42794

UNCLASSIFIED

UNITED KINGDOM ATOMIC ENERGY AUTHORITY

ATOMIC WEAPONS RESEARCH ESTABLISHMENT

Manual for the S2 Language

Edited by

Joan Knock

Mary U. Thomas

AWRE,
Aldermaston, Berks.

August 1964
Reprinted April 1967

UNCLASSIFIED

United Kingdom Atomic Energy Authority

ATOMIC WEAPONS RESEARCH ESTABLISHMENT

Manual for the S2 Language

Edited by

Joan Knock
Mary U. Thomas

Summary

The S2 compiler operates on the IBM7030 (STRETCH) and accepts a dialect of FORTRAN 2, which is termed S2, which is described in the present manual.

Approved for issue by

A. H. Armstrong, Senior Superintendent

681.3
681.3(IBM7030)

TABLE OF CONTENTS

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

1. INTRODUCTION

The S2 compiler operates on the IBM7030 (STRETCH) and accepts a dialect of FORTRAN 2 which will be termed S2. It is not the purpose of this reference manual to be a text for beginners learning FORTRAN - they are recommended to read McCracken's handbook; this manual is more concerned with points of detail, so that users of S2 may be in no doubt about what is the proper notation.

Since the FORTRAN language has never been adequately defined, an attitude has developed which can be expressed in the phrase "if it compiles and runs, it is legal". This is to be deplored. Firstly, it lessens the chance that a program written for one compiler will be fit for compilation on another compiler, say for a different machine, and secondly, the program may run correctly because of some accidental property of the compiler or the machine on which programs run.

Accordingly, this reference manual will attempt to indicate which features of FORTRAN are likely to be available in all FORTRAN compilers; this will be termed "Standard FORTRAN". In addition the other features that are available in the S2 dialect will be mentioned. It must be admitted at once that the definition of what is "Standard FORTRAN" is rather arbitrary and expresses the opinion of the author. This is all that can be done until such time as "Standard FORTRAN" is properly defined.

The following persons took part in the construction of the S2 compiler:-

Miss M. U. Thomas and Messrs. B. Blythe, A. E. Glennie and C. Hart of AWRE and Mr. F. R. A. Hopgood of AERE.

Messrs. P. Ellis, J. Nash and J. Pether of IBM (UK) Ltd.

2. ELEMENTS OF S2

This section describes the elements out of which S2 statements are constructed.

2.1 Characters

The alphabet	A to Z
The numerals	0 to 9
Others	+ - = / ( ) . , £ * space

The character ' is available in S2 where it is treated as a letter. Its use is not recommended.

2.2 Operators

Addition	+	
Subtraction	-	
Sign change	-	used as a unary operator
Multiplication	*	
Division	/	
Exponentiation	**	A**B is A^B
Equals	=	

2.3 Names

Variables, parameters and routines have names and much confusion has arisen because the syntax of names has changed gradually with extensions to FORTRAN. The following rules apply to S2 and probably to all other FORTRAN compilers, at least from now on:-

A name is a string of letters or digits whose first character is a letter. "Standard FORTRAN" requires that there should be fewer than 7 characters in a name.

There are a number of safety precautions which should be adopted by the careful programmer:-

(a) If the name of a function ends with F, do not use the name obtained by stripping the last F.

(b) Do not use as a name the key works like "IF", "CALL", etc., which specify statement types.

(c) If you are writing a program that may be compiled by the IBM7090 FORTRAN 2 compiler, and uses functions, you should consult the appropriate reference manual for the rules of the naming of functions; these rules are too long and complicated to reproduce here.

Such safety rules are not mandatory, but lead to compatibility with other dialects of FORTRAN.

The actual S2 rules for naming are:-

(a) The first character is alphabetic.

(b) Seven or fewer characters in the name.

(c) If there are 7 characters, the seventh character not being E, then the 6-character name obtained by stripping the seventh character is not used for a different object from the 7-letter name. In other words, if the name has seven characters, only the first six are used, unless the seventh character is E.

Examples:

The first two names are the same to the compiler; the others are all different.

XKLIBR	XKLIBRF
PRELUD	PRELUDE
ALPHA	OMEGA
A1	S2

2.4 What Can Be Named

The following classification shows what use may be made of names:-

2.4.1 INTEGER

These are named by names whose first letter is I, J, K, L, M or N. In S2 it is possible to declare any other name to be that of an INTEGER by means of the declaration

INTEGER list

where list is a list of names separated by commas if there is more than one name in the list.

An INTEGER is represented within S2 programs on STRETCH by a floating point number with an exponent of 38, and the appropriate fraction to give the floating point number the INTEGER value required.

Thus, 38 bits are used to store the significant bits of the number, which must therefore be less than 2^{38} in magnitude.

Let I and J be INTEGERs and K the result of adding I to J by an S2 program. Then if the sum of the values of I and J is greater than or equal to 2^{38}, then K should be considered undefined.

Similarly with other overflow situations arising from other operations on INTEGERs. No warning is given of this situation.

The user is advised to use INTEGER arithmetic only if the numbers used are not too large, particularly if he wishes his programs to run on other machines. For example, on the IBM7090 INTEGERs are less than 2^{15}.

2.4.2 REAL

These are floating point numbers in whose names the first letter is not I, J, K, L, M or N. In S2 it is possible to use a type declaration

REAL list

as with INTEGERs to declare as REALs any quantities whose names begin with I, J, K, L, M or N.

In STRETCH, floating point numbers, x, have a 48 bit fraction, A (and hence 48 significant bits) together with an exponent, B, which can be in the range

$- 1023, + 1023,$

where $x = \pm A * 2^B$, $A = 0$ or $\frac{1}{2} \leq A < 1$.

As well as this normal range there are two other ranges called the XFP and XFN ranges. For practical purposes all numbers in the XFP range can be considered identical; they are numbers with exponent greater than 1023. Similarly all XFN numbers can be considered identical; they have exponents less than - 1023. STRETCH has the useful feature that the arithmetic includes the XFP and XFN numbers in such a manner that calculation is consistent with the interpretation that an XFP number is considered like infinity and an XFN number like an indefinitely small number (but not necessarily zero). Thus, we have the following rules:-

XFP + XFN = XFP

and many others given in full in Appendix A, by which the programmer is relieved (at least on STRETCH) from considering overflow and underflow of numbers.

The treatment of zero within STRETCH is a complicated subject and is treated in Appendix B.

2.4.3 Fixed Point Constants

1 or more decimal digits. The value of the constant must be less than 2^{38} in magnitude.

Examples: 1 12345

2.4.4 Floating Point Constants

Any number of decimal digits with a decimal point.

Examples: 1.0 1. .1 1.23456789

A decimal exponent may immediately follow a decimal number and is written by placing E and then the signed or unsigned exponent after the decimal number. The decimal exponent should be less than 512 in magnitude.

Examples:	1.23 E2	(means 123.)
	1.23 E+2	(also means 123.)
	123. E-2	(means 1.23)

The decimal point is not always required.

Example: 123 E-2 (means 1.23)

2.4.5 Function Values

Examples: SINF(X) B(I,J)

The form of a function value is:-

Name of function followed by the argument or argument list enclosed in parentheses. An argument may be any expression; an argument list is a list of arguments separated by commas. The expressions used as arguments may be as general as you please.

Whether the value of the function is REAL or INTEGER depends on whether the name of the function obeys the definition rules for the names of REAL or INTEGER variables.

This rule does not apply to a predetermined set of special functions, the built in and library functions, which will have REAL values unless their names begin with X in which case they will have INTEGER values.

2.4.6 Subscripted Variables

Examples: C(I,J) D(3,2*K-6)

The form of a subscripted variable is:-

Name of array followed by a subscript expression or a subscript list enclosed within parentheses.

The name of an array is a name of an INTEGER or REAL (see Sections 2.4.1 or 2.4.2) about which a DIMENSION statement has been made; and it is this that differentiates it from the function value whose form can be the same as that of a subscripted variable.

Subscripts can appear only in certain forms, namely:-

v, where v is an INTEGER

c, where c is a fixed point constant

v+c

v-c

c*v+c

c*v-c

These are the "Standard FORTRAN" subscripts. However, S2 allows an additional class of subscripts similar to the above forms in which c may be parameters, which for this purpose are:-

Subroutine parameters

Function parameters

Parameters of DO statements in whose range the subscript appears

They must still be INTEGERs either by form or declaration.

Whereas "Standard FORTRAN" limits the number of subscripts within a subscripted variable to 3 (and hence handles arrays of three dimensions at most), in S2 the limit depends on the complication of the subscripted variables actually used.

Let N be the number of dimensions of the array and P be the number of + and - in the subscripts, then 2N + P must be less than 18 and (N + 1) (2N + P + 2) must be less than 100.

These conditions mean that four dimensional arrays can be used freely, but that for arrays of higher dimension some of the subscripts must be of the simpler type. Also the number of dimensions must not exceed 6.

3. EXPRESSIONS AND ARITHMETIC STATEMENTS

By expressions we mean formulae that are to be evaluated when the program is run on the computer.

Examples: X Y + Z ABSF(C/D) etc.,

of which the second and third examples are undoubtedly expressions, but the first is an example of an expression in certain contexts only.

By the mode of an expression we mean to indicate the type of value that is calculated, whether it is fixed point (INTEGER) or floating point (REAL).

We now give rules for the formation of expressions.

3.1 Components of Expressions

These are the items whose values are used when the expression comes to be computed, they are:-

Fixed point variables (INTEGERs)

Fixed point constants

Fixed point subscripted variables

Floating point variables (REALs)

Floating point constants

Floating point subscripted variables

Function values of either mode

3.2 Named Variables

In order to be able to change a variable, there are certain statements of the language in which variable names appear to show which variable acquires a new value. Such appearances are called named variables and only the following items can so appear:-

Fixed point variables

Floating point variables

Fixed point subscripted variables

Floating point subscripted variables

3.3 Arithmetic Statements

The most common way of changing the value of a variable is by the arithmetic statement which has the form:-

Named variable = expression

Examples:	A=B+C	meaning calculate B+C and set A to have the value calculated.
	I=J	get the value of J and set I to have that value.

These two examples illustrate the case where the mode of the named variable on the left hand side of the equals sign is the same as the mode of the expression. When the modes of the two sides of the arithmetic statement differ the compiler inserts an operator on the right hand side so that its value is converted to the mode specified by the named variable on the left hand side. The prototypes of these forms of the arithmetic statement are:-

```
A = I
J = B
```

which are interpreted as meaning

```
A = FLOATF(I) and
J = XFIXF(B), respectively.
```

The function FLOATF changes the representation of the value of its argument expression of INTEGER mode to be in REAL mode. This has no effect on the value considered as a number, but the effect is to get the representation into the form appropriate for storage as a REAL.

The function XFIXF takes the integral part of the value of its REAL argument expression and alters its representation to that appropriate for INTEGERs.

Note that on STRETCH the function XFIXF(X) takes as its result the nearest integer to X between zero and X.

