RC9000-10/RC8000

SW8585 Compiler Collection

RC FORTRAN, User's Manual

Keywords:
RC9000-10, RC8000, FORTRAN, Compiler, ALGOL

Abstract:
This manual desribes the RC FORTRAN Compiler for RC8000 and RC9000-10.

Date:
January 1989

PN: 991 11292

Table of Contents

1. External And Internal Representations

1.1 Character Set and Compounds

The RC FORTRAN character set is a subset of the ISO 7-bit character set extended with the Danish letters: æ, Æ, ø, Ø, å, Å (see Ref. 5). Programs written in an external representation (console typewriter, punched cards, punched paper tape, etc.) are converted internally to the ISO representation. The RC FORTRAN character set is an extension of the ISO FORTRAN character set as small letters are allowed and also many special characters may appear in texts and comments.

The character set table (Table 1.1.1) shows

(1) the decimal value of the ISO 7-bit representation.
(2) the character, or an abbreviated name for the character.
(3) the program input class.

The treatment of the characters depends on their appearence in the program text as defined in Section 1.2. The effect is shown in Table 1.1.2.

RC FORTRAN distinguishes between capital and small letters in texts only. For simplification the syntax using capital letters only is given. Names of characters are written in the abbreviated form shown in Table 1.1.1. The used syntax description is explained in Appendix B.

The legal characters are divided into syntactical groups:

```
<symbol>    ::=    { <letter>                    }
                   { <digit>                     }
                   { <separator>                 }
                   { <terminator>                }
                   { <graphic>                   }
                   { <arithmetical operator>     }
                   { <blind>                     }
                   { <in text>                   }

<blind>     ::=    { NUL                         }
                   { DEL                         }

<in text>   ::=    { SP                          }
                   { _                           }
```

Table 1.1.1. Character Set and Program Input Class

(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)
0	NUL	blind	32	SP	in text	64	@	graphic	96	'	graphic
1	SOH	illegal	33	!	graphic	65	A	basic	97	a	basic
2	STX	illegal	34	"	graphic	66	B	basic	98	b	basic
3	ETX	illegal	35	£	graphic	67	C	basic	99	c	basic
4	EOT	illegal	36	$	basic	68	D	basic	100	d	basic
5	ENQ	illegal	37	%	graphic	69	E	basic	101	e	basic
6	ACK	illegal	38	&	graphic	70	F	basic	102	f	basic
7	BEL	illegal	39	'	basic	71	G	basic	103	g	basic
8	BS	illegal	40	(	basic	72	H	basic	104	h	basic
9	HT	basic	41	)	basic	73	I	basic	105	i	basic
10	NL	basic	42	*	basic	74	J	basic	106	j	basic
11	VT	illegal	43	+	basic	75	K	basic	107	k	basic
12	FF	basic	44	,	basic	76	L	basic	108	l	basic
13	CR	blind	45	-	basic	77	M	basic	109	m	basic
14	SO	illegal	46	.	basic	78	N	basic	110	n	basic
15	SI	illegal	47	/	basic	79	O	basic	111	o	basic
16	DLE	illegal	48	0	basic	80	P	basic	112	p	basic
17	DC1	illegal	49	1	basic	81	Q	basic	113	q	basic
18	DC2	illegal	50	2	basic	82	R	basic	114	r	basic
19	DC3	illegal	51	3	basic	83	S	basic	115	s	basic
20	DC4	illegal	52	4	basic	84	T	basic	116	t	basic
21	NAK	illegal	53	5	basic	85	U	basic	117	u	basic
22	SYN	illegal	54	6	basic	86	V	basic	118	v	basic
23	ETB	illegal	55	7	basic	87	W	basic	119	w	basic
24	CAN	illegal	56	8	basic	88	X	basic	120	x	basic
25	EM	basic	57	9	basic	89	Y	basic	121	y	basic
26	SUB	illegal	58	:	graphic	90	Z	basic	122	z	basic
27	ESC	illegal	59	;	basic	91	Æ	basic	123	æ	basic
28	FS	illegal	60	<	graphic	92	Ø	basic	124	ø	basic
29	GS	illegal	61	=	basic	93	Å	basic	125	å	basic
30	RS	illegal	62	>	graphic	94	^	graphic	126	¯	graphic
31	US	illegal	63	?	graphic	95	_	in text	127	DEL	blind

Table 1.1.2. Treatment of Program Input Classes

class	input character in comment or text	label or statement field
basic	meaningful	meaningful
graphic	meaningful	illegal
in text	meaningful	skipped except after compound
blind illegal	no effect, not even counting as symbol of the line skipped, but causes warning	

RC FORTRAN includes the compounds listed in Table 1.1.3. A compound is an element made up of several characters. The compounds of the two rightmost columns in Table 1.1.3 must be followed by an in text' whenever the compound symbol is followed by a name or a digit. Accordingly, to avoid misinterpretation a name starting with the same letter as a compound should never have an <in text> separation at that place.

Table 1.1.3. Compound Symbols

Logical values	Declaratives	In statements
.TRUE. .FALSE.	PROGRAM SUBROUTINE FUNCTION	ASSIGN TO
Operators	ENTRY	GOTO GO TO
**	DATA COMMON EQUIVALENCE	CONTINUE CALL RETURN
.LT. .LE. .EQ. .NE. .GE. .GT.	INTEGER LONG REAL DOUBLE PRECISION COMPLEX LOGICAL	STOP IF DO
.NOT. .AND. .OR. .SHIFT.	DIMENSION ZONE EXTERNAL	END
	FORMAT FORMATO	READ WRITE

1.2 Line Format

A FORTRAN program is a sequence of symbols divided into lines by either the NL symbol or the FF symbol. When using punched cards a NL is generated at the end of each card. A program consists of program text and comments. In a line the symbols are counted starting with number 1, only blind symbols do not count.

Normally all symbols of a line are considered to be program text. By the compiler option cardmode.yes symbol 73 and onwards are treated as a comment. The first 6 symbols of a correct line are either (1) a control field, or (2) a label field. The examination has the steps:

1.1 If symbol 1 is a / (slash), the line will be treated as line starting with M (a message line).

The compiler will not read any further source-text lines.

1.2 If symbol 1 is one of the letters A, B, C, ..., K, L, M, or an asterisk (*), the line is normally a comment line.

By choice of parameters for the compiler the letters A, B, C, ..., K may be substituted by an <in text> symbol. The line is then a program line and can hold statements for testing.

The letter L may start a conditional listing, while the letter M leads messages that may be output during compilation.

1.3 If symbol 6 is neither <in text> nor 0, the line is a continuation line and symbols 1 through 5 must then be <in text>. The line is treated as a continuation of the program text of the previous line, i.e. cancelling the terminating effect of the foregoing NL, FF or comment.

2. If the 6 symbols are not a control field, symbols 1 through 5 must either contain a label or all be <in text>, and symbol 6 must be <in text> or 0. The label consists of digits only, leading spaces and zeroes are ignored. The line is an initial line.

The 7th and following symbols either including the 72nd symbol or the rest of the line make up the statement field. The statementfield may hold several statements seperated by the terminators ; (semicolon) or $ (dollar). If a line holds less than 6 symbols, <in text> filling is supposed and the statement field is empty.

When the statement field contains END, the line is an end line and the program unit terminates. Comments are allowed after the END.

The compiler continues reading of source-text lines until:

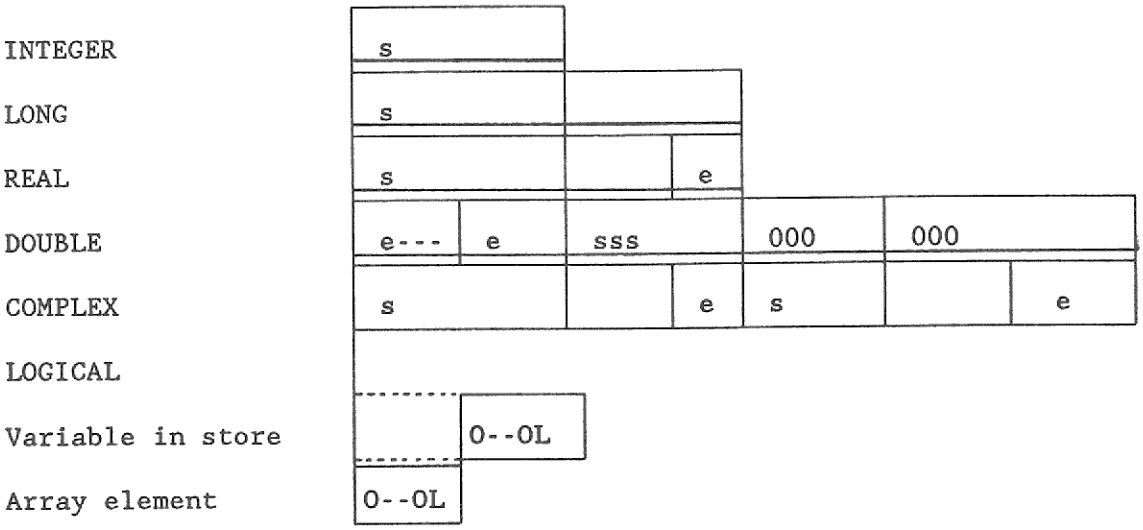
a. the source-list is exhausted (see appendix B, Call of compiler).

or b. a line is read, with a / (slash) as symbol 1 (see step 1.1. above).

1.3 Constants and the Internal Representation of Values

Integer, real, double precision, complex, and logical constants agree with the ISO FORTRAN syntax. RC FORTRAN includes long integer constants, short texts, and bitpatterns as described below, while the detailed syntax is given in Appendix B.

The internal representation of values, as constants, variables, and array elements will claim from 1 halfword to 8 halfwords (1 halfword = 12 bits) depending on the type:



s indicates the sign of integer, long or fixed-point parts.
e indicates the sign of exponent.
L indicates the determining bit of logical
0 indicates zeroed bits.

For each type details of the representation and the range of values are described. The representation of arithmetical values is binary using the 2-complement for negative values.

1.3.1 INTEGER

```
1 word = 24 bits.
```
Fixed-point representation:
```
  bit 0     : sign
  bit 23    : unit position
Range       : -2**23 =< integer =< 2**23-1
  2**23     = 8 388 608
Significant digits: 6-7
```

Integer constants in the program text exceeding the integer range are treated as long constants.

1.3.2 LONG

```
2 words = 48 bits.
```
Fixed-point representation:
```
  bit 0     : sign
  bit 47    : unit position
Range : -2**47 =< long =< 2**47-1
  2**47     = 140 737 488 355 328
Significant digits: 14-15
```

Long constants in the program text exceeding the long range cause error messages during compilation.

1.3.3 REAL

```
2 words = 48 bits.
```
Floating-point representation:
```
  bit 0     : sign of fixed point part
  bit 0-35  : fixed-point part
  bit 36    : sign of exponent
  bit 36-47: exponent
Range       : 2** (2**(-11)) =< abs(real)
              < 2**(2**11-1)
2**(2**11) = 2**2048 = appr. 10**616
Significant digits: 10-11
```

Real constants in the program text holding more than 11 significant digits are treated as double precision constants.

1.3.4 DOUBLE PRECISION

```
4 words = 96 bits.
Floating-point representation:
  bit 0-11    : extended sign of exponent
  bit 12      : sign of exponent
  bit 12-23   : exponent
  bit 24-26   : sign of fixed-point part
  bit 27-47   : fixed-point part 1, 21 bits
  bit 48-50   : 000
  bit 51-71   : fixed-point part 2, 21 bits
  bit 72-74   : 000
  bit 75-95   : fixed-point part 3, 21 bits.
Range         : as reals
Significant digits: approx. 19.
```

Double precision constants exceeding the precision mentioned cause error messages during compilation.

1.3.5 COMPLEX

```
4 words = 96 bits.
```
Floating-point representation of real and imaginary parts:
```
  bit 0-47    : real part, represented as real
```

```
bit 48-95   : imaginary part,
              represented as real.
```

Complex constants in the program text holding more than 11 significant digits in the real or the imaginary part cause error messages during compilation.

1.3.6 LOGICAL

```
1 halfword = 12 bits.
.TRUE.     000 000 000 001
.FALSE.    000 000 000 000
```

For simple variables a word is reserved but only the last halfword, i.e. bits 12-23 is used. Logical array elements occupy each one halfword only.

1.3.7 SHORT TEXT

```
2 words = 48 bits.
```

Internally short texts are handled as long integers inferring a maximum of 6 characters each of 8-bits to be stored in a long integer.

Short texts are allowed in expressions and as parameters. Text are written either (1) as a Hollerith text, i.e. nHxxx where n is integer and xxx is n non-blind symbols that are left-justified with SPACE filling when n < 6, or (2) as an apostrophed text i.e. 'xxx', in this case left-justified with NUL filling.

Texts with more than 6 characters appearing where only short texts are allowed cause error messages during compilation.

1.3.8 BITPATTERN

```
short 1 word = 24 bits.
long 2 words = 48 bits.
```

Bitpatterns are written as gBddd, where g is the number of bits in a group and may take the values 1, 2, 3, or 4, causing the digits ddd to be interpreted accordingly.

```
The binary digits are      : 01
The quaternary digits are  : 0123
The octal digits are       : 01234567
The sedecimal digits are   : 0123456789ABCDEF
```

Bipatterns are right-justified with zero filling for missing digits. If possible only 1 word is used. Short bitpatterns are treated internally as INTEGER constants while long bitpatterns are treated as LONG constants. Bitpatterns violating the syntax cause error messages during compilation.

1.3.9 Examples of Constants

In program	type
0	integer
-1	integer
-8388608	integer
23	integer
8388607	integer
-8388609	long
8388608	long
-12345678	long
12345678	long
0.0	real
-123.456	real
123.4e1	real, with value 1234
123.4e-1	real, with value 12.34
1234.56789012	double precision
123.4d1	double precision, with value 1234
123.4d-1	double precision, with value 12.34
(1.2, 0.0)	complex
5hhello	short text (as long constant)
'hello'	short text (as long constant)
4b123456	bitpattern (as integer constant with the value 1193046)
4b1234567	bitpattern (as long constant with the value 19088743)

1.4 Variables and Arrays

1.4.1 Names

A FORTRAN symbolic name is a string of letters and digits beginning with a letter. RC FORTRAN allows and distinguishes names of any length.

Names are used for simple variables, arrays, zones, commons, and program units. Two different entities must not have identical names within one program unit.

1.4.2 Variables

A variable is a datum defined by its name and type.

1.4.3 Arrays

An array is an ordered set of data. Arrays may have any number of dimensions and are stored by columns in ascending storage locations.

An array element is identified by following the array name by a parenthesized list of subscript expressions. The index check secures that elements refered to lie within the array.

A subscript expression is an arithmetical expression. There is no restriction on the complexity and type of the expression; if the type is not integer the conversions follows the rules of assignment to integer (see arithmetical assignment, Section 4.1). The number of subscript expressions must agree with the dimensionality of the array wherever array elements are referenced, except in EQUIVALENCE statements (see 2.3.2 step 6).

The element A(i1, ..., in) of an array declared A(c1, ..., cn) is identified by use of the successor function f = i1 + c1*(i2+ ... + c(n-1)*in)...). By inserting ij = 1 and ij = cj for j = 1, ..., n the two limiting numbers f1 = f(ij = 1) and fc = f(ij = cj) define the first and the last element of the array. The index check is performed on the value of successor function, and there is no check that 1 =< ij =< cj for j = 1, ..., n.

2. Declarations

2.1 Type Statement

The type and dimensioning associated with a name may be controlled by type statements and/or DIMENSION statements.

2.1.1 Form

```
                {                  } 1 *
<type> [<name> {(<fixed bounds>)}  ]
                {                  } 0 1

<type> :: = { LOGICAL          }
            { INTEGER          }
            { REAL             }
            { LONG             }
            { COMPLEX          }
            { DOUBLE PRESICION }

                                 *
<fixed bounds> :: = [<integer>]
                                 1
```

Example 2.1.1

```
real ra (3, 7, 4, 2)
```

declares an array of type real with 4 dimensions. The first dimension ranges from 1 to 3, the second from 1 to 7, etc.

Example 2.1.2

```
integer x, ia(2, 3, 4), y
```

declares two simple integers x,y and an integer array ia.

2.1.2 Rules

1. The type statement associates the specified type with all names occuring in the list of names. If fixed bounds are specified the name is associated with an array with dimensions and range corresponding to the bound list.

2. The type and dimensions of the name may be specified only once within a program unit.

3. If a name is not explicitly type declared, type integer is implicitly assumed if the name starts with i, j, k, l, m, n, otherwise type real is assumed.

4. Any number of dimensions are allowed (ISO standard: 3).

5. Arrays are stored column by column - in consecutive storage locations. E.g., the array ia of Example 2.1.2 will occupy 24 storage locations as shown:

```
ia(1, 1, 1)
ia(2, 1, 1)
ia(1, 2, 1)
ia(2, 2, 1)
ia(1, 3, 1)
ia(2, 3, 1)
ia(1, 1, 2)
ia(2, 1, 2)
etc.
```

2.1.3 Notes

1. Variable bounds may be used, when an array appears as a formal parameter. See Section 6.1.5.

2. A few names of intrinsic functions and basic external functions are implicitly of type complex or double precision (see Appendix E).

2.2 DIMENSION Statement

2.2.1 Form

```
                                          *
DIMENSION [<name> (<fixed bounds>)]
                                          1

                                        *
<fixed bounds> :: = ( <integer> )
                                        1
```

2.2.2 Rules

The DIMENSION statement may be used for stating separately the dimensionality of one or more names. See Section 2.1.2 above for further rules.

2.3 EQUIVALENCE Statement

The order in which names occur in type statements and/or DIMENSION statement does not control the mutual placement of the variables in core, i.e. names declaring after each other in a type statement are not necessarily placed in the same order in core.

The storage allocation may be controlled by the EQUIVALENCE statement or by the COMMON statement (see Section 6.2).

2.3.1 Form

```
                                                          *
            {  { <variable name>                        }*}
EQUIVALENCE { ({                                        })}
            {  { <array name>(<constant subscripts>)}2}
                                                          1

                                        *
<constant subscripts> :: ={<integer>}
                                        1
```

Example 2.3.1

```
real a(5), b(3, 4), x
integer q(6)
equivalence (a(2), b(2, 3) x, q(2))
```

The variables a(2), b(2, 3) and x now refer to the same four halfwords. The integer variable q(2) corresponds to the two first of these. Note that q(3) will correspond to the two last halfwords, a(3) and b(3, 3) are equivalent, etc.

Example 2.3.2

```
long p
integer ia(2), i1, i2
equivalence (p, ia(1), i1), (ia(2), i2)
```

i1 and ia(1) now correspond to the first half of the long integer p while i2 and ia(2) correspond to the last half.

2.3.2 Rules

The EQUIVALENCE statement permits the use of different names in connection with the same element(s). The variables referenced within an equivalence group describe the same data element with the following clarifications and restrictions:

1. The variables referenced may be of different type and kind.

2. If the variables are of different size the shorter one correspond to the first halfword(s) of the longer one.

3. Within a group of equivalenced variables at most one variable may directly or through equivalence be a common variable (Section 6.2).

4. The equivalences must not be contradictory, e.g., by making elements of the same array equivalent.

5. Only local or COMMON variables may occur in EQUIVALENCE statements.

6. The number of subscripts in a subscripted variable must

   a. be exactly one
   or b. agree with the declaration of the array.

   An instance of form a, like: a(7) is the same as a(7,1,...,1) of form b.

2.3.3 Notes

1. A special form of equivalence connected with zones is described in Section 5.5.7.

3. Expressions

3.1 Arrays and Subscripts

A subscript expression is an arithmetical expression with no restriction on the complexity and type of the expression; if the type is not integer the conversion follows the rules of assignment to integer (see Arithmetical assignment, Section 4.1).

