INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED.

Programming Languages Division.

I.C.T. 1900 Series

FORTRAN NOTE 1

Issue 2

25th September, 1967.

1900 Fortran Arithmetic

This note replaces Fortran Note 1 dated 11.3.65. which should be destroyed.

1. The Accumulator

For the prupose of compiling instructions for arithmetic expressions an I.C.T. 1900 series computer is considered to be a one accumulator, one index register, single level store machine. Note is taken, however, that only quantities which might be in upper storage to be indirectly addressed.

The index register used is X3. The single accumulator will be called ACC in this document. The floating point accumulator will be called FPA.

The location of ACC is different according to the mode (INTEGER,REAL, COMPLEX etc.) of the quantity it contains. The locations of ACC for the different modes are given in the following table. The manner in which the quantity is held in ACC is described in Fortran Note 8.

Mode	ACC
Integer	X6
Real	FPA
Double Precision	FPA, X4, X5
Complex	FPA, X4, X5
Logical	X6

If an operation which calls for an operand in ACC has a result which is of a different mode, and the result is to be in ACC, then the accumulator will be the correct one for the new mode.

2. Arithmetic operations

Arithmetic operations in FORTRAN can be divided into 6 main types as follows

1. Binary operations e.g. A + B
2. Unary operations e.g. -A, .NØT. B
3. Array operations e.g. A(I)
4. Function operations e.g. FUN(X,Y)
5. Conversion operations e.g. Integer ⟶ Real
6. Store and Load operations e.g. ACC ⟶ Store

2.1. Binary Operations

This is the set of all definable operations of the form A op B where A and B may be any legitimate quantity of any mode and op is any operator which occurs as an infix with two such operands

viz. + - * / ** .OR. .AND. .EQ. .NE. .LT. .GE. .GT. .LE.

In such an operation the operands will be in the following form at the time the operations is executed.

1. One operand in ACC
 The other operand in Store
or 2. One operand in ACC
 The address of the other operand in X3

The result of a Binary operation will be a quantity in ACC. It is assumed that as the result of a Binary operation the original contents of X3 and ACC are destroyed.

2.2 Unary Operations

These are the operations of the set -A and .NØT. B where A is any legitimate quantity of mode other than LØGICAL and B may only be LØGICAL.

For these operations the operand is in ACC. The result of a Unary operation is a quantity in ACC. No unary operation affects X3.

2.3 Array Operations

These are operations which define the addres of an array element. The operands for the operation depend on the actual method used to generate the address.

The result of an Array operation is an address (of the correct array element) in X3. Nothing may be assumed about the contents of ACC following an array operation.

2.4 Function Operations

These are operations of the form FCN (A,B, ...) where A, B etc. are the arguments of the function.

The result of a function operation is a quantity in ACC where ACC is specified by the mode of the name of the function. The original contents of X3 and ACC are destroyed by a function operation.

2.5 Conversion Operations

These are operations which change the mode of a quantity without changing its value. There are of two types.

1. Those which change the mode of a quantity which is in ACC

2. Those which change the mode of a quantity which has its address in X3.

Type 1. Any Conversion operation which changes the mode of a quantity in ACC will leave the converted quantity in ACC according to the final mode of that quantity.

Type 2. Any conversion operation which changes the mode of a quantity whose address is in X3 will leave the converted quantity in some location and will give as a result in X3, the address of that location.

Operations of type 1 will not destroy the contents of X3

Operations of type 2 will not destroy the contents of ACC. ACC is defined by the final mode of the quantity converted.

2.6 Store and Load Operations

These are basically operations which are required to produce operands of acceptable form for the other types of operations and also to store final results. They are two types

1. ACC operations — Load ACC from Store

   Store ACC in Store (Assignment)

2. X3 operations — Load address into X3 from Store

   Store address from X3 in Store

Type 1 operations do not destroy X3

Type 2 operations do not destroy ACC

3. Arithmetic Packages

The operations referred to above are carried out either by compiling open ended subroutines or by compiling a calling sequence to a closed subroutine. Most of the simple operations are dealt with in the former manner, e.g. A = B + C where A, B, and C are of REAL mode results in the following being compiled.

```
LFP      C      Load ACC from store
FAD      B      Binary operation (type 1)
SFP      A      Store ACC in store
```

However had A,B and C been of DOUBLE PRECISION mode then the sequence

```
LFP      C     )
LDX  4   C+2   )Load ACC from store
LDX  5   C+3   )
LDN  3   B      Load X3 with address of B
CALL 1   %FDP   DP addition, binary operation (type 2)
SFP      A     )
STO  4   A+2   )Store ACC in store
STO  5   A+3   )
```

would have been compiled.

In the above, %FDP is the first entry point to a double precision package. There are a number of these arithmetic packages present in the Fortran Library.

%FAP4 - General purpose arithmetic package:-

(i) REAL binary operations* (redundant now)
(ii) Array operations of all types
(iii) INTEGER to INTEGER exponentiation
(iv) REAL to INTEGER exponentiation
(v) REAL/INTEGER conversion operations
(vi) Other special operations

%FEX4 - REAL to REAL exponentiation

%FDP - DOUBLE PRECISION package

(i) D.P. binary operations*
(ii) REAL to DOUBLE PRECISION conversion

%FD2 - DOUBLE PRECISION to INTEGER exponentiation
%FDR - DOUBLE PRECISION to REAL exponentiation
%FDD - DOUBLE PRECISION to DOUBLE PRECISION exponentiation

%FCP - COMPLEX package
(i) COMPLEX binary operations*
(ii) REAL to COMPLEX conversion

%FC2 - COMPLEX to INTEGER exponentiation

*Exponentiation which is also a binary operation is itemised separately since it is a much more complicated operation.

A brief specification of these subroutines is given in the Appendix.

4. Mixed Mode binary operations

Where the two operands are of different mode the operation is normally carried out in two stages.

Stage 1 One or two conversion operations to make the two operands of the same mode.

Stage 2 Binary operation

The net effect is as defined in section 1.1. A similar situation exists for mixed mode assignment operations.

As an example consider A = B + C where A is INTEGER, B is DOUBLE PRECISION and C is INTEGER

The code generated is:-

```
LDN   6   C            Load ACC
CALL  1   %FAP4+15     Convert to REAL
LDN   4   0   )
LDN   5   0   )        Convert to DOUBLE PRECISION
LDN   3   B            Load X3 with address of B
CALL  1   %FDP         DP addition
CALL  1   %FAP4+18     Convert to integer
STO   6   A            Store ACC
```

Note that the R.H.S. is always calculated in the 'highest level' mode (i.e. Double Precision is higher than integer) and that the mode of the result is always determined by the mode of the L.H.S.

5. Functions and their Calling Sequence.

There are a number of different types of 'functions'.

1. Basic intrinsic functions
   e.g. ABS, MIN, INT
2. Basic external functions e.g. SIN, SQRT
3. Private Fortran subroutines; those beginning %F,
   e.g. %FAP4, %FINOUT
4. Other external functions e.g. PLOT, ITIME
5. Source language FUNCTION procedures
6. Source language SUBROUTINE procedures
7. Arithmetic statement functions
8. 'Implied DO' functions.

Those of type 1,2,3 or 4 are held in the Fortran Library, the remainder are defined by the user.

Those of type 1,2,5 and 7 return the result in ACC and are as defined in section 1.4

Apart from those of type 3 and certain basic intrinsic functions which have a variable number of arguments the following standard calling sequence is used:

```
CALL   1   SUBR
LDX    3   ARGI   ⎫
LDN    3   ARG2   ⎪     One instruction per argument
 .     .    .     ⎪
 .     .    .     ⎬
 .     .    .     ⎪
 .     .    .     ⎪
LDX    3   ARGN   ⎭
```

The instructions in the argument list are determined as follows:

1. If the argument is a variable, the argument word must be an instruction which, if obeyed, will place the address of the argument into X3. This will normally be a LDN or a LDX instruction

2. If the argument is a function (to be used as a dummy function) the argument word must be an instruction which, if obeyed, will place the instruction BRN Start of Function. into X3.

3. If the argument is an array, the argument word must be an instruction which, if obeyed, will place the address of the array header (see Fortran Note 25) into X3.

The called routine will return to the first location after the calling sequence. This implies that both the calling and the called routines are independently aware of the number of arguments and of their types.

The 1900 FORTRAN there are certain Basic Intrinsic Functions which have a variable number of arguments. For these functions, the compiler will insert as the first word of the argument list (prior to the first actual argument) a word which contains the 'number of arguments'.

6. Array Addressing

This is fully described in Fortran Note 25.

Appendix.

Brief Specification of the Fortran Arithmetic Subroutines

Notation

X = REAL number held in FPA
Y = REAL number whose address is in X3
Z = REAL number whose address is in X3 but which is temporarily stored in common area %LIB
I = INTEGER number held in X6
J = INTEGER number whose address is in X3
K = INTEGER number whose address is in X6
N = INTEGER number held in X3 (<4096)
DX = DOUBLE PRECISION number held in FPA, X4, X5
DY = DOUBLE PRECISION number whose address is in X3
CX = COMPLEX number held in FPA, X4, X5
CY = COMPLEX number whose address is in X3

General

The link accumulator used is always X1. The contents of accumulators are not preserved. Entry points to each package are relative to the package name.

1. %FAP4 - Arithmetic Package

Entry points : +0 NULL (used to be X=X+Y)
+1 NULL (used to be X=X*Y)
+2 NULL (used to be X=X-Y)
+3 NULL (used to be X=Y-X)
+4 NULL (used to be X=-X)
+5 NULL (used to be X=X/Y)
+6 X3 = Address of 1 or 2 word array element (3 or more dimensions)
+7 NULL (used to be X=Y/X)

+8	I=I ** J
+9	I=I ** N
+10	X=X**J
+11	X=X**N
+12	I=J**I
+13	I=N**I
+14	X=Y**I
+15	X=I
+16	Z=J, X unchanged
+17	Z=N, X unchanged
+18	I=X
+19	TRACE initialization
+20	X3=Address of 4 word array element
+21	X=I/J
+22	X=I/N
+23	X=J/I
+24	X=N/I
+25	X3+Address of 1 or 2 word array element (2 dimensions)

2. %FDP - Double Precision Arithmetic Package

Entry points:

+0	DX=DX+DY
+1	DX=DX*DY
+2	DX=DX-DY
+3	DX=DY-DX
+4	DX=-DX
+5	DX=DX/DY
+6	DX=DY/DX
+7	DX=Z

Entry points +8 to +15 also exist, and are used by double precision routines to perform the above operations on non-standard (unpacked) D.P. numbers.

3. % FCP - Complex Arithmetic Package

Entry points: +0 CX=CX+CY
+1 CX=CX*CY
+2 CX=CX-CY
+3 CX=CY-CX
+4 CX=-CX
+5 CX=CX/CY
+6 CX=CY/CX
+7 CX=X**CY

4. % FEX4 - Real Exponentiation

Entry points: +0 X=X**N
+1 X=N**X

5. % FD2 - D.P. to Integer Exponentiation

Entry points +0 DX=DX**J
+1 DX=DX**N
+2 DX=DY**K

6. % FDR - D.P. to Real Exponentiation

Entry points +0 DX=DX**Y
+1 DX=DY**X

7. % FDD - D.P. to D.P. Exponentiation

Entry points +0 DX=DX**DY
+1 DX=DY**DX

8. % FC2 - Complex to Integer Exponentiation

Entry points +0 CX=CX**J
+1 CX=CX**N
+2 CX=CY**K

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED

Scientific Programming Department
I.C.T. 1900 Series

FORTRAN NOTE 2
26.3.65

A Proposed Method of Describing a FØRTRAN Program to the FORTRAN IV (ICT) Compiler

Comment on ASA Specifications

The ASA Fortran language enables programs to be written in a machine independent form, but in itself does not allow enough information to enable programs to be completely described and successfully run on the 1900 Series computers.

In particular, the language does not allow a complete description of object time peripheral operations in that no explicit definition of the type of peripheral device to be used can be given; nor is there any way within the language of specifying auxiliary information which may be required for the successful use of some types of peripheral device (e.g. a File name for magnetic tape).

Also if a compiled program is too large for a particular machine, the language does not indicate a way of generating or referencing Overlay procedures to enable the program to be run in the space available.

In order to complete the information required by the compiler the statements of the first segment of program to be compiled must be preceded by a "Program Description". This program description is only necessary if a complete program is being compiled and consolidated, not if a segment or series of segments not comparing a complete program is being compiled. It is immaterial whether this 'complete program' actually contains any FØRTRAN source segments or contains only semi-compiled segments.

Program Description

A Program Description consists of a series of statements from the following family of statements (written in FØRTRAN Format but with no continuation lines).

INPUT

ØUTPUT

INPUT/ØUTPUT

ØUTPUT/INPUT

ØVERLAY

introduced by the statement PRØGRAM

PRØGRAM Statement

This statement is written in the form

PRØGRAM Name, priority, retention cycle
or PRØGRAM Name, priority
or PRØGRAM Name

"Name" is the name under which the program will run at object time.

The first four characters of Name (including spaces) are taken to be the program name.

"Priority" is an integer between 1 and 99 which is the priority of the object program. If this is missing from the PRØGRAM statement the priority is taken to be 50.

"Retention Cycle" is the period (in days) for which the object program is protected. No retention cycle is required if compilation is to punched tape or punched cards. If the retention cycle is required but not stated it is taken as 0.

Peripheral Statements

These are the INPUT, ØUTPUT, INPUT/ØUTPUT and ØUTPUT/INPUT statements. They describe the various methods of operation of the peripheral devices and all have the same general format.

The format of a peripheral statement is as follows

Type $n_1, \dots n_k$ = peripheral information.

"Type" is INPUT, ØUTPUT, INPUT/ØUTPUT or ØUTPUT/INPUT.

The "n_i" are the values by which the peripheral device is known within the object program. They are small integer (<4096).

"peripheral information" designates the peripheral device which is known by the values "n_i" and may give auxiliary information such as a "file name" for that device. The format for "peripheral information" is as follows.

peripheral designation (auxiliary information)
or peripheral designation.

The "peripheral designation" is a 3 character code which defines the device. The first two characters define the type of device, and the third character specifies the device number (0-7).

Initially the types of device are as follows

TR - tape reader
TP - tape punch

CR - card reader

CP - card punch

MT - magnetic tape

LP - line printer

The "auxiliary information" if applicable consists of a File name and a retention cycle separated by a comma. If it is not necessary to have a retention cycle the auxiliary information consists of a File name alone.

INPUT Statement

This statement is used to describe a peripheral device which is only going to be used for input operations.

e.g. INPUT 3,5 = TRO indicates that within the program, the value 3 or the value 5 in an input operation describes the primary tape readers as the current peripheral device.

ØUTPUT Statement

This statement is used to describe a peripheral device which is only going to be used for output operations.

e.g. OUTPUT 10 = MTO (JACK/SYSTEMS,3) indicates that within the program, the value 10 in an output operation describes the first magnetic tape deck as the peripheral device. Further the magnetic tape will be opened with a File name "JACK/SYSTEMS" and will have a retention cycle of 3 days.

INPUT/ØUTPUT Statement

This statement is used to describe a peripheral device (initially only a magnetic tape unit) which is going to be used for both input and output operations. The essential feature about the use of this statement is that the object program expects to input information from the device before it uses the device in an output operation.

e.g. INPUT/ØUTPUT 15 = MT1 (FORTRAN DATA) indicates that within the the program, the value 15 in a peripheral operation describes the second magnetic tape unit. Further, the first peripheral operation using this tape unit will be an input operation from a tape with a File name "FØRTRAN DATA".

ØUTPUT/INPUT Statement

This statement is used to describe a peripheral device (initially only a magnetic tape unit) which is going to be used for both input and output operations. The essential feature about the use of this statement is that the object program expects to write to the device before reading from it.

e.g. ØUTPUT/INPUT 20 = MT2 indicates that within the program the value 20 in a peripheral operation describes the third magnetic tape unit. Further the first peripheral operation using this tape unit will be a "write" operation to a "Scratch" tape. (no File name).

ØVERLAY Statement

This statement is written in the following form

ØVERLAY n, a = b = --- = p

n is a small integer called the "Overlay Area Number" (n ⩾ 1)

a,b---p are the names of segments in the program which are obeyed from the Overlay Area numbered "n". No two segments in this list are in the main store at the same time.

The only segments which may lie in an overlay area are SUBRØUTINE segments with no formal arguments.

e.g. a subroutine called by the statement CALL JØE may lie in an overlay area, but a subroutine called by the statement CALL JACK (A) may not lie in an overlay area.

Monitor Peripherals

A FØRTRAN program may have a monitor routine associated with it. If this routine is to be used, it is necessary to indicate to the Compiler from what device monitor steering information will be read, and to what device the monitor results are to be sent. This is done by means of the peripheral statements INPUT and ØUTPUT where the peripheral value is given by the word "MØNITØR".

Thus the statement INPUT 3, MØNITØR = TRO indicates that monitor information will be read from the primary tape reader. The value 3 in a standard input operation also indicates that reader.

The statement ØUTPUT MØNITØR = LPO indicates that monitor results will be expected on the primary Lineprinter.

Summary

A Program Description consists of a selection of statements of the types described which will fully define all relationships among object time peripheral values and devices, and define all overlay areas and their contents according to the rules given above. This Program Description is introduced by a PROGRAM statement. This statement must occur at the head of a Program Description and may not occur anywhere else.

The Program Description to the Compiler

The Compiler treats the Program Description in the same manner as it treats a Program Segment. For the various statements it generates constants and lists for use at object time in the form of Semi-compiled program. This psendo segment is given the name %%%F and is compiled as a "subroutine" rather than as a "master". The leader generated for this psendo segment contains all the cue information about peripheral devices and I/Ø object time packages and about overlays and overlay packages that is required by the Consolidator.

All Program Descriptions compile into the same name psendo segment (%%%F) so that a semi-compiled version of the Program Description can be superceded by a new version if a segment of the original program has to be recompiled, followed by the original semi-compiled version of the complete program.

V.K. Taylor
K.F. James

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED

Scientific Programming Department
I.C.T. 1900 Series

Superceded by Note 10 and 10A

FORTRAN NOTE 3
27.3.65.

A Proposed Method of Describing and Implementing Program Overlays for the FØRTRAN IV (ICT) Compiler

General

If there exists a suitable "backing store" for the object program there will have to exist a facility through which more than one segment of a program may be compiled to occupy the same main store locations, being read from the backing store to the main store when necessary.

The notes which follow give a proposal for program overlay facilities within the framework of FØRTRAN IV (ICT) without adding to the FØRTRAN language.

For the purpose of this document the following terms are now defined.

1. Overlay Segment - A program segment which is obeyed from the same area of main store as another segment.

2. Overlay Area - an area of main store from which overlay segments may be obeyed. At any time an overlay area contains only one complete overlay segment. An overlay segment of an overlay area has its first main store program address at the beginning of that area.

3. Overlay Area Number - A small integer which 'labels' a particular overlay area.

FØRTRAN Overlays

A FØRTRAN IV (ICT) program may consist of a main section of program with no overlay segments; or it may contain a main section of program which may not be overlaid, and two or more overlay segments which may be obeyed from one or more overlay areas. No one segment may be obeyed from more than one overlay area. At any time an overlay area contains only one overlay segment.

For FØRTRAN IV (ICT) it is proposed that the only segments which may be overlay segments - and are therefore the only segments which may themselves be overlaid - are segments which are introduced by a statement of the form.

SUBRØUTINE Sub

where "Sub" is the name of a segment. "Sub" may <u>not</u> have an argument list.

All overlay segments which are to be obeyed from a particular overlay area are described to the FØRTRAN IV (ICT) compiler by a statement in the "Program Description" of the form.

ØVERLAY n, Sub1 = Sub2 = ... = Sub k

where "n" is the overlay area number, and the "Sub i" are the names of the segments which will be obeyed from that overlay area.

(For details of the "Program Description" read "A Proposed Method of Describing a FØRTRAN Program to the FØRTRAN IV (ICT) Compiler" - FORTRAN NOTE 2)

Program Communication between Overlay Segments

Within FØRTRAN segments, at the source program level, there is no indication as to whether a given segment of the approporate type is an overlay segment. Thus, regardless of the "status" of a segment the programmer will still write

CALL Sub

to enter the segment "Sub"; and from within that segment, the statement

RETURN

will cause the original segment to be re-entered at the statement following the CALL statement.

The actual communication between any segment and a possible overlay segment which is being called proceeds via a subroutine (%FOVL).

In the "calling" routine (call it A), the statement CALL B causes the following sequence to be compiled.

```
CALL 1 %FØVL
       M1
       M2
BRN    Start of B
```

where M_1 is the address of a "Constants" area which contains the name A. Similarly M_2 is the address of the name "B". In both cases the first cases the first character of the name is the number of words in the name (including that first character). Thus if the name of "A" is JACK, the "name" in the Constants area will be "2 JACK sp sp sp".

From within B, the statement RETURN causes the following instruction to be compiled

```
BRN    %FØVL + 1
```

Note that these sequences are only compiled if B is of a form which might allow it to be used as an overlay segment.

The Operation of %FØVL

This routine ensures that the correct program segment is in store when a call for that segment is made. It also keeps track of which segments have been "calling" possible overlay segments so that on execution of a RETURN statement it can ensure that the original calling segment is in store.

To perform these operations, %FØVL makes use of two (possibly three) lists. The compiler generates a list of Overlay areas and overlay addresses along with the names of the overlay segments. %FØVL generates a pushdown list (stack) of the names of segments which have used it to enter possible overlay segments. This stack also contains the links for these segments. For certain types of Backing Store there may have to be a third list (generated by the program loader) of where on the Backing Store the overlay segments have been stored. Even for magnetic tape storage the tape deck must be known to %FØVL.

%FØVL has two entries - at the first word to enter a possible overlay segment, and at the second word to return from such a segment. When %FØVL is entered at the first word, the word following the call to %FØVL contains the address of an area holding the name of the calling routine. The second word following the call contains the address of the name of the called routine. In each case, the first character of the name gives the number of words in the name (including that first character). The third word following the call contains a branch to the desired subroutine.

Whether or not the called routine is actually an overlay segment, the name of the calling segment and the link (X1) is stored in the "Stack". %FØVL then compares the name indicated by the second word following the call to its overlay list. If the name is not in the list or if the name is in the list but the segment is already in the main store, the third word following the call is obeyed as an instruction.

If the name is in the Overlay List and the segment is not in the main store, %FØVL causes the segment to be read into store at the address specified by the starting point for the correct overlay area, after which the third word following the call for that segment is obeyed as an instruction. Note that in case the calling segment is in the same overlay area as the called segment the third word following the call must be obeyed from within %FØVL's own working store.

When %FØVL is entered at the second word (to return from a subroutine), the name at the top of the stack is compared with the items in the overlay list. If the required segment is not in store it is read down from the Backing store. The Link (also in the stack) is extracted and both the link and the segment name are removed from the stack. The link is loaded into X1 and the instruction "EXIT 1 3" is obeyed.

Note that any operations involving the reading of an overlay segment into store requires that the overlay list for the particular overlay area be updated to indicate the correct segment in store.

%FØVL Lists

1. Overlay List

This list is generated by the compiler and occupies the Common area %FØVT.

The List contains two types of item.

Type 1. Overlay Area Item. This consists of two parts.
part 1 (one word) - The main store address of the start of the Overlay area.
part 2 (one word) - The address of the List item of type 2 (Overlay Segment Items) for the segment currently in the overlay area.

Type 2 Overlay Segment Item. This consists of three parts.
part 1 (one word) - The address of the previous item of type 2 in the list. If there is no "previous item", this word is zero.
part 2 (one word) - The address of the list item of type 1 for the particular overlay area.
part 3 - The name of the overlay segment. The first character of the name gives the length of the name in words (e.g. for a name JACK, this part of the list item will be 2JACK sp sp sp)

These will be as many items of type 1 as there are overlay areas

There will be as many items of type 2 as there are overlay segments.

The first word of the Common area %FØVT gives the address of the last item of type 2 put into the list.

2. %FØVL Pushdown List (Stack)

This consists of a word pointing to the last item put in the stack.

An item consists of two parts.

Part 1 (one word) the link for the segment.
Part 2 The name of the segment. The first character of the name gives the length of the name in words.

The Compiler - Consolidator - Loader Communication for Overlays

When the Compiler processes an ØVERLAY statement within a Program Description it generates a standard blank cue for the overlay routine (%FØVL) but also generates on "Overlay Area" cue for each overlay area number and an "Overlay Segment" cue for each segment name mentioned in the ØVERLAY statement. This overlay segment cue is of the same family as "unsatisfied" (blank) cues but it has attributes which indicate which overlay area the segment will finally occupy.

When the Compiler processes any segment (including overlay segments) it generates a "Program" cue which gives the length of that segment.

When the Consolidator finds a program cue when it already has an overlay cue for that segment, the overlay cue is marked as "satisfied" and the value in the program cue is compared with the value of the associated overlay area cue. The greater of the two values becomes the new value for the overlay area cue. The original program cue is then forgotten. If the Consolidator finds a program cue but does not have an overlay cue for that segment, the Consolidator takes the program cue itself as completely describing the program segment.

In the Consolidated Leader, a cue for a program segment is either a "program" cue or an "overlay" cue. A program cue will contain the starting address of the program segment. An overlay cue will give the overlay area number for the segment. If a program segment is defined in an overlay cue, the Consolidator will also have generated an "overlay area" cue for the particular overlay area number. This overlay area cue will contain the starting address of the overlay area.

The Loader now has enough information from the Consolidated Leader to load each segment of the object program.

Program Segments represented by Program Cues in the Consolidated Leader are loaded correctly. Segments represented by Overlay Cues are loaded into a "Temporary Overlay Area" of the loader and then written to the Backing Store in binary form. Depanding on the Backing Store the loader may have to generate a list of Backing Store addresses, the starting addresses in the Backing store for the various segments. This list will have to be in an area accessible to %FØVL (or at least an index word to the list will have to be accessible.)

V. K. Taylor

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED

Scientific Programming Department
I.C.T. 1900 Series

FORTRAN NOTE 4
12.4.65.

FØRTRAN IV (ICT) Input and Output Operations

In FØRTRAN IV the following types of statement are available to the programmer

```
READ (i,n) list
READ (i) list
WRITE (i,n) list
WRITE (i) list
BACKSPACE i
REWIND i
ENDFILE i
```

where i is the "Peripheral Value" (P.V.) and may be either a small integer (<4096) or an integer variable reference. "n" is either an integer which is the label of a FØRMAT statement or is an Array name in which case the Array itself is taken as being a FØRMAT specification.

By association with a peripheral type statement in the Program Description the Peripheral Value at the time the I/Ø statement is initialized determines the particular peripheral device which is to be used for that operation.

Thus ifiin the Program Description there is a statement

```
INPUT 10 = TRO
```

and in the body of the program there is a statement

```
READ (10,105) A,B,I
```

The input operation specified would use the primary tape reader. The following two statements would also have the same effect.

```
I = 10
READ (I,105) A,B,I
```

Input and Output Operations - to the Compiler

On recognizing any I/Ø statement, the Compiler generates cues to the private library routine %FINØUT which performs the actual peripheral operations.

For all entries to %FINØUT, at the time the call to %FINØUT is obeyed, X3 and X6 will give the parameters required. The actual entries (and entry requirements) are as follows.

Entry	Operation	X3	X6
%FINØUT + 2	ENDFILE	____	P.V.
%FINØUT + 4	Initiate WRITE	address of FØRMAT (0 if no FØRMAT)	P.V.
%FINØUT + 6	Initiate READ	address of FORMAT (0 if no FØRMAT)	P.V.
%FINØUT + 8	Read or Write Integer Item	address of Item	-
%FINØUT + 10	Read or Write Real Item	address of Item	-
%FINØUT + 12	Read or Write double Precision Item	address of Item	-
%FINØUT + 14	Read or Write Complex Item	address of Item	-
%FINØUT + 16	Read or Write Logical Item	address of Item	-
%FINØUT + 18	Terminate READ or WRITE	-	-
%FINØUT + 20	BACKSPACE	-	P.V.
%FINØUT + 22	REWIND	-	P.V.

In addition to those entries to %FINOUT, there are also the following four entry points which are used in associated PLAN compiled subroutines for the operations "Runout" (paper tape), "Disengage", "Release" and "Allocate". In each case X6 = P.V.

%FINØUT + 0	Runout
%FINØUT + 24	Disengage
%FINØUT + 26	Release
%FINØUT + 28	Allocate

The list of a READ or WRITE statement may include an Array name, in which case it is required to transfer all the elements of the array. When this situation occurs, the compiler generates a call to a private library routine %FIØTA which processes the whole array.

%FIØTA has the following entries.