Thus, XFIXF(- 3.4) is - 3 and not - 4. This is an accidental result of the number representation within that particular machine. It is not wise to assume that this definition will apply to machines like ATLAS with a different type of number representation for negative numbers.

3.4 How Expressions are Constructed

Expressions follow the normal rules for algebra in most cases. The complete rules are given below.

To describe the rules it is necessary to say what the hierachy of the operators is. This we do by defining a hierachy number as follows:-

Hierachy Number	Operator
4	+ - (when used as unary signs)
3	** (exponentiation)
2	* /
1	+ -

The ordering of the calculation is primarily dominated by brackets if they exist. Operators of high hierachy number act before operators of low hierachy number, and operators of the same hierachy number act from left to right. Additional rules are:-

(a) No operator may immediately follow another without the intervention of a term, except when ** is followed by a (unary) + or -.

(b) Subexpressions like A**B**C, A**B** ... **X are illegal. The effect desired should always be indicated by using brackets.

Examples:-

(1) A+B+C+D is equivalent to ((A+B)+C)+D

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

If the expression (A*B)/(C*D) is meant then it is necessary to place brackets around the denominator.

3.5 Modes of Expressions

S2 allows expressions containing a mixture of INTEGER and REAL quantities. The mode of an expression is found by finding the modes of the partial expressions taken in the order in which they would be calculated.

The mode of the expression in which two sub expressions are combined by an operator is REAL unless both components are of INTEGER mode, in which case the result is of INTEGER mode.

Examples:	I+J	INTEGER mode
	X*Y	REAL mode
	X+I	REAL mode

A unary sign does not alter the mode.

The effect may be stated in another way. The mode of an expression is REAL unless all its components are INTEGER, in which case the expression is of INTEGER mode.

Allowance must of course be made for mode changes specifically requested by the built in functions XFIXF, FLOATF etc.,; Section 5.3 gives full details.

Programmers are warned of misinterpreting these rules when using division. Consider the two examples:

(a) (I/J)*X and

(b) (X*I)/J

(the parentheses are inserted to remind the reader of the ordering of the calculation. They could be omitted without altering the expressions.)

In example (a), the division is of INTEGERs and the result is an INTEGER. Thus, if I = 5, J = 3, X = 2.0 the first expression would have the value 2.0 (= 1*2.0) since the result of dividing 5 by 3 is 1. In the second example the numerator X*I is REAL and the division is then conducted in floating and the result is 3.33 ... 3 ...

4. CONTROL STATEMENTS

We have already mentioned Arithmetic Statements in Section 3.3, without saying what a statement is. A statement is a generic term for a step of the calculation as expressed in the S2 language. Normally the steps of the calculation proceed from statement to statement in the order of writing, but to make choices of alternatives within the calculation we require "Control Statements" by which the stepping from one statement to the next in order of writing may be arrested and a new sequence of statements performed.

So that reference may be made to these sequences of statements we require to be able to label the first statements of such sequences by a "Statement Label", or statement number, which is an integer composed of 5 or fewer decimal digits appearing to the left of the statement in a field preserved for this purpose. Only those statements that need to be labelled should be labelled since the efficiency of translation is reduced (but not its accuracy) by the presence of unnecessary labels.

4.1 Unconditional GO TO

General Form: "GO TO n", where n is a statement label.

Example: GO TO 5

This statement causes transfer of control to the statement with statement label n.

4.2 Computed GO TO

General Form: "GO TO(n_1, n_2, ..., n_m), i", where n_1, n_2, ..., n_m are statement labels and i is a non-subscripted fixed point variable.

Example: GO TO (30, 42, 50, 9), K

Control is transferred to the statement with statement label n_1, n_2, n_3, ..., n_m, depending on whether the value of i at time of execution is 1, 2, 3, ..., m, respectively. Thus, in the example, if i is 3 at the time of execution, a transfer to the 3rd statement of the list, namely statement 50, will occur. If the value of i is not in the range i,m the program will fail. This statement is used to obtain a computed many-way branch.

4.3 Assigned GO TO

General Form: "GO TO K, (n_1, n_2, ..., n_m)", where K is a non-subscripted fixed point variable appearing in a previously executed ASSIGN statement, and n_1, n_2, ..., n_m are statement labels.

Example: GO TO K, (17, 10, 19)

This statement causes transfer of control to the statement whose label n was last assigned by an ASSIGN statement; n_1, n_2, ..., n_m are a list of the possible labels. The assigned GO TO is used to obtain a pre-set many-way branch.

4.4 ASSIGN

General Form: "ASSIGN i TO n", where i is a statement label and n is a non-subscripted fixed point variable which appears in an assigned GO TO statement.

Example: ASSIGN 12 TO I

This statement causes a subsequent GO TO n,(n_1,...,n_i, ..., n_m) to transfer control to the statement with the label i.

4.5 IF

General Form: "IF (a) n_1, n_2, n_3", where a is an expression and n_1, n_2, n_3 are statement labels.

Example: IF(A(J,K)-B)10, 14, 30

Control is transferred to the statement with the statement label n_1, n_2 or n_3 if the value of a is less than, equal to, or greater than zero, respectively.

4.6 SENSE LIGHT

General Form: "SENSE LIGHT i", where i is 0, 1, 2, 3 or 4.

Example: SENSE LIGHT 3

If i is 0, all four SENSE LIGHTs will be turned Off; otherwise SENSE LIGHT i only will be turned On.

4.7 IF SENSE LIGHT

General Form: "IF (SENSE LIGHT i)n_1, n_2", where n_1 and n_2 are statement labels and i is 1, 2, 3 or 4.

Example: IF (SENSE LIGHT 3) 80, 40

Control is transferred to the statement with the statement label n_1 or n_2, according to whether SENSE LIGHT i is On or Off, respectively. If the light is On it will be turned Off automatically.

4.8 DO

General Form: "DO n i = m_1, m_2" or "DO n i = m_1, m_2, m_3", where n is a statement label, i is a non-subscripted fixed point variable, and m_1, m_2, m_3 are each either an unsigned fixed point constant or non-subscripted fixed point variable. If m_3 is not stated, it is taken to be 1.

Examples: DO 30 I = 1, 100 DO 30 I = 1, M,3

The DO statement is a command to perform repeatedly the statements which follow, up to and including the statement with statement label n. The first time the statements are performed with i = m_1. For each repitition i is increased by m_3. After they have been repeated with i equal to the highest value <u>which does not exceed m_2</u>, control passes to the statement following the last statement in the range of the DO.

The <u>range</u> of a DO is that set of statements which will be performed repeatedly; i.e., it is the sequence of consecutive statements immediately following the DO, up to and including the statement labelled with n.

The <u>index</u> of a DO is the fixed point variable i, which is available for computation, either as an ordinary fixed point variable or as the variable of a subscript. After the last execution of the range, the DO is said to be <u>satisfied</u>. The index i must not be altered by any statement within the range.

4.9 CONTINUE

General Form: "CONTINUE".

Example: CONTINUE

CONTINUE is a dummy statement which gives rise to no instructions in the object program. It is most frequently used as the last statement in the range of a DO. Its use in this way is essential to provide a transfer address for IF and GO TO statements which are intended to begin another repetition of the DO range.

Suppose that control has reached statement 10 of the program

```
.
.
.
10  DO 12 I = 1, 10
11  A(I) = I*N(I)
12  CONTINUE
.
.
.
```

The range of the DO is to statement 12, and the index is I. The DO sets I to 1 and control passes into the range. The value of 1.N(1) is computed, converted to floating point, and stored as A(1). Since statement 12 is the last statement in the range of the DO and the DO is unsatisfied, I is increased to 2 and control returns to the beginning of the range. The value of 2.N(2) is then computed and stored as A(2). The process continues until statement 11 has been executed with I = 10. Since the DO is satisfied, control then passes to the statement following the CONTINUE.

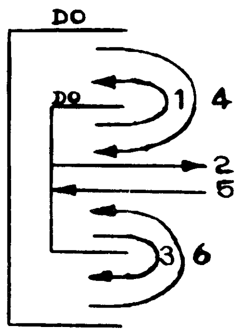
<u>DOs within DOs</u>. Among the statements in the range of a DO may be other DO statements. If so, the following rule must be observed:-

Rule: If the range of a DO includes another DO, then all the statements in the range of the latter must also be in the range of the former.

A set of DOs satisfying this rule is called a nest of DOs.

<u>Transfer of Control and DOs</u>. Transfers of control to and from the range of a DO are subject to the following rule:-

Rule: No transfer is allowed into the range of any DO from outside its range. Thus, in the configuration below, 1, 2 and 3 are legal transfers, but 4, 5 and 6 are not.



Status of the Cell Containing I Outside the Range of the DO

A DO loop with index I does not affect the contents of the storage location for I, except if exit occurs by a transfer out of the range, when the I cell contains the current value of I. Therefore, if a normal exit occurs from a DO, the I cell contains what it did before the DO was encountered.

4.10 STOP

General Form: "STOP".

Example: STOP

This statement causes immediate termination of the job, and the initiation of the next job.

4.11 END

General Form: "END".

Example: END

The END statement is the last written statement of any sub-program; it signals the end of written sub-program to the compiler. It is not intended to be an executable statement, but if control does reach it in programs compiled by the S2 compiler the action will be as if the RETURN statement had been performed.

5. FUNCTIONS

The following are examples of the use of functions:-

(1) X = SINF(Y)

(2) A = B(I,J)

A function is a term appearing on the right hand side of an arithmetic statement and its form is:-

name of function, left bracket, argument(s), right bracket,

where commas are used to separate the arguments (if there is more than one argument) which may be expressions.

The form of a function is distinguished from the form of a subscripted variable only because of the absence of a DIMENSION declaration about the name before the left bracket.

Note: The omission of such DIMENSION declarations is a fruitful source of error, since what the programmer may have intended to be a subscripted variable <u>is</u> (and is treated as) a function.

5.1 Types of Functions

There are three types of functions:-

(a) Closed (or Library) functions,

(b) Open functions, and

(c) Sub-program functions.

To the user the first two are virtually indistinguishable in use; their names and definitions are given in the following section. For conventions about the names of sub-program functions see Section 5.4.

5.2 Closed (or Library) Functions Available

SQRTF(X)	square root function
ATANF(X)	arctangent function with result in the first or fourth quadrant
LOGF(X)	natural logarithm
EXPF(X)	exponential function

COSF(X)	cosine
SINF(X)	sine
TANH(X)	hyperbolic tangent

These elementary functions with single REAL argument are always available. They give a REAL result and have the highest possible precision and range. With the exception of SQRTF they can be safely used with INTEGER argument, although this is not a recommended procedure.

The trigonometrical functions are for angles in radians.

