII Stream Mode

DEV = Integer number of drive, or 0 to select the logged-in device. This may be overridden by the file specification.

FILE = CHARACTER variable or literal, (STRING) of the form D:FILENAME.EXT as in standard CP/M file descriptions, except that only numbers and letters are allowed.

Note that ASCII files are stream files composed of ASCII characters ending with a Z (Control-Z), as an end-of-file mark.

LOGICAL FUNCTION IOREAD (UNIT, MODE, DEV, FILE)

OPENS the specified file on the indicated logical unit. Returns .TRUE. if no such file exists or unit is already open.

LOGICAL FUNCTION IOWRIT @NIT, MODE, DEV, FILE)

DELETES, then CREATES the specified file on the indicated unit. Returns .TRUE. if no directory space or the file is already open.

LOGICAL FUNCTION IORAND (BLKSIZ, RECSIZ, UNIT, DEV, FILE)

Initializes the unit for random access I/0 on the specified file. BLKSIZ is the size in bytes of a block that contains BLKFAC records each of size RECSIZ. I/0 is optimized if BLKSIZ is 128, the size of a disk block, or a multiple thereof. Each I/0 operation computes the address of the record by:

```
BLKFAC = BLKSIZ/RECSIZ
BLOCK = (RECORD-1)/BLKFAC
REC = IMOD(RECORD-l,BLKFAC)
```

ADR = BLOCK*BLKSIZ + REC*RECSIZ

I/0 proceeds from the calculated ADR in the file in Stream Mode. Returns .TRUE. if error condition. Note that RECORD=1 is the first record in a file.

LOGICAL FUNCTION IOAPND (UNIT, MODE, DEV, FILE)

CREATES the file if non-existent, or APPENDS to the next free block if it already exists. The last block of the file was zeroed by IOCLOS which will result in zeroes if the file is read in Stream Mode. Returns .TRUE. if no directory space or unit is already open.

LOGICAL FUNCTION IOCLOS@NIT)

Breaks unit-file correspondence. If the file is a sequential file, IOCLOS zeroes the remainder of the disk block and updates the directory. If the file is a random access unit, IOCLOS updates all file control blocks. Returns .TRUE. if unit is not open.

LOGICAL FUNCTION IOREW (UNIT)

RESETS the unit to read from or write to the start of the file. Equivalent to IOCLOS followed by either IOWRIT or IOREAD. Returns .TRUE. if unit is not open.

LOGICAL FUNCTION IODEL (DEV, FILE)

DELETES the specified file or returns .TRUE. if no such file exists.

LOGICAL FUNCTION IOREN (DEV, FILEI, FILE2)

RENAMES FILE1 to FILE2. Returns .TRUE. if FILE1 does not exist or FILE2 already exists.

LOGICAL FUNCTION IOLOOK @EV, FILE)

Returns .TRUE. if the specified file exists.

5.3 USER INTERFACE

The user may write device controllers and interface them in the following manner. USRINZ, a user supplied subroutine, is called to initialize a users device before an I/0 operation. The following locations contain the READ/WRITE parameter.

IORDWT	nonzero byte if WRITE
IOUNIT	byte with unit number
IOFFLG	nonzero byte if Formatted I/0
IORFLG	nonzero byte if Random I/0
IORECD	record number or zero

The user should place the address of his device service routine for the indicated unit at IOXADR and the address of his device s end-of-transfer routine at IOEADR. All I/0 operations use IOBYTE for the read/write byte. The user s service routines should input or output the byte at IOBTYE each time they are called. On end-of-file, the service routine should JUMP to IOENDX. On device errors, the service routines JUMP to IOERRX with the C register equal to the appropriate error code.

The USRINZ routine has stored the address of the devices end-of-transfer routine at IOEADR so that any cleanup or device control to cease the transfer may be accomplished. This does not happen, however, if the user's routines jumped to IOENDX or IOERRX. Appendix 1 gives an example of adding a user device.

The FORTRAN device control statements are not currently implemented by the system I/0 package, but are reserved for tape-like devices. These statements (REWIND, BACKSPACE, ENDFILE, and PAUSE) generate calls to REWIND, BACKSPACE, ENDFILE, and PAUSE with the unit number in the

HL register pair. The user may currently utilize these calls for appropriate control of his custom interfaced devices, but such use is not encouraged.

5.4 NOTES

1. Items of type CHARACTER output to units greater than 4 and less than 51 are preceded by the length of the string (binary stream mode only). On input from these devices, strings are assumed preceded by their length. Strings input from devices less than 5 or greater than 50 are terminated by a carriage return (ASCII 13).

