905

FORTRAN

Book No. 306

COPY OF Copy No. 5

Amendment No. 0

Maritime
Aircraft
Systems
Division

This Book forms part of the MASD Software Library.

If there is a GREEN MASD LIBRARY stamp on this page, then this is a Recorded Copy of the Master Book, for which an UPDATING service is available; so it could be up-to-date.

If there is NO stamp or only a XEROXED stamp on this page, then this copy has NO updating service; and the reader uses it at his PERIL.

Marconi-Elliott Avionic Systems Limited
Airport Works, Rochester, Kent. ME1 2XX

A GEC-Marconi Electronics Company

Telephone: Medway (0634) 44400
Telegrams: Elliotauto Rochester
Telex: 96333/4

Compiled from Various Sources;
Issued by T.J.Froggatt.

Terry Froggatt 31/1/74

PREFACE.

This book describes the following tapes:-

```
    905 FORTRAN COMPILER,       1/1/74, Binary Mode 3;
    905 FORTRAN LIBRARY VOL 1, 1/1/74, Intermediate Mode 3;
    905 FORTRAN LIBRARY VOL 2, 1/1/74, Intermediate Mode 3.
```

These tapes enable FORTRAN programs to be run on a 905 or 920C computer with at least 16K store.

905 FORTRAN COMPILER, 1/1/74, Binary Mode 3;

905 FORTRAN LIBRARY VOL 1, 1/1/74, Intermediate Mode 3;

905 FORTRAN LIBRARY VOL 2, 1/1/74, Intermediate Mode 3.

905 FORTRAN

PREFACE

The information contained within this publication describes the FORTRAN language applicable to 905 series 18-bit machines and termed '905 FORTRAN'; the origin of the word FORTRAN is derived from the words FORmulae TRANslations. 905 FORTRAN contains most of the features of full standard ASA FORTRAN, see Appendix 2.

905 FORTRAN can be used on any 905 Series 18-bit machine which includes teleprinter, punch and reader facilities and has a minimum store size of 16K.

FORTRAN programs must only be written and/or punched in characters which are contained in the 900 series internal character code, see Chapter 1.

As with all other languages (e.g. English, Mathematics and the programming languages COBOL, USERCODE etc.) rules are applicable to the use of 905 FORTRAN. These rules are termed the 'syntax' of the language. The meanings given to elements of the language are termed the 'semantics' of the language. Elements of the language take the form of characters used singly or in various combinations to form statements, the uses of which are governed by the syntax of the language.

Although a programmer may prepare a program which is syntactically perfect, the semantics of that program may not necessarily fulfil the purpose of that program. It is therefore essential that programmers understand fully the implication of both the syntax and the semantics of the language and of a program written in that language.

CHAPTER 1 : THE CHARACTER SET

As all elements consist of characters used either singly or in combination, characters are the first items to be detailed.

The standard FORTRAN character set consists of the following characters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and:

Character	Name of Character
	Blank (space in paper tape code)
=	Equals
+	Plus
-	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point
$	Currency Symbol

Standard FORTRAN programs must be written using the above characters. Other characters, which are not part of the standard character set, may be used in Hollerith strings and Hollerith constants; the programmer is warned that use of other characters may restrict compatability with other FORTRAN implementations.

The full set of characters available for writing 905 FORTRAN programs are the 64 characters having internal code equivalents in the 900 series character code. This code, with the full paper tape representation, is given in the table at the end of this chapter. It is based on the British Standard variant of the International Standards Organisation (ISO) code. The following characters have special significance:

Space — This is the character referred to as 'blank' in the FORTRAN standard, in accordance with punched card conventions. Except where explicitly stated, space is not a significant character and may be used freely to improve the appearance of programs.

Newline — (Line feed on some teleprinters)
Newline normally indicates the beginning of a new record, or line. Within a program, lines may be up to 72 characters long, ignoring null (paper tape blank), carriage return and erase. Newline may not be included in a Hollerith string or constant.

NOTE: The codes for CARRIAGE RETURN, ERASE and RUNOUT, though they may appear in a FORTRAN data or program tape, are ignored when read by the compiler.

The character for HALTCODE is used to halt the machine at the end of a tape. If a program comprises more than one tape, each tape must be terminated with a HALTCODE. This termination of tape makes possible the reading in of consecutive tapes. Each HALTCODE must follow a new line.

905 FORTRAN Character Set

ISO Code Value	Value with Parity	Telecode Character	Binary Pattern	SIR Internal Code		NOTES
				Octal	Decimal	
0	0	Null	00000.000			Blank on paper tape; ignored by program.
1	129		10000.001			Illegal characters.
2	130		10000.010			
3	3		00000.011			
4	132		10000.100			
5	5		00000.101			
6	6		00000.110			
7	135	Bell	10000.111			
8	136		10001.000			
9	9	Hor. Tab	00001.001			Horizontal tab. Ignored by teleprinter; equivalent to space.
10	10	Line Feed	00001.010	01	1	Line feed. Causes newline
11	139		10001.011			Illegal characters
12	12		00001.100			
13	141	Car Ret.	10001.101			Carriage return; ignored by program.

ISO Code Value	Value with Parity	Telecode Character	Binary Pattern	SIR Internal Code		NOTES
				Octal	Decimal	
14	142		10001.110			Illegal Characters
15	15		00001.111			
16	144		10010.000			
17	17		00010.001			
18	18		00010.010			
19	147		10010.011			
20	20	Halt	00010.100			Halt; ignored by teleprinter. Halts input of paper tape.
21	149		10010.101			Illegal characters
22	150		10010.110			
23	23		00100.111			
24	24		00011.000			
25	153		10011.001			
26	154		10011.010			
27	27		00011.011			
28	156		10011.100			
29	29		00011.101			
30	30		00011.110			
31	159		10011.111			

ISO Code Value	Value with Parity	Telecode Character	Binary Pattern	SIR Internal Code		NOTES
				Octal	Decimal	
32	160	Space	10100.000	00	0	Space
33	33	!	00100.001			Exclamation mark - illegal character.
34	34	"	00100.010	02	2	Quotes
35	163	£	10100.011	03	3	½ on some teleprinters.
36	36	$	00100.100	04	4	Dollar - currency symbol.
37	165	%	10100.101	05	5	Percent
38	166	&	10100.110	06	6	Ampersand
39	39	´	00100.11	07	7	Acute (Open quote)
40	40	(	00100.000	10	8	Left parenthesis
41	169	)	10101.001	11	9	Right Parenthesis
42	170	*	10101.010	12	10	Asterisk
43	43	+	00101.011	13	11	Plus
44	172	,	10101.100	14	12	Comma
45	45	-	00101.101	15	13	Minus (hyphen)
46	46	.	00101.110	16	14	Full Stop
47	175	/	10101.111	17	15	Solidus (slash)
48	48	0	00110.000	20	16	Digits
49	177	1	10110.001	21	17	
50	178	2	10110.010	22	18	

ISO Code Value	Value with Parity	Telecode Character	Binary Pattern	SIR Internal Code		NOTES
				Octal	Decimal	
51	51	3	00110.011	23	19	Digits
52	180	4	10110.100	24	20	
53	53	5	00110.101	25	21	
54	54	6	00110.110	26	22	
55	183	7	10110.111	27	23	
56	184	8	10111.000	30	24	
57	57	9	00111.001	31	25	
58	58	:	00111.010	32	26	Colon
59	187	;	10111.011	33	27	Semicolon
60	60	<	00111.100	34	28	Less than
61	189	=	10111.101	35	29	Equal to
62	190	>	10111.110	36	30	Greater than
63	63	?	00111.111	37	31	Subscript ($_{10}$) on some teleprinters.
64	192	@	11000.000	40	32	Grave \ on some teleprinters.
65	65	A	01000.001	41	33	Letters
66	66	B	01000.010	42	34	
67	195	C	11000.011	43	35	
68	68	D	01000.100	44	36	

ISO Code Value	Value with Parity	Telecode Character	Binary Pattern	SIR Internal Code		NOTES
				Octal	Decimal	
69	197	E	11000.101	45	37	Letters
70	198	F	11000.110	46	38	
71	71	G	01000.111	47	39	
72	72	H	01001.000	50	40	
73	201	I	11001.001	51	41	
74	202	J	11001.010	52	42	
75	75	K	01001.011	53	43	
76	204	L	11001.100	54	44	
77	77	M	01001.101	55	45	
78	78	N	01001.110	56	46	
79	207	O	11001.111	57	47	
80	80	P	01010.000	60	48	
81	209	Q	11010.001	61	49	
82	210	R	11010.010	62	50	
83	83	S	01010.011	63	51	
84	212	T	11010.100	64	52	
85	85	U	01010.101	65	53	
86	86	V	01010.110	66	54	
87	215	W	11010.111	67	55	

ISO Code Value	Value with Parity	Telecode Character	Binary Pattern	SIR Internal Code		NOTES
				Octal	Decimal	
88	216	X	11011.000	70	56	Letters
89	89	Y	01011.011	71	57	
90	90	Z	01011.010	72	58	
91	219	[	11011.011	73	59	Left bracket
92	92	\	01011.100	74	60	Reverse slash (£ on some teleprinters)
93	221	]	11011.101	75	61	Right bracket.
94	222	↑	11011.110	76	62	^ on line printer.
95	95	_	01011.111	77	63	Underline char. (on some teleprinters).
96	96	`	01100.000			Grave accent (@ on some teleprinters).
97	225	a	11100.001	41	33	Upper case on teleprinter.
98	226	b	11100.010	42	34	
99	99	c	01100.011	43	35	
100	228	d	11100.100	44	36	
101	101	e	01100.101	45	37	
102	102	f	01100.110	46	38	
103	231	g	11100.111	47	39	
104	232	h	11101.000	50	40	
105	105	i	01101.001	51	41	

ISO Code Value	Value with Parity	Telecode Character	Binary Pattern	SIR Internal Code		NOTES
				Octal	Decimal	
106	106	j	01101.010	52	42	Upper case on Teleprinter
107	235	k	11101.011	53	43	
108	108	l	01101.100	54	44	
109	237	m	11101.101	55	45	
110	238	n	11101.110	56	46	
111	111	o	01101.111	57	47	
112	240	p	11110.000	60	48	
113	113	q	01110.001	61	49	
114	114	r	01110.010	62	50	
115	243	s	11110.011	63	51	
116	116	t	01110.100	64	52	
117	245	u	11110.101	65	53	
118	246	v	11110.110	66	54	
119	119	w	01110.111	67	55	
120	120	x	01111.000	70	56	
121	249	y	11111.001	71	57	
122	250	z	11111.010	72	58	
123	123		01111.011			Illegal characters.
124	252		11111.100			
125	125		01111.101			
126	126		01111.110			
127	255		11111.111			Erase - ignored by program.

CHAPTER 2 : ELEMENTS OF THE 905 FORTRAN LANGUAGE

This chapter defines the elements from which 905 FORTRAN statements are composed. They are:

a) Constants

b) Variables

c) Identifiers

d) Operators

e) Expressions

f) Functions

2.1 Constants-Definition

A constant is a value which remains unchanged throughout its use; it can be used in one or more statements. For example, in the statements:

x=a+3

y=b+4

x, a, y and b can vary (i. e. are variables)

+3 and +4 remain unchanged (i. e. are constants).

Constants can be of various types, i. e.

Integer constant

Real constant

Double precision constant

Complex constant

Logical constant

Hollerith constant

For many straightforward programs it is sufficient to use integer and real constants only; programmers unfamiliar with FORTRAN should avoid the use of other types until more experience is obtained in the use of FORTRAN.

If the value of an integer, real or double precision constant is positive, the inclusion of a plus sign preceding the constant is optional; if the value represented is negative, a minus sign must precede the constant. An unsigned value is assumed to be positive.

2.2 Integer Constants

These constants consist of whole numbers (integers) and are written as a set of digits either optionally preceded by a plus sign, or preceded by a minus sign.

An integer constant can take any integer value in the range:

$- 131071 \leqslant \text{constant} \leqslant 131071$

Examples of valid integer constants are:

0

+9387

-2001

6

Examples of invalid integer constants are:

12.78	-	contains a fractional part and is therefore a real constant
-10,000	-	contains invalid character i.e. comma
16748932	-	outside the value range for an integer constant

2.3 Real Constants

A real constant can take any (real) value in the range:

$-10^{19} \leqslant \text{constant} \leqslant 10^{19}$ (including zero)

Real is used in this instance in its mathematical sense of any numerical value not containing an imaginary part.

A real constant may be written in one of the forms which follows:

a) A set of digits containing a decimal point E.g.

2.5 .05 123.

b) A set of digits which can contain an optional decimal point, followed by a capital letter E and an optionally signed integer of one or two digits length. E.g.

2.5E1 36E-15 .03E+12

NOTE: In either form the signing of a real constant is optional.

The integer after the character E represents a power of 10 multiplying the number that precedes the E. In the examples given the values represented are:

$2.5*10^{1}$ $36*10^{-15}$ $.03*10^{+12}$

Examples of valid real constants are:

0.0

-2000.0

? +234 ,

5.0E+6 ($5*10^6$)

-7.E-12 ($-7.0*10^{-12}$)

Examples of invalid real constants are:

12,345.6	-	comma is an invalid character
+234	-	no decimal point and is therefore an integer constant
5.86E2.5	-	exponent not an integer
1.6E+81	-	value too large for the range of real constant values

2.4 Other Types of Constants

2.4.1 Double Precision Constants

For some calculations it is necessary to use a higher precision than that used within the computer for real values. This means that a greater number of significant digits are necessary to provide for that higher precision. These higher precision values in FORTRAN are termed 'double precision' values. They are real in the mathematical sense but occupy more space in the computer store. Double precision constants can take any value in the range:

$-10^{99} \leq$ constant $\leq +10^{99}$

A double precision constant is written as a string of digits optionally containing a decimal point and followed by the capital letter D and an optionally signed one or two digit integer; if not signed, a positive value is assumed.

Examples of double precision constants and the values they represent are:

.25D-12	$0.25*10^{-12}$
-25D+33	$-25*10^{33}$
+3.99D1	39.9
1.234567890D-2	0.01234567890
-12D0	-12.00000000

2.4.2 Complex Constants

Complex constants represent complex values in the mathematical sense; i.e. numbers having a real and an imaginary part. In FORTRAN either or both the real and imaginary parts may be zero.

A complex constant is written in the form:

Left parenthesis, real constant (representing the real part), a comma, real constant (representing the imaginary part), right parenthesis.

The real constants may be optionally signed; if unsigned, a positive value is assumed. They may take any of the forms and values for real constants described in Section 2.3.

Examples of complex constants and the values they represent ($j=\sqrt{-1}$) are:

(-1.0,+.3)	-1.0+.3j
(2E-5,5.6789)	0.00002+5.6789j
(+6.7,-2E4)	6.7-20000.0j
(0.0,-6.)	-6.0j
(2.7,0.0)	2.7

2.4.3 Logical Constants

Logical constants are used in conjunction with FORTRAN logical type variables (section 2.5.8). There are two logical constants permissible and these represent the Boolean values true and false. They are written:

.TRUE.

.FALSE.

i.e. the name of the value preceded and followed by a decimal point

2.4.4 Hollerith Constants

A Hollerith constant represents a string of characters which may include any characters having representation in the 900 Series internal character code, with the exception of newline (see Chapter 1). For standard FORTRAN compatibility, only those characters in the standard FORTRAN character set should be used.

Hollerith constants are written in the form, unsigned integer (positive), capital letter H and a string of characters. The number of characters in a string, counting spaces (blanks) as significant, must be equal to the integer value before the character H.

Examples of Hollerith constants are:

1HX

28HTHIS IS A HOLLERITH CONSTANT

14H**3-£ (=),+END/

The only positions in which a Hollerith constant may appear in a FORTRAN program are:

a) In the argument list of a CALL statement.

b) In a DATA INITIALISATION statement.

NOTE: Although a Hollerith string may appear in a FORMAT statement, in the same form as a Hollerith constant, in this use it is not strictly a Hollerith constant.

Hollerith constants are represented in the 900 Series store by 6-bit characters in 900 Series internal code. They are packed 3 to an 18-bit word and left justified; i.e. the first character is in the most significant bits of a word and zeros (spaces) are used to pad any remaining bits of the word.

The maximum number of characters that can be stored in a variable by a DATA statement containing a Hollerith constant will depend on the type of variable:

One word integers	3 characters
Two word integers	6 characters
Real variables	6 characters
Double precision variables	12 characters
Complex variables	12 characters

NOTE: Programmers must be careful when transferring programs containing Hollerith constants between different types of machine. Programmers are advised not to use Hollerith constants unless they are essential to their program. The inexperienced FORTRAN programmer should avoid their use altogether.

2.5 Variables

2.5.1 Definition

'Variable' is the term given to the identifier of a value and the location in which that value is stored. This value may change according to the use of a variable in either:

a) A specific program or subprogram, or

b) A number of programs, subprograms to which it is COMMON (see Section 4.2)

The location in which this value is held can be similarly either:

a) Unique to a specific program, sub-program, or

b) Common to a number of programs, sub-programs.

Variables (including subscripted variables) can be of the following types:

Integer

Real

Double precision

Complex

Logical

For many programs only integer and real variables need be used. The inexperienced FORTRAN programmer is advised to leave the consideration of other types until the language is thoroughly understood.

2.5.2. Uses of Variables

a) As common variables - When COMMON to a number of programs, sub-programs. In this instance their identifiers will be declared in a COMMON statement within a FORTRAN program.

b) As local variables - When used in a specific FORTRAN program or sub-program.

c) As formal parameters - When specified in the argument list of a FORTRAN 'FUNCTION' or "SUBROUTINE' statement; the variable is regarded as a formal parameter (argument) within that function or subroutine.

In any of the above uses the variable name may be subscripted if its identifier has been declared as an array name by means of a specification statement (see Chapter 4).

2.5.3 Identifiers

An identifier is a name given by the programmer to an entity within a FORTRAN program. An identifier may be the name of:

A variable

An array (see Chapter 4)

A FUNCTION or SUBROUTINE program unit (see Chapter 6)

A COMMON block (see Chapter 4).

Within a unit of FORTRAN program an identifier can be used to name one entity of one of the types listed.

If an identifier is not explicitly declared to be one of the types stated, it is assumed by the compiler to be a real or integer variable (An identifier is explicitly declared by writing it in a SPECIFICATION, FUNCTION, or SUBROUTINE statement).