%FIØTA + 0	Process Integer Array
%FIØTA + 2	Process Real Array
%FIØTA + 4	Process Double Precision Array
%FIØTA + 6	Process Complex Array
%FIØTA + 8	Process Logical Array

On entry to the routine %FIØTA, X3 contains the address of the "Array Header" for the particular array.

As an example of the use of %FINØUT and %FIØTA note the results of compiling the following statement.

READ (10,105) A,B,I

where B is an array name. A and B are both "Real", I is "Integer".

```
LDX   3      a              a contains address of Format labelled 105
LDN   6      10             X6 = P.V.
CALL  1   %FINØUT + 6       Initiate "Read" operation
LDN   3      A              X3 = Address of A
CALL  1   %FINØUT + 10      Process "Real" Item
LDN   3      b              b = 1st word of Array Header for B
CALL  1   %FIØTA + 2        Process the entire array B
LDN   3      I              X3 = Address of I
CALL  1   %FINØUT + 8       Process "Integer" Item
CALL  1   %FINØUT + 18      Terminate  "Read" operation
```

For the statement ENDFILE J, the following is compiled

```
LDX   6      J              X6 = P.V.
CALL  1   %FINØUT + 2       Do "Endfile" operation
```

V.K. Taylor

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED

Scientific Programming Department,
I.C.T. 1900 Series

FORTRAN NOTE 5
30.4.65.

The FORTRAN IV Object Time Input/Output System

This note gives a brief description of the input/output system used by FORTRAN IV object programs. The emphasis is on how communication between the different parts of the system is achieved. Further information on the system will appear in later notes. (Some information has already appeared in Note 4.)

Amendment to FORTRAN Note 4, page 2.

Entries 0, 24, 26 and 28 to %FINOUT are replaced by entry 0 meaning: all other operations. The actual operation is specified by an integer parameter in X3.

The FORTRAN IV Object Time Input/Output System

1. Outline of System

The FORTRAN IV input/output system is modular. At run time, only those parts that are needed will be present. The relationship between the different parts is shown diagramatically in Figure 1.

For each source language input/output statement the object program will contain one or more calls to %FINOUT. %FINOUT is a segment that carries out the bulk of the work of I/O in a way that is independent of the I/O medium involved. %FINOUT enters one of a series of peripheral routines to carry out actual I/O operations. Each peripheral routine is a segment.

%FIOTA is a segment that is called for each occurrence of a total array in a READ or WRITE statement.

2. Communication between %FINOUT and the Peripheral Routines

%FINOUT does not know about the different I/O media but merely enters appropriate peripheral routines to perform the actual I/O operations. Communication takes place via four common areas: %FIOLIST, %FIOBUF, %FIOPER and %FIOLEN.

The Program Description for a FORTRAN IV program generates in the common area %FIOLIST a list of programmer's numbers that may be used by that program. Associated with these numbers are the details of the actual devices and all auxiliary information required for the successful operation of these devices, including the addresses of the appropriate peripheral routines.

The term "Programmer's number" replaces "peripheral value" of Note 4.

An initial entry to %FINOUT will search %FIOLIST for auxiliary information corresponding to a specified programmer's number. Then the one word area %FIOPER is set to point to this information for use by the peripheral routine.

The layout of the common area %FIOBUF is shown in Figure 2. It is used to transmit information between %FINOUT and the peripheral routines during READ and WRITE operations. This information is transmitted in units of blocks, where one record may consist of one or more blocks. The block size depends on the peripheral routine. A formatted record is likely to consist of one block and an unformatted record of several blocks. Formatted information will be in standard 6-bit code (without shifts). Unformatted information will be in binary form. %FIOBUF is defined as one word in %FINOUT but its actual length is the maximum of the lengths defined in the peripheral routines.

During a READ or WRITE operation the initial entry to the peripheral routine will set the one word common area %FIOLEN equal to the maximum number of characters that %FINOUT may assume to exist in %FIOBUF.

3. Structure of %FIOLIST

%FIOLIST contains a chained list of information about the peripherals used by a program. It contains an entry for each of the INPUT, OUTPUT, USE or CREATE statements in the Program Description (see Note 2). USE and CREATE replace INPUT/OUTPUT and OUTPUT/INPUT of Note 2. An entry consists of one or more pointers and an information block.

For each programmer's number in the statement there is a three word pointer:

1. List word
2. Programmer's number
3. Address of information block.

The 'list word' contains the address of the previous pointer in %FIOLIST. If there is no previous pointer the word is zero. The first word of %FIOLIST points to the end of the list, i.e. it contains the address of the last pointer put into the list.

For each occurrence of the word MONITOR in a statement there is a similar pointer, except that the second word is n, where n < o. The value of n indicates whether the device is available for input, output or both.

An information block contains the following items:

1. Address of the start of the Peripheral Routine (1 word).
2. Device details (2 words).
3. Zero word.
4. Length in words of rest of information block (1 word).
5. Length in words of file name (1 word). } Optional
6. File name (0 or more words). } Optional
7. Any further information. } Optional

N.B. Before entering a peripheral routine, %FINOUT sets %FIOPER equal to the address of the information block.

Item 1 of the information block gives the starting address of the appropriate peripheral routine and is used by %FINOUT.

Item 2 consists of two words as follows:

	Value of word		Meaning
Word 1	Top bit = 1		Device not released
	Rest of word = 1	Indicates type of statement in Program Description	INPUT
	= 2		OUTPUT
	= 3		USE (ex INPUT/OUTPUT)
	= 7		CREATE (ex OUTPUT/INPUT)
Word 2	Top bit = 1		Device not yet used
	Rest of word = n		n is the number in the 1900 device name, e.g. 3 in MT3.

N.B. The top bits of both these words may be referred to and changed only within peripheral routines since they may not have precisely the same meaning for all peripheral routines; some peripheral routines may ignore them altogether.

Item 3 of the information block is a word that is available as working space for the peripheral routine.

Item 4 may be zero.

Item 5 may be zero.

Item 6 will contain the file name left justified. Spaces will be added on the right to make a whole number of words. The compiler imposes no limit on the length of a name.

Item 7 may be a retention cycle and/or any other information that proves desirable.

K. F. James.

V. K. Taylor.

Diagram of Input/Output System

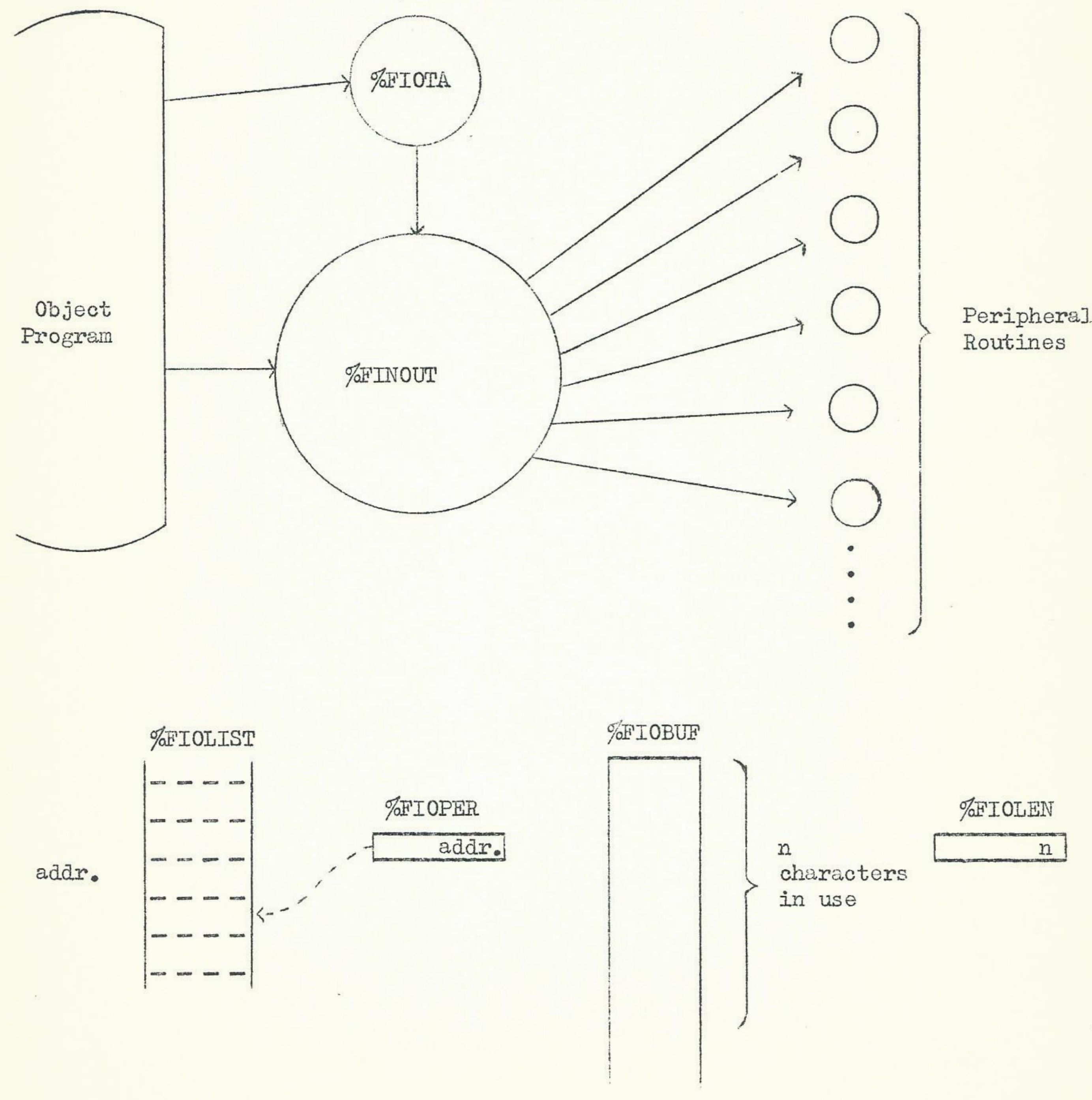


FIGURE 1

%FIOBUF

Layout		Formatted	Unformatted
	Words 1-4	Used by Peripheral Routines	
	Word 5	Top bit = 1 - last block of a record; 0 - other than last block of a record. Rest of word - no. of characters of information *	
	Word 6	[1 | | | A] Top bit = 1, A = 1st char. of information.	[0]
		Rest of Information	Information

* No. of characters of information starts at 3rd character of word 6 for formatted information and at 1st character of word 7 otherwise.

FIGURE 2

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED

Scientific Programming Department
I.C.T. 1900 Series

FORTRAN NOTE 6
3.5.65.

Trace System for FORTRAN 4 and Current 1900 FORTRAN

The first part of this note describes, from the user's point of view, the trace system to be implemented for FORTRAN 4. The second part of the note indicates how the current 1900 FORTRAN system differs.

This series of notes is intended for reference by the writers of the compilers and associated software. This particular note is being given a wider circulation than usual because it gives details of interest to programmers using the current 1900 FORTRAN compiler.

Trace System for FORTRAN 4

1. Outline

The FORTRAN Trace System provides assistance in diagnosing faults in a FORTRAN program by providing additional output from the program while it is running.

To use the trace system, the program, or alternatively only selected segments of the program, must be compiled in trace mode. When this is done, a special library segment is included in the object program and is called once during the execution of every statement.

If trace is to be used the source program must define one output and one input device to be 'monitor' peripherals (the method for FORTRAN 4 is described on page 4 of Note 2). The same devices may also be used for normal input or output.

2. Steering List

To operate the trace system a steering list must be prepared defining certain labelled statements as "triggering statements" and specifying the action required at these triggering statements.

The list is prepared as follows:-

n	NAME OF SEGMENT		Heading line	List for one segment
l_1	d_1	c_1	Detailed line	
l_2	d_2	c_2	" "	
....			" "	
l_n	d_n	c_n	" "	
n	NAME OF SEGMENT			Lists for other segments
l_1	d_1	c_1		
....				
....				
0				

The heading line for each segment contains an integer n in columns 1 to 6 which defines the number of detailllines to follow. The name of the segment must be given starting in column 7 and must not contain any spaces.

The detail line(s) consist of three integers separated by one or more spaces.

The first integer (l) is the label of a statement within the segment. This label must not appear in more than one detail line.

The second integer (d) specifies a "delay" and may be in the range -99 to 2047.

The third integer (e) specifies a "count" and may be in the range 0 to 4095.

After the lists for each segment to be traced, there must be a terminating line containing a zero in column 1.

If the steering list is prepared on paper tape, "tab" must **not** be used.

3. <u>Output</u>

When the object program executes a triggering statement:-

1. Any trace output still in progress from a previous triggering statement is terminated.

2. A line is printed giving the label number and segment name of the triggering statement. This is preceded by a blank line.

3. If the delay (d) is negative and the count (c) is non-zero, then statement lines are printed starting d statements in the past up to the present statement (except that at most c lines are printed).

4. Arrangements are then made to print statement lines as they are executed commencing with the current statement if $d \leq 0$ or after a delay d such that the total number of lines (both past and future) = c.

Each statement line consists of:-

1. The line number. The triggering statement itself ≙ line number 0. Other lines are numbered positively or negatively from this.

2. The label if any.

3. An abbreviation of the type of statement.

4. V if the overflow indicator is set.

5. The result of the statement, if applicable.

If a change of segment occurs between statement lines, the name of the new segment is given on a line by itself.

The end of a DO loop, although not written as a statement in the source program, is treated as a statement (type LOOP) by the trace system and is "executed" each time around the loop.

The statements, their abbreviations and results are as follows:-

STATEMENT	ABBREVIATION	RESULT
Arithmetic assignment	ARTH	Value of right hand side
Logical assignment	LOGC	Value of r.h. side: TRUE or FALS.
Arithmetic IF	IF	Value of expression in parenthesis.
Logical IF	LIF	Value of logical expression: TRUE or FALS.
GO TO	GO	
DO (	DO	m_1
(	LOOP	m_2-i
Computed GO TO	CGO	i
Assigned GO TO	AGO	
READ	READ	
WRITE	WRTE	
PAUSE	PAUS	
STOP	STOP	
CALL	CALL	
RETURN	RETN	
ENDFILE	ENDF	
CONTINUE	CONT	
BACKSPACE	BACK	
REWIND	REW	
ASSIGN	ASGN	

4. Notes on Operation of Trace System

a) The steering list will normally be read immediately after the object program has been loaded, but a new steering list may be read at any time and will replace any previous list, after which the program may be restarted.

b) Trace output may be switched on or off at any time during the execution of the object program.

c) When the object program is run in trace mode all array subscripting is checked and:-

ARRAY SUBSCRIPT ERROR

will be output if the selected element does not lie within the confines of the array. The program cannot be continued beyond this point.

d) Any time after the program has run it is possible to output the last 100 statements executed. This is useful if the program should reach an "illegal" operation or otherwise comes to an undesired halt.

Trace System for current 1900 FORTRAN

This is similar to the system described above. The method of assigning monitor peripherals is to call them MONIN and MONOUT for input and output respectively. The names are assigned in PERIPHERAL Statements in the normal way, e.g.

PERIPHERAL (MONIN, TRO), (MONOUT, LPO)

The list of possible statements given in 3 above is slightly shorter: Logical assignment, Logical IF, Assigned GO TO, BACKSPACE, REWIND and ASSIGN are not available.

K.F. James.

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED

Scientific Programming Department
I.C.T. 1900 Series.

FORTRAN NOTE 7
21.5.65.

The FORTRAN IV Object Time Peripheral Routines

This note describes the structure of the peripheral routines that may be used in FORTRAN IV object programs. It assumes a knowledge of FORTRAN NOTES 2, 4 and 5.

FORTRAN IV Object Time Peripheral Routines

Input/Output in FORTRAN IV object programs is handled by one basic segment called %FINOUT and one or more peripheral routines. There is a separate peripheral routine for each type of input/output medium. %FINOUT does most of the work; the peripheral routines carry out only the physical transfers and any required conversion to or from 1900 internal code. Communication between %FINOUT and the peripheral routines is described in FORTRAN NOTE 5.

The compiler will decide which peripheral routines are needed by looking at the peripheral statements in the program description. These may be typified by:

```
INPUT   1 = p/q
OUTPUT  5 = p/q (....)
```

where: p is a 1900 peripheral name, e.g. MT3

q is a qualifying phrase.

In a particular case one of p or /q may be absent.

Usually, only p will be present. The compiler will include details of the device in an information block and will cause a standard peripheral routine to be included in the object program. The following routines will be available as soon as possible:

p	Peripheral Routine	For
TRn) TPn)	%FIOPT	8-track paper tape in 1900 code.
CRn) CPn)	%FIOCARD	punched cards in 1900 code.
LPn	%FIOLP	line printer.
MTn	%FIOMT	magnetic tape.

n is an integer.

If q is present it acts as a qualifier and indicates that a special purpose peripheral routine is required. The actual routines that may be provided will depend on the demand. Possibilities are:

p/q	Peripheral Routine	For
TRn/FIVE	%FIOPT5	5-track tape in ICT code
MTn/EVEN	%FIOEVEN	magnetic tape recorded in even parity.
CRn/IBM	%FIOIBMC	punched cards in IBM code.

If p is omitted the specified peripheral routine is included but no information about an actual peripheral is recorded. It follows that the routine must sort out for itself what peripherals it may use. One very special qualifier is 'NONE', e.g.

OUTPUT 3 = /NONE

This causes the peripheral routine %FIONONE to be included in the program. This routine is a dummy and any output (or input) associated with the programmer's number 3 is suppressed. This feature may be useful during program testing.

If it proves feasible the programmer will be allowed to define his own values of q and associate them with his own peripheral routines. The relevant statement could be:

DEFINE (q, r)

where q is a qualifier
r is the name of a peripheral routine; this may be a string of up to 11 digits and letters starting with a letter.

Then, wherever q appears in a peripheral statement in the Program Description the compiler will arrange to include the routine r. For example, the statements

DEFINE (MINE, MYROUTINE)

INPUT 3 = MT1/MINE

would cause the routine MYROUTINE to be used in association with MT1.

[Question: Can anyone think of a better word than 'DEFINE'?]

Entry Points

All entries to peripheral routines made by %FINOUT are relative to the first word of the routine. It follows that entry points are completely standard and some routines must have entry points that will not sensibly be required; e.g. a paper tape routine must have a Backspace entry. Entry points are as follows ('name' is the name of the peripheral routine):

name	+0	Initiate Read
	+1	Read one formatted block
	+2	Read one unformatted block
	+3	Terminate Read
	+4	Initiate Write
	+5	Write one formatted block
	+6	Write one unformatted block
	+7	Backspace
	+8	Rewind
	+9	Endfile
	+10	All other operations.

Initiate Read sets up any initial conditions required for input. %FINOUT obtains an input record one block at a time by using one of the entries Read one formatted or Read one unformatted block. The peripheral routine signals to %FINOUT when the last block of a record has been read by setting a particular bit in %FIOBUF. The Terminate Read entry reads up to the end of the current record if this has not already been reached.

Initiate Write sets up any initial conditions required for output. %FINOUT passes a record over to a peripheral routine one block at a time by using one of the entries Write one formatted or Write one unformatted block. The peripheral routine terminates a record when a particular bit in %FIOBUF is found to be set.

How a record is divided into blocks is dictated by the buffer size. In practice, a formatted record will usually consist of only one block; in particular, 1 punched card = 1 block = 1 record.

Backspace and Rewind are faults in most peripheral routines. End-file will output an end-of-file record whose form will depend on the output medium.

The effect of entry 10 is decided by an integer parameter in X3 as follows:

0 Release device
1 Regain device
2 Disengage device
3 Runout device

Other operations may be implemented later.

Special actions

For some peripherals some special action is required on the first transfer (e.g. a magnetic tape file must be opened, an extra new line is output on a paper tape punch, etc.) and a bit is available in a peripheral's information block to indicate whether or not a device has previously been referred to.

Availability of Peripherals

Slow peripherals are reserved by the compiler in the program's request slip. Others must be reserved by the peripheral routine when it is entered for the first time.

Checks on whether or not a peripheral is available occur in %FINOUT, so all entries to peripheral routines (except of course 'regain') may assume that a device is available, though not necessarily opened.

K.F. James.

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED

Scientific Programming Department
I.C.T. 1900 Series

FORTRAN NOTE 8
21.6.65.

FORTRAN IV DATA STORAGE

This note describes the way in which data is stored internally for and by a FORTRAN object program.

The information on double-precision (4-word) floating point representation has not been published before. It should be noted that the same representation will be used in EMA and in PLAN subroutines (e.g. the scientific I/O system).

FORTRAN IV DATA STORAGE

1. General

According to the ASA specification, a variable is allocated one or more "storage units", depending upon its type. On the 1900 one storage unit will normally be two 24-bit words, even though one word would suffice for integers. Constants will never have such superfluous words.

2. Numerical and Logical Variables

TYPES OF VARIABLE	NUMBER OF WORDS OCCUPIED
INTEGER	2
REAL	2
DOUBLE PRECISION	4
COMPLEX	4
LOGICAL	2

Note: A switch (set by the program description) exists to arrange that INTEGER and LOGICAL Variables be stored in one word. Certain restrictions will then be imposed upon the language.

INTEGER (2 words)

FIRST WORD : 24-bit signed integer in fixed-point form.

SECOND WORD : ignored.

REAL (2 words)

A floating-point number in the usual 1900 form.

1st WORD :

SIGN	23 BITS OF MANTISSA
BIT 0	1 … 23

2nd WORD :

0	14 MORE MANTISSA BITS	9 BIT EXPONENT
BIT 0	1 … 14	15 … 23

Mantissa = 37 bits + sign bit.

The Stored exponent = true (binary) exponent + 256.

Floating point zero = two words cleared to zero.

Non-zero numbers are normalised (i.e. bit 1 of word 1 = 1).

<u>DOUBLE-PRECISION</u> (4 words)

A floating-point number with the same range as a REAL number but extended precision.

1st and 2nd WORDS : exactly as for REAL.

3rd WORD :

IGNORED	23 FURTHER BITS OF MANTISSA
BIT 0	1 23

4th WORD :

IGNORED	9 FINAL MANTISSA BITS	IGNORED
BIT 0	1 9	10 23

Double-precision mantissa = 69 bits + sign bit.

<u>COMPLEX</u> (4 words)

The real and imaginary parts of a complex number are each held as a single-precision floating point number (as described above under REAL).

WORDS 1 & 2 : Real part

WORDS 3 & 4 : Imaginary part.

<u>LOGICAL</u> (2 words)

A logical variable can assume one of two values, TRUE or FALSE.

1st WORD : TRUE ~~1 followed by~~ 23 zeros. followed by 1

FALSE 24 zeros.

2nd WORD : ignored.

3. Numerical and Logical Constants

These are stored in the same way as variables, except that integer and logical constants will always be one word only.

4. Text data

Text (or Hollerith) constants may appear only as actual arguments in function or subroutine references, and in DATA statements. They will be held as a string of 6-bit characters in 1900 internal code, with spaces added on the right to make an integral number of words.

A variable of any of the types listed in 2 above may be assigned a text value by a function or subroutine call or by a DATA statement. In either case, if the variable consists of more words than the constant, the values assigned to the excess words are undefined. Text values may also be assigned to the first two words of a variable by the A format specification on input.

A. Wilson.
K.F. James.

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED

Scientific Programming Department
I.C.T. 1900 Series.

FORTRAN NOTE 9
21.6.65.

ARRAY ADDRESSING IN FORTRAN IV

This note supersedes the section on array addressing in FORTRAN NOTE 1.

Array Addressing in FORTRAN IV

1. The general basis of array element addressing is the same as for the current FORTRAN on the 1900. Associated with each array is an "Array Header" which contains information about the structure of the array.

Entry points in the Arithmetic Package (%FAP4 - compiled into all FORTRAN IV programs) are provided to calculate the address of the actual array element from the information given in the header and from parameters which give the required subscripts.

For single word or two word array elements the entry point is %FAP4 + 20.

Both entry points require the same parameters as follows:

On entry, X3 = address of array header.

X1 = link.

Other parameters (one per subscript) must follow the CALL instruction and must be instructions which, if obeyed, will place the address of the subscript in X3.

On exit, the address of the desired array element is in X3. The calling routine will be re-entered at the location following the last subscipt instruction.

Array Header

The array header is made up as follows:

1st word	Sign bit	=	0 for 1 word or 4 word elements.
		=	1 for 2 word elements.
	next 5 bits	=	No. of dimensions in array.
	next 18 bits	=	Base address (Address of element 0, 0, -- 0).
2nd word	Address of element 1, 1, 1		
3rd word	Address of 1st word past end of Array.		

This is the array header for a one-dimensional array. For an array of more than one dimension, succeeding words give the partial products of dimensions as follows:

4th word	D_1	≥ 2 dimensions.
5th word	$D_1 \; D_2$	≥ 3 dimensions.
⋮		
last word	$D_1 \; D_2$ --- D_{n-1}	n dimensions.

where D_i is the maximum value the ith subscript may take.

Communication of array information between procedures by means of Formal Parameters.

A procedure (i.e. function or subroutine) is introduced by one of the statement types.

```
SUBROUTINE  X (...., A, ....)
FUNCTION    X (...., A, ....)
```

where 'A' may be a variable, a function name or an Array name. If 'A' is to be an array name it must be included in a statement of one of the following forms:

1. DIMENSION A

2. DIMENSION A (i, j, k, ...)   This form could alternatively appear in a Type or COMMON statement.

If 'A' is introduced by a statement of the first type, then the array is completely defined as to structure and size outside the called procedure.

If 'A' is introduced by a statement of the second type, then the starting address is defined outside the procedure but the array structure and size to be used within the procedure are as defined in this statement.

The dimensions (i, j, k, ...) for an array defined within a procedure may be either unsigned integers or integer variables.

For a dummy array introduced by a statement of the second type, the structure is defined by the values of i, j, ... at the time the subroutine is entered and can change for different uses of the same subroutine.

The structure of such an array is independent of the original definition of that array (in the calling routine). An array must, however, be given its maximum size in its original definition.

Note that only arrays passed as formal parameters of a procedure may have a variable structure.

%FARHD

The structuring of a dummy array is done by means of a FORTRAN library routine called %FARHD.

%FARHD has two entry points

1. Word 0 - to structure a one or two word element array.

2. Word 1 - to structure a four word element array.

On entry to %FARHD:

X3 contains the address at which %FARHD is to generate an array header.

X6 contains the address of a word which contains the address of the header for the original array.

X1 = link (CALL 1 %FARHD or %FARHD + 1)

Following the call to %FARHD is a parameter list:

1st word. number of Dimensions in new array structure.
Following words. - a series of instructions (one per dimension), which if obeyed would place in X3 the address of a word containing that dimension.

On exit from %FARHD, a new array header will have been generated starting at the address originally specified in X3.

The word originally specified by the contents of X6 will now contain the address of the new header.

V.K. Taylor.

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED

Scientific Programming Department
ICT 1900 Series

FORTRAN NOTE 10
31.8.65.

A Proposed Method of Describing and Implementing Program Overlays for the FORTRAN IV (ICT) Compiler.

This note supersedes FORTRAN NOTE 3 of 27.3.65.

A Proposed Method Describing and Implementing Program Overlays for the FORTRAN IV (ICT) Compiler.

General

If there exists a suitable "backing store" for the object program. There will have to exist a facility through which more than one segment of a program may be compiled to occupy the same main store locations, being read from the backing store to the main store when necessary.

The notes which follow give a proposal for program overlay facilities within the framework of FORTRAN IV (ICT) without adding to the FORTRAN Language.

For the purpose of this document the following terms are now defined.

1. Overlay Segment - A program segment which is obeyed from the same area of main store as another segment or segments and can therefore not co-exist in the main store with the other segment or segments.

2. Overlay Unit - A series of Overlay Segments which are always in the main store at the same time.

3. Overlay Area - An area of main store into which a number of Overlay Units may be loaded - At any one time an Overlay Area may contain only one Overlay Unit.

4. Overlay Area Number A small integer which 'labels' a particular Overlay Area.

5. Overlay Unit Number - A small integer which 'labels' a particular Overlay Unit within an Overlay Area.

FORTRAN Overlays

A FORTRAN IV (ICT) program may consist of a main section of program with no overlay segments; or it may contain a main section of program which may not be overlaid and a number of overlay segments in two or more Overlay Units which may be obeyed in one or more overlay areas. No one segment may be obeyed from more than one overlay area. At any time an overlay area contains only one overlay unit.

For FORTRAN IV (ICT) it is proposed that the only segments which may not be overlay segments are SUBROUTINE or FUNCTION segments which have more than 20 formal arguments, MASTER segments and routines which are integrally associated with the compiler (e.g. %FAP4, %FINOUT).

A subroutine or a function segment which has more than 20 formal arguments may not be called from an overlay unit

All Overlay Segments which belong to a particular Overlay Unit are described to the FORTRAN Compiler by a statement in the 'Program Description' of the form.