The number of subscript expressions must agree with the dimensionality of the array wherever arrays elements are referenced, except in EQUIVALENCE statements (see 2.3.2 step 6).

3.2 Precedence Rules

The syntax (Appendix A) defines 3 different sorts of expressions, namely arithmetical, masking, and logical. The precedence of operators is not included in the syntax. The following rules hold for the sequence of operations in any expression:

a. In an expression the precedence of the operators is

```
1st: ** .SHIFT.
2nd: * /
3rd: + -
4th: .LT. .LE. .EQ. .NE. .GE. .GT.
5th: .NOT.
6th: .AND.
7th: .OR.
```

b. Operators of the same precedence level are executed from left to right.

c. Expressions enclosed in parantheses and function calls are evaluated by themselves and the value is used in subsequent calculations.

Monadic operators operate on one operand, e.g. -A. Dyadic operators combine two operands, e.g. A/B.

Operands combined through dyadic operators may be of different types except in shift and logical operations. This often implies that one operand has to be converted before the operation.

Operations that are not mathematically defined or violate value range may cause alarm.

If the execution of procedure calls refer to variables in the same statement, side effects may be triggered. This is prohibited in the ISO standard, but no check is performed.

3.3 Arithmetical and Masking Expressions

The monadic operators + - may be considered to operate as if a zero precedes the same dyadic operator; the result is of the same type as the original operand.

The dyadic operators + - * / ** combine operands of arithmetical type integer, long, real, double precision, and complex.

The types may be mixed, the type of the result is given in the table below.

Only the power operator ** is restricted:

- the exponent must not be of type long, double precision or complex.
- the radicand must not be of type double precision or complex.

Table 3.3.1. Resulting types of arithmetical operations + - * / **

a <op> b a \ b	integer	long (not**)	real	double (not**)	complex (not**)
integer	integer	long	real	double	complex
long	long	long	real	double	complex
real	real	real	real	double	complex
double (not**)	double	double	double	double	complex
complex (not**)	complex	complex	complex	complex	complex

Real overflow and underflow. The reaction on real overflow is controlled by the external integer variable OVERFLOWS in the FORTRAN system as follows:

OVERFLOWS < 0	The run is terminated at overflow.
OVERFLOWS >= 0	The value of OVERFLOWS is increased by one. The result of the actual arithmetical operation is 0.

When the program execution starts, the value of OVERFLOWS is -1.

Real underflow is controlled analogously by the UNDERFLOWS.

When using the mask operators .NOT., .AND., .OR., and .SHIFT. the already shown precedence of operators holds. Mixing of types integer, real, and long is permitted as shown in the syntax and further explained below and in table 3.3.2, other types are prohibited.

.NOT. The operator is monadic, all bits of the operand are complemented. The result is either short or long depending on the operand.

.AND. .OR. The operators are dyadic and work as the same logical operators on all bits of the operands. If short and long operands are mixed the short operand is combined with the last 24 bits of the long operand, i.e. the result is short. The type is long when a real operand is masked.

.SHIFT. The operator is dyadic. The result of A .SHIFT. B is either short or long agreeing with A, and the operation is to shift the A-operand B bits to the left (i.e. to the right when B<0) with zeroes entering from either side. A may be of type integer, real or long, while B must be an integer operand.

Arithmetical expressions resulting in type integer, real, or long (table 3.3.1) may appear as operands of .NOT., .AND., and .OR.

Table 3.3.2. Resulting types of masking operations.

	.NOT. A	A .AND. B A .OR. B			A .SHIFT. B
A		B integer	B long	B real	B integer
integer	integer	integer	integer	integer	integer
long	long	integer	long	long	long
real	real	integer	long	real	real

4. Executable Statements

4.1 Arithmetical and Logical Assignment

4.1.1 Arithmetical Assignment

In arithmetical assignments v=e the value of the arithmetical expression e is evaluated first resulting in a value of arithmetical type. The element v must also be of arithmetical type, and if it corresponds to the type of e the assignment is made, if not an implied conversion precedes the assignment as shown in table 4.1.1. Some of these will be two-stage conversions.

Table 4.1.1. Arithmetical Assignment.

v=e v \ e	integer	long	real	double	complex
integer	=2=	L.CONV.I =2= x	R.TRUNC.I =2= xy	D.TRUNC.R R.TRUNC.I =2= xy	C.CONV.R R.TRUNC.I =2= xy
long	I.CONV.L =4=	=4=	R.TRUNC.L =4= xy	D.TRUNC.L =4= xy	C.CONV.R R.TRUNC.L =4= xy
real	I.FLOAT.R =4=	L.FLOAT.R =4= y	=4=	D.TRUNC.R =4= y	C.CONV.R =4=
double	I.FLOAT.R R.CONV.D =8=	L.FLOAT.D =8=	R.CONV.D =8=	=8=	C.CONV.R R.CONV.D =8=
complex	I.FLOAT.R R.CONV.C =8=	L.FLOAT.R R.CONV.C =8=	R.CONV.C =8=	D.TRUNC.R R.CONV.C =8= y	=8=

Explanation:

x	Integer overflow may occur.
y	Precision may be lost.
=n=	Assign means transmit the resulting value, without change, to entity. n 12-bit halfwords are transmitted.
e.CONV.v	Convert the resulting value of e-type to representation of the v-type.
e.TRUNC.v	Truncate means preserve as much precision of the resulting value of e-type as can be contained in datum of receiving v-type. See below.
e.FLOAT.v	Float means change fixed-point value of e-type to floating-point representation of v-type

Truncation of Real Values:

The truncations R.TRUNC.I and R.TRUNC.L are controlled by the compiler-option: trunc.yes or trunc.no (see Appendix B).

trunc.yes:	if |R| < 1, the result is 0; if |R| >= 1, the result is the integer (respectively long) whose magnitude does not exceed the magnitude of R and whose sign is the same as the sign of R.
trunc.no:	The result is the nearest integer (respectively long).

The Conversions:

L.CONV.I	The first word of L is excluded, overflow when the bits 0-23 of L are not equal to bit 24 of L.
I.CONV.L	The sign of I is extended 24 bits.

Multiple arithmetical assignments are allowed when all elements assigned to are of the same arithmetical type.

4.1.2 Logical Expressions and Assignments

A relation operator compares the values of two arithmetical expressions. When these values are of different type an implied conversion takes place. Logical expressions agree with the ISO standard. A part of an expression need only be evaluated when necessary to establish the value of the total expression. This will be used to optimise the calculation of logical expressions without rearranging the terms in the expression. Logical assignment v=e demands both v and e to be of type logical. Multiple logical assignments are allowed when all elements assigned to are of type logical.

4.2 GOTO statements

4.2.1 Simple GOTO

Form: `GO TO <statement label>`

The statement causes unconditional transfer of control to the statement identified by the statement label. Both GO TO and GOTO are permitted.

4.2.2 Assigned GOTO

Form: `GO TO <label name>, (<statement label>)` with `*` above the opening parenthesis and `1` below it

Rules:

1. The statement acts as a many branch GOTO statement. Before executing an assigned GOTO the referenced label variable must be assigned (see below). By the assigned GOTO statement control is transferred to the statement identified by the assigned label value. If the label variable is unassigned the execution will cause a run time alarm.

2. A label name must explicitly or implicitly be of type integer and it must not be COMMON.

3. When an integer name is referenced in an ASSIGN statement or an assigned GOTO statement it is considered a label name. It must not be used as an arithmetical operand within the same program unit. Violating this rule will cause a type error at compilation time.

4. The comma and the label list is optional. No check is performed based on the label list.

4.2.3 ASSIGN Statement

Form: `ASSIGN <statement label> TO <label name>`

The statement assigns the specified statement label to the label variable. See 4.2.2, rule 2 and 3.

Example 4.2.1

```
assign 1017 to lab
...
GOTO lab, (1010, 1012, 1017)
```

After the execution of the shown ASSIGN statement, the assigned GOTO statement will transfer control to the statement with label 1017.

4.2.4 Computed GOTO

Form: `GO TO ([<statement label>]*1 ), <integer expression>`

The statement works as follows: the value of the integer expression is evaluated giving the result, r. Control transferred to the statement identified by the r'th statement label in the label list. If r is less than 1 the first label is selected, if r is greater than the number of labels in the list the last label is selected.

4.2.5 Examples

Example 4.2.2

The following GOTO statements all effect a transfer to the statement with label 100:

```
assign 100 to lab
GOTO lab
GOTO 100
inx = 3
GOTO (300, 200, 100, 50), inx
```

Example 4.2.3 Administration of Actions

Processing of records may be implemented as a set of actions, numbered from 1 to k, and an action table, integer actab(n), where actab(i) specifies the actions to be executed for records of type i.

If action numbers range from 1 to 15, the action numbers may be packed in 4 bit groups, so that up to 6 actions may be specified for each record. The administration of the action execution may then look like:

```
      iac=actab(record type)
c extract action number
50    action=iac .and. 1b1111
c prepare extraction of next action number
      iac=iac .shift. (-4)
c select action, goto 9999 if no more
c actions for this record
      goto(9999, 100, 200 ..., 1500),action + 1
      ...
c action 1 ...
100   num=record(numbinx)
      ...
      goto 50
c action 2 ...
```

```
200   if (type .gt. max) ...
      ...
      goto 50
      etc.
```

4.3 IF Statement

4.3.1 Logical IF

Form: `IF (<logical expression>) <statement>`

Rules

1. The statement works as follows: the logical expression is evaluated. If the value is true, the conditional statement is executed, otherwise the statement is bypassed.

2. The statement must not be a DO statement.

4.3.2 Arithmetical IF

Form: `IF (<arith expression>)[<statement label>`$_3$`,`$_3$`]`

Rules

1. The statement works as follows: the value of the arithmetical expression is evaluated. If the resulting value is negative the first label second label is chosen, else the third label is is chosen, if the value is equal to zero the chosen. Control is then transferred to the statement identified by the chosen label.

2. The expression must not be of type complex.

Example 4.3.1

The algol statement

```
goto if x<0 then L100 else
  if x<y then L10 else
  if x=y then L20 else L30
```

may be written in fortran as

```
if (x .lt. 0) goto 100
if (x-y) 10,20,30
```

4.4 DO Loops

4.4.1 Form

```
                                                    3
DO<statement label><integer name> =[<expression>]
                                                    2
```

4.4.2 Execution of a DO Loop

The statement causes a sequence of statements starting with the DO statement, up to and including a terminal statement to be executed repeatedly. The terminal statement is determined by the specified label. The number of repetitions is controlled by the three expressions. The variable specified by the integer name is called the control variable, the first expression determines the starting value, the second expression determines the limit and the third expression determines the step. If the third expression is omitted, the step 1 is used. The execution of a DO loop may be described as follows:

1. Evaluate values of step and limit: The second and third expressions are evaluated and if necessary truncated to integers.

2. Compute value of control variable: At first entry the control variable is assigned the starting value, the following times the current value of step is added to the control variable.

3. Conditional execution of DO range. The statements within the range of the DO loop are executed if

   ```
   (limit-control variable)*step .ge. 0
   ```

After execution of the DO range the procedure is repeated from 1. above. If the condition is not fulfilled the DO loop is exhausted and execution proceeds with the statement after the terminal statement of the DO loop.

4.4.3 Rules

1. The control variable must be of type integer.

2. The expressions must be of arithmetical type. After each evaluation the value is truncated to an integer before it is used in the DO control, i.e. addition to the control variable and test is performed in integer mode.

3. The terminal statement must be found after its corresponding DO statement within the program text.

4. A DO loop may contain other DO loops but the inner loop must be completely contained within the surrounding DO loop.

5. A nest of DO loops may have the same terminal statement, the terminal statement is then considered as part of the innermost DO loop (see Example 4.4.1).

6. When a DO loop is exhausted the control variable retains its last assigned value (see 4.4.2 step 2).

Example 4.4.1

```
        N = 0
        DO 100 I = 1,10
        J = I
        DO 100 K = 1,5
        L = K
100     N = N + 1
101     CONTINUE
```

After execution of these statements and at the execution of the CONTINUE statement, I = 11, J = 10, K = 6, L = 5, and N = 50.

Example 4.4.2

```
        N = 0
        DO 100 I = 1,10
        J = I
        DO 100 K = 5,1
        L = K
200     N = N + 1
201     CONTINUE
```

After execution of these statements and at the execution of the CONTINUE statement I = 11, J = 10, K = 5, and N = 0. L is not defined by these statements.

4.5 CONTINUE Statement

The CONTINUE statement consists solely of the word CONTINUE and serves as a dummy statement to which a label may be attached. The statement has no effect.

4.6 STOP Statement

Form: `STOP [<integer>]` (1 or 0)

The execution of a STOP statement terminates program execution in a way similar to passing the END statement of the main program. All zones are released, the word 'end' is written on current output.

5. Input/Output

5.1 Zones And Logical Units

5.1.1 Introduction

The input/output system of RC FORTRAN utilizes a concept called a zone.

A zone is a compound entity consisting of three distinct parts:

1) zone descriptor
2) one or more share descriptions (hereafter just called shares)
3) buffer area

A zone may be assigned a unitnumber whereby the zone is accessible as a traditional logical unit.

The READ/WRITE statements may specify either the zone name or the unitnumber when indicating the logical unit to be worked upon.

The extended input/output system of RC FORTRAN, like INREC/OUTREC etc, providing tools for record handling, may only specify the zone name.

5.1.2 Zone Descriptor

A zone descriptor consists of the following set of quantities which specify a process or a document connected to the zone and the state of this process:

Process name:
A text string specifying the name of a process or a document.

Mode and kind:
An integer specifying mode and kind for a document (see 5.2.3 open).

Logical position:
A set of integers specifying the current position of a document.

Give up:
An integer specifying the conditions under which <block proc> is to be called.

State:
An integer specifying the latest operation on the zone.

Line change mode:
An integer specifying the proper line change action for WRITE (in ALGOL this field is called: Free parameter).

Record:
Two integers specifying the part of the buffer area nominated as the zone record.

Used share:
An integer specifying a share descriptor within the zone.

Last byte:
An integer specifying the end of a physical block on a document.

Block procedure:
The procedure <block proc> (see 5.1.5).

The normal use of these quantities is explained in details in chapter 5.2. Further details may be found in ref. 2, GETZONE.

5.1.3 Share Descriptor

Each zone contains the number of share descriptors given by <shares> in the parenthesis following the zone identifier (see 5.1.5). The share descriptors are numbered 1, 2, ..., <shares>.

A share descriptor consists of a set of quantities which describes an external activity sharing a part of the buffer area with the running program. An activity may be a parallel process transferring data between a document and the buffer area under supervisory control of the FORTRAN program. Further details may be found in ref. 2.

The set of quantities forming one share descriptor is:

Share state:
An integer describing the kind of activity going on in the shared area.

Shared area:
Two integers specifying the part of the buffer area shared with another process by means of the share descriptor.

Operation:
Specifies the latest operation performed by means of the share descriptor.

5.1.4 Buffer

Area The buffer area is used for containing the data for input/output.

When using the record procedures, like INREC/OUTREC etc., a part of this zone buffer is made available for the program as the socalled zone record.

The zone record is a real array of one dimension with the same name as the zone itself.

The zone elements are numbered 1, 2, ..., <record length>.

5.1.5 Declaration of Zones

5.1.5.1 Simple Zones

A simple zone may be declared as follows:

```
ZONE <zone name>(<bufsize>,<shares>,<blproc name>)
```

`bufsize (integer)`
The size of the buffer area expressed in double words.

`shares (integer)`
The number of shares.

`blproc name (procedure)`
The name of the attached block procedure.

The buffer area.
This parameter defines the size of the total buffer area attached to the zone. The area is measured in double words.

The parameter must be a literal integer, not an integer.

The number of shares.
A general multibuffer administration is used by the input/output system. The number of shares defines the number of buffers into which the total buffer area is divided. Normally the user will choose the values 1 or 2 for single or double buffered input/output. The size of a share should normally match the block size of the connected external device. If the share does not match the block size, part of the block may be lost when transferred. This is considered a hard error (see Section 5.3). A longer share is just waste of core.

The block procedure.
The block procedure is called by the input/output system when hard errors occur during the input/output operations. Normally the use of a standard block procedure is recommended (see Section 5.3.4 and Ref. 3), but the user may design individual block procedures according to the detailed conventions found in Section 5.3.3 and Ref. 2. The use of a block procedure in a zone declaration works as an EXTERNAL declaration of the name.

5.1.5.2 Zone Arrays

An array af zones is declared by combining the zone declaration with a DIMENSION statement. Only one dimension is allowed.

The parameters specified in the zone declaration are common to all zones in the zone array, but each zone has its own description and its own buffer area of the specified size.

A single zone from a zone array is referred by writing the name of the zone array followed by one subscript.

5.1.6 The Standard Zones IN and OUT

The standard zones IN and OUT are preopened zones which may be used on character level (i.e. by formatted READ/WRITE).

The zone IN is preassigned the unitnumber 5, while OUT has been preassigned the unitnumber 6.

If the names of these standard zones are to be used in a program unit they must be declared as external zones as follows:

```
External in, out; zone in, out
```

If these standard zones are used solely by the unitnumbers they need not be declared as external zones.

Example 5.1.1

The declaration

```
zone zo(256, 2, stderror)
```

declares a zone named zo, with a buffer of 256 double words. The buffer is divided in two shares and the standard block procedure, stderror, is used.The name stderror is automatically declared external.

Example 5.1.2

The declarations

```
zone (10, 1, stderror); dimension za (3)
```

declares a zone array consisting of three zones, each with a buffer of 10 double words in a single share.

5.1.7 Multishare Input/Output

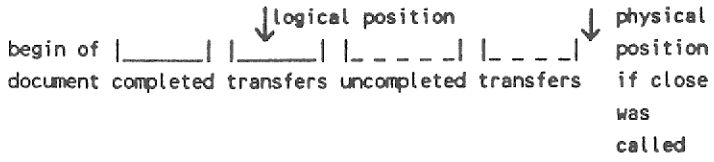
The amount of information transferred to or from a share in one operation is called a block. On a magnetic tape a block is a physical block or a tape mark. On a backing storage area a block is one or more

segments. On a paper tape reader a block is usually one share of characters.

Input

During input from a document via a zone with sh shares the system uses one of the shares for unpacking of information and the remaining sh-1 shares for uncompleted input of later blocks. The following picture shows the state of the blocks of the document.

Input, sh = 3

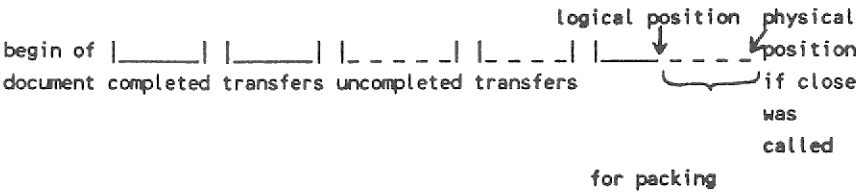


Note that when the document is closed the physical position of the document is far ahead of the logical position. This is particular important at the end of magnetic tapes where the 'stippled' blocks may be absent and the tape then comes off the reel.

Output

During output to a document via a zone with sh shares one share is used for packing of information, and 0 to sh-1 of the remaining shares are used for uncompleted output of previous blocks. The following picture shows the state of the blocks in the output stream.

Output, sh = 3



Note that when the document is closed the physical position is just after the block corresponding to the logical position.

Swoprec

The procedure swoprec utilizes the shares as follows: One share is used for packing and unpacking of information. If sh > 1 another share is used for uncompleted output. Remaining shares are used for uncompleted input of future blocks.