An identifier must conform to the following rules:

a) It must be a string of letters or letters and digits (not including spaces), the first character of which must always be a letter.

b) It may contain from one to six characters, but must not exceed six characters.

c) If the first character of an identifier is Q, the second character must be U, unless it is an identifier defined in a standard software manual. Strictly this rule only applies to the names of COMMON blocks, SUBROUTINES and FUNCTIONS, and to identifiers of the general form Qn, where n is a set of digits. This rule prevents confusion with machine code and software global identifiers.

d) If the identifier is the name of an integer variable, array, or FUNCTION, it must commence with one of the letters:

   I, J, K, L, M, or N

   an exception to this rule is the identifier included in a TYPE statement (see Chapter 4).

e) For real variables, arrays and FUNCTIONs, the first letter of an identifier can (with the exception of identifiers included in TYPE statements) be any letter other than:

   I, J, K, L, M, or N.

An identifier occurring in a FORTRAN program unit which does not appear in a specification statement (see Chapter 4), or as a SUBROUTINE or FUNCTION name, is assumed to be a real or integer variable of the type implied by the first letter.

These rules for identifiers must be observed at all times; of particular importance is the distinguishing between real and integer variable identifiers. In some instances a violation of these rules will cause the FORTRAN compiler to output an error message; but as all errors cannot be covered, the possibility of a program giving incorrect results is present. For this reason many programmers will consider it advisable to explicitly declare all identifiers by means of Type statements. (Use of a Type statement may override the implicit type given by the first letter, see Chapter 4).

From the preceding paragraphs the need for accuracy in assigning identifiers is evident and this need cannot be over stressed.

Examples of acceptable integer variable identifiers are:

I

KLM

MATRIX

L123

Examples of incorrect integer variable identifiers are:

ABC	(incorrect first letter unless an explicitly declared integer)
5M	(does not begin with a letter)
$J78	(incorrect character viz. dollar sign)
INTEGER	(too many characters)
J34.5	(incorrect character viz. decimal point)
JOB-STEP	(incorrect character viz. hyphen, and too many characters)

Examples of acceptable real variable identifiers are:

AVAR

FRONT

F009

QUIZ

Examples of incorrect real identifiers are:

QA1234	(letter following Q not U)
SERVICE	(too many characters)
8BOX	(first character not a letter)
*BCD	(invalid character viz. asterisk)
KLJ1	(invalid first character unless explicitly declared real)
A+B	(invalid character viz. +)

2.5.4 Integer or Fixed point Variables

These may take any integer value (including zero) in the range:

$-131072 \leq \text{value} \leq 131071$

If this range is exceeded, overflow occurs; however this error is not detected, except on real to integer conversion.

Integer variables occupy one 18-bit word in the 905 store. However, the FORTRAN standard specifies that an integer variable takes the same number of logical storage units as a real variable. Therefore two words are reserved for each integer value, unless the option bit to select single word packed integers is used (see Chapter 9). This option will save store, but may lead to incompatibility in the use of COMMON and EQUIVALENCE statements when running FORTRAN program on computers other than 905 Series 18-bit machines.

When the two word option is used, the first word contains the value, the second will (unless used for a Hollerith constant) be spare.

It is important that all units of a program be compiled with either the one or two word option applied. A check is made by the loader that this is so; if not, at load time an error message will be output. If the program does not contain integer or logical arrays, this check will not be performed.

2.5.5 Real Variables

These are held in store in floating point form; i.e. a fraction times a power of 2. They must be in the range:

$-2^{63} < \text{value} < 2^{63}$

i.e. approximately:

$-9X10^{18} < \text{value} < 9X10^{18}$

If this range is exceeded at run time, exponent overflow error is reported. On continuation, the value is assumed to be the largest possible magnitude number (of the correct sign).

Real numbers are held to an accuracy of approximately 8 decimal digits (28 binary bits).

If the magnitude of a real variable becomes less than 2^{-64} (10^{-19} approx.), its value is automatically set equal to zero. This action is not reported as an error.

2.5.6 Double Precision Variables

These variables are used to represent real numbers to a finer degree of approximation than that used for real variables. A wider range of values is also allowed.

They are held in the store in three word form. For compatibility with standard FORTRAN, double precision arrays use four words of store per element.

The range of values allowed is approximately:

$-10^{300} < \text{value} < +10^{300}$

If this range is exceeded at run time, an exponent overflow error is output. The numbers are held to an accuracy of 10 decimal digits (35 binary bits).

If at run time the absolute magnitude of a double precision value becomes less than 10^{-300} approximately, the value is set automatically to zero. This action is not reported as an error.

2.5.7 Complex Variables

These are stored in the form of a pair of real values. The first value represents the real part of the complex number, the second part the imaginary part.

In 905 FORTRAN, complex variables occupy four store locations. The limits and accuracy of real variables (see Section 2.5.5) apply to each part of the complex variable.

2.5.8 Logical Variables

These variables may take two values only: .TRUE or FALSE. In 905 FORTRAN, they normally occupy two words of store, but occupy only one word if the one word (packed) integer option is used.

They may be used in logical assignment statements and logical IF statements (see Section 3.4).

2.5.9 Arrays (Subscripted Variables)

Variables in FORTRAN can be grouped into sequential sets in the form of one, two, or three dimensional arrays of data. A one dimensional array is sometimes termed 'a vector', a two dimensional array may represent a Matrix. The use of arrays enables large quantities of data to be handled efficiently.

In mathematical notation it is permissible to write:

$x_1, x_2, x_3, x_4 \ldots\ldots\ldots x_n$

and in FORTRAN to write:

X(1), X(2), X(3), X(4), ...X(N) also:

$m_{1,1}, m_{1,2}, m_{1,3} \ldots\ldots\ldots\ldots\ldots m_{1,j}$

$m_{i,1}, m_{i,2}, m_{i,3} \ldots\ldots\ldots\ldots\ldots m_{i,j}$ hence, FORTRAN gives:

M(1, 1), M(1, 2), M(1, 3), ... M(1, J)

M(I; 1), M(I, 2), M(I, 3)................ M(I, J)

The name of an array is an identifier formed according to the rules stated for ordinary variables. It must be explicitly declared as the name of an array by a DIMENSION statement or by dimension information in another specification statement (see Chapter 4).

In the examples given previously, subscripts 1, 2, 3 etc. are integer constants and M, I and J are integer variables. All allowable forms of subscript are listed as follows:

FORM	EXAMPLE OF SUBSCRIPTED VARIABLE
k	X(16)
I	VARRAY (INDEX)
k*I	IARRAY(2*JN)
I+ℓ	Z(J+2)
I-ℓ	Z2(KAPPA-100)
k*I+ℓ	A(10*N+5)
k*I-ℓ	A(2*IN-1)

Where:

k and ℓ represent any positive integer constants

I may be replaced by any integer variable.

NOTE: No other subscript forms are permissible. As indicated in the examples given, each subscript of a two or three dimensional array must be separated by a comma. Viz.

X(I, J, K)

MATRIX (3*KAPPA-1, 2*J+1)

If declared as an array of the correct explicit/implicit type, any of the variable types, integer, real, double precision, complex and logical may be used as a subscripted variable.

The value of any subscript, either a constant or an expression, must be positive, greater than zero, and must not exceed the maximum value for that subscript in the specification statement by which it was declared. The number of subscripts of a subscripted variable must correspond with the number in its declaration (use in an EQUIVALENCE statement excepted).

2.5.10 Storage of Arrays

The elements of arrays with more than one subscript are stored with the first element being the value which most frequently changes when counting the elements in sequence. For example, if the elements of a two subscripted array represent rows and columns, the rows element will be stored first as it will change more frequently than the column element. Viz.

R1	C1	A(1, 1)
R2	C1	A(2, 1)
R3	C1	A(3, 1)

R4	C1	A(4, 1)
R5	C1	A(5, 1)
R1	C2	A(1, 2)
R2	C2	A(2, 2)
R3	C2	A(3, 2)
R4	C2	A(4, 2)
R5	C2	A(5, 2)
.		
.		
.		
.		
.		
.		
etc.		

In precise form, the storage of the array elements of two or three dimensional arrays correspond to storage of an equivalent one dimensional array as follows:

For a two dimensional array A 1: m, 1: n element i, j corresponds to the element i+m(j-1)

For a three dimensional array A 1: ℓ, 1: m, 1:n elements i, j, k correspond to element $i+\ell(j-1)+\ell m(k-1)$

Examples of arrays are:

a) Elements of Real Arrays:

A(1)

DOG(3, 7*ITEM-5000, 2*MEAN+10)

b) Elements of Integer Arrays:

LIST(4*JULIET)

JIG(2*I-9, 4*J+100)

An array comprising three rows and three columns could be shown initially in mathematical notation thus:

$$\begin{array}{ccc} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{array}$$

FORTRAN subscript notation of this array would be written thus:

A(1, 1), A(1, 2), A(1, 3), A(2, 1), A(2, 2), A(2, 3)....... etc.

The identifier for subscripted variables is subject to the rules stated for non-subscripted variables (section 2.5).

2.6 Expressions

2.6.1 General

A FORTRAN expression is a rule for computing a numerical value. In many instances an expression consists of a single constant, a single variable, a single function reference. Two or more of these elements may be combined (using operation symbols and delimiters) to build more complex expressions.

Each operation is represented by a unique symbol thus:

+ indicating positive or addition

- indicating negative or negation

* indicating multiplication (used instead of the character X to avoid confusion with the letter X)

/ indicating division

** indicating exponentiation (i. e. raising to a power).

The delimiters used are as follows:

( ) which enclose subscripts or parameter lists, and modify the order of evaluation of the terms within an expression.

<space> Only used within an expression to clarify reading of an expression, but ignored by the compiler.

NOTE: Parentheses may be used to group expressions in the same manner to that in mathematical notations. Thus $(X+Y)^3$ must be written (X+Y)**3 in order to convey the correct meaning. An expression as ambiguous as:

A^{B^C} must be written as A**(B**C) or (A**B)**C according to the requirement intended.

2.6.2 Order of Evaluation

The current values of the variables within the expression are first determined. This may necessitate the evaluation of subscripts or functions before evaluation of the main expression can commence.

When the order of operators within an expression cannot be defined clearly by the use of parentheses, the order of evaluation (of other permitted operators) is as follows:

a) Exponentiations

b) Multiplications and divisions

c) Additions and subtractions

Thus, the two expressions which follow have equivalent mathematical meanings:

a) A*B+C/D-E**F

b) (A*B)+(C/D)-(E**F)

Within a sequence of consecutive multiplications and/or divisions or additions and/or subtractions, in which the order of evaluation is not clearly defined by parentheses, the meaning is evaluated from left to right. Thus the expression A/B*C would mean (A divided by B) times C and not A (divided by B times C) and I-J+K would mean (I minus J) plus K and not I minus (J plus K).

Sub-expressions with parentheses are evaluated in the same manner but not necessarily in the same sequence as subscripts or functions; the value thus attained is used to assist in the evaluation of the main expression.

The effect of division extends only to the next element, for example:

A/B*C is equivalent to (A/B)*C or A*C/B

A/B/C is equivalent to A/(B*C)

The incorrect order of evaluation of an expression can result in a loss of significance or even in a failure to obtain an answer. E.g. In the expression : A*B/C if the values of A, B, and C were approximately 10^{30}, the result of the multiplication would be 10^{60} and this result is outside the permitted range for values of real variables. However, after division the result returns within the permitted range. If the computer cannot represent an intermediate result (e.g. 10^{60}), the outcome of the evaluation will depend on the mode of working (i.e. Integer or Floating Point). The outcome in integer working is for the evaluation to continue and this results in an erroneous answer. The outcome in floating point working is for the evaluation to stop; an error message is output and continuation of the program is left to the discretion of the operator.

2.6.3 Treatment of Integer and Real Values

The evaluation of an expression containing integer and real values is achieved by initially converting the integer values or integer sub-expressions into real values.

2.6.4 Results of Integer Division

The result of an integer division is always an integer truncated so that the fractional part of the exact answer is omitted. Hence,

7/4 gives the result 1(not 2)

-5/3 gives the result -1 (not-2)

These results mean that the remainder of an integer division can be easily found. However, these results sometimes produce unexpected effects, e.g.

5/3*6 gives a result of 6

5/(3*6) gives a result of 0

5*(6/3) gives a result of 10

2.6.5 Exponentiation Results

An integer, real, double precision or complex number raised to an integer power always gives a result of the same type; i.e. integer, real, double precision or complex respectively.

NOTE: This means that the expression I**J (where J is negative) will always give a result of zero due to the truncation rule.

A real or double precision value may be raised to a real or double precision power. The result is double precision unless both values are real. Combinations of exponentiation other than those listed in the table (Section 3.3) are not permitted.

Attempts to exponentiate zero by zero and a negative real number by a negative real power will give rise to an error at run time.

2.6.6. Types of Allowable Expressions

The rules for the mixing of types of expressions are given in Chapter 3. It should be noted that a sub-expression of one type can usually be converted into another type by the use of the functions:

IFIX FLOAT DBLE SNGLE REAL AIMAG

CHAPTER 3 : STATEMENTS

3.1 Format

A FORTRAN program consists of a sequence of statements. These statements can be divided into two types:

a) Executable statements, which are obeyed when the program is run.

b) Non-executable statements which further define the meaning of executable statements. These statements may be divided into four types:

   (i) Those which provide the compiler with information e.g. COMMON statements.

   (ii) Those which further define run-time operations, e.g. FORMAT statements.

   (iii) Those which provide information during both compilation and at run-time, e.g. DIMENSION statements.

   (iv) Those which contain information which ease reading of programs, but have no actual effect on the compilation or running, i.e. Comments.

3.2 Arithmetic Assignment Statements

To enable a new value of a variable to be computed, an arithmetic assignment statement is necessary. The statement takes the form:

a = b

where:

'a' represents a variable name written without a sign and,

'b' represents any expression.

An Arithmetic Assignment Statement is an order to FORTRAN to compute the value of the expression on the right and to give that value to the variable named on the left. When a real result is assigned to an integer variable, then:

a) it is rounded towards zero

b) integer overflow may occur

The equals sign within an arithmetic assignment statement is not used as in normal mathematical notation. Thus it is not permissible to write a statement Z-RHO=ALPHA+BETA, in which the value of Z is unknown whilst the other values are known. The only legal form of arithmetic assignment statement is one in which the left hand side of the statement is the name of a single variable. The precise meaning of an equal sign within an arithmetic assignment statement is thus: Replace the value of the variable named on the left with the value of the expression on the right. Thus the statement A=B+C is an order to form the sum B+C and to replace A with this value. If the right hand side of an expression is in integer and the left hand side is a real variable, the integer result is converted into floating-point (real) and this value is assigned to the variable.

The variable on the left may be subscripted.

3.3 Mixed Mode Arithmetic

The ASA standard imposes a number of restrictions on arithmetic modes. Thus:

a) With the basic operations add, subtract, multiply and divide, an element can be combined with another element of the same type, or a real element may be combined with a double precision or complex element.

b) The validity of a**b depends on the values of a and b and on the type of b. The result (where is exists) is always real. If b is an integer variable, constant, or expression, the operation is valid unless both a and b are zero. In all other instances, the result is that obtained by multiplying a by itself $|b|$ times and (if b is negative) taking the reciprocal. If b is a real variable, constant, or expression, the operation is only valid if a is positive. The result (where it exists) has the value exp $(b*\log_e(a))$.

c) An integer, real or double precision value may be assigned to an integer, real, or double precision element in an arithmetic assignment statement. A complex value may be assigned to a complex element only.

d) A relational operator can combine two expressions of each of the following types:

(i) Integer

(ii) Real

(iii) Double precision

(iv) Real with double precision

NOTE: 905 FORTRAN provides a wider range of mixed modes with basic operations of add, subtract, multiply and divide in that an integer element may be combined with a real or double precision element. Incorrect use of this facility may result in an object program which is appreciably less efficient than it might otherwise have been.

A relational expression is one that compares integer, real or double precision values by using the relational operators:

.LT.
.LE.
.EQ.
.NE.
.GT.
.GE.

The table which follows indicates the only permissible combinations (where: R=Real, I=Integer, D=Double precision and C=Complex).

Add, Subtract, Multiply, Divide

	I	R	D	C
I	I	R*	D*	X
R	R*	R	D	C
D	D*	D	D	X
C	X	C	X	C

Exponentiation

E A	I	R	D	C
I	I	X	X	X
R	R	R	D	X
D	D	D	D	X
C	C	X	X	X

Assignment

Exp Var	I	R	D	C
I	√	√	√	X
R	√	√	√	X
D	√	√	√	X
C	X	X	X	√

Relational

	I	R	D	C
I	√	√*	√*	X
R	√*	√	√	X
D	√	√	√	X
C	X	X	X	X

(The entries in the MATRIX show the result type: and X indicates when a combination is not permitted; √ indicates that combination is permitted) √ with * indicates that this combination is an extension to ASA Fortran.

3.4 Control Statements

Control statements are used when a break is required in the order in which executable statements of a program are to be obeyed. A breakdown of the elements available for making transfers of control in the FORTRAN language follow.

3.4.1 Statement Numbers or Labels

A statement number is an unsigned positive integer, which is prefixed to a statement. The number can consist of from 1 to 5 (maximum) digits and is coded in columns 1 to 5 of the coding sheet. Statement numbers may run in random order throughout a program, but no two statements in a program unit may have the same statement number. Statement numbers are used to provide for interaction between statements and to provide an identity to the numbered statement for transfer of control or FORMAT references.

3.4.2 GOTO Statements

A GOTO statement provides the means of transferring control to any statement other than the next in sequence. The next executable statement after a GOTO statement must bear a statement number otherwise it could never be executed.

There are 3 types of GOTO statments available in 905 FORTRAN they are:

1) The unconditional GOTO statement

2) The computed GOTO statement

3) The assigned GOTO statement

3.4.3 Unconditional GOTO Statement

Unconditional GOTO Statements are written in the form:

```
GOTO n
```

where n is the statement number of the next statement to be executed. Control is transferred unconditionally.

Example:

```
   GOTO  15
15 GOTO   8
```

3.4.4. Computed GOTO Statement

A computed GOTO statement provides the user with a n-way switch based on the value of an integer variable. The statement has the form:

GOTO $(n_1, n_2. \ldots\ldots\ldots\ldots, n_m)$, i

where $n_1, n_2, n_3, n_4. \ldots, n_m$ are m statement numbers (which need not all be different) and i is a un-subscripted integer variable which, whenever the statement is obeyed, must have a value in the range 1 to m.