OVERLAY (a,u) Sub1,Sub2, -----Subk

where 'a' is the Overlay Area Number, and 'u' is the Overlay Unit number within that Area in which the 'Subi' are to co-exist. 'Subi' are the names of the routines which comprise the overlay unit.

(For the basic concept of the 'Program Description' see FORTRAN Note 2. 'A proposed Method of Describing a FORTRAN program to the FORTRAN IV (ICT) Compiler').

Program Communication with Overlay Segments.

Within FORTRAN segments, at the source program level, there is no indication whether a given segment is or is not an overlay segment. Thus, regardless of the 'status' of a segment the programmer will still write.

CALL Sub (Argument list)

to enter the segment 'Sub'; and from within that segment, the statement

RETURN

will cause the original segment to be re-entered at the statement following the CALL statement.

The actual communication between a segment and an overlay segment which is being called proceeds via a special overlay subroutine (%FOVL).

If in a routine 'A' a statement of the form CALL B $(a,---a_k)$ is encountered, where, [one or both of A,] 'B' has been defined previously in an OVERLAY statement, [and if A, B are not routines in the same overlay area, unit] the following sequence is compiled.

```
CALL 1  1  %FOVL
     X_B
     X_A
     N
Standard FORTRAN Calling Sequence for 'B'
```

where X_B indicates the Overlay Area and Ovrlay Unit for which 'B' is defined, and X_A gives the Area Unit for routine 'A'.

X_A and X_B are each one word long and have the following form

top 6 bits - zero
next 8 bits - Overlay Area (Binary Integer)
l.s.10 bits - Overlay Unit (Binary Integer)

'N' gives the number of words in the Standard FORTRAN Calling Sequence for 'B'. This will now be called the Overlay Calling Sub-sequence. 'N' may not be greater than 21.

If A,B are in the same overlay unit, only the Standard FORTRAN calling sequence is compiled

The Operation of %FOVL

This routine ensures that the correct Overlay Unit is in store when a call for an Overlay segment is obeyed. It also keeps track of The Unit from which calls to Overlay Segments have been made so that on execution of a RETURN statement the information is available to ensure that the original calling segment is in store.

%FOVL is basically divided into two parts - that processing the Overlays CALL and that processing the RETURN.

%FOVL uses a push-down list to determine the 'level' of Overlays and to keep the original Overlay Unit and Area numbers and the return address for each Overlay Call. If the overlay call was made from 'permanent' program (non-overlay), then the Overlay Unit and Area numbers are both zero. This list is in common area %FOVC2.

a) Processing the Overlay Call

On entering %FOVL for an Overlay Call, the Overlay Calling Sub-sequence is moved to an area of %FOVL (called %FOVC1) so that the last word in the sub-sequence is stored in the last word of the area. In this way, after the subsequence has been obeyed from this area, a RETURN statement will always return to the same location in %FOVL. (see b) - Processing the RETURN).

After the sub-sequence has been moved, the actual return address in the calling segment is calculated and stored along with the Overlay Area and Unit number for the calling segment in the push-down list (the Overlay Stack).

At this stage %FOVL uses the 'Mark II Overlay Object Program Routine' (M2OPR) to ensure that the Overlay Unit containing the called routine is in store. On exit from M2OPR, the calling sub-sequence is obeyed from the area %FOVC1.

b) Processing the RETURN

All Overlay Calling Sub-sequences are obeyed from the %FOVC1 area such that the return from the called segment is to the same word in %FOVL.

On return to this word from the called segment, %FOVL ensures that the Overlay Unit and Area which is in the top item of the Overlay Stack is in store. (by use of M2OPR) and notes the 'return' address stored in that item. The top item in the stack is then removed and a branch to the return address is obeyed.

The Overlay Stack

The Overlay Stack is stored in the Area %FOVC2 such that the first item put in that stack is stored starting at the first word of the area. Each item which is put in the stack is two words long and is as shown below.

Word 1. Area/Unit for Calling routine.
Word 2. Return Address in Calling routine.

The 'Stack Pointer' (FOSP) initially contains the address of the start of the area %FOVC2. The action of putting an item into the Stack is to store the two words starting at the address given by FOSP and then incrementing FOSP by two.

The action of processing the top item in the stack is to decrement FOSP by two and then look at the item addressed by FOSP. The item is also effectively removed from the stack.

V. K. Taylor.

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED

Scientific Programming Department
I.C.T. 1900 Series

FORTRAN NOTE 10A
9. 9. 65.

Amendment to FORTRAN Note 10

Amendment to FORTRAN Note 10 of 31.8.65.

It has been pointed out that a small modification has to be made to the proposed ØVERLAY system to make it completely general.

Page 2. Insert following the last paragraph

A SUBROUTINE or FUNCTION segment which has more than 20 formal arguments may not be called from an overlay segment.

Page 3. Delete paragraph "If in routine 'A' a statement ------ following sequence is compiled" and insert the following.

If in a routine 'A' a statement of the form CALL B(a_i, -- a_k) is encountered when one or both of A, B have been defined previously in an ØVERLAY statement, and if A and B are not routines in the same overlay unit, then the following sequence is compiled.

Page 4. After first paragraph insert.

If 'A' and 'B' are in the same overlay unit only the standard FORTRAN Calling sequence is compiled.

V.K. Taylor

V.K. TAYLOR. MA.

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED

Scientific Programming Department
ICT 1900 Series

Fortran Note 11
20.9.65.

FORTRAN IV LIST SYSTEM STRUCTURE

This note gives in detail the structure of all lists currently in use in The ICT FORTRAN IV Compiler.

The List System in the Compiler - Structure

The 'List System' of the compiler contains three types of list.

1. Local lists are produced and used by the compiler during compilation of a segment; they are lost when compilation of that segment ends.
2. A Consolidated Cue list is produced and used by the Consolidator and exists throughout the compilation of a complete program.
3. An Overlay list is produced during the Compilation of the Program. It is used by the Compiler during the compilation of each Program segment and therefore exists throughout the compilation of the complete program.

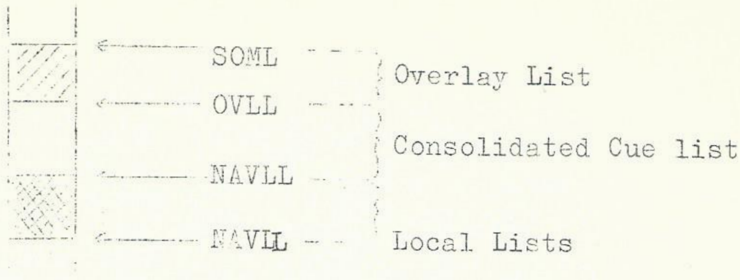
These Lists are stored in the top of store in the following order

1. Overlay List
2. Consolidated Cue List
3. Local Lists

In the compilation of a complete program, the Overlay List is the first list generated. The list is completed before any Consolidated Cue List is produced. The list is completed before any Consolidated Cue List is produced. The Overlay list is started at a location SOML. The Overlay List pointer OVLLL points to SOML at the start of the generation of the Overlay List and points to the word following the end of the final Overlay list item inserted by the time it is required to start forming the Consolidated Cue List. The Consolidated Cue List Pointer NAVLL is set to the value of OVLLL before the first item is generated for that list.

The Consolidated Cue list grows during the Compilation of a complete program, but does not grow during the list of the Local Lists for any segment. Because of this, the Local Lists for any segment can start at the current value of NAVLL (which points to the word following the end of the last item put into the Consolidated Cue List). The Local Lists Pointer is NAVL and for any segment this is initially set to the current value of NAVLL.

The layout of List Store during the compilation of a segment is shown in the following diagram.



Structure of the Individual Lists.

1. Overlay List.

This is a simple chained list which is addressed through the index word ΘVNDX

Each list item is divided into three parts as follows

Word 1. Control Word. This is a c/m to the previous item put into the list. 'c' is the length in words of the 'name' of the previous item and 'm' is the address of the Control Word in the previous item.

Word 2. Data Word

Rest of Item. 'Name' of Item.

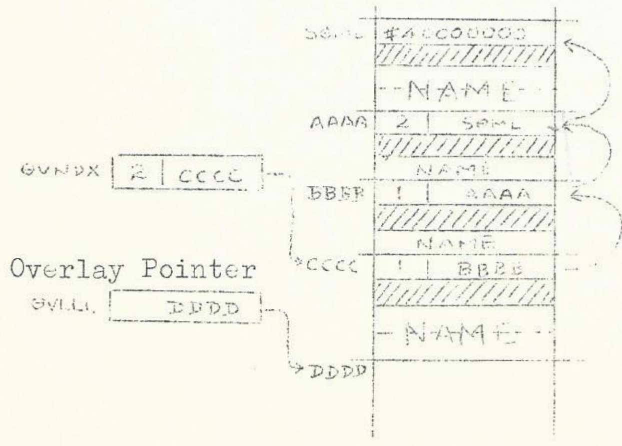
The Index Word is the Control Word for the next item to be put into the list.

The first item put into the list has a control word of #40000000

The Overlay List Pointer is ΘVLLL

The following diagram gives an example of the Overlay List Structure.

Overlay Index Word



2. **Consolidated Cue List.**

This is a simple chained list which is addressed through the index word QNDX. The Consolidated Cue List Pointer is NAVLL

This list is identical in form to the Overlay List except that the Control word for the first item put into the list is zero instead of # 40000000.

3. Local Lists.

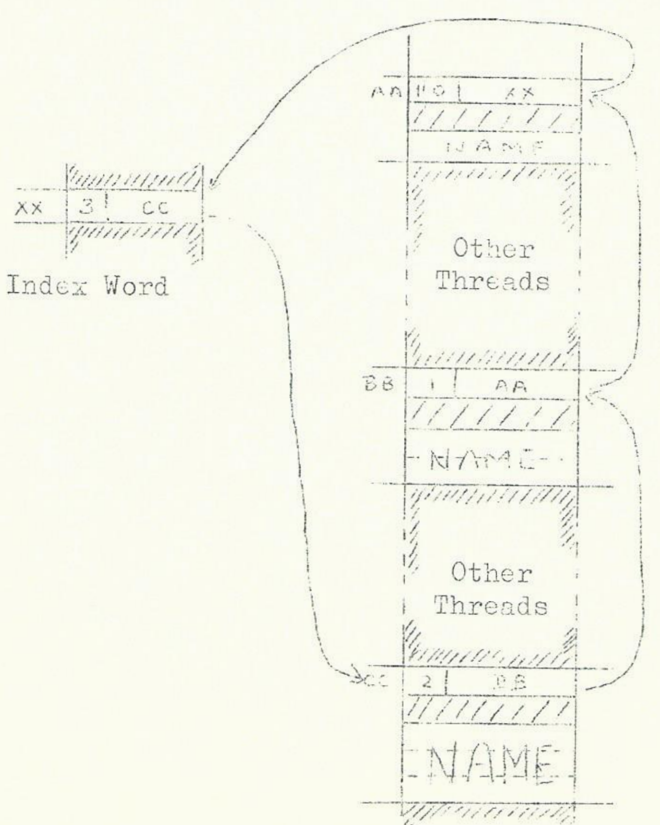
There are three different types of list used in the Compilation of a segment - Main List, Cue List and DQ list.

a). Main List

This is a multiple thread, simple chained list which is addressed through a series of index words (one per thread) starting at INDE. Each complete thread is identical in form to the Overlay List except that the Control Word for the first item put into a thread contains #400 in the counter and the address of the index word for the thread in the modifier.

Although there is one index word per thread there is only one Main List Pointer (NAVL). Thus the items in all threads are intermingled. This is of no consequence as a thread item is always addressed through the thread index word which is connected by address to all items on the same thread.

The following diagram gives an indication of the Thread Structure in the Main List.



The items between the List Items on any thread may belong to other threads of the main List or to DQ Lists

b). Cue List.

This is a simple chained list which is addressed through the index word QFST. The structure of this list is slightly different to that of the other lists described so far.

Each list item is divided into two parts as follows.

Word 1. Control Word. This is a c/m to the next item in the list. If this is the last item put into the list then the control word is zero. 'c' is the length of the name of the next item in the list and 'm' is the address of the Control Word (Word 1) of the next item.

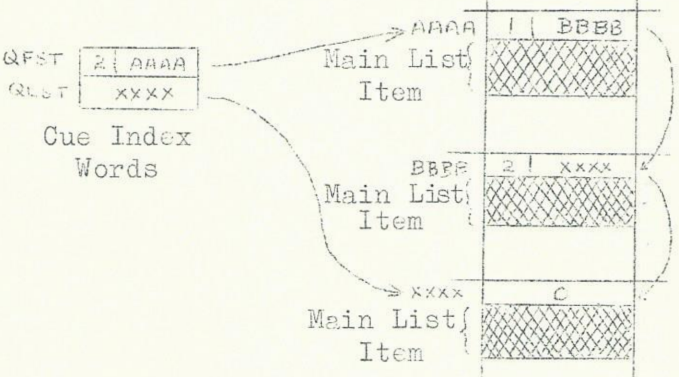
Rest of item. A complete Main List item.

The 'name' of a Cue List item is the same as the name of the Main List item it contains.

The Index Word QFST is the control word for the first item put into the Cue List.

An auxiliary index word QLST contains the address of the last item put into the Cue List, and is used for inserting a new list item.

Since a Cue List item contains a Main List item, after the generation of a Cue List item, NAVL points to the first word following that item. The following diagram gives an indication of the structure of the Cue List.



The Cue List is simply part of the Main List with an additional control word for each item. It is used to generate Cues during the production of the segment Leader. During segment compilation the compiler finds items by using the Main List Index words, not the Cue List index word.

c) DΘ List.

The DΘ List is divided into two independent lists called the Dead DΘ list and the Live DΘ list (DDΘ list and LDΘ list)

Each of these lists is a simple chained list containing 7 word list item, the first word being a control word and the remainder being data words for use by the DO statement section of the Compiler.

The DO lists are the only lists which may fluctuate in size during the compilation of a Segment. At the beginning of a DO statement (or in an implied DO of a READ or WRITE statement) an entry is made in the LDO list. On completion of the DO range this entry is removed from the LDO list and transferred to the DDO list. The basic LDO list item was made in the first place by transferring the last DDO item to the LDO list. If the DDO list was empty, the LDO list item was made starting at the address specified by NAVL.

The Index Word for the DDO list is VLST.

The Index Word for the LDO list is DLST.

Both are initially zero.

The action of putting an item into the LDO list and if necessary deleting an item from the DDO list as follows.

1. If VLST is zero, then the address specified by NAVL is set to the value of DLST and DLST is set to NAVL. NAVL is incremented by 7.
2. If VLST is not zero then the address specified by VLST is set to the value of DLST and DLST is set to VLST. VLST is set to the original contents of the address specified by VLST.

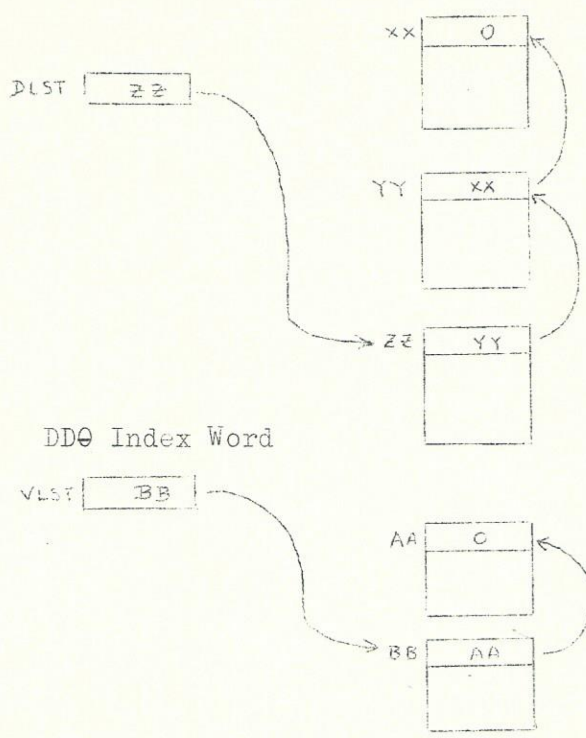
The action of putting an item into the DDO list and deleting the last item put in the LDO list is as follows.

The Contents of VLST is stored in the first word of the list item to be deleted (specified by DLST) and the contents of that word having been saved. The contents of DLST replaces the contents of VLST, and the saved word replaces the contents of DLST.

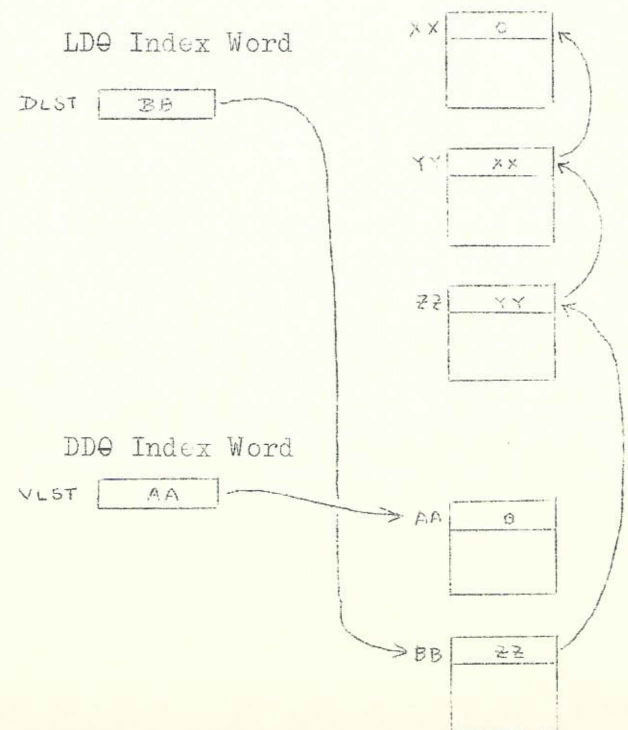
To illustrate the generation of DDO and LDO items consider the examples on the following page.

1. Adding a LDΘ item.
a) Original Situation

LDΘ Index Word

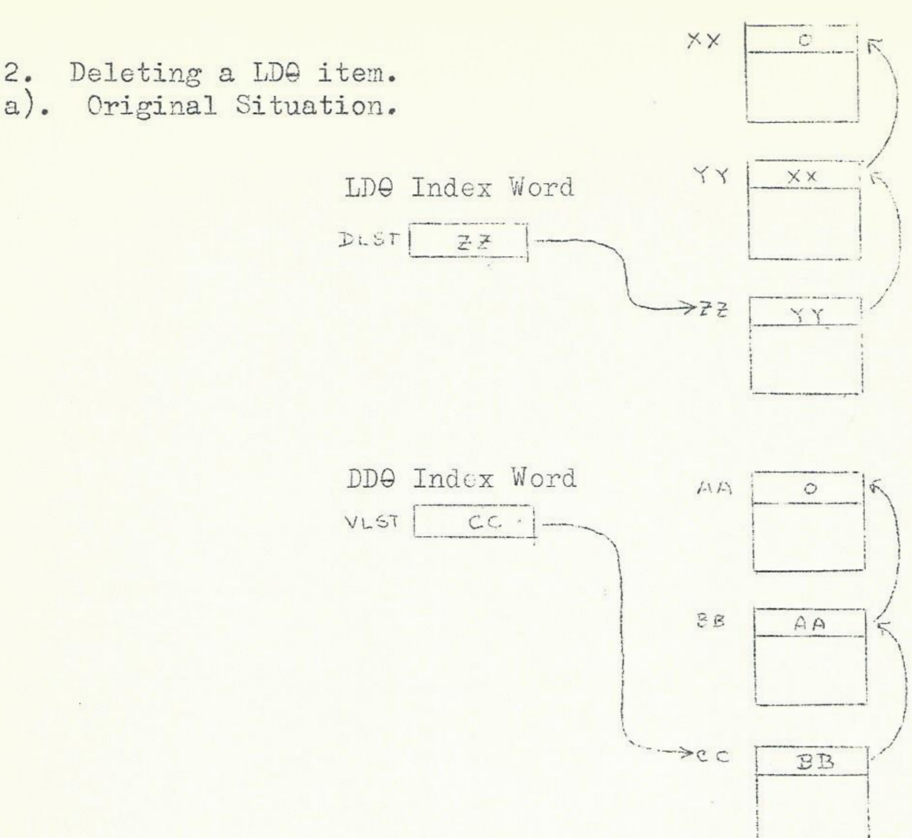


b). After inserting a LDΘ item (and hence Removing a DDΘ item).

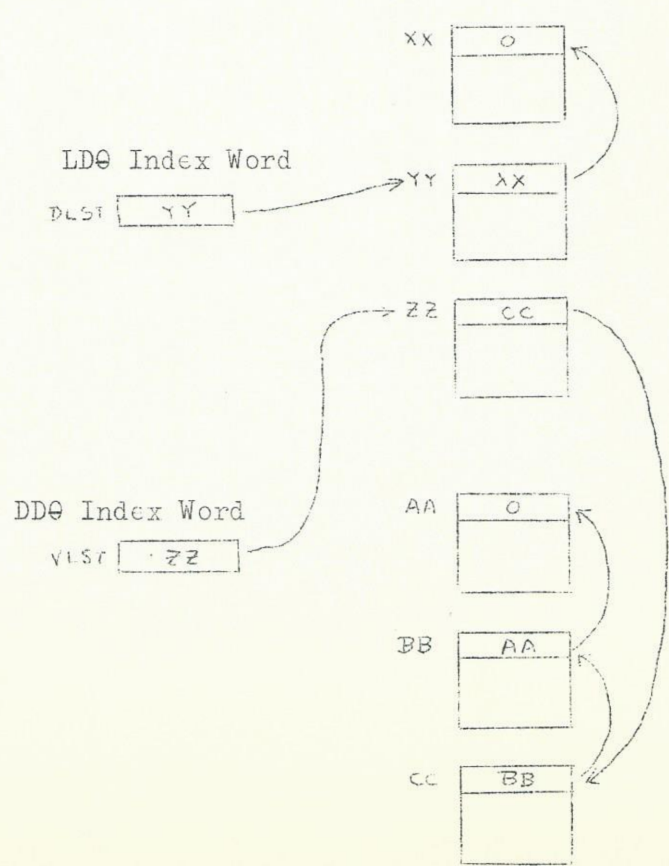


2. Deleting a LDΘ item.

a). Original Situation.



b). After Deleting a LDΘ Item (and hence Insertin, a DDΘ).



Local Lists in the Program Description

The Compilation of the Program Description may involve the generation of some Local and Overlay Lists at the same time. Because of this, the Local lists are not generated in the normal list area but are formed in the 'Polish' area of the compiler normally used for the analysis of arithmetic operations. At the start of the Program Description, NAVL is set to PNPI (the start of this area) which allows this area to be used for temporary list space.

V.K. Taylor

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED

Scientific Programming Dept. — Fortran Note 12

ICT 1900 Series — 20.9.65.

General Analysis Routines in the FORTRAN IV Compiler

This note describes the dictionary search routine, the 'field extract' routine and the List System data used in the FORTRAN IV Compiler

RECOGNITION OF STATEMENT TYPES

When it is required to discriminate among the various statement types, the complete statement is in some area of store (either STATEMENT or CIMAGE) in character form, with a pointer (PØSI) pointing to the first character of the statement name.

For all statements except Assignment statements, Statement Function definitions and some Logical IF statements a routine STS is used to differentiate among the different statement types.

With this routine the first few non-space characters of the statement name are compared against a "Dictionary" to determine the type of statement.

On entry to STS, X2 points to the first character of the Statement name.. On exit, X2 points to the character following the name found and X3 contains a small integer which indicates what dictionary entry was found. If no dictionary entry is found, on exit from STS X2 will be unchanged and X3 will be zero.

e.g. For the statement

DOUBLE PRECISION X

on entry to STS, X2 should be pointing to 'D'. On exit from STS, X2 would be pointing to the character (space or non-space) following 'N'. X3 would be a small integer indicating which dictionary entry was found. Although in theory it is necessary to compare the whole statement name against a dictionary item, in practice only the first few characters of the longer statement names are compared.

For example, the statement

INTEGER X

is recognised as "INTEGER" by searching for a dictionary item INT rather than for the complete name.

Associated with every dictionary item is a number which tells how many non-space characters need to be ignored following recognition to bring the character pointer to the end of the name.

The dictionary therefore consists of two tables - a 'Compare' table, and an associated 'Ignore' table.

These have the following form:

1. Compare Table

   1st word - An Integer which is one less than the number of items in the "Dictionary List". This consists of a series of dictionary items each separated from the next by a space character.

The last item in the list is terminated by a space character. These dictionary items may be any length up to the length for total comparison for the particular statement name.

2. <u>Ignore Table</u>

This is a string of characters (one character per dictionary item) which gives the number of non-space characters of the original statement that are to be ignored following recognition of the dictionary item. The value of the character will be zero if a complete name comparison is made for a particular dictionary item. For example, in the case of INTEGER X the Dictionary item is INT, and the associated Ignore Table character is "4".

The Ignore Table string of characters is stored in the reverse order to the order of items in the Dictionary List, i.e. the character associated with the last item in the Dictionary List is the first item in the Ignore Table and vice versa.

In the FORTRAN Compiler, there are five different dictionaries which are used at different times (i.e. within the Program Description, between program segments, at the beginning of a Program segment, within a MASTER, FUNCTION or SUBROUTINE segment, or within a BLOCK DATA segment).

Before the routine STS is entered to make the dictionary search, the address of the start of the appropriate 'Compare Table' and the address of the start of the associated 'Ignore Table' must be placed in registers LSTP and ISTP of common area LC3.

The value of X3 which is returned upon exit from STS is the number of the entry in the Ignore Table corresponding to the Dictionary Item found. Thus if the item was the last item in the Dictionary List (and therefore the first character in the Ignore Table), the value of X3 would be '1' on exit from STS. If no item is found (i.e. the statement type is not in the current dictionary) the value of X3 on exit from STS would be zero.

X3 on exit from STS can be used as a modifier to branch to the section of the compiler dealing with the type of statement found.

Between the exit from STS and the execution of the appropriate branch instruction, the pointer PØSI is reset to the current value of X2. Thus at the beginning of the analysis of any particular statement (excluding Assignment, Statement Fuction and some Logical IF statements) PØSI is known to point to the first character following the name of the statement.

Assignment statements, Statement Function definitions and Logical IF statements of the form "IF(1) Assignment" are not detected by the STS mechanism.

These statements are distinguished from all the other statements by the fact that they all have an "Equals" character at level zero and do not have a "comma" at level zero anywhere following the equals. Level indicates the number of "("characters which have been found without a corresponding")" character. Thus in the following character string "equals" is at level two.

A + (B + (C = D))

whereas in the following example, "Equals" is at level zero

A = (B + (C + D))

The DO statement is the only statement which has "=" at level zero and also has "," at level zero following the equals.

The "status" of a statement (number of "equals" at level zero and "commas" at level zero following the "equals") is determined at the time the statement is formed in STATEMENT and is available for any statement in a register STAT (of Common area LC3).

For Assignment Statements, Statement Function definitions and Logical IF statements of the form "IF(1) Assignment", the value of STAT is one. For all other statements (except DO) the value of STAT is zero. For DO, the value of STAT is two or three, but this fact is not used in the compiler at present.

Statement Analysis (GETFIELD)

For all statements except FØRMAT, the basic routine used to ascertain the structure of the statement is GETFIELD.

On entry to this routine (at GFE1), PØSI is pointing to the first character of a "field" which is to be found. On exit, PØSI is pointing to the first character following the field terminator.

At this point, the field is in a store area starting at register NAME. The length of the field is held as a counter in the register LENG. The field terminator is in the register TERM (and in X5). The type of field found is given by the value of X4 as follows.

X4 < 0 No field.

In this case, PØSI was not originally pointing to a letter or the start of a number or a logical constant. On exit, TERM holds the character originally pointed at unless that character started a character string with the form of a logical or relational operator (e.g. .AND.) in which case TERM holds the internal single character representation for that operator. PØSI points to the first non-space character following the original character (or logical or relational operator).

X4 = 0 Alphanumeric Field

In this case PØSI was originally pointing at a letter. On exit, the store area starting at NAME will contain the character string starting at the letter and terminating with the first non-sapce, non-alphanumeric character (which will appear in TERM) after that letter. Space characters are ignored in the character string. Thus the field 'AL PH A' is the same as the field 'ALPHA'.

X4 = 1 Floating Point Number

In this case PØSI was pointing at the first character of a character string representing an unsigned floating point number. On exit NAME --> NAME + 1 (and X6, X7) contains the normalised internal representation of that number. LENG is 2 (in the counter position).

X4 = 2 Small Integer

In this case PØSI was pointing to a digit starting a digit string representing a small integer (< 4096). The terminator of this string is not a point (.) unless this starts a logical or relational operator in which case the terminator is the complete operator (in internal single character form).

On exit NAME (and X7) contains the internal binary representation of the integer and LENG is 1 (in the counter position).