Choice of sh

The advantage of the multishare input/output is that differences in speed between the program and the device may be smoothed to any degree. The most frequent choise is between single or double buffer input/output. The following rule of thumb may help you to choose in cases where you scan a document sequentially:

th = time spent by the program with handling of the information in a block.
td = time spent by the device with transfer of a block. td + th is the total time in single buffer mode (sh = 1).
max. (td, th) is the total time in double buffer mode (sh = 2).

If th varies from block to block the situation is more complicated and sh > 2 may pay.

The following rule of thumb concerns the sequential use of swoprec:

th + 2*td is the total time per block with sh = 1.
max (th, td) + td is the total time per block with sh = 2.
max (th, 2*td) is the total time per block with sh = 3.

You should always use single buffering on printer, plotter, and punch, except when you know for sure that your job is not stopped and started by the operating system. The reason is that an output operation is terminated halfway when the job is stopped, but with sh > 1 the next output operation is started before the first is checked and output again.

You should always use single buffering for typewriter output because the operator at any moment may stop the output operation to send a console message to send a console message.

Message Buffers Occupied

Input/output by means of sh shares occupies permanently sh-1 of the message buffers available for the job (see ref. 10). From the moment SETPOSITION has been called for a magnetic tape and until the first input/output operation is performed one message buffer is occupied (even when sh = 1).

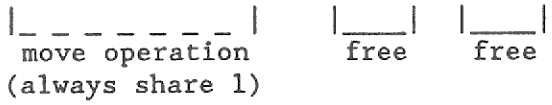
5.1.8 Algorithms for Multishare Input/Output

You must know about these algorithms if you want to interfere with the system in the block procedure of the zone. More details about the variables in a zone may be found in Ref. 2. Ref. 8 while Ref. 10 explain the rules behind the communication with devices.

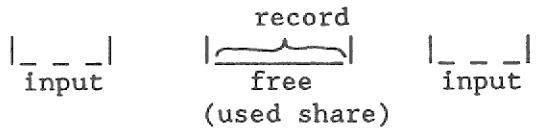
The algorithms below are written in algol, and sh denotes the number of shares in the zone.

Snapshots of Shares in Typical Situations (sh = 3)

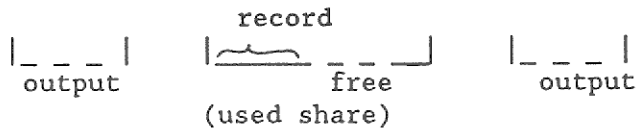
Just after setposition on a magnetic tape:



After inrec:



After several outrecs:



Change of Block at Input

```
rep: if share_state (used_share) = free then
     begin start transfer (input);
       used_share:= used_share mod sh + 1;
       goto rep
     end;
comment now all shares are busy with transfers
     except after a positioning;
wait_transfer (used share); comment share state
     becomes free. The operation checked might
     be a positioning operation;
last byte := top transferred (used share) -1;
comment now the share contains data from record
     base to last byte;
```

Change of Blocks at Output

```
if share_state (used share) <> free then
begin wait_transfer (used share);
    comment a positioning operation might be
    uncompleted;
end;
start transfer (output);

used_share := used_share mod sh + 1;
comment one or more shares behind_used share are
    busy with transfers;
wait_transfer (used_share);
comment share state becomes free and the share may
    be filled from record base to last byte;
```

Start Transfer (Operation)

This procedure works only on used share. It sets a part of the message and sends it:

```
first absolute address of block := abs address of
    first shared;
segment number of message := segment count;
update segment count for next transfer;
operation in message := operation;
comment the mode is left unchanged;
send message;
share state := uncompleted transfer;
```

Wait Transfer

This procedure waits for the answer from a transfer or tape positioning, checks it, and performs the standard error actions (error recovery). Finally, it may call the block procedure of the zone. In details this works as follows:

```
record_base:= abs address of first_shared(used_share) -1;
last_byte:= abs address of last_shared(used_shared) +1;
record_length:= last_byte - record_base;
st:= share_state(used share);
if st <> running child process then
   share_state(used share):= free;
if st <> uncompleted transfer then goto return:

wait_answer(st);
if kind = magnetic tape then
begin if some words were transferred then
      block_count:= block_count + 1;
if tape mark sensed and operation is input or
      output mark then
begin file_count:= file_count +1; block_count:= 0
end
end;
compute logical status word; comment the logical
      status word is 24 bits describing the error
      conditions of the transfers (see 5.3);
top_transferred(used_share):= if operation = io
   then 1 + address_of_last_byte_transferred
   else first_shared(used_share);
users_bits := common ones in logical status and
   give up mask;
remaining_bits := logical_status - users_bits;
perform standard error actions for all ones in
   remaining bits (see 5.3).
if a hard error is detected then
   logical_status := logical_status + 1;
if hard error is detected or usersbits <> 0 then
begin
   b:= toptransferred(usedshare) - 1 - record base;
   let record describe the entire shared area from
   first shared to last shared;
```

```
    save:= zone state;
    if operation = input and tapemark and b = 0 then
       b:= 2
    blockproc (z, logical_status, b);
    zone_state:= save;
    if b < 0 or b + record base > lastbyte then
       index_alarm;
    top transferred(used_share):= b + 1 + record_base;
  end;
return:
```

5.2 Documents, Basic Input/Output

5.2.1 Documents

The various external media which may be used in RC FORTRAN are called documents. A document may be a deck of cards, a roll of paper tape, a reel of magnetic tape, etc. Documents are identified by a document name, which is a string of up to 11 small ISOletters or digits starting with a letter and terminated by the NUL-character.

A document may be thought of as a string of information, either a string of 8-bit characters or a string of binary words. The string is on some documents broken into physical blocks (e.g., on magnetic tapes and backing storage areas). The procedures for input/output on character level and record level keep track of the current logical position of the document. The logical position points to the boundary between two characters or two elements of the document. During normal sequential use of the document, the logical position moves along the document correspon ding to the calls of the input/output procedures.

For documents consisting of physical blocks, the logical position is given by a position within the physical block, plus a block number, plus (for magnetic tapes) a file number. Note that the block number is ambiguous in the case where the logical position points to the boundary between two physical blocks. This ambiguity is resolved explicitly in the description on the individual procedures: The term 'the logical position is just before a certain item' implies that the block number is the block number of that item.

The following sections give a survey of some documents and the way they transfer information to and from the zone buffer. The rules for protection of documents and further details are found in Ref. 1.

5.2.1.1 Backing Storage

The backing storage may consist of drums and/or discs. You have no direct access to the entire backing storage, but only to documents which are backing storage areas consisting of a number of segments. Each segment contains 512 halfwords, equivalent to 128 real variables. The segments are numbered 0, 1, 2, ... within the area, and the block numbers mentioned above are exactly these segment numbers. File numbers are irrelevant.

One or more segments may be transferred directly as bit patterns to or from the core store in one operation. The number of segments transferred is the maximum number that fits into the share used.

Details about the various types of backing storage devices may be found in Ref. 8.

5.2.1.2 Typewriter

A typewriter may be used both for input and output. The sequence of characters input forms one document (infinitely long), and the sequence of characters output forms another document. File number and block number are irrelevant on a typewriter.

One input operation transfers one line of characters (including the terminating New Line character) to the share. If the share is too short, less than a line is transferred, but that is an abnormal situation. The characters are packed in ISO 7-bit form with 3 characters to one word, and the last word is filled up with NULs. One output operation transfers characters packed in the same form to the typewriter. Several lines may be output by one operation.

5.2.1.3 Paper Tape Reader

A document consists of one roll of paper tape. It may be read in various modes: with even parity, with odd parity, without parity, or with transformation from flexowriter code to ISO code. File number and block number are irrelevant for a paper tape.

One input operation will usually fill the share with characters packed 3 per word, but fewer characters may also be transferred, for instance at the tape end. In such cases, the last word is filled up with NUL characters. The characters are not necessarily ISO characters, that depends on the meaning you assign to them.

5.2.1.4 Paper Tape Punch

A document is infinitely long, even when the operator divides the output into more paper tapes. A paper tape may be punched in various modes: with even parity, with odd parity, without parity, or with transformation from ISO code to flexowriter code. File number and block number are irrelevant for tape punch.

One output operation may punch any number of characters packed 3 per word. In all modes, except the mode without parity, only the last 7 bits of the characters are output and extended with a parity bit.

5.2.1.5 Line Printer

A document is infinitely long. File number and block number are irrelevant on a printer.

One output operation may print any number of characters packed 3 per word. Several lines may be output by one operation. The characters must be in ISO 7-bit code.

5.2.1.6 Card Reader

A document is one deck of cards. It may be read in various modes: in binary, in decimal, and with conversion from Hollerith to ISO. File number and block number are irrelevant on a card reader.

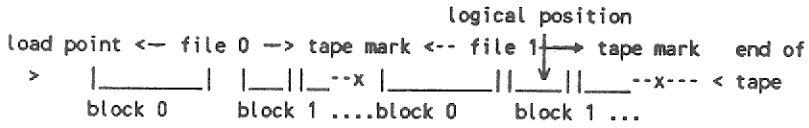
One input operation will usually fill the share, but fewer cards may also be read, for instance at the end of the deck. One column contains always one character. The characters are packed 2 per word in binary mode, and 3 per word in the other modes. In the latter case, a card is stored as 81 characters, where the 81st is a New Line character generated by the monitor.

5.2.1.7 Magnetic Tape

A document is one reel of tape. It consists of a sequence of files seperated by a single file mark. Each file consists of physical blocks with possibly variable lengths. The blocks may be input or output in even or odd parity. The files and blocks are numbered 0, 1, 2, ... as shown in the figure.

One operation transfers one physical block to or from a share. If an input block is longer than the share, only the first part of the block is transferred.

A magnetic tape document:



Two kinds of tape stations exist: 7-track stations where a block consists of a sequence of 6-bit bytes; one word of the share is here transferred as four 6-bit bytes. 9-track stations where a block consists of a sequence of 8-bit bytes; one word of the share is here transferred as three 8-bit bytes. The difference causes no trouble as long as the tapes are written and read on RC4000, RC6000, or RC8000. But if you try to move a 7-track tape to another computer (or to an off-line converter), you should remember that READ and WRITE work with 8-bit characters.

The share length used for output to a magnetic tape determines the physical block length. Because the blocks are seperated by a block gap of 3/4 inch, the share length has influence on the amount of information the tape can hold and also on the maximum transfer speed. With density of 556 bpi (bytes per inch), a share length of 60 elements will generate blocks of about 3/4 inch (more or less depending on the kind of the station). In this case half of the tape is used for blocks and half for block gaps. The data are transferred with 0.38 times the maximum tape speed, because block gaps take 1.6 the time of blocks of the same length. If you

use a share length of 600 elements, 10/11 of the tape is used for data and the transfer rate is 0.86 of the maximum.

5.2.1.8 Internal Process

An internal process (another program executed at the same time as your job) may read or write a document. The process may be designed to react according to the rules given for the document. After calling OPEN with the name of the internal process and the kind corresponding to the document the process may then be used exactly as the document.

The internal process may also handle the information in its own way, and then no general rules can be given, but usually, the end of the document is signalled as explained in Section 5.3.

5.2.1.9 Devices without Documents

Some peripheral devices, for instance the clock do not scan documents, and they cannot be handled by the high level zone procedures. However, the primitive input/output level may handle such devices too.

5.2.2 Principles of Communication

The following principles generally apply for the use of a document:

a. The document must be loaded on a specific device and connected to a zone. This is done by calling the standard procedure OPEN.

b. Input/output operations and maneuvering is thereafter performed by calling i/o-procedures referring exclusively to the zone.

c. The document may finally be released from the zone andpossibly unloaded by use of the procedure CLOSE.

NOTE: if the document is a magnetic tape the call of OPEN must be followed by a call of the procedure SETPOSITION before input/output operations are legal.

The main features of the input/output system may be illustrated by the example below and the comments following it. In the example file 1 from a magnetic tape is read by unformatted READ and printed on a line printer by formatted WRITE.

Example 5.2.1. Printing of a Magnetic Tape File

```
line
no.
 1          program taprint
 2          common/eofcom/ eof, prnam
 3          data eof, prnam/ 0, 'printer'/
 4          logical setposition; long prnam(2); integer eof
```

```
 5            zone tapzon(256,2,blp)
 6            zone przon(40,2,stderror)

 7            call open (tapzon,18, 't5011', 1 .shift. 16)
 8            call open (przon,14,prnam(1), 0)
 9            call setposition (tapzon,1,0)
10       10   read(tapzon) x,y,z,v
11            if(eof) 100,20,100
12       20   write(przon, 1000) x,y,z,v
13            goto 10
14     1000   format(4h x= ,f8.2, e.t.c.
15      100   call close(tapzon, .true.)
16            call close(przon, .true.)
17            end

18            function blp(z,s,b)
19            zone z; integer s,b,eof; long prnam(2)
20            common/eofcom/eof, prnam
21            if( s .and. 1) 10,20,10
22       10   call stderror(z,s,b)
23       20   if (b) 40, 40, 30
24       30   eof = 1; b =16
25       40   end
```

Comments to example 5.2.1:

line 3 — eof is initiated to zero, the array prnam is initiated to the document name, printer, with trailing NUL characters (see Section 6.2.4 about DATA and long texts).

line 5 — The zone for tape reading is declared. The zone buffer is 256 reals in two shares corresponding to a block length of 128 reals. The function blp is to be called, when hard errors occur on the magnetic tape at the end of file.~

Details about the block procedure are given in Section 5.3.3.

line 6 — The zone for the line printer is declared. The share size here is 20 double words corresponding to 120 8-bit characters. With this zone the standard block procedure STDERROR is to be used. This means that in case of a hard error the run is terminated with a standard message on the current output unit. See section 5.3.4 about STDERROR.

line 7 — The tape zone is opened. The second parameter, 18, means that the document to be connected is a magnetic tape in odd parity. The possible codes for kind and mode of document are given in Section 5.2.3. The third parameter is the name of the relevant document to be mounted on a tape unit.

The fourth parameter says, that when end of file is read

(indicated by status bit, see Section 5.3.1), the normal reaction (described in Section 5.3.2) is to be replaced by a call of the block procedure.

line 8 — The line printer zone is opened. Note, that a special document name, printer, is used. Usually each installation has certain document names connected with devices as tape reader, tape punch, and printer, where the documents are anonymous.

The giveup mask is zero, which means that errors are treated as described in the subsection of 5.3.2 headed Paper tape punch, line printer.

line 9 — The document connected to the zone tapzon is now positioned to file 1, block 0, as specified by the two last parameters.

lines 10-14 — These lines should be considered together with lines 18-25 declaring the special block procedure. This is the central loop, which reads from the input tape, writes on the line printer until end of file is met on the magnetic tape.

When this happens the block procedure is called and indicates the end of file by setting the variable eof to 1.m

lines 15-16 — The zones are closed, i.e. transfers are terminated and the documents are disconnected from the zones. The parameters .true. specifies that the documents are made available to other users.

lines 18-19 — The block procedure is called after a block transfer if end of file is sensed or if an i/o-error occurs (parity error, etc.). The giveup mask of 1 .shift. 16 in line 7 signals, that the block procedure must be called at end of file instead of executing the standard action.

The parameters are:
z the zone, s the status word with information about the reason of the call, b the length of the block transferred.

lines 21-22 — If the call reason is an i/o-error the standard block procedure is called.

line 23 — The block procedure may be called when positioning the tape to file 1, but the value of b determines if the tapemark was sensed during a read or during a positioning. (See section 5.3.2, subsection for magnetic tape).

line 24 — End of file is signalled to the main program by setting eof to 1. The block length b is set to 16 corresponding to 4 reals (see Section 5.3.3).

The following pages contain the procedure descriptions for OPEN and CLOSE together with some notes on SETPOSITION, which are necessary for use with simple input/output.

5.2.3 Subroutine OPEN

Connects a document to a given zone in such a way that the zone may be used for input/output with the high level zone procedures.

```
CALL      OPEN(z, modekind, doc, giveup)

z         (call and return value, zone). After return, z
          describes the document.~

modekind  (call value, integer). Mode.shift.12 + kind.
          See below.

doc       (call value, text). A text specifying the name
          of the document as required by the monitor,
          i.e. a small letter followed by a maximum of
          10 small letters or digits ended by a
          NUL-character. Short texts of up to 5
          characters + NUL may be given as a LONG
          variable or a text constant. Longer document
          names must be given in two consecutive element
          of a LONG array. The first element of the
          array is given as parameter to OPEN (see
          Example p5.2.1).

giveup    (call value, integer). Used in connection with
          the checking of a transfer. See below.
```

Modekind
Specifies the kind of the document (typewriter, backing storage, magnetic tape, etc.) and the mode in which it should be operated (even parity, odd parity, etc.).

The kind of the document tells the input/output procedures how error conditions are to be handled, how the device should be positioned, etc. As a rule, the procedures do not care for the actual physical kind of the document, but disagreements will usually give rise to bad answers from the document and an error condition arises. If you, for example, open a backing storage area with a kind specifying printer, and later attempt to output via the zone, the backing storage area will reject the message because the document was initialised as required by a printer.

On the other hand, if new kinds of devices are introduced, these may be used directly in fortran if you can find a kind which corresponds to the way these devices should be handled. Mode and kind must be coded as shown in the table below. If you attempt a mode or kind which does not fit into the table, the run is terminated.

Kind

0 Internal process, mode = 0.
4 Backing storage area, mode = 0.
8 Typewriter, mode = 0.
10 Paper tape reader, mode = 0 for odd parity, 2 for even parity (the normal ISO form), 4 for no parity, and 6 for conversion from flexowriter code to ISO.
12 Paper tape punch, mode = 0 for odd parity, 2 for even parity (the normal ISO form), 4 for no parity, and 6 for conversions from ISO to flexowriter code.
14 Line printer, mode = 0 for all printers, except centronics 101A via medium speed tmx where mode = 64.
16 Card reader, see Ref. 8 for full details.
18 Magnetic tape (tapes of 6 or 8 bit physical characters). For RC 747 and RC 749:
Mode = 0 or 4 means odd parity.
Mode = 2 or 6 means even parity.
For RC 4739 and RC 4775 modekind is defined to be:
T .shift. 16 + Mode .shift. 12 + 18, where
Mode = 0 means 1600 bpi, PE, odd parity.
Mode = 2 means 1600 bpi, PE, even parity.
Mode = 4 means 800 bpi, NRZ, odd parity.
Mode = 6 means 800 bpi, NRZ, even parity.

For output 0 < = T < 6 specifies that the last T physical characters in a block should not be output to the tape.

For input T should be 0.

If you use T < > 0 during output, you should set the word defect bit (1 shift 7) and the stopped bit (1 shift 8) in your giveup mask and after a check of bytes transferred simply ignore the bits in your block procedure.

Initialisation of a Document

Open prepares the later use of the document according to kind:

Internal process, backing storage area, typewriter:
Nothing is done. When a transfer is checked later, the necessary initialisation is performed.

Paper tape reader, card reader:
First, open checks to see whether the reader is reserved by another process. If it is, the parent receives the message `'wait for <name of document>'` and open waits until the reader is free. Second, open initialises the reader and empties it. Third, open initialises the reader again (in order to start reading in lower case), sends a parent message asking for the reader to be loaded, and waits until the first character is available.

Paper tape punch, line printer:
Open attemps to reserve the document for the job, but the result of the reservation is neglected.

Magnetic tape:
If the tape is not mounted, a parent message is sent asking for mounting of tape. The message is sent without wait indication (see Ref. 7).

Some of these rules have been introduced to remedy a possible lack of an advanced operating system.