Example:

```
GOTO (4,600,13,9,526)IAC
```

If the value of IAC is 1 then control would be transferred to statement 4, if IAC had the value 2 control would be transferred to statement 600 and so on. If the value of the integer variable lies outside the range, at run time an error is reported.

3.4.5 Assignment Statement and Assigned GOTO

The combination of the GOTO Assignment statement and the Assigned GOTO gives an alternative to the computed GOTO. A statement label (number) is associated with an integer variable by means of a GOTO Assignment. At some point later in the program this may be used in an Assigned GOTO statement to branch to that numbered statement. In programming terms, it is a means of presetting a switch.

A GOTO Assignment takes the form:

ASSIGN K TO i

where K is a statement label of an executable statement in the current program unit.

i represents any integer variable.

Once the Assignment statement has been obeyed, integer i must not be referenced or changed until an assigned GOTO is obeyed. That assigned GOTO will cause control to be transferred to the statement numbered K. The form of an assigned GOTO is

GOTO i, (K_1, K_2, K_3, ..., K_n)

where i is an integer variable that must have been previously set by a GOTO assignment. K_1.....K_n represents statement numbers (one of which must be the label number K used in the ASSIGN statement).

Note that the value set in i is not meaningful, in particular it is not the numerical value of K.

The ASSIGN statement and its associated GOTO statement must be in the same program unit.

Example:

```
    ASSIGN 99 TO JJ
    GOTO  10
99  X=Z+Y
10 GOTO JJ, (97,98,99,100)
```

The equivalent computed GOTO would be

```
      J2=3
      GOTO 10
99    X=Z+Y
10    GOTO(97,98,99,100), J2
```

In 905 FORTRAN , the computed GOTO statement is checked at run-time to ensure that the integer variable is within the permitted range. Transfer of control, using the assigned GOTO statement, is executed in a shorter time, than a computed GOTO statement.

3.4.6 The Arithmetic IF Statement

An Arithmetic IF Statement has the form:

IF(e) n_1, n_2, n_3

where (e) is an arithmetic expression and n_1, n_2, n_3, are three statement numbers (not necessarily different). Control is transferred to either n_1, n_2, or n_3 according to the sign of the value of (e).

sign of e	
NEGATIVE	n_1
ZERO	n_2
POSITIVE	n_3

Example:

```
53 IF (NX-2) 150, 151, 999
```

3.4.7 The Logical IF Statement and Logical Statements

A logical IF statement is written in the form:

IF (e)S

where (e) is a logical expression and S is any other executable statement except an IF or DO statement. The simplest form of logical expression is one that asks a question about two arithmetic expressions.

The course of action taken by the logical IF statement is as follows:

If the logical expression is true statement S is executed:
If the logical expression is false, statement S is not executed. The next statement executed is the one following the logical IF unless S was GOTO and the expression was true.

The power of the logical IF statement can be heightened by the inclusion of logical operators these are written in the form:

.AND.

.OR.

.NOT.

A logical expression combines logical values and/or relational expressions. A relational expression is one that compares integer, real, or double precision values by using relational operators.

Relational operators are written in the form:

Relational Operators	Meaning
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not~~e~~ equal to
.GT.	Greater than
.GE.	Greater than or equal to

A logical assignment statement is written in the form:

Logical variable = logical expression

If L1, L2 and L3 are logical variables, the logical assignment statements written:

```
L1   =   D.LT.EPS.OR.ITER.GT.20
L2   =   D.GE.EPS.AND.ITER.LE.20
L3   =   BIG.GT.TOLER.OR.SWITCH
```

would assign L1 the value .TRUE. if either or both of the relations were true, and .FALSE. if both the relations were false.

L2 would be given the value .TRUE. if and only if both the relations were true.

L3 would be given the value .TRUE. if the relation was satisfied and/or the logical variable SWITCH was true.

The logical operator .NOT. reverses the truth value of the expression in which it is declared.

The .AND., .NOT. and .OR., .NOT. combinations are the only ones in which two operators may be declared adjacent to one another.

An example of usage of the IF statement follows. The arithmetic problem:

y = 0.5X+0.75 if $X\leq 3$

y = 0.25X+0.12 if $X>3$

can be accomplished by combining two IF statements thus:

```
IF(X.LE.3)Y=0.5*X+0.75
IF(X.GT.3)Y=0.25*X+0.12
```

If X is less than or equal to (.LE.)3, the statement Y=0.5*X+0.75 will be executed which is the correct formulae for computing y in that case. If X is greater than (.GE.) 3 the first IF statement is non-executed, but the second IF statement is.

3.4.8 CONTINUE Statement

A CONTINUE statement is a dummy (executable) statement that causes no action when the object program is executed. It is mainly used at the end of a DO statement to satisfy the rule that the last statement in the range of a DO statement must not be one that can cause transfer of control.

3.4.9 DO Statement

This statement makes it possible to execute a section of a program repeatedly with automatic changes in the value of an integer variable between repetitions. It is written in either of the forms:

DO n i = m_1, m_2

DO n i = m_1, m_2, m_3

where n = statement number of the last statement to be obeyed before attempting to repeat the loop with a new value of i.

i = unsubscripted integer variable written without a sign.

m_1 = the value to be assumed by i the first time the DO loop is traversed.

m_2 = must be greater than m_1 and is the greatest value which i may assume (It need not be attained nor, if $m_3 > 1$, need it be attainable).

m_3 = is the constant interval of the arithmetic progression of values assumed by i (must be > 0). If this value is ommitted, the value 1 is assumed.

'm3' is added to i at the end of each traverse of the DO loop. If the new value of i is $\leq m_2$, the loop is traversed again, otherwise the loop is terminated and the next executable statement is obeyed.

m_1, m_2 and m_3 may be unsubscripted integer variables or integer constants.

When the DO loop is left (because if i were again incremented it would exceed m_2) the value of i is undefined. However, if the DO loop is left by means of a GOTO statement then the value of i is preserved.

The values of i, m_2 and m3 may not be altered within a DO loop. But for one exception a GOTO statement may not cause a jump into a DO loop that by-passes its initial DO statement. The exception is when the DO loop is itself left by a GOTO and the values of i, m_2 m3 have not been altered; since, a GOTO statement may be used to re-enter the DO loops. DO loops must have the following properties:

(i) The first statement must bean executable statement.

(ii) They may occur within DO loops.

(iii) They may not intersect each other.

(iv) Terminating statement of a DO loop must not be a GOTO statement (including IF statement) nor a RETURN, STOP, PAUSE or DO statement. When this situation is required a CONTINUE statement should be used.

The following is an example of correctly nested DO loops

```
      DO  307 I   =  1, 10
      DO  307 J   =  1, 30, I
      DO  314  INDEX  =  5, 15
              A(I, J) = A (I, J) - B (INDEX)
314   CONTINUE
      DO  330 INDEX = -10, 0, 3
      DO  329 K = LLB, 12
      IF (B(INDEX+10)) 345, 329, 345
329   A (INDEX+10, K)  =  0
330   CONTINUE
      DO  307  K  =  1, 10
      ..    ..    ..    ..
      ..    ..    ..    ..
      GOTO 307
345   G = G + 1
      GOTO 329
307   CONTINUE
```

3.4.10 PAUSE Statement

An executable statement that causes the computer to wait until the operator indicates that compilation is to continue. The programmer should usually precede it by a statement which causes a message to be displayed informing the operator of some special action to be taken (e.g. load next data tape). The PAUSE may be followed by up to 5 octal digits. These will be displayed on the teleprinter.

3.4.11 STOP Statement

An executable statement which causes the computer to abandon the current calculation. The program cannot be continued, but it may remain in store, so that it can be re-entered at the beginning with a new set of data. In automatic operating systems STOP will cause the program to be overlayed and lost from core.

3.4.12 END Statement

An executable statement which

(i) Informs the compiler that it has reached the physical end of the program or sub-program that it is currently translating.

3.4.13 RETURN Statement

An executable statement which may only occur in a sub-program, indicates that the sub-program has completed calculation; and so control is to be returned to the program or sub-program which called it (i.e. at the executable statement following its call).

CHAPTER 4: SPECIFICATION AND DATA STATEMENTS

Five types of specification statements are permissible in 905 FORTRAN. They are:

a) DIMENSIONS statements

b) COMMON statements

c) EQUIVALENCE statements

d) EXTERNAL statements

e) The various 'Type' statements, i.e. INTEGER, REAL, DOUBLE PRECISION, COMPLEX and LOGICAL.

DATA initialisation statements are also considered in this chapter. Specification and DATA statements are said to be 'non-executable' i.e. they only provide information to the FORTRAN compiler and do not directly result in the creation of instructions in the program to be run.

4.1 DIMENSION Statements

A DIMENSION statement is used to indicate that one or more identifiers are the names of arrays (subscripted variables). The statement is of the form:

DIMENSION V_1 (i_1), V_2 (i_2) ..... V_n (i_n)

where:

v is the name of a subscripted variable

i indicates the permissible range of values for that subscript.

Each i is either

a) An unsigned integer constant c indicating that v is a one dimension array (vector) of c elements, which may be referred to as: v(1), v(2).......v(c), or

b) A pair of unsigned integer constants c_1, c_2 indicating that v is a two dimensional array of c_1 * c_2 elements which may be referred to as v(1, 1), v(2, 1)....v(c_1, 1), v(1, 2)..........v(c_1, c_2).

c) A group of 3 unsigned integers c_1, c_2, c_3 indicating that V is a three-dimensional array of c_1*c_2*c_3 elements which may be referred to as:

v(1, 1, 1), v(2, 1, 1). .... v(c, 1, 1),
v(1, 2, 1). v(2, 2, 0). ....v(c1, 2, 1),
........v(c_1, c_2, c_3)

If, and only if v is the name of an express formal parameter of a sub-program (Section 6. 7). i can include the names of one, two or three integer variables which are also formal parameters of the sub-program (Section 6. 7).

Example:

a) DIMENSION A(10), B(5, 15), CAT(99)

b) DIMENSION A(1), B(5, J), C(I, J)

This could only occur in a sub-program which numbered B, C, I and J among its formal parameters. See Section 4. 6 for further examples of DIMENSION statements.

4. 2 COMMON Statements

It has been stated that each program unit has its own variable names; the name X in the main program is not necessarily taken to be the same as the name X in a sub-program. However, if it is necessary for the values of both X's to be the same, a COMMON statement (written in both the main-and-sub-program) can be used.

A COMMON statement is used, in general, to communicate data between program units (main programs, SUBROUTINES and FUNCTION sub-programs). It informs the compiler that a list of variables and/or arrays in one program unit is to Share the same block of core store as a similar list(s) of variables/arrays in other program units.

A COMMON statement takes one of the following forms:

COMMON list e.g. COMMON A, I, K, LAMBA
or
COMMON/x_1/list_1/x_2/list_2/ ....../x_n/list_n
e.g. COMMON/BLOCK/B, J, X/BLOCK2/Z, KK

This second form is described more fully in 4. 2. 1. The first form describes a block of store locations termed 'blank' or 'unlabelled' common. The compiler allocates this block of store and assigns the list of variables/arrays to this block in the order they appear in the list.

If there is another COMMON statement in the same program unit, the first item in that statement is allocated store following the last item in the previous statement.

The list or lists may consist of: variables of any tape, array names of any type, or array declarators of the form V(i). V(i) takes the same form and same meaning as the V(i) in a DIMENSION statement, see 4. 1.

It is often convenient to have the same identifiers used in the corresponding positions of COMMON lists in different program units. However, there is no need for the names to be identical; only their order within the COMMON list is important.

4.2.1 COMMON Statement with Named COMMON

Apart from the block known as blank common area, there may be one or more labelled (named) common blocks. Tha names of such blocks are identifiers chosen by the programmer. In choosing the name the programmer must not use the name of any program unit (FORTRAN or MASIR) or intrinsic function; there are no other restrictions (The name has no implicit or explicit type, and it can even be the same name as a variable in the program, without having any relationship with that variable). The compiler uses the name of a common block only to allocate the block to the same core store area as common block(s) with the same name in other program units.

Named (or labelled) common blocks are described by the general form of COMMON statement:

COMMON/x_1/$list_2$/x_2/$list_2$/...../x_n/$list_n$

Example:

```
COMMON/BLOCK/A,B,C,K(20),Z(10,10)
```

Each x is the name of a common block. If a name is omitted between the slashes, the corresponding list describes blank common.

Example:

```
COMMON/B1/X,Y/ /I,J(20)
```

I and J are in blank common.

If the blank common is the first described, the two slashes may be omitted, for example:

```
COMMON  I,J(20)/B1/X,Y
```

would have exactly the same effect as the previous example.

If a named or blank common block is defined in more than one COMMON statement in a single program unit, the associated lists are strung together in the order in which they appear; this enables the compiler to work out the positions and total size of COMMON.

Within one complete executable program, the size of each named common block must not be greater in any program unit than that allocated in the first unit, encountered by the Loader, in which it occurs. A Loader warning is printed if the sizes are not the same. There is, however, no such restriction on the size of the blank common area.

Assume the two statements which follow were written in a main program and in a sub-program:

```
COMMON A,B,C/B1/D,E/B2/F(20),G(2,5)

COMMON R,S,T/B1/U,V/B2/X(10),Y(10,2)
```

Blank COMMON would contain A, B, C in that order in the program containing the first COMMON, and R, S, T in the program containing the second COMMON. A and R would thus be assigned to the same storage location as would B and D and C and T. The COMMON block labelled B1 would establish D and U in the same location and E and V in the same location (assuming all the foregoing variables were not mentioned in a DIMENSION statement elsewhere). B2 in the first program contains the 20 elements of F and the 10 elements of G. The same 30 locations would also contain the 10 elements of X and the 20 elements of Y. The overlap between the four arrays involved would cause the compiler no difficulty (the compiler would not need to even consider the situation).

4.3 EQUIVALENCE Statements

EQUIVALENCE statements are written in the form:

EQUIVALENCE $(k_1), (k_2), \ldots\ldots. k_n$

where:

each k is a list of the form:

$a_1, a_2, a_3, \ldots\ldots\ldots. a_m$ each a is the name of a variable or an array element.

An EQUIVALENCE statement assigns two or more variables within the same main program or within the same sub-program to the same storage location.

If one large array is to be equivalenced to one or more small arrays and all are to be in COMMON, the larger array must be declared in COMMON and the smaller not explicitly declared in COMMON.

An example of an EQUIVALENCE statement is given in section 4.6.

The array element name must have only constant subscripts. It is possible to use a single constant subscript for an array with two or three dimensions e.g.

```
DIMENSION A(3,4)

EQUIVALENCE (X,A(5))
```

This would cause X to share the same store locations as element A(2,2).

If two variables occupying different numbers of computer words are equivalenced together, the first word of each variable occupies the same storage location.

The effect of an EQUIVALENCE statement may add a variable or an array to a common block. This may cause an increase in the size of the common block. However, an EQUIVALENCE statement must not extend a common block 'backwards' i.e. alter the position of the first variable or element in the block.

Examples:

DIMENSION A(20)

COMMON/BX/X

EQUIVALENCE (X, A(1))

are valid, and would cause block BX to occupy 20 store units (i.e. 40 words), but:

EQUIVALENCE (X, A(2))

would be illegal, since X is the first location of BX, and this would put A(1) before X.

4.4 Restrictions on Sequence of Items in Equivalence Group

In an EQUIVALENCE group (i.e. a set of parenthesised items in an EQUIVALENCE statement), there are two restrictions on the sequence of items, they are:

1) If a group of equivalenced items includes an item in COMMON, that item must be the first in the group.

2) If the same appears in more than one group, that name must appear at the beginning of the second and any subsequent group in which it appears.

4.5 Restrictions on Names in Specification Statements

Within a single program unit, a name may occur in any or all of the following statements:

DIMENSION

COMMON

Type statements

The following rules are to be observed:

1) A name may not occur in any of the forms of statements given in the previous paragraph more than once.

2) A name may not be declared as an array (by having dimension information) in more than one statement.

3) A formal parameter (dummy argument) must not appear in a COMMON or EQUIVALENCE statement.

4) The name of a COMMON block must not correspond to the name of any sub-program whether in FORTRAN or MASIR.

4.6 Examples of Statements

In relation to the store map given following this paragraph the specifications in program CAT are:

SUBROUTINE CAT

DIMENSION A(5), I(3, 2), B(2), L(3)

COMMON A, I, J, G

COMMON K, LL

coupled with the specification sub-program DOG (see map):

SUBROUTINE DOG

DIMENSION TAIL(6), L(3)

COMMON EAR, BARK, TAIL, F, K, BK

EQUIVALENCE (BK, L(1))

this would lead to the allocation of space (in the first 22 locations of the COMMON area) indicated in the map below assuming the "packed integers" option was in use.

LOCATION	VARIABLES DECLARED IN	
	CAT	DOG
COMMON	A(1)	EAR
+1		
+2	A(2)	BARK
+3		
+4	A(3)	TAIL(1)
+5		
+6	A(4)	TAIL(2)
+7		
+8	A(5)	TAIL(3)
+9		
+10	I(1, 1)	TAIL(4)
+11	I(2, 1)	
+12	I(3, 1)	TAIL(5)
+13	I(1, 2)	

LOCATION	VARIABLE DECLARED IN	
	CAT	DOG
+14	I(2, 2)	TAIL (6)
+15	I(3, 2)	
+16	J	F
+17		
+18	G	K of DOG
+19	K of CAT	BK L(1)
+20	LL	L(2)
+21		L(3)

4.7 Use of Store Map

Programmers using COMMON and EQUIVALENCE are advised to prepare a store map similar to that given in section 4.6 Effects like the overlap of J and G of CAT with F and K of DOG are not erroneous, but their effect is unlikely to be that desired by the programmer. If used correctly, COMMON and EQUIVALENCE statements save space and simplify the calling and construction of sub-programs. If used incorrectly, they can cause chaos.

4.8 Type and EXTERNAL Statement

4.8.1 Type Statement

These consist of one of the declarations:

INTEGER

REAL

DOUBLE PRECISION

COMPLEX

LOGICAL

followed by as many variable names as necessary (separated by commas).

Examples:

```
REAL J

INTEGER B

INTEGER I, ABC, ROOTS

REAL MATRIX, NUMBER, X
```

```
DOUBLE PRECISION DENOM,REVX,TERM,N
COMPLEX T,N1,N2,D1
LOGICAL A1,A2,K
```

A type statement is used to inform the compiler that the given names are to be associated with variables of the appropriate type. In the case of Double Precision, Complex and Logical variables a Type statement must be used. For real and integer variables a Type statement may be used to override the implicit type suggested by the first letter of the identifier.