X4 = 3 Large Integer.

This is the same as for X4 = 2 except that the value of the integer is in the range 4096 to 8388607.

X4 = 4 Double Precision Number

In this case PØSI was pointing to the first character of a character string representing an unsigned double precision number.

On exit NAME —> NAME + 3 contains the standardized internal representation of that number. LENG is 4 (in the counter position).

X4 = 5 Logical Constant

In this case POSI was pointing to the first character of a character string representing one of the constants.TRUE. or .FALSE. On exit NAME contains the internal representation of that constant (1 for.TRUE., 0 for FALSE). LENG is 1 (in the counter position).

Except for thecase of logical or relational operators or the exponentiation group (**), as terminators, the value of TERM is the internal representation of the non-space character following the end of a field. At the end of a Statement, the terminator is given the value 63 ('E of S' character).

The character strings which act as terminators are given the following values by GETFIELD.

.NOT.	1
.AND.	2
.OR.	3
.LT.	4
.LE.	5
.EQ.	6
.NE.	7
.GT.	8
**	10

Other than by the interpretation of these character strings the characters 1 - 9 cannot appear as terminators in their own right (as they would always be part of either an alphanumeric or numeric field). 10 is the internal value for ":" which may act as a field terminator but which is not basically in the FORTRAN character set.

Statement Analysis (LISTSCAN)

In the analysis of a statement, many fields which are produced by GETFIELD are names of constants or variable or functions which may have been generated earlier in the program or to which reference is expected to be made at a later stage. All such items are stored in the compiler (along with details about the item) in the "Main List" of the Compiler. All basic communication with the Main List is through a series of routines collectively known as "LISTSCAN".

This series of routines is used to determine whether a given item is in the Main List and to make new list items. The entry points to the system are LSE1, LSE5 and LSEA.

For each entry, the name of the item is in an area starting at NAME. The length (in words) of the item name is in the counter position of the register LENG. Normally these parameters will be generated by GETFIELD

The form of a list item is as shown below

Word 1 Control word used by LISTSCAN

Word 2 Data word used by Compiler

Word 3 etc. Name of Item as presented in NAME etc.

Entry LSE1 is used to investigate the existence of a List Item with a particular name (presented in NAME etc.), If X5 is not zero, the investigation starts at the beginning of the Main List. On exit from LISTCAN, X5 is zero, and X3 contains the address of the first word of the item found. If this item is not the item required (correct name but wrong characterstics), a re-entry at LSE1 (with X5 = 0) will continue the search from the point reached by the previous search.

There are two exit points after entry at LSE1:

1. First word after the call if no list of the correct name is found.
2. Second word after the call if a list item of the correct name is found.

LSE5 is used to insert a List item of the correct name after LSE1 has been used previously. The item produced will have a clear data word. X3 will contain the address of the control word for that item.

LSEA is the same as LSE5 except that it is used when the item to be inserted will eventually generate a Cue in the segment leader. (eg. subroutine items).

For most List items the single Data Word is sufficient to hold all the pertinent information about the item. However, for certain classes of items, for example Array names, one word is not sufficient to hold all the required data. In these cases a List Item "Fray" is made (starting at a location given by NAVL) which is long enough to contain the required information. The modifier part of the Data Word of the original List Item is then set to NAVL, and NAVL incremented by the length of the Fray.

Frays are not made within the routines of LISTSCAN but by the sections of the compiler which recognize the item data requirements (such as the sections processing the DIMENSION statement).

The List Item Data Word

The Data Word of a List Item, if not zero, is divided into three parts as shown in the following diagram

3 Bits	6 bits	15 bits
A	B	C

'A' indicates the class of item eg. whether it is a variable or an array or a constant or a label etc.

'B' is a qualifier for 'A' giving the particular characterstics of the type for the item. eg. for a Variable - whether it is of mode INTEGER or REAL etc. and whether it is a COMMON variable or a DATA or a standard variable etc.

'C' is an address which is interpreted according to the values of A and B. eg. for an array, it must be the address of the Fray which contains the rest of the data about the array.

When a List item is initially made by LISTSCAN (using entry LSE5 or LSEA) the data word is left clear. Any information which is to be put into the data word is done from the sections of the compiler which are defining the item.

The data word of a List Item may change progressively throughout a compilation as more is discovered about the item. For example consider the quantity X in the following series of statements

```
INTEGER    X
DIMENSION X (10)
COMMON     X
```

Before the statement INTEGER is encountered there is no List Item for 'X'. In INTEGER, a list item will be made for X with 'Type' marked as 'Undefined' and with a qualifier marked 'INTEGER'.

In DIMENSION, the 'Type' will be changed to 'Array, unassigned' and a Fray will be made for the Item. The address part of the Data Word will be the address of the Fray.

In COMMON, the Type will be changed to 'Array, assigned' and the qualifier will be marked as 'COMMON item' as well as 'INTEGER'.

At this stage, the Item X is completely defined and can not be changed during the remainder of the compilation.

The details of the construction of the Data Word for any list item is given on the following pages. In these details, 'List Item Address' refers to the address of the Control Word, used by LISTSCAN (ie. Word 1), of that item.

List Item Data Word

|← A →|← B →|← C →| List Item Details |

(0) Item Unknown as to type

A	B	C		List Item Details
000	← 3 →	000	List Item Address	Unassigned as to Type
000	← 3 →	101	Address of Dummy Arg(LW)	Dummy Argument Unassigned

(1) Variables

A	B	C		List Item Details
001	← 3 →	000	Address of Variable(LV)	Normal Variable
001	← 3 →	001	Address of Variable(LC)	DATA variable
001	← 3 →	100	Address of LC referencing Var.	Variable Equivalent to non COMMON Array.
001	← 3 →	101	Address of Dummy Var(LW)	Dummy Variable
001	← 3 →	110	Address of Fray	COMMON Variable

(2) Assigned Arrays

A	B	C		List Item Details
010	← 3 →	000	Address of Fray	Standard Array
010	← 3 →	001	"	DATA Array
010	← 3 →	101	Address of Array Header Add. (LW)	Dummy Array
010	← 3 →	110	Address of Fray	COMMON Array

(3) Unassigned Arrays

A	B	C		List Item Details
011	← 3 →	000	Address of Fray	Unassigned Array
011	← 3 →	101	Addr.of Array Header Add. (LW)	Dummy Array

(4a) Functions and Subroutines

A	B	C		List Item Details
100	← 3 →	000	Address of List Item	Normal Function, Subroutine
100	← 3 →	001	Addr.of Start of routine	Dummy Function, Subroutine
100	← 3 →	010	Address of List Item	Special Intrinsic Function

(4b) Block Names and Other Cued Items

A	B	C		List Item Details
100	X	100	Length of Block	Normal COMMON Block name
100	X	101	Length of Block	BLOCK DATA COMMON Block name
100	X	110	Length of Block	Special Lower Common Preset
100	X	111	Peripheral Value	Peripheral Cue

(5) Statement Function Items.

101	<—3—>	X	0	Address (UP) of Statement Fctn	Statement Function
101	<—3—>	X	1	Address of Argument (LW)	S.F. Argument

(6) Constants

110	<—3—>	X	Address of Constant (LC)	Constant

(7) Statement Labels

111	00	<—4—>	Address of Label in Program	Program Label
111	01	X	Address of 'Stepping Stone' (UP)	Label encountered within Statement *
111	10	X	Address of FORMAT Constant (LC)	FORMAT Label
111	11	X	Address of FORMAT Constant (LC)	Label encountered within READ, WRITE **

* This item is made when a label appears within any executable statement other than READ or WRITE if it has not appeared at the beginning of a statement. It is changed to the 'Program Label' item when the label appears at the beginning of a statement (other than FORMAT).

** This item is made when a label appears within a READ or WRITE statement if it has not previously appeared at the beginning of a FORMAT statement. It is changed to a FORMAT label item when the label appears at the beginning of a FORMAT Statement.

For Data Words of Type 0, 1, 2, 3, 4a, 5, and 6, the first three bits of 'B' give the mode of the item as follows:-

1 - Integer
2 - Real
3 - Double Precision
4 - Complex
5 - Logical

The items which require Frays are COMMON variables and Arrays.

A Fray for a COMMON Variable is three words long and has the following form:-

Word 1 Address of a Constant which references the item *

Word 2 Address of Item within COMMON block

Word 3 C/m to COMMON block name **

* For a COMMON Variable in all but a BLOCK DATA segment a constant is required which contains the actual address of the variable. The address of this constant is in the first word of the Fray.
If the COMMON variable is in a BLOCK DATA segment, no constant is required. In this case word 1 contains # 40000000.

** Word 3 must contain the LISTCAN Control word used to find the COMMON Block name Item. It is of the form c/m where 'c' is the length (in words) of the block name, and 'm' is the address of the COMMON block List Item.

Frays for n-dimensional arrays are 4 + n words long and have the following forms.

1. Unassigned Arrays - These are arrays which have yet to be assigned actual space in the object program.

   Word 1 [top 6 bits - Number of Dimensions
   Bottom 18 bits - Address of Array Header in Object Program.

   Word 2 Product of Dimensions (e.g. I.J.K for a I,J,K array)

   Word 3 Sum of Partial Products of Dimension (for an I,J,K array this is 1 + I + I.J)

   Word 4 Zero

   Rest of Fray (n Words) Partial Products (For an I, J, K array, this is 1, I, I.J in the 3 words)

2. Assigned Arrays. These are arrays which have been assigned space in the object program.

   Word 1 top 6 bits - Number of Dimensions
   bottom 18 bits-Address of Array Header in Object Program

   Word 2 Address of 1st element of the Array

   Word 3 1) Zero - standard Array (not COMMON or DATA)
   2) #40000000 - DATA, non-COMMON array
   3) c/m to COMMON block name if a COMMON Array

Word 4 Zero

Word 5 top 6 bits Number of Dimensions

bottom 18 bits Array Base Address *

Rest of Fray (n - 1 words) Partial Products (same as for unassigned array fray e.g. I,I.J for an I,J,K array)

* For an array 'a' defined by a statement of the form 'DIMENSION' $a[d_n]$ the address of an array element $a[s_n]$ is given by the relation

$$a[s_n] = b + q\left(s_1 + \sum_{i=2}^{n}\left(s_i \prod_{j=1}^{i-1} d_j\right)\right)$$

Where 'q' is the number of locations taken by one element (1, 2 or 4 registers) and 'b' is the "Base Address".

<u>V.K. Taylor</u>

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED

Scientific Programming Dept. | Fortran Note 13
ICT 1900 Series | 21.9.65

Storage Areas in FORTRAN IV

The following table indicates the general areas of storage used for certain fundamental items of the FORTRAN language.

Storage Areas Associated with Fundamental Items of the FORTRAN Language

Item Type	Remarks	Area Assigned	Remarks
Variable	Normal	LV	
	COMMON	Common - UV	Addressed through a Constant in LP
	DATA	LP	
	BLOCK DATA	Common - UP	
	Subroutine Argument	LW	
	Statement Function Argument	LW	
	Equivalenced to Array Element		Addressed through a Constant in LP
Array	Normal	UV	Addressed through Array Header in LP
	COMMON	Common - UV	"
	DATA	UP	"
	BLOCK DATA	Common - UP	
	Subroutine Argument		Addressed through Array Header, the address of which is in LW
Constant		LP	
FORMAT	Statement	UP	Addressed through a Constant in LP associated with the FORMAT Label
	Array		Addressed through the second word of the Array Header for the Array

V.K. TAYLOR 19A.

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED.

SCIENTIFIC PROGRAMMING DEPT.
I.C.T. 1900 SERIES

Fortran Note 14
24.9.65

Some Aspects of Expression Evaluation in the FORTRAN IV Compiler

This document attempts to show how expressions in FORTRAN are analyzed. The first part (pages 1-10) introduces the transformation of expressions from standard to Polish notation. This is followed by sections (pages 11-31) which indicate in general terms how the Polish notation equivalent of the original expression is used to define operations which can be separately compiled. This is followed (pages 31-41) by more detailed descriptions of how the operations are represented within the compiler. There follows a description (pages 42-61) of how the actual instructions for these operations are compiled. The document finishes with a short description of the compiler's internal representation of the various items of an expression.

TABLE OF CONTENTS

The Evaluation of Expressions in FORTRAN	1
Generation of PN equivalent for a FORTRAN expression	4
Rules for Generating PN List	4
Unary Algebraic Operations	6
Function and Array References	7
Commas within Expressions	7
Range of Operators	8
Partial PN List - The Operation List	9
The Actual Evaluation of Operations in FORTRAN	11
'Store Auxiliary' Operations	12
Operation Mode - Mode Auxiliary Operation	16
Temporary Store Allocation	20
Operands	27
Relationship between Operators and Operands	27
Summary of Operator Association	30
Practical Compilation	31
Operation Compilation (I)	32
1. Binary Operations	32
1.1 Arithmetic Binary Operations	32
1.1.1 Standard Binary	33
1.1.2 Double Precision Binary	34
1.1.3 Complex Binary	34
1.1.4 Small Integer Binary	35
1.1.5 Standard Exponentiation	35
1.1.6 Small Integer Exponentiation	36
1.2 Logical Binary Operations	36
2. Relational Binary Operation	37
3. Unary Minus	38
4. LOGICAL .NOT.	38
5. Assignment Operations	38
6. Store Auxiliary Operations	40
6.1 Load <u>Acc</u>.	40
6.2 Store <u>Acc</u>.	41
6.3 Load <u>X3</u>	41
6.4 Store <u>X3</u>	41
Group Identifiers	42
Instruction Generator (I)	43
Group Table	43
Position Tables	44
Instruction Tables and Relativisor Tables	47
(Instruction Generator Examples)	48

TABLE OF COMMENTS (Continued)

Restriction on the General Use of the Instruction Generator 55

Instruction Generator (II) 56

Differences between ACΘM and ACΘ2 entries to Instruction Generator 56

Operation Compilation (II) 58
- Function Operations 58
- Intrinsic Functions 59
- Array Operations 59
- Function (Array) Argument (Subscript) Lists 60

Construction of the PN List in the Compiler 62
- PN List Item 62
- Complex Constants 65
- Standard and Intrinsic Functions 66
- Operator List 66

THE EVALUATION OF EXPRESSIONS IN ~~FORTRAN IV~~ Psycle

General A ~~FORTRAN~~ Psycle expression is a sequence of operands, operators and ~~separators~~ bracket followed by a terminator, which in itself constitutes an algebraic, relational or logical expression.

An operand is any constant or any name (e.g. A, JACK, 3·5)

An operator is a symbol from the set

+,-,*,/,** (algebraic)

.EQ.,.NE.,.LE.,.GE.,.LT.,.GT. (relational)

.AND.,.OR.,.NOT. (logical)

= (assignment)

A ~~Separator~~ bracket is a symbol from the set ( ), [,]

A Terminator is the meta-symbol 'E of S'

Under certain circumstances ',' may act as a terminator.

To evaluate any expression it is necessary to determine the order of operations implied in that expression.
For example for the expression

A+B*C

it is necessary to know that B*C is to be evaluated before A is added to the result. On the other hand, for the expression

(A+B)*C

the quantity A+B is to be evaluated and then the result is to be multiplied by C.

For an expression which contains no brackets, the effective order of operations is determined by the relative 'Weights' of the various operators. An operation involving an operator of greater weight will take place before an operation involving an operator of lesser weight if both operations share a common operand. For two operations involving operators of the same weight, where, both operations share a common operand, the left most operation will be performed first.

*(multiply) is defined to have a greater weight than '+' (add).

In the following expression

A + B*C + D*E + F

The order of operations is

1. B*C
2. Result + A
3. D*E
4. Result of 3 + Result of 2
5. Result of 4 + F

For an expression which contain brackets, the effective order of operations is determined as before with the following rule: All operations within brackets are performed prior to any operation outside the brackets which requires the result of the bracketed operation as an operand.

Thus in the expression

A + B*C + (D*E + F)

The order of operations is

1. B*C
2. Result + A
3. D*E
4. Result of 3 + F
5. Result of 4 + Result of 2

Note that A+B*C could be evaluated before D*E+F was looked at, because the result of the bracketed operations was not required for those operations.

To determine the order of operations, it is necessary to scan the expression backwards and forwards to compare operator weights and bracket levels. Within the Compiler this is not an easy or efficient operation either in space or time.

Fortunately the expressions can be fairly easily manipulated within the compiler so that the operations are defined sequentially. In other words, in the manipulated expression the operators are found in the order in which they are required.

For example, the expression

A + B * C + D*E + F

is converted to

ABC* + DE * + F +

and the expression

A + B* C + (D*E+F)

is converted to

ABC* + DE*F + +

Note that in the second case, the brackets have been eliminated from the expression.

The revised expression is in 'post fix' notation. That is at the time of an operation, the operators immediately follow the operands with which they are associated .

To demonstrate this, consider the following converted expression

ABC* + DE* + F +

If at each Stage in evaluating this expression, the result of an operation (R) replaces the operator and operands used in that operation, the following phases occur during the evaluation.

Phase	Post Fix Operation	Resultant Expression	Complete Operation
1.	BC*	A R_1 + DE* + F +	B*C
2.	AR_1 +	R_2 DE* + F +	B*C + A
3.	DE*	R_2R_3+ F +	D*E
4.	R_2R_3+	R_4F +	D*E + B*C + A
5.	R_4F +	R_5	D*E + B*C + A + F

This Particular Post Fix notation equivalent to the original expression is called the 'Polish' Notation (PN) equivalent. It has the characteristics that the operators all appear in order of use, and that no parentheses in the original expression appear in the Polish equivalent.

The Generation of the PN Equivalent of a ~~FORTRAN~~ Expression.

To form the PN Equivalent, the original ~~FORTRAN~~ expression is written element by element into two Lists - the Operand List (PN List) and the Operator List - both initially empty, according to the set of rules given in this section.

The relative 'weights' of the operators for ~~FORTRAN~~ expressions is given in the following table.

Weight	Operators	Type
1 ----	←	Assignment
2 ----	.ØR.	Logical
3 ----	.AND.	
4 ----	.NØT.	
5 ----	.EQ., .NE., .GT., .LE., .LT., .GE.	Relational
6 ----	+, -	Algebraic
7 ----	*, /	
8 ----	~~**~~ ↑	

The numerical 'weights' are meaningless in themselves. However, an operation involving an operator of greater weight will be evaluated before an operation involving an operator of lesser weight if the two operations share a common operand.

The 'top item' of a list is the last item put in the list or currently in the list.

The rules for generating the PN list are set out below

1. Elements are extracted from the original ~~FORTRAN~~ expression in their original order.

2. If the element is an operand it is inserted at the top of the operand list.

3. If the element (e) is an operator, it is compared for weight with the top item (t) in the operator list.

    3.1. If 'e' is greater than 't', then 'e' becomes the top item in the operator list.

an 'opening bracket' ie '(' or '['

3.2. If 't' is '(', then 'e' becomes the top item in the operator list.

3.3. If the operator list is empty, then 'e' becomes the top item in the operator list.

3.4. If 'e' is not greater than 't' and if 3.2 or 3.3 does not apply then 't' is transferred to the top of the operand list. The element is then compared with the new 't' according to the rules 3.1, 3.2, 3.3 and 3.4.

4. If the element is ~~the separator '('~~ an opening bracket it is inserted at the top of the operator list.

5.1 If the element is ')' (a 'closing bracket' ie ')' or ']') then unless 't' is '(' (an opening bracket) or the operator list is empty, 't' is transferred to the top of the operand list and the element is then compared with the new 't' according to this rule.

5.2 If 't' is '(', it is removed from the operator list and the comparison stops.

5.3 If there are no items in the operator list the comparison stops. ')' never goes into either list.

New Paragraph

6. If the element is ',' (as a separator) it acts as an operator of the same weight as '='.

7. If the element is 'E of S' it acts in the same manner as ')' except that the comparison stops only when there are no items in the operator list.

As an example of the generation of a PN equivalent, consider the expression

A + B * (C - D)

This will eventually be converted into

ABCD - * +

by the following steps.

Step	Operation	Rule	List Contents Operand (PNL)	Operation (ΘL)
1	A → PNL	2	A	Empty
2	'+' → ΘL	3.3	A	+
3	B → PNL	2	AB	+
4	'*' → ΘL	3.1	AB	+*
5	'(' → ΘL	4	AB	+*(
6	C → PNL	2	ABC	+*(
7	'-' → ΘL	3.2	ABC	+*(-
8	D → PNL	2	ABCD	+*(-
9	')' compared with ΘL			
	'-' → PNL	5.1	ABCD -	+*(
	'(' removed from ΘL	5.2	ABCD -	+*
10	'E of S' compared with ΘL			
	'*' → PNL	7	ABCD - *	+
	'+' → PNL	7	ABCD - *+	Empty

The final PN Equivalent for the original FORTRAN expression is the final Operand List.

The Unary Algebraic Operations (-A, +A)

The symbol '-' may stand for two different operations, one involving two operands (eg A-B) and one involving one operand (eg - A).

Although these operations can be differentiated with the expression in standard form, this is not true with the expression in Post Fix notation. For example, the expression

X + A *(-B)

would turn into

X AB - *+

As it stands, this is n ot distinguishable from the PN equivalent for The expression.

+ X * (A-B)

where '+' is an operation involving one operand (unary +)

To avoid ambiguity with these two symbols (+ and -), when the PN equivalent for an operation using '-' as a unary operation is generated, the operator is converted to a new operator '$\underset{u}{-}$' before the equivalent is made. This operator has the same weight as '-'.

Thus the PN equivalent for X+A* (-B) is XAB $\underset{u}{-}$ *+

The operator '+' as a unary operator is always redundant and is ignored when making PN equivalents.

Thus the PN equivalent for +X*(A-B) is XAB - *

Function and Array References

Not all FORTRAN expressions involve only simple elements. For example the expression A + SIN (X) contains the complex element SIN(X) as an operand of the '+' operator.

In order to accommodate elements of this type it is necessary to extend the rules and definitions described so far.

In an expression, A complex element of the form Name (- - -) is conceptually converted to a series of simple elements of the form

Name op (------)

before the PN equivalent for the expression is found.

'Name' is the name of a Function or of an Array op is the pseudo-operator f or a depending on whether the complex element is a Function reference (f) or an array reference (a)

For example, the expression A + SIN (X) is conceptually converted to the form A + SIN f (X) before any post-fix notation equivalent is formed.

The operators f and a are of the same weight, and are of greater weight than **.

Thus the PN equivalent form for A + SIN (X) is

A SIN X f +

If B is an array name, then the PN equivalent form for B(I) + SIN(X) will be

BIa SIN X f +

Commas Within Expressions

Commas may occur in three places within FORTRAN expressions (including assignment statements) as follows.

1. As a separator on the l.h.s. of an assignment statement

   eg. A,B,C = X

   The PN equivalent of this statement is

   AB,C, X=

2. As an argument separator in a function reference

   eg FCN (X,Y,Z)

   The PN equivalent of this function reference is

   FCN XY, Z, f

3. As a subscript separator in an array reference

   eg ARRAY (I,J,K)

   The PN equivalent of this array reference is

   ARRAY IJ,K, a

The 'Range' of Operators

In a normal FORTRAN expression the 'range' of an operator is intuitively obvious. Thus in the expression A+B*COS(C), the range of the + is the operand A and the result of the operation B*COS(C). The range of * is the operand B and the result of the operation COS(C). The range of the pseudo-operator f is the name COS and the single argument C.

The same expression in PN form is AB COS C f *+

Breaking this down into operations, replacing an operation in the list by its result,

0. AB COS C f * +
1. AB R_1*+ $\quad$ R_1 = COS(C)
2. AR_2+ $\quad$ R_2 = B*COS(C)
3. R_3 $\quad$ R_3 = A+B*COS(C)

it can be seen that at the time of a particular operation, the operands for that operation are in standard positions relative to the operator. The positions of these operands determine the 'Range' for the operator.

There are four basic 'ranges' for operators as shown below.

1. Binary form. a b op where a and b are operands.
   op is one of the algebraic, relational or logical operators (except unary minus (-u) or .NOT.)
2. Unary form a op
   op is one of the operators -u, or .NOT.
3. 'Function, Array' form.
   a) Single argument (subscript) form
      F a op
   where F is the Function or Array name
      a is the argument
      op is the pseudo-operator f or a
   b) Multi-argument (subscript form)
      Fa_1a_2, a_3, - - -, op.
      For a two argument (subscript) form, this reduces to
      Fa_1a_2, op
4. 'Assignment' Form
   a) Simple assignment
      ae+ =
   b) multiple assignment
      a_1a_2, a_3, - - - e =

where a, a_1, -- etc. are l.h.s. operands in the Assignment Statement. e is the r.h.s. operand in the Assignment Statement.

Knowing the operator, it is easy to determine the operands for any operation.

The Partial PN List - The Operation List

So far it has been shown how the complete PN equivalent can be generated for a FORTRAN expression.

In that the PN equivalent generates operators in their order of use, and in that all operands for an operator are already in the PN list when the operator is put into the list, it follows that for each operation all the information required for evaluating that operation is available at the time that the operator is transferred from the operator list to the PN list and that it is in fact not necessary to form the complete PN equivalent before starting to evaluate the FORTRAN expression.

In the FORTRAN Compiler, a partial PN list - the Operation List - is generated. This only contains operands and commas. When an operator would otherwise be passed to the PN list, it is used immediately to evaluate the operation. The result operand for the operation replaces all operands and commas used in the operation.

As an example of this Partial PN list generation, consider the evaluation of the expression A+B*(C-D). The complete PN evaluation has been given previously.

Step	Operation	Rule	List Contents	
			PNL	θL
1	A -> PNL	2	A	Empty
2	'+' -> θL	3.3	A	+
3	B -> PNL	2	AB	+
4	'*' -> θL	3.1	AB	+*
5	'(' -> θL	4	AB	+*(
6	C -> PNL	2	ABC	+*(
7	'-' -> θL	3.2	ABC	+*(-
8	D -> PNL	2	ABCD	+*(-
9	')' compared with θL			
	'-' -> PNL	5.1		
	(evaluate CD-)		ABR_1	+*(
	'(' removed from θL		ABR_1	+*
10	'E of S' compared with θL			
	'*' -> PNL	7		
	(evaluate BR_1*)		AR_2	+
	'+' -> PNL			
	(evaluate AR_2+)		R_3	Empty

R_1, R_2 and R_3 are the 'Result' operands for the evaluation of the operations CD -, BR_1* and AR_2+
(i.e. C - D, B*(C - D), and A + B * (C - D)).

The Actual Evaluation of Operations in FØRTRAN

In the previous example, it was indicated that the evaluation of the expression required three operations, namely C-D, $B*R_1$, and $A+R_2$ where R_1 was the result of the operation C-D, and R_2 was the result of the operation $A+R_2$. The result of the overall expression was R_3.

The meaning of R_1, R_2, and R_3 depend on the way arithmetic operations are compiled by the FØRTRAN compiler.

For the purpose of compilation, a 1900 series computer is considered to be a single accumulator (ACC), one index register (X3) machine.

ACC when defined as an operand or as the result of an operation contains a 'value'.

X3 when defined as an operand or as the result of an operation contains the 'address' of some storage area which contains a 'value'.

The basic operations defined so far have the following characteristics with regard to this assumption.

1. Binary Operations a b op

   Either 'a' or 'b' is ACC. The other operand is in store or is X3.

   The Result Operand is ACC.

2. Unary Operations a op

   'a' is ACC
   The Result Operand is ACC

3a. Function, Operation Fa_1, a_2,---, f

   F is Function Name
   a_1, a_2 ---- etc. may not be ACC or X3
   The result operand is ACC

3b. Array Operation $As_1 s_2$,--, a
A is the Array Name
s_1, s_2, ---- etc. may not be ACC or X3
The result operand is X3

4. Assignment Operation $a_1 a_2$,--, e=
a_1,a_2 may not be ACC
e is ACC
The result operand is ACC
although the actual result is a store location. (This result operand is rarely used)

Store Auxiliary Operations

Since an expression involves a series of operations, the result of one being an operand for the next, it can be seen that there is no guarantee that at the time it is required to perform an operation, the appropriate operands will be in the required form. In particular, unless the first operation is a Function Operation, at least one of the operands will be of the wrong form for the operation.

It is therefore necessary to define certain auxiliary operations which if the operands are of the wrong form may have to take place before the main operation.

The auxiliary operations to be defined are lac, lx3, sac, sx3. These have the following characteristics.

lac - Load ACC

Single Operand not ACC
Result Operand is ACC

lx3 Load X3

Single Operand is Tx or 'Store' Operand
Result Operand is X3

sac   Store ACC

Single Operand is ACC
Result Operand is Ta

sx3   Store X3

Single Operand is X3
Result Operand is Tx

Ta is a Temporary Storage Area used to store ACC if it is not required for the current operation.