Giveup

The parameter giveup is a mask of 24 bits which will be compared to the logical status word each time a transfer is checked. If the logical status word contains a one in a bit where giveup has a one, the standard action for that error condition is skipped and the block procedure is called instead (the block procedure is also called if a hard error is detected during the checking).

Zone State

The zone must be in state 4, after declaration. The state becomes positioned after open (ready for input/output) except for magnetic tapes, where setposition must be called prior to a call of an input/output procedure.

The entire buffer area of z is divided evenly among the shares and if the document is a backing storage area, the share length is made a multiple of 512 halfwords. If this cannot be done without using a share length of 0, the run is terminated.

The logical position becomes just before the first element of block 0, file 0.

5.2.4 Subroutine CLOSE

Terminates the current use of a zone and makes the zone ready for a new call of open. Close may also release a device so that it becomes available for other processes in the computer.

```
CALL    CLOSE(z, rel)

z       (call and return value, zone). Specifies the
        document, the position of the document, and the
        latest operation on z.

rel     (call value, logical). True if you want the
        document to be released, false otherwise.
```

CLOSE terminates the current use of the zone as described for SETPOSITION. If the document is a magnetic tape which latest has been used for output, a tape mark is written.

Finally, CLOSE releases the document if rel is true. Releasing means for a backing storage area that the area process description inside the monitor is released for use by other zones of yours. The area itself is not

removed and you may later open it again, provided that it has not been removed meanwhile by some other process (this may be prevented as described in Ref. 1).

Releasing means for other documents that the corresponding peripheral device is made available for other processes.

Zone State

The zone may be in any state when CLOSE is called. After the call the zone is in state 4, after declaration, meaning that it must be opened before it can be used for input/output again.

5.2.5 Logical Function SETPOSIION

```
CALL          SETPOSITION(z, file, block)

setposition   (return value, logical). True if a
              magnetic tape positioning has been
              started, false otherwise.

z             (call and return value, zone) Specifies
              the document, the position of the
              document, and the latest operation on z.

file          (call value, integer). Irrelevant for
              documents other than magnetic tape.
              Specifies the file number of the wanted
              position. Files are counted from 0.

block         (call value, integer). Irrelevant for
              documents other than magnetic tape or
              backing storage. Specifies the block
              number of the wanted position. Blocks are
              counted from 0.
```

SETPOSITION performs the following actions:

1. If the zone is used for output remaining data blocks in the buffer area are transferred to the document. A tape mark is then output if the document is a magnetic tape. If the zone is used for input possible running transfers are waited for.

2. If the document is a magnetic tape, this is now moved to a position immediately in front of the block determined by the specified file number and block number. If the document is a backing storage area no movement takes place, block to be transferred is determined by the specified block number. A detailed description of SETPOSITION is found in Section 5.5.6.

5.2.6 REWIND, BACKSPACE, and ENDFILE

ISO FORTRAN (see Ref. 6) specifies some auxiliary procedures for magnetic tape operation. The functions of these procedures are included in the procedures SETPOSITION and CLOSE as follows:

REWIND — Rewind magnetic tape to load point. Replace by: CALL SETPOSITION(z,0,0)

BACKSPACE — Backspace one block. A subroutine performing this operation within the current file may look like:

```
SUBROUTINE BACKSPACE(z)

zone z; integer file , block
call GETPOSITION(z, file, block)
if (block .gt. 0)
call SETPOSITION(z, file, block-1)
end
```

ENDFILE — Write file mark. This is done automatically when SETPOSITION or CLOSE is called with a zone, which has latest been used for output.

5.2.7 Subroutine ZASSIGN

```
CALL        ZASSIGN (z, unitnumber)

z           (call value, zone)

unitnumber  (call value, integer)
```

Assigns a unitnumber to the specified zone so that future READ/ WRITE statements may access the zone by specifying the unitnumber.

If the zone already has been assigned a unitnumber the zone will only be accessible via the new unitnumber.

If the unitnumber already has been assigned to a zone only the new zone will be accessible via the unitnumber.

If the unitnumber is negative the zone is not accessible via a unitnumber.

The assignment is only defined as long as the zone is defined.

The zone is (of course) always accessible by means of the zone name itself.

The standard zones IN and OUT have been preassigned the unitnumbers 5 and 6 respectively.

5.3 Treatment of I/O Errors

5.3.1 Logical Status Word, Kind of Errors

Errors occurring during i/o-operations are indicated in the logical status word, which is generated by the basic i/o system at the end of each operation of a document. The following sections give a survey of the conventions for the logical status word and the standard actions taken for each kind of error and each kind of document. The bits of the logical status word:

`1 .shift. 23:` **`Intervention`**. The device was set in local mode during the operation, presumably because the operator changed the paper or the like.

`1 .shift. 22:` **`Parity error`**. A parity error was detected during the block transfer.

`1 .shift. 21:` **`Timer`**. The operation was not completed within a certain time defined in the hardware.

`1 .shift. 20:` **`Data overrun`**. The high speed channel was overloaded and could not transfer the data.

`1 .shift. 19:` **`Block length`**. A block input from magnetic tape was longer than the buffer area allowed for it.

`1 .shift. 18:` **`End of document`**. Means various things, for instance: Reading or writing outside the backing storage area was attempted; the paper tape reader was empty; the end of tape was sensed on magnetic tape; the paper supply was low on the printer; end of deck on card reader. See Ref. 1 for further details.

`1 .shift. 17:` **`Load point`**. The load point was sensed after an operation on the magnetic tape or output tray was full on card reader.

`1 .shift. 16:` **`Tape mark`**. A tape mark was sensed or written on the magnetic tape, or the attention button was pushed during typewriter i/o.

`1 .shift. 15:` **`Write-enable`**. A write-enable ring is mounted on the magnetic tape.

`1 .shift. 14:` **`High density`**. The magnetic tape is in high density mode.

`1 .shift. 13:` **`Reading error on card reader.`**

`1 .shift. 12:` **`Card rejected on card reader.`**

`1 .shift. 8:` `Stopped`. Generated by the check routine when less than wanted was output to a document of any kind or less than wanted was input from a backing storage area.

`1 .shift. 7:` `Word defect`. Generated by the check routine when the number of characters transferred to or from a magnetic tape is not divisible by the number of words transferred, i.e. when only a part of the last word was transferred.

`1 .shift. 6:` `Position error`. Generated by the check routine after magnetic tape operations, when the monitors count of file and block number differs from the expected value in the zone descriptor.

`1 .shift. 5:` `Process does not exist`. The document is unknown to the monitor.

`1 .shift. 4:` `Disconnected`. The power is switched off on the device.

`1 .shift. 3:` `Unintelligible`. The operation attempted is illegal on that device, e.g., input from a printer.

`1 .shift. 2:` `Rejected`. The program may not use the document, or it should be reserved first

`1 .shift. 1:` `Normal answer`. The device has attempted to execute the operation, i.e. 1 shift 5 to 1 shift 2 are not set. (Other bits may be set).

`1 .shift. 0:` `Hard error`. The standard error action has classified the transfer as a hard error, i.e. the error recovery could not succeed.

5.3.2 Standard Error Actions

The bit 'normal answer' is always ignored; the remaining standard error actions depend on the document kind given in OPEN as shown below. This kind has not necessarily any relation to the actual physical kind. Situations not covered by the description are hard errors, which mean that the block procedure is called (see below).

Backing Storage

End of document after an input operation:
The empty block read is replaced by a block of two halfwords containing 3 End of Medium characters.

Stopped:
If the original status w not show 'end of document', the operation is repeated (see below).

Process does not exist:
The corresponding area process is created. If the creation fails, it is a hard error. Otherwise, the operation is repeated.

Rejected:
The area process is reserved for exclusive access. If the reservation fails, it is a hard error. Otherwise the operation is repeated (see below).

Typewriter

Intervention:
Ignored.

Timer after input operation:
Ignored.

Stopped:
The remaining part of the block is output (see below).

Paper Tape Reader, Card Reader

Parity error:
Ignored.

End of document:
If some characters were input, the bit is ignored. Otherwise the empty block is replaced by a block of two halfwords containing 3 End of Medium characters.

Paper Tape Punch, Line Printer

Intervention:
Ignored.

End of document:
A message is sent to the parent asking for stop of the job until the paper has been changed (see parent message below).

Stopped:
The remaining part of the block is output (see below).

Magnetic Tape

Note that the handling of 'does not exist' and 'rejected' makes a magnetic tape very robust against operator errors. In fact, the operator may unload a magnetic tape at any moment and later mount it on another station without harming the job which used the tape.

Intervention, load point, write-enable, and high density:
Ignored

Parity error and word defect:
The stopped bit is ignored in this case. An input operation is repeated

up to 5 times (see below), but if the parity error persists, the error is a hard one. An output operation is repeated up to 5 times with erasure of 6 inches of tape the first time, 12 the second, and so on. If the parity error persists, the error is a hard one. During erasure and positioning only parity error, timer, and technically impossible bits cause a hard error.

Tape mark:
(Ignored after a sense or move operation.) The position error bit, word defect bit, and parity error bit are ignored when the tape mark bit is set. If the tapemark occurs after an input operation the block is replaced by a block of two halfwords containing 3 End of Medium characters.

Stopped:
If the write-enable bit is 1, the output is repeated (see below). Otherwise, a parent message is sent asking for stop of the job until the ring has been mounted. The error action continues as if the operation had been rejected.

Does not exist:
This bit is ignored after a sense operation or a move operation. In other cases, a message is sent to the parent asking for stop of the job until the tape has been mounted (see parent message below). Next, the tape is reserved for exclusive access and if this fails, the parent message is sent again. Thirdly the tape is positioned according to file and block count, and the operation is repeated as explained below.

Rejected:
Handled as 'does not exist', except that the parent message is not sent initially.

Internal Process

End of document after an input operation:
If a non-empty block was input, the bit is ignored. Otherwise, the empty block is replaced by a block of two halfwords containing three End of Medium characters.

Stopped:
If the original logical status word does not show 'End of document' the remaining part of the block is output (see below).

Output Remaining

The remaining part of a block is output as described under repeat operation.

Repeat Operation

An operation is repeated in 4 steps: 1) If the document is a magnetic tape, it is positioned in front of the latest block which was succesfully transferred and then the tape is upspaced one block (this is a safe method when rewriting a block). 2) The operation (or the remaining part of an output transfer) is started again. 3) All other operations in the

zone are waited for (but not checked), and then started again. 4) The first operation is waited for and checked again. Only the last check involved in a series of repetitions may cause a call of the block procedure.

Parent Message

The parent (the operating system for your job) may either handle a message according to its own rules, or it may pass the request on to the operator. The job may ask the parent to stop the job temporarily until the operation has been performed. The exact rules depend on the operating system in question.

5.3.3 The Block Procedure and the Giveup Mask

If the standard error actions described in Section 5.3.2 are sufficient, the giveup mask 0 should be used in the call of OPEN and STDERROR should be used as block procedure. This will usually be sufficient with formatted READ/WRITE. Formatted READ will stop on end of medium and signal this through the external variable READERR (see Section 5.4.5).

The standard error action for a given kind of error may be suppressed by calling OPEN with a giveup mask containing a 1 in the bitposition connected to the relevant error situation.

The procedure for each completed block transfer proceeds in the following steps:

1. Perform standard error actions for all bits in the logical status word, which are not suppressed by the giveup mask. This may include repetition of the operation, in which case the procedure starts with 1. again.

2. Call the block procedure if either (a) one of the standard error actions has terminated with a hard error, or (b) if an error corresponding to one of the bits in the giveup mask has occured.

The block procedure is called as follows:

```
CALL      BLOCKPROC(z,s,b)

z         (call value, zone). The record of z is the
          entire share available for the transfer.

s         (call value, integer). The logical status word
          after the transfer.

b         (call and return value, integer). The number
          of halfwords transferred.
```

If the block procedure is called because of a hard error, the last bit of the status word is 1. If the call is caused by a bit in the giveup mask, the bit is 0.

Purpose and Return

In the block procedure, you can do anything to the zone by means of the primitive zone procedures (see Ref. 2) and the high level i/o-zone procedures (in the latter case you must be prepared for a recursive call of the block procedure).

You signal the result of the checking back to the high level zone procedure by means of the final block length, b. The value of b has no effect when an output operation is checked, but after an input operation you may signal a larger or a shorter block or even an empty block (b = 0). However, the value of b at return must correspond to a block which is inside the shared area specified by the value of used share at return. Otherwise, the run is terminated with an index alarm. Further details may be found in Ref. 2.

5.3.4 Subroutine STDERROR

Terminates the run with an error message specifying an error condition on a peripheral device.

```
CALL      STDERROR(z, s, b)

z         (call value, zone). Specifies the name of the
          document.

s         (call value, integer). The logical status word
          after a device transfer.

b         (call value, integer). The number of halfwords
          transferred.
```

The run is terminated with the alarm message:

```
giveup<value of b> ...
called from ...
```

The file processor prints the logical status word 's' after the alarm message from the fortran program.

5.4 READ/WRITE Statements

5.4.1 Introduction

The READ/WRITE statements are used for transmission of data between the core store and the external media. The transmission may be on character level with format controlled READ/WRITE or it may be a transmission of binary values without format control.

5.4.2 READ/WRITE with Format Control

The formatted READ/WRITE statement has the form

```
{ READ  }                                    {             }1
{       } (<logical unit>,<format label>) { <data list> }
{ WRITE }                                    {             }0

                  { <zone name>                          }
<logical unit>::= { <zone array name> (<expression>) }
                  { <unit number>                        }

<unit number>::= <expression>

               [ <implied do list>  ]*
<data list>::= [                    ]
               [ <simple data list> ] 1

<implied do list>::=
                                    {            }3
        (<data list>,<integer name>= {<expression>}  )
                                    {            }2

                        [                       ] *
<simple data list>::= [ <simple data elements> ]
                        [                       ] 1

                           [ <expression> ]
<simple data elements>::= [ <array name> ]
                           [ <zone name>  ]
```

The READ/WRITE statement specifies

a. The operation to take place (read or write).
b. The zone (maybe specified by means of a unit number)(to which a document must be connected) to be involved in the input/output operation.
c. A controlling format, which contains a list of format elements each describing the picture of an external field and a possible conversion to take place between the external and the internal representation (or vice versa) of a data element.
d. A list of data elements to be transferred to or from the document.

Depending on their kind the data elements are treated according to the following basic rules:

simple variables:
A value is transmitted to or from the variable.

subscripted variable:
The subscript value(s) are evaluated and a value is from the determined array element.

array:
Values are transmitted to or from all array elements column by column. Values are transmitted to or from the whole zone record.

expression or constant:
With WRITE the value of the expression or the constant is transmitted as a simple variable. With READ the operation is meaningless. A value is read but lost internally.

Implied DO

A list of i/o elements may be controlled by an implied DO loop. The interpretation of the control parameters in the implied DO loop is quite analogous with that of the normal DO loop (see Section 4.4). For each value of the control variable the controlled i/o list is processed according to the rules described in the remaining part of the paragraph. An implied DO list is considered a basic element in an i/o list and may again be controlled by another implied do construction.

Example 5.4.1

```
(x(2,j), (a(i,j), b(i), i = 1,7), c(j), j = 1,3)
```

is analogous to the following DO loop:

```
      do 20 j = 1, 3
      transfer to x(2, j)
      do 10 i = 1, 7
      transfer to a(i, j)
  10  transfer to b(i)
  20  transfer to c(j)
```

Note that all the parameters in a <simple data list> are evaluated before any data are transferred to or from the data elements, so that the call:

```
read (unit,10) n, A(n)
```

will mean:

```
i = n
n = number
A (i) = number
```

while the call:

```
read (unit, 10) n, (A (n), i = 1,1)
```

will mean:

```
n = number
A (n) = number
```

5.4.3 The FORMAT Statement

Form:

```
{ FORMAT  }
{         } (<format list>)
{ FORMATO }

                 [{         }1                                  ]*
                 [{<integer>}    <repeatable format element>    ]
<format list>::= [{         }0                                  ]
                 [ <non-repeatable format element>              ]1

                                {<simple format element>}
<repeatable format element>::= {                        }
                                {<format group>         }

<format group>::= (<format list>)

<simple format element>::= { B <integer>.<integer>  }
                           { I <integer> .<integer> }
                           { F <integer>.<integer>  }
                           { E <integer>.<integer>  }
                           { D <integer>.<integer>  }
                           { A <integer>            }
                           { L <integer>            }
                           { X                      }

<non-repeatable format element>::= { <integer> H <symbols> }
                                   { '<symbols>'            }
                                   { <integer> P            }
                                   {  /                     }
```

The G-conversion is not implemented.

Rules

1. The comma between format elements may be replaced by a / (slash), which will work as a change of line (see Section 5.4.5).

2. A format statement must be labelled. The label is called a format label.

3. The <integer> in front of <repeatable format element> is called the repeat-factor.
   If the <integer> is omitted it is the same as the repeat-factor = 1.

4. A comma after a P-format-element is not needed.

Open Formats

Formats declared by the word FORMATO are called open formats. They differ from the usual (closed) formats in two situations: (1) When a READ/WRITE statement is terminated, a line change will take place if the format is closed, but not with an open format. (2) When the terminal right parenthesis is passed during the format interpretation, a closed format will effect a line change, but an open format will not. See Section 5.4.5 for further description of the execution of READ/WRITE statements. See Ex. 5.4.10 about open format.

5.4.4 Details about the Format Elements

5.4.4.1 E-Conversion

Form: `E <field length>.<fraction length>`

The E conversion may be used with real values.

Input

An external field of the specified length is read and converted to a real value. The external field must have the form:

```
(           ) ( D )               ) 1
( <integer> ) (   ) <integer>     )
(           ) ( E )               )
( <real>    )                     )
(           )   <signed integer>  ) 0
```

The fraction length defines the number of digits behind an implicit decimal point. If an explicit decimal point is found this overrules the implicit point defined by the format.

In the following examples b denotes a blank position.

Example 5.4.2

external field	format	internal value
b1234E02	E8.4	12.34
bb1234+5	E8.0	123400000.0
b1.23E04	E8.3	12300.0
1.23D+4b	E8.3	1.23 * 10 ** 40
12345678	E8.3	12345.678

Output

The external field produced will be of the form:

```
{   } 1
{ - }    <digit>.<fraction part><exponent>
{   } 0

<exponent> ::= { E <sign><digit><digit>      }
               { ( E ) <digit><digit><digit> }
               { ( - )                       }
```

The number of digits in the fraction part is determined by the fraction length of the specified format. The specified field length includes the positions occupied by the exponent. If the value cannot be accomodated by the format specified this is indicated by printing an asterisk in the first position followed by as many digits as possible discarding the leftmost digits.

Example 5.4.3

```
  internal value        format     external field
  1234.567              E10.3      b0.123E+04
 -1234.567              E10.3      -0.123E+04
1.234*10**107           E10.3      b0.123+108
1.234*10**(-107)        E10.3      b0.123-106
```

5.4.4.2 F-Conversion

Form: `F <field length>.<fraction length>`

The F-conversion may be used with real values.

Input

An external field of the specified length is read and converted to a real value. The external field must be as described under the E-conversion.

Example 5.4.4

```
external field      format      internal value
1234567             F7.3        1234.567
bbbbb67             F7.4        0.0067
12.3456             F7.3        12.3456
123bbbb             F7.2        12300.00
```

Output

The internal value is converted to an external value and rounded according to the specified fraction length and total field length. If the value requires more positions than provided by the format an alarm takes place as follows: the number is printed with with E-format thus supplying the approximate size. The fractional part of the alarm print contains an asterisk followed by some irrelevant digits.

Example 5.4.5

```
internal value      format      external field
123.45432           f8.3        b123.454
123.45432           f8.2        bb123.45
123.45432           f8.1        bbb123.5
123.45432           f7.5        *54E+03
```

5.4.4.3 D-Conversion

Form: `D <field length>.<fraction length>`

The D conversion may be used with double precision values.