The Type statement must precede the first use of the name in any executable statement in the program. In the examples, the I from the INTEGER statement and the variable X from the REAL statement may be omitted since the names are identified integer and real respectively, by their first letters.

4.8.2 EXTERNAL Statement

An EXTERNAL statement has the form EXTERNAL $X_1, X_2, X_3, \ldots, X_n$ where each X is the name of an external subroutine or function.

905 FORTRAN permits the use of a function name as an argument in a sub-program call. When this occurs, it is necessary to list the function name in an EXTERNAL statement in the calling program, to distinguish between a function name and a variable name,

Example:

```
         EXTERNAL SIN,COS,SQRT
         CALL SUBR (2.0,SIN,RESULT)
         WRITE (6,129)RESULT
129      FORMAT (10H  SIN(2.0)=,F10.6)
         CALL SUBR (2.0,COS,RESULT)
         WRITE(6,130)RESULT
130      FORMAT(10H  COS(2.0)=,F10.6)
         CALL SUBR(2.0,SQRT,RESULT)
         WRITE (6,131)RESULT
131      FORMAT(11H  SQRT(2.0)=,F10.6)
         STOP
         END
_______  _______
         SUBROUTINE  SUBR(X,F,Y)
         Y=F(X)
         RETURN
         END
```

This program contains a main (calling) program and a SUBROUTINE sub-program. The program contains only one executable statement viz.

```
Y=F(X)
```

The arguments listed are X, F and Y making the function F a matter of choice in the sub-program call. The main program calls this sub-program three times. Each time the value of X is 2.0 and the actual variable corresponding to Y is RESULT. The arguments corresponding to F are successively SIN, COS, SQRT; these three supplied function names are listed in an EXTERNAL statement.

4.9 DATA Statement

The use of the DATA statement is brought about when it becomes necessary to couple data (from the source program) into the object program. DATA statements take the form:

DATA list/$d_1, d_2, \ldots\ldots d_n$/, list/$d_1, d_2, K{*}d_3 \ldots\ldots\ldots d_m$

In this symbolic description a 'list' contains the names of variables to receive values, the d's are values and the K (if used) is an integer constant.

Example:

```
DATA A,B,C/14.7,62.1,1.5E-20/
```

This statement would assign the values 14.7, 62.1 and 1.5 X10^{-20} to A, B and C respectively. This action is performed at the compilation and not at the time when the object program is executed.

The values assigned by the DATA statement are placed in storage when the object program is loaded and that is end of the actions required by the DATA statement.

It is legal to redefine values of these variables but having so acted it is not possible to re-execute the DATA statement to put the variables back to their original value.

The two statements which follow have identical meanings, choice of statement is a matter of personal preference:

```
DATA A/60.75/,B/10.0/,C/5.0
DATA A,B,C/60.75,10,5.0
```

Any of the constants may be preceded by a multiplier, that is an unsigned positive integer constant and an asterisk. If the multiplier has the value n this is equivalent to writing the constant it precedes n times.

Example:

```
DATA X,Y,Z,W/3*0.0,1.0/
```

This will cause X, Y and Z to have initial values of zero, and W to have a value 1.0.

NOTE: If these values are changed, they will only be reset if the program is reloaded into core store.

The statement which follows assigns the value 10.5 to all five variables:

```
DATA A,B,C,D,E/5*10.5/
```

A DATA statement maycontain Hollerith text, for example:

```
DATA DOT,X,BLANK/1H.,1HX,1H/
```

If the number of characters of text is not the same as the number of characters in a storage location, the characters are left justified and space filled. In the example given, the point would be left justified in DOT and the remainder of DOT filled with blanks; X would be similarly treated. BLANK would be filled with blanks as intended.

The items in the list must not be in COMMON (blank or named), nor can they be formal parameters (dummy arguments). They may be subscripted variables with constant subscripts.

4.10 Restrictions on the Sequence of Items within a Subprogram

In 905 FORTRAN the statements which make up a program unit must appear in the following sequence.

1. SUBROUTINE or FUNCTION (except in a main program)

2. Specification statements

3. DATA Statements

4. Statement function definitions

5. Executable statements, FORMAT statements and in-line machine code.

6. END statement

CHAPTER 5: INPUT AND OUTPUT

To read input or write output data requires the programmer to declare four categories of information in the source program. They are:

1) Selection of input or output device, which is handled by a combination of the statement verb and the unit designation.

2) The variables to which new input values are to be designated or whose values are to be sent to an output device. These are specified by the list of variables in the input or output statement.

3) The order in which the values are to be transmitted, is governed by the order in which the variables are named in the list.

4) The format in which the data appears for input, or is to be written for output. This is specified by a FORMAT statement which must be referenced by the input or output statement in all but a few special cases (see Sect. 5.4)

5.1 Input and Output Statements

Input and Output statements take the form:

READ (u, f) k or READ (u) k

WRITE (u, f) k or WRITE (u) k

where u represents an integer constant or variable indicating a device (see table which follows).

f represents the statement number of FORMAT statement (see Section 5.4). If 'f' is absent, statements are known as unformatted; otherwise they are known as formatted. The name of an array may be used in place of f (see Section 5.7).

k represents a list of items to be input or output. The list may contain variables, subscripted variables, array names and DO-implied lists.

The value of u must be in the range 1 to 10.

Value of u	Device referred to in a	
	READ Statement	WRITE Statement
1	Paper Tape *	Paper Tape *
2	-	-
3	Teleprinter *	Teleprinter *
4	-	Lineprinter
5	-	Digital Plotter
6	-	-
7	Card Reader	-
8	-	-
9	Display Keyboard	Display Screen
10	Special Devices	Special Devices

The standard run-time package for paper tape systems has the device marked * pre-set (i.e. automatically available). Input or Output to any other number in the range 1 to 10 is diverted to the paper tape reader (using READ Statement) or punch (using WRITE Statement). If any other numbers are to be used, the device routines should be set before obeying the READ or WRITE statement e.g. a call of QLPOUT sets device number 4 for output of information to the line printer. Device numbers 10 for input and output is reserved for special on-line devices (normally interfaced by the user) and will not be used for any standard hardware peripherals.

5.2 The List of an Input or Output Statement.

The simplest type of list is one in which all the variables are named explicitly and in the order in which they are to be transmitted. The fundamental idea of 'scanning' carries through; the first data field is associated with the first variable name and first field specification (in the associated FORMAT statement), and so on.

However, when entire arrays or parts of arrays are to be transmitted it is not necessary to name each element explicitly. When transmitting an array, it is only necessary to name the array in a list excluding any subscripts. The name of the array must appear elsewhere in the program in a specification statement that gives dimensioning information, but in the list it need not carry any subscripting information. The elements may (but need not) have the same field specification; this one field specification may be given by itself in the FORMAT statement.

For example:

```
      DIMENSION A(10,5)
      WRITE (6,21)A
21    FORMAT (1PE20.8)
```

would cause output of all 50 elements of array A, one to a line (in 1PE 20.8 format).

When an entire array is moved this way, the elements are transmitted in a sequence in which the first subscript varies the most rapidly and the last subscript the least rapidly.

When only some of the array elements are to be transferred or when the 'natural order' just mentioned is not required, it is still possible to avoid naming each element explicitly. The elements can be specified instead in the list required in a way that parallels a DO loop.

Example:

Suppose the array 'BLOCK' has 90 locations, in the form of 15 rows and 6 columns, of which only 24 of these are required for output. These locations are located between rows 7 and 14 and columns 2 and 4. The WRITE statement could be in the form:

```
      DO 50 I = 7,14
      DO 50 J = 2,4
50    WRITE(3, 51) BLOCK(I,J)
51    FORMAT(3I6)
```

With DO-implied lists this could be written:

```
      WRITE (3,51) ((BLOCK (I,J), J = 2,4), I = 7,14)
51    FORMAT (3I6)
```

In general, a list of a READ or WRITE statement can be made up of variables, subscripted variables, array names and DO implied lists. If there is more than one item in the list they must be separated by commas.

A DO-implied list is made up in the general form:

(List, I = m_1, m_2, m_3)

where I represents any integer variable and m_1, m_2, m_3 have the same meaning as in a DO statement (Chapter 3).

The 'List' may be made up of variables, subscript variables, array names and DO-implied lists; which means that DO-implied lists can be "nested", as shown in the previous example. The innermost DO-implied loop is executed most frequently.

5.3 Effect of Numeric Items in READ and WRITE Lists

Item	Effect in a READ list	Effect in a WRITE list
A simple variable (which does not occur in a DIMENSION statement) e.g.	A number is input from the specified device and its value is assigned to the variable	The value of the variable in output to the specified device
A subscripted variable e.g. B(7,J)	As for a simple variable (See Note 1)	As for a simple variable (see Note 1)
The name of a DIMENSIONED variable	The appropriate number of values is read from the specified device and they are assigned, in order, to the elements of B	The value of the elements of B are output in order.

NOTE: The identity of the items in a READ or WRITE statement is determined before the values of any item in that list are input or output. Consequently if the next two items on a data tape are 3 and 3.14159, the effect of

```
I=7
READ(1,1)I,A(I)
```

is to assign the value 3 to I and the value 3.14159 to A(7) (the value of A(3) will remain unaffected).

5.4 FORMAT Statement

The function of a FORMAT statement is to declare how information is to be arranged either on input or output. To each value transmitted there must correspond a field specification which lists the kind of information and the layout details of the value contained in that field (in terms of its internal representation and what it 'looks like' externally).

5.4.1 General Form of FORMAT Statements

The form generally used for a FORMAT statement is the word FORMAT, followed by a list of one or more items enclosed in parenthesis, that is :

```
FORMAT(List)
```

In 905 FORTRAN, 'List' may consist of the single word FREE which indicates free format for use on input only. However, standard FORTRAN does not include this facility since FORTRAN was mainly used in a card input environment. When data is input from cards, each item is normally punched in a fixed column width with a fixed number of items per card. Using paper tape input it may be inconvenient if the value 2.0 has to be filled out with 10 spaces because the number 12345678E-5 has to be punched in a corresponding data field. For similar reasons, it may be inconvenient to always have a fixed number of items per line of text. Therefore 905 FORTRAN allows both fixed (standard) and free format input.

Example:

```
   READ(1,7) I,J,X,Z
7  FORMAT(FREE)
```

will cause the next four numbers to be read from a data tape, and assigned to I, J, X, Z with the correct value type (see Section 5.8 for further details of free format input). On output, it is both convenient and essential to specify how many character positions are occupied by each item output and the form of output e.g. floating point or fractional , number of significant digits etc..

Standard FORMAT statements thus consist of a set of field descriptors which specify the width of a field (i.e. the number of character positions on the external medium), the corresponding type of internal representation and other necessary information for output control. When a READ or WRITE corresponding to a FORMAT statement is obeyed, each item present (or implied) in the READ/WRITE list is matched against a field descriptor in the FORMAT list by a scanning process which works through both lists in parallel.

Special rules are used to:

(i) avoid repeatedly writing identical field specifiers

(ii) cater for READ/WRITE lists when they are longer than the FORMAT statement lists.

Each field descriptor implies a conversion between a number or a group of characters represented on an external medium, and an internal representation of the same item within the computer. The internal representation may consist of binary numbers, packed internal code characters, floating point etc.. For most purposes, the programmer only needs to be aware of the type of internal representation available i.e. integer, real, Hollerith etc.. The external medium may be paper tape, teleprinter or any other character handling device with suitable software.

A general FORMAT list is made up of field descriptors separated by field separators (either commas or slashes). Field descriptors consist of one of the letters I, F, E, G, D, A, H, X, L followed by either number or characters conveying special information. For example, using the I descriptor to control integer conversion.

```
      WRITE(1,99)I,J
99    FORMAT(I2,I4)
```

would cause the two integer values in I and J to be output in field of two and four characters respectively.

5.4.2. Repeat Counts

All the descriptors except X and H may be preceded by a repeat count (r) indicating that the descriptor is treated as though it were written r times.

Example:

```
            FORMAT(3I5)
```

is equivalent to: FORMAT (I5, I5,I5).

A group of field descriptors may be enclosed in parentheses to make a basic group (The basic group may be preceded by a repeat count). Field separators and basic groups may be further grouped by enclosing in parentheses with a repeat count. The "nest" of groups must not be more than two in depth.

Example:

```
      FORMAT (2(I6,3(I4,I3)))
```

is equivalent to:

```
      FORMAT(I6,I4,I3,I4,I3,I4, I3,I6,I4,I3,I4,I3,I4,I3)
```

If the READ/WRITE list contained more than 14 integers, these statements would not have exactly the same effect (see section 5.4.3).

5.4.3 External Records and Newlines

Apart from separating text, the separator / (slash) is also used to start a new record (for punched card input a record is defined as one card). In general, a record on paper tape is a string of characters (text) followed by newline, although the FORTRAN standard does not clearly define this. In 905 FORTRAN, the separator / in a FORMAT statement causes on output a newline sequence to be punched, and on input causes characters up to and including the next newline (linefeed) character to be skipped. The end of a FORMAT statement scan, when the last right parenthesis is reached, produces the same effect.

There may be more than one slash between field descriptors indicating multiple newlines, and a group of slashes may be used at the beginning or end of a FORMAT statement.

Example:

```
      WRITE(3,97)I,J
97    FORMAT( //I3/I4 /)
```

This would cause output of two newline sequences, a three character integer, newline and a four character integer followed by two newlines. Of the last two newlines one is causes by / separator, and the other by the end of the FORMAT scan.

If in the par~~r~~allel scan of a FORMAT statement and READ/WRITE list, the former list is exhausted before the latter; format control returns to the beginning of the FORMAT list, or if there are nested groups of descriptors, to the repeat count at the start of the group which ended most recently. In either case a skip to a new record or output of new line occurs.

Example:

```
        WRITE(1, 999)M, ((IA(J, K, J = 1,  2),  K= 1, 3)
999     FORMAT(I6, 2( /I3, I4))
```

causes output of the values as follows:

```
M
IA(1, 1) IA(2, 1)
IA(1, 2) IA(2, 2)
                          Extra newline for
                          end of format.
IA(1, 3) IA(2, 3)
```

5.4.4 Field Descriptors Available

There are nine field descriptors available in 905 FORTRAN, which are written in the following symbolic forms:

s r E w.d

s r F w.d

s r G w.d

s r D w.d

r Iw

r Lw

r Aw

n H $h_1, h_2, h_3, \ldots\ldots h_n$

n X

where:

s represents a scale factor in the form kP which can be omitted if not required (k is an integer constant, optionally signed).

r represents a repeat count which may be omitted if not required. It is written as a positive unsigned integer constant.

w represents the width of the field on the external medium in characters. It is written as an unsigned positive integer constant.

d represents the number of digits after the decimal point in a real or double precision number. 'd' is an unsigned positive integer.

n represents the number of characters in a field.

The effect of the various number descriptors (I, F, E, D, G) on input is described in section 5.4.6. The effects of output are described under individual headings for each descriptor (section 5.4.7 and onwards).

5.4.5 Scale factors.

A scale factor may be written before an E, F, G or D format descriptor, in the form:

jP

where j is an unsigned positive integer constant. or a signed negative integer constant. Once a scale factor has been specified, this factor will apply to all F, E, G and D field specifications which follow(in the rest of the FORMAT statement processing), unless cancelled by another scale factor. An implied scale factor of zero is set up when a READ or WRITE Statement commences. A scale factor has no effect on I, L, A, H or X specifications.

A non zero scale factor has different effects on input and output, On input, for F, E, G and D specifications if there is an exponent in the input field, the scale factor is ignored. If the external input field does not contain an exponent, the internal number = external number divided by 10^n, where n is the scale factor.

For the effect of scale factor on output, see the separate descriptions of F, E, G and D specifications.

5.4.6 Input of Numbers Under Format Control.

Numbers are input under format control (as opposed to free format) by the descriptors I, E, F, G, D. In each case a field of w characters is input, that is the next w significant characters are read from the external source. Line feed (newline), carriage return, null and erase are all ignored by the compiler.

For integer conversion I the external field must be in one of the forms permitted for integer constants (signed or unsigned).

For real and double precision conversions (E, F, G and D), the external field may be signed or unsigned, with a string of digits which may or may not contain a decimal point. These digits may, but need not, be followed by an exponent in one of the following forms:

	Example of complete number.
+ integer constant	.0009+3
- integer constant	0.000-1
E integer constant	0.009E2
E signed integer constant	90.0E-2

D integer constant	.009D02
D signed integer constant	bb90D-2
(No exponent)	0.9bbbb

In the examples b represents space (blank). All these numbers could be input under control of F7.0 to give the same internal value.

If a decimal point occurs in the field the d value is ignored. If an exponent is used the number is raised to that power of ten, and any scale factor ignored. If there is no exponent, the internal number = external number divided by 10^n, where n is the current scale factor (zero if none specified).

Spaces (blanks)are significant in formatted input, they are treated as zeros. If spaces occur at the beginning of the field they are generally regarded as non-significant zeros, but at the end of the field they may have some effect, particularly if there is no decimal point or if there is an exponent.

An all blank field represents zero.

5.4.7 Field Specification I (Integer)

This takes the form Iw where I specifies conversion between an internal integer and an external decimal integer. 'w' specifies the total number of characters in the field, including any sign or blanks.

Examples:

(i)
```
9 FORMAT (I6)
  READ (1,9) J
```

On input this would cause 6 characters to be read from paper tape, converted to integer form and stored in variable J.

(ii)
```
J = -987
WRITE (3,9)J
```

On output this would cause the number -987 to be output on the teleprinter, with two spaces, a minus sign and digits 987 (a total of six characters).

5.4.8 Field Specification F (external fixed point)

The form of this specification is Fw.d,where F indicates conversion between an internal real value and an external number written without an exponent. The letter w specifies the total number of characters in the field, including sign, decimal point and any blanks; d specifies the number of decimal places after the decimal point.

For the effect of F format on input, see 5.4.6. On output, there will be d digits to the right of the decimal point (Spaces are inserted for leading zeros).

The scale factor may be used with the F field specification by writing the specification in the form:

sPrFw.d

where s= scale factor (scale factor may either be positive or negative)

r= repetition number

The effect of scale factor on output is that: external number = internal number $*10^s$.

5.4.9 Field Specification E (Floating Point)

The form of this specification is Ew.d where E specifies conversion between an internal real value and an external number written with an exponent. The total number of characters in the external medium is w, including sign, decimal point, exponent and any blanks. The number of decimal places after the decimal point (not counting the exponent) is specified by d.