5.3 Open Functions Available

ABSF(V)	absolute value of REAL argument
XABSF(V)	absolute value of INTEGER argument
INTF(V)	integral part of REAL argument, e.g., INTF(3.14) = 3. but INTF(- 5.3) = 5.
XINTF(V)	integral part of REAL argument with INTEGER result, in contrast to INTF which has REAL result
XFIXF(V)	synonymous with XINTF
MODF(V1,V2)	defined as V1 - V2 * INTF(V1/V2). The arguments and result are REAL
XMODF(V1,V2)	defined as V1 - V2 *XINFT (V1/V2). The arguments and result are INTEGER
MAXoF(V1,V2 ...)	The REAL result is the algebraically largest of the two or more INTEGER arguments
XMAXoF(V1,V2 ...)	as MAXoF but with INTEGER result
MAX1F(V1,V2 ...)	as MAXoF but with REAL arguments

XMAX1F(V1,V2...)	INTEGER result of the maximum of REAL arguments. The definition of this function is doubtful unless the arguments have integral values
MINoF(V1,V2...)	the REAL result is the algebraically smallest of the two or more INTEGER arguments
XMINoF(V1,V2...)	as MINoF but with INTEGER result
MIN1F(V1,V2...)	as MINoF but with REAL arguments
XMIN1F(V1,V2...)	INTEGER result of the minimum of REAL arguments. The definition of this function is doubtful unless the arguments have integral values
FLOATF(V)	the result is the REAL form of the INTEGER argument
SIGNF(V1,V2)	the REAL result has the sign of V2 and the magnitude of V1. V1 and V2 are REAL
XSIGNF(V1,V2)	the INTEGER result has the sign of V2 and the magnitude of V1, where V1 and V2 are INTEGERs
DIMF(V1,V2)	V1 - V2 if V1 is greater than V2, otherwise zero. The arguments and result are REAL
XDIMF(V1,V2)	As DIMF but with INTEGER result and arguments

These are the standard FORTRAN definitions of these functions; they should be available on all FORTRAN compilers. Note particularly the lack of definition of XMAX1F and XMIN1F when the arguments are not whole numbers. In the interest of safety it would be better to use XFIXF(MIN1F( )) instead of XMIN1F( ), because there is no doubt about the interpretation of the former.

5.4 Function Sub-Programs

These are used as described in Section 5 on functions, but unlike the specific functions mentioned there, they must be written by the programmer as described below.

There is a convention about the mode of the result produced by a function; it is REAL (floating point) or INTEGER (fixed point) according to the interpretation of the name of the function if it were the name of a variable. Thus, function sub-programs whose names begin with I, J, K, L, M or N (or whose names are declared to be INTEGER) have INTEGER result.

The value calculated by a function is a single value, being a function of its stated arguments only. It is assumed not to change COMMON or any of its arguments unless the function occurs once in an isolated statement. An isolated statement is a labelled statement, followed by another labelled statement or by a control statement.

FUNCTION

General Form: "FUNCTION Name (a_1, a_2'..., a_n)", where Name is the symbolic name of a single-valued function, and the arguments a_1, a_2, ... a_n, of which there must be at least one, are non-subscripted variable names, all different. The function name must not occur in a DIMENSION statement in the FUNCTION sub-program, or in a DIMENSION statement in any program which uses the function. The arguments may be any variable names used in the sub-program.

Examples:
```
FUNCTION ARCSIN (RADIAN)
FUNCTION ROOT (B, A, D)
FUNCTION INTRST (RATE, YEARS)
```

In a FUNCTION sub-program, the name of the function must be used at least once as the variable on the left hand side of an arithmetic statement, or alternately in an input statement list, e.g.,

```
FUNCTION NAME (A, B)
.
.
.
NAME = B + A
.
.
.
RETURN
```

The arguments that follow the name in the "FUNCTION statement" are the names of locations within the program compiled for the FUNCTION. When a FUNCTION is evaluated, the actual values of the arguments are copied from the calling program into these locations, and the FUNCTION sub-program uses these copies. A SUBROUTINE (see Section 6) uses this same technique to get its arguments, but in addition, also returns the values of its arguments to the calling program on exit to it.

For correct use of FUNCTIONs (and SUBROUTINEs) it is necessary to ensure that there is an exact correspondence between the types of arguments in the "FUNCTION Statement" and in the FUNCTION as used. Since in the compiled programs the name of an array defines a location containing the base address of an array it is possible to specify arrays that are arguments only by the non-subscripted array name. Only the value of the base address is copied, not the elements of the array, which the FUNCTION can now access using the base address value, provided the values of the dimensions of the array in the calling program and FUNCTION agree.

6. SUBROUTINES

In S2, the programmer can not only write his own functions in the language, but also his own subroutines.

To do this a SUBROUTINE statement is required to define that a compiled program is to be a SUBROUTINE. This declaration must occur with the other declarations before an obeyable statement is written. This statement defines the name of the SUBROUTINE and its parameters, if any.

To indicate when control leaves a SUBROUTINE (or FUNCTION sub-program), a RETURN statement must be written so that control is returned from the sub-program to the program that called it. What the SUBROUTINE is to calculate is written in the S2 language.

To indicate when a program requires to enter a SUBROUTINE, a CALL statement must be written which specifies the name of the SUBROUTINE required and the values or names of its parameters required for that particular use of the SUBROUTINE.

6.1 SUBROUTINE

General Form: "SUBROUTINE Name (a_1, a_2, ..., a_n)", where Name is the symbolic name of a sub-program, and the arguments a_1, a_2, ..., a_n, if any, are non-subscripted variable names, all different. The name of the sub-program must not be listed in a DIMENSION statement of any program which calls the sub-program, or in a DIMENSION statement of the SUBROUTINE itself. The arguments may be any variable names used in the sub-program.

Examples:
```
SUBROUTINE MATMPY(A,N,M,B,LD)
SUBROUTINE QDRTIC(B,A,D, ROOT1,
ROOT2)
SUBROUTINE NOARG
```

This statement defines the sub-program to be a SUBROUTINE.

6.2 CALL

General Form: "CALL Name (a_1, a_2, ..., a_n)", where Name is the name of a SUBROUTINE, and a_1, a_2, ..., a_n are arguments.

Examples:
```
CALL MATMPY(X, 5, 10, Y, 17, Z)
```

This statement is used to send control to a SUBROUTINE, and to state a list of arguments, if any. An argument is either an expression (special case, a single value) or a named variable (see Section 3.2). Only the latter type of argument may be used when the argument is changed by the SUBROUTINE.

Extravagant effects may result from disregard of this rule; if a constant is used as an argument and the SUBROUTINE changes that argument internally, then the constant is destroyed, its value changing according to the action of the SUBROUTINE. Such effects should not be deliberately exploited.

The arguments are not restricted to ordinary variables and expressions but may be array names or variable dimensions of arrays. For example, the SUBROUTINE headed by

```
SUBROUTINE MATMPY (A,N,M,B,L,D)
```

could be called by the main program via the statement

```
CALL MATMPY (X, 5, 10, Y, 17, Z)
```

where the variables A, B, C are the names of arrays. A, B and C must appear in a DIMENSION statement in SUBROUTINE MATMPY and X, Y and Z must appear in a DIMENSION statement in the calling program. The dimensions assigned must be the same in both sub-programs, and can be given to the SUBROUTINE as arguments, if variable.

6.3 RETURN

General Form: "RETURN".

Example: RETURN

This statement ends any sub-program, whether a SUBROUTINE or a FUNCTION, and returns control to the calling program. A RETURN statement must be the last executed statement of the sub-program, but need not be physically the last statement. Any number of RETURN statements may be used.

7. INPUT/OUTPUT STATEMENTS

There are thirteen statements for the transmission of information between storage on the one hand, and magnetic tapes, disc, card reader, card punch and printer on the other hand.

7.1 Types of INPUT/OUTPUT Statements

(1) Five statements (READ, READ INPUT TAPE, PUNCH, PRINT and WRITE OUTPUT TAPE) cause transmission of a list of quantities between storage and an input/output medium: cards, printed sheet, or magnetic tape, for which information is expressed in Hollerith punching, alphanumerical print, or binary-coded-decimal tape code, respectively.

(2) One statement (FORMAT), which is a non-executable statement, specifies the arrangement of the information in the input/output medium for the five source statements of group 1 above. (See Section 10.)

(3) Four statements (READ TAPE, READ DRUM, WRITE TAPE and WRITE DRUM) cause information to be transmitted in binary code. (Note: Variant spellings are allowed DISK=DISC=DRUM.)

(4) Three statements (END FILE, BACKSPACE and REWIND) manipulate binary magnetic tapes.

Additional methods of using tape and disc are given in Appendices I and J.

It is important to note that the different types of tape and disc manipulative statements can not be used indiscriminately. The statements of Section 9 use a system of buffering (hidden from the programmer's knowledge) and thereby go faster than the statements of Section 8, for which the transmission is unbuffered. The methods of Appendix I also do not use hidden buffering, but allow the programmer to control his own timing.

When a tape is used with the statements that use the method of buffering, they must never be manipulated by any statements of the non-buffering type, and vice versa.

7.2 Lists of Quantities

Of the input/output statements, nine call for the transmission of information and include a list of the quantities to be transmitted. This list is ordered, and its order must be the same as the order in which the words of information exist (for input), or will exist (for output) in the input/output medium.

The formation and meaning of a list is best described by an example:-

A, B(3), (C(I), D(I,K), I = 1, 10)

If this list is used with an output statement the information will be written in this order:-

A, B(3), C(1), D(1,K), C(2), D(2,K), ..., C(10), D(10,K).

Similarly, if this list were used with an input statement, the successive words, would be placed into the sequence of storage locations given.

The order of the above list can thus be considered the equivalent of the "program" below in which the repetition induced by the indexing component within the parentheses is seen to follow the conventions of DO statements:-

```
(1)  A
(2)  B(3)
(3)  DO 5 I    1, 10
(4)  C(I)
(5)  D(I,K)
```

When reading a list of the form K, (A(K)) or K, (A(I), I = 1,K), where an index or indexing parameter itself appears earlier in the list of an input statement, the indexing will be carried out with the newly read-in value.

8. DECIMAL INPUT/OUTPUT

Decimal INPUT/OUTPUT routines are called by the "FORTRAN Statements":-

(a) PRINT, PUNCH, READ, and

(b) WRITE OUTPUT TAPE, READ INPUT TAPE.

The routines of group (a) use the system tapes; those of group (b) use programmer's tapes, unless the tape number specified is zero, in which case WRITE OUTPUT TAPE 0 acts like PRINT and READ INPUT TAPE 0 acts like READ.

Tape records written or read by routines of group (b) have up to 132 characters in A6 code (a subset of the 1401 code) and tapes prepared by WRITE OUTPUT TAPE can be read by READ INPUT TAPE. Users are advised to use statements of group (a) whenever possible since the error correction and detection is more satisfactory.

During execution of these INPUT/OUTPUT programs error conditions may occur, causing the job to be rejected and a diagnostic message to be printed. The conditions are:-

(1) Data field errors in input records.

(2) Errors in the format.

(3) A tape mark is read.

(4) An end of tape is sensed when writing (except if the tape is a system tape).

(5) "Permanent" reading or writing errors with a programmer's tape. An additional message will be typed to warn the operator.

8.1 READ

General Form: "READ n, List", where n is the statement label of a FORMAT statement, and List is as previously described.