Input

An external field of the specified length is read and converted to a double precision value. The external field must have a form as described for E-conversion.

Output

The external field corresponds to that of an E-field except that the maximum number of digits in the basic constant is 19.

5.4.4.4 A-Conversion

Form: `A <integer>`

The A-conversion may be used with type integer, long, real and complex. On input external characters are converted to internal 8-bit ISO codes. The codes are packed left-adjusted, 3 per word filling with SPACE. On output each 8-bit group is considered an ISO character code and converted to an external character.

If the number of characters specified exceeds the capacity of the variable, the last characters are lost at input and replaced by SPACE at output.

Note that NUL-characters do not count, that is: NUL-characters are ignored at input and are never output by write.

5.4.4.5 I-Conversion

```
                              {                    } 1
Form:      I <field length>{.<point position> }
                              {                    } 0
```

The I-conversion may be used with integer or long values.

Input

An external field of the specified length is read and converted to an integer value. A point position specified with input has no effect. The external input field may not contain a decimal point.

Output

The integral internal value is converted to an external integer field adding leading spaces if necessary. If the value is negative a minus sign is printed in front of the first digit. If the point position is specified a decimal point is printed in front of the 'point position' rightmost digits. If the value cannot be accomodated by the specified format this is indicated by an asterisk followed by the last digits of the number.

5.4.4.6 B-Conversion

Form: `B <field length>.<radix power>`

The B-conversion may be used with type integer, long, real and complex.

Input

An external field of the specified length is read and interpreted as an integer with the radix 2** <radix power>. The radix power may take the values 1, 2, 3, or 4 only.

Output

The internal value is converted to an external integer each digit occupying <radix power> bits internally.

Example 5.4.6

```
internal bit pattern              format  external field
000 001 010 011 100 101 110 111   B12.2   bb1103211313
                                  B8.3    b1234567
                                  B6.4    b53977
```

5.4.4.7 L-Conversion

Form: `L <field length>`

The L-conversion may be used with logical values.

Input

An external field of the specified length is read and converted to a logical value. The external field must have the form

```
{          }* { F } {             }*
{ <spaces> }  {   } { <character> }
{          }0 { T } {             }0
```

T will result in the value .true. and F in the value .false.

Output

If the logical value is .true. a T will be output, F otherwise.

5.4.4.8 Scaling Factor

A scaling factor may be imposed on reals converted with E, F and D conversions. The prefix

```
<10-exponent> P
```

will cause that external values are equal to their corresponding internal values multiplied by 10 raised to the integer given. With E- and D-conversions and WRITE, the fraction part of the external number is multiplied by the scaling factor and the exponent part is reduced correspondingly. The scaling factor is deleted at the termination of the actual READ/WRITE statement. If an exponent part is explicitly present in an input field the scaling factor is ignored during the conversion of that field.

5.4.4.9 Spaces and Text

The element

```
X
```

has no corresponding internal value. At input a single external character is skipped. At output a single SPACE is printed.

Text elements, which may be used for output only may be specified in two ways:

```
<length of textstring> H <textstring>
```

or

```
'<text string not containing '>'
```

5.4.5 Execution of Formatted READ/WRITE

The execution of a formatted READ/WRITE is controlled by the i/o list and the format as follows:

1. The format is scanned, and each format element is inspected.

2. Whenever a repeatfactor occurs, the succeeding repeatable format element is repeated the specified number of times.

   In case of a format group, the whole group is repeated the specified number of times.

3. Whenever the current format element requires a value to be converted, an element from the i/o list is processed. If the i/o list already was exhausted, the execution of the formatted READ/WRITE is terminated.

4. In case the i/o list is not exhausted when the terminating parenthesis is met, the interpretation of the format is restarted.

The restart of a format takes place from:

a. The beginning of the format, in case the format contains no format groups,

b. From the last non-nested format group (i.e. the last format group at the outermost level), including a possible repeat factor, in case the format contains format groups.

The execution of a formatted READ/WRITE is terminated when:

a. the current format element requires an element from the i/o list, and the i/o list is exhausted, or

b. the terminating right parenthesis is met and the i/o list is exhausted, or

c. the terminating right parenthesis is met, and no values have been converted since the last restart of the format (or since the initial start),

whichever occurs first.

The following examples may clarify the restart-point of a format:

```
10  format  (<anything><repeatfactor> (<anything>)
                        x              <anything not
                                       <containing ()>)

20  format (<anything not containing ()>)
           x
```

The x's show where the format scan will be restarted. See also example 5.4.7.

5.4.5.1 Line Change

Record change (in RC FORTRAN: line change) is controlled by the format.

The line change action is envoked when:

a. A / (slash) is encountered in the format

b. A closed format is restarted

c. Interpretation of a closed format is terminated

Line Change in READ

By READ the line change action means that all characters up to and including the first occurring NL-character are skipped.

Line Change in WRITE

By WRITE there are two different line change modes, depending on the zone:

1. If the zone is connected to a printer-like device (e.g. a terminal or a printer etc.) the line change action is actually delayed until the first character after the line change.

   This character is used as control character for vertical spacing, i.e. it is replaced as follows:

Vertical Character	spacing before printing	Converted character(s)
0	two lines	NL, NL
1	to first line of next page	FF
+	no advance	CR
Anything else	one line	NL

   Notice: The 'no advance' facility will not work on all output devices.

2. If the zone is connected to a non-printer-like device (e.g. a disc file etc.) the line change action is carried out immediately.

   The line change action consists of output of a NL character.

The discrimination between the two line change modes is controlled by means of the 'line change mode' in the zone descriptor:

The first bit (i.e. `1 .shift. 23`) means:

0: printer-like device
1: non-printer-like device

The last bit (i.e. `1 .shift. 0`) means:

0: line change action is pending (i.e. after FORMAT)
1: no pending line change action (i.e. after FORMATO)

When a zone has been declared, the 'line change mode' is zero, i.e. printer-like device with pending line change action.

The standard zones IN and OUT both have the 'line change mode' set to zero, when the program is started.

The following subroutine will set the 'line change mode' to **nonprinter-like:**

```
SUBROUTINE NONPRINT(Z)
ZONE Z
INTEGER ZONEDESCR(20)
CALL GETZONE6(Z, ZONEDESCR)
ZONEDESCR(11) = ZONEDESCR(11) .OR. 1 .SHIFT. 23
CALL SETZONE6(Z, ZONEDESCR)
END
```

The following subroutine will set the 'line change mode' to **printer-like:**

```
SUBROUTINE SETPRINT(Z)
ZONE Z
INTEGER ZONEDESCR(20)
CALL GETZONE6(Z, ZONEDESCR)
ZONEDESCR(11) = ZONEDESCR (11) .SHIFT. 1 .SHIFT. (-1)
CALL SETZONE6 (Z, ZONEDESCR)
END
```

(details about the procedures GET/SETZONE6 may be found in the ALGOL manual, see ref. 2).

5.4.5.2 READ/WRITE Errors

Logical READ/WRITE errors occur if the type of a data element conflicts with the conversion specified by the format, e.g. a logical variable to be converted by an E-version. At READ operations an error also occurs if the length of an input line is shorter than specified by the controlling format. The following table shows the result of combining the conversion codes from the format with data elements of different types.

elem. type / conv. code	integer	long	real	double pr.	logical
I	X	X	(X)	F	F
E, F	F	(X)	X	F	F
D	F	F	F	X	F
L	F	F	F	F	X
A	X	X	X	F	F
B	X	X	X	F	F

F: illegal
X: legal
(X): legal but result is strange

5.4.5.3 Treatment of WRITE Errors

With WRITE the illegal combinations are indicated by filling the external field with asterisks.

5.4.5.4 Treatment of READ Errors, Standard Variable READERR

After the execution of a formatted READ statement information about the result is available from the standard integer variable READERR. The variable contains the ISO-values of the last processed character and of a possible erroneous character. READERR has the following values:

(1) if an error occured: error char .shift. 8 + last char
(2) if no error: last char

If more than one error occurs within a READ statement, the first is indicated in READERR. After a line change, error synchronisation is performed as follows:

The format is scanned without reading any input text until a line change is required by the format. Elements from the i/o list are taken according to the scanned format but are left unchanged. Reading then proceeds from the current position within the format, the i/o list and the input text.

If an i/o list element is involved in a conversion error the element is left unchanged.

Note, however, that elements read with A-conversion will be filled with spaces.

Example 5.4.7. Integer Standard Variable READERR.

The program will read a collection of v inserted values to the array elements a(1), a(2) ... a(v). These values are sorted in ascending order

and finally the p'th through q'th lowest af the v values are written on current out. This process is repeated until the input file is exhausted.

In case of error during input the erroneous character and the last character read is written on current out.

```
          program simplesort
          real a(500)
          integer p, q, v

10      read(5,1000) p, q, v
          if (inerror(0)) 20,100,99
20      read(5,1010) (a(i),i=1,v)
          if (inerror(2) .ge. 0) goto 100

          do 50 i=1,v
             j = i
30        index = j ; small = a(index)
             do 40 j=j+1,v
40           if (small .gt. a(j)) goto 30
             a(index) = a(i)
50        a(i) = small

          write(6,2000) (a(i),i=p,q)
          goto 10

1000    format(3i5)
1010    format(7f10.1)
2000    format(//,4(3x, f10.1))

99      call inerror(1)
100     end

          function inerror(number)
          external readerr
          integer readerr, errorchar

          errorchar = readerr .shift. (-8)

          inerror = -1
          if (errorchar .eq. 0) return
          inerror = errorchar
          if (errorchar .eq. 25) inerror = 0
          if (number .eq. 0) return

          write(6,2000) number, errorchar, readerr .and  4bff
2000    format(/,' ***readerror in situation ', i3,
         1          /,' errorchar value: ', i3,
         2          /,' last char value: ', i3)

          end
```

5.4.6 READ/WRITE without Format Control

An unformatted READ/WRITE statement has the form

```
{ READ  }                    {              }1
{       } (<logical unit>) { <data list> }
{ WRITE }                    {              }0
```

Data are transferred between external media and data elements in core store without conversion.

Example 5.4.8. Unformatted READ

```
zone zo(256, 2, stderror)
integer i1, ia(10)
real a, b(4)
-
read(zo) i1, (ia(j), i = 3, 6), a, b
```

The following words are transferred:

```
word 1       i1
words 2-5    ia(3), ... ia(6)
words 6, 7   a
words 8-15   b(1), ... b(4)
```

Example 5.4.9. Record Input with Unformatted READ

A file consists of records with the following layout:

```
word
1              item no
2              number of transactions, N
3              price of item
4              no of items on store
5 -            One subrecord per transaction
           (1) Salesman no
           (2) no of items sold
           (3) customer no
           (4) total
```

This file may be read like

```
 READ(stfile) itemno, notrans, price,
1onstore, (salem(j), sold(j), cost(j), tot(j), j = 1, notrans)
 do 100 j = 1, notrans
 if (sold(j) .gt. 50) write(out,20) salem(j), cost(j), tot(j)
 altot = altot + tot(j)
 etc.
```

Example 5.4.10. Reading a Paper Tape from the Tape Reader.

Usually the paper tape reader of an RC4000/6000/8000 installation is named 'reader', and a typical piece of program may look like

```
long rname(2)
zone z(50, 2, stderror)
rname(1) = 'reader'
rname(2) = 0
  -
call open(z, 2.shift.12 + 10, rname(1), 0)
  -
read (z, ...)
call close (z, .true.)
```

Note that (1) document names must be terminated by a NUL-character, (2) names exceeding 5 characters are stored in a long array, and the first element of this array is given as parameter to OPEN.

Example 5.4.11. Syntax Checking with Open Format.

READ may be used for partly checking of input records of various layouts. Suppose that three types of input records are defined. The format corresponding to the types are:

```
type        format
1           (i2, 2x, i6, 1x, i9, 1x, i4)
2           (i2, 2x, i6, 2x, 10a6)
3           (i2, 2x, i6, 4x, i6, 1x, i9)
```

The input program may then look like:

```
100       formato (i2, 2x, i6)
111       format(1x, i9, 1x, i4)
112       format(2x, 10a6)
113       format(4x, i6, 1x, i9)
          integer readerr; external readerr
            -
          read(zon, 100) rectype, partnumber
c         go to write error message etc.
          if(readerr .gt. 255) goto 9991
c         check value of record type and branch
c         to read remaining record
          goto (9992, 10, 20, 30, 9992), rectype
            -
10        read(zon, 111) amount, supplier
c         go to error procedure if read error
          if(readerr .gt. 255) goto 9993
          etc.
```

5.5 Record Handling

5.5.1 Zone Record

The input/output system of RC FORTRAN contains a set of procedures for reading and writing on record level. For this purpose a ZONE RECORD is introduced as the part of a zone buffer currently available to the user. The definition of the current record may be changed by the record input/output procedures. Thus data may be transferred blockwise between the document and the core store but communicated to the user as a sequence of zone records. The zone name may be used in two different ways: (1) as the name of a zone and (2) as the name of the zone record, considered as a real array with one dimension. E.g., a new record may be input from a zone by a statement like

```
call inrec(z, recordlength)
```

and the elements of the zone record may be used as operands like this:

```
x = y + z(2)
```

The rules for use of a zone record and elements hereof in executeable statements are exactly as those for a one-dimensional real array.

A record element of a subscripted zone is referred with two subscripts, e.g.

```
za (2,7)
```

where the first subscripts selects the zone from the zone array, while the second subscript selects the element from the current zone record of that zone. The example above thus refers to the 7th element of the 2nd zone of the zone array.

The following sections contain detailed descriptions of the record input/output procedures INREC, OUTREC and SWOPREC, and of the procedures SETPOSITION and GETPOSITION used for positioning of documents.

RC FORTRAN handles other record procedures, like INVAR/OUTVAR. For further details, see the ALGOL manual (ref. 2).

The index-sequential system may also be used from RC FORTRAN, see ref. 9.

5.5.2 Integer Function INREC

Gets a sequence of doublewords (elements) from a document and makes them available as a zone record. The document may be scanned sequentially by means of INREC.

```
CALL      INREC(z, length)

inrec     (return value, integer). The number of
          elements left in the present block for further
          calls of inrec.

z         (call and return value, zone). The name of the
          record.
          Determines further the document, the
          buffering, and the position of the document.

length    (call value, integer or real). The number of
          elements in the new record. Length must be >=
          0.
```

Zone State

The zone state must be open and ready for INREC, i.e. the previous call of record handling procedure must be either OPEN, SETPOSITION, or INREC. When using magnetic tape SETPOSITION must follow OPEN. To make sense, the document should be an internal process, a backing storage area, a typewriter, a paper tape reader, a card reader, or a magnetic tape.

Blocking

INREC may be thought of as transferring the elements just after the current logical position of the document and changing the logical position to after the last element of the record. However, all elements of the record are taken from the same block, so if the length exceeds the remains of the current block, the block is changed. Then the record becomes the first elements of the new block, but if this still cannot hold the record the run is terminated (empty blocks are completely disregarded).

Records of length 0 need a special explanation. If not even a single element is left in the block, the block is changed and the logical position points to just before the first element of the new block. Note that inrec changes the blocks in such a way that a portion at the end of a block may be skipped. So be careful to read a backing storage area with the same share length as that with which it was written, otherwise wrong portions might be skipped at reading.

Note furthermore the special considerations about variable length records on backing storage mentioned in Section 5.5.3 on OUTREC.

5.5.3 Integer Function OUTREC

Creates a zone record which later will be transferred to a document as a sequence of doublewords (elements). The content of the record is initially undefined but the user is supposed to assign values to the record variables. The document may be filled sequentially by means of OUTREC.

```
CALL      OUTREC(z,length)

outrec    (return value, integer). The number of
          elements available for further calls of OUTREC
          before change of block takes place.

z         (call and return value, zone). The name of the
          record.
          Determines further the document, the
          buffering, and the position of the document.

length    (call value, integer or real). The number of
          elements in the new record. Length must be >=
          0.
```

Zone State

The zone z must be open and ready for OUTREC, i.e. the previous call of a record handling procedure must be either OPEN, SETPOSITION, or OUTREC. When using magnetic tape SETPOSITION must follow OPEN. To make sense, the document should be an internal process, a backing storage area, a typewriter, a line printer, a punch, a plotter, or a magnetic tape.

Blocking

OUTREC may be thought of as transferring the record to the elements just after the current logical position of the document and moving the logical pointer to just after the last element of the record. The user is supposed to store information in the record before OUTREC is called again. As the output is blocked, the actual transfer of the elements to the document is delayed until the block is changed or until CLOSE or SETPOSITION is called. All elements of the record are put into the same block, so if the block cannot hold a record of the length demanded, the block is changed in this way:

1. Documents with fixed block length (backing storage): The remaining elements of the share are filled with binary zeroes, and the total share is output as one block.

2. Documents with variable block length (all others): Only the part of the share actually used for records is output as a block.

The transfer is checked and the record becomes the first elements of the next share, but if the record still is too long, the run is terminated. A record length of 0 is handled as for INREC. The first rule above requires special attention when variable length records are stored on backing storage. The input administration must be prepared to skip possible binary zeroes.

The following piece of program shows one way of doing this. The record length is supposed to be stored in the fir record.

Example 5.5.1

```
        integer remaining, length, rlng
        zone z (...)
        equivalence (length, z(1))
          -
50      remaining = inrec (z, 1)
        rlng= length
c       save record length before next call of inrec if
        (rlng) 100, 100, 200
100     call inrec (z, remaining)
        goto 50
200     call inrec(z, length - 1)
```

5.5.4 Integer Function SWOPREC

This procedure gives you direct access to a sequence of elements of a document. The elements become available as a zone record, and you may modify them directly without changing the surrounding elements of the document. This makes sense for a backing storage area, only. The procedure works as a combination of INREC and OUTREC in the sense that a sequence of elements is taken from a document and later transferred back to the same place on the document. The document may be scanned and modified sequentially by means of swoprec.

```
CALL        SWOPREC(z, length)

swoprec     (return value, integer). The number of
            elements left in the present block for further
            calls of swoprec.

z           (call and return value, zone). The name of the
            record. Specifies further the document, the
            buffering, and the position of the document.

length      (call value, integer or real). The number of
            elements in the record. Length must be >= 0.
```

Zone State

The zone z must be open and ready for SWOPREC, i.e. the previous call of record handling procedure must be either OPEN, SETPOSITION, or SWOPREC. The document must be a backing storage area.

Blocking

SWOPREC may be thought of as transferring the elements just after the current logical pointer of the document and moving the logical pointer to the last element of the record. As the records are blocked, the actual transfer back to the device is delayed until the block is full or until CLOSE or SETPOSITION is called. All elements of the record are taken from the same block and when the block cannot supply the record requested, the block is transferred back to the document and the next

block is read. The checking of all transfers is performed. If the block still cannot supply the record, the run is terminated. A record length of 0 is handled as for INREC. If the zone contains 3 shares, one is used for input, another is used for output, and the last holds the current record. This ensures maximal overlapping of computation and input-output.

5.5.5 Subroutine GETPOSITION

Gets the block and file number corresponding to the current logical position of a document.

```
CALL        GETPOSITION(z,File,Block)

z           (call value, zone). Specifies the document,
            the position of the document, and the latest
            operation on z.

File        (return value, integer). Irrelevant for
            documents other than magnetic tape. Specifies
            the file number of the current logical
            position. Files are counted 0, 1, 2,...

Block       (return value, integer). Irrelevant for
            documents other than magnetic tape and backing
            storage. Specifies the block number of the
            current logical position. Blocks are counted
            0, 1, 2,...
```

GETPOSITION does not change the zone state and it may be called in all states of the zone. If the zone is not opened, the position will be undefined. The position is also undefined after a call of CLOSE.