Example:

```
    Y = 1.5E-2
    X = -123.4567
    WRITE(3,9) X,Y
9   FORMAT(2E13.6)
```

This would cause output as follows:

-0.123457E +03b0.150000E-01 (where b represents c a space or blank)

If a scale factor is used on output, it causes the fractional part to be multiplied by 10^s and the exponent to be reduced by s. For example, if the previous FORMAT statement were:

```
9   FORMAT (1P2E13.6)
```

the output would be

-1.234567E+02␣1.500000E-02

In 905 FORTRAN the standard form without scale factor, for example:
.10000E+03
will sometimes be output as:
1.00000E+02

5.4.10 The Field Specification G(Freepoint)

The form of specification is G w.d. The internal value must be of type real. On input, the G w.d specification is treated as for F w.d specification. On output, a field of width w words with d significant characters is output, according to the following rules:

If N is the magnitude of the value to be output, the sPG w.d specification produces an equivalent conversion as follows:

Magnitude	Equivalent output conversion
$0.1 \leq N < 1$	F(w-4).d,4X
$1 \leq N < 10$	F(w-4).(d-1),4X
$10^{d-2} \leq N \leq 10^{d-1}$	F(w-4).1,4X
$10^{d-1} \leq N < 10^{d}$	F(w-4).0,4X
Other values	sP Ew.d.

The scale factor has no effect unless output is in the Ew.d form.

Example with G10.4

-123.45	output as	-123.4bbbb
-12.345	output as	-12.34bbbb
-1.2345	output as	-1.234bbbb

where b represents space (blank) character.

5.4.11 Field Specification D (double precision)

In the Dw.d specification, the corresponding internal value must be double precision. The exponent in the output field is written with D instead of E, but in all other respects this is analogous to the E field specification.

5.4.12 Field Specification L(logical)

In the Lw form, the L specifies conversion between an internal logical value (.TRUE. or .FALSE.) and one of the letters T or F externally. The total number of character positions is specified by w.

On input, the external field may contain spaces (which are optional), the letter T or F, followed by any other characters which would fill up the remaining w positions. On output, the external field consists of (w-1) spaces (blanks) followed by a letter T or F.

5.4.13 Conversion for Complex Numbers.

A complex number is input or output as though it were two real numbers, the 'real part' followed by the 'imaginary part'. Therefore there must be two real format conversions (F, E or G) in the FORMAT statement, corresponding to each complex variable or array element in the READ/WRITE list. Only in free format input (q.v) are complex numbers specially treated.

5.4.14 Field Specification A (Alphanumeric)

In the Aw form of field specification, the associated variable may be of any kind. The field specification causes w characters to be read into or written from, the associated 'list' element. The alphanumeric characters may be any symbols representable in the internal code character set, including letters,digits and the character space (blank)

In 905 FORTRAN to every basic 'A' descriptor there must correspond one word in the input/output list. 'Aw' descriptor will only transfer one word (one storage unit) and at the most three characters. If, on input, w>3 then the first w-3 characters are ignored and the remaining 3 characters transferred into storage. If w<3 then the rightmost (w-3) characters appear as blanks in storage. On output if w>3 the first w-3 character will be blank in the output field.

Example 1 (I, J, K = integers)

```
        READ (1, 10) I, J, K
        WRITE (2, 10) I, J, K
10      FORMAT (A2, A3, A5)
```

Input: $\underbrace{\text{A-}}_{2}\underbrace{\text{FOR}}_{3}\underbrace{\text{MAT **}}_{5}$

Storage

I : A - b

J : FOR

K : T**

Output : A - FORbbT**

where b represents space (blank)

A field specification requires use of implied DO to handle arrays correctly

ADHeibel 24/06/12

Example 2 (F=real, E=real array)

```
        DIMENSION E(20)
        READ (1, 20) F, (E(I), I=1,2)
20      FORMAT (2A2, 4A3)
```

Input : $\underbrace{\text{TE}}_{2}\underbrace{\text{ST}}_{2}\underbrace{\text{ING}}_{3}\underbrace{\text{ b FO}}_{3}\underbrace{\text{RMA}}_{3}\underbrace{\text{T, *}}_{3}$

where b represents space (blank)

Storage

F : TEb

F + 1 : STb

E(1) : ING

E(1)+1 : bFO

Example 2 (Contintued)

```
    E(2)      :   RMA
    E(2)+1    :   T, *
```

Example 3 (N=packed integer array)

```
      DIMENSION N(20)
      READ (1,30)              (N(I), I=1,2)
      WRITE (2,30)             (N(I), I=1,2)
30    FORMAT (2A3)
```

Example 4 (M=unpacked integer array)

```
      DIMENSION M(20)
      READ (1, 40)             (M(I), I=1,2)
      WRITE (2, 40)            (M(I), I=1,2)
40    FORMAT (4A3)
```

5.4.15 Field Specification X (Skip)

This specification takes the form wX, where w is the field width. On output, w spaces (blanks) are inserted in the output text. On input, w characters are read and ignored (Newline, null, carriage return and erase are ignored on counting w). The X descriptor is not associated with an item in the READ/ WRITE list, but is activated after the action specified by the previous FORMAT descriptor.

N.B. The letter X must be followed by a comma, slash or right parenthesis.

5.4.16 Field Specification H (Hollerith)

This specification takes the form wH, where w characters immediately following the letter H are printed or punched in the position indicated by the position of the Hollerith field specification in the FORMAT statement. The Hollerith Field specification differs from the other specifications in as much as it does not call for the transmission of any values from the list, instead, it calls for the input or output of the text itself.

If an 'H' descriptor is used on input, w characters are read from the external medium, and stored in the FORMAT data replacing the w characters which follow the letter H (In counting w characters; newline, null, carriage return and erase are ignored). If the letter H is subsequently used for output, the new string of characters is output.

NOTE: Newlines cannot be stored in the H text, either in the original FORMAT, or by input using the letter H.

Example:

```
        WRITE(3, 8)
        READ (1, 9)
        WRITE (3,9)
8       FORMAT (6X, 7HHEADING)
9       FORMAT (20H01234567890123456789)
```

This reads a heading of twenty characters from the input paper tape, and outputs it on the teleprinter. Such input and output may provide a more machine independant form than the use of the 'A' descriptor for similar purposes. The last character of the Hollerith string must be followed by comma, slash or right parenthesis.

5.5 Examples of Field Specifications

a) Integer type
To output the numbers 16 and -64

i) on the same line, the FORMAT statement could be
FORMAT (I2, I3)
which will be output as
16-64

ii) on the same line but separated by three spaces use
FORMAT (I2, I6)
which will be output as
16 ⓢⓢⓢ -64

iii) on separate lines but under each other the FORMAT statement
FORMAT (I3)
which will output
ⓢ16
-64

b) External fixed point
To output the numbers -187.654 (with two spaces before the minus sign, the FORMAT statement would be
FORMAT (F10.3)
FORMAT (F8.5) will give -187.65400
FORMAT (3PF8.3) will cause a number 0.1234 giving the current in amperes to be printed as milliamperes;
b123.400

c) Floating Point
To output the number 497863.31, the FORMAT statement used, could be FORMAT (E14.8)
which would output;
0.49786331E+06

d) Logical

If the variable is .TRUE. then the FORMAT statement:

FORMAT (L2)

would output:

ⓢT

However, if the variable is .FALSE. the FORMAT statement

FORMAT(L6) will output

ⓢⓢⓢⓢⓢ F

If a FORMAT statement contains nothing but Hollerith and blank field specifications, there must be no variables listed in the associated input or output statement. This is common practice when the WRITE statement produces page and column headings or causes line and page spacing.

Example:

The two statements:

```
      WRITE (3,7)
7     FORMAT(5HYARDS, 8X, 4HFEET, 8X, 6HINCHES)
```

will output:

YARDSⓢⓢⓢⓢⓢⓢⓢⓢFEET ⓢⓢⓢⓢⓢⓢⓢⓢ INCHES

5.6 Number Out of Range on Output

If the character field (w) is not wide enough to contain the output value an asterisk is inserted in the high order position of the field. If the exponent is also printed, its absolute value must be less than 99, otherwise '**' replaces the exponent part in the output.

Examples:

Format	Value	Output
F6.2	3456.7	*56.70
F6:2	234.56	234.56
F6.2	-234.56	*34.56
F6.0	123.0	bb123.
F6.6	.123	*23000
F6.7	1.834	*34000
F6.4	0.123	0.1230
F6.5	0.123	.12300
D10.3	312.4E+100	0.124D+**

where b represents space (blank)

With E-format, the standard form which is

$$\pm 0.x_1x_2\ldots x_n \mathrm{E} \pm y_1y_2$$

occasionally becomes $\pm 1.00 \ldots 0 \mathrm{E} \pm y_1y_2$ when $x_1 = 1$, and $x_2 \ldots x_n = 0$ since the number is rounded after its format has been determined.

5.7 Run-time FORMAT Statement Input

The ability to read a FORMAT statement at the time of execution of the object program adds great flexibility to FORTRAN. In order to achieve this, an array must be declared which will hold the FORMAT specification in the form of alphanumeric data (see Field Specification A). The FORMATS are read into this array at run time. These variable FORMATS must reference the array by name in the READ or WRITE statement.

Example:

Suppose we have three variables to output but do not at the time of writing the program know the form of output. A one-dimensional array FMT which is of a suitable size,in this case 5 words is declared. The array is real, and has 10 locations in which a maximum of 30 alphanumeric characters may be stored. The format which is to be in the form:

(I6, 8X, F8. 3, 1PE20. 8)bbbbbbb

which consists of 23 characters plus seven spaces which make up the 30 alphanumeric characters. The program would be written as:

```
      REAL X, Y, FMT
      INTEGER I
      DIMENSION FMT(5)
      READ(1, 209)(FMT(J), I=1, 5)
209   FORMAT(10A3)
      .
      .
      .
      .
      .
      WRITE(3, FMT)I, X, Y
```

NOTE: It should be pointed out that in the data input from tape, the enclosing parenthesis of the FORMAT must be included, but the word FORMAT itself should be omitted from the data tape.

5.8 Free Format Jnput

The FORMAT statement for a free-format input operation is as follows:

FORMAT (FREE)

The effect of a READ statement which refers to such a FORMAT is to cause numbers to be read from the input paper tape, converted according to their appearance, and the resulting values assigned to successive items in the READ list (the latter being interpreted according to the standard rules for fixed-format). The operation is terminated when the end of the list is reached, and is temporarily halted by the appearance of a halt code on the input tape.

When reading free-format data, the mode of conversion is determined in the first place by the formation of the number on the input tape The value is then stored in the form appropriate to the type (integer, real, etc.) of the item in the READ list.

5.8.1 Data Tapes for Free Format Input

Integer, real, double-precision, and complex numbers may be punched on data tapes. They appear in the same form as constants of equivalent type in a source program, and each number is terminated by one or more spaces or line feeds. The real part of a complex number is terminated by ',' complex part by')'. Blank tape, carriage return, and erase are ignored; a halt code stops the program pending manual restart.

Acceptable characters are: digits, decimal point, +, -, D, E, comma, parenthesis, subscript 10, space, tab, line feed, carriage return, blank tape, erase, halt code. The appearance of any other character on the input tape will give rise to an error indication.

In a complex number, there must not be any spaces between the end of each number and the comma or parentheses.

5.8.2 Example of Free Format Input

The statements:

```
      COMPLEX C
      READ (1,100)F1,F2,J1,J2,J3,C
100   FORMAT (FREE)
```

with input data tape:

```
      10.3    10      7.6      2    5.3E1       (2.4,5)
```

will result in the following assignments:

```
      F1    =      10.3
      F2    =      10.0
      J1    =      7
      J2    =      2
      J3    =      53
      C     =      2.4
      C+2   =      5.0
```

CHAPTER 6: FUNCTIONS AND SUBROUTINES

6.1 Subprograms - General

Functions and subroutines form a means whereby a single FORTRAN statement may cause the computer to obey a section of program which may contain many statements. They may be used to obtain one or more of the following advantages:

(a) To save the programmer writing the same long statement or group of statements many times at different points in his program.

(b) To save core store, by avoiding the repetition of code performing the same or similar functions.

(c) To divide the program into units which may be compiled separately. This has the advantage that if an alteration is necessary in one unit, it is only necessary to re-compile that one unit.

(d) A second advantage of separate compilation is for convenience, especially when several programmers are sharing a task. They need not worry about clashes due to use of the same identifier for different purposes. The parameters and/or Common Blocks help to provide a defined interface.

(e) Once written a single subprogram may be used with different Main Programs.

6.2 Main Programs, Subprograms and Program Units

A complete program in the 905 FORTRAN system, with all the statements necessary to run it, is known as an executable program. It may consist of one or more program units.

Each program unit is either a Main program or a subprogram written in either FORTRAN or 905 MASIR assembly code. There must only be one Main program which should be written in FORTRAN (but could be written in MASIR code). A FORTRAN main program is identified by the absence of either a FUNCTION or SUBROUTINE statement at the beginning of the program; (when compiled the program unit takes on the name MAIN.)

A FORTRAN subprogram is either a FUNCTION subprogram or a SUBROUTINE subprogram, identified by the appropriate statement as the first significant line of text.

NOTE: A subroutine is sometimes referred to as a procedure.

No program unit in 905 FORTRAN may be so large that its compiled code plus the local arrays and variables (i.e. not in COMMON) exceeds

8100 words of computer storage.

6.3 Types of Procedure

In 905 FORTRAN, the following types of procedure can be used:

(a) Statement Functions.

(b) Intrinsic Functions.

(c) Basic External Functions.

(d) FUNCTION Subprograms.

(e) SUBROUTINE Subprograms.

Statement functions are single statements embedded within a program unit, and are not therefore classed as subprograms (they are described in detail later in this section).

Intrinsic functions are a set of functions provided with the 905 FORTRAN Compiler system, and listed in Appendix 1 Table A 1.2. Their names should not be used for any other purpose. They perform commonly required operations such as finding the absolute magnitude of a number.

Basic External Functions are a set of functions also supplied with the Compiler system. They perform useful Mathematical functions such as taking the square root, finding the sine etc. A number of other trignometric functions can be easily derived from the functions supplied, for example:

$$\text{arcsin}\,(x) = \text{arctan}\,(\text{sqrt}\,(x^2/(1-x^2)))$$

The differences between instrinsic and external functions are that, external functions may be mentioned in EXTERNAL statements and one may write external functions to replace the standard functions (if considered necessary). For example, the programmer may write a SQRT routine which took special action when a negative argument was given.

6.4 Subprogram Head

A subprogram head is declared in the form:

FUNCTION $f(m_1, \ldots\ldots\ldots\ldots m_k)$
SUBROUTINE $s(m_1, m_2 \ldots\ldots\ldots m_k)$
or SUBROUTINE s

where:

(i) f is the name of the FUNCTION and specifies its type in accordance with the implicit type rules (see (vi)).

(ii) s is the name of SUBROUTINE (apart from the Q Rule - see 2.5.3 - a subroutine name is not governed by set rules but, care must be taken to avoid clashes of names; therefore the choice of s is completely arbitrary).

(iii) $m_1, m_2, m_3, \ldots\ldots\ldots m_k$ are express formal parameters. Each m_i must be the name of a variable or an array, or a procedure. There must be at least one parameter per FUNCTION statement but there need not be any explicit parameters for a SUBROUTINE.

(iv) Each n_i which represents an array must appear in a DIMENSION statement within the body of the subprogram. In this DIMENSION statement the upper bounds of its suffices may be given either as integer constants or as integer variables which are themselves express formal parameters.

(v) In addition to the express formal parameters, a subprogram may refer to variables in COMMON, these may be regarded as implicit parameters.

(vi) The word FUNCTION may be preceded by one of the following:

REAL, INTEGER, DOUBLE PRECISION, LOGICAL or COMPLEX, which causes the appropriate type to be associated with the FUNCTION name.

6.5 The Subprogram Body

A subprogram body is subject to special rules as in a normal FORTRAN main program. They are:

(i) A subroutine does not have a value and no assignment may be made to its name. It may communicate information to the program that called it (main program or another subprogram) by altering the values of one or more of its parameters.

(ii) Within a FUNCTION subprogram, its name (f) acts as an ordinary variable of the appropriate type. It is undefined on entry to the FUNCTION but a value must be assigned to it, before exit is made from the FUNCTION subprogram.

(iii) In 905 FORTRAN, the alteration by a FUNCTION of any of its parameters is not considered to be an error. However, this should be avoided wherever possible, particularly as the evaluation of a FUNCTION statement may not validly alter the value of any other elements within any expression, assignment statement or CALL statement in which the FUNCTION appears.

(iv) A subprogram body may not itself contain a declaration of a subprogram.

(v) An explicit formal parameter may not occur in a COMMON or EQUIVALENCE statement (see CHAPTER 4).

(vi) When a subprogram has completed its computation, it returns control to the program that called it by means of a RETURN statement. This comprises of the word RETURN on a new line.

(vii) The body of a subprogram is terminated by an END statement. This comprises of the word END on a new line.

6.6 Examples of Function and Subroutine Subprograms

```
  FUNCTION   MAX(I,J)
  IF (I-J) 1,1,2
1 MAX = J
  RETURN
  MAX = I
  RETURN
  END

  SUBROUTINE MTXMLT (A, N, M, B, L, C)
  DIMENSION A(N, M), B(M, L), C(N, L)
C C BECOMES A TIMES B .
  DO 1 I = 1, N
  DO 1 K = 1, L
  D = 0.0
  DO 2 J = 1, M
2 D = D+A (I, J) * B(J, K)
1 C (I, K) = D
  RETURN
  END
```

6.7 Calling a Subprogram

(1) A FUNCTION subprogram is activated by writing:

$$f(m_1, m_2, \ldots\ldots\ldots m_k)$$

in some statement which can make use of the value of f.

(2) A SUBROUTINE subprogram is activated by a call statement, which takes the form:

$$\text{CALL } s(m_1, m_2, \ldots\ldots\ldots m_k)$$

The FORTRAN word CALL must be terminated by at least one space.

(3) If an express formal parameter (integer variable) is used as a subscript bound then the corresponding actual parameter must be an integer variable to which the correct value of the subscript bound has been assigned prior to the call of the procedure.

(4) If an express formal parameter is an array the corresponding actual parameter should be an array of the same type.

(5) If an express formal parameter is a simple variable, the corresponding actual parameter must be a simple variable, array element, constant or expression of the same type. If an actual parameter is a constant or expression, then the corresponding formal parameter:

- (i) must not occur in a DIMENSION statement
- (ii) must not have a value assigned to it during the execution of the subprogram.