Example: READ 1, ((ARRAY (I,J), I = 1, 13), J = 1, 15)

The READ statement causes the reading of cards from the card reader using the system input tape. Card after card is read until the complete list has been brought in, converted, and stored in the locations specified by the list of the READ statement. The FORMAT statement describes the arrangement of data on the cards and the type of conversion to be made.

8.2 READ INPUT TAPE

General Form: "READ INPUT TAPE i, n, List", where i is an unsigned fixed point constant or a fixed point variable, n is the statement label of a FORMAT statement, and List is as previously described.

Examples: READ INPUT TAPE 4, 30, AK, A(J)

READ INPUT TAPE N, 30, IK, A(J)

The READ INPUT TAPE statement causes reading of BCD information from tape unit i ($1 \leq i \leq 8$). Record after record is brought in, in accordance with the FORMAT statement, until the complete list has been read.

8.3 PUNCH

General Form: "PUNCH n, List", where n is the statement label of a FORMAT statement, and List is as previously described.

Example: PUNCH 30, (A(J), J = 1, 20)

The PUNCH statement causes the program to punch Hollerith cards via the system output tape. Cards are punched in accordance with the FORMAT statement until the complete list has been punched.

8.4 PRINT

General Form: "PRINT n, List", where n is the statement label of a FORMAT statement and List is as previously described.

Example: PRINT 2, (A(J), J = 1, 20)

The PRINT statement causes the program to print output data via the system output tape. Successive lines are printed in accordance with the FORMAT statement, until the complete list has been printed.

8.5 WRITE OUTPUT TAPE

General Form: "WRITE OUTPUT TAPE i, n, List", where i is an unsigned fixed point constant or a fixed point variable, n is the statement label of a FORMAT statement, and List is as previously described.

Examples: WRITE OUTPUT TAPE 2, 30, (A(J), J = 1, 10)

WRITE OUTPUT TAPE L, 30, (A(J), J = 1, 20)

The WRITE OUTPUT TAPE statement causes the program to write BCD information on symbolic tape unit i ($1 \leq i \leq 8$). Successive records are written in accordance with the FORMAT statement until the complete list has been transmitted.

9. BINARY INPUT/OUTPUT

Binary Tape INPUT/OUTPUT Routines

These routines are called by the "FORTRAN Statements" READ TAPE, WRITE TAPE, END FILE, BACKSPACE and REWIND and they may use programmer's tapes only.

During execution of these routines the following errors are detected causing the job to be rejected:-

(1) End of tape sensed while writing on a tape.

(2) "Permanent" read or writing error.

(3) An uncorrectable error signalled by MCP such as UK or EE interrupt occurring without EOP, or an EPGK interrupt.

In cases (2) and (3) operator messages are typed in addition to diagnostic information on the programmer's printed output.

Binary Disk INPUT/OUTPUT Routines

These are called by the "FORTRAN Statements" READ DISK and WRITE DISK (allowing variant spellings DISK = DISC = DRUM).

Errors detected causing diagnostic messages to the programmer and to the operator are:-

(1) "Permanent" locate, read or write errors.

(2) Uncorrectable errors signalled by MCP.

These cause job rejection. In this as in all other cases of job rejection, any possible machine error is signalled to the operator. The operator will not be informed of programmer's errors.

9.1 READ TAPE

General Form: "READ TAPE i, List", where i is an unsigned fixed point constant or a fixed point variable and List is as previously described.

Examples: READ TAPE 4, (A(J), J = 1, 20)

READ TAPE K, (A(J), J = 1, 20)

The READ TAPE statement causes the program to read binary information from symbolic tape unit i ($1 \leq i \leq 8$) into locations specified in the list. A record is read completely only if the list specifies as many words as the tape record contains; no more than one record will be read. The tape, however, always moves to the beginning of the next record.

The program checks tape reading. In the event that a record cannot be read properly, the job is terminated.

9.2 READ DRUM

General Form: "READ DRUM i, j, List", where i and j are each either an unsigned fixed point constant or a fixed point variable, with the value of i between 9 and 12 and List is as previously described.

Examples:
```
READ DRUM 9, 100, A, B, C, D (3)
READ DRUM K, J, A, B, C, D (3)
```

The READ DRUM statement causes the program to read words of binary information from the disc from consecutive locations on file i, beginning with the word in arc number j. Reading continues until all words specified by the list have been read. Instead of DRUM one may also write DISC or DISK.

Every logical record must start at a fresh arc. There are 512 words per arc. About 1½ million words are available and these may be divided into several distinct logical files, each of a specified number of tracks. One track holds 4096 words.

9.3 WRITE TAPE

General Form: "WRITE TAPE i, List", where i is an unsigned fixed point constant or a fixed point variable, and List is as previously described.

Examples:
```
WRITE TAPE 4, (A(J), J = 1, 10)
WRITE TAPE K, (A(J), J = 1, 10)
```

The WRITE TAPE statement causes the program to write binary information on the tape unit with tape number i ($1 \leq i \leq 8$). One record is written consisting of all the words specified in the list.

The object program checks tape writing. In the event that a record cannot be written properly, the job is terminated.

9.4 WRITE DRUM

General Form: "WRITE DRUM i, j, List", where i and j are each either an unsigned fixed point constant or a fixed point variable, with the value of i between 9 and 12 and List is as previously described.

Examples: WRITE DRUM 9, 10, A, B, C, D (6)

WRITE DRUM K, J, A, B, C, D (6)

The WRITE DRUM statement causes the program to write words of binary information on the disc on to consecutive locations on i, beginning with arc number j. Writing continues until all the words specified by the list have been written. Instead of DRUM one may also write DISC or DISK.

9.5 END FILE

General Form: "END FILE i", where i is an unsigned fixed point constant, or a fixed point variable.

Examples: END FILE 2

END FILE K

The END FILE statement causes the program to write an end-of-file mark on tape unit i ($1 \leq i \leq 8$).

9.6 REWIND

General Form: "REWIND i", where i is an unsigned fixed point constant, or a fixed point variable.

Examples: REWIND 3

REWIND K

The REWIND statement causes the program to rewind tape unit i ($1 \leq i \leq 8$).

9.7 BACKSPACE

General Form: "BACKSPACE i", where i is an unsigned fixed point constant, or a fixed point variable.

Examples: BACKSPACE 1

BACKSPACE K

The BACKSPACE statement causes the program to backspace tape unit i ($1 \leq i \leq 8$) by one record.

10. FORMATS

General Form: "FORMAT (Specification)", where Specification is as described below.

Example: FORMAT (I2/(E12.4,F10.6))

The five INPUT/OUTPUT statements of Section 8 contain, in addition to the list of quantities to be transmitted, the statement label of a FORMAT statement describing the information format used. It also specifies the conversion to be performed between the internal machine-language and external notation. FORMAT statements are not executed, they merely supply information to the object program.

For illustration, the details of writing a FORMAT specification are given below for use with PRINT statements. However, the description is valid for any case by generalizing the concept of "printed line" to that of record in the input/output medium. A record may be for this description:-

(1) A printed line with a maximum of 132 characters.

(2) A punched card with a maximum of 80 characters.

(3) A BCD tape record with a maximum of 132 characters.

Three types of decimal-to-binary or binary-to-decimal conversion are possible, and are shown as follows:-

INTERNAL	Type	EXTERNAL
Floating point variable	E	Floating point, decimal
Floating point variable	F	Fixed point, decimal
Fixed point variable	I	Decimal integer

The FORMAT sepcification describes the line to be printed by giving, for each field in the line (from left to right, beginning with the first printing position):-

(1) The type of conversion (E, F, or I) required.

(2) The width (w) of the field.

(3) For E- and F-type conversion, the number of places (d) after the decimal point that are to be printed.

10.1 Basic Field Specifications

These basic field specifications are

Iw, Ew.d and Fw.d,

with the successive specifications separated by commas. Thus the statement FORMAT (I2, E12.4, F10.4) might give the line

```
29 -0.9321E 02   -0.0076
```

As in this example, the field widths may be made greater than necessary so as to provide spacing blanks between numbers. In this case, there is 1 blank following the 29, 1 blank after the E (automatically supplied except in cases of (a) a negative 2 digit exponent when a minus sign will appear, and (b) a 3 digit exponent, the E being changed to N if negative), and 3 blanks after the 02. Within each field the printed output will always appear in the right-most positions.

It may be desired to print n successive fields within one record with the same specification. This may be specified by giving n before E, F or I. Thus, the statement FORMAT (I2, 3E12.4) might give

```
29 -0.9321E 02 -0.7580E-02  0.5536E 00
```

10.2 Alphanumerical Fields

S2 provides two ways by which alphanumerical data may be read or written; the specifications for this are Aw and wH. Both result in storing the alphanumerical information in BCD form. The difference is that information handled with the A specification is given a variable or array name and hence can be referred to by means of this name for processing. Information handled with the H specification is not given a name and may not be manipulated in storage in any way.

The specification Aw causes w characters to be read into, or written from, a variable or array name.

On input nAw will store the next n successive fields of w characters as BCD information. If w is greater than 8, only the 8 right-most characters will be significant. If w is less than 8, the characters will be right adjusted and the word filled out with blanks.

On output nAw will transmit n successive fields of w characters from storage as BCD characters without conversion. If w exceeds 8 then only 8 characters will be transmitted preceded by w-8 blanks. If w is less than 8 then the w right-most characters will be transmitted.

The specification wH is followed in the FORMAT statement by w alphanumerical characters. For example

28H THIS IS ALPHANUMERICAL DATA

Note that blanks are alphanumerical characters and must be included in the count w.

The effect of wH depends on whether it is used with input or output.

(1) Input

w characters are extracted from the input record and replace the w characters of the specification.

(2) Output

The w characters following the specification, or the characters which replaced them, are written as part of the output record.

Example: The statement FORMAT (1Hb, 3HXY=, F8.3, A8) might produce the following lines:-

```
XY b-93.210bbbbbbb
XY 9999.999bbOVFLOW
XY bb28.768bbbbbbb
```

(b is used to indicate blank characters: the first H specification is used for carriage control, see Section 10.9).

This example assumes that there are steps in the source program which read the data "OVFLOW", store this data in the word to be printed in the format A8 when overflow occurs, and stores six blanks in the word when overflow does not occur.

10.3 Blank Fields

Blank characters may be provided in an output record and characters of an input record may be skipped by means of the specification wX, where $0 \leq w \leq 132$ (w is the number of blanks provided or characters skipped). When the specification is used with an input record, W characters are considered to be blank regardless of what they actually are, and are skipped over.

10.4 Scale Factor

To permit more general use of F-type conversion, a scale factor followed by the letter P may precede the specification. The scale is such that

$$\text{Printer number} = \text{Internal number} \times 10^{\text{scale factor}}.$$

Thus, the statement FORMAT (I3, 1P3F11.3) used with the data of Section 10.1, would give

```
29   -932.096     -0.076      5.536
```

whereas FORMAT (I3, - 1P3F11.3) would give

```
29     -9.321     -0.001      0.055
```

(Note that the leading zero of the 3 digit number has been suppressed.) A positive scale factor may also be used with E-type conversion to increase the number and decrease the exponent. Thus, FORMAT (I3, 1P3E12.4) would produce with the same data

```
29 -9.3210E 01 -7.5804E-03  5.5361E-01
```

The scale factor is assumed to be zero if no other value has been given. However, once a value has been given, it will hold for all E- and F-type conversions following the scale factor within the same FORMAT statement. This applies to both single-record and multiple-record formats. Once a scale factor has been given, a scale factor of zero in the same FORMAT statement must be specified by 0P. Scale factors have no effect on I-conversion.