5.5.6 Logical Function SETPOSITION

Terminates the current use of a zone and positions the document to a given file and block on devices where this makes sense. The positioning will only involve time-consuming operations on the document if this is a magnetic tape.

```
CALL        SETPOSITION(z,File,Block)

setposi-    (return value, logical). True if a magnetic
tion        tape positioning has been started, false
            otherwise.

z           (call and return value, zone). Specifies the
            document, the position of the document, and
            the latest operation on z.

File        (call value, integer). Irrelevant for
            documents other than magnetic tape. Specifies
            the file number of the wanted position. Files
            are counted 0, 1, 2,... File 0 will normally
            contain the tape volume label, file 1 is then
            the first file available for data. File = -1
```

```
          specifies that the tape is to be unwound.

Block     (call value, integer). Irrelevant for
          documents other than magnetic tape and backing
          storage. Specifies the block number of the
          wanted position. Blocks are counted 0, 1,
          2,...
```

SETPOSITION proceeds in 3 steps: Terminate the current use, write tape mark, and start positioning.

Terminate Current Use

If the zone latest has been used for output, the used part of the last block is sent to the document. A block sent to a backing storage area is not filled with zeroes, contrary to OUTREC. If the zone latest has been used for character output, the termination may involve output of one or two NUL-characters in order to fill the last word of the buffer. Next, all the transfers involving z are completed, the input transfers are just waited for, and the output transfers and other operations are checked. The physical position of a magnetic tape used for input is sh-1 blocks ahead of the logical position where sh is the number of shares. If some of these sh-1 blocks are tape marks, the positioning strategy is affected, as explained below.

Write Tape Mark

If the document is a magnetic tape used latest for output, a tape mark is written. The document is then in a position after this tape mark, which influences the positioning strategy (see below).

Start Positioning

SETPOSITION assigns the value of Block to the zone descriptor variable "segment count" and returns then for all devices other than magnetic tape. If the document does not exist or if the job is not a user of the device, SETPOSITION sends a parent message asking for stop of the job until the tape is ready.

SETPOSITION starts the first operation involved in the tape positioning. The remaining operations are executed the first time the zone is used for input or output, or the first time SETPOSITION is called again. That may be used for simultaneous positioning of more tapes.

The positioning is accomplished by means of the operations rewind, backspace file, upspace file, backspace block, upspace block, and unwind tape. The positioning is complete as soon as File and Block match the monitors count of the tape position for that device. Checking against tape labels is not performed.

Positioning Strategy

If the actual physical file number differs from File, the tape is first positioned to block 0 of that file. SETPOSITION chooses between rewind and backspace file in this way:

```
if actual file number / 2 >= File then rewind
                                  else backspace file
```

This tends to minimise the number of tape operations. During positioning within a file, SETPOSITION chooses between backspace file (rewind for File = 0) and backspace block in this way:

```
if actual blocknumber / 2 >= Block then backspace file
                                   else backspace block
```

If the tape is not mounted when SETPOSITION is called, the normal mounttape action is performed before the positioning starts.

Zone State

The zone must be open when SETPOSITION is called. SETPOSITION changes the zone state so that the zone is ready for input/output.

The logical position of a magnetic tape or a backing storage area becomes just before the first element of the block specified by File and Block. The logical position is unchanged for other devices.

5.5.7 EQUIVALENCE and ZONES

In RC FORTRAN equivalences involving zone record elements are allowed, i.e.:

```
             [                                  ] *
             [    [ <variable name>       ] *   ]
EQUIVALENCE  [  ( [ <array element>        ]   ) ]
             [    [ <zone record element>] 2    ]
             [                                  ] 1
```

The zone equivalence allows the user to connect different names and/or types to the zone record or elements hereof.

Rules

1. At run time it is controlled that zone equivalenced variables are in accordance with the length of the currently defined zone record when used. If the variable is a subscripted variable the usual optional index check is also performed.

2. Normal index checking is performed on arrays equivalenced to zones independent of rule 1.

3. Zone records may not directly or indirectly be equivalenced to other zone records or to variables in COMMON.

4. All subscripts in an EQUIVALENCE statement must be integer constants.

Example 5.5.2

A file contains records consisting of three parts.

a 3 variables of type real.
b an integer array with 6 elements.
c a long array with 8 elements containing text.

The declarations etc. may look like

```
real price, sum, total
integer count(6)
long text(8)
zone z(500,2,stderror)
equivalence (z(1),price),(z(2),sum)
equivalence (z(3),total),(z(4),count(1)),(z(7),text(1))
```

Now, after having called INREC the user may refer to the different parts of the zone record by the names, e.g.,

```
if (price .gt. lim) write(out, 100) price,text(2)
```

Example 5.5.3

Merging of two tape files into a third file. The records are 10 doublewords long and the first element is the merging key.

```
      function endf(z, s, b)
      zone z
c     simulate record with a large key
      if (s .and. 1) 10, 20, 10
c     error
10    call stderror (z,s,b)
20    b=4*10; z(1) = 1.E100
      end

c     start of main program
      program merge
      zone in1(512,2,endf); dimension in1(2)
      common innames(2)
      long innames
      data innames/'doc1', 'doc2'/
      zone result (512,2,stderror)
      logical setposition; integer inrec,outrec
      call open (result, 18, 'docu', 0)
      call setposition (result, 1, 0)
      do 10 i = 1,2
      call open (in1(i), 18, innames(i), 1 .shift. 16)
      call setposition (in1(i), 1,0)
```

```
10    call inrec (inl(i), 10)
20    i=1
      if (inl(2,1) - inl(1,1)) 30,40,50
30    i=2
c     check for end of file
40    if (inl(2,1) .ge. 1.E100) goto 70
50    call outrec (result,10)
      do 60 j=1,10
60    result (j)= inl (i,j)
      call inrec (inl(i),10)
      goto 20
70    call close (result, .true.)
      do 80 i=1,2
80    call close (inl(i), .true.)
      end
```

6. Program Structure

A FORTRAN source text may contain any number of program units. A program unit may be a main program, a subroutine or a function. The source text may contain one subroutine or function, or it may contain one main program together with any number of subroutines or functions. The mutual order of the program units is free.

6.1 Program Units and Their Mutual Communication

6.1.1 Structure of a Program Unit

All program units must contain the following parts in the shown order.

1. Heading
2. Declarations except equivalences
3. Equivalences
4. Executable statements
5. END statement.

Any of the parts may be omitted, except the END statement. If the heading is omitted, a PROGRAM heading is assumed.

A program unit may be a main program, a subroutine or a function. The headings for each of the three kinds of program unit have the following forms:

```
PROGRAM <program name>

                                 {                   }1
SUBROUTINE <subroutine name>     { (<formal list>)   }
                                 {                   }0

{       }1
{<type>}  FUNCTION <function name> (<formal list>)
{       }0
```

where

```
                    [                    ]*
<formal list> ::= [ <parameter name> ]
                    [                    ]1
```

The names listed in the formal list are called the formal parameters. Parameters must either be explicitly or implicitly specified with both type and dimensionality. Formal parameters may not be equivalenced and they may not be in COMMON.

6.1.2 Calling Functions and Subroutines

A subroutine or function may be activated by a CALL statement of the following form:

```
                              {                                 }1
CALL <program unit name>      { (<actual parameter list>)       }
                              {                                 }0

<actual parameter list> ::= { <expression> }*
                            { <array>      }
                            { <zone>       }
                            { <external>   }
                            { <zone array> }1
```

A function may further be called in an expression by using an operand of form:

```
<function name> ( <actual parameter list> )
```

The number and types of the actual arguments must correspond to the number and types of parameters in the declaration of the called program unit. The name of a subroutine or function may be used as an actual parameter. If the name does not occur in a call situation it must be declared in an EXTERNAL statement (see Section 6.1.4).

The legality of the possible actual/formal correspondances are shown in the table below.

Formal	Legal actual parameters
simple variable	constant, variable, expression
array	array, zone, array element, zone record element
zone	zone
external	external
zone array	zone array

If the actual parameter is an expression or a constant, the actual value is stored in an anonymous working variable, and this working variable is transferred to the program unit called.

6.1.3 Parameter Checking

When a program unit is compiled, the compiler searches a description of the referenced externals. The description may be found in the compiler's catalog of program units processed in the current compilation, or it may secondly be looked up in the catalog maintained by the operating system. The description contains information about the kind and type of the external and its possible parameters, and this information is used by the compiler to check the calls of each external. When a single subroutine or function is compiled separately the descriptions of all its externals may not be available at compile time. An assumed description is then deducted from the actual use of the external but only consistency of the use can be checked. When the separately compiled program unit is included in an executable program the unknown external must be available. If the description of the external is still missing or if the description does not match the assumption an alarm occurs.

Notice: The compiler checks at most the first seven parameters of a procedure call.

6.1.4 EXTERNAL Statement

To be recognized as the name of an external a name must be called within the program or it must be declared in an EXTERNAL statement of the form.

```
          [         ]*
EXTERNAL  [ <name>  ]
          [         ]1
```

An external may be

1. A subroutine or function described in the catalog or found in the source text of current compilation.

2. A variable in the permanent core area of the FORTRAN running system, named and described in the catalog.

3. A standard zone, supplied by the operating system and described in the catalog.

6.1.5 Formal and Adjustable Arrays

For formal arrays the forms of the DIMENSION statement and of type declarations are slightly modified to

```
{ <type>    }               [<parameter name>  ]*
{           }<array name>  ([                  ] )
{ DIMENSION }               [<integer constant>]1
```

Only integer parameter names may occur in the list of bounds. If the parameters occur in the list, the array is called an adjustable array. A formal array is defined at entry in the program unit as follows: The dimensions of the array are determined by the bounds given in the

declaration of the formal array. The first element of the array will depend on the actual parameter as shown below:

actual parameter	first element
array	first element of array
zone	first element of zone record
array element	given array element
zone record element	given zone record element

The resulting adjustable array must not exceed the array given as actual parameter. If this happens an index alarm will occur at run time when the called program unit is entered (unless the program unit is translated with the compiler-option "index.no", see appendix B).

Example 6.1.1. Adjustable Arrays

Assume the following declarations:

```
          subroutine pip(ar,dim)
          real ar(dim,3); integer dim
          -
          end

          program pop
          real tab(10,20)
          -
10        call pip(tab,7)
20        call pip(tab(3,4), 3)
20        call pip(tab(5,20),8)
          end
```

The resulting adjustable array will be:

```
statement 10:
first element: ar(1,1) same as tab(1,1)
last element:  ar(7,3)  -    - tab(1,3)

statement 20:
first element: ar(1,1) same as tab(3,4)
last element:  ar(3,3)  -    - tab(1,5)

statement 30:
first element: ar(1,1) same as tab(5,20)
```

but as only 6 elements remain from tab(5,20) until the end of the array tab, an index alarm will occur at entry into subroutine pip, when called from statement 30.

6.1.6 Formal and Adjustable Zones

A formal zone is declared by writing the name of the zone in a ZONE statement without specifying buffer size, number of shares and block procedure (see Section 5.1.5).

An adjustable zone array is declared as a formal zone with an additional

```
                                {<parameter name>  }
DIMENSION <zone array name> ({                     })
                                {<integer constant>}
```

for which the rules given in section 6.1.5 apply.

The resulting adjustable zone array is established at entry into the program unit according to the rules for other adjustable arrays (see section 6.1.5).

The first element of the formal zone array will match the first element of the actual zone array.

6.1.7 END Statement

Program units are terminated by the statement

```
END
```

6.1.8 RETURN Statement

Return of control from a subroutine or a function is performed by passing a statement of the form:

```
RETURN
```

When control passes the END statement an implicit RETURN is executed.

6.1.9 ENTRY Statement

A program unit may have several different entries with separate names. An entry is located in the program unit by a statement of the form:

```
ENTRY <entry name>
```

All entries must have the same type and are assumed to have the same set of formal parameters as the main entry declared in the SUBROUTINE or FUNCTION statement.

6.2 COMMON and DATA

6.2.1 COMMON

Program units may share data areas by means of the COMMON facility. The COMMON statement has the form

```
       {                  1                  }*
COMMON {/ {<common name>}    / <common list>}
       {                  0                  }1


<common list> ::=

     [<variable name>                          ]*
     [              {                    *   }1]
     [<array name>  { ([<constant bound>] )  } ]1
                    {                    1   }0
```

The COMMON statement declares one or more COMMON blocks identified by a common name and containing the variables listed in the common list. Within a program the COMMON block is accessible to all program units which contain a declaration of that specific COMMON block. The declaration of a COMMON block within different program units must agree with respect to the length of the COMMON block but the name and type of the variables in the common list may vary from one program unit to another. If the name of the COMMON block is omitted the variables are stored in a COMMON block called BLANK COMMON. The treatment of this COMMON block is exactly as for other COMMON blocks.

Note that in RC FORTRAN an integer occupies one word but a real occupies two words. This may affect the COMMON correspondance as shown in the following example.

Example 6.2.1

A COMMON block, c1, is declared in two different ways in the program units A and B as follows:

```
Program unit A
        common/c1/p,q,r(2)
        integer p,r; real q
Program unit B
        common/c1/dummy(3),r(2)
        integer dummy, r
```

The two different declarations will result in storage allocations as shown below:

```
word boundaries   |1 | 2 | 3 | 4 | 5|
unit A            |p |   q   |   r  |
unit B            |  dummy    |   r  |
```

The array r corresponds to the same storage locations in program unit A and B. However, if dummy is not declared as integer this will not happen.

6.2.2 Local Variables versus COMMON Variables

The variables not declared in COMMON are called local variables. There are some important differences in the treatment of these two classes of variables.

COMMON areas are static and thus all COMMON variables permanently occupy core store during the run. COMMON variables may be initiated by DATA statements in any program unit. The storage allocation for COMMON variables reflect the order of declaration, i.e. variables are allocated contiguously in the order in which they occur in the COMMON declarations.

Local variables are dynamic. Core area is reserved and allocated to local variables and arrays at entry into a proram unit, and the variables are deleted by releasing the core at exit from the program unit. Thus local variables occupy core only as long as they are active, that is during execution of the program unit in which they are declared. As a consequence local variables may not be initiated by DATA statement, and the values of local variables are always undefined at entry into the program unit in which they are declared.

6.2.3 Zones in COMMON

A zone may be declared in COMMON, but a COMMON block may only contain one zone or one zone array. Zones and other kinds may not be in the same COMMON block. Only the name of the zone may be written in the common list while the remaining declaration should be given in the normal ZONE statement. The zone declaration for a zone in COMMON must be identical in all program units having declared the zone.

6.2.4 DATA Statement

Form

```
     [                                  ] *
DATA [ <variable list>/<value list>/ ]
     [                                  ] 1
```

```
<variable list> ::=

          [ <variable name>                    ] *
          [ <array name>                       ]
          [                [             ] *   ]
          [ <array name>([ <integer> ]     ) ] 1
                           [             ] 1

<value list> ::=

          [ <constant>                         ] *
          [ <extended text>                    ]
          [ <repeating integer>*<constant>     ] 1
```

Rules

1. Single variables or whole arrays may be initiated by DATA statements.

2. A sequence of identical values may be specified by writing a repeating integer and a * before the value to be repeated. The repetition integer must be a positive constant.

3. The number of values in the value list (including repetitions) must match the number of variables in the variable list.

4. If the type of a variable and its corresponding value differ the constant is converted according to the rules for assignments (see Section 4.1).

5. Only variables in COMMON may be initiated.

BLOCK DATA

As DATA initiation may be specified within any program unit, the special program unit BLOCK DATA is not implemented in RC FORTRAN. Existing programs may be modified for RC FORTRAN by (1) replacing the BLOCK DATA statement by a subroutine heading, and (2) declaring the subroutine name as EXTERNAL in the main program.

Extended Text Constants in DATA Statements

In DATA statements it is allowed to write text constants of more than six characters. The extended text string is broken down into short text constants by the compiler, and the last short constant is filled up with NULs if necessary.

Example 6.2.2

```
long errtxt(4)
common/c/errtxt
data errtxt/'key missing in record'/
```

The text is stored six characters per word in the array errtxt, as the number of characters is not a multiple of six, NUL characters are added to fill the last element.

6.3 Program Units from the Catalog

A catalog is maintained by the RC operating system. Compilers and assemblers may insert catalog entries describing program units, which are compiled seperately for later use within a main program. An RC FORTRAN program may include catalog externals programmed in FORTRAN, Algol, or in assembly language. Non-FORTRAN program units must obey certain rules as described below.

6.3.1 Algol Externals

An Algol program unit must be programmed according to Ref. 2, and it must be compiled separately by the Algol compiler to be established as an external described in the catalog. It may then be included in a FORTRAN program nearly as a FORTRAN program unit. The following special rules apply:

1. Parameters of Algol types string, long and label must not be used.

2. In FORTRAN arrays are stored column after column, according to ISO. In Algol arrays are stored row after row.

3. FORTRAN routines will not treat name parameters correctly. This may cause unpredictable reactions, if a FORTRAN routine is given as actual parameter for an Algol external.

4. The Algol procedure may not use field-variables or longs, nor may it use any features from ALGOL 7.

The library procedures of the Algol library may also be used within the FORTRAN system. However, the Algol procedures read and write may not be referenced from a FORTRAN program as these names are reserved for FORTRAN READ/WRITE. Descriptions of the Algol library procedures are found in Ref. 2.

6.3.2 Program Units in Machine Language

Program units programmed in machine language may be included in a FORTRAN program. The program units must be assembled by the Slang assembler for RC4000, RC6000 or RC8000, and described as catalog externals. Detailed conventions for a machine coded procedure are given in Ref. 4. However, the use of this possibility should be limited as far as possible to maintain the high reliability given by the Algol and FORTRAN systems.

Appendix A. References

Part numbers in references are subject to change as new editions are issued and are listed as an identification aid only. To order, use package number.

1) *System Utility,*
User's Guide, Part One PN: 991 11263
User's Guide, Part Two PN: 991 11264
Included in SW8010i-D, System Utility Manual Set
System 3 Utility Programs PN: 991 11294
Part of Assembler Manual Set, in SW8585-D

2) *ALGOL8*
Reference Manual PN: 991 11278
User's Guide, Part 1 PN: 991 11279
User's Guide, Part 2 PN: 991 11280
Part of ALGOL Manual Set, in SW8585-D

3) *Magnetic Tape System*, (obsolete) PN: 991 03278

4) *Code Procedures and Run Time Organization of ALGOL 5 Programs* PN: 991 11296

5) ISO: *R646-1967(E), 6 and 7 bit coded character for information processing interchange*

6) ISO: *DR 1539, Programming Language FORTRAN*

7) *BOSS, User's Guide* PN: 991 11274
Part of SW8101i-D, BOSS Manual Set

8) *Definition of External Processes*, (obsolete) PN: 991 03392

9) *ISQ File System* PN: 991 11286
Part of Backing Storage,
in SW8585i-D Compiler Collection Manual Set

10) *RC9000-10 System Software* PN: 991 11255
Delivered with SW9910i-D

Appendix B
RC FORTRAN Syntax Description

B.1 Explanation

RC FORTRAN is described using the Backus-Naur Form expanded with a choice bracket and a list bracket. Both brackets can be supplied with repetition numbers. The definition is:

```
{ <construction 1> }
{ ...              } ::= <construction 1> /.../ <construction N>
{ <construction N> }

{               } b                    {               } b-1
{ <construction> }    ::= <construction> { <construction> }
{               } a                    {               } a-1

[               ] b                    {                } b-1
[ <construction> ]    ::= <construction> { ,<construction> }
[               ] a                    {               } a-1
```

where a is the minimum number of repetitions and b is the maximum number of repetitions. a = 0 means empty, b = * means any number of repetitions, and if a and b are not given a = b = 1 is understood.