Tx is a Temporary Storage Area used to store X3 if it is not required for the current operation.

This set of operations are the 'Store Auxiliary' operations of the Compiler.

The operations lx3 and sx3 do not destroy ACC

The operations lac and sac do not destroy X3

Consider again the example A+B*(C-D)

This had previously been broken down into the operations as shown below.

	Operation List	Operation Performed	Total Operation
1.	ABCD-	$C-D \rightarrow R_1$	C-D
2.	ABR_1*	$B*R_1 \rightarrow R_2$	B*(C-D)
3.	AR_2+	$A+R_2 \rightarrow R_3$	A+B*(C-D)

The FORTRAN Compiler treats this in more detail according to the rules governing operands for the various operators as follows

	Operation List	Operation Performed	Total Operation
1.	ABCD(lac)-	D → ACC	D
2.	ABCACC -	C - ACC → ACC	C-D
3.	ABACC*	B*ACC → ACC	B*(C-D)
4.	AACC+	A+ACC → ACC	A+B*(C-D)

The operator lac was used to put the operands into a standard form for the binary operation '-'. It operates on the top item in list, namely D.

Consider the following example

A*B + C*D

The FORTRAN Compiler breaks this down to the following set of operations.

	Operation List	Operation Performed	Total Operations
1.	AB(lac)*	B → ACC	B
2.	AACC*	A*ACC → ACC	A*B
3.	ACC CD (sac)(lac)*	ACC → Ta_1	A*B → Ta_1
4.	Ta_1 CD (lac)*	D → ACC	D
5.	Ta_1 C ACC*	C*ACC → ACC	C*D
6.	Ta_1 ACC +	Ta_1 + ACC → ACC	A*B + C*D

At steps 1 and 4, the operator lac is used to produce an operand in the standard form for the binary operation*.

At step 3 it is necessary to use the operator sac to store ACC in a temporary storage location Ta_1 becuase, although ACC is in use it is not one of the operands for the current operation.

The operator sac does not work on the top item, but on the item which is ACC. The item is replaced by the result operand Ta (Ta_1 in this case).

The fact that sac works only on the item which is ACC, and that there may not be two ACC items in the list at the same time requires (and enables) the compiler to keep a pointer to the current list item for ACC. If no item is ACC, then this pointer is clear.

Consider the following example

A(I) + B*C   where A is an array name.

The FØRTRAN Compiler will break this down to the following set of operations.

	Operation List	Operation Performed	Total Operation
1.	AIa	A(I) → X3	A(I) in X3
2.	X3 BC (lac)*	C → ACC	C
3.	X3 B ACC*	B*ACC → ACC	B*C
4.	X3 ACC +	A(I) + ACC → ACC	A(I) + B*C

Here the operation B*C is defined as not requiring or destroying X3 so there is no requirement for a SX3 operation at step 2.

On the other hand in the expression

A(I) + F(X)   where A is an array name and F is a Function.

The following operations are compiled.

1.	AIa	A(I) → X3	A(I) in X3
2.	X3 FX sx3 f	X3 → Tx_1	X3 → Tx_1
3.	Tx_1 FX f	F(x) → ACC	F(x)
4.	Tx_1 ACC +	Tx_1 + ACC → ACC	A(I) + F(X)

In this example, the Function operation is defined as possibly destroying X3 which, since it holds an operation result, must be stored using the operator sx3. Note that as for the operator sac the operator sx3 does not work on the top item but on the item which is X3. Therefore as for sac, the compiler is required to keep a pointer to the current list item for X3. If no item is X3 then this pointer is clear.

Operation Modes - Mode Auxiliary Operations

In FORTRAN, an operand of an expression will be defined as having one of five 'modes' (INTEGER, REAL, DOUBLE PRECISION, COMPLEX or LOGICAL)

As the actual representation in the compiled program of an operand depends on the operand mode, and the instructions compiled for an operation depend on the mode of the operands, it is necessary to give the operand mode information in the Operation List. The mode of the result of an operation is defined by the modes of the original operands and by the operator. The result mode is not necessarily the same as that of the operands.

The modes of the operands have to be in a standard form for an operation to be defined.

The standard operand modes are given in the following table.

1 a. Binary Operators +,-,*,/ ab<u>op</u>

'a','b' must be of the same mode (not LOGICAL)
The Result is of the same mode as the operands

b. Binary Operator ** ab**

'b' must be INTEGER if 'a' is INTEGER or COMPLEX
'b' must be REAL or INTEGER if 'a' is REAL or DOUBLE PRECISION
Neither 'a' nor 'b' may be LOGICAL
The Result is the mode of 'a'

c. Binary Operators .EQ.,.NE.,.LE.,.GE.,.LT.,.GT. ab<u>op</u>

a,b must be of the same mode (not Logical)
The Result is LOGICAL mode

d. Binary Operators .AND.,.OR. ab <u>op</u>

a,b must be LOGICAL mode
The Result is LOGICAL mode

2. a. Unary Operator '$-_u$' $a-_u$

'a' must be any mode except LOGICAL
The Result is the same mode as 'a'

b. Unary Operator .NOT. a.NOT.

'a' must be LOGICAL mode
The Result is LOGICAL mode.

3. a. Function Operator <u>f</u> Fa_1---<u>f</u>

a_1,--- may be any mode
The Result is the mode of F

b. Array Operator <u>a</u> As_1---<u>a</u>

s_1,--- must be INTEGER mode
The Result is the mode of A.

4. Assignment Operator = a_1,a_2,---,e=

The modes of a_1,--- and e are the same.
The Result is the mode of a_1,

If the modes for the operands in an operation are not standard certain auxiliary operations may have to be performed to make the operand modes standard before the main operation can be performed. Certain combinations of modes may produce an undefined operation. This will cause the Compiler to indicate an Error.

Consider the following examples.

The subscripts R, I, and D indicate that the operand is REAL, INTEGER, or DOUBLE PRECISION.

Example 1. $A_R + B_R$

In this operation the modes are both the same and standard for the operator '+'. Therefore the operation is "immediately definable" (after doing the auxiliary lac operation on B_R)

1. $B_R \rightarrow \underline{ACC}_R$ (lac)

2. $A_R + \underline{ACC}_R \rightarrow \underline{ACC}_R$ +

Example 2. $A_R + B_I$

In this operation the modes are not the same and therefore are not standard for the operator '+'. The operation (after having done the lac operation) is not 'immediately definable'. An auxiliary operation 'Float ACC' (fa) must be performed on B before the operation becomes standard.

The fa operation takes an INTEGER operand in ACC and returns a REAL result in ACC

1. $B_I \rightarrow \underline{ACC}_I$ (lac)

2. $\underline{ACC}_I \rightarrow \underline{ACC}_R$ (fa)

3. $A_R + \underline{ACC}_R$ +

Example 3. $A_D + B_I$

Once again an auxiliary operation is required to bring the operands to a standard form. In this case, two auxiliary operations are required, fa and da.

The da operator takes a REAL operand in ACC and returns a DOUBLE PRECISION result in ACC.

1. $B_I \rightarrow \underline{ACC}_I$ (lac)

2. $\underline{ACC}_I \rightarrow \underline{ACC}_R$ (fa)

3. $\underline{ACC}_R \rightarrow \underline{ACC}_D$ (da)

4. $A_D + \underline{ACC}_D$ +

The two auxiliary operations were required to put 'B' in Double precision form. In theory, only one operation is required (INTEGER → DOUBLE PRECISION) but such an operation is not in the set of operations currently available as auxiliaries in the compiler.

The mode changing auxiliary operations used by the compiler are outlined below. They all use one operand and give one result.

Operator	Operand	Result	Remarks
fa	$\underline{ACC}_I$	$\underline{ACC}_R$	Float INTEGER ACC
fs	$\underline{X3}_I$	$\underline{X3}_R$	Float Integer Quantity not in ACC
da	$\underline{ACC}_R$	$\underline{ACC}_D$	Convert REAL ACC to Double Precision.
ds	$\underline{X3}_R$	$\underline{X3}_D$	Convert REAL Quantity, not in ACC, to Double Precision.
ca	$\underline{ACC}_R$	$\underline{ACC}_C$	Convert REAL ACC to Complex
cs	$\underline{X3}_R$	$\underline{X3}_C$	Convert REAL Quantity, not in ACC, to Complex
ifa	$\underline{ACC}_R$	$\underline{ACC}_I$	Convert REAL ACC to Integer
ida	$\underline{ACC}_D$	$\underline{ACC}_R$	Convert DOUBLE PRECISION ACC to Real.

The operations using fa, da, ca, ifa and ida do not destroy X3.
The operations fs, ds and cs do not destroy ACC.

Temporary Store Allocation

Consider the compilation of the following Assignment statement.

X = A*B+SQRT(X+Y)*(X-Y)

It is assumed that the mode of all items is the same, and is REAL.

The 'Compilation Table' which follows is divided into the following Columns.

1. Step. This gives the number of main and auxiliary operations which have been done (including the current operation).

2. Markers. These give the values of the X3 and ACC marker. They contain the addresses in the PN List in which the X3 operand and the ACC operand appear. If X3 or ACC do not appear in the PN List, then the value of the associated marker will be zero.

3. Op This gives the main operator which is being processed.

4. PN List. This contains the items which are in the Operand List at the time of the start of a main or auxiliary operation.
The sequence of numbers along the top give the "address" of the items in the List, and are, in the Compilation Table, associated with the values in the Markers.

5. Aux Op. This gives the auxiliary operation which is currently being performed. '-' in the field indicates that the main operation is being done.

Original Statement.

X = A*B+SQRT(X+Y)*(X-Y)

Step	Markers X3	Markers ACC	Op	PN List 1	2	3	4	5	6	Aux Op	Comments
1	0	0	*	X	A	B				lac	B → ACC
2	0	3	*	X	A	ACC				–	A*B → ACC
3	0	2	+	X	ACC	SQRT	X	Y		sac	ACC → Temporary Storage 1
4	0	0	+	X	Ta_1	SQRT	X	Y		lac	Y → ACC
5	0	5	+	X	Ta_1	SQRT	X	ACC		–	X+Y → ACC
6	0	4	f	X	Ta_1	SQRT	ACC			sac	ACC → Temporary Storage 2
7	0	0	f	X	Ta_1	SQRT	Ta_2			–	SQRT(X+Y) → ACC
8	0	3	–	X	Ta_1	ACC	X	Y		sac	ACC → Temporary Storage 3
9	0	0	–	X	Ta_1	Ta_3	X	Y		lac	Y → ACC
10	0	5	–	X	Ta_1	Ta_3	X	ACC		–	X-Y → ACC
11	0	4	*	X	Ta_1	Ta_3	ACC			–	SQRT(X+Y)*(X-Y) → ACC
12	0	3	+	X	Ta_1	ACC				–	A*B+SQRT(X+Y)*(X-Y) → ACC
13	0	2	=	X	ACC					–	X=A*B+SQRT(X+Y)*(X-Y)

The original statement breaks down into 13 separate immediately definable operations. Certain of these operations involve the use of Temporary Storage Areas to store intermediate results (Steps 3, 6 and 8). In the table these areas have been called Ta_1, Ta_2 and Ta_3.

The size of a storage area depends on its use. The 'Ta' areas are 2 or 4 words long depending on the mode of the operand for the sac operation. The 'Tx' areas are 1 word long regardless of the mode of the operand for the SX3 operation.

In compiled program these areas occupy a single area of store (LW area) and therefore a working store assignment rule is required by the compiler. A marker (CWSA) is required to determine the address of the next Temporary area which may be assigned.

A simple Temporary store assignment rule would be following.

1. When a Temporary store is required, the address of the area is given by the current value of CWSA. The value of CWSA is then incremented by the size of the area required.

By this rule, the following Temporary store assignment would be given in the previous example. Assuming that the original value of CWSA was To, and that each 'Ta' requires 2 locations,

Then $Ta_1 = To$
$Ta_2 = To + 2$
$Ta_3 = To + 4$

Thus 6 Temporary store locations are required for the statement.

This is obviously extravagant in storage space, and in a complicated assignment statement could lead to considerable waste space.

Due to the rules for arithmetic used in the current FORTRAN Compiler, a Temporary store location is only used in one operation and can therefore be reused once that operation has been processed.

In the previous example, the Temporary area Ta_2 was made at Step 6 and was used in the operation of Step 7. It then follows that the area Ta_3 generated at Step 8 can be the same as area Ta_2.

The rules for Temporary Store Assignment actually used in the compiler as follows.

1. When a temporary store is generated by an auxiliary operation, the address of the area is given as the current value of CWSA. The value of CWSA is then incremented by the size of the area.

2. When a temporary store is used in an operation, CWSA is given the address of that temporary store.

Following again the previous example, the value of CWSA after the various steps is given below.

Start.	CWSA = To		
Step 3	CWSA = To + 2	Ta_1 = To	
Step 6	CWSA = To + 4	Ta_1 = To	Ta_2 = To + 2
Step 7	CWSA = To + 2	Ta_1 = To	
Step 8	CWSA = To + 4	Ta_1 = To	Ta_3 = To + 2
Step 11	CWSA = To + 2	Ta_1 = To	
Step 12	CWSA = To		

At the end of evaluation of an expression the value of CWSA should be the same as it was at the start. Only in certain peculiar circumstances (generation of Statement Functions) is this not the case.

The rules stated above for Temporary Store assignment will only work if one can guarantee to use the Storage areas in the inverse order to which they were assigned. It can be seen by an example that with the auxiliary operations as described to this point that this is not necessarily the case.

In the following example, a Temporary Store item is designated by 'T' followed by 'a' or 'x' followed by an integer.

'a' indicates that the Temporary Store is holding a value and was generated by a <u>sac</u> operation
'x' indicates that the Temporary Store is holding an 'address' and was generated by a <u>SX3</u> operation.

The Integer indicates the order of assigning Temporary Storage areas.

Example

Original Expression (Assignment Statement)

Y = A(I) + (B + C) * ((D + E) * (A(J) + F) + ((G + H) + FN(X)))

The main operations which are used have the following characteristics with regard to ACC and X3

Array Op. (a)	destroys ACC and X3	leaves result as X3
+,*,=	destroys ACC	leaves result as ACC
Function op (f)	destroys ACC and X3	leaves result as ACC

In this example 'A' is an array name. 'FN' is a function name

Step	Markers		op	PN List								Aux	Comments
	X3	ACC		1	2	3	4	5	6	7	8	Op	
1	0	0	a	Y	A	I						–	A(I) → X3
2	2	0	+	Y	X3	B	C					lac	C → ACC
3	2	4	+	Y	X3	B	ACC					–	B+C → ACC
4	2	3	+	Y	X3	ACC	D	E				sac	ACC → Ta_1
5	2	0	+	Y	X3	Ta_1	D	E				lac	E → ACC
6	2	5	+	Y	X3	Ta_1	D	ACC				–	D+E → ACC
7	2	4	a	Y	X3	Ta_1	ACC	A	J			sac	ACC → Ta_2
8	2	0	a	Y	X3	Ta_1	Ta_2	A	J			sx3	X3 → Tx3
9	0	0	a	Y	Tx3	Ta_1	Ta_2	A	J			–	A(J) → X3
10	5	0	+	Y	Tx3	Ta_1	Ta_2	X3	F			lac	F → ACC
11	5	6	+	Y	Tx3	Ta_1	Ta_2	X3	ACC			–	A(J) + F → ACC
12	0	5	*	Y	Tx3	Ta_1	Ta_2	ACC				–	(D+E) * ACC → ACC
13	0	4	+	Y	Tx3	Ta_1	ACC	G	H			sac	ACC → Ta4
14	0	0	+	Y	Tx3	Ta_1	Ta_4	G	H			lac	H → ACC
15	0	6	+	Y	Tx3	Ta_1	Ta_4	G	ACC			–	G + H → ACC
16	0	5	f	Y	Tx3	Ta_1	Ta_4	ACC	FN	X		sac	ACC → Ta_5
17	0	0	f	Y	Tx3	Ta_1	Ta_4	Ta_5	FN	X		–	FN(X) → ACC
18	0	6	+	Y	Tx3	Ta_1	Ta_4	Ta_5	ACC			–	(G+H) + FN(X) → ACC

Assuming the initial value of CWSA was To, and that Tx areas occupy 1 word and that Ta areas occupy two words, the following Temporary area allocations would be made at the end of the steps shown.

Start	CWSA = To			
Step 4	CWSA = To + 2	Ta_1 = To		
Step 7	CWSA = To + 4		Ta_2 = To + 2	
Step 8	CWSA = To + 5			Tx_3 = To + 4
Step 12	CWSA = To + 2		Ta_2 used	
Step 13	CWSA = To + 4		Ta_4 = To + 2	
Step 16	CWSA = To + 6			Ta_5 = To + 4

It can be seen that the sac operation of step 16 assigned the same starting address to Temporary area Ta_5 as to the Temporary area Tx3. Thus the information stored at step 8 is lost.

The problem arises because the working store areas are not used in the inverse order to which they were assigned (eg. Ta_2 was used before Tx3).

The problem is solved in the FØRTRAN compiler by making the sac operator and the sx3 operator slightly more complex as follows.

1. sac — The operand is the PN list item pointed to by the ACC marker. If the X3 marker is set to a smaller PN list address than the ACC marker at the time a sac operation is to be performed, a sx3 operation is performed first.

2. sx3 — The operand is the PN list item pointed to by the X3 marker. If the ACC marker is set to a smaller PN list address than the X3 marker, at the time a sx3 operation is to be performed, a sac operation is performed first.

In the previous example, this is shown as follows.

Step	Markers		Op								Aux Op		Comments
	X3	ACC		1	2	3	4	5	6		1	2	
1	0	0	a	Y	A	I					-		A(I) → X3
2	2	0	+	Y	X3	B	C				lac		C → ACC
3	2	4	+	Y	X3	B	ACC				-		B + C → ACC
4	2	3	+	Y	X3	ACC	D	E			sac	sx3	X3 → Tx_1
5	0	3	+	Y	Tx_1	ACC	D	E			sac	-	ACC → Ta_2
6	0	0	+	Y	Tx_1	Ta_2	D	E			lac		E → ACC
7	0	5	+	Y	Tx_1	Ta_2	D	ACC					D + E

At step 4, the value of the X3 marker was less than the value of the ACC marker and so the simple sac operation was preceded by a sx3 operation. In this way the Temporary areas are assigned in the inverse order to their use in subsequent operations.

Operands — At source language level, the FORTRAN Compiler is dealing with 'names' and 'constants' as operands. During the evaluation of an expression it is also processing Temporary Storage operands (Tx and Ta) as well as Acc and X3.

Excluding Function names, an operand will generally be associated with a storage area in the final compiled program and is manipulated within the compiler, not by name, but by the address of that area. Most variable names and constants are of this type as are Ta operands. These are called 'Store' operands.

Some operands are associated indirectly with a storage area by means of words which themselves contain the address of the relevant storage areas. The X3 operand and the Tx operand are of this type. These are called 'Indirect' operands.

If the operand is a small integer (<4096), for some classes of operation the compiler manipulates the operand by its value rather than by the address of a word containing the constant. This type of operand is a 'Value' operand.

Relationship between Operators and Operands

By prior use of Store Auxiliary operations, the operands for any main operator can be put into a standard form (e.g. for Binary operations one operand is Acc and one is either a 'Store' operand, X3 or a 'Value' operand)

By investigating the mode, position and type of each operand along with the relevant operator, an operation can be completely defined. Wherever possible in the FORTRAN Compiler, the 'characteristics' of an operation are associated with the operator as shown below.

1. Arithmetic Relational and Logical Operations

1.1 Arithmetic Binary Operations (+, -, *, /, **)

These operations can always be put into the form a Acc op or Acc b op. These forms define two distinct classes of operation - the 'reverse' (a Acc op) and the 'forward' (Acc b op) classes. For the operators '+' and '*', these classes are identical, a fact which is not used in the compiler. These two classes will be denoted by the letters 'r' and 'f'. e.g. '+(f)' implies an operation of the form Acc b +

The operands may have the same or different modes. Each combination of mode defines a separate subclass of operation within the main classes just described. If the modes are indicated by the letters 'I' - INTEGER, 'R' - REAL, 'D' - DOUBLE PRECISION, 'C' - COMPLEX, and 'L' - LOGICAL, these subclasses can be described by associating two letters (representing the mode of Acc and the mode of the other operand) with the operator. Thus an operator presented as '+ (f,I,R)' would imply an operation of the form Acc b + where 'Acc' is of mode INTEGER and 'b' was of mode REAL.

The 'type' of the non-Acc operand defines another subclass of operation depending on whether the type is 'Store'(S), X3 (X), or 'Value'(V).

An operator represented as '+(f,I,R,S)' would completely define the operation to be of the form Acc b + where Acc is of mode INTEGER and 'b' is a 'Store' operand of mode REAL.

In general an algebraic binary operation can be defined by a construct of the form

$$a\ b\ \underline{op}\ (d,\ m_A,\ m_o,\ t)$$

where a, b are the operands
- d - direction of Operation
- m_A - mode of Acc
- m_o - mode of other operand
- t - Type of other operand

1.2 Relational Binary Operations (.LE.,.GE.,.EQ.,.NE.,.GT.,.LT.)

These operations of the form a b op are considered by the compiler to be two distinct operations of the form ab- (or ba-) followed by a unary operation on the result.

The result of the '-' operation is Acc. The unary operation following this operation is a function only of the original relational operator and is independent of the mode of Acc.

Thus a complete Relational Binary operation is defined by a construct of the form

$$a\ b\ -\ (d,\ m_A,\ m_o,\ t)\ \underline{op}'$$

where a, b are the operands
- d, m_A, m_o, t are as defined for Algebraic Binary operations
- op' is a unary operation which depends only on the original Relational Operator.

1.3 Logical Binary (.AND.,.OR.)

These operations can always be put in the form a Acc op, or Acc b op. The modes of Acc and the other operand are the same and must be LOGICAL. The non-Acc operand may be either a 'Store' operand or X3.

Thus a logical Binary operation can be defined by a construct of the form

a b op (d,t)

where a, b are the operands
d - direction of Operation
t - Type of non Acc operand (S or X)

1.4 Unary Minus ($-_u$)

This operation is of the form Acc$-_u$ where Acc may be of any mode other than LOGICAL

Thus a unary minus operation can be defined by a construct of the form

$a -_u (m_A)$

where a is Acc
m_A - mode of Acc

1.5 Logical .NOT.

This operation is of the form Acc op where Acc may be of mode LOGICALonly Thus there is only one .NOT.operation.

2. Store Auxiliary Operations

These operations involve only one operand and so are conceptually of the form a op although the operand is not necessarily the top item in the Operation List.

2.1 Store Accumulator Operations (sac)

These operations put Acc into a storage area. They are a function of the 'mode' of Acc only.

Thus all operations using this operator can be described by a construct of the form

Acc sac (m_A)

where m_A = mode of Acc

2.2 Store X3 Operation (sx3)

This operation is independent of the mode of X3 and involves storing X3 in a Tx operand.

There is only one sx3 operation.

2.3 Load Acc Operations (lac)

These operations take a 'Store', X3, Tx or 'Value' operand and produce a 'Value' in Acc in the standard form required for main operators.

The actual operation depends on the mode and type of the operand.

Thus all operations using this operator can be described by a construct of the form

a lac (m_o, t)

where m_o = mode of operand
t = type of operand

2.4 Load X3 Operations (lx3)

There are two operations in this class which depend on whether the operand is a 'Store' or a Tx operand. There are no 'Value' operands for this class.

Thus the operation is described by a construct of the form

a lx3 (t)

where t = type of operand (S or Tx)

3. Function and Array Operations

The standard form for Function or Array operations allows the arguments or subscripts to be 'Store' or Tx type operands. Other types as yet not mentioned (Function and Array names) are also allowed but these will not be discussed at this point.

The operations are independent of the mode of the Function or Array name, but depend on the type of operand of each argument or subscript.

Once the operands are in a standard form for a Function or Array operation, the operation is treated as a composite series of simple operations which generate first the CALL to the Function or to the Array routine and then the argument or subscript list.

Summary of Operator Associations

The following table summarizes the operator associations of the previous section.

Operation	Operators	Operation Form	Operator Association
Arithmetic Binary	+,-,*,/,**	a b op	op (d, m_A, m_o, t)
Relational Binary	.LE.,.LT.,.EQ. .NE.,.GE.,.GT.	a b op	$-(d, m_A, m_o, t)$ then op'
Logical Binary	.AND.,.ØR.	a b op	
Unary Minus	$-_u$	a $-_u$	$-_u (m_A)$
Logical .NØT.	.NØT.	a.NØT.	.NØT.
Store Acc	sac	Acc sac	sac (m_A)
Store X3	sx3	X3 sx3	sx3
Load Acc	lac	a lac	lac (m_o, t)
Load X3	lx3	a lx3	lx3 (t)

d - direction of binary operation f = Acc b op

r = a Acc op

m_A, m_o - mode of Acc, mode of operand (not Acc)

t - type of Operand s - store

x - x3) Indirect
Tx -)

V - Value

Practical Compilation

So far, this discussion has been in general terms without any particular reference to the actual way in which these operators and their associations are interpreted within the compiler, so as to generate actual instructions. The following sections of this document will show in some detail how the various operations are actually compiled.

Operation Compilation (I)

When an operator is transferred from the Operator List, it is classified as to type - Arithmetic Binary, Relational Binary, Logical Binary, Unary minus, Logical.NOT., Function and Array reference, Assignment operations etc. Each type of operation is processed by separate routines, although these routines use common subroutines to generate instructions and perform auxiliary operations if necessary. The operation types that will be described here are the following.

1. Binary Operations (Arithmetic and Logical)
2. Relational Binary
3. Unary Minus
4. Logical.NOT.
5. Assignment (=)
6. Store Auxiliary

1. Binary Operations (Arithmetic and Logical)

A Binary operation uses as operands the top two items in the PN list. There is no guarantee when starting to evaluate a binary operation that these top items are in the standard form required to satisfactorily compile the operation. To ensure standard conditions, a routine BINSTANDARD is used. This performs all required Store auxiliary operations.

1.1 Arithmetic Binary Operations

As indicated in the 'Summary of Operator Associations' an arithmetic binary operation can be defined by the following parameters

1 op - operator
2 d - direction of Operation Acc b op, a Acc op
3 m_A - mode of Acc
4 m_o - mode of 'a' or 'b'
5 t - type of operand

Not all values of M_A, M_o are allowed for all operators or operand type. In particular neither m_A or m_o may be of mode LOGICAL. Also, if 't' is a 'Value' operand (i.e. a small integer), then m_o may only be of type INTEGER and is therefore redundant.

In order to compile instructions, these parameters are used to address a table entry, the entry indicating whether the particular operation described is valid and if so, what instructions are to be compiled for the operation. The details of this phase of compilation will be described in a later section.

If no cognisance was taken of the redundancies and of the large numbers of invalid mode-operator-etc combinations, a table with 480 entries would be required to describe all possible Arithmetic Binary operations. By separating out operations as indicated below, the number of table entries required is reduced to 250.

The Parameter Groups separated out for Arithmetic Binary Operations are the following

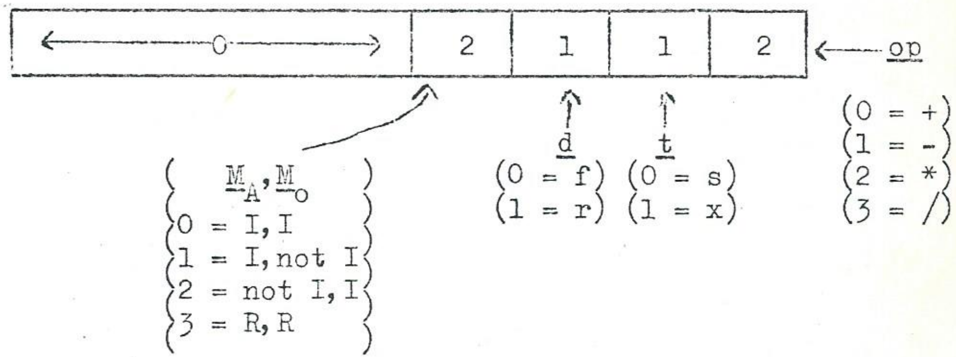
1.1.1 Standard Binary

op = +, -, *, /

t = s,x

modes M_A = I, M_o = I,R,D,C
or M_A = R, M_o = I,R
or M_A = D,C, M_o = I

The position in the group is defined by a binary value as shown in the following diagram.



Thus an operation Acc b * with M_A = R and M_o = I has a value of 18 as defined in this group.