10.5 Repetition of Groups

A limited parenthetical expression is permitted in order to enable repetition of data fields according to certain format specifications within a longer FORMAT statement specification. Thus, FORMAT (2(F10.4, E10.2), I4) is equivalent to FORMAT (F10.4, E10.2, F10.4, E10.2, I4).

10.6 Multiple-Record FORMATs

To deal with more than one line of print, a FORMAT specification may have several different one-line formats, separated by a slash (/) to indicate the beginning of a new line. Thus, FORMAT (3F9.2, 2F10.4/8E14.6) would specify a multi-line print in which lines 1, 3, 5, ... have FORMAT (3F9.2, 2F10.4), and lines 2, 4, 6, ... have FORMAT (8E14.6).

If a multiple-line format is required such that the first two lines will be printed according to a special format and all remaining lines according to another format, the last line-specification should be enclosed in a second pair of parentheses; e.g., FORMAT (I2, 3E12.4/2F10.6, 3F9.4/(10F12.4)). If data items remain to be transmitted after the FORMAT specification has been completely "used", the FORMAT repeats from the last open parenthesis.

These examples show that both the slash and the closing parenthesis of the FORMAT statement indicate a termination of a record.

Blank lines may be introduced into a multi-line FORMAT statement by listing consecutive slashes. N + 1 consecutive slashes produce N blank lines.

10.7 FORMAT and INPUT/OUTPUT Statement Lists

The FORMAT statement indicates, among other things, the maximum size of each record. It must be remembered that the FORMAT statement is used with the list of some particular input/output statement, except when a FORMAT statement consists entirely of alphanumerical fields. In all other cases, control in the object program switches back and forth between the list (which specifies whether data remains to be transmitted) and the FORMAT statement (which gives the specifications for transmission of that data).

10.8 Ending a FORMAT Statement

During input/output of data, the object program scans the FORMAT statement to which the INPUT/OUTPUT statement refers. When a specification for a numerical field is found and items remain to be transmitted, input/output takes place according to the specification and scanning of the FORMAT statement resumes. If no items remain, transmission ceases and execution of the INPUT/OUTPUT statement is terminated.

10.9 Carriage Control

WRITE OUTPUT TAPE and PRINT statements prepare a decimal tape which can later be used for off-line printed output. The first character of each BCD record controls the spacing of the off-line printer and that character is not printed. The control characters and their effects are

Blank	Single space before printing.
0	Double space before printing.
1	New page before printing.

Any other character will also cause a single space before printing. Thus, a FORMAT specification for WRITE OUTPUT TAPE or PRINT will usually begin 1H followed by the appropriate control character, unless as in the examples of Section 10.4 etc., a blank is provided by zero-suppression.

10.10 Data Input to the Object Program

Decimal input data to be read by means of a READ or READ INPUT TAPE when the program is executed, must be in essentially the same format as given in the previous examples. Thus, a card to be read according to FORMAT (I3, E12.4, F10.4) might be punched

```
29 -0.9321E 02    -0.0076
```

Within each field, all information must appear at the extreme right. Plus signs may be omitted or indicated by a blank or +. Minus signs are punched with an 11-punch. Blanks should not appear between the digits of the number being read.

To permit economy in punching, certain relaxations in input data format are permitted.

(1) Numbers of E-type conversion need not have 4 columns devoted to the exponent field. The start of the exponent field must be marked by an E. Thus, E2, E02, E 02 and E+02 are all permissible exponent fields.

(2) Numbers for E- or F-type conversion need not have their decimal point punched. If it is not punched, the FORMAT specification will supply it; for example, the number -09321E2 with the specification E12.4 will be treated as though the decimal point had been punched between the 0 and the 9. If the decimal point is punched in the card, its position over-rides the position indicated in the FORMAT specification.

11. SPECIFICATION STATEMENTS

These are the DIMENSION, COMMON and EQUIVALENCE statements which control the allocation of storage used by the compiled program.

In contrast to "Standard FORTRAN", S2 allows the allocation of storage to be determined when the compiled program is loaded, so that allocation of storage may be varied from one run to the next. This permits economy in the use of storage because the number of storage cells for each array can be chosen for the requirements of an individual run and not for any possible run. The allocation of storage for arrays whose size varies from one run to another requires that DIMENSION statements be written with variable dimensions; such variables are called array parameters and are set by a special routine, whose name is PRELUDE, at loading time. PRELUDE is written in S2 language and can be used to set up other parameters as well as array parameters. It is important to note however, that array parameters should not be altered in the course of a program.

11.1 DIMENSION

DIMENSION V, V, V, ... V is the general form and

DIMENSION A(10), B(J), C(5, K, 6) is an example.

The V are in the form $A(d_1, d_2, d_3)$, where A is the name of an array and d_1, d_2, d_3 are the dimensions, which are either

(1) INTEGER constants, n,

(2) INTEGER variables which are COMMON, or

(3) INTEGER variables which are parameters (explicitly stated) of the SUBROUTINE or FUNCTION.

The dimensions are the maximum values that the corresponding subscripts should attain. There is no fixed limit on the number of dimensions that an array may have. Up to three-dimensional arrays are always acceptable; the limit is on the number of symbols in a subscripted element referring to the array, and is given in Section 2.4.6.

11.2 COMMON

The COMMON statement, where general form is

COMMON V, V, ... V and example is

COMMON A, B, C

has two functions. Firstly to declare that the variables concerned (V) are used not only by the routine being compiled but by others as well. Secondly, it establishes an ordering of the COMMON list.

Neglecting for the moment the effect of EQUIVALENCE statements, the COMMON statements cause single storage cells to be allocated from location 145 octal (or 101 decimal) to each variable declared COMMON, in order of writing. Duplication of the COMMON declaration about a variable causes the declaration after the first to be ignored, e.g.,

```
COMMON  A, B, C, D
COMMON  D, A, F
```

causes the following allocation:-

address (octal)	145	146	147	150	151
variable	A	B	C	D	F

Since communication between two routines is established by the COMMON list rather than by the names used, care must be taken to ensure that if the same name is used for the same variable in two different routines within the same program, then they are allocated the same address in the COMMON list.

11.3 EQUIVALENCE

The EQUIVALENCE statement in S2 has two functions which intermingle:-

(1) To provide a synonym.

(2) To overlay arrays.

The general form is

EQUIVALENCE $(L_1, R_1), (L_2, R_2), \ldots, (L, R)$ and an example is

EQUIVALENCE (A, B), (C, D(K))

We enumerate the various combinations of L and R:-

(a) EQUIVALENCE (A, B), (X, I), (J, W), where all are variable names means that A uses the storage allocated to B, but retains its own mode; X uses the storage for I and remains REAL; J uses the storage for W and remains INTEGER.

(b) EQUIVALENCE (A, B(I)), (I, C(15)), where B(I), C(15) are subscripted variables and A, I are ordinary variables. The left variable uses the same storage as the right array element.

(c) EQUIVALENCE (A, B(I)), where A, B are array names and the right variable is subscripted by one INTEGER constant or array parameter. The effect is to make the first element of the array A coincide with the $(I+1)^{th}$ element of the array B.

(d) EQUIVALENCE (A, B), where A, B are arrays, makes the array A start at the same point as the array B.

If an EQUIVALENCE (A, B) is made between an array name and a variable name (either way) then a diagnostic message appears. This indicates that the program may produce unexpected results if run.

In S2 EQUIVALENCES, the left variable is defined in terms of the right variable, and in each EQUIVALENCE only a pair of terms appears. This is different from FORTRAN.

EQUIVALENCES of type (a) and (d) may be chained. Thus the EQUIVALENCES (A, B), (B, C), (C, D) will cause A, B, C to use the storage allocated to D. There is an exceptional case, for which see the effect of EQUIVALENCES on the COMMON list. The compiler will recognise redundant EQUIVALENCE statements and print a diagnostic message. The cases are:-

(a) (A, B), (B, C), (C, A), such a circular definition is treated by ignoring the last EQUIVALENCE (C, A) which leads to the circularity.

(b) (A, B), (B, C), (A, C). The last EQUIVALENCE is redundant and is ignored after diagnosis.

11.4 The Effect of EQUIVALENCES on the COMMON List

In EQUIVALENCE (L, R) we have the following cases:-

(1) L, R COMMON; L does not get a separate entry in the COMMON list but uses the entry for R. The COMMON list is constructed as if L had not been declared COMMON.

(2) Only R COMMON; L uses the space allocated to R. Thus this case does not affect the formation of the COMMON list.

(3) Only L COMMON; L takes its place in the formation of the COMMON list, but the routine is compiled using the storage of R for L.

When cases (1), (3) occur a diagnostic mark is made in the listing of the COMMON list at the position that the L variable occupies (case 3) or would have occupied but for the EQUIVALENCE (case 1).

The preceding description is for unchained EQUIVALENCES. For chained EQUIVALENCES the same rules apply if R is the "ultimate EQUIVALENT", the symbol at the end of the chain. In the example EQUIVALENCE (A, B), (B, C) this would be C and the rules would apply as if EQUIVALENCE (A, C), (B, C) had been written.

11.5 The Effect of Using FUNCTION or SUBROUTINE Parameters in Specification Statements

The COMMON list is formed as if there were no variables which were SUBROUTINE parameters. The SUBROUTINE parameters, however, do not use any storage within the COMMON list but use private storage within the routine being compiled. This situation is diagnosed.

Similarly if a SUBROUTINE parameter appears on the left of an EQUIVALENCE, it has no effect on the other variables involved, but again it uses private storage. Thus if S is a SUBROUTINE parameter appearing in the EQUIVALENCE (A, S), (S, B) the effect is as if only (A, B) had been written but a diagnostic message appears.

Only if S is an ultimate EQUIVALENT will any other variables share its storage. For example in the EQUIVALENCES (A, S), (B, C), (C, S), S is the ultimate EQUIVALENT of A, B and C which share a private storage cell with S.

11.6 Standard Practice with Specification Statements

Since this is a reference manual much of the information it contains explains the more out-of-the way properties of the S2 language and the S2 compiler. That certain effects are stated to occur is not an invitation to exploit them. Many of the effects of the compiler are there as aids to the programmer who may make a trivial error which the compiler might be able to correct. Most of the effects of mixtures of declarations in the preceding sections describe what the compiler does to retrieve a programmer's slip; they are not put in to make the language more slipshod.

The following rule is a guide to the writing of COMMON, EQUIVALENCE, SUBROUTINE and FUNCTION statements:-

Avoid making the following types of declaration about the same variable:-

(a) COMMON, COMMON.

(b) COMMON, EQUIVALENCE with variable on the left (Left-EQUIVALENT).

(c) COMMON, SUBROUTINE parameter.