2. The I/0 package allocates buffers dynamically. If memory is exceeded an appropriate message is typed out. Approximately 160 bytes are utilized for each sequential file and 420 bytes for each random access file.

3. Device or formatter errors that are not trapped are typed out with the message:

```
RUNTIME ERROR = XX ON LAST UNIT = YY
```

4. The list of runtime errors is included in APPENDIX A. A list of I/0 unit assignments in included as APPENDIX F.

5.5 EXAMPLE PROGRAMS

The following programs are examples of sequential and unformatted random access disk I/0.

```
C EXAMPLE OF AN UNFORMATTED RANDOM ACCESS ROUTINE.

     CHARACTER*8 RANFIL
     DATA RANFIL /TEST.DAT/
C    OPEN THE FILE
     IF (IORAND(2,2,5,0,RANFIL)) GOTO 300
C    WRITE EVERY 2**N RECORD
     DO 10 I=0,12
10   WRITE(5/ISHIFT(l,I)) I
     IF (IOCLOS(5)) GOTO 300
C    TEST IF DATA WRITTEN CORRECTLY
     IF (IORAND(2,2,5,0,RANFIL)) GOTO 300
     DO 20 I=0,12
     READ(5/ISHIFT(l,I)) J
20   IF (J .NE. I) GOTO 300
     WRITE(1)  ALL OK
     STOP
300  WRITE(1)  ?ERROR?
     STOP
     END

C  EXAMPLE OF SEQUENTIAL I/0 ROUTINE

     CHARACTER*15 FILE
     REAL*8 BUF(16)
10   WRITE(1)  FILE =
     READ(1) FILE
     IF (IOREAD(5,0,0,FILE)) GOTO 40
     IF (IOWRIT(6,0,0, ;@X.XXX)) GOTO 50
20   READ(5,ENDFILE=30) BUF
     WRITE(6) BUF
     GOTO 20
30   IF (IOCLOS(6) .OR. IODEL(O,FILE)) GOTO 60
     IF (IOREN(0, .@XX.XXX,FILE)) GOTO 60
     WRITE(1)  DONE@
```

```
      STOP
40    WRITE(1)  NO SUCH FILE
      GOTO 10
50    WRITE(1)  CANNOT CREATE
      STOP
60    WRITE(1)  RENAME ERROR
      STOP
      END
```

6 LIBRARIES

6.1 GENERAL USAGE

The SSS FORTRAN package is supplied with the FORTRAN Systems Library (LIBRARYS.REL) and the FORTRAN Math Library (MLIB.REL). LIBRARYS.REL consists of the following modules.

1. FORTRAN Formatter Library
2. FORTRAN I/0 Package Library
3. FORTRAN String Handling Library
4. FORTRAN Dynamic Allocation Library
5. Miscellaneous routines used by the FORTRAN system, and assembly language linkage routines.

MLIB.REL consists of the following modules.

1. FORTRAN Intrinsic Function and Built-in Function Library
2. Single Precision Math Library
3. Double Precision Math Library
4. Complex Math Library.

LIBRARYS.REL is always automatically searched by LOADER; however, if a program requires any portion of MLIB.REL, then MLIB.REL must be searched via the LOADER /LIB option (see Section IV).

The next sections of this chapter will deal with pertinent information concerning the SSS FORTRAN system library that has not been previously detailed in this manual.

The user can of course construct his own libraries by merging the object modules of several programs that he has produced together. This process can be accomplished by using the CP/M PIP command. It should be noted that the ordering of modules in a library is important. Modules should be put in an order such that modules that are to be included because of the inclusion of another module appear in the library after the first module to be included. For example, if the user has two modules A and B that are to be made into a library, if module A is to be called directly by a program to be loaded, or referenced by a program previously loaded, and if module B is to be included because of the presence of module A, then module A should appear before module B in the library. Consider the following sequence of commands.

```
A>FOR A
A>FOR B
A>PIP LIBl.REL=A.REL,B.REL
A>LOADER TEST/L,LIBl
```

File A.FOR is compiled to yield A.REL, and file B.FOR is compiled to yield B.REL. These two modules are then merged using the PIP utility to yield a single module called LIB1.REL. Note that the ordering in the library of the modules is A then B. In the last command the object module TEST.REL is loaded and LIB1.REL is now searched as a library. That is, only if referenced by TEST, will modules A or B be loaded. Note that the system library LIBRARYS.REL is always searched after all originally specified loading is complete; in this case after TEST has been loaded and LIB1 has been searched.