Note that if one (but not both) operand mode is I, then there are 3 operations which are defined to be identical, namely those in which the other operand is REAL, DOUBLE PRECISION or COMPLEX. This is due to the lack of any Mode Auxiliary operations for integer items except fa and fs (see 'Mode Auxiliary Operations' -p.16)

1.1.2 DOUBLE PRECISION Binary

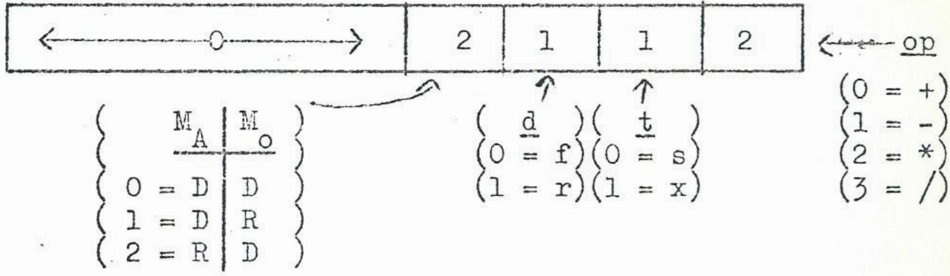
<u>op</u> = +, -, *, /

<u>t</u> = s, x

<u>modes</u> M_A = D, M_o = R,D

or M_A = R, M_o = D

The position in the group is defined by a binary value as shown in the following diagram



This is identical in form to that for Standard Binary. The only difference is in the interpretation of the two mode bits.

As an example, the operation a <u>Acc</u> + with M_A, M_o both 'D', has a value of 8 as defined by this group.

1.1.3 COMPLEX Binary

<u>op</u> = +, -, *, /

<u>t</u> = s, x

<u>modes</u> M_A = C, M_o = R,C

or M_A = R, M_o = C

The position in this group is defined in an identical manner to that for DOUBLE PRECISION Binary except that the interpretation of the two mode bits is as follows.

0 - M_A = C, M_o = C

1 - M_A = C, M_o = R

2 - M_A = R, M_o = C

1.1.4 Small Integer Binary

op = +, -, *, /

t = V

modes M_A = I,R,D,C

M_o = I

For this group of operations, the non-Acc operand is a small integer ('Value' operand). The mode of this operand is INTEGER by definition and therefore need not be explicitly mentioned.

The position in the group is defined by a binary value as shown in the following diagram

← 0 →	1	1	2	← op
	M_A	d		
	(0 = I)	(0 = f)		(0 = +)
	(1 = not I)	(1 = r)		(1 = -)
				(2 = *)
				(3 = /)

Thus the operation Acc 3 + with M_A = I is given a value 0 for this group.

1.1.5 Standard Exponentiation

op = **

t = s,x

modes M_A = I,R,D,C

M_o = I,R,D,C

This group of operations contains all the exponentiation operations except for those with a 'Value' operand.

The position in the group is defined by a binary value as shown in the following diagram

← 0 →	2	2	1	1	← t
	M_o	M_A	d		(0 = s)
	(0 = I)	(0 = I)	(0 = f)		(1 = x)
	(1 = R)	(1 = R)	(1 = r)		
	(2 = D)	(2 = D)			
	(3 = C)	(3 = C)			

Thus the operation a Acc ** for a-REAL and Acc-INTEGER is given the value 16 for this group.

1.1.6 Small Integer Exponentiation $\underline{op} = **$

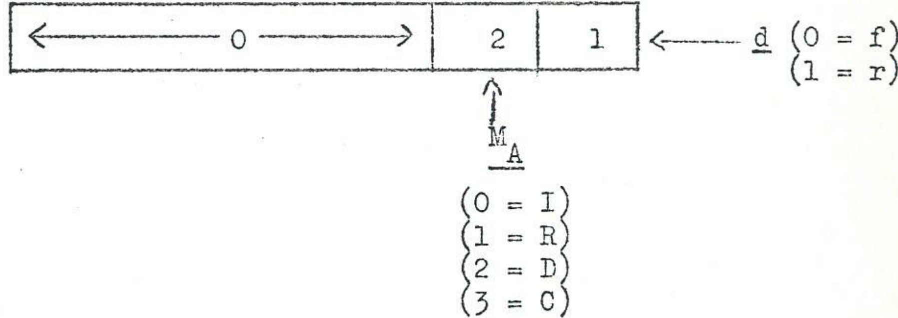
$t = V$

$\underline{modes}$ $M_A = I,R,D,C,$

$M_o = I$

For this group of operations, the non-Acc operand is a small integer ('Value' operand). The mode of this operand is INTEGER by definition and therefore need not be explicitly mentioned.

The position in the group is defined by a binary value as shown in the following diagram



Thus the operation Acc N op with M_A = R has the value 2 for this group.

1.2 Logical Binary Operations

As indicated in the 'Summary of Operator Associations' a logical binary operation can be defined by the following parameters

1. op - operator
2. d - direction of Operation Acc b op, a Acc op
3. t - type of operand

The logical binary operations are symmetric. Thus as far as compiled instructions are concerned, the direction of operation is immaterial.

e.g. A.AND.Acc is the same operation as Acc.AND.A.

At present, there is no 'Value' operand which can be used with logical operators. Thus there are only two types of operands, s, and x.

There are only two operators which produce logical binary operations - .AND. and .OR.

There are therefore only four logical binary operations, which are grouped together and given values as follows

0 op = .AND.; t = s

1 op = .AND.; t = x

2 op = .OR. ; t = s

3 op = .OR. ; t = x

Thus the operation X3 Acc.OR. would have the value 3 for this group.

2. Relational Binary Operations

A Relational Binary operation uses as operands the top two items in the PN list. Depending on the actual operator, the routine processing Relational Binary operations may interchange the top two items. If one of the two items is Acc, the Acc pointer is changed to point to the new location of Acc.

At this stage the routine which processes arithmetic binary operations (BINARITH) is entered to compile an operation using the two top PN items as operands with '-' as the operator. On exit from this routine, the top PN item will be Acc (the result of an operation of the form 'ab-').

The Relational Binary operation is completed by performing a unary operation on Acc. This operation leaves a LOGICAL result as Acc. The operation is independent of the mode of Acc but depends on the original relational operator.

These operations can be defined as a group, the position in the group being as shown in the following list.

0 = .LT.

1 = .LE.

2 = .EQ.

3 = .NE.

4 = .GT.

5 = .GE.

3. Unary Minus

This operation requires one operand (the top item in the PN list) which must be Acc. At the time a unary minus operation is to be compiled, since there is no guarantee that the operand will be Acc, a routine UNSTANDARD is entered to compile all required Store Auxiliary operations.

As indicated in the 'Summary of Operator Associations', a unary minus operation can be defined by one parameter - the mode of Acc (M_A). Thus the unary minus operations can be defined as a group, the positions in this group being defined as follows

0 - M_A = INTEGER
1 - M_A = REAL
2 - M_A = DOUBLE PRECISION
3 - M_A = COMPLEX
4 - M_A = LOGICAL

Although there is a position in the group for Acc in LOGICAL mode, it represents an illegal operation, this is found as such by the operation instruction generating routine.

4. LOGICAL .NOT.

This operation requires one operand (the top item in the PN list) which must be Acc at the time the actual .NOT. operation is to be compiled. As there is no guarantee that the operand will be Acc a routine UNSTANDARD is entered to compile all required Store auxiliary operations.

Acc may only be of LOGICAL mode for this operation. Thus there is only one .NOT. operation.

5. Assignment Operations

a. Simple Assignment (a = e)

For simple assignment, the Assignment operation uses the top two items in the PN list as operands. It is necessary in an Assignment operation for the top item in the PN list to be Acc. To ensure this prior to the actual generation of the Assignment operation, store auxiliary operations are performed if required.

At this stage, the non-Acc operand is either of type 'S' or 'X'.

The Assignment operations require the following three parameters

1. t̲
2. M_A
3. M_o

t̲ = s,x

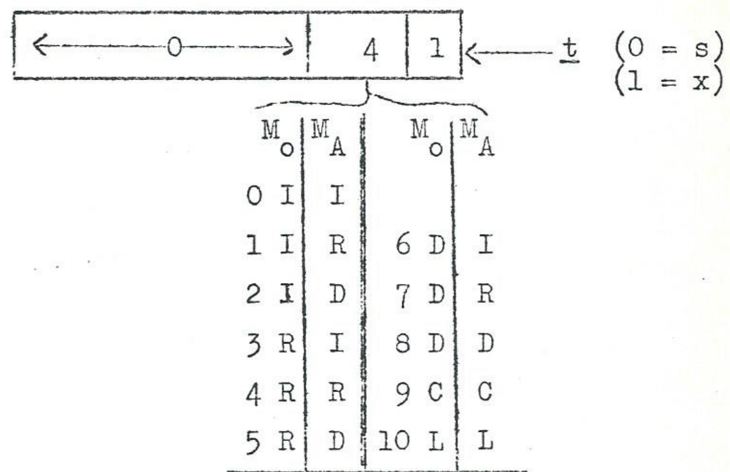
for M_A = I,R,D ; M_o = I,R,D,

for M_A = C; M_o = C

for M_A = L; M_o = L

All allowable combinations produce a group of 22 Assignment operations.

The position of an operation in this group is defined by a binary value as shown in the following diagram



	M_o	M_A		M_o	M_A
0	I	I			
1	I	R	6	D	I
2	I	D	7	D	R
3	R	I	8	D	D
4	R	R	9	C	C
5	R	D	10	L	L

In the PN list, the result of a simple Assignment operation is Acc with the same mode as the original non-Acc operand

i.e. $M_A' = M_o$.

The position of Acc in the PN list is the same as the original position of the non-Acc operand.

Thus the operation 'a Acc =' with M_A = I, M_o = R has a value of 6 in this group of operations. In the PN list the result of the operation is Acc (overwriting 'a') with M_A = R.

b. Multiple Assignment ($a_1, a_2, --- a_k = e$)

The operation of multiple Assignment is a series of simple Assignment operations.

In the PN list, for multiple Assignment, the top two items will be ', Acc' rather than 'a Acc'. Physically moving Acc down one item (overwriting the ',') and amending all associated pointers (including the PN list pointer) will then produce a simple operation of the form 'a Acc ='. This simple operation is then evaluated, and the resultant top two items of the PN list are checked to see whether the top two items are in the form for multiple or simple Assignment.

If the form is still that for multiple Assignment the process is repeated. If the form is that for simple Assignment, the whole operation is terminated as soon as the simple Assignment operation is completed.

Example

If the Assignment produces the following PN list a b,c, Acc the operations proceed as follows

1. Move Acc — a b, c Acc
2. do C = Acc — a b, Acc
3. Move Acc — a b Acc
4. do b = Acc — a Acc
5. do a = Acc — Acc

The resultant operations performed for a,b,c = Acc are c = Acc, b = c, a = b

6. Store Auxiliary Operations

There are basically four types of Store Auxiliary operations, lac, sac, lx3 and sx3 operations.

6.1 Load Acc (lac)

As indicated in the Operator Association table, a lac operation can be defined by two parameters

1. M_o

2. t

t may be one of four types - S,X,Tx,V

For each of these types (except V), M_o may be I,R,D,C or L giving a total of 16 operations (including V)

These define a group. The value of an operation within this group is given in the following table

	M_o	t		M_o	t
0	I	S	8	L	S
1	I	Tx	9	L	Tx
2	R	S	10	I	V
3	R	Tx	11	I	X
4	D	S	12	R	X
5	D	Tx	13	D	X
6	C	S	14	C	X
7	C	Tx	15	L	X

6.2 Store Acc (sac)

A sac operation can be defined by one parameter, M_A.

This defines a group of 5 operations. The value of an operation within this group is shown below

0 - $M_A = I$
1 - $M_A = R$
2 - $M_A = D$
3 - $M_A = C$
4 - $M_A = L$

6.3 Load X3 (lx3)

A lx3 operation depends only on the type of the operand. This may be either S or Tx, giving a total of two operations.

6.4 Store X3 (sx3)

There is only one sx3 operation. This produces a Tx result.

Group Identifiers

Within the compiler, at the time that an operation is defined, the operation is presented to the 'Instruction Generator'as a group number and a position number.

The groups are numbered as follows

1. Main Operations
   - 0 - Standard Binary
   - 2 - Double Precision Binary
   - 4 - Complex Binary
   - 6 - Small-Integer Binary
   - 8 - Standard Exponentiation
   - 10 - Small Integer Exponentiation
   - 16 - Unary Minus
   - 20 - Relational Binary
   - 22 - Logical Binary
   - 24 - Assignment.

2. Store Auxiliary Operations
   - 12 - Load ACC (lac)
   - 14 - Store ACC (sac)

The operations .NOT., lx3, and sx3 are not associated with groups in the compiler because there is only one .NOT., and one sx3 operation. There are only two lx3 operations. These operations are presented to the Instruction Generator in a different way.

Thus any of the operations defined in a group can be described by using two integers (g,p) to specify the group and position. These are used by the Instruction Generation routine (ACOMPIL, Entry ACOM) to extract the correct complete and skeleton instructions for the operation, complete the skeleton instructions and then output them along with the appropriate instruction relativisors.

A skeleton instruction is an instruction which is lacking its 'n' address or part of its 'n' address. For example, if the instruction to be output was 'LDN 6 10', the skeleton instruction would probably be 'LDN 6 0'. The addition of '10' to the instruction completes the instruction.

The Instruction Generator (I)

The Instruction Generator uses the integer pair (g,p) to reference a set of linked tables which determine whether an operation is immediately definable - i.e. whether instructions for the complete operation can be compiled - or whether an operation is definable only after doing a mode-auxiliary operation, or whether the operation is illegal. The tables also determine the type of the result operand. If this is a Tx or a Ta operand, the Generator determines its storage address.

There are four sets of tables used in the Instruction Generator.

1. Group Table (QUERY table)
2. Position Table for each Group (CM tables)
3. Instruction Table for each Group (TINS tables)
4. Relativisor Table for each Group (TREL tables)

These tables have the following characteristics.

1. Group Table.

There is only one Group Table. This is referenced by the 'g' integer of the (g,p) pair.

This is a table of two word items, one item per operation group. 'g' gives the address, relative to the start of the Group Table, of the item required. An item gives the addresses of the three tables associated with the group.

The form of a Group Table item is shown below.

1st word.	top 9 bits.	Relative address of the start of the correct TREL table for the group. This is given relative to the start of the first TREL table. All TREL tables are necessarily in successive areas of store.

rest of word. Absolute address of the start of the correct CM table for the group

2nd word. Address of the start of the correct TINS table for the group.

2. Position Table.

Associated with each Group Table item is a Position Table (CMtable). This is referenced by the 'p' of the (g,p) integer pair. Thus any entry in a Position Table is unique to the (g,p) integer pair and can be used to define exactly how an operation is compiled.

An entry in a Position Table consists of one word. If the word is zero, then the required operation is illegal and can not be compiled. A zero entry causes the compiler to flag an error.

If the entry in the Position Table is positive, the required operation can not be completed before a subsidiary operation (inevitably a mode auxiliary operation) has been compiled.

The single word entry is divided into sections as follows.

1. Top bit (bit 0) This determines whether the operation is completed with this compilation
   0 - not completed
   1 - completed.
   (in considering the rest of the sections of this word, if an operation is 'not completed', the other sections refer to the subsidiary operation).

2. Bit 1. This determines whether the PN List will change as a result of this compilation
   0 - no change to PN List
   1 - PN List to change.

3. Bit 2. If the PN List is to change as a result of this operation, this determines which PN item is to be changed.

The local PN list pointer (PNP1) will normally be pointing to the PN item in question. For a sac operation this will not be the top item in the List, but the item to which the ACC marker is pointing. For a Binary operation PNP1 will be pointing to the second top item in the List. For a unary operation, PNP1 will be pointing to the top item in the List.

The item to be changed is determined by this bit as follows

0 - the change is to the item pointed at by PNP1.
1 - The change is to the item immediately above that pointed at by PNP1. In this case PNP1 is changed to point at this item.

(As an example of this, if a Binary operation of the form A ACC op is not immediately definable, but requires a mode auxiliary operation on ACC, PNP1 which would originally have been pointing at 'A' will now be changed to point at ACC because ACC is the item in the list which is to be changed).

4. Bits 3-5. For an operation in which the PN List is to change, these bits indicate the type of PN item at the end of this operation. They also indicate whether X3 and/or ACC will be destroyed by the operation.

The bit values and interpretations are as follows.

Value	Result PN item	ACC	X3
0	ACC	destroyed	preserved
1	ACC	destroyed	destroyed
2	X3	preserved	destroyed
3	X3	destroyed	destroyed
4	Ta	preserved	destroyed
5	Ta	destroyed	preserved
6	Tx	destroyed	preserved
7	Tx	preserved	destroyed

If ACC or X3 is destroyed, the associated marker is cleared. However, if the result of an operation is ACC or X3 then the marker will be reset to the PN item for the result.

5. Bits 6-8 These define the number of consecutive TINS table entries which are to be used for this operation. These TINS table entries are generally complete or skeleton instructions.

6. Bits 9-11. For operations where the PN List is changed, these bits give the final mode of the PN item which is changed.

   The bits have significance as follows

   1. Integer
   2. Real
   3. Double Precision
   4. Complex
   5. Logical

7. Bits 13-24. These give the absolute address of the first entry in the TINS table which is to be used for this operation. (Bits 6-8 indicate the number of consecutive TINS table entries which are to be used).

Occasionally it is useful to define an operation which is null (i.e. no instructions are compiled). For example there is a group of operations called 'Load X6' which compile instructions to load a value into X6 from some accumulator. If the original accumulator happend to be X6 then there is no requirement to compile instructions.

A null operation is defined by a CM table entry which is zero except in bits 10-12. These may have any non-zero value.

It is a requirement of the system that all entries to a TINS table from a particular CM table refer to the same TINS table. Further this must be the table specified in the Group Table Entry which specifies the particular CM table.

3. Instruction Table and Relativisor Table.

Associated with each Group Table item is an Instruction Table (TINS table). This is referenced via entries in the CM table which itself is associated with the same Group Table item.

The items in a TINS table are one word long. These may represent one of the following three types of item

1. Complete Instruction
2. Skeleton Instruction
3. C/M

Associated with each TINS table is a Relativisor table (TREL table). This is a character table each entry of which is used to determine the type of item represented by a particular TINS table entry. The character associated with a TINS table entry is the same number of characters distant from the head of the TREL table as the TINS entry is words from the start of the TINS table.

The characters in the TREL table have meaning as shown below.

TREL char.	Associated TINS entry.
0	C/M
1-5	Skeleton Instruction
6-63	Complete Instruction.

If the TREL char. is zero, then the associated TINS entry is taken as a c/n to a store area which contains the name, in characters, of a subroutine. When the Instruction Generator routine references such an item, it ensures that the next time the special relativisor R_4 ('SPRE' within the compiler program sheets) is used in a semi-compiled 5-character group it refers to this subroutine. The subroutine name is also put into the Cue List of the Compiler if it is not already there. 'c' gives the number of words in the name of the subroutine.

If the TREL character is 1, 2, 3, 4 or 5, then the TINS entry is taken as a skeleton instruction. When the Instruction Generator routine references such an item it adds to the skeleton instruction the contents of register NWRD-1 $+\alpha$ where 'α' is the TREL character. This (complete) instruction is then output using the contents of register REL-1 $+\alpha$ as the relativisor. In nearly every case for skeleton instructions, the character is 1 and the registers used are NWRD and REL.

The contents of these registers are defined by the sections of program which process the particular type of operation - e.g. BINARITH for Binary arithmetic operands.

Normally NWRD would be the address in the object program of the non-ACC operand. REL would be the relativisor for that particular address.

If the TREL character is greater than 5, then the TINS entry is taken as a complete instruction. In this case, if the Instruction Generator references such an item it takes the TREL character -1 as the relativisor for the instruction and then outputs the instruction using that relativisor.

As an example of the use of the Instruction Generator, consider the compilation of the operation A+B. A and B are both real variables. A has been assigned location 10 in LV space and B has been assigned 14 in LV space. Both are standard variables.

The PN list contains the addresses assigned to A and B as well as their types (standard variable).

Thus at the time the operation A+B is to be performed, the top of the PN List appears as follows ('var'. indicating 'variable')

Real var./14 (B)
Real var./10 (A)

PNP is pointing at B.

The main (+) operation can not take place until the initial lac operation has been performed. This is done on the variable B in the routine BINSTANDARD using the Instruction Generator.

For this operation, NWRD is set in LOADACC (called from BINSTANDARD) to the address of B (=14) and REL is set to 'VREL' (the relativisor for LV). PNP1, the local PN List pointer is pointing at the item for B.

For the lac operation, 'g' is 12 (See Section on 'Group Identifiers') and 'p' is 2. (2 = REAL variable).

The Instruction Generator is entered with these parameters. The CM table entry for (g,p) = (12,2) is the following (as written in PLAN).

#601/LR+#20000

This indicates that
1. The operation will be completed.
2. The PN List will change.
3. The change is to the item pointed at by PNP1.
4. The result of the operation is ACC and that X3 is not destroyed.
5. The Instruction Generator will reference only one TINS entry.
6. The final mode for the operation will be REAL.
7. The TINS entry is a location labelled in PLAN as LR.

The TREL table entry associated with the TINS table entry is '1'.
The TINS table entry is 'LFP 0'

The Instruction Generator recognizes the TINS entry as a skeleton instruction, and since the TREL entry was 1, adds the contents of NWRD to the instruction. Outputting the instruction using the contents of REL produces the final result.

LFP 14 VREL

The PN List is then changed according to instructions and will now read

Real ACC/- (ACC)
Real var/10 (A)

The operation is now instandard form for a Binary arithmetic operation.

On the next entry to the Instruction Generator, 'g' is zero and 'p' is 56. NWRD is set to the address of A (=10) and REL is set to 'VREL'. PNP1 points at A (the botton item in binary operations).

The CM table entry for (g,p) = (0,56) is the following
#611/ARRR+#20000

This indicates that
1. The operation will be completed.
2. The PN List will change.
3. The change is to the item pointed at by PNP1.
4. The result of the operation is ACC, and X3 may be destroyed.
5. The Instruction Generator will reference only one TINS entry.
6. The final mode for the operation will be REAL.
7. The TINS entry is a location labelled in PLAN as ARRR.

As for the previous case the appropriate TREL table entry is '1'. The TINS table entry is FAD 0 0.

This will produce a final instruction of

FAD 0 10 VREL

Thus A+B would compile into the two instructions

LFP 14 <u>VREL</u>
FAD 10 <u>VREL</u>

As a further example of the use of the Instruction Generator, consider the compilation of D + I where I is INTEGER and D is DOUBLE PRECISION. I has been assigned location 20 in <u>LV</u> space and D has been assigned 24 in <u>LV</u> space. Both are standard variables.

At the time the operation D+I is to be performed, the top of the PN list appears as follows:-

Integer var/20 (I)
D.P. var/24 (D)

PNP is pointing at I, The top item in the PN list.
The operation '+' is stored in register ΘPER.
The main operation (+) cannot take place until the initial <u>lac</u> operation has been performed. For this operation NWRD is set to the address of I (=20) and REL is set to VREL.
PNP1, The local PN list pointer is pointing at the item for B.
For the <u>lac</u> operation, 'g' is 12, and 'p' is 0. (INTEGER)
The CM table entry for (12,0) is the following.

#601/LI + #10000.

This indicates that the Instruction Generator will reference only one TINS entry, and that the PN list will be changed to <u>ACC</u> at the item indicated by PNP1. <u>ACC</u> will be REAL.
The TREL table entry associated with the TINS table entry is '1'.
The TINS table entry is 'LDX 6 0.'
The Instruction Generator adds NWRD to this entry to form the complete instruction for the operation. This instruction is output using the contents of REL as the relativisor.
The instruction output then is.

LDX 6 0 <u>VREL</u>

The PN list is then changed according to instructions and will now read:-

Integer <u>ACC</u>/- (<u>ACC</u>)
D.P. Var/24 (D)

This is now in standard form for a binary operation.

On the next entry to the Instruction Generator, 'g' is zero and 'p' is 24 (INTEGER ACC, not integer operand). NWRD is set to the address of D and REL is set to 'VREL'. PNP1 points at 'D'.
The CM table entry for (g,p) = (0,24) is the following:-

#301/SBFX + #20000

This indicates that 1. the operation will not be completed, but requires an auxiliary operation.

2. the PN list will change
3. the change is to the top item rather than the one pointed to by PNP1.
4. The result of the operation is ACC; . X3 is not destroyed.
5. the final mode for the operation will be REAL.
6. the Instruction Generator will reference only one TINS entry (at location SBFX).

The TREL table entry associated with the TINS entry is #55.
Thus the actual relativisor to be used with this entry is #54.
This is the special relativisor R_1 which will have been set earlier to the name of the Object time Arithmetic Package (%FAP4)
The TINS entry is a complete instruction.

CALL 1 15

which when output with relativisor R_1 is equivalent to

CALL 1 15 %FAP4

(This entry to %FAP4 takes an integer in X6 and generates the equivalent floating point number in the floating point accumulator).
The PN list is now changed according the instructions and will now read.

REAL ACC / -	(ACC)
DP. Var/24	(D)

Since the operation was defined as "not complete", the complete operation is attempted again.using the current top items in the PN list.
The operator is still available in ΘPER (+), and the global PN pointer PNP has not been changed since the start of the complete operation and is still pointing at ACC

This time the PN list is in the correct form for a binary operation. On this entry to the Instruction Generator, 'g' is 2 and 'p' is 40 (REAL ACC, Double Precision Operand). NWRD is set to the address of D and REL is set to VREL. PNP1 points at D.

The CM table entry for (g, p) = (2,40) is the following.

#302/DPFX + #30000

This indicates that.

1. the operation will not be completed but requires an auxiliary operation.
2. the PN list will change.
3. the change is to the top item.
4. the result of the operation is ACC;and X3 is not destroyed.
5. the final mode for the operation will be DOUBLE PRECISION
6. the Instruction Generator will reference two consecutive TINS entries (starting at location DPFX).

The TREL table entries associated with the TINS entries are both #41. This shows that both TINS entries are complete instructions which will be output using the relativisor #40.

These instructions are:-

```
LDN  4  0
LDN  5  0
```

The PN list is now changed according to instructions and will now read

```
DP ACC / -          (ACC)
DP Var/24           (D)
```

Once again, since the operation was defined as "not complete" the complete operation is attempted again.

On the next entry to the Instruction Generator, 'g' is 2 and 'p' is 8 (both operands DOUBLE PRECISION). NWRD is set to the address of D (i.e. 24), and REL is set to VREL. PNP1 points at D.

The CM table entry for (g, p) = (2,8) is the following

#613/ADDR + #30000

This indicates that the operation will be completed, the PN list item pointed at by PNP1 will change to ACC and that ACC will be of mode DOUBLE PRECISION. It further indicates that X3 may be destroyed during the operation.

The Instruction Generator will reference three consecutive TINS entries starting at location ADDR.

The TREL table entry associated with ADDR is 1. The TINS table entry (ADDR) contains

```
LDN  3  0
```

This is a skeleton instruction which is output after adding NWRD. The relativisor is the contents of REL.

Thus the instruction output is

```
LDN  3  24  VREL
```

The second TREL entry is 'O'. This indicates that the TINS entry (which is 1/DPAR1) is a c/m to an area which contains the name of a routine . This name is to be output if necessary as a setting for R4 so that the next time SPRE is used as a relativisor (R_4), it will refer to this routine.

The area DPAR1 (one word long) contains the name %FDP.

The third TREL entry is '#60'. This indicates that the corresponding TINS entry (CALL 1 0) is a complete instruction which will be output with the relativisor '#57' (which is R_4)

This then generates CALL 1 0 %FDP

Thus the instructions which are compiled for the complete operation D+I are the following:

LDX	6	20	VREL	(I → X6
CALL	1	15	%FAP4	(Float I)
LDN	4	0		(Convert to D.P.)
LDN	5	0		
LDN	3	24	VREL	(D-Address → X3
CALL	1	0	%FDP	(I + D)

On completion of the whole operation, the Instruction Generator returns control to the Arithmetic Binary routine, which performs certain red-tape operations (such as setting the global PN pointer (PNP) to the local pointer (PNP1))before returning to look for the next operation.

Restriction on the General Use of the Instruction Generator

The Instruction Generator is used throughout the whole compiler. Where it is used from routines other than those which are used for expression evaluation (e.g. from the routines processing the READ and WRITE statements), an operation must be defined to be complete.

When an operation is defined as "not complete" by the Instruction Generator, the return from the Instruction Generator upon generating the auxiliary operation is to a particular location (CMPE) in the Expression Evaluation routine rather than back to the routine from which it was called.

Apart from this restriction, The Instruction Generator is used throughout the compiler in the manner described in the previous section wherever instruction compilation can best be described by 'g, p' values.

Instruction Generator (II)

For many classes of operation, where a group only contains one or two items a different entry to the Instruction Generator (at AC92) is used. For this entry an operation is described by a position number such that no two operation which use this entry have the same position number.

The operations lx3, Sx3, and .NOT. are operations which are suitable for this entry to the Instruction Generator. They have position numbers as shown below.