(d) COMMON, FUNCTION parameter.

(e) SUBROUTINE parameter, Left-EQUIVALENT.

(f) FUNCTION parameter, Left-EQUIVALENT.

12. THE SUBROUTINE PRELUDE

Whenever a COMMON declaration is made, one store is allocated to each item (in order) in the COMMON list, which is a fixed part of storage (location $101_{(10)}$ upwards) as explained in Section 11.2.

If the item is a variable this store is intended to contain its value.

If the item is an array name, this store is intended to contain the base address of the array, and the programmer has to define this base address. (The base address of an array, A, in this context is the address of the location immediately before that which contains the first element A(1) of the array.) This definition must be done in

SUBROUTINE PRELUDE

which must be provided for every program, even if no COMMON variables are used.

This routine is written as any other SUBROUTINE and compiled in the same way.

The differences between it and other SUBROUTINES are (apart from its function of defining the storage of variables):-

(a) During an execution job it must be the first routine of the program to be loaded and it will be obeyed immediately. (The main program and the other SUBROUTINES are loaded afterwards and overwrite PRELUDE.)

(b) It can call and use any of the routines in the S2 library but cannot refer to any program routines.

(c) The statement RETURN will transfer control to the first executable statement of the main program.

Storage in STRETCH is allocated by S2 as follows:-

The COMMON list starts at location $101_{(10)}$ (upwards). Above the COMMON list will be loaded all routines (except PRELUDE with their private variables. The COMMON statements in PRELUDE must therefore contain at least as many items as any of the other routines.

The library routines and other important material are loaded in a region above $80000_{(10)}$. The region suitable for storing the COMMON arrays is therefore from 79 999 downwards. One has considerable freedom in the allocation as long as storage does not overlap sections of program, M.C.P. etc. Arrays do not need to be stored in the same order as their names appear in COMMON declarations.

Base addresses of all arrays in COMMON should be loaded into the store in the COMMON list which corresponds to the array name. This is done by such statements as

```
A = 79000
B = A - 500
```

where A and B are array names in COMMON. This causes stores from 79001 upwards to be allocated to the array A, those from 78501 upwards to B. Dimensions, which define storage allocations, and which have been left as parametric in other routines must be defined in PRELUDE, usually by reading them as data during execution. This does not apply when such dimensions and arrays are arguments of SUBROUTINES, since the calling routine defines the actual storage space.

Example: Let a routine SUBR1 contain the following DIMENSION and COMMON statements:-

```
DIMENSION A(I, J), B(100), C(I, N)
COMMON P, A, B, C, I, J, N, X, U, DUMMY, Z
```

The following PRELUDE would fit

```
    SUBROUTINE PRELUDE
    COMMON Q, A, B, F, I, J, N, X, Y, W, Z, V, M          ...(1)
    L = 79999
    READ 100, I, J, N, M
100 FORMAT (4I6)
    A = L - I*J
    B = A - 100
    F = B - I*N                                           ...(2)
    W = F - M                                             ...(3)
    PRINT 101, A, B, F, W
101 FORMAT (1H1, 4I6)                                     ...(4)
    RETURN
    END
```

(1) The scalars V and M are mentioned in PRELUDE, but are not used in SUBRI. This is allowed, but it should never be the other way round.

(2) Variables in COMMON do not need to be called by the same name in different routines. They will occupy the same stores if their names correspond to the same position in the COMMON list.

(3) The array W of the PRELUDE is not referred to in SUBRI. In such a case the dimension statement is optional, but the corresponding item (DUMMY) in the COMMON list of SUBRI cannot be omitted, as Z would become displaced.

(4) This format will cause the base addresses of the arrays to be printed as INTEGERS.

In general one does not wish to have discrepancies between the COMMON and DIMENSION statements of different routines, but in some cases the features mentioned can be useful.

Warning: The loader economises on the coding by computing, at load time, as many of the addresses of variables as possible throughout the program.

Example: If N is in the COMMON List, and had been defined during PRELUDE, the address of A(N) will be computed before execution starts, wherever it happens to occur. A(N) will then refer to the same store throughout the job. One should therefore not borrow quantities from the COMMON list for use as working variables, such as DO loop indices.

Similarly dimensions of variables must be defined at load time and should not be changed during execution.

Up to 2000 stores are reserved by the computer for scalar variables and base addresses of arrays, and it loads these stores with a large number (0.5×2^{1024}). This corresponds to the machine's definition of infinity and calculations on these numbers usually yield meaningless results. One should never assume that stores contain 0, or any other definable quantity, when execution starts.

APPENDIX A

RULES FOR XFN AND XFP ARITHMETIC

Let XFN, XFP stand for numbers whose exponents are less than - 1023, greater than + 1023 respectively and let N stand for a number whose exponent is less than 1024 in magnitude. Then the following equations show the effects of using these numbers in arithmetic:-

A1. ADDITION (SUBTRACTION)

N+N = N*   A result N* may go out of range if the result is
N+XFN = N*   too large or too small.
N+XFP = XFP

XFP+XFP = XFP
XFP+XFN = XFP

XFN+XFN = XFN

A2. MULTIPLICATION

N*N = N*
N*XFN = XFN
N*XFP = XFP

XFP*XFP = XFP
XFP*XFN = XFP

XFN*XFN = XFN

A3. DIVISION

N/N = N*
N/XFN = XFP
N/XFP = XFN

XFP/XFP = XFP
XFP/XFN = XFP

XFN/XFN = XFP

For ease of memory, note that the form of these equations is like the similar equations for operations on infinity (XFP), zero (XFN) and finite (N) numbers, with the rider that any indeterminate result is taken to be infinity. <u>This does not say that an XFN number is zero; in general it is not</u>.

APPENDIX B

THE MYSTERIOUS ZERO

There are more than 4000 different patterns of bits within STRETCH that can be called zero, namely all those floating point numbers with zero fraction but with different exponents. The various types generated within S2 programs are:-

(a) Fixed point zero, the exponent is 38. This number is generated by any INTEGER arithmetic, by writing O as an INTEGER constant in the program, by reading, as data, a zero INTEGER in I mode or by operation XFIXF(X), provided X has an exponent of 38 or less, and is zero.

(b) Floating point zero, where the exponent is - 1023. This is generated by writing a floating point zero constant in the source program, or by reading a floating point zero in E or F mode.

(c) Order of magnitude zeros. These are zeros arising by operations on REALS or during mixed arithmetic. They may have almost any exponent according to how they arise. The following are important cases:-

FLOATF(O)	the exponent is - 2047. This number is therefore an XFN number
X-X	the exponent of this zero is the same as that of X, provided X is not in the XFN or XFP ranges, in which case the operation X-X produces X, not zero, unless X itself happens to be zero
X*Y	this will produce a zero provided one of X or Y (or both) is zero. If the factors are both REAL and in the normal range, then the exponent of the zero result will be the sum of the exponents of X and Y minus 38. If only one of the factors is REAL the result will be FLOATF(O)

The use of a zero in an IF statement will always give the expected result provided the result tested has a zero fraction. The only important case where the expected zero result is not always realized is when the difference is taken between two numbers that would be the same in ideal calculation but are outside the normal range.

The order of magnitude zero does not always behave as an ideal zero when added or subtracted. For example if m is the exponent of an order of magnitude zero, and n the exponent of X is less than m, then the effect of the addition is to replace the m-n least significant bits of X by zero bits. In extreme cases of m-n greater than 47, all significant bits of X are lost.

One way of viewing order of magnitude zeros which may be helpful is to think of them as arising by cancellation during subtraction of two ideal numbers not necessarily identical whose most significant 48 bits coincide. The resulting zero lacks precision to exactly the same degree as the original numbers.

Thus, a good working rule with REALs is to assume that the calculation is always approximate unless the numbers in the calculation can all be expressed as

$$\pm\ N*2^{m},$$

where N is an INTEGER less than 2^{48} and m is an INTEGER chosen once for all.

APPENDIX C

PUNCHING CONVENTIONS

C1. PUNCHING A SOURCE PROGRAM

Each statement of an S2 FORTRAN source program is punched into a separate card. However, if a statement is too long to fit on one card, it can be continued on as many as nine "continuation cards". The order of the source statements is governed solely by the order of the statement cards.

Cards which contain a "C" in column 1 are not processed by the S2 compiler. Such cards may be used to carry comments which will appear when the source program deck is listed.

Numbers may be punched in columns 1 - 5 of the initial card of a statement. When such a number appears in these columns, it becomes the statement label of the statement. These statement labels permit cross references within a source program.

Column 6 of the initial card of a statement must be left blank. Continuation cards (other than for comments), on the other hand, must have column 6 punched with some character other than zero, and should be punched with numbers from 1 through 9. Continuation cards for comments need not be punched in column 6; only the "C" in column 1 is necessary.

The statements themselves are punched in columns 7 - 72, both on initial and continuation cards. Thus, a statement may consist of not more than 660 characters (i.e., 10 cards). A table of the admissible characters for FORTRAN is given in Appendix M. Blank characters, except in column 6 and in certain alphanumerical fields in FORMAT statements are simply ignored by FORTRAN, and may be used freely to improve the readability of the source program listing.

Columns 73 - 80 are not processed by the compiler and may, therefore be punched with any desired identifying information.

C2. ARRANGEMENT OF DECLARATIONS

The declaratory statements should be placed (in any order) at the beginning of the deck. These are the FUNCTION, SUBROUTINE, DIMENSION, COMMON, EQUIVALENCE, INTEGER, REAL, statements.

APPENDIX D

DECK MAKE-UP

D1. TYPE, GO

For jobs of this type, control cards are:-

```
B  JOB,                               JOB Card with JOB number.
B  TYPE, GO                           Type Card
B  LIM,                               Limit Card
B  IOD,               )
B  REEL,              ) As required.
   Set E dump 1 to 6 )
   FETCH 1, 2 and 3
   Prelude binary deck
   Prelude data if any
   Binary decks produced by S2
   END 1
   END 2
   Data, if any
```

D2. TYPE, COMPILE, S2, OPTION

Option is PRINT if a mnemonic dump of the object program is required. A null field suppresses the dump. For jobs of this type, each S2-coded deck must be preceded by a card having (S2) starting in or before cc.10. STRAP decks must be preceded by (STRAP) cards. Binary decks should not be present. A (QUIT) card should normally be put at the end of the job. If there is no (QUIT) card, a diagnostic will be printed, but the processor will assume that one was intended and no error will result.

```
B  JOB,
B  TYPE, COMPILE, S2
   (S2)
    S2-coded routine, terminated     )
    by END statement                 )
      .                              )
      .                              )
      .                              ) in any order
      .                              )
      .                              )
   (STRAP)                           )
   STRAP-coded routine, terminated  )
    by END statement                 )
      .                              )
      .                              )
      .                              )
      .                              )
      .                              )
   (QUIT)  (may be omitted)          )
```

D3. TYPE, COMPILGO, S2, OPTION

Option is as described in Section D2. Jobs of this type have the same control cards as TYPE, COMPILE, S2, but binary decks may be included for which no control cards are necessary. The (QUIT) card should normally follow the program deck. It is mandatory if data follows.

```
B  JOB,
B  TYPE, COMPILGO, S2   (NB: NO LIM-card)
B  IOD                 )
B  REEL                ) As required
   Set E dump 1 to 6   )
   FETCH 1, 2 and 3
   PRELUDE binary or          )
     symbolic deck            ) Symbolic decks must
   Binary or symbolic         ) be preceded by the
     decks for other          ) appropriate control-card-
     routines, in any         ) (S2), or (STRAP)
     order
   END 1
   END 2
   (QUIT) (may be omitted if no data follows)
   PRELUDE data, if any
   Data, if any
```

APPENDIX E

OUTPUT FROM THE S2 COMPILER

Printed output for S2 compilations consists of a listing of the source program, and a map of the COMMON list. Diagnostic messages generally follow immediately after the statement to which they refer. (All detectable error conditions cause a diagnostic in S2, though some types of error cause compilation to be terminated). Timing and identification messages are printed at the end of each compilation, together with an optional mnemonic dump of the object program.