(6) The actual parameters need not all be distinct.

6.8 Examples of Calling Subprograms

This example is based on the example of subprograms in Section 6.6.

```
      DIMENSION K50,A(5,10),B(10,20),C(5,20)
      ............
C     THE ELLIPSIS INDICATES THE ASSIGNMENT OF
C     VALUES TO THE ELEMENTS OF K,A AND B
      I=MAX (K(1),K(2))
      DO 1 J=3,50
      I=MAX (I,K(J))
      I1 = 5
      I2 = 10
      I3 = 20
      CALL MTXMLT (A,I1,I2,B,I3,C)
      END
```

H

6.9 Statement Functions

It often happens that a programmer will find some relatively simple computation recurring through his program, making it desirable to be able to set up a function to carry out the computation. This function would be needed in only the one program, so that there would be no point in setting up a new supplied function for the purpose - which involves further work. Instead, a function can be defined for the purpose of the one program and then used whenever desired in that program. It has no effect on any other program.

A statement function is defined by writing a single statement of the form a = b, where a is the name of the function and b is an expression. The name, which is invented by the programmer, is formed according to the same rules that apply to a variable name: one to six letters or digits, the first of which must be a letter. If the name of the statement function is mentioned in a prior type statement, there is no restriction on the initial letter; if the name is not mentioned in a type statement, the initial letter distinguishes between real and integer in the usual way. The name must not be the same as that of any supplied function.

The name of the function is followed by parentheses enclosing the argument(s), which must be separated by commas (if there is more than one). The arguments in the definition must not be subscripted.

The right-hand side of the definition statement may be any expression not involving subscripted variables. It may use variables not specified as arguments and it may use other functions (except itself). All function definitions must appear before the first executable statement of the program. If the right-hand side of a statement function uses another statement function, the function definition of the latter must have appeared earlier in the program.

As an illustration, suppose that in a certain program it is frequently necessary to compute one root of the quadratic equation, $ax^2 + bx + c = 0$, given values of a, b and c. A function can be defined to carry out this computation, by writing :

```
ROOT (A,B,C) = (- B + SQRT(B**2 - 4.*A*C))/(2.*A)
```

The compiler will produce a sequence of instructions in the object program to compute the value of the function, given three values to use in the computation.

This is only the definition of the function; it does not cause computation to take place. The variable names used as arguments are only dummies; they may be the same as variable names appearing elsewhere in the program. The argument names are unimportant, except as they may distinguish between integer and real.

A statement function is used by writing its name wherever the function value is desired and substituting appropriate expressions for the arguments. "Appropriate" here means, that if a variable in the definition is real, the expression substituted for that variable must also be real, and similarly for the other types of variables. The values of these expressions will be substituted into the program segment established by the definition and the value of the function computed. The <u>actual</u> arguments may be subscripted if desired.

Examples of the use of the statement function terms defined are:

```
Z = ROOT (2.0,8.0,3.0) + Y
```

which finds a root of $2.0\ x^2 + 8\ x + 3$ and adds value Y

```
Z = ROOT (E, DM + 5.0, DM) * BETA - ATAN (C)
```

which finds the root of $(Ex^2 + (DM+5)\,x + DM)$ and multiples it by BETA, before subtracting ATAN C.

Variables in the right-hand side of the statement function definition need not all be dummy arguments. If a variable name is not a dummy argument, it has the same meaning as that name anywhere else in the program unit.

CHAPTER 7: USE OF MASIR/SIR CODING WITHIN FORTRAN TEXT

7.1 Code Sections

There are certain operations which are faster and more economical when written directly in MASIR than when written in FORTRAN and translated into machine code.

The examples used in this chapter illustrate the method of writing SIR coding as part of a FORTRAN program.

It is assumed that the reader of this chapter is familiar with the programming language MASIR.

7.1.2 Format

A code section may either be a complete subprogram or may be a part of a program unit, the remainder of which is written in FORTRAN source text. In the latter case, the machine code instructions are preceded by the directive CODE written as a FORTRAN statement on a line by itself and terminated by the directive FORTRAN written on a new line.

7.1.3 Form of Machine Code Instructions Within a FORTRAN Unit

Machine code instructions are written in a form similar to 900 series SIR coding. These instructions are written one per line. Labels should always be written on the left hand edge of the coding sheet i.e. to the left of the vertical line if using a 'free-format' coding sheet. The instructions labelled should be separated by two spaces from the label, or alternatively the label may be on a line of its own.

The function and operand are written to the right of the vertical line on a 'free-format' coding sheet. The function consists of an unsigned one or two digit number in the range 0 to 15, preceded by a / if the instruction is to be modified. The operand (address part) follows the function on the same line separated from it by one or two spaces.

A comment it introduced by left parentheses either at the beginning of a line or following an instruction. This indicates that the remainder of the current line of text is to be ignored by the compiler. It should be noted that the compiler does not check for right parentheses to terminate the comment and so comments must not extend over more than one line.

7.1.4 Labelling Instructions

Machine code instructions may be labelled, but only with identifiers of the form Qn, where n represents a number consisting of one to five digits.

Examples:

Q1 Q200 Q99999

The numeric part n is treated by the compiler in the same way as FORTRAN statement labels. It should be noted that these numbers should not be duplicated with any other label in the program unit. The number n may be used in a GOTO statement within the program unit and similarly any Fortran statement number m within the unit may appear in a machine code jump (branch) instruction as an identifier preceded by the letter Q i.e. Qm. In either case the rules of FORTRAN must be obeyed; for example a GOTO or machine code jump must not cause control to be transferred into a DO loop from outside its range (except for an extended range DO).

7.1.5 Operand

The operand (or address part) of a machine code instruction may take one of the following forms:

(i) Constant. An integer (+ or -) or octal (&) literal constant may be introduced. These are handled by the compiler as FORTRAN constants and are allocated a position in local workspace.

(ii) Variable or Array Name. The address placed in the machine code instruction depends on whether the identifier is a local variable, array, item in COMMON or a formal parameter. For an item in a local data area, writing the name as an operand causes the address of the variable to be placed in the instruction. If the name has not been previously encountered by the compiler in the current program unit, it is classed as an integer or real variable according to FORTRAN implicit type and the allocated space is local data (implicit types follow the rules state in Chapter 2 - integer variables must start with one of the letters I, J, K, L, M or N).

If the variable name quoted in the address part is a <u>local</u> array name, the address placed in the instruction is that of the first element i.e. (1), (1, 1) or (1, 1, 1) of a one, two or three dimensional array respectively. In either of these cases, the identifier may be followed by a positive offset +n for referencing multi-word items. If the identifier is a variable or array in COMMON, or a formal parameter of a subprogram, writing the name as an operand causes the address of a local data location to be placed in the instruction. This location holds the address of the variable or first array element relative to the store module of the current program unit.

(iii) Absolute addresses. The machine code function may be followed by an unsigned integer in the address part of the instruction, indicating a core store address, input/output address or a number of shifts.

7.1.6 Example

The example which follows is a subroutine containing machine code instructions.

```
[ THIS SUBROUTINE SHOWS EXAMPLES OF MACHINE CODE SECTIONS

SUBROUTINE SUB(IP)
DIMENSION IA(100)
COMMON K, J(100)
CODE
   4     J          (ADDRESS OF J)
   5     IWS
FORTRAN
DO 9 N=1, IP
IF (N-120) 1, 2, 2
1 CODE
   0     IP
  /4     0          (VALUE OF IP)
   0     K
  /2     0          (NEGATE AND ADD VALUE OF K)
   9     Q2
   0     N
  /4     IA         (GET IA  [N+1] )
   0     IWS
  /5     0          (STORE IN J)
Q2
   10    IWS
   4     IA+20      (IA  [21]  )
   1     -6
   14    3
   6     &077770
   7     Q9
   5     M
FORTRAN
WRITE (3, 8) M
8   FORMAT (15)
9   CONTINUE
RETURN
END
```

7.1.7 Return to FORTRAN Text

After a group of SIR machine code instructions a return to the FORTRAN source program will be necessary, this can be achieved by using the directive FORTRAN written on a new line.

7.1.8 Constraint on Symbolic Names

When machine code instructions are included in a program unit, possible confusion is brought about when using variable names composed of the letter Q followed by 1 to 5 digits. Such names cannot be referenced within a machine code section as they would be treated as label references and so should be avoided in the FORTRAN text.

7.2 Program units in Machine Code

The facility for in-line machine code will cover the majority of requirements not catered for by the FORTRAN language, with the advantage that the standard subroutine linking code is automatically inserted by the computer. The loader, however, also allows independently compiled MASIR blocks to be incorporated into an object program. The following points summarise the rules for calling MASIR program units from within FORTRAN texts.

1) On entry to a block of SIR code, a correct module-relative link has been planted in the first word of the block and a jump made to the second word. At this time, the accumulator contains in bits 17-14 the module number of the call minus the module number of the SIR block. Following the call are the addresses of any operands, relative to the module of the call. A parameter address word containing a direct address has bit 18=1; one level of indirection is provided by setting bit 18=0.

2) The macro CALLG (name) should be used to call any further subroutines, and parameter addresses set up as previously defined. In principle the Main program of an executable system can be in MASIR, calling FORTRAN subprograms by CALLG.

3) It is not possible to access FORTRAN COMMON storage, except by passing addresses of items in COMMON as parameters to the SIR block.

4) Return should be made to the location following the last parameter of the call.

These rules are now expanded in greater detail.

Within each store module (block of 8192 words) into which a program is loaded, the FORTRAN/MASIR loader places a set of instructions known as module code; these provide a means of transferring between subroutines in different modules. When the FORTRAN compiler generates a call of a SUBROUTINE or FUNCTION, it generates a special macro which is processed by the Loader. The Macro Assembler MASIR generates the same macro when the source code macro CALLG is used. CALLG is written in the form:

```
CALLG(SUB)
```

where SUB is the name of the subroutine to be entered.

The loader macro, previously mentioned,always generates three words of code. If the subroutine in question is loaded into the same module as the calling routine, the loader generates a direct subroutine call, equivalent to the assembly code sequence:

```
 4  +0
11  SUB
 8  SUB+1
```

If the subroutine is loaded into a different store module, the loader generates, for each call, 3 words equivalent to the assembly code sequence:

```
 4  +SUB
11  QMC (Call SUB via Module Code)
 8  QMC+11
```

where +SUB represents the address of a location holding the address of SUB relative to the calling module.

The module code QMC has the form:

```
          Word
          0;                  QMC  >1
1;        to       10;        (Reserved for FORTRAN, etc. use)
          11;                 5 W (Store Relative Address)
          12;                 0 W
          13;                 6  &760000
          14;                 2 QMC
          15;                 /5 0 (Store adjusted link)
          16;                 6  &760000
          17;                 /8 1 (Jump to subroutine entry)
```

This code is automatically duplicated in each module in which code is stored.

The called subroutine may be written in Assembly code or FORTRAN. If SUB is written in Assembly code it should have the usual form:

```
SUB   >1   (Link)
   .       (Entry point following link)
   .
   .
   .
(Body of Subroutine)
   .
   .
 0 SUB     (Exit)
/8 1
```

If the call of the subroutine is from FORTRAN, this example is equivalent to a SUBROUTINE with no explicit formal parameters.

If the Subroutine has two explicit formal parameters, e.g. SUB2 (I, J), the 3 word calling macro will be followed by two addresses referencing the actual parameters. Exit from the Assembly code subroutine would be to the third location after the call (e.g. by /8 3 jump).

For each parameter address, if Bit 18 = 1 (i.e. the word is negative) Bits 17 to 1 hold the direct address of the parameter, relative to the calling module. If the word is positive, (Bit 18 = 0), then Bits 17 to 1 contain an indirect address. This address points to a location of store holding the actual address of the parameter, relative to the calling module.

Example:

SUB2 starts at 7000↑0 (location 7000 of store zone 0), and is called from a program in zone (Module) 1 say at 500↑1. There are two parameters to the call, the first direct, an array starting at location 800↑1, the second indirect, an integer at 2000↑2 (i.e. 18384). The loader has allocated QMC to location 8000↑1 in store zone 1. The call might take the form:

```
 4   600
11   8000
 8   8011
/0   800   (Direct address, relative to Zone 1)
 0   700   (Indirect address)
```

where 600↑1 will hold -1192 (= 7000-8192) the relative address of SUB2. And 700↑1 will hold +10192 (= 8192+2000) the relative address of the second parameter.

In MASIR assembly code this may be written:

```
CALLG(SUB2)
/0   ARRAY
 0   ADRB
```

where ADRB is a local data location holding the address of the second parameter. If the second parameter is fixed one could write in ADRB:

```
ADRB  +X
```

where X is the (global) name of the actual parameter, but this could be simplified further by omitting ADRB and writing:

```
CALLG(SUB2)

/0  ARRAY

 0  +X
```

When writing in Assembly code any parameters referenced by direct address must be in the same module as the call. Any non-local parameters which are, or might be, in another module must be referenced indirectly. This is because the +LABLL facility of the assembler may generate a negative or a positive address, depending on the relative position of the label.

Therefore, it is not permissible to use the form +LABEL on its own, (i.e. not preceded by a function number) in the words following the CALLG. 0 +LABEL is permissible because its generates an indirect ("literal") address, and the parameter word itself will always be positive.

If written in Assembly Code, the subroutine SUB2 might take the form:

```
    [SUB2]
    SUB2      >1
         5    ADJA  (Address adjustment)
         0    SUB2
        /4    1
         9    STPA1 (Direct address)
         1    ADJA
         5    W
         0    W
        /4    0
  STPA1  6    &377777
         1    ADJA
         5    PA1    (Store address of first parameter)
         0    SUB2
        /4    2
         9    STPA2
         1    ADJA
         5    W
        0      W
       /4      0
STPA2  6       &377777
       1       ADJA
       5       W            (Store address of second parameter)
       0       W
       /4      0            (Pick up value of second parameter)
```

(If the program is to be run on 905 or 920C only, the 5 W, 0 W sequences may be replaced by ATB).

Exit from the subroutine would normally be in the form:

```
 0  SUB2
/8  3
```

The reader may find it helpful to work through the given examples, using numeric examples of addresses, to confirm that the parameters will be accessed correctly, and that return will be made correctly to the calling program.

A FORTRAN function call will be compiled in the same way as a subroutine call. If it is an integer function, the result will be held in the machine A-register on exit from the FUNCTION. If a real, double precision or complex FUNCTION the result will be in the appropriate software pseudo-accumulator of QFP.

If the subroutine written in assembly code will never be called directly from FORTRAN, it is of course possible to simplify the subroutine body, for example by only allowing direct address parameters, or by a completely different method of parameter passing. The use of the CALLG macro does not dictate any particular method of parameter passing, it merely supplies an address adjustment factor in the A-register on entry to a subroutine.

CHAPTER 8: WRITING FORTRAN PROGRAM

8.1 Program Writing

The format for Standard FORTRAN programs is based on the use of punched cards. Since the majority of 905 FORTRAN users input programs via the medium, paper tape, an alternative format called free-format is provided. The Standard FORTRAN format for program input is referred to as fixed format.

Column numbers in fixed format input are determined by counting the number of printing positions from the left hand margin (i.e. the number of significant printing characters since the last new line (linefeed) character, including space but excluding null (blank paper tape), carriage return and erase.

8.2 Fixed Format

When writing programs in fixed format, a FORTRAN coding sheet should be used, with individual character positions marked on each line (squared graph paper may be used as an acceptable alternative). The first six columns are reserved for special use, and columns 7 to 72 usually contain statements (spaces are not significant in this area except where specifically stated e.g. Hollerith strings).

The significance of the various lines are as follows:

a) COMMENT lines

A comment line must commence with the letter C written in the first column; the remainder of the line contains text inserted by the programmer. They are used to improve visual interpretation of the text to a programmer or user who wishes to understand or modify the program, or for the purposes of the original programmer who returns to modify the program after considerable absence from the program(Comment lines are ignored by the compiler, but must not occur between a line and its continuation line, or between two continuation lines).

b) Initial lines

The initial line is the first line of a statement (frequently it will be the only line of a statement. It is distinguished by leaving column 6 blank or zero. i.e. the sixth significant character in either space or digit 0). Columns 1 to 5 will either be blank or contain a statement number.

c) Continuation lines

A continuation line is used to extend a statement which requires more characters that may be punched on a single line. It must follow an initial line or another continuation line (Comment may not be used in the middle of a statement).

A continuation line is written with a character other than space (blank) or zero (0) in column six. In practice, it is usual to use the digits 1 to 9 to number the continuation lines after an initial line. There may be up to 19 continuation lines to a single statement in a Standard FORTRAN program, but 905 FORTRAN will not detect the limit.

There is also a limit on the complexity of a statement, the complexity being expressed by the number of nested expressions and function calls.

NOTE: It is recommended that columns 1 to 5 of a continuation line are left blank.

d) END line

An END line is the line which terminates a program unit. It should be written with spaces (blanks) in columns 1to 6, and the letters E, N, D in columns 7, 8, 9 respectively. The END line is not an executable statement and the statement preceding it must be a GOTO, STOP or similar statement. If the program execution apparently leads to an END line, the effect is undefined.

8.3 Free Format

The compiler discriminates between free and fixed format input as follows. Fixed Format is assumed initially, but if the first character or the first line of a program unit is a character [(apart from new line) this introduces a comment line; the program unit is read as free format. Every free format program unit must start with a comment.

NOTE: It is not sufficient for only the first of a group of units to start with a comment.

When either writing or punching programs, the following rules must be observed:

a) Programs are written on lined paper with a vertical line approximately $1\frac{1}{2}$ inches from the left hand margin.

b) Each FORTRAN statement starts on a new line and the statement proper is written to the right of the vertical line (columns 7-72 incl.).

c) Statement numbers are written to the left of the vertical line (columns 1-5 incl.).

d) Continuation lines are to be indicated by a currency symbol (£) to the left of the vertical line (Continuation lines are used where the statement is too long for one line of text).

e) Comment lines in free format are to be indicated by the symbol [written to the left of the vertical line (In fixed format, the letter C is used for this purpose, again to the left of the vertical line). A comment line is ignored by the compiler.

f) When punching, any code to the left of the vertical line is punched first; two spaces follow and finally the statements.

8.4 Punching Instructions

An example of program punched from coding is given in Section 8.6. Punching rules are as follows:

a) The program can be punched on any type of tape punching equipment operating in 900 series, ISO, British Standard or ASCII code. Whatever equipment is used, the punched tape produced should be verified (using a verifier punch) by a second operator, or should be printed out on the teleprinter. The print-out produced should be checked against the original program coding to ensure that no punching errors have occurred.