The following abbreviations are used:

`arith`	for arithmetic
`decl`	for declaration
`expr`	for expression
`mask`	for masking

Limitations in the use of names are given by supplying the kind. Limitations in expressions are given by adding the type, and here arith(metic) includes integer, long, real, double precision, and complex; type long includes integer, and type mask is an auxiliary type used in connection with masking operations but the internal treatment is as integer, long, or real (further explained in Section 3.3 Arithmetical and Masking Expressions).

B.2 Symbols and Primitives

```
<symbol>                ::=  { <letter>              }
                             { <digit>               }
                             { <separator>           }
                             { <terminator>          }
                             { <graphic>             }
                             { <arithmetical operator> }
                             { <blind>               }
                             { <in text>             }

<blind>                 ::=  NUL | DEL

<in text>               ::=  SP  | _

<letter>                ::=  a | b | ... z | æ | ø | å | A ... Å

<digit>                 ::=  0 | 1 | ... | 9

<separator>             ::=  ( | ) | , | . | = | '

<terminator>            ::=  ; | $ | NL | FF

<graphic>               ::=  ! | " | £ | % | & | : | < | > | ? | @ | ^ | `

<arithmetical
 operator>              ::=  + | - | * | / | **

<relational
 operator>              ::=  .LT. | .LE. | .EQ. | .NE. | .GE. | .GT.

<logical operator>      ::=  .NOT. | .AND. | .OR.

<masking operator>      ::=  <logical operator> | .SHIFT

<type>                  ::=  { INTEGER          }
                             { LONG             }
                             { REAL             }
                             { DOUBLE PRECISION }
                             { COMPLEX          }
                             { LOGICAL          }

<name>                  ::=  <letter> { <letter> } *
                                      { <digit>  } 0

<sign>                  ::=  { + } 1
                             { - } 0

                                       *
<digits>                ::=  { <digit> }
                                       1

<integer>               ::=  <digits>

<long>                  ::=  <digits>

<basic real>            ::=  { <digits>.<digits>          }
```

```
                                { <digits>.                 }
                                {                 .<digits> }

<real>                  ::=     { <basic real>                      }
                                { ( <basic real> ) E<sign><integer> }
                                { ( <integer>    )                  }

<double>                ::=     { <basic real> ) D<sign><integer>
                                { <integer>    )

<complex>               ::=     (<sign><real>,<sign><real>)

<number>                ::=     <integer> | <real> | <double> | <complex>

<logical>               ::=     { .TRUE.  }
                                { .FALSE. }

                                                          n
<text>                  ::=     { <integer n> H ( <symbol> )  }
                                {                         n }
                                {               *           }
                                {  ' ( <symbol> ) '         }
                                                0

                                            48
<bitpattern>            ::=     { 1B ( 0 | 1 )                          }
                                {            1                          }
                                {                    24                 }
                                { 2B ( 0 | 1 | 2 | 3 )                  }
                                {                    1                  }
                                {                                   16  }
                                { 3B ( 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 )  }
                                {                                    1  }
                                {                                   12  }
                                { 4B ( <digit> A | B | C | D | E | F )   }
                                                                     1

<constant>              ::=     { <sign><number> }
                                { <logical>      }
                                { <text>         }
                                { <bitpattern>   }

                                {        }5
<label>                 ::=     { <digit> }
                                {        }1

                                { <variable name>   }
                                { <array name>      }
<formal>                ::=     { <zone name>       }
                                { <zone array name> }
                                { <procedure name>  }

                                { <expr>            }
                                { <array name>      }
<parameter>             ::=     { <zone parameter>  }
                                { <zone array name> }
                                { <procedure name>  }
```

```
<zone parameter>          ::=  { <zone name>                      }
                               { <zone array name>(<arith expr>) }

                                            [             ] *
<array element>           ::=  <array name> ([ <arith expr> ]  )
                                            [             ] 1

<zone element>            ::=  { <zone name>(<arith expr>)                    }
                               { <zone array name>(<arith expr>,<arith expr>) }
```

B.3 Declarations

```
<program decl>                  ::=   PROGRAM <name>

                                      {                    [       ]*   1     }
<procedure decl>                ::=   { SUBROUTINE <name> { ([<formal>] ) }      }
                                      {                    [       ]1   0     }
                                      {          1                [       ]*  }
                                      { { <type> }  FUNCTION <name>([<formal>] ) }
                                                 0                [       ]1

<common decl>                   ::=
COMMON     [{/<name>/}  [ <variable name>                  ]* ]*
           [{       }  [                [       ]*   1 ]  ]
           [{   //   }  [ <array name> { ([<integer>] ) }  ]1 ]
           [                             [       ]1   0    ]
           [                                               ]
           [{/<name>/}    <zone name>                      ]
           [{       }                                  1  ]
           [{   //   }    <zone array name> { (<integer>) }   ]1
                                                         0

<type decl>                     ::=   <type> [ { <name>                          } ]*
                                             [ { <array name>([ <integer>     ] *) } ]
                                             [ {              [ <integer name>] 1  } ]1

<dimension decl>                ::=
DIMENSION          [ <array name>( [ <integer>      ]* )     ]*
                   [                [ <integer name> ]1       ]
                   [                                          ]
                   [ <zone array name>( { <integer>      } ) ]
                   [                    { <integer name> }   ]1

<zone decl>          ::=
       [                                                 1 ]*
ZONE   [ <name> { (<integer>,<integer>,<procedure name>) }  ]
       [                                                 0 ]1

<external decl>                 ::=   EXTERNAL [ <procedure name> ]*
                                              [ <variable name>  ]
                                              [ <zone name>      ]1

<equivalence decl> ::=
EQUIVALENCE        [ ( [ <variable name>                      ]*) ]*
                   [   [               [          ]*          ]   ]
                   [   [ <array name> ([ <integer> ]1)        ]   ]
                   [   [                                      ]   ]
                   [   [ <zone name> ( <integer> )            ]   ]
                   [   [ <zone array name>(<integer>,<integer>) ]2  ]1

<initiation>       ::=
       [                                                          ]*
       [[<variable name>            ]*   [{         }1         ]*   ]
DATA   [[<array name> [         ]*  ] /  [{<integer>*) <constant>]  / ]
       [[<array name>([<integer>]1) ]1   [{         }0         ]1   ]
       [                                                          ]1
```

B.4 Expressions

```
<arith operand>                  ::=   { <number>                                }
                                       { <arith variable name>                   }
                                       { <arith array element>                   }
                                       { <zone element>                          }
                                       {                        [          ]*    }
                                       { <arith function name>([<parameter>] )   }
                                       { <mask operand>         [          ]1    }
                                       { (<arith expr>)                          }

<arith expr> ::=
                                                                 *
<sign><arith operand> { <arith operator><arith operand> }
                                                                 0

<mask operand> ::=

{ { <integer operand> } .SHIFT.<integer operand> }
{ { <long operand>    }                          }
{ { <real operand>    }                          }
{ { <mask operand>    }                          }
{                                                }
{ (<mask expr>)                                  }

<mask expr> ::=
{     }1                              {     }1
{.NOT.}0 {<integer expr> } {{.AND.} {.NOT.}0 {<integer expr>}}
         { <long expr>   } {{.OR. }          {<long expr>   }}
         { <real expr>   } {                 {<real expr>   }}
         { <mask operand>} {                 {<mask operand>}}

<logical operand>                ::=   { <logical>                                    }
                                       { <logical variable name>                     }
                                       { <logical array element>                     }
                                       {                                             }
                                       {                            [          ]*    }
                                       { <logical function name> ( [<parameter>]1 )  }
                                       { <arith expr><relational operator><arith expr>}
                                       { (<logical expr>)                            }

<logical expr> ::=
{      } 1                {       {     } 1                 } *
{ .NOT.}   <logical operand>{ {.AND.}{.NOT.} 0 <logical operand> }
{      } 0                { {.OR. }                          } 0
```

B.5 Executable Statements

```
<simple statement>              ::=  { <variable assignment>   }
                                     { <label assignment>      }
                                     { <jump statement>        }
                                     { <conditional statement> }
                                     { <empty statement>       }
                                     { <procedure statement>   }
                                     { <transfer statement>    }
                                     { <inout statement>

<statement>                     ::=  { <simple statement> }
                                     { <loop statement>   }

<variable assignment>           ::=  { <variable name> } = } * <expr>
                                     { <array element> }   }
                                     { <zone element>  }   }
                                     { <function name> }   } 1

<label assignment>              ::=  ASSIGN <statement label> TO <label name>

<jump statement> ::=
GOTO   { <statement label>                        }
       {                  [                ]*   1 }
       { <label name> { ,([<statement label>] ) }  }
       {                  [                ]1   0 }
       {   [                ]*                    }
       { ( [<statement label>]  ),<arith expr>    }
           [                ]1

                                     {          }1
<empty statement>               ::=  { CONTINUE }
                                     {          }0

                                                                {   [           ]*  }1
<procedure statement>             ::=  CALL <procedure name> { ( [<parameter>] ) }
                                                                {   [           ]1  }0

<transfer statement>            ::=  { RETURN             }
                                     {                  1 }
                                     { STOP { <integer> }  }
                                                        0

<conditional statement> ::=
{                    [                 ]3 }
{ IF ( <arith expr> ) [ <statement label> ]3 }
{ IF ( <logical expr> ) <simple statement>   }

<loop statement> ::=
                                    [           ]3
DO <statement label> <integer name> = [<arith expr>]
5                                   [           ]2

                                        {          }*
<end statement>                 ::=  END { <symbol> }
                                        {          }0
```

B.6 Input/Output Statements

```
<inout statement> ::=
{ READ  }                {                ) 1  [    ]*
{       } (<logical unit> { ,<format label> )  ) [<put>]
{ WRITE }                {                ) 0  [    ]0

<logical unit>                    ::=   { <zone parameter> }
                                        {  <arith expr>    }

<put> ::=
{ <expr> but in READ limited to { <variable name> } }
{                               { <array element> } }
{                               { <zone element>  } }
{ <array name>                                      }
{ <zone parameter>                                  }
{    [    ]*                                        }
{ (  [<put>]1 )                                     }
{                                                   }
{    [    ]*                    [          ]3     }
{ (  [<put>]1 , <integer name> = [<arith expr>]2 )  }

                                                    {               } 1
<format decl>                     ::=   { FORMAT  } ( { <format field> } 0 )
                                        { FORMATO }

<format field> ::=
{ {{   }1          } 1 {        } 1                      }
{ {{ - }0 <integer>P} 0 {<integer>} 0 {F<integer>.<integer>} }
{                                     {E<integer>.<integer>} }
{                                     {D<integer>.<integer>} }
{ {         }1                                             }
{ {<integer> }0   { I <integer>          }                   }
{                 { I <integer>.<integer> }                  }
{                 { L <integer>          }                   }
{                 { A <integer>          }                   }
{                 { B <integer>.<integer> }                  }
{                 { X                    }                   }
{                                                           }
{   <text>                                                  }
{   /                                                       }
{   [{   } * [              ] * {   }* ]*                   }
{   [{ / } 0 [ <format field> ] 1 { / }0 ]1                 }
{            1                                              }
{ ( <integer>)   (<format field>)                           }
{            0                                              }
```

B.7 Program Structure

```
<program unit>                  ::=
                                    {                   1  }
                                    { { <program decl> }   }
                                    {                   0  }
                                    {                      }
                                    {  <procedure decl>    }

                                    {  <common decl>       }*
                                    {  <type decl>         }
                                    {  <dimension decl>    }
                                    {  <zone decl>         }
                                    {  <external decl>     }
                                    {  <format decl>       }
                                    {  <initiation>        }0

                                    {                      }*
                                    {  <equivalence decl>  }
                                    {                      }0

                                    {  <statement>         }*
                                    {  <format decl>       }
                                    {  ENTRY <name>        }0

                                       <end statement>
```

Appendix C. Call of Compiler

The compiler is activated by an fp-command of the following form:

```
<bs file> = fortran { <text file> } *
                    { <modifier>  } 0


<modifier> ::=
{ { index    } . { yes }                                          }
{ { spill    }   { no  }                                          }
{ { trunc    }                                                    }
{ { list     }                                                    }
{ { names    }                                                    }
{ { message  }                                                    }
{ { survey   }                                                    }
{ { testin   }                                                    }
{ { cond     }                                                    }
{ { cardmode }                                                    }
{                                                                 }
{   stop   .{ yes          }                                      }
{           { no           }                                      }
{           { <last pass>  }                                      }
{                                                                 }
{   details .{ yes                                              } }
{           { no                                                } }
{           {                                                 1 } }
{           { <first pass>.<last pass> {.<first line>.<last line> } } }
{                                                             0 } }
{       {          } 1                                          }
{   test { <letter> }   . { yes }                                }
{       {          } 0   { no  }                                 }
```

<bs file>
A file descriptor describing a backing storage area. The area is used as working area for the compilation, and for the object code. The file descriptor is then changed to describe a program unit.

The RC FORTRAN compiler will work in two different modes:

1. A source text containing a single subroutine or function may be compiled into a binary program unit. The object is changed to

describe an algol/fortran subroutine. For each entry point in the compiled subroutine/ function a shared entry is created describing the entry point and referring to the name of the object file. The name given in the subroutine/function statement is lost and thus irrelevant.

2. A source text containing a number of subroutines and/or functions and one main program may be compiled into an executable program. The object file is changed to describe an executable program. The name of the program is lost and thus irrelevant.

<text file>
A text file is an fp text file or a text ending with the EM or HT-character. The list of text files specifies the wanted order of input files to the compiler. If no source is specified, the compiler reads the source from current input.

Notice: Input of sourcetext is terminated when a line, starting with a slash in position 1, is encountered. The line is listed as a message line.

<modifier>
The list of modifiers is scanned from left to right. Each modifier changes the variables controlling the compilation. When the scan starts, the variables are initialized to the value explained below.

index.no
Code for dynamic check of subscripts against bounds is omitted. Initial setting: index.yes.

trunc.no
Conversion from real (and double precision) to integer (and long) yields a rounded result. Otherwise a truncation will be performed (see section 4.1). Initial setting: trunc.yes.

spill.yes
Dynamic check of integer overflow is performed. Even if the external procedures referenced were translated with spill.no, a partial check of integer overflow is performed when they are executed. Initial setting: spill.no.

list.yes
The entire source text is listed on current output with line numbers in front of each line. Initial setting: list.no.

names.yes
A primitive crossreference listing is printed giving for each program unit:

list of entry point names
list of external names
list of common names and the size (in halfwords) of each common
list of zone common names and the size (in halfwords) of each common
the number of halfwords used for local variables.

Initial setting: names.no.

message.no
Normally all message lines in the source text (i.e. lines starting with an M as first symbol, see section 1.2) are listed with line numbers. With 'message.no' this listing is omitted. Initial setting: message.yes.

survey.yes
A summary is printed on current output after the completion of each pass of the translation. The meaning of the summary is explained in ref. 4. Initial setting: survey.no.

stop.<last pass>
The translation is terminated after the pass specified. stop.yes terminates the translation after the last pass. The translation is regarded as unsuccessful. Initial setting: stop.no

details.yes
Intermediate output from all passes of the compiler is printed on current output. The output may be restricted to an interval of pass numbers and to an interval of line numbers. Initial setting: details.no.

testin.yes
Works as details.yes, but prints the intermediate compiler data as it is input by the passes. The option is blind in case of details.no. Initial setting: testin.no.

cond.yes
A listing is performed, starting with a line with the letter L in the first position and continuing to the first end statement. Initital setting: cond.no.

cardmode.yes
Only the first 72 characters of a source line are treated as fortran text. Exceeding characters are skipped. Initial setting: cardmode.no.

test.yes
Lines with one of the possible <letter>-values in the first position are procecssed as program lines instead of being comment lines. Initial setting: test.no.

test<letter>.yes
Lines with <letter> in the first position are processed as program lines instead of being comment lines. Initial setting: test<letter>.no.

Requirements of the compiler

The compiler occupies about 80 segments on the backing store. A process size of 11000 halfwords is necessary to run a compilation. The compilation of large programs usually requires a process size, which is one or a few thousand halfwords larger. If the available core area is too small the compilation will terminate with an error message.

Appendix D. Messages from the Compiler

The messages from the compiler may concern the call of the compiler or it may be diagnostics produced during processing of the source text. The first kind of messages have the form

***** fortran <text>**

and are followed by termination of the compilation.

Other messages may have one of three forms:

Form 1: **<pass no> line <line no>. <operand no> <text> <aux1> <aux2>**

Form 2: **<pass no> line <line no> <text>**

Form 3: **<pass no> <name> <text>**

pass no
is the number of the current compiler pass.

line no
is the number of the relevant line of the source text. The first line has number 1; only lines containing visible symbols are counted.

operand no
indicates the position within the line, where the error is found. An operand is a name or a constant. Operands are counted from the left starting with 1.

text
is a short description of the error.

aux1, aux2
are auxiliary values helping to describe the error as specified in the list below.

name
with form 3: the name given is the name of a program unit or common block connected with the error.

The pass no is only given with the first error line. The operand number may be irrelevant with some of the messages which are marked by (NB) in the list. Aux1 and aux2 are often omitted.

In the following list the messages are sorted according to <text> and they are classified as:

(alarm)
The translation is terminated as an unsuccessful execution. The program cannot be executed.

(warning)
The message has no effect.

Otherwise the translation continues and the program may be executed until the erroneous construction is met.

ftn. end <i>
This is not an error message. The fortran program has been translated. The ok-bit is set to yes. The warning-bit is set to no if no error messages have occured, otherwise it is set to yes. The object code occupies <i> segments.

fortran sorry <i>
An alarm has occured. The ok-bit is set to no. The integer <i> shows the number of segments the compiler has attempted to make.

***** fortran <text>**
The compilation is stopped and the ok-bit is set to no.

adjustable bound
Declarator subscript should be integer constant.

bitpattern
Three types of error may occur. The type is given in aux1 and further information in aux2 as follows:

aux 1	description	aux2
1	illegal bit group size	bit group size
2	illegal digit	illegal digit
3	overflow	digit in process

call
A procedure call has a wrong number of parameters.

<name> catalog
<result>
Trouble with catalog lookup, for instance because a standard identifier is missing in the catalog. The name of the identifier and the lookup result is printed (alarm).

<name> commo
Inconsistent common declaration, for instance a common declared with different length in different program units. The name of the common is printed (alarm).

common error
(NB) Mixing of zones and other variables in common, two zones in the same common, or a variable is two times in common.

constant index
During compilation a subscript is outside the declared range.

constant outside allowed range
The value of a real constant, a double precision constant or the real or imaginary part of a complex constant is outside the allowed range, i.e.,

```
1.547174*10**(-617) <= abs (value)
                    <= 1.615850*10**(616)
```

continuation mark on a labelled line <char in position 6>
A label and a continuation mark is not allowed on the same line.

+declaration
Identifier declared twice or more times.

dimension equivalenced common variable
Start of dimension outside lower bound of common area, equivalence illegal.

dimension equivalenced zone
Start of dimension outside zone record, equivalence illegal.

do after if
A conditional statement must not be a do statement.

do construction
Illegal number of control parameters in do or implied do.

entry name
Entry name should be simple local.

equivalence impossible
Two variables cannot be equivalenced because they are placed in common areas, maybe by previous equivalence statements.

equivalence index
Subscription missing for array or zone trouble identifier, or subscription on a simple variable in an equivalence statement.

equivalence
Index error for array or zone in an equivalence subscript statement.

erroneous terminated do range
The terminating statement is not situated after the do statement and before the end statement or before the terminating statement of an outer do statement.

exponent too big <converted value>
A preliminary conversion has shown that the exponent value is outside the range:

```
-1000 <exponent < 1000
```

external zone not in catalog
(NB) One of the zone identifiers, declared as external, is not present in the catalog. It can be any of the external zones if more than one is declared.

formal in common
A parameter is declared as a common variable.

format error before comma no. <comma no.>
(NB) Illegal structure or illegal field specification values has been detected in a format statement.

graphic <char value>
The character with decimal value <char value> is illegal in this context.