Punched output comprises the programmer's JOB card, followed by the binary decks resulting from compilations or assemblies. These binary decks are separated by T-cards, and a T-card also precedes the first binary deck and follows the last binary deck; the deck as a whole is therefore intact and ready for use after removing the JOB card and any trailing blanks/logger cards. (In the output from a STRAP-coded routine, a TYPE card and LIM card are provided by STRAP.)

In order to check that the deck of binary cards is correct or to sort them when they have become shuffled, one can make use of the following properties:-

The time clock reading at the beginning of the compilation is punched in column 2 of all binary cards of the routine. This column is therefore identical on all cards of the same routine and different for cards from different routines.

The cards of each routine are numbered in binary from 1 onwards. This number is punched in column 3.

All cards have column 1 punched in one of the following ways:-

(a) Rows 7, 8, 9 (origin cards, the first card and others).

(b) Rows 7, 9 (flow cards).

(c) Rows 6, 7, 9 (branch card, the last card only).

APPENDIX F

S2 DIAGNOSTICS

The following diagnostic messages are produced by the S2 compiler. Those marked * stop the punching of binary cards:-

Programmer's Errors

```
 Parameter has changed
*Variable implied constant not in COMMON
*Invalid subscript
 A SUBROUTINE parameter is COMMON
 A parameter is left-equivalent
*Duplicated parameter
*Variable DIMENSION is private
 Redundant EQUIVALENCE
 Duplicated COMMON entry
 Incorrectly nested DO
 DO closure up the creek
*DO nesting incomplete at end
 Branch into DO range to this label
*FUNCTION name unused
 Label used but not set
 Label set but not used
 Illegal branch into DO range
*Array argument is floating point
*Illegal DIMENSION in 1 - D array
 DIMENSION variable used as array name
 Array name used as DIMENSION variable
*EQUIVALENCE expression invalid
 Unused parameter
 EQUIVALENCE between array and scalar
 DO variable is SUBROUTINE parameter
*Subscripted variable too long
*20 or more terms in expansion of variable.
 SENSE LIGHT No. out of range
*No path to next instruction
*FUNCTION has too many arguments
*No path to this instruction
*Illegal DIMENSION in N - D array
 Array declared more than once
 Duplicated label
*Illegal instruction at end of DO
*DO argument is an expression
*Declaration not at front of routine
*FUNCTION on left of equals
*A constant appears before equals
```

*Error in FORMAT label
*Error in FORMAT statement
*An INTEGER is followed by a letter
Illegal FORMAT , has been inserted
*A fraction is followed by a letter
*Wrong exponent in constant
*A letter follows
*(follows a constant
*A '.' follows a name
*A '.' follows a constant
*A '.' follows a '.'
* '.' appears initially
*A (appears initially
*A statement ends incorrectly
** follows another operator
*/ follows another operator
*- follows another operator
*+ follows another operator
*, follows another operator
*) follows another operator
*= follows another operator
*More) than (somewhere
*More (than) somewhere
*A statement begins incorrectly
*Illegal statement
More than 7 characters used in a name
*A blank statement was found
*A $ appears illegally
*Error in DO statement
*Error in GO TO statement
*Error in READ statement
*Error in computed GO TO statement
*Error in assigned GO TO statement
*Error in COMMON statement
*Error in DIMENSION statement
*Error in PRINT statement
*Error in PUNCH statement
*Error in EQUIVALENCE statement
*Error in READ INPUT TAPE statement
*Error in WRITE OUTPUT TAPE statement
*Error in READ DISK statement
*Error in WRITE DISK statement
*Error in READ TAPE statement
*Error in WRITE TAPE statement
*Error in CONTINUE statement
*Error in RETURN statement
*Error in IF statement
*Error in ASSIGN statement

*Error in IF (SENSE LIGHT) statement
*Error in SENSE LIGHT statement
*Error in STOP statement
*Error in ARITHMETIC expression
*Error in BACKSPACE statement
*Error in REWIND statement
*Error in END FILE statement
*Dictionary full
*Dictionary look up error
Duplicated declaration
Nested ranges with same index
Illegal DO statement
*DO variable is RELOCATOR constant
*The DO index is not a variable
No RETURN statement
*FUNCTION without parameters
*Dictionary nearly full
*No executable instruction in routine

The following diagnostic messages are the result of a machine or compiler error and prohibit the punching of binary cards:-

Illegal operator was found in DECODING
Machine or compiler error in GE
Machine or compiler error in GBH
Machine or compiler error in GD
Machine or compiler error in GR
Machine or compiler error in GA
Machine or compiler error in GA1
Machine or compiler error in GA9
Machine or compiler error in GA11
Illegal operator in Macro level
Macro Mode incompatibility
Machine or compiler error at INSTG3
Continual UK's on Disk - Compilations Terminated
EPGK on Disk - Compilations Terminated
UK without EOP on Disk - Compilations Terminated
Error in stacking algorithm
Index switch failure
Compiler error - invalid level
Compiler error - negative level
Unable to fix S2 to carry on
Error return from GET
Undefined switching position
Compiler error. Bushy index expression
Continual UK's on Disk in S2

The following diagnostic messages are the result of overflow of tables within the compiler and stop the punching of binary cards.

Object program overflows loading region
Too many work space cells required
Data cells required exceed space available
Private Prelude overflows data region
Too many DIMENSIONED variables
More than 100 EQUIVALENCES
More than 100 SUBROUTINE parameters
Dictionary nearly full
Underflow in expanding EQUIVALENCES
Index tree is full
Tree dictionary is full
I - string underflow
Too many nested DO ranges
Loading region full
Principal node list is full

APPENDIX G

ERROR INDICATIONS DURING EXECUTION

Diagnostic messages produced by the Library Package. (All preceded by "JOB REJECTED".)

G1. THE LOADER

(a) COMMON LIST TOO LONG

2000 words are presently allowed.

(b) PROGRAM OVERLAPS LOADING REGION

Too much program in store.

(c) A ROUTINE COMMON LIST IS TOO LONG

This means that COMMON list of a routine is longer than that of the PRELUDE.

(d) ROUTINE X HAS NOT BEEN LOADED

A program contains a reference to a SUBROUTINE, x, which has not been supplied.

(e) NO MAIN ROUTINE HAS BEEN LOADED

(f) TOO MANY LINKS

More than 20.

(g) TOO MANY ROUTINES HAVE BEEN LOADED OR CALLED FOR

You can have up to about 200 of your routines per link.

(h) NO PROGRAM PRELUDE LOADED

(i) NEXT LINK CONTAINS NO ROUTINES

G2. CHAIN, SUBROUTINE CALLING DURING EXECUTION

(a) LINK NO i DOES NOT EXIST

(b) ROUTINE CALLED AT LOC i IS UNKNOWN

This occurs when a routine is known to have been loaded but cannot be found. Probably caused by failure of LINK calling.

(c) DUMP TAPE ERROR

DISK ERROR

Caused by irrecoverable errors in forming links on DISK or TAPE.

G3. OTHERS

(a) END OF TAPE i REACHED $15 = e

Writing tape.

(b) TAPE i ERROR $15 = e

Writing tape.

(c) TAPE MARK ON TAPE i REACHED $15 = e

Reading tape.

(d) DATA ERROR $15 = e

(e) FORMAT ERROR $15 = e

(f) NO MORE DATA $15 = e

(g) I/O UNIT i ERROR $15 = e

In RELIBR, WRLIBR etc.: (a) shows irrecoverable error, or (b) use of IOD number greater than 32.

(i) AN ARC CANNOT BE LOCATED

(j) TAPE i, WRITE, EPGK error

READ UK

EE

EOP

These are illegal conditions.

(k) DISK ERROR

(l) POWER ROUTINE ERROR $15 = e

(m) AT LOCATION *e* THE SQRT OF "X" WAS ASKED FOR

(n) SIN/COS ERROR $15 = e ARGUMENT GREATER THAN 2**42

(o) LOG ROUTINE ERROR $15 = e ARGUMENT = "X"

NOTES:

(1) e is the location where the SUBROUTINE was called from the problem program.

(2) i stands for a tape or disc IOD number.

(3) "X" stands for the actual value of an argument.

APPENDIX H

CHAIN JOBS

The relocation system for S2 programs can be used to form chain jobs, i.e., a program of several sections, called links, where each section is brought to core storage from disk as required. This allows programs which are too big to be held in core store in one piece to be segmented.

H1. THE FORMATION OF A CHAIN

A chain is formed on disk file number 19 by loading the following type of deck:-

(a) PRELUDE (for whole chain).

(b) Link No. 1 program.

(c) Link Control Cards (a standard set of cards).

(d) Link No. 2 program.

(e) Link Control Cards.

...Other links and control cards.

(p) Initiating Program (Optional).

(q) Program Start Cards.

The links are loaded and are numbered in the order of loading. The number of links must be 20 or less. When loading is complete, control goes to the initiating program or to link 1 if the initiating program is omitted.

Links are entered from one another by the statement

CALL CHAIN (NEXT)

where the INTEGER value NEXT specifies which link is to be entered next. NEXT is a constant or INTEGER variable. Each link is written as a main program and when called will begin operating at the first statement of the main program.

H2. DUMPING A CHAIN

To allow the formation of tapes containing a program chain the following statement can be used:-

CALL CDUMP (A, B)

This causes the chain of programs on disc 19 to be transmitted to tape 20 together with the region of core storage from the address A to the address B inclusive, and the index registers. A tape mark is then written on tape 20.

The job is then ended without control returning to the program.

H3. REVIVING A CHAIN

If tape 20 contains a dumped chain, then

CALL RELOAD

will revive the job by transferring the chain to disc 19, refreshing the core store and index registers as required by the CDUMP statement that formed the dumped chain. Control then goes to the next statement after the CDUMP statement that caused the chain to be dumped.

Note that RELOAD can restart a link, not an initiating program (which is not considered part of the chain). Thus, CDUMP cannot be used in the initiating program.