Note again that a module currently being scanned by LOADER cannot CALL in A PREVIOUSLY SCANNED MODULE. IF THIS SITUATION OCCURS, UNDEFINED SYMBOLS will result, and a second pass through the same library, by LOADER, would be necessary. Some care must be exercised in ordering modules in a library in complex referencing situations.

The user need not concern himself with the details of library search and ordering of modules if only FORTRAN work using the provided library routines is to be done. The FORTRAN System Library is always automatically searched by LOADER.

6.2 INTRINSIC AND BUILT-IN FUNCTION LIBRARY

The following is a list of the routines provided in the SSS FORTRAN Math Library (MLIB.REL) that are considered as built-in or intrinsic functions. The details of the operation of these routines can be found in any reference on Standard FORTRAN.

NAME	# OF ARGS	TYPE OF ARGS	RETURN TYPE
INT	1	REAL	INTEGER
AMOD	2	REAL	REAL
ABS	1	REAL	REAL
IABS	1	INTEGER	INTEGER
DABS	1	DOUBLE	DOUBLE
FLOAT	1	INTEGER	REAL
IFIX	1	REAL	INTEGER
SIGN	2	REAL	REAL
ISIGN	1	INTEGER	INTEGER
MOD	2	INTEGER	INTEGER
AIMAG	1	COMPLEX	REAL
DBLE	1	REAL	DOUBLE
DSIGN	2	DOUBLE	DOUBLE
DIM	2	REAL	REAL
IDIM	2	INTEGER	INTEGER
SNGL	1	DOUBLE	REAL
REAL	1	COMPLEX	REAL
CMPLX	2	REAL	COMPLEX
CONJG	1	COMPLEX	COMPLEX
CABS	1	COMPLEX	COMPLEX
DMOD	2	DOUBLE	DOUBLE
IDINT	1	DOUBLE	INTEGER
AINT	1	REAL	REAL
DINT	1	DOUBLE	DOUBLE
DMAXI	*	DOUBLE	DOUBLE
DMINI	*	DOUBLE	DOUBLE
AMAXO	*	REAL	INTEGER
MAXO	*	INTEGER	INTEGER
AMINO	*	REAL	INTEGER
MINO	*	INTEGER	INTEGER
AMAXI	*	REAL	REAL
MAXI	*	INTEGER	REAL
AMINI	*	REAL	REAL
MINI	*	INTEGER	REAL
INTI	1	INTEGER*2	INTEGER*1

MAX2	*	INTEGER*1	INTEGER*2
MIN2	*	INTEGER*1	INTEGER*2
AMAX2	*	INTEGER*1	REAL
AMIN2	*	INTEGER*1	REAL

NOTE 1: INTEGER means 2-byte INTEGER, REAL means 4-byte floating point, DOUBLE means 8-byte floating point.

NOTE 2: FUNCTIONs which return DOUBLE PRECISION or COMPLEX results must be declared as to Type in the calling program. For example:

```
DOUBLE PRECISION DSIGN
REAL*8 DBLE,Q
A = DSIGN(DBLE(B),Q)
```

NOTE 3: A * in the number of arguments column above indicates that the number of arguments must be at least 1, but not greater than 85.

NOTE 4: DINT above is the DOUBLE PRECISION equivalent of AINT.

NOTE 5: INT1, MAX2, MIN2, AMAX2, and AMIN2 are the INTEGER*1 equivalents of IFIX, MAX1, MIN1, AMAX1, and AMIN1, respectively.

6.3 UTILITY ROUTINE LIBRARY

The following routines provide the user with the ability to read from and write to Z80 I/0 ports directly and also to directly modify or read memory locations.

INTEGER@. FUNCTION INPT(PORT)

A function that inputs the current @ byte value of the specified port.

SUBROUTINE OUTPT(PORT,VALUE)

A subroutine that outputs 1 byte to the specified port.

INTEGER*2 FUNCTION IPEEK(ADDR)

A function that returns the value of the memory word at address ADDR.

SUBROUTINE POKE1(ADDR,VALUE)

A subroutine that inserts (POKEs) 1 byte into memory at the address ADDR.

SUBROUTINE POKE2(ADDR,VALUE)

Same as POKE1, except 2 bytes are inserted into memory.

6.4 SSS FORTRAN MATH LIBRARY

The following is a short description of the single, double, and complex math functions available in SSS FORTRAN. Note that functions returning DOUBLE PRECISION or COMPLEX results must be declared as to type in the program in which they are called.