1. Load X3 Operand is 'S' type Position Number is 4
2. Load X3 Operand is T_x type Position Number is 5
3. Store X3 Position Number is 6
4. .NOT. Position Number is 10

These numbers are used to reference a table (similar to the CM tables) which starts at location AUXCM. The position in the table for an operation determines the position number assigned to the operation.

A table entry is one word long and is a c/m to a second table (identical to a TINS table) which is used for the generation of the actual instructions for the operations.

'c' gives the number of consecutive words in this table which are to be used for the current operation. This may not exceed 7.

'm' gives the address of the first item in this table which is to be used for the operation.

Associated with this table (the AUXINST table) is a relativisor table with the same characteristics as a TREL table. This table is the AUXREL table.

Difference between AC9M and AC92 entry to Instruction Generator

The basic difference between the two entry points is that an operation which uses AC9Z is automatically defined to be complete. This is not the case for entry AC9M. In this case the 'completeness' of an operation is determined by the CM table item for the operation.

A second difference is that an operation defined through entry AC92 does not directly influence the PN list.

If such an operation actually does influence the PN list this is effected externally and not from within The Instruction Generator. Thus although the 1x3 operation will change some PN item from a 'store' or 'Tx' item to an X3 item the actual change to the list is made in the LOADX3 routine which defines the operation ,rather than in the Instruction Generator which compiles it.

Operation Compilation (II)

Function Operations.

In the current compiler, a Function reference in an expression consists of evaluating the function arguments and then generating a standard calling sequence

CALL   1   Function Name
argument List.

At the time the CALL is obeyed, none of the arguments may be assumed to be in [illegible] or in X3. The result of a function reference is a value in Acc, the mode being given by the mode assigned to the function 'name'.

Due to the rules for interpreting the PN list, at the time a function operation is to be generated the argument expressions will have been evaluated and the function operator and operands will be in a standard form. This is one of the following

Name a f   (one-argument function)
Name $a_1 a_2$, a_3 ,----,f   (multi argument function)

'Name' is the function name
a and the a_i are the function arguments.

Also, due to the rules for interpreting the PN list one of the arguments may be X3 or Acc

For example, if the Function reference is

F (A(I)) where A is an array name,

this will be in the PN list as shown below at the time the function operation is defined.

F X3 f.

If the function reference had been

F(X+Y)

the PN list at the time the function reference is defined would be

F Acc f.

If one of the arguments is X3 or Acc, a Store auxiliary operation is carried out before the function operation so that in the previous two cases, the end result in the PN list is

1)   [illegible] Tx f
2)   F Ta f

The operands are now in standard form to be processed for the function operation.

Basically the function operation consists of the generation of the instruction 'CALL   1   Function Name' followed by a list of instructions, one per argument.

The instructions compiled depend on the arguments. For example if the argument is a standard variable, the instruction would be 'LDN   3   Address of Variable' whereas if the argument is a Function

name, the instruction would be 'LDX 3 ADD' where ADD is the address of a constant containing the instruction 'BRN Start of Function'.

The method of recognizing the Function name and generating the associated argument list of instructions is as follows.

1. If the top item in the PN list is not ',' the function has one argument. The second top item in the PN list is the name of the function.

2. If the top item in the PN list is ',' the function has more then one argument. If the list is scanned backwards looking at every second operand (i.e. the third top operand first) the Function name can be found. If the scanned operand is a ',' there are more operands. If the scanned operand is not ',' it is the first argument for the function. The Function name is the next item back in the list.

   The number of arguments is one more than the number of commas found.

At this stage the call instruction can be generated and then the PN list can be scanned forwards starting at the Function name to process the function arguments, ignoring commas until the correct number of arguments has been found.

The resultant PN list item for a function reference replaces the name of the function in the list. This item is Acc and is given the same mode as the mode given to the function name.

Intrinsic Functions

In general, functions allow only a fixed number of arguments. For example SIN is a function written to use one argument and will not work with more than one argument. However there is a small class of functions (called 'Intrinsic' functions) which are defined in the FORTRAN language as allowing a variable number of arguments. For example, MAX1. The names belonging to this class are known to the compiler which marks the function name as belonging to this special class before the name is put in the PN list.

Thus during the analysis of the function operation, the type of the function (whether 'Intrinsic' or not) is known at the time the function name is found.

If the function is an Intrinsic function, the CALL instruction is followed by a word which contains a count of the number of arguments. For example, if the original function reference had been

MAX1(I,J,K,L)

The sequence of instructions compiled would start:-

```
CALL   1  MAX1
          +4              (count of arguments)
```

followed by the argument list.

The complete list of Intrinsic Functions is the following.

AMAXO, AMAX1, MAXO, MAX1, DMAX1.
AMINO, AMIN1, MINO, MIN1, DMIN1.

Array Operations

In the current compiler an Array reference in an expression consists

of evaluating the Array subscript expressions and then generating a standard instruction sequence.

1. Load X3 with address of the Array Header
2. CALL 1 %FAP4+6 or CALL 1 %FAP4+20
3. Subscript list.

At 1, the instruction compiled is either

```
        LDN    3    'Address of Header'
    or  LDX    3    'Address of Address of Header'
```

depending on whether the Array is not or is a 'dummy' array.

At 2, the CALL is to %FAP4+6 if it is an INTEGER, REAL or LOGICAL array. If it is a COMPLEX or DOUBLEPRECISION Array then the CALL is to %FAP4+20.

At 3. The subscript list is identical to the argument list for functions and consists of instructions to load the address of successive subscripts into X3, one instruction per subscript.

The method of determining the number of subscripts in the PN list and the name of the array is identical to the method used to find the number of arguments and the name for a function.

Function (Array) Argument (Subscript) Lists.

These are identical except that the subscript list only allows constants, variables and expressions. The function list also allows array names and function names.

At the time in the function operation that the PN list is being scanned backwards looking for the Function (or Array) name, the argument (or subscript) list items are being analysed (using a routine ANALYZE). This routine ensures that the argument list will only consist of "LDN 3 α and 'LDX 3 β type instructions.

If the argument is an 'unassigned variable' (one which has not been used before in an executable statement or appeared in a COMMON, EQUIVALENCE or DATA statement or appeared as the argument of a FUNCTION or SUBROUTINE statement) the ANALYZE routine assigns space (according to its mode) to the variable and changes the PN list to 'assigned variable'.

If the argument is a workstore (Tx or Ta), the workstore register, CWSA, is set to the value of the argument. e.g. if the argument was the sixth location in LW space, then CWSA would be reset to 6. This would allow the next Tx or Ta item required to be assigned the sixth location in LW space.

If the previous value of CWSA was greater then the 'Maximum Work Store' register (WSMA) then this is set to CWSA before the aforementioned operation takes place.

No change is made to the PN list.

If the argument is a small constant, which normally is a 'Value' operand rather than a 'store operand', a constant of this value is compiled into the LC area and a normal PN item for the argument as a Standard Constant is made to replace the 'Value' item.

If the argument is a function or a Statement Function, a branch to the start of the function or Statement function is compiled into the LC area, and the PN item for the argument is set to look like an indirectly addressed variable (e.g. a COMMON variable) which uses that area.

In this way, the actual instruction compiled for this argument will be LDX 3 'Area', an instruction which, if obeyed, will enable the branch instruction to be passed through for use in the main function for which it is an argument.

If the argument is an array name, the PN item will be changed to indicate "Whole Array as an Argument".

For standard, dummy, DATA or COMMON variables or for constants (other than small integers) the ANALYZE routine does nothing.

Thus at the time of generation of compiled instructions for the argument list, the PN list is such that all arguments can be compiled as 'LDN 'LDN 3 α' or 'LDX 3 β' type instructions.

This is done using a routine LOADX3 which looks at PN list item and compiles the instruction according to the item. Before entry to this routine, PNP1, the local PN indicator is set to point to the item concerned. The routine leaves PNP1 unchanged.

Construction of the PN list in the Compiler

The PN list and operator list in the compiler are contained in a block of 300 words such that the PN list starts at the first location and is generated in a forwards direction, and the operator list starts at the last location and is generated in the reverse direction. Thus the two lists grow towards each other. Each item in each list is one word long. The top of each list is marked by a pointer - PNP for the PN list, and OPTP for the Operator list. Each time an operand is put into the PN list, the value of PNP is incremented by 1. Each time an operator is put into the Operator list OPTP is decremented by 1. The reverse procedure is followed when an operator is removed from the Operator list.

Items are only removed from the PN list when an operation is performed. A local PN list pointer (PNP1) is used during operations so that auxiliary operations can be compiled even though the actual operand for the operator is not the top item in the PN list (e.g. sac, sx3 operations).
At the end of a main operation, the actual PN pointer is set to the current value of the local pointer.

PN List item

A PN list item is one word long and is divided into 3 parts as follows

bits 0-5	Item type
bits 6-8	Item mode
bits 9-23	Item Data

The Item Type indicates whether the PN item represents a Standard Variable, constant, Temporary Store Location (Ta or Tx), Acc, an array name, or a Function name.
In General it discriminates among all possible operand types which can be distinguished either by their use or by their type of storage area. For example a variable defined in a DATA statement will have a different 'Type' to a normal variable. A variable which was defined as an argument in a FUNCTION or SUBROUTINE statement will again have a different Item type. Ta and Tx operands, although they occupy the same type of storage have different Type bits because they are not both used in the same manner. A small integer has a different Type to any other constant because it does not normally exist in a store location.

The item mode gives The mode of the operand as follows

1 - INTEGER
2 - REAL
3 - DOUBLE PRECISION
4 - COMPLEX
5 - LOGICAL

The Item Data usually gives the address in the object program for the item in question. This address is given relative to the start of the local storage area.
For example if the item was a Standard Variable which had been assigned a block of store starting at location 10 in Lower Variable space, the Item Data for that variable would be 10.

For small integers, the Item Data gives the value of the constant.

For certain operands (e.g. Function names and undefined items), the Item Data gives the address in the Compiler Main List of the list item for that operand.

The Item Data for the ACC operand is undefined.
A 'comma' is put into the PN list as a zero word.

The following table gives the Type codes and Data in the PN list for all operands.

Quantity	Type Code	Data	Rel	D or I
Absolute Store	0	Address of Store	NORE	D
Standard Variable	1	Address of Variable	VREL	D
Constant	2	Address of Constant	CREL	D
Small Integer	3	Value of Integer	NORE	D
Common Variable	4	Address of Constant referencing var.	CREL	I
Dummy Variable	5	Address of location referencing var.	WSRE	I
Stat. fcn. argument	6	Address of location referencing arg.	WSRE	I
Dummy Array Name	7	Address of location referencing Header of actual Array	WSRE	I
Dummy Function Name	8	Address of location branching to actual Function	WSRE	I
Function Name	9	Address of List Item for Function	SPRE	I
Stat.fcn. Name	10	Address of start of Stat. fcn.	P2RE	I
Work Store (Ta)	11	Address of Store	WSRE	D
Normal Array Name	12	Address of Array Header	CREL	I
Work Store (Tx)	13	Address of Store	WSRE	I
Total Array	14	Address of Array Header	CREL	D
Dummy Array	15	Address of word referencing Header of Actual Array	WSRE	I
Intrinsic Function Name	16	Address of List Item for Function	SPRE	I
DATA variable	17	Address of Variable	CREL	D
X3 = address of result (X3)	18	--	NORE	D
Unknown	20	Address of List Item.	--	-
Stat. fcn. definition	21	--	--	-
Accumulator (Acc)	32	--	--	-

The column 'Rel' in the table indicates which relativisor is to be used if the operand is to be used in a skeleton instruction. Thus, if the operand is a Standard Variable, it would appear with the Item Data in the modifier part of the instruction which would be output using the relativisor 'VREL'.

The column D or I indicates whether a 'Load' operation (load X3 or Acc) is to assume the item in lower storage. If 'D', the item is in lower storage. If 'I', the item must be considered as though it is in upper storage.

These two characteristics of the operands ('Rel' and 'D or I') are found by referencing a character table which contains the relativisors and a 'bit' table which gives the D and I information.

Thus from these tables, all the information necessary to define an operand can be found.

Complex Constants

A complex constant has the form

'(a,b)' where a and b are optionally signed REAL constants. This construct must effectively give rise to one PN list item for the constant and no Operator list items.

This is effected as follows:

1. The '(' goes into the Operator list
2. The sign, if there, goes into the Operator list.
3. The constant 'a' is recognized as a REAL constant.
   The constant terminator is recognized as ','

Only at this point is the construct recognized as that of a complex constant. Further, the construct will only be recognized as such if it is not immediately preceded by an Alphanumeric or numeric field.

e.g. F(+3.1, - 2.4) defines a function reference
whereas +(3.1, - 2.4) defines a complex constant.

At this stage, the top operator is removed from the operator list. If it is 'unary minus', the constant 'a' is negated and stored in a temporary store. The '(' is also removed from the operator list.

'b' is then found. If this is signed and if the sign is 'unary minus', 'b' is negated and stored in NAME+2, NAME+3. The original Temporary Store is transferred to NAME, NAME+1 giving a block of four words containing the internal representation for the complex constant.

The final ')' is checked and then ignored.

The constant 'terminator' is then found. (The operator separator or terminator following ')').

The whole construct has now been transformed to the form of a constant in NAME→NAME+3, and a terminator in TERM, which is the standard form for the analysis of any operand-operator pair.

Standard and Intrinsic Functions

Most functions, if not defined in 'Type' statements are assigned a mode INTEGER or REAL according to the first letter of the function name. However, there is a small class of functions which do not follow the normal rule. These are standard functions which are defined to have a Double Precision COMPLEX or LOGICAL mode.

There is a second class of function, which has the characteristic that each member of the class allows a variable member of arguments. This is the class of 'Intrinsic' functions. This second class contains some members of the first class.

The two classes of function names form a closed set, all entries of which are known to the compiler.

Whenever the compiler processes a function name found in an expression for the first time, it checks whether the name belongs to this set. If it does, the Main List Item for the name is updated to contain the characteristics for the name as defined in the set (i.e. mode and type of function - standard or intrinsic) unless the List Item as already made has had a mode assigned to it (by a 'Type' statement) which is different from the mode defined in the set. If the mode is different, the function is considered as similar to a Standard Function even if it has the same name as an Intrinsic function.

Operator List

The operator list consists of one word items each of which represents an operator or a separator.

The design of the operator item is shown on the next page

1. Top 21 bits of the word.	These indicate the 'weight' of the operator. This varies from -1 (#7777777) for ')' to 9 for the Function and Array pseudo-operators (There is another pseudo operator which has not been mentioned with a weight of 10.)
2. Bottom 3 bits.	These are used to distinguish among operators of the same weight.

The complete list of operator items is given below

Item	Representation	Item	Representation
)	#7777777 7	.GT.	#4 4
(	#0 0	.GE.	#4 5
=	#0 2	+	#5 0
,	#0 4	-	#5 4
.OR.	#1 0	$-_u$	#6 0
.AND.	#2 0	*	#7 0
.NOT.	#3 0	/	#7 4
.LT.	#4 0	**	#10 0
.LE.	#4 1	<u>a</u>	#11 0
.EQ.	#4 2	<u>f</u>	#11 2
.NE.	#4 3	<u>ifq</u>	#12 0

The operator <u>ifq</u> is a special pseudo-operator which is used when the operation type is not defined but may later be found to be one of

1. Statement Function definition
2. IF statement
3. Array operation
4. Function operation.

In this table the 'weight' is used to reference a separate character table to define the class of operation involved (e.g. Arithmetic Binary, Unary Minus etc.)

V.K. Taylor

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED

Scientific Programming Department
ICT 1900 Series

FORTRAN NOTE 15
29.9.65.

The Statement Function and IF Mechanism in the ICT FORTRAN IV Compiler

General

Statements thought to be Assignment, and Statements known to be IF, are passed through the Expression Evaluator at entry FPNE1. All other entries to the Expression Evaluator are at either entry FPNE or FPNB.

If a construction of the form 'name (' is found at the start of a statement during use of the Expression Evaluator, entry FPNE1, the statement may be either a Statement Function Definition or an IF statement or an Assignment statement starting with an array element.

If and only if this construction is found an operator #120 is stored in the operator list before the '(' is put there. This operator is in place of the f or a operation which might otherwise be put into the list. This operator is the 'IFQ' operator and is used to delay having to make a commitment about the type of operation until the last possible moment. Further the 'name' is stored in a special area (SECRET) and no Main List Item is made for it. The PN list item for this name is a c/m to 'SECRET'.

When this operator (IFQ) is forced out of the Operator list the operands will be in the standard form for an Array or Function operation.

This operator is the one exception to the rule that a ')' forces out all operators until '(' and is then anihilated along with the ')'. If the operator which precedes '(' in the list is the operator IFQ, it is forced out by the influence of the ')'.

Considering the Operands:

If the top operand is mode LOGICAL, the statement is assumed to be a Logical IF statement.

If the field following the ')' in the statement is an integer, then the statement is assumed to be an Arithmetic IF statement.

If neither of these cases apply, the PN list item for the 'name' (which holds a c/m to SECRET) is used to enable the name to be looked for in the Compiler's Main List. If it is not there or is there but is undefined at to 'type', the statement is assumed to be a Statement Function Definition.

If none of these cases apply, the statement is considered an Assignment Statement which is introduced by an array element. In this case the PN list item for the 'name' is changed to that for the array name, and an array operator is put in the top of the operator list. This puts the PN and Operator lists back in a standard condition.

All IF statements are passed through the expression evaluator using entry FPNE1. This is due to the fact that in the early stages of investigation, certain logical IF statements can not be distinguished from Assignment statements.

Logical IF

At the stage in the expression evaluation at which the IFQ operator is recognized, the parenthesized logical expression of the IF will have been evaluated. The PN list item for the name in 'SECRET', which must be the second operand is looked at to see whether the name is 'IF'. If it is not, the construction is illegal. If it is IF, the expression is compiled in X6 unless it is already there, the current transfer address is stored in a stack (of length 9) and the subsidiary statement is generated. On exit from the Statement generator, a conditional branch from the stored T.A. to the current TA completes the IF statement.

A stack link system is used in the Statement Generator because it is being called recursively from itself. This is due to the Statement Generator finding IF in the first place and also having to generate the subsidiary operations.

Arithmetic IF

At the stage in the expression evaluation at which the IFQ operator is recognized, the parenthesized arithmetic expression of the IF will have been evaluated. The PN list item for the name in 'SECRET', which must be the second operand, is looked at to see whether the name is IF. If it is not, the construction is illegal. If it is IF, the expression is compiled into X6 unless it is already there. The IF successors are then investigated and suitable branch instructions compiled taking into consideration any useful facts known about the successors (e.g. two successors the same).

Statement Function Definition

At the time a Statement function definition is recognized, the PN list has processed all the arguments for that definition. For a suitable definition no operand may be ACC or X3. i.e. no operand was originally an expression.
Since no expressions have been evaluated, no items will have been assigned object program space during the scan of the argument. Thus at the time that the construct is recognized as the start of a Statement Function definition the argument can be reappraised without having to decompile any of the statement.

At this stage the statement up to the end of the construct 'name ( )' is rescanned outside the PN list (i.e. linearly). The name of the Statement Function is put in the main list (as a S.F.) and the arguments are processed into the Main List as S.F. arguments. If the character following the construct is not '=', the whole construction is illegal otherwise a subroutine prologue is compiled.

The statement is then scanned again from the beginning of the line, using the Expression Evaluator - this time using entry FPNE.

Eventually this scan will require that the f operation for 'name ( )' be evaluated. It is determined that the operation is a SF because the 'name' is so marked. It can be determined that it is a definition statement because the character following the ')' is '='. The f operation under these circumstances merely changes the 'name' operand to a special operand (#210/0 + mode in the counter) which indicates 'S.F. Definition'. This will eventually be processed by the '=' operator after the r.h.s. expression has been evaluated.

If the '=' operator discovers a bottom operand of this type, it will only perform mode auxiliary and lac operations; It will not generate store instructions.

V.K. Taylor.

V. K. Taylor.
19A.

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED

Scientific Programming Department

I.C.T. 1900 Series

FORTRAN NOTE 16
12.10.65.

Consolidated Leader Listing and Private Error Codes

This Fortran Note describes in detail the format of the listing of the consolidated leader. The interpretation of the private error code number printed out when an error occurs is also explained.

Note: This facility has been implemented for internal use only at present, since the consolidated leader is printed without any frills and without any explanation. Listing facilities in issued compilers may differ radically.

Consolidated Leader

A listing of the consolidated leader will only occur if the statement "LEADER" is present within the program description.

The appendix shows a typical example of such a listing which is explained in the following notes.

a) The relativisor block, (=)

This block contains ten octal fields, followed by the terminator '4', with the significance:

Octal field	Starting address of	Mnemonic
43	Lower workspace	W
45	Lower variables	V
61	Lower common variables	
100	Lower present	C
310	Literals	LT
311	Lower common preset	
2421	Upper preset	UC
2515	Upper common preset	
2515	Upper variables	UV
2651	Upper common variables	

b) The parameter block, (>)

The Fortran compiler puts out five octal fields; a typical example is :

633	Offset (used by GPL)
0	Release instruction (null in this case)
51576445	Program name in character form (IθTE)
3350	Total core store used
201	Listspace used during compilation

c) The cue blocks, (?)

These contain two fields : an octal field and a character field. In the octal field, the top two digits give the type of field and the bottom five digits give the starting address within core store. The character field contains the name associated with the cue (% indicates that it is a private routine).

Cue value	Type	Rest of word
41	Program	Starting address
33	Lower common preset	" "
23	Lower common variable	" "
04	Peripheral	# xy(x = type, y = unit)
02	Entry	Entry address
00	Blank	#70077
03	Upper common variable	Starting address
13	Upper common preset	" "

Thus using the information contained in the consolidated leader it is possible to build up a complete picture of the core store giving both starting address of area and size of area.

Private Error Code

The compiler has recently been modified so that every time an error occurs, and provided full listing mode has been specified, the contents of accumulator X0 are printed out in octal . Since the vast majority of subroutines in the compiler use X0 to store the link, this provides a method, in conjunction with a printout of the consolidated leader of the relevant version of the compiler, of locating where the error occured.

L.R. Fairbrother
12.10.65.

Appendix - Example of Listing

```
   FORTRAN COMPILATION BY #FORT MK 1A    DATE 14/09/65

   NAME(IOTEST)                            )
   LIST                                    )
   LEADER                                  ) Program Description
   INPUT 4, 7=TRO                          )
   OUTPUT 3, (MONITOR)=TPO                 )
   OUTPUT 1=LPO                            )
   MASTER PL04
   DIMENSION A(13)
   F = 3, 1415926536/180
   DO 1 I = 0, 360, 3
   X = I * F
   KS = 50 + NINT (SIN(X) * 50  )
   KC = 50 + NINT (COS(X) * 50  )
   CALL PLOT (64, 10, 101, A(1) )
   CALL PLOT (51, KS, 101, A(1) )
   CALL PLOT (35, KC, 101, A(1) )
1  WRITE (1, 2)  I, A(1)
2  FORMAT (I4, 3XA101)
   PAUSE JJ
   END

END OF SEGMENT PL04, LENGTH   72


   FINISH


   =  43 45 61 100 310 311 2421 2515 2515 2651 4
   >  633 0 51576445 3350 201 4
   ?  41000324  %%%F 4
   ?  33000311  %FICLIST 4
   ?  23000061  %FIOPT   4
   ?  40000000  %TPO 4
   ?  40000104  %TPO 4
   ?  23000062  %FIOLP   4
   ?  40000200  %LPO 4
   ?  41000324  PL04 4
   ?  41000434  %FAP4    4
   ?  41002343  NINT 4
   ?  41002227  SIN  4
   ?  41002224  COS  4
   ?  41002361  PLOT 4
   ?  41000702  %FINOUT  4
   ?  23000063  %LIB 4
   ?  23000073  %FMC 4
   ?  23000076  AAAA 4
   ?  23000077  %FIOCARD 4

   PROGRAM NAME ≠ IOTE, CORE  1768

   END OF COMPILATION - NO ERRORS
```

INTERNAL USE ONLY

INTERNATION COMPUTERS AND TABULATORS LIMITED

Programming Languages Division FORTRAN NOTE 17

I.C.T. 1900 Series 3.12.65.

Source and Semi-compiled Program Formats on Magnetic Tape

This note describes the structure of a composite magnetic tape file from which the FORTRAN 4 compiler can select specified subfiles as input.

Source and Semi-compiled Program Formats on Magnetic Tape

The first versions of the FORTRAN 4 compiler will accept only single-reel files and single record blocks. The layout is compatible with the proposed standard composite file formats (PDCC/171).

Tape Layout

The tape starts with a standard header label. The name may contain only the characters

Letters
Digits
Space
Hyphen

and must start with a letter.

The tape should end with an end-of-composite file Trailer Label, which is a tape mark followed by a 20 word qualifier block as follows:

Word	Contents or Use
0	# 00000007
1-19	Zero

Subfiles Between the labels may appear 0 or more subfiles. These must be simple subfiles, i.e. must not themselves contain subfiles. Each subfile starts with a subfile sentinel and ends with an end-of-subfile sentinel. The qualifier blocks are as follows. (The letter I indicates that the contents of the word are at present ignored by the FORTRAN compiler. They should be set as in PDCC 171 or left zero.)

Subfile Sentinel

Word	Contents or Use
0	# 00000006
1	I
2-4	Subfile name. May contain letters, digits, spaces and hyphens, and must start with a letter.
5-11	I
12	Indicates whether source or semi-compiled subfile as follows.

Word	Contents or Use
	B2F4: Source
	A200: Consolidated semi-compiled program; i.e. preceded by GPL etc.
	A300: Semi-compiled program
13	I
14	21 (source)
	20 (semi-compiled)
15	1
16	I Intended to specify position of sequencing information in record. Likely possibilities for FORTRAN are # 00100107 Sequencing in columns 73-80.
	0 No sequencing
17-19	I

End-of-subfile Sentinel

Word	Contents or Use
0	# 40000000
1	Count of blocks in subfile (excluding sentinels).
2-3	Zero
4-19	User information.

Source Subfiles

Within a source subfile may appear one or more segments and/or any steering statements allowed by the compiler. Each line of source is held by a record of between 2 and 21 words. The first word contains a count of the number of words in the record. Each record is a block (which is zero-filled up to five words if necessary).

Semi-compiled Subfiles

Within a semi-compiled subfile may appear one or more segments and/or any other records of semi-compiled allowed by the FORTRAN consolidator.

Each record is in standard semi-compiled form with between 1 and 20 words.

Not more than 18 words can contain useful information; another two may hold sequencing information. Each record is a block.

<u>K. F. James.</u>

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED

Programming Language Division
I.C.T.1900 Series

FORTRAN NOTE 18
13.12.65

PERIPHERAL ROUTINES

This note gives a brief summary of the action of the currently available peripheral routines for FORTRAN 4, namely:-

- %FIOPT Paper Tape.
- %FIOLP Lineprinter.
- %FIOCARD Cards.
- %FIOMT Unformatted magnetic tape.
- %FIOMTF Formatted magnetic tape.

A knowledge of FORTRAN NOTES 7 and 5 (shortly to be replaced) is assumed.

PERIPHERAL ROUTINE

%FIΘPT

Input This routine reads, in shift mode, 124 characters or to terminator and converts the block to graphic mode.
α and β shift characters are ignored so that l.c. letters becomes u.c. and - (UL) becomes @. Translation terminates on TC4, NL, FF, CR, LF and any of these except TC4 indicates end of record. DC4, TCO and ERASE are ignored. Other δ shifts become ! except], ←, $, ↑ which become their graphic equivalents. Initially, single NL characters are ignored, until the first non-empty block is read in. After that, all records are space-filled to 80 characters if necessary, A record may contain any number of blocks, so there is no maximum record size.

Output Punches in graphic mode. When %FINΘUT indicates end of records a NL character and 3 runout characters are punched. A record may contain any number of blocks, so there is no maximum record size.
On ENDFILE a block consisting of FF, TC4, DC4, 3 Erases and NL is punched, followed by 12" of runout.
RUNOUT gives 4" of runout.

%FIΘLP

For formatted records the first character in the buffer is taken as the paper control character translated as follows:-

0	Two line feed
1 - 7	Throw to Channel 1 - 7
+	No advance
space	1 line feed

Any other character is treated as space.

For unformatted records, single line feed is inserted.

The buffer is space-filled to 120 character positions to allow for buffered and unbuffered printers. Long lines are continued, using the first character of the continuation as a control character.

ENDFILE and RUNOUT will throw to head of form.

%FIΘCARD

<u>Input</u> This routine reads 80 column of a card in normal mode for both formatted and unformatted records. Initially blank cards are ignored for formatted records.

<u>Output</u> The buffer is space-filled if necessary and 80 characters are punched.