The revival of a chain may be done in a variety of ways of which the following may the most useful:-

A* Revival of a chain which had not run. Here it is envisaged that the CDUMP mechanism has been used to form a program tape. The RELOAD statement can then be the last statement of a PRELUDE which revives the chain. This revival PRELUDE must recreate the storage structure as constructed by the original PRELUDE of the chain, but may alter parameters not concerned with storage allocation. If no parameters alter, see D.

B Revival of a chain in process may be made by obeying the statement CALL RELOAD (which can be written as a single statement PRELUDE), provided that the CDUMP statement forming the chain tape has specified the dumping of a segment of core storage containing the COMMON list and any other data that is to be refreshed.

C As A but reviving the COMMON list as originally dumped.

*If a chain is revived from a PRELUDE, Program-start-cards will be unnecessary (in the restart program).

D A PRELUDE followed by the program start cards will also call RELOAD - this saves time writing another PRELUDE with CALL RELOAD in it.

H4. The program must give IOD cards for his use of disk 19 and tape 20 and in particular must specify how much disk space is required.

Each link requires a number of disk arcs to hold the link program plus one more arc to hold the link SUBROUTINE symbol table. The extent of a link can be discovered from the storage map printed by the relocator.

APPENDIX I

SPECIAL INPUT/OUTPUT SUBROUTINES FOR TAPE AND DISK (UNBUFFERED)

The INPUT/OUTPUT routines used by the FORTRAN INPUT/OUTPUT statements do not allow execution of succeeding steps of the calculation until transmission is complete, thus removing from the programmer any worry about timing but at the cost of a delay. The binary INPUT/OUTPUT routines use buffering which may reduce the delays if the data transmission rate is low.

For the programmer who is prepared to look after timing problems a set of SUBROUTINES is provided, which initiate data transmission. Another FUNCTION sub-program is provided by which the progress of transmission may be sensed.

I1. TRANSMISSION

Transmission is initiated by the calls:-

(1) CALL WRLIBRF (iod, location, length), and

(2) CALL RELIBRF (iod, location, length),

for transmission from and to core storage, respectively, where

iod = tape or disk number (variable or constant),

location = name of the first word to be transmitted,

length = number of words to be transmitted.

Note that the transmission of an array requires the specification of the first element location which is not the same as the name of the array as a whole. Thus if A is an array to be transmitted then write

CALL RE LIBRF (I(J), A(1), 10)

not CALL RELIBR (I(J), A, 10)

I2. LOCATION OF DISK ARCS

CALL LOLIBR (iod, arc number)

This is used to locate the disk heads so that transmission may start at the required "arc number" on logical disk "iod". In transmission involving consecutive arcs on the disk, it is necessary to locate only the first arc in the sequence.

I3. SAMPLING

To allow the progress of a transmission to be sensed a function CWLIBRF (iod, location) is provided, where

iod	=	tape or disk number
location	=	address in core storage

CWLIBRF (iod, location) is positive if a higher address than "location" has been used in transmission; otherwise it is negative. This function should be used rarely, and only when a long transmission is in progress since the time taken to sample is quite long.

During the sampling time at least 25 words will have been transmitted to or from the disk or 3 words to or from high density tape. Note: The sampling process of this section and the check process of the next section must not be employed together on the same unit. Use one method or the other, but not both for the same transmission.

Example: A transmission of 5 000 words of an array starting at A(I + 5) to tape 4 is started and further computation must be delayed until the words A(I + 5) to A(I + 1 000) have been transmitted.

```
      CALL WRLIBRF (4, A(I + 5), 5 000)
1     IF (CWLIBRF (4, A(I + 1 000)) 1, 1, 2
2     now computation can proceed.
```

I4. END OF TRANSMISSION

More usually, the programmer wishes to delay at a certain point in his program until transmission is complete and checked. The FUNCTION XKLIBRF (iod) allows this to be done, by holding control until transmission is complete and then returning the following values:-

- 1	if the transmission is successfully completed (EOP).
0	if EE and EOP interrupts occurred.
1	if UK and EOP interrupts occurred.
2	if UK and EOP and EE interrupts occurred.
3	any other type of interrupt - correction is impossible.

The first case indicates success

XKLIBRF (iod) = 0	=	means that a tape mark has been read or that the end of tape has been reached during writing.
	= 1	a data error has occurred.
	= 2	a data error has occurred, together with the conditions of case 0.

The last two conditions would require backspacing and retransmission for tape transmissions, or location and retransmission for disk transmissions.

If XKLIBRF is not used and a further transmission initiated, then a delay will occur until the first trnsmission is completed. If the first transmission was not completed successfully then the job will be rejected.

I5. MISCELLANEOUS ROUTINES

These are called by CALL subroutine (iod),

where iod = tape number
subroutine = xy LIBRF, and

xy = BF	backspace file
BR	backspace record
EG	the erase long gap trigger is set
FR	signal the operator that this tape is not used again
KN	turn on the check light
RN	turn on the reserve light
RF	turn off the reserve light
TN	turn on the tape indicator light
TF	turn off the tape indicator light
SF	space forward one file
SR	space forward one record
EF	write end of file
UL	tell operator to load next reel for this unit
RW	rewind

The successful completion of these actions should be checked by XKLIBRF which will notice special cases that arise, namely

XKLIBRF (iod) = 0 when

(a) The end of tape reflective strip is crossed by EFLIBRF

(b) A tape mark (i.e., end of file mark) is crossed by BFLIBRF, BRLIBRF, SFLIBR, SRLIBR. This is the normal case for BFLIBRF and SFLIBRF. BRLIBRF and SRLIBRF will space over a tape mark and give XKLIBRF (iod) = 0

Note: Backspacing at the beginning of the tape will give

XKLIBRF (iod) = 3

I6. TAPE AND DISC NUMBERING

The FORTRAN INPUT/OUTPUT routines use tape numbers in the range 1 to 8 and disk numbers 9 - 12. These conventional numberings allow control cards for MCP to be standardised.

For the non-buffered INPUT/OUTPUT routines, the user may have more freedom in numbering tape and disc sections.

APPENDIX J

SPECIAL INPUT/OUTPUT SUBROUTINES FOR BUFFERED TAPES

As mentioned in Section 7, the use of tapes falls into two distinct classes. To extend the "FORTRAN Statements" of Section 9 (buffered binary INPUT/OUTPUT) four special SUBROUTINES are provided to:-

(a) Backspace a file (BFFORT).

(b) Forward space file (SFFORT).

(c) Forward space a record (SRFORT).

(d) Unload a tape and reload the next reel (ULFORT).

These are all activated by means of the CALL statement, using the above names with the tape number as the argument.

Example: CALL BFFORT (I)

causes tape I to backspace the file until the previous file mark is passed.

APPENDIX K

IDENTIFICATION OF BINARY DECKS

A declaration is available in S2 which enables programmers to identify binary decks by including in their S2 deck a card having the word IDENTITY punched on it, starting in column 7, followed by a comma, followed by the identification. The S2 compiler transfers the contents of the 8 columns immediately following the comma, to columns 73 - 80 of the binary cards produced by the compilation.

Any character may be used in making up the identification, including blanks, e.g.,

```
IDENTITY, ABCD/12A

IDENTITY, +*-A.,A2
```

If this statement is not included in a compilation, the identification MAIN will appear for a MAIN program and the SUBROUTINE name will appear for a SUBROUTINE.

APPENDIX L

IOD AND REEL CARDS

L1. TAPE IOD CARDS

For each logical tape used, the programmer must provide an IOD card. There are two types of prepunched cards available - normal IOD's and "lazy IOD's".

In normal IOD's, the programmer must punch his specification of

CHANNEL, UNIT, MODE, DENSITY, DISPOSITION

according to the MCP rules. The uniqueness of tape is determined by the UNIT symbol and different CHANNEL symbols indicate a preference for having the tapes on different channels. MCP will put these tapes on different channels if it can, but if not will place them on the same channel. The following mnemonics should be used in normal tape IOD cards for the mode, density and disposition.

Mode field	ODD	odd parity, no ECC
	EVEN	even parity
	ECC	odd parity, ECC
Density field	HD	high density
	LD	low density

(For binary tapes the use of ECC, HD is recommended for the additional security of information given by The Error Checking and Correction Modes.)

Disposition field	NSAVE	-	the tape is not to be saved
	CSAVE	-	the tape is to be saved only if the job completes
	ISAVE	-	the tape is to be saved only if the job does not complete
	SAVE	-	the tape is to be saved in any case

If there is no punching NSAVE is assumed.

For most uses, however, "lazy IOD's" will be more convenient. In them CHANNEL, UNIT, MODE, DENSITY are prepunched in such a manner that to each logical tape a separate physical unit is allocated, and separate channels are used if possible. The mode will be ODD parity no ECC and the recording will be HD (high density). The programmer must then punch the DISPOSITION symbol as described for normal IOD's.

L2. REEL CARDS

If any of the tapes to be mounted are not "COMMON" tapes, then the programmer must nominate the reel by using a REEL card which should be placed immediately following the IOD for the tape concerned. A REEL card has

B	in column 1
REEL,	in columns 10 - 14
Symbol 1	in columns 15 - 22
, Symbol 2	in columns 23 - 31
, Symbol 3	etc.,

where Symbol 1, Symbol 2 etc., are 8 character symbols of the form xxxyyyyy, where xxx describes the status of the reel and is one of the following:-

PLB	protected and labelled
PUL	protected and not labelled
NLB	not protected but labelled
NUL	not protected and not labelled

and yyyyy is the tape label.

An example of a REEL card is

```
B REEL, PLBA1234, NLBA4321
```

which specifies that reel A1234 is to be mounted first on the unit, followed by reel A4321. The reel A1234 is protected from being written upon, whereas A4321 can be written.

L3. DISC IOD CARDS

Partly punched disc IOD cards are available by which logical files on the disc can be allocated. A channel symbol (any symbol not used as a tape channel symbol will do) followed by a comma and the number of tracks required should be punched starting at column 24 of the card. One track has 4096 words. If no number is specified the whole available disc will be given to the file; in this case only one file can be used.

APPENDIX M

TABLE OF SOURCE PROGRAM CHARACTERS

CHARACTER	CARD	BCD	CHARACTER	CARD	BCD	CHARACTER	CARD	BCD	CHARACTER	CARD	BCD
1	1	01	A	12 1	61	J	11 1	41	/	0 1	21
2	2	02	B	12 2	62	K	11 2	42	S	0 2	22
3	3	03	C	12 3	63	L	11 3	43	T	0 3	23
4	4	04	D	12 4	64	M	11 4	44	U	0 4	24
5	5	05	E	12 5	65	N	11 5	45	V	0 5	25
6	6	06	F	12 6	66	O	11 6	46	W	0 6	26
7	7	07	G	12 7	67	P	11 7	47	X	0 7	27
8	8	10	H	12 8	70	Q	11 8	50	Y	0 8	30
9	9	11	I	12 9	71	R	11 9	51	Z	0 9	31
blank	blank	20	+	12	60	-	11	40	0	0	12
=	8 - 3	13	.	12 8 - 3	73	$	11 8 - 3	53	,	0 8 - 3	33
'	8 - 4	14	)	12 8 - 4	74	*	11 8 - 4	54	(	0 8 - 4	34