NOTE: On some type of punching equipment newline is punched as a single character, whilst on other types a combination of carriage return and line feed characters is used. On this latter type of equipment, N consecutive new lines should be punched as:

carriage-return, N line-feeds, blanks

b) A program can be written on a pre-printed FORTRAN coding Sheet or on lined paper as specified in Section 8.3.

c) Always punch the full written program (i.e. include all blank lines, spaces etc.) to ensure a correct print-out.

d) For Free Format text, always ensure that two spaces are left between code to the left of the vertical line and the rest of the information carried on that line of coding.

e) Exercise care to avoid confusion between the following sets of characters:

Figure 0 and the Letter O

Figure 1 and the Letter I

Figure 2 and the Letter Z

Figure 5 and the Letter S

These characters must be punched correctly and punch operators must familiarise themselves with the various punching conventions used by the various programmers in their coding.

NOTE: There is no universally accepted convention, even for distinguishing between letter O and figure 0, although it is common practice to slash a zero (Ø).

f) Always run-out about 6"(15 cm) of blank tape at the beginning of every tape punched.

g) If an incorrect character is punched, this may be rectified by backspacing and overpunching with an 'erase' character. The 'erase' character does not count towards the maximum number of characters that can be punched on a line (see b).

h) A line of text must not include more than 80 characters (blank and erase do not count towards this total).

8.5 Names Starting with Q

If the first character of an identifier starts with the letter Q, the second character must be the letter U.

8.6 Example of Written Program in Free -Format

```
[       MATRIX MULTIPLICATION
        SUBROUTINE MXMULT (A,B,C,I,J,k)
        DIMENSION A(I,k),B(I,J),C(J,k)
        DOUBLE PRECISION AA
[       A= B*C
        DO 1 II = 1,I
        DO 1 kk = 1,k
        AA = 0
        DO 2 JJ = 1,J
2       AA = AA+B (II,JJ) * C (JJ,kk)
        IF (AA-1D19)1,1,4
1       A(II,kk) = AA
        RETURN
4       WRITE (3,9) AA
        GOTO 1
9       FORMAT (22HINNER PRODUCT TOO BIG = ,
$       D20.10)
        END
```

This program would then be punched thus:

```
[   MATRIX MULTIPLICATION
SUBROUTINE MXMULT (A,B,C,I,J,k)
DIMENSION A(I,k),B(I,J),C(J,k)
DOUBLE PRECISION AA
[    A=B*C
DO 1 II=1,I
DO 1 kk=1,k
AA=O
DO 1 II=1,I
DO 1 kk=1,k
AA = O
DO 2 JJ=1,J
2      AA=AA+B(II,JJ) *C (JJ,kk)
       IF (AA 1D19)1,1,4
1      A(II,kk)=AA
RETURN
4      WRITE (3,9)AA
GOTO 1
```

```
 9      FORMAT  (22HINNER PRODUCT TOO BIG=,
 $      D20.10)
        END
```

An alternative layout for the coding of the example in fixed format would be:

```
C     MATRIX MULTIPLICATION
      SUBROUTINE MXMULT (A,B,C,I,J,K)
      DIMENSION A(I,K),B(I,K),C(J,K)
      DOUBLE PRECISION AA
C     A=B*C
      DO 1 II=1,I
      DO 1 KK=1,K
      AA=O
      DO 2 JJ=1,J
2     AA=AA+B(II,JJ)*C (JJ,KK)
      IF (AA-1D19) 1,1,4
1     A (II,KK) = AA
      RETURN
4     WRITE (3,9)AA
      GOTO 1
      FORMAT  (22HINNER PRODUCT TOO BIG=
      1 D20.10
      END
```

8.7 Correction of FORTRAN Programs

Corrections to a FORTRAN program must be made to the original FORTRAN text. Individual units changed should be re-compiled.

CHAPTER 9 COMPILER OPERATION

The compiler is designed to process independent program units, which it converts into relocatable binary form suitable for presentation to the linking loader. Once compiled, a program can be incorporated in any number of object programs.

A secondary output identifies the program unit and specifies to the programmer any errors detected during compilation (see Chapter 10). It also optionally supplies a store map and a list of external identifiers referenced by the program.

9.1 Options

Options are expressed as an octal number formed from the sum of the individual options required by a program; if all options are omitted, a standard option 00 is assumed. Values of other options are:

01	Syntax check only
02	Data map required
04	Data map output to punch
10	Pack integer arrays

Hence, a program with standard option 00 calls for normal compilation with standard integer arrays (i.e. not packed). A program with option value octal 13 calls for syntax check (octal 01), data map requirement (octal 02), with packed integer arrays (octal 10).

9.2 Secondary Output

The heading FORTO2 is output when compilation of a program unit commences (the serial number identifying the version of compiler in use, which can vary). Error messages,if any,are next output and are followed by the data map (if required) and finally the terminating message. This output takes the form:

UNIT xxxxxx ;   SIZE=nnnn

where:

xxxxxx is the unit name, and

nnnn is the number of words occupied by code, constants and local variables and arrays but NOT variables or arrays in COMMON.

When an error occurs, a value for the unit size is output which includes code, constants etc. This value does not include the error statements and the value should therefore be treated with caution.

9.3 Error Reports

The general form of an error output indication is:

ttt nnnn 1111

where:

ttt is the error type code (see Chapter 10)

nnnn is the last statement number encountered, and

1111 is the count of non blank lines since this statement number.

If an error is encountered before the end of a statement, the part of the statement already processed will be displayed on the next line.

EQUIVALENCE statements are not processed by the compiler until the end of the specification section of a program unit. If an error is detected during this processing, the statement number and count of lines only indicate the first statement following in the unit which is not a specification statement. A further number (cc) is output which shows the position reached within the total EQUIVALENCE information (regarded as a single continuous line with blanks and the word EQUIVALENCE omitted).

If the line (indicated by 1111) is a comment line, the error recorded refers to the preceding statement. For all except warning errors (identified by the initial letter W), the output of RLB is terminated.

With the exception of error ZZ (compiler workspace full), the remainder of the program unit is scanned and checked for correct syntax. Hence, any error messages output from this point (i.e. store full) although they may be useful to the programmer, must be regarded with caution.

9.4 Data Map

If the data map option is set, upon detection of the END line statement the following information is output:

DATA MAP : xxxxxx

yyyyyy aaaa b

yyyyyy aaaa 1 bbbbbb

where:

xxxxxx is the unit name; there is an entry in the map list for each variable or array referred to in the program unit.

yyyyyy refers to the symbolic name of an item

aaaa refers to the relative address of that item, and

t refers to that address type. The address type can be:

0 Undefined

1 COMMON

3 Local data

4 Indirect address in local data

In the instance of COMMON, the block name bbbbbb is also given.

All addresses are relative to the start of the program unit or COMMON block. Absolute addresses can be derived from a global list after loading of the program.

The 'undefined' type (value 0) includes parameters of statement functions, but can also indicate miss-spelt names.

After the variables and arrays are printed the label list is of the form:

LABELS

nnnnnn aaaa

where:

nnnnnn is the statement number

aaaa is the relative address of that statement (relative to the start of the program unit)

NOTE: If the address is omitted, this means that the label has been referred to but not defined.

The last section of output lists the external references in the form:

EXTERNAL

pppppp

bbbbbb ssss

All external procedure names pppppp or COMMON block names bbbbbb referenced within the program unit are included, the latter indicating the block size by ssss. Blank COMMON is displayed as a blank with a size.

CHAPTER 10: ERROR MESSAGES

Included in this chapter are the error messages which are output at Compile-time, loading time, object time and the Control error reports.

10.1 Compile-time Errors

The following table contains error messages which are output during compilation.

Compile-time errors

Error Code	Description	Comment or Example
ARD	Array declarator error	Dimension not constant, or formal variable if formal array.
ASS	ASSIGN statement syntax error	'TO' missing or miss-spelt
CHx	Character x not found	Particular character expected on the basis of syntax so far.
CMN	Name usage in COMMON	Item is formal parameter, or has already appeared in a COMMON statement.
CN1	Illegal constant formation	Exponent overflow in real constant; integer constant overflow; exponent digits missing; zero Hollerith count; possible error in .EQ. operator leading to confusion with real constant.
CN2	Invalid complex constant	Imaginary part not a correctly formed real constant.
DA1	DATA statement syntax error	List not ending with comma or oblique.
DA2	DATA statement list error	Number of items in value list not equal to number of data elements; element declared in COMMON.
DO1	DO statement error	Statement number already defined; DO misspelt; DO in logical IF.
DO2	DO loops not nested	

Error Code	Description	Comment or Example
EQS	Illegal subscript within EQUIVALENCE	Subscript is not constant, or number of subscripts does not match number of dimensions.
EX1	Replacement operator usage	e.g. more than one "=" in assignment.
EX2	Unmatched parenthesis within parameter expression	
EX3	Relational usage	Complex item in relational expression, or two relational operators.
EX4	Illegal operator or operand/ operator combination	e.g: logical operand with arithmetic operator.
EX5	Array or function usage	Array or function name not followed by subscript/ argument, and not as single argument; statement function quoted as an argument.
EX6	Incorrect unary usage	Successive unary operators, etc.
EX7	Illegal type association	Logical operand in arithmetic expression or vice versa; complex or logical within relational; integer or DP in complex.
EX8	Too many open parentheses	
EX9	Too many closed parentheses	
FOR	FORMAT statement unnumbered	

Error Code	Description	Comment or Example
FUN	FUNCTION without arguments	
GTO	GOTO statement syntax error	
HOL	Hollerith constant error	Incorrect character count, or Line Feed within Hollerith string.
LIF	Logical IF error	Logical IF within logical IF
NU1	Name missing	Variable, array, or procedure name expected from context but not found.
NU2	Variable missing	Variable or array name (not formal parameter) expected from context but not found.
NU3	Invalid procedure name	Name quoted as procedure name previously defined as array or used as variable.
NU4	Integer variable expected, not found.	
NU5	Integer variable or constant expected, not found.	
NU6	Array name error	Name preceding array declarator already defined as array or procedure.

Error Code	Description	Comment or Example
PBN	Procedure name = blockname	Not permitted by the loader; this check detects some but not all occurrences.
RET	RETURN statement in main program	
RW1	READ/WRITE format reference error	Format reference is not statement number or array name
RW2	Input/Output list name error	Name is not variable or array name
RW3	Input/Output list sýntax error	Implied DO without opening parenthesis, or null implied DO.
SBX	Subscript expression syntax error.	
SN1	Invalid statement number definition	Non-numeric character within statement number definition
SN2	Invalid statement number reference	Statement number reference exceeds five digits.
SSQ	Statement sequence error	No statement number in a DO statement.
STF	Statement function name error	Already defined as array or external procedure name, or used as variable.
STM	Improper termination of statement	

Error Code	Description	Comment or Example
STY	Statement type error	Not correctly formed assignment, DO, or statement function; first 4 letters of keyword do not match; keyword wrong length.
TYS	Type statement syntax error	
	Warning Errors	
WD1	Unterminated DO loop	
WD2	Illegal DO termination	Termination statement is not of permitted type
WEX	Exponent underflow in real constant	Maximum value of exponent is 19 (approximately); constant is set to zero.
WF1	Improper zero in FORMAT	Format is stored as written
WF2	Parentheses nested too deep	Format is stored as written
WF3.	Improper scale factor	Syntax indicates scale factor, but "P" missing. Format stored as written
WF4	Scale factor not followed by conversion format	Format stored as written
WF5	Decimal point missing from conversion format	Format stored as written

Error Code	Description	Comment or Example
WF6	No digit following decimal point	Formal stored as written
WHC	Hollerith constant count error.	Maximum 12 characters in Hollerith constant quoted as argument or in DATA statement. First 12 are taken.
WN1	Doubly-defined statement number	New definition is used in any subsequent references
WN2	Statement number usage error	FORMAT - associated label used in normal reference, or normal label in input/output reference. (First appearance defined usage).
WN3	No path	Unlabelled statement following GOTO. RETURN, IF.
WN4	Numbered END line	
WQ1	Formal parameter of multiple COMMON in EQUIVALENCE	Item is ignored.
WQ2	COMMON base extended back by EQUIVALENCE.	Item ignored.
WQ3	Special EQUIVALENCE rules contravened	
WS2	Logical constant spelling	

Error Code	Description	Comment or Example
	Warning Errors	
WX2	Procedure call disagreement	Number of parameters quoted in two calls of same procedure do not match
	Machine Code Errors	
X1	Illegal first character	First character is letter, not F or Q
X2	Function code exceeds 31	
X3	Invalid operand	Not constant, variable, or array
X4	Invalid character	
X5	Field too long	Function exceeds 2 digits, label exceeds 5 digits, label exceeds 5 digits.
YLO	Invalid logical operator	Presumed operator starts with . but no correct form follows.
YXM	Exponentiation mode error	
YY	Free-format line conversion error.	First character not one of those permitted, or line too long.
ZZ	Compiler workspace full	Workspace is used for accumulating dictionary entries, and also for transient purposes such as expression

10.2 Loader Error Messages

See MASIR manual.

10.3.1 Error reports from Mathematical Functions

In all these reports the first address is the call address and the second (if present) the operand address.

Error Code	Description	Routine which may notify this error
ECI	Overflow on conversion to integer	QFP (arithmetic package), IFIX, INT, IDINT.
EDI	Integer divident = -131072	QID (integer division routine), IABS.
EDZ	Attempted division by zero	QFP, QID, AMOD, MOD, DMOD.
EML	Logarithm of negative or zero argument requested	ALOG, DLOG, ALOG10, DLOG10, CLOG.
EMM	Attempted exponentiation of negative real argument by negative real exponent.	QER, QED. (exponentiation routines)
EOI	Integer overflow as result of operation	QFP, QEI (integer exponentiation), MAXI, MINI, ISIGN, IDIM.
EOO	Exponent overflow as result of operation.	QFP, QEC, EXP, DEXP, CEXP, DIM, SNGL, CSIN, CCOS, DIM.
ESN	Square root of negative number requested	SQRT, DSQRT
EZZ	Attempted exponentiation of zero by zero	QEI, QER, QED, QEC

On continuation, a zero result is returned except in the case of EDZ from MOD, AMOD, or DMOD, when the first argument is returned.

10.3.2 Input/Output Error Reports.

Routine name is always QIO (input/output package), the first address is the call address, and the second (if present) the operand address. Final 6 characters are the beginning of the FORMAT.

Error Code	Description	Comment
EOI	FREE format specified with WRITE operation, or logical item in list for free-format input.	
EO2	Type disagreement between format and list.	
EO3	Initial character of format is not left parenthesis.	(Any leading blanks are ignored)
EO4	Illegal character in format.	
EO5	Format syntax error	
EO6	Unmatched parenthesis in format or parenthesis level greater than 2	
EO7	Improper operation for device type.	e. g: REWARD paper tape. (Error in unit number likely).
EO8	Improper QIO call	Probably indicates serious error in program, e. g: overwriting.
WOI	Illegal character in data	Restart causes continuation ignoring illegal character.

Error Code	Description	Comment
WO2	Integer out of range	Restart causes continuation with truncated number of exponent.
WO3	Exponent out of range	

10.3.3 Control Error Reports

The table which follows contains control error reports.

Routine Name	Error Code	Description	Comment
QFP	EIF	Invalid parameter code	) Should only arise) when QFP is called) from SIR segment or) in-line code
QFP	EMS	Invalid mode setting	
QCG	ERR	Index out of range in computed GOTO	First address is call and second is index variable. Restart causes continuation as though index variable had value 1.

CHAPTER 11: OPERATING INSTRUCTIONS

The following sections contain the operating instructions (for operating within FAS or RADOS operating systemssee the appropriate Operating System description) for a paper tape environment:

11.1 Compilation

1) Input the tape "905 FORTRAN COMPILER" by initial instructions (Entry at location 8181).

2) Enter at location 16. Symbol ← will be output on the on-line teleprinter.

3) Specify the required option, by typing letter O followed by one or two digits, then newline.

The digits specify an octal number, the sum of the values required are made up from:

01	Syntax check only, no code output.
02	Data Map required.
04	Output data map to punch.
10	Compressed integer array storage allocation.

If option zero (O0) is used, this implies normal compilation, no data map, and two words allocated to each element of integer arrays (for compatability with other FORTRAN compilers layout of COMMON areas). Option 3 requests a syntax check, with output of data map to the on-line teleprinter.

4) Load the FORTRAN source program tape in the reader and type M or R, to compile the program. Program units will be read and processed until a halt code is read (or 'dictionary full' error occurs).

5) To process further programs, repeat from step 3).

6) Wind up each relocatable binary output tape as follows:

   When punch output is completed tear off the tape and label the underside of the end nearest the tear. Wind the tape to ensure that the labelled end will be read first by the reader.

7) The units compiled are now ready to load and run. Program units may be compiled in various ways i.e:

   Together

   In a batch

   Singly

   As a group of sub-programs at a time.

Compilation of each unit (main, function or subroutine) is independant of other units in a program.

11.2 900 Loader Operating Instructions for FORTRAN use.

The '900 Loader' tape distributed with MASIR is also used to load 905 FORTRAN relocatable binary tapes generated by the compiler (sum-checked binary tape). Proceed as follows:

1) Load the '900 LOADER' by initial instructions (entry at location 8181).

2) Enter at location 16; symbol ← is displayed on the teleprinter.

3) Type options into the loader in the form:

O followed by the option value in octal (see section 11.3).

4) Load the first relocatable binary tape in the reader and type L to enter the Loader. This tape is read until the tape reader unloads.

5) Load subsequent tapes in the reader and press the READ button.

6) Press RESET, enter at location 16 and type option 03L to load the first library tape called '905 FORTRAN LIBRARY VOL. 1'. However, if output is required to paper tape, the option 017L should be used.

7) Load the tape '905 FORTRAN LIBRARY VOL. 2' and press the READ button.

8) When all programs are loaded, press RESET and enter at location 16.

9) Type M.

If there are any unlocated labels, these are printed on the teleprinter (i.e. global labels, program names or data labels referenced from programs loaded but not included on any tape actually loaded).

If there are no unlocated labels, GO is output.

If loading was direct into store, the program may be started by re-typing M. If the loading process produced a binary tape, this should now be complete; type M, runout the tape and tear off from punch.