ENDFILE punches 1 blank card.

RUNΘUT punches 2 blank cards.

%FIΘMT (unformatted) and %FIΘMTF (formatted)

One reel holds one simple file. The layout is:-

i) Header label

ii) Start of data sentinel

iii) Data blocks

iv) End of file trailer label

For details of the layout of individual blocks see Appendix 1

The tape is opened when the first read, write or end file instruction is given. Checks are made to prevent an attempt to input from an output only tape. An input tape must be named in the program description, If no name is given for an output tape, a scratch tape is found. Input only tapes are checked for the absence of a WP ring, output and use tapes for the presence of a WP ring.

A marker is kept in the information block (see Appendix 2) of the last instruction to be given to the tape, and this is checked for appropriate action and to prevent illegal combinations, i.e. READ after WRITE or ENDFILE, and WRITE after ENDFILE. Most combinations are straightforward except:

<u>BACKSPACE</u>. If the tape is not open, or the data block count is zero this instruction is ignored. If the previous instruction was WRITE, an end-of-file trailer label is written and backspaced. over before the normal procedure is carried out. This is to backspace two blocks, read forward and check the block read:

i) tape mark - i.e. last instruction ENDFILE - straight exit.

ii) start of data sentinel - i.e. only one data block - one is subtracted from data block count and then exit.

iii) data block - a) formatted tape: one subtracted from data block count and exit.

b) unformatted tape: word 3 of the block is checked to see if last block of record. If it is, as a). otherwise n + 1 blocks are backspaced where n is the block number relative to the start of the record (given in word 2), and one block is read forward. Then n + 1 is subtracted from the data block count and exit.

REWIND. If the tape is not opened, the instruction is ignored, If the last instruction was WRITE, an end-of-file trailer label is written, then a rewind instruction is given and exit. When the next READ, WRITE or ENDFILE instruction is encountered, before being carried out the tape will be positioned after the start of data sentinel. This allows time sharing as the tape is rewinding.

For a summary of tape instructions, see Appendix 3.

Alison Finch.

Appendix 1

Magnetic tape layout

FORMATTED		UNFORMATTED

Header block

Standard		Standard
Generation no. zero on o/p ignored on i/p		

Start of data sentinel

FORMATTED		UNFORMATTED
BO = 1, B22 = 1)) Other bits = 0)	wd 0	(BO = 1, B22 = 1 ((Other bits = 0
BL size = 42	wd 1	BL size = 124
0	wd 2, 3	0
BO = 1)) Other bits = 0)	wd 4	0
Reserved for FORTRAN	wds 5-9	Reserved for FORTRAN
zero on o/p)) Ignored on i/p)	wds 10-19	(zero on o/p ((Ignored on i/p

Data blocks

FORMATTED		UNFORMATTED
no. of wds in record (2≤ n 42)	wd 0	no. of wds in block (5≤ n ≤124)
chars 0, 1 zero char 2 = I char 3 = 1st char of rec.	wd 1	block no.
RECORD	wd 2	Block no. rel to start of rec.
	wd 3	BO = 1: last bl of rec. BO = 0 otherwise. last 12 bits: no of information
	wd 4	RECORD.

Trailer label

wd 0	BO = 1, other bits = 0
wd 1	Data blocks count
wds 2, 3	0
wd 4 - 9	reserved for FORTRAN
wd 5 -19	zero on o/p
	ignored on i/p

A formatted tape has one record per block.

An unformatted tape has 1 or more blocks per record.

Both magnetic tape formats are compatible with MTH convention.

Appendix 2

Structure of Information block for magnetic tape (see FORTRAN NOTE 5)

Wd 0	Address of %FIθMT or %FIθMTF
Wd 1)	Device details
Wd 2)	
Wd 3	B0 - B5 : instruction type
	rest of wd: data block count
Wd 4	zero : scratch tape
	non-zero : named tape
Wd 5 onwards	File name and other parenthesised information from Program Description line in character form (terminated by right parenthesis.)

Instruction type		
	0	read
	1	write
	3	endfile
	4	backspace
	#40	rewind

Appendix 3

CURRENT	PREVIOUS OPERATION					
OPERATION	None	Read	Write	Backspace	Rewind	Endfile
READ	Open then OK	OK	ERROR	OK	RF2B OK	Error
WRITE	Open then OK	OK	OK	OK	RF2B OK	ERROR
BACKSPACE	Ignore	OK	Endfile Backspace OK	OK	Ignore	OK
REWIND	Ignore	OK	Endfile OK	OK	Ignore	OK
ENDFILE	Open then OK	OK	OK	OK	RF2B OK	Ignore

FORTRAN 4 Magnetic tape operations.

RF2B : skip to tape mark and read start of data sentinel.

Certain of the above combinations will give unpredictable results if an input tape was not produced either by a FORTRAN program or to FORTRAN standards.

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED

Programming Languages Division FORTRAN NOTE19

I.C.T. 1900 Series Date 13.1.66.

Program Description and Intersegment Statements.

This FORTRAN NOTE describes the program description and intersegment statements which are shortly to be implemented in A.S.A. Fortran. It supersedes the appropriate sections of FORTRAN NOTES 2,3 and 7.

GENERAL

This document describes the Program Description and intersegment statements which will appear in A.S.A. FORTRAN. It is intended to retain the existing Program Description statements for a short time as well. Thus existing programs may still be compiled but should be modified as soon as possible to conform to the new program description formats. All the statements are written in the normal FORTRAN manner in columns 7 to 72 of a line, spaces being ignored (except in file and subfile names). However, no continuation lines are allowed for these statements.

A complete program is made up of a program description, intersegment statements and source or semi-compiled program arranged in the following manner.

a) Intersegment statements (see note below).

b) Program Description.

c) Source or semi-compiled program mixed as required with intersegment statements.

d) The statement FINISH.

NOTE: The first statement presented to the compiler should define the listing peripheral and the listing mode required, e.g. LIST (LP). By default SHORTLIST (LP) is assumed. The second statement should define the output medium, but is, at present, redundant since the type of compiler used automatically defines the output medium.

PROGRAM DESCRIPTION.

The Program Description may be considered as a segment introduced by one of the statements PROGRAM or SEGMENTS and terminated by the statement END. It must be the first segment read in any A.S.A. FORTRAN compilation but may optionally be preceded by intersegment statements. The program description is compiled as a "Master" segment with a "Force-in" Bit.

1. The PROGRAM Statement.

The PROGRAM statement introduces the Program Description and specifies the name of the object program. It has the forms:-

PROGRAM (n)

PROGRAM (nc)

where "n" is the name to be given to the object program and "c" is an accounting code.

"n" consists of 4 alphanumeric characters starting with a letter. "c" consists of up to 8 alphanumeric characters and is optional, but may be insisted upon at som. installations which may ·se it to charge the user for the compilation and running of a program.

The PROGRAM statement is essential if a complete program is to be consolidated into a loadable, semi-compiled object program.

2. The SEGMENTS Statement.

The SEGMENTS statement is used instead of the PROGRAM statement when one or more segments are to be compiled, but not consolidated into a loadable program at this stage; i.e. request slip, loader and consolidated leader are not to be output. The statement has the forms:-

SEGMENTS

SEGMENTS (n)

SEGMENTS (nc)

where "n" and "c" are the same as in the PROGRAM statement. The second form is necessary if output is to magnetic tape and the third form is necessary if an accounting code is obligatory.

3. The TRACE anc NOTRACE Statements.

The TRACE and NOTRACE statements indicate to the compiler whether to compile in trace mode or not. They take the forms:-

TRACE

NOTRACE

These statements may also appear between segments to compile different segments in different modes. The effect of TRACE and NOTRACE is to set and reset a switch, initially NOTRACE is assumed by default.

4. The OVERLAY Statement.

The OVERLAY statement specifies which segments of the program are not to be in permanent program and which overlay area and unit they are to appear in. It has the form:-

OVERLAY (a,u) s_1, ---, s_n

where "a" is an integer between 1 and 255 inclusive and is the overlay area, "u" is an integer between 1 and 1023 and is the overlay unit, and "s_i" are the names of the segments to appear in this overlay unit of this overlay area. Segments which may appear in an overlay unit are SUBROUTINE segments with not more than 20 dummy arguments.

5. <u>The PRIORITY Statement</u>

The PRIORITY statement specifies the priority to be assigned to the object program and has the form:-

PRIORITY n

where "n" is an integer in the range 1 - 99.

It is only relevant if the Program Description is introduced by a PROGRAM statement.

Omission of PRIORITY implies

PRIORITY 50

6. <u>The Peripheral Statements.</u>

These are the INPUT, OUTPUT, USE and CREATE statements. They describe what peripheral devices are to be used by the object program and they have the general format:-

TYPE n_1, ----, n_k = peripheral information

where:-

"TYPE" is one of INPUT, OUTPUT, USE or CREATE.

INPUT and OUTPUT may be used with any peripheral indicating that the peripheral is to be used only for input or output respectively. USE and CREATE may only be used with magnetic tape, USE incidating that the tape may be used for both input and output, and CREATE indicating that a Scratch Tape is to be opened, re-named and may then be used for both output and input.

"n_i" are the values by which the peripheral device is known within the object program. They are either small integers, 1 to 4095 inclusive, or (MONITOR) which indicates that the peripheral concerned is to be used as the input or output device for TRACE information.

"peripheral information" designates the peripheral device, "p", which is known in the object program by the values "n_i". It may also give auxiliary information, "a", such as file name, and a qualifier, "q", may also appear to indicate that a special peripheral routine is to be used.

The format is one of

p/q (a)

p/q

p (a)

p

where:-

"p" is the usual code for peripheral devices, e.g. CRO, LP1, MT7.

"q" may only be, at present, "FORMATTED", which may be used in conjunction with magnetic tape to indicate that formatted information is to be recorded on and/or read from it. Omission at "FORMATTED" implies unformatted information.

"a" is used at present only to give the file name of the magnetic tape and consists of up to 12 characters chosen from A to Z, 0 to 9, space and hyphen. In the case of magnetic tape omission of "a" will imply a scratch tape. The auxiliary information will shortly be extended to specify subfile name, retention period and generation number.

7. The COMPRESS Statements.

These statements have the forms:-

COMPRESS INTEGER AND LOGICAL

COMPRESS DOUBLE PRECISION

The first one causes all Integer and Logical variables to be assigned one word of core store instead of two. It should be emphasised that this does not reduce the range of integer constants since although they normally occupy two words, to conform with the ASA specification, only the first of these is used to hold the constant, the second being ignored. Thus in either case integer constants are held in one word length only. The second statement causes all Double Precision variables to be treated as Single Precision (Real) variables and assigned two words of core store instead of four. These two facilities should be used with caution particularly if they affect variables used in COMMON or EQUIVALENCE.

8. The ON and OFF Statements.

These statements may be used to switch bits of word 30 of the compiler on or off and have the form:-

ON n_1, - - - - , n_n

OFF n_1, - - - - , n_n

where "n_i" are integers between 0 and 23 and indicate the bits of word 30 to be switched on or off.

These statements may also appear between segments. They are intended for use only by the compiler writers.

9. The LEADER Statements,

This statement, which has the form:-

LEADER

causes the Consolidated Leader to be listed as described in FORTRAN NOTE 16.

10. The COPY and NOCOPY Statements.

These statements, which have the forms:-

COPY

NOCOPY

cause any acceptable semi-compiled program to be copied or not copied, depending on the statement. These statements will be ignored when output is to magnetic tape as everything must be copied.

They may appear between segments to copy segments as required.

Omission of either normally implies COPY.

11. The END Statement.

This statement, which has the form:-

END

must terminate the Program Description.

INTERSEGMENT STATEMENTS.

The following are the only statements which may appear between segments.

1. The LIST and SHORTLIST Statements.

These statements indicate the type of listing required and on what device it should be output. They have the forms:-

LIST

LIST (p)

SHORTLIST

SHORTLIST (p)

where "p" is LP or TP for listing on a line printer or paper tape punch respectively. Omission of "p" implies LP.

LIST gives a full listing of source (except semi-compiled) plus diagnostic information and a listing of control statements. SHORT LIST gives a listing of control statements, segments headings and diagnostic information.

The form:-

LIST (TP)

will be interpreted as SHORT LIST(TP).

By default SHORTLIST (LP) is assumed.

LIST or SHORT LIST should be the first statement of any program, to indicate which peripheral to list on. They may also appear between segments so that some segments may be full listed and others short listed.

2. The TRACE and NOTRACE Statements.

See Program Description number 3.

3. The COPY AND NOCOPY Statements.

See Program Description number 10.

4. The ON and OFF Statements.

See Program Description number 8.

5. The READ FROM Statement.

This statement specifies the peripheral from which the subsequent source segments are to be read and has the forms:-

READ FROM (p)

READ FROM (MT, f)

where "p" is CR or TR for reading from a card reader or paper tape reader respectively and "f" is information about the magnetic tape file or subfile to be read, in a form yet to be specified.

6. The SEND TO Statement.

This statement specifies the output peripheral to receive the object program. Versions of the compiler that permit output to only one fixed slow peripheral (paper tape or cards) will ignore it. It has the forms:-

SEND TO (p)

SEND TO (MT, f)

where "p" is one of TP, CP or MT indicating output to a paper tape punch, card punch, or magnetic tape respectively; and "f" is information about the output file in a form yet to be specified.

This statement must immediately precede the PROGRAM or SEGMENTS statement of the Program Description.

7. The FINISH Statement.

This statement has the form:-

FINISH

and has the following effects.

1) If these have been errors the compiler is suspended and the console message:-

HALTED ZZ

is typed. It is not then possible to continue compilation.

2) The mode of compilation is switched to "selective"; i.e. segments read after a FINISH will only be accepted if they have either been called for or have a "force-in" bit.
3) If there are any unsatisfied cues the FORTRAN library is scanned (or the leaders read from a LIBRY segment in paper tape output versions).
4) When all cues become satisfied consolidation will be completed and Request Slip, GPL and Consolidated Leader output.
5) If there are any unsatisfied cues remaining after 3) above, the compiler is suspended and the console message:-

    NEEDS c

   is typed, where "c " is one of the unsatisfied cues. A list of all the unsatisfied cues is output on the listing peripheral.

   Compilation may be continued by typing the console message:-

    GO # name

   where "name" is the name of the compiler.

8. <u>The LIBRARY Statement.</u>

This statement, which has the form:-

    LIBRARY

has the effects 1), 2) and 4) given in the FINISH statement (see above). It is useful if a programmer wants his own library read before the standard FORTRAN library.

N.F. Bearne.

APPENDIX I

This appendix summarises the statements available. Any parameter within square brackets is optional.

Program Description Statements.

1) PROGRAM (n[c])
2) SEGMENTS [(n[c])]
3) TRACE
   NOTRACE
4) OVERLAY (a,u) s_1,---, s_n
5) PRIORITY n
6) INPUT n_1, ----, n_n = pn [/q] [(a)]
   OUTPUT n_1, ---, n_n = pn [/q] [(a)]
   USE n_1, ---, n_n = MTn [/q] [(a)]
   CREATE n_1,---, n_n = MTn [/q] [(a)]
7) COMPRESS INTEGER AND LOGICAL
   COMPRESS DOUBLE PRECISION
8) ON n_1, --- , n_n
   OFFn_1, --- , n_n
9) LEADER
10) COPY
    NOCOPY
11) END

Intersegment Statements.

1) LIST [(p)]
   SHORT LIST [(p)]
2) TRACE
   NOTRACE
3) COPY
   NOCOPY
4) ON n_1, ---, n_n
   OFF n_1, ---, n_n
5) READ FROM (p[,f])
6) SEND TO (p[,f])
7) FINISH
8) LIBRARY

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED

Programming Languages Division
I.C.T. 1900 Series

FORTRAN NOTE 20
18.1.66

Provisional specification of the 1900 Fortran compiler

This specification relates to the pre-release versions of the ASA Fortran compilers; ##ZFAM (magnetic tape version) and ##ZFAP (paper tape version). The compilers due for release on 1.3.66 will be named ##XFAM and ##XFAP and will not necessarily be to the same specification as given in this note.

This series of notes is intended for reference by the writers of the compilers and associated software. This particular note and Fortran Note 19 is being given a wider circulation than usual because they give details of interest to programmers using the ASA Fortran compilers.

- DRAFT -

(PROVISIONAL)

#ZFAM
#ZFAP

TITLE

1900 Fortran compiler

ISSUE

1

HARDWARE REQUIREMENT

	#ZFAM	#ZFAP
Core store:	12000 words	13000 words
Input peripheral:	1 paper tape reader and/or	1 paper tape reader and/or
	1 card reader and optionally	1 card reader and optionally
	1 or more magnetic tape decks	1 or more magnetic decks
Output peripheral:	1 magnetic tape deck	1 paper tape punch
Listing peripheral:	1 lineprinter and/or	1 lineprinter and/or
	1 tape punch	1 tape punch
Other peripherals:	2 magnetic tape decks (Work tape plus program library tape)	None

USE OF PERIPHERALS

Source program and/or semi-compiled program may be input from either paper tape, cards, or magnetic tape. Initially, according to choice, a paper tape reader or card reader is allotted; thereafter the input device is selected dynamically under control of the program. A paper tape or card reader is retained throughout a compilation, even if the bulk of the source is on magnetic tape.

Normally a lineprinter is allotted at the start of the run to receive a listing of the source program and error information, if any. A tape punch may be used as an alternative, in which case a listing will be given in an abbreviated form.

The other peripherals used depend on the version of the compiler.

a) #ZFAN

A scratch magnetic tape is opened at the start of a run and used as a work tape. It is left scratch.

A magnetic tape, normally the program library tape, containing the Fortran library must be available throughout the run.

At the end of a successful compilation a second scratch tape, or a tape with a specified name, is opened, given a specified name and retention period, and used to receive the object program.

b) #ZFAP

A paper tape punch is used to receive the object program. Output will cease on the occurrence of an error, and the punch will be released.

Allocation and release of peripherals:

All peripherals are allocated dynamically as and when required.

A special entry point is provided to release all peripherals. During the course of a run peripherals will only be released when a change is made (e.g. switching from cards to paper tape input).

Since a list is kept within the compiler of all peripherals being used, console allocation and release of peripherals should be avoided.

DESCRIPTION

a) General

The A.S.A. FORTRAN IV compiler will translate FORTRAN segments into standard 1900 semi-compiled form and will optionally attempt to consolidate these segments, and any other semi-compiled segments that may be provided, into a complete object program. The FORTRAN segments should be written in the version of the language described in the "I.C.T. 1900 Manual" (Provisional edition Dec. 1965).

For a complete program the compiler will automatically produce an executable program, complete with General Purpose Loader. For one or more segments not forming a complete program the compiler will produce only semi-compiled segments. At a later stage these segments may be re-input to the compiler along with other FORTRAN and/or semi-compiled segments that together form a complete program.

For the A.S.A. Fortran IV compiler a Program Description must be provided. This is read prior to input of source or semi-compiled program, and enables compile time and object time parameters to be specified by the user. Further details are given under the heading INPUT PARAMETERS.

During compilation, unless specified otherwise, a listing is given on a lineprinter of the source program and error information (if any).

b) Input

Input to the compiler may be on paper tape in the standard 1900 eight-track code and/or on punched cards in the 1900 card code. Source or semi-compiled program may also be input from magnetic tape.

A program basically consists of one or more segments followed by a FINISH statement. There must be one and only one MASTER segment. The program may contain only FORTRAN segments, only semi-compiled segments or a mixture. Segments may be in any order: in particular, the MASTER segment need not appear first.

All of these segments will be incorporated in the object program, except that if two or more segments with the same name are input, only the first one is incorporated. The FINISH statement causes the compiler to arrange for any required segments from the FORTRAN library to be incorporated. This includes segments that are private to the Fortran compiler, e.g. Fortran Input/Output package.

If the user has a series of subroutine and function segments that form a private library they may be read after a LIBRARY statement and should precede the FINISH statement. Before a FINISH or LIBRARY statement the compiler will incorporate into the object program all segments encountered. After a FINISH or LIBRARY statement has been encountered only those segments to which a reference has previously been made are incorporated. Segments in a private library will normally be in semi-compiled form, but they could be in FORTRAN source form.

Any semi-compiled segments input to the FORTRAN compiler may have been previously produced by the FORTRAN compiler or by the PLAN assembler.

The compiler can start reading from either paper tape or punched cards. Wherever there is to be a switch from one medium to the other the programmer should insert one of the following statements:

READ FROM (CR)	Switch from paper tape to cards.
READ FROM (TR)	Switch from cards to paper tape.
READ FROM (MT,.....)	Read subfile from magnetic tape specified and then continue reading from slow peripheral.

As usual, the statements should not start before character position 7 of the line.

c) <u>Output</u>

The Program Description allows the user to specify that either;-

A) the source is to be compiled to produce one or more semi-compiled segments, in which case the output is in semi-compiled form on magnetic tape (#ZFAM) or paper tape (#ZFAP).

or B) the source is to be compiled and consolidated to produce a loadable program as described below.

i) #ZFAM

If no errors are found and the program is complete, the output is a magnetic tape containing an executable program ready to be input by an Executive FIND directive.

If no errors are found but the program is not complete, there is no useful output unless the operator has been requested to force the output of a General Purpose Loader. Then the output will be as above. This action is sensible only if it is known that the missing segments, although referred to, will not be needed when the program is run. If errors are found there is no useful output and consolidation cannot be completed.

ii) #ZFAP

If no errors are found and the program is complete, the output is a length of paper tape containing:

a) semi-compiled segments.

b) runout.

c) request slip and General Purpose Loader.

The request slip and General Purpose Loader are torn off. When the program is to be run they must be input <u>before</u> the semi-compiled segments.

If no errors are found but the program is not complete the output is a length of paper tape containing only semi-compiled segments. If the operator has been requested to force the output of the General Purpose Loader, the output will be as in the last paragraph. This action is sensible only if it is known that the missing segments, although referred to, will not be needed when the program is run.
If errors are found, the output is a length of paper tape containing any semi-compiled segments produced before the first error occurred plus an incomplete semi-compiled version of the segment in which the first error was found.

d) Listing

During the course of compilation a listing will be produced on either a lineprinter or a tape punch. On the lineprinter two forms of listing are allowed; either a short list or a full list, the difference being that, except for segment names, source statements are not printed in the case of short list. Unless otherwise specified short list is assumed and if the tape punch is used then only short list is allowed.

Statements occurring within the program description or between segments are always listed. The occurrence of semi-compiled program will cause one line to be printed giving the segment name e.g.

SEG2B (SEMI-COMPILED)

Additional information is output by the compiler giving the compiler name, date and time run, name and number of instructions of each segment compiled, program name and core store required and the number of errors detected.

e) Error Interpretation

If the compiler detects an error in the input it ceases to output object program but continues to search for errors. Errors discovered by the compiler will be output on the listing peripheral in the form:-

ERROR n IN LAST STATEMENT, LINEp, CHARq

where n is the error number

p is the line number within the statement (first line = 0)

q is the character number (first character = 1) and gives an indication of the last field investigated by the compiler before the error occurred; the actual error may have occurred earlier.

The significance of the error number is as follows:-

ERROR NUMBER	INTERPRETATION
1	Mis-matched parenthesis.
2	Missing parenthesis
3	Some other character found when /) or, was expected.
4	Error in format of arithmetic expression.
5	Constant out of range.
6	Should be an integer variable.
7	Array not declared, or declared twice.
8	First character of name non-alphabetic, or name omitted.
9	Statement too long.
10	Statement incomplete
11	Label set twice or error in columns 1 to 5.
12	Error in label reference
13	Illegal subscript, e.g. too many or too few subscripts.
14	Illegal name, e.g. subroutine has illegal name.
15	Label missing
16	Do label not found.
17	Statement not recognised or out of context.
18	Error in statement other than a source statement.
19	Polish list exhausted.
20	Compiler error.

The above interpretation of errors having a code number in the range 1 to 25 applies regardless of statement type, but the interpretation of error numbers above 25 depend on statement type as follows:-

Statement	Error Number	Interpretation
MASTER	26	Master name in overlay list.
FUNCTION	26	Segment name omitted.
	27	No arguments
SUBROUTINE	26	Segment name omitted.
CALL	26	Not subroutine name.
GO TO	26	Error in statement after GO TO.
	27	Error after) of computed GO TO
DO	26	Do label followed by end of statement.
	27	'=' missing or misplaced.
	28	Parameters not integer expressions?
IF	26	Do Statement in 'Logical IF'.
FORMAT	26	Unlabelled.
READ	26	Unit number not integer constant. or integer variable.
WRITE		
	27	Format reference error
	28	Error in list following.
INTEGER	26	Type already specified.
REAL		
LOGICAL		
DOUBLE P		
COMPLEX		
COMMON	26	Illegal name for common block.
	27	Illegal item in list.
DIMENSION	26	No dimension given.
EQUIVALENCE	26	Dummy argument.
	27	Illegal argument, e.g. constant, statement label, subroutine name.
	28	Argument in list is followed by '(' and it is not an array name.
	29	Argument already equivalenced or otherwise defined so that equivalence is impossible.

DATA	26	Illegal name in variable list, e.g. a subroutine name.
	27	Illegal field in constant list.
	28	Illegal form of complex constant.

When short list mode is specified the first line only of the statement in error is printed prior to the above error message.

If on the occurence of the END statement there exists any labelled statements which have not yet been found then for each one a message will be output, following the END statement, of the form

ERROR n - LABEL 1

where n is the error number (see above)

l is the label not found.

INPUT PARAMETERS

The Fortran IV compiler permits a very versatile set of input parameters to be supplied either as part of the program description or as control statements between segments. A full description is given in Fortran Note 19.

A complete job presentable to the compiler is of the form:-

1. Initial Statements. (e.g. LIST(LP))
2. Program Description.
3. Source or semi-compiled segments mixed as required with statements between segments.
4. The FINISH statement.

STORE SPACE USED

#ZFAM	12000 words
#ZFAP	13000 words

PRIORITY

The program as supplied has a priority of 50.

OPERATING INSTRUCTIONS

(a) ≠ZFAM

1. Load program ≠ZFAM

2. Activate by one of the messages

GO ≠ZFAM AT 20	Compile, reading initially from paper tape.
GO ≠ZFAM AT 21	Compile, reading initially from cards.

   A paper tape reader or card reader is allotted and a scratch magnetic tape opened as a work tape.

3. Load the first paper tape or pack of cards on the reader. Ensure that another scratch tape is available to receive the object program and that a tape containing the FORTRAN library is available.

4. Load any further paper tapes or packs of cards or magnetic tapes in the specified order. Whenever there is a change of medium one reader is released and another of the appropriate type is allotted. However, when a magnetic tape is allocated the card or paper tape reader is not released since control will be passed back to the slow peripheral when the magnetic tape has been read.

5. After a successful run, another scratch tape or a named tape is opened and labelled, and the compiled program copied to it. The end-of-run message

   ≠ZFAM HALTED:- COMPILED 'Name' EC

   is then output on the console, where 'Name' consists of 12 characters; the first four being the name of the program; the remainder being the charge number or accounting code. The work tape is left scratch.

6. Return to step 2 to compile another program.

(b) ≠ZFAP

The operating instructions are the same as above with ZFAP in place of ZFAM except that output is to paper tape in the usual manner, (i.e. semi-compiled, followed by the loader if consolidation has been completed successfully).
Magnetic tapes are not used, unless of course they contain input data.

EXCEPTION CONDITIONS

In the following, ##ZFA$_j$ stands for either ##ZFAM or ##ZFAP.

1. ##ZFA$_j$ HALTED:- NEEDS 'Segment'

This will occur, if on the occurrence of the FINISH statement, there remains one or more unsatisfied cues, the first of which will appear as 'Segment' above.

Further source or semi-compiled program may then be read after giving the console message:

GO ##ZFA$_j$

Alternatively consolidation can be forced by:

GO ##ZFA$_j$ 29 (See note (ii))

2. ##ZFA$_j$ HALTED:- COMPILED 'Name' ZZ

This indicates that errors have occurred. Consolidation cannot be completed but further programs may be compiled by returning to step 2 of the operating instructions. In the case of ##ZFAM the output magnetic tape is not allocated and the work tape is left scratch. If ##ZFAP was the compiler used then some useful semi-compiled program may have been output.

3. ##ZFA$_j$ HALTED:- PD

This indicates that an error has occurred either before or within the program description. It is possible to continue compilation by:

GO ##ZFA$_j$

but only for the purpose of checking for errors in the source program.

4. ##ZFA$_j$ HALTED:- OV

This will occur when one or more segments declared as overlay segments. It is necessary to use a separate overlay consolidator to produce a loadable program.

5. ##ZFA$_j$ HALTED:- TR
##ZFA$_j$ HALTED:- CR
##ZFA$_j$ HALTED:- TP
##ZFA$_j$ HALTED:- LP

The occurrence of one of these messages indicates that the peripheral mentioned is not