ATAN(A)	REAL*4 FUNCTION	REAL*4 ARGUMENT
	DEFINITION:	ARCTAN A
	ERROR CODE:	NONE
	INPUT:	ANY REAL*4 NUMBER
	OUTPUT:	-PI <= ATAN(A) <= PI

DATAN(DA)	REAL*8 FUNCTION	REAL*8 ARGUMENT
	DEFINITION:	ARCTAN DA
	ERROR CODE:	NONE
	INPUT:	ANY REAL*8 NUMBER
	OUTPUT:	-PI <= DATAN(DA) <= PI

ATAN2(Y,X)	REAL*4 FUNCTION	REAL*4 ARGUMENT
	DEFINITION:	ARCTAN(Y,X)
	ERROR CODE:	23 BOTH ARGUMENTS = 0
	INPUT:	ANY PAIR OF REAL*4 NUMBERS
	OUTPUT:	-PI <= ATAN2 <= PI

DATAN2(DY,DX)	REAL*8 FUNCTION	REAL*8 ARGUMENT
	DEFINITION:	ARCTAN(DY,DX)
	ERROR CODE:	23 BOTH ARGUMENTS = 0
	INPUT:	ANY PAIR OF REAL*8 NUMBERS
	OUTPUT:	-PI <= DATAN2 <= PI

ALOG(A)	REAL*4 FUNCTION	REAL*4 ARGUMENT
	DEFINITION:	LOG BASE e OF A
	ERROR CODE:	24 INPUT < 0
	INPUT:	ANY REAL*4 NUMBER > 0.0
	OUTPUT:	APPROX -172. TO +172

DLOG(DA)	REAL*8 FUNCTION	REAL*8 ARGUMENT
	DEFINITION:	LOG BASE e OF DA
	ERROR CODE:	24 INPUT < 0
	INPUT:	ANY REAL*8 NUMBER > 0.0
	OUTPUT:	APPROX -172. TO +172.

ALOGIO(A)	REAL*4 FUNCTION	REAL*4 ARGUMENT
	DEFINITION:	LOG BASE 10 OF A
	ERROR CODE:	24 INPUT < 0
	INPUT:	ANY REAL*4 NUMBER > 0.0
	OUTPUT:	APPROX -79. TO +77

DLOGIO(DA)	REAL*8 FUNCTION	REAL*8 ARGUMENT
	DEFINITION:	LOG BASE 10 OF DA
	ERROR CODE:	24 INPUT < 0
	INPUT:	ANY REAL*8 NUMBER > 0.0
	OUTPUT:	APPROX -79. TO +77.

TANH(A)	REAL*4 FUNCTION	REAL*4 ARGUMENT
	DEFINITION:	TANH A
	ERROR CODE:	21 INPUT RANGE EXCEEDED
	INPUT:	-171. < A < 171.
	OUTPUT:	-1. < TANH < 1.

DTANH(DA)	REAL*8 FUNCTION	REAL*8 ARGUMENT
	DEFINITION:	TANH DA
	ERROR CODE:	21 INPUT RANGE EXCEEDED
	INPUT:	-171. < DA < 171.
	OUTPUT:	-1. < DTANH < 1.

SIN(A)	REAL*4 FUNCTION	REAL*4 ARGUMENT
	DEFINITION:	SIN A, A IN RADIANS
	ERROR CODE:	22 INPUT RANGE EXCEEDED
	INPUT:	-102937. < A < 102937
	OUTPUT:	-1. <= SIN <= 1.

DSIN(DA)	REAL*8 FUNCTION	REAL*8 ARGUMENT
	DEFINITION:	SIN DA, DA IN RADIANS
	ERROR CODE:	22 INPUT RANGE EXCEEDED
	INPUT:	-102937. < DA < 102937.
	OUTPUT:	-1. <= DSIN <= 1.

COS(A)	REAL*4 FUNCTION	REAL*4 ARGUMENT
	DEFINITION:	COS A, A IN RADIANS
	ERROR CODE:	22 INPUT RANGE EXCEEDED
	INPUT:	-102935. <= A <= 102935.
	OUTPUT:	-1. <= COS <= 1.

DCOS(DA)	REAL*8 FUNCTION	REAL*8 ARGUMENT
	DEFINITION:	COS DA, DA IN RADIANS
	ERROR CODE:	22 INPUT RANGE EXCEEDED
	INPUT:	-102935. <= DA <= 102935
	OUTPUT:	-1. <= DCOS <= 1

EXP(A)	REAL*4 FUNCTION	REAL*4 ARGUMENT
	DEFINITION:	E**A
	ERROR CODE:	21 INPUT RANGE EXCEEDED
	INPUT:	-171. <= A <= 171.
	OUTPUT:	E**-171. <= EXP <= E** 71.

DEXP(DA)	REAL*8 FUNCTION	REAL*8 ARGUMENT
	DEFINITION:	E**DA
	ERROR CODE:	21 INPUT RANGE EXCEEDED
	INPUT:	-171. <= DA <= 171.
	OUTPUT:	**-171. <= DEXP <= E** 71.

SQRT(A)	REAL*4 FUNCTION	REAL*4 ARGUMENT
	DEFINITION:	SQUARE ROOT OF A
	ERROR CODE:	25 INPUT NEGATIVE
	INPUT:	ANY REAL*4 NUMBER >= 0.0
	OUTPUT:	0 TO APPROX 1.E38

DSQRT(DA)	REAL*8 FUNCTION	REAL*8 ARGUMENT
	DEFINITION:	SQUARE ROOT OF DA
	ERROR CODE:	25 INPUT NEGATIVE
	INPUT:	ANY REAL*8 NUMBER >= 0.0
	OUTPUT:	0 TO APPROX 1.E38

Note 1: For the above functions, on error, zero is returned.

Note 2: INPUT and OUTPUT refer to allowable ranges.

CSIN(A)	COMPLEX FUNCTION	COMPLEX ARGUMENT
	DEFINITION:	SINE CA, CA IN RADIANS
	ERROR CODE:	21 IMAGINARY INPUT RANGE EXCEEDED 22 REAL INPUT RANGE EXCEEDED
	INPUT:	REAL PART -102935. TO +102935. IMAGINARY PART -171. TO +171.
	OUTPUT:	REAL PART -(E**171./2.) TO (E** 71./2.) IMAGINARY PART (E**171./2.) TO (E** 71./2.)

CCOS(CA)	COMLEX FUNCTION	COMPLEX ARGUMENT
	DEFINITION:	COS CA, CA IN RADIANS
	ERROR CODE:	21 IMAGINARY INPUT RANGE EXCEEDED 22 REAL INPUT RANGE EXCEEDED
	INPUT:	REAL PART -102935. TO +102935 IMAGINARY PART -171. TO +171.
	OUTPUT:	REAL PART -(E** 71./2.) TO E ** 71./2.) IMAGINARY PART (E**171./2.) TO (E**171./2.)

CEXP(CA)	COMLEX FUNCTION	COMPLEX ARGUMENT
	DEFINITION:	E**CA
	ERROR CODE:	21 REAL INPUT RANGE EXCEEDED 22 IMAGINARY INPUT RANGE EXCEEDED
	INPUT:	REAL PART -171. TO +171. IMAGINARY PART -102935. TO +102935.
	OUTPUT:	REAL PART -E** 71. TO E** 71. IMAGINARY PART -E**.71. TO E**171.

CLOG(CA)	COMLEX FUNCTION	COMPLEX ARGUMENT
	DEFINITION:	LOG BASE e CA
	ERROR CODE:	23, 24 REAL AND IMAGINARY PARTS BOTH = 0
	INPUT:	ANY COMPLEX NUMBER
	OUTPUT:	REAL PART APPROX -172. TO +172. IMAGINARY PART APPROX -PI TO PI

CSQRT(CA)	COMLEX FUNCTION	COMPLEX ARGUMENT
	DEFINITION:	SQUARE ROOT OF CA
	ERROR CODE:	NONE
	INPUT:	ANY COMPLEX NUMBER
	OUTPUT:	ANY COMPLEX NUMBER

Note: For all COMPLEX functions, error conditions may give undefined results.

6.5 STRING AND DYNAMIC ALLOCATION LIBRARY

The routines in this library may be grouped as: SPECIAL ARITHMETIC, DYNAMIC ALLOCATION, and STRING.

The SPECIAL ARITHMETIC routines provide quick shifts of INTEGER, REAL, and DOUBLE PRECISION arguments. They also include routines to return the INTEGER binary logarithm of the argument, and set the binary exponent. These routines are used for range reduction by the routines in the Math Library.

The DYNAMIC ALLOCATION routines have been included to alleviate one of FORTRANs major deficiencies, namely its lack of pointers to accommodate linked lists.