<name> kind
This alarm may occur for two different reasons:

1. If the use of a name from the catalog does not match the catalog description of the name. Such error may occur when a subroutine or function is compiled without its externals being present in the catalog (see Section 6.3), or if the catalog description is changed after being used in the compilation.

2. If a common declaration from a catalog external differs from the common declaration in the program unit under compilation (alarm).

illegal <char value> <line no.>
The character is of class illegal, see Table 1 in the manual. It is marked with a question mark in a listing and does not count on the line.

illegal number of main programs
A complete source text must be either one single program unit or a number of program units with one main program (alarm).

label
A label is not declared or multiple declared.

labelling error
Missing or misplaced label.

label not referred
A declared but not referred label (warning).

label syntax <error char value>
The character pointed out is not allowed in a label field.

list structure
Illegal element type or illegal separator in a list.

missing)
One or more right parenthesis are missing.

missing end
An EM-char is read within a program unit. An end statement is generated and the compiler goes on.

more actions
More actions in action stack at statement end (compiler error).

<name> name trouble
The object area is not renamed because the program name already exists in the catalog.

non-common element, rightmost group no. <group no.>
(NB) A simple variable or an array of the data group in the data statement is not in common.

no. of subscripts illegal
Number of subscripts in an array declaration must be < 32.

no. of zones illegal
Number of zones in a zone array must be < 32.

not implemented
Action not implemented (compiler error).

operand stack
Operands in stack at statement end (compiler error, may occur after a fault in a state- ment).

overflow
During compilation an expression holding constants causes arithmetical overflow.

param
Illegal parameter in the FP-command, calling the compiler. The parameter is ignored (Warning).

pass trouble
The job area is too small to load the next pass of the compiler or the next pass has been destroyed. (Alarm).

program too big
The backing storage area specified cannot hold the object code. (Alarm).

run stack full
A program unit has too many variables and/or uses too many working variables.

short text <no. of chars processed>
For hollerith constants the following condition is not satisfied:

```
1 <= no. of characters <= 6
```

statement sequence
The statement sequence is wrong within a program unit, e.g., declarative statement after executable statement.
Note: equivalences must be placed after all other declarations.

statement structure
The main structure of a statement is wrong.

subscripts
A subscripted variable has a wrong number of subscripts.

syntax error
Syntactical error within a subconstruction of a statement.

too many significant digits <no. of digits converted>
Leading zeroes do not count, and one of the conditions below is not satisfied:

```
long constant:                abs(value)<2**47
double precision constant:    no. digits<20
```

Note that a floating point number with no. digits > 11 is always a double precision constant.

top not in w1
The operand stack pointer is wrong (compiler error, but the program will run anyway).

type
The declaration or type of an operand is not in accordance with its use.

unassigned elements, rightmost group no. <group no.>
(NB) The number of constants does not equal the number of elements in the data group of the data statement.

<name> unkn.
The external with the specified name did not exist in the main catalog. (Alarm).

wrong standard action
Error in the action tables of pass 7 compiler error).

zone
Wrong number of subscripts after zone or zone array.

zone specification
The zone declaration must specify buffersize, number of shares and block procedure except for formal zones.

Appendix E. Program Execution

E.1 Execution of an RC FORTRAN Program

A compiled RC FORTRAN program may be executed by a file processor command of the form

```
<bs file> { <empty>             }
          { <source><anything>  }
          { <integer>           }
          { <param><anything>   }

            <param> ::= { <integer> }.{ <integer> }
                        { <name>    } { <name>    }
```

<bs file>
A file descriptor describing a backing storage area which contains the compiled FORTRAN program.

<empty>
The program is called with the current input file of the file processor connected to IN.

<source>
A file descriptor. The program is called with the described file connected to IN. The current input file of the file processor is untouched.

<integer>
The program cannot use IN and OUT and it cannot print error messages. When the program terminates, it sends a parent message corresponding to a ,break, and specifying the cause of the termination.

On the other hand, 3000 - 4000 halfwords more are available in this way. The possibility is mainly intended for operating systems, which "never" are terminated, never use IN and OUT, and work satisfactorily in a very short core area.

<param>
Works as <empty>. The commmand parameters <param> may be

accessed from the running program by means of the procedure 'system' (see Ref. 2).

<anything>
As <param>.

Example:

```
progxx = set 50
progxx = fortran progtxt list.yes message.no
progxx dataxx
```

The program text found in the file 'progtxt' is compiled into the object file 'progxx'. The program is then executed with the file 'dataxx' connected to the zone IN.

Example
(corresponding to ref. 7, introduction):

```
progyy = fortran                ; read from current input
       program test
       zone in, out
       external in, out
       read (in, 1) a, b
1      format (f1.0, x, f2.0)
       write (out, 2) a**b
2      format (x, f6.0)
       end
/this is the last line
progyy                          ; call program
2  10                           ; with this line as data
```

The output from these call will look like:

```
    9/this is the last line
ftn. end 22
  1024
end    15
```

E.2 Run Time Alarms

E.2.1 Initial Alarm

Before the program is entered, the alarm

```
*** <program name> call
```

may appear. It is due to either: the program is not on backing store, the source is not a text, or the job process is too short.

E.2.2 Normal Form

When the program is called with <program> <integer>, a run time alarm appears as a parent message (see Ref. 2).

In normal case, a run time alarm terminates the program with a message of the form:

```
<cause>     <alarm address>
called from <alarm address>
called from ...
```

A list of the possible alarm causes is given in E.2.4. The program is terminated unsuccessfully, except after the message 'end'. An alarm address shows where the error occurred. If this is a procedure or a name parameter, a line specifying the call address or the point where the name parameter was referenced is also printed. The process is repeated if several calls or references were active at the time of the alarm. If more than 10 calls or references are active, the process stops after having printed the last 'called from' but before the last alarm address is printed.

An alarm address may take 3 forms:

1. name of a standard procedure or a set of standard procedures
2. line <first line> - <last line>
3. ext <first line> - <last line>

Form 2 specifies a line interval in the source text of the main program.

Form 3 specifies a line interval in an external procedure. The accuracy of a line interval corresponds to about 16 instructions of generated code. The first line number may sometimes be 1 too large. The line number of a procedure call points to the end of the parenthesis.

The following alarm addresses from library procedures are used:

char input
(read, readall, readchar, readstring, repeatchar, intable), (see Ref. 2)

check
(all high level zone procedures use the procedure check)

complex op
(complex or long arithmetic)

fortranfct.
(iabs, abs, amod, mod, min0, max0, amin0, amax0, min1, max1, amin1, amax1)

ftn io
(fortran read/write)

open
(open)

position
(getposition, setposition, close)

readon
(fortran read)

recprocs
(inrec, outrec, swoprec)

rl convert
(conversion between real and long, DATA initiation)

stand.fct. 1
(exp, alog, sinh)

stand.fct. 2
(atan, arg, sin, cos)

stand.fct. 3
(arcsin, sqrt)

writeon
(fortran write)

zone declar
(the code that declares zones and zone arrays)

zone share
(getzone, getshare, setzone, setshare), (see Ref. 2)

E.2.3 Undetected Errors

If all parts of a program have been translated with both index.yes and spill.yes, the following errors may still pass undetected:

1. Parameters in the call of a procedure which is a formal parameter do not match the declaration of the corresponding actual procedure. Any reaction may result.

2. A subscript may exceed the bounds in an array declaration with more dimensions as long as the lexicographical index is inside its bounds. The control of the program remains intact.

3. The program may write into the backing storage area occupied by the program itself. Any reaction may result.

4. Undebugged standard procedures in machine language may cause any reaction.

5. Parameters, beyond the seventh parameter, in the call of a procedure do not match the declaration of the procedure. Any reaction may result.

The monitor and the operating system will usually limit the consequences of errors in such a way that no other job or process in the computer can be harmed.

E.2.4 Alphabetic List of Alarm Causes

The error messages below cover only the standard procedures described in this manual. The set of messages is expected to grow with the standard procedure library.

arcsin 0
Illegal argument to arcsin.

block <i>
Too long record or record with a negative length in call of inrec, outrec, or swoprec. The block length is shown.

break <i>
An internal interrupt is detected. <i> is the cause of interrupt, usually meaning:

0 illegal long text string as parameter,
6 too many message buffers used (see Ref. 1),
8 program broken by the parent, often because it is looping endlessly. In this case, the alarm address should be taken with some reservation. The break alarm will often be called as a result of the undetected errors described in E.2.3.

end <i>
The program has passed the final end. The integer printed after end shows the value of the standard integer BLOCKSREAD as the program terminated (see appendix E.3). This is not an error message.

entry <i>
Illegal function code or entry conditions in a call of monitor, system, or systime (see Ref. 2). The function code attempted is shown.

exp 0
Illegal argument to exp.

giveup <i>
Printed by stderror. The number of halfwords transferred is shown. The file processor prints the name of the document and the logical status word.

index <i>
Subscript outside bounds. The lexicographical index is shown. This message occurs also for subscripted zones or record variables.
The index alarm is called if a block procedure specifies a too long block. In this case, the value of the block length is shown.

integer
Integer overflow.

label
Attempt to goto an unassigned label variable.

length <i>
Illegal record length in call of inrec, outrec, or swoprec. The attempted length is shown.

ln 0
Argument to alog < = 0.

not unit <i>
An unassigned unitnumber is used. The attempted unitnumber is shown.

modekind <i>
Illegal modekind in call of open. The kind is shown.

param
Wrong type or kind of a parameter.

real
Floating point overflow or underflow.

share <i>
An illegal share number is specified. The number attempted is shown.

sh.state <i>
A share in an illegal state is specified. The share state is shown.

sinh 0
Illegal argument to sinh.

sqrt 0
Argument to sqrt is < 0.

stack <i>
The number of variables exceeds the capacity of the job area, or an array or a zone is declared with a nonpositive number of elements. The number of halfwords attempted in the reservation of storage is shown.

syntax
The program is terminated at a point where an error was detected during the translation.

uncoded <i>
Format in array is not implemented. (The number shown is irrelevant).

value <i>
The contents of ia(i) in setzone(z,ia) or setshare(z,ia,sh) is illegal. The value of i is shown (see Ref. 2).

z.kind
Swoprec is not used on a backing storage area.

z.length <i>
The buffer length is too short. The actual buffer length is shown.

z.state <i>
A high level zone procedure is called in an illegal zone state. The actual state is shown.

The value of the zone state is determined by the latest call of the standard procedures using the zone. The most important values are given below. Other values may be defined together with other zone based procedures and are then given in the procedure descriptions.

zone state	situation
0	For magnetic tapes: after setposition; for other documents: after open or setposition.
1	After reading on character level.
2	After repeat char (algol, see Ref. 2
3	After writing on character level.
4	Just after declaration of the zone.
5	After inrec, unformatted read.
6	After outrec, unformatted write.
7	After swoprec.
8	After open on magnetic tape.

z.units
Too many zones are assigned units numbers at one time. The maximum number is shown.

E.3 The Object Code

The object code is partitioned in segments of 128 double words each. If a single subroutine or function is compiled, the object code consists of a number of segments corresponding to the source text. If a main program is compiled, the compiler will add a few segments called the running system. The running syste routines for segment transfer, core reservation, alarm administration, basic input/output, etc.

Further the compiler will add to the program all segments corresponding to external subroutines or functions referenced by the program.

The running program will require a minimum process size of about 4500 halfwords to be executed. The available core area will be shared dynamically among the variables and the program. The necessary core area for local zones, arrays and variables are reserved at entry into a program unit and released at exit from the program unit. Further the program segments are transferred to the core gradually as required by the execution. If no unused core area is available, the segment transfer will overwrite one of the segments already in core.

The transfer of a segment to core store requires 8-100 milliseconds (depending on kind of backing storage, position of read/ write heads etc.), while the transfer of control to a segment already present in the core store takes about 0.01 millisecond. Therefore the program execution time may be highly dependent on whether the most frequently used loops of the program may be held entirely in the core store or not. The number of segment transfers is available as the value of an external integer variable BLOCKSREAD, which is initiated to zero at the

beginning of program execution and increased by one at every transfer of a program segment.

Appendix F. Survey of Standard Names

This appendix contains a survey of names which are reserved for special use in the RC FORTRAN. The first group of names consists of names of standard externals available to the user. The second group contains names of routines belonging to the running system of RC FORTRAN.

Table F.1 List of Standard Externals

Definition	No. of para-meters	Name of function	Type of parameter	Type of function
Absolute value of p1	1	abs	real	real
		iabs	integer	integer
		cabs	complex	real
		dabs	double	double
p1 modulo p2	2	amod	real	real
		mod	integer	integer
max(p1,p2,...)	>2	amax0	integer	real
		amax1	real	real
		max0	integer	integer
		max1	real	integer
min(p1,p2,...)	>2	amin0	integer	real
		amin1	real	real
		min0	integer	integer
		min1	real	integer
real part of complex number	1	real	complex	real
imaginary part of complex number	1	aimag	complex	real
p1 + p2 * $\sqrt{-1}$	2	cmplx	real	complex
exponential, e**p1	1	exp	real/integer	real
		cexp	complex	complex
natural logarithm, ln p1	1	alog	real/integer	real
		clog	complex	complex
sine p1	1	sin	real	real
		csin	complex	complex
cosine p1	1	cos	real	real
		ccos	complex	complex
the square root of p1	1	sqrt	real/integer	real
		csqrt	complex	complex
arctangent of p1 radians	1	atan	real/integer	real
conjugate of p1	1	conjg	complex	complex
argument of p1	1	cang	complex	real
sign of p1	2	dsign	double	double
truncated value of p1	1	ifix	real	integer
floated value of p1	1	float	integer	real

The names listed below are reserved for special purpose FORTRAN and should not be used for other purposes by the programmer.

Name	Connected to
inwrcrcrcrc	WRITE
wrircrcrcrc	WRITE
inrrcrcrcrc	READ
rearcrcrcrc	READ
lmlrcrcrcrc	long multiplication
ldlrcrcrcrc	long divison
cacrcrcrcrc	complex addition
cscrcrcrcrc	complex subtraction
cmcrcrcrcrc	complex multiplication
cdcrcrcrcrc	complex division
dadrcrcrcrc	double precision addition
dsdrcrcrcrc	double precision subtraction
dmdrcrcrcrc	double precision multiplication
dddrcrcrcrc	double precision division
icdrcrcrcrc	convert integer to double precision
lcdrcrcrcrc	convert long to double precision
rcdrcrcrcrc	convert real to double precision
dcircrcrcrc	convert double precision to integer
dclrcrcrcrc	convert double precision to long
dcrrcrcrcrc	convert double precision to real

Appendix G.
Deviations from ISO FORTRAN

G.1 Limitations

- Local variables cannot be initiated by DATA statements.
- Logical variables may not appear in EQUIVALENCE statements.
- PAUSE, REWIND, BACKSPACE and ENDFILE are not implemented.
- G-format not implemented, H-code for output only.
- Format in array not implemented.
- Statement function not implemented.
- Exponentiation of entities declared double precision is not included.
- Integers, logicals, and reals do no occupy the same amount of storage.
- Texts cannot easily be handled in integer or real variables, and not at all in logical variables.
- DATA statements may only occur among declarations.
- BLOCK DATA is not implemented.
- Recognition of compound symbols is defined as terminated by an <in text> symbol and not from the context. It is thus not possible to have a variable with the name WRITE etc.
- DO loops (and implied DO loops) are not always executed at least one time.

G.2 Extensions

- Extended character set.
- Names of more than 6 characters.
- 48-bit integers.
- Bit pattern constants, apostrophed text constants.
- B-format allowing binary, ternary, octal and hexadecimal input/output.
- Masking operations and shifting operation.
- Mixing of types in arithmetical expressions.
- General expressions allowed as subscripts, in DO statements, and in computed GOTO statements.
- Multiple entries in procedures.
- DATA initiation in any program unit.
- Open formats.

- Zones and record handling procedures.
- The parameters in a DO statement (and in implied DO) may be negative and the loop may be totally skipped.

Appendix H
Execution Times in Micro Seconds

The times given are based on a RC8000/45.

The times given below represent the total physical times for execution of algorithmic constituents. The total time to execute a program part is the sum of the times for the constituents. The times are only valid under the following assumptions:

1) The time for transfer of program segments from the backing storage negligible (see appendix E.3).

2) The program is not waiting for peripheral devices (see 5.2).

3) The time slice interval is 25.6 milliseconds or more (see ref. 10).

4) The program is the only internal process running in the computer.

When the computer is time shared, assumption 4 is not fulfilled, but then the times represent the CPU time used by the program.

H.1 Operand References

```
Reference to local identifiers and constants      0
Reference to common variable                    0-4
```

H.2 Constant Subexpressions

Operations are performed during the translation and thus do not contribute to the execution time in the following cases:

```
+-*/.shift. working on constant operands
conversion of an integer constant to a real constant,
or vice versa
```

The result of an operation performed during translation is again treated as a constant.

Examples:

```
A(-2+6/5)          is reduced to A(-1)
1+0.5-0.25         is reduced to 1.25
p+3*2-4/4          is only reduced to p+6-1 because p+6
                   must be evaluated first.
```

H.3 Saving Intermediate Results

By the term "composite expression" we mean any expression involving operations to be executed at run time.

Examples:

```
A(i)   B+1    a .shift. 8    pr(i,i, >ab>)   are composite
A(2)   >ab>   5 .shift. 20         11.5    are not composite
                                           (see G.2)
```

During evaluation of expressions one intermediate result is saved in the following cases:

`+, *, .and., .or.,all relations,.shift. when working on 2 composite expressions.`

`-,/ when the right hand expression is composite.`

Saving of an intermediate result takes the same time as an assignment.

H.4 List of Execution Times for a Selection of FORTRAN Constructs (in microseconds)

```
GOTO                          =   2.6
Assigned GOTO                 =  23.3
Computed GOTO                 =  16.3
Logical if (true)             =   5.7
Logical if (false)            =   2.9
Arith. if                     =   6.4
Continue                      =   0.0
DO (1 cycle)                  =  22.7
Call subroutine               =  91.7
For each param. add           =   6.3
Integer = integer             =   5.0
Real = real                   =   7.9
Long = long                   =   7.9
Double = double               =  39.7
compl. = compl.               =  39.7
Integer = real                =  14.4
Real = integer                =  18.3
Logical = logical             =   7.0
1 subscript, index.yes        =  15.0
```

```
1 subscript, index.no          =  11.0
Integer + integer              =   2.9
Real + real                    =  13.4
Long + long                    =   4.2
Double + double                = 279.1
Compl. + compl.                = 125.5
Real - real                    =  13.7
Integer * integer              =   8.4
Real * real                    =  34.2
Long * long                    = 144.3
Compl. * compl.                = 286.8
Double * double                = 377.2
Integer/integer                =  14.2
Real/real                      =  25.8
Long/long                      = 145.0
Compl./compl.                  = 411.9
Double/double                  = 530.5
Real ** Integer                = 236.7
Real ** Real                   = 625.0
And , or                       =   5.5
Integer.rel.integer.           =   6.2
Real.rel.real                  =  17.8
I-I * I                        =  16.5
R-R * R                        =  55.8
Ref. to real param.            =   2.0
```

Appendix I. Index