If there were unlocated labels, either return to step 3) or 6) to load further tapes, or type M to run the program disregarding the missing labels (0V will be output to indicate override, in the latter case). If output was to paper tape, it will be completed. If loading was direct to core store, the program may be started by typing M again (i.e. for the third time).

If there is a MAIN program, the executable program will be entered at this point, irrespective of the order of loading tapes. The MAIN program may be either a FORTRAN main unit, or a MASIR unit with global label MAIN.

If there is no MAIN program, the executable program will be entered at the first location of the first tape to be loaded. (It is possible for this unit to be a FORTRAN subroutine with no parameters, in which case the RETURN statement of that subroutine must never be obeyed).

If a sum checked binary tape was produced, this will be loaded by initial instructions. When loaded, the program may be entered by jumping to location 16 and typing M.

If an option is typed before typing M to enter the program, this will be held in the A Register on entry. This option may consist of up to 15 digits (i.e. 5 octal digits) long.

10) Type C if continuation required after a halt code on inputting data, PAUSE, or run time error message. R or C may be typed whenever the symbol ← is displayed.

11.3 Loader Option Bits

The loader option must be either a one or two octal digits, the sum of the digit values implied follow:

Bit 1	=	0	if loader to be initialised (first program tape).
	=	1	if loader not to be initialised (subsequent tapes)
Bit 2	=	0	if everything read is to be loaded
	=	1	if library scan (only load program units which have been referenced but have not been located). Ignore units which have already been loaded.
Bit 3	=	0	if loader is to store program in core
	=	1	if loader is to output program on paper tape or backing store.
Bit 4	=	0	if loader is to store program on backing store.
	=	1	if loader is to output program on paper tape.
			(Bit 4 is ignored if bit 3 = 0)
Bit 5	=	0	if the program to be loaded is to use the built-in routines
	=	1	if program to be loaded does not use the built-in routines.

Bit 6	=	0	ignore
	=	1	freeze current dictionary and store layout. If option with bit 5 = 0 and bit 1 = 0 is now typed, the loader will be reset but the programs already loaded will not be lost. They will be preserved in store for use by future programs, unless overwritten at runtime.
Bit 7	=	0	ignore
	=	1	list labels
Bit 8	=	0	print first/last messages
	=	1	suppress first/last messages
Bit 9	=	0	halt after warnings *CLW, *COM
	=	1	continue after warnings

11.4 Store Layout

11.4.1 Loading

The loader occupies locations 128 to 2800 (approx.)

The program entry instructions occupy location 15 to 19

The dictionary, formed by the loader, occupies store from location 2800 upwards, with five locations more for each global label.

Program is stored downwards in each module between the free store limits set by LODSET. A store full indication is given if the program cannot be stored without overwriting the loader or its dictionary. Blank COMMON overwrites the loader from 128 upwards to the higher addressed store.

If a program cannot be inserted into a given store module it is loaded into the next module down (Down indicates towards the lower addressed end of core store), if sufficient space is available.

If loading via paper tape reader, the program is not actually stored, so that the store used by the 'Loader' and its dictionary may be filled with program, down to the top of the blank COMMON block, or location 128 if there is no COMMON.

11.4.2 Different Store Sizes

If a core store of more than 16K is used, LODSET is used to set the actual store limit LODSET is built into the '900 Loader' and its use is described in the MASIR operating instructions.

905 FORTRAN programs require 16K of store for compilation since the computer and its dictionaries etc. occupy at least 10K of store. However, it is possible to load and run programs on an 8K 900 series computer, if the loader is set for 8K operation by LODSET.

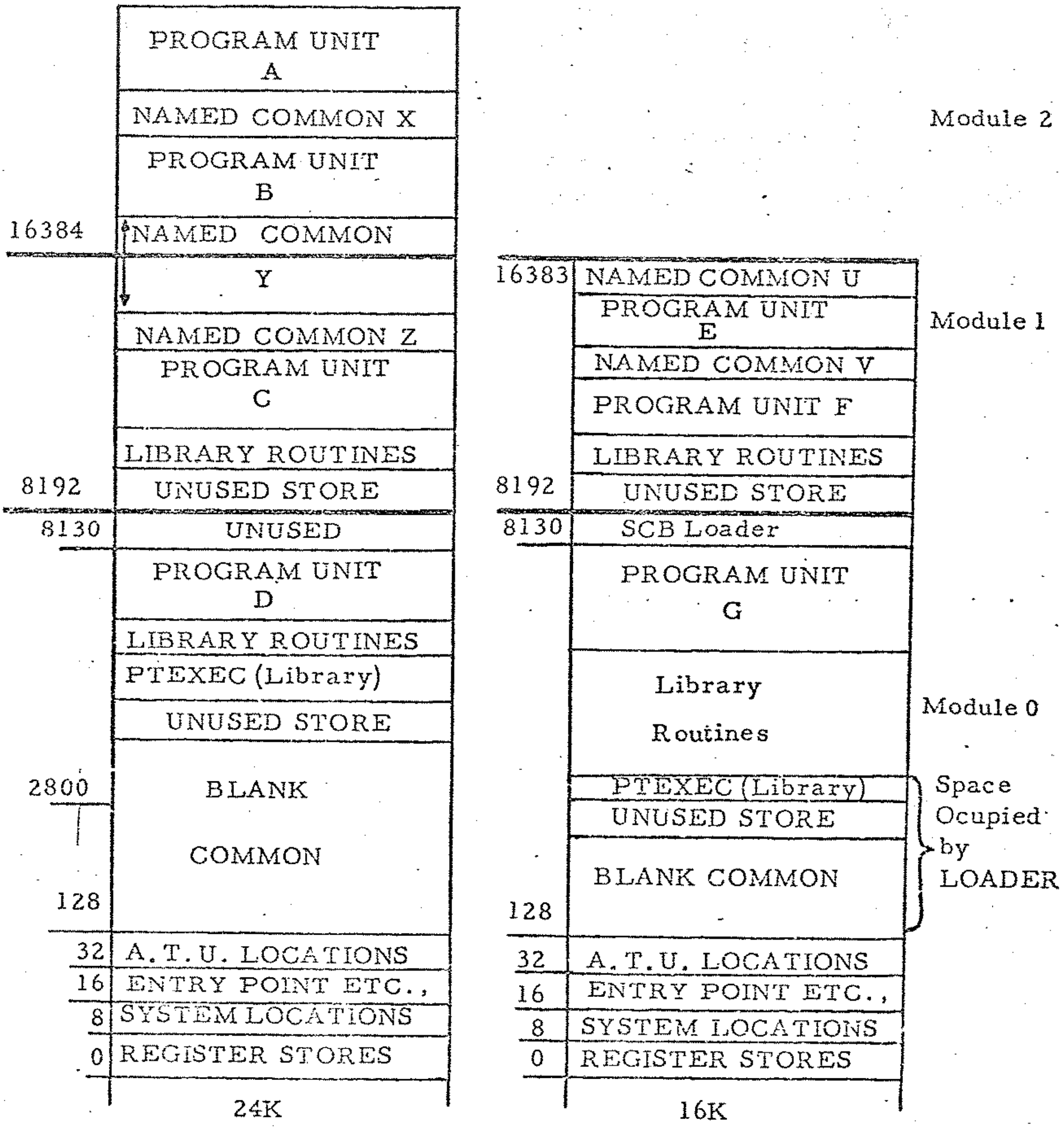
11.4.3 Library

The '905 FORTRAN LIBRARY VOL. 1' contain all the Intrinsic and Basic External Functions and special FORTRAN object time routines.

'905 FORTRAN LIBRARY VOL. 2' contains the floating point, double length and complex arithmetic routines (QFP), the READ/WRITE routines (QIO) and PTEXEC. PTEXEC contains character input/output, error routines and program control (see EX900).

11.5 Store Map at Run-time

Store maps for typical program in either 16K or 24K store follow:



The 16K version on the right can only be loaded by allowing the loader to generate sum checked binary paper tape.

These maps only apply to paper tape environment (but not to either the FAS or RADOS environment).

The total library routines should occupy about 9K of core store, but it is highly unlikely that any one program would use all the routines. Only those routines required are actually loaded with a few exceptions, e.g. if SIN is required DSIN will also be loaded, since these are to be found in the same program and use the same main code section.

The majority of programs will require routines QFP, QIO and PTEXEC, which together occupy approximately 4000 words. Integer only programs will omit the routine QFP (approx. 700 words) and programs without READ or WRITE statements will omit routine QIO (input/output package - approx. 2800 words). Input and/or output could be entirely in Assembly code. Any FORTRAN program using any STOP, PAUSE, READ or WRITE statements will use PTEXEC (approx. 500 words).

APPENDIX 1: BASIC SUPPLIED FUNCTIONS

Tables A.1.1 and A.1.2 lists both basic external functions and intrinsic functions supplied with all 905 Compilers. The principal difference between Basic External functions and Intrinsic functions is that the former may be used in EXTERNAL statements, and be replaced by user's own versions. However, the Intrinsic function cannot be replaced by user's own version.

TABLE A.1.1 BASIC EXTERNAL FUNCTIONS

EXTERNAL FUNCTION	DEFINITION	NUMBER OF ARGUMENTS	SYMBOLIC NAME	TYPE OF	
				ARGUMENT	FUNCTION
EXPONENTIAL	e^a	1	EXP	Real	Real
		1	DEXP	Double	Double
		1	CEXP	Complex	Complex
NATURAL LOGARITHM	$\log_e(a)$	1	ALOG	Real	Real
		1	DLOG	Double	Double
		1	CLOG	Complex	Complex
COMMON LOGARITHM	$\log_{10}(a)$	1	ALOG10	Real	Real
			DLOG10	Double	Double
TRIGONOMETRIC SINE	sin(a) (a in radians)	1	SIN	Real	Real
		1	DSIN	Double	Double
		1	CSIN	Complex	Complex
TRIGONOMETRIC COSINE	cos(a) (a in radians)	1	COS	Real	Real
		1	DCOS	Double	Double
		1	CCOS	Complex	Complex
HYPERBOLIC TANGENT	tanh(a)	1	TANH	Real	Real
SQUARE ROOT	$\sqrt{a}$	1	SQRT	Real	Real
		1	DSQRT	Double	Double
		1	CSRT	Complex	Complex

TABLE A. 1. 1 BASIC EXTERNAL FUNCTIONS (Cont'd)

EXTERNAL FUNCTION	DEFINITION	NUMBER OF ARGUMENTS	SYMBOLIC NAME	TYPE OF	
				ARGUMENT	FUNCTION
ARCTANGENT (Result is the principal value, in radians)	acrtan (a)	1	ATAN	Real	Real
		1	DATAN	Double	Double
	arctan (a_1/a_2)	2	ATAN2	Real	Real
		2	DATAN2	Double	Double
REMAINDERING	a_1(mod a_2)	2	DMOD	Double	Double
MODULUS		1	CABS	Complex	Complex

TABLE A.1.2 INTRINSIC FUNCTIONS

EXTERNAL FUNCTION	DEFINITION	NUMBER OF ARGUMENTS	SYMBOLIC NAME	TYPE OF	
				ARGUMENT	FUNCTION
ABSOLUTE VALUE	$\| a \|$	1	ABS	Real	Real
			IABS	Integer	Integer
			DABS	Double	Double
TRUNCATION	Sign of a times largest integer $\leqslant \| a \|$	1	AINT	Real	Real
			INT	Real	Integer
			IDINIT	Double	Integer
REMAINDERING	$a_1 \pmod{a_2}$	$\geqslant 2$	AMOD	Real	Real
			MOD	Integer	Integer
CHOOSING LARGEST VALUE	MAX $(a_1, a_2 \ldots.)$	$\geqslant 2$	AMAXO	Integer	Real
			AMAX1	Real	Real
			MAXO	Integer	Integer
			MAX1	Real	Integer
			DMAX1	Double	Double
CHOOSING SMALLEST VALUE	MIN$(a_1, a_2, \ldots.)$	$\geqslant 2$	AMINO	Integer	Real
			AMIN1	Real	Real
			MINO	Integer	Integer
			MIN1	Real	Integer
			DMIN	Double	Double

TABLE A.1.2 INTRINSIC FUNCTIONS (Cont'd)

EXTERNAL FUNCTION	DEFINITION	NUMBER OF ARGUMENTS	SYMBOLIC NAME	TYPE OF	
				ARGUMENT	FUNCTION
FLOAT	Conversion from integer to real	1	FLOAT	Integer	Real
FIX	Conversion from real to integer (as for INT)	1	IFIX	Real	Integer
TRANSFER OF SIGN	sign of a_2 times $\lvert a_1 \rvert$	2	SIGN	Real	Real
			ISIGN	Integer	Integer
			DSIGN	Double	Double
POSITIVE DIFFERENCE	a_1-Min (a_1, a_2)	2	DIM	Real	Real
			IDIM	Integer	Integer
Obtain most sig. part of Double Precision Argument		1	SNGL	Double	Real
Obtain Real part of Complex Argument		1	REAL	Complex	Real
Obtain Imaginary part of COMPLEX argument		1	AIMAG	Complex	Real
Express Single Precision Argument in Double Precision		1	DBLE	REAL	DOUBLE

TABLE A.1.2 INTRINSIC FUNCTIONS (Cont'd)

EXTERNAL FUNCTION	DEFINITION	NUMBER OF ARGUMENTS	SYMBOLIC NAME	TYPE OF	
				ARGUMENT	FUNCTION
Express Two Real Arguments in Complex Form	$a_1+a_2\sqrt{-1}$	2	CMPLX	Real	Complex
Obtain Conjugation of a complex Argument		1	CONJG	Complex	Complex

APPENDIX 2: DIFFERENCES BETWEEN ASA AND 905 FORTRAN

905 FORTRAN is ASA standard FORTRAN, as defined in USASI document X3-9-1966 with the following extensions and restrictions.

a) The following are extensions to ASA standard FORTRAN.

1. Optionally, free format may be used for program and data (see Chapter 5).

2. Facilities for in-line machine code (see Chapter 7).

3. Some relaxation of ASA standard rules on the mixing of arithmetic modes (see Chapter 3).

4. Option of packed integer arrays (see Chapter 2).

b) The following are restrictions in 905 FORTRAN which are not presentinASA FORTRAN.

1. Restrictions on the sequence of statements within a subprogram. The statements which make up a program unit must appear in the following sequence:

   (i) SUBROUTINE or FUNCTION (except in a main program)

   (ii) Specification statements

   (iii) DATA statements

   (iv) Statement function definitions

   (v) Executable statements, FORMAT statements and in-line machine code sections intermixed (in any order).

   (vi) END line.

2. Restrictions on the sequence of items within an EQUIVALENCE group i.e. a set of parenthesized items in an EQUIVALENCE statement.

   a) If a group of equivalenced items includes an item which is also in COMMON, that item must appear first in the equivalence group.

   b) If the same name appears in more than one group that name must appear at the beginning of the second and any subsequent group in which it occurs.

3. Restrictions on names

   The name of a COMMON block must not be the name of a FUNCTION statement. There are also certain restrictions on names beginning with the letter Q (see Chapter 2).

4. Printing of Formatted Records

Standard Fortran specifies that the first character of a formatted record is not printed, but used for vertical format control. 905 FORTRAN does print this character.

5. No BLOCK DATA subprograms in 905 FORTRAN.

c) Compatability between different FORTRAN implementations.

The use of 'A' format almost always causes compatability problems. 905 FORTRAN makes it possible to store up to three characters per 18-bit word for each 'A' format specifier (see Chapter 5).

This problem may be overcome by

(i) storing only one character per storage unit

and (ii) avoiding arithmetic operations or tests on the stored characters.

A program written in FORTRAN in one FORTRAN implementation (on one particular machine) will not necessarily produce the same results on another FORTRAN implementation. The factors which may cause the program to give different results include:

a) Different word lengths and internal number representation.

b) Use of compiler facilities which are extensions to the standard FORTRAN, such as those described for 905 FORTRAN.

c) Conscious or unconscious dependance on effects which are specific to given compiler system.

An example of c) would be use of a labelled COMMON block in a subroutine; the block not declared in the main program. In 905 FORTRAN paper tape compiler environment, the data in the COMMON block will be preserved when the subroutine is re-entered after a return is made to the main program. In systems which use overlays, the data will not necessarily be preserved. Thus a working program may (unknown to the programmer) be using data which is explicitly undefined within the language, and so would therefore have a different effect on different machines. These points are discussed in the National Computing Centre (N.C.C.) 'Standard Fortran Programming Manual' in the N.C.C. Computer Standards Series.

APPENDIX 3: EFFICIENCY CONSIDERATIONS

It is impractical to give detailed rules on the writing of efficient programs, since these would involve detailed knowledge of the compiler. However, the following notes may assist the programmer in economising on either the store used, or the time taken when running his program.

Program Size

Use of the "packed integer arrays" option will reduce the space required for holding such arrays; the only disadvantage arising is in the lack of strict compatibility with ASA standards, and hence with various other compilers.

Each subprogram requires some 20 - 30 words for prologue and argument addresses, and these "red tape" operations also take extra time; it is therefore inefficient to break a program up into a large number of small segments. On the other hand, in an 8k store there is a definite limit to the size of program unit which can be handled by the compiler. Some compromise is therefore needed.

Somewhat similar considerations apply to statement functions, which carry an overhead of about 18 - 25 words. A statement function which performs a trivial operation may take up more space than it saves.

Program Speed

In any program involving floating-point computation, the factor governing the running speed is likely to be time taken by the arithmetic package to perform the computations, together with any type conversions involved. The extent of the computation is normally related in a fairly obvious way to the source program - repetition of sub-expressions, for example, will tend to give rise to repeated calculation (though the compiler will eliminate this in some instances). Any calculation within a DO loop will be repeated as often as the loop, so that any process not related to the value of the control variable should be performed before the DO statement. The question of the type conversion is less obvious, however. It is permissible to introduce integer constants into a real expression, but they must then be converted at object time whenever the expression is evaluated, which is inefficient; if an integer sub-expression involves variables, on the other hand, any constants should be written in integer form, and one conversion will then be made on the value of the expression.

A program which involves only integer working will not be dominated by any one factor in the same way, and the overheads involved in subroutine entry, for example, may start to become significant. In this context, it should be noted that integer division is performed by a subroutine involving about 40 instructions obeyed in a normal manner, and it is therefore very much slower than any arithmetic operation.

The difference between the implementation of assigned GOTO and computed GOTO results in the former being faster; it would normally only be noticeable in an integer program. There is good reason for using the computed form as a normal practice, but a reliable program may gain a little extra speed from conversion to use the assigned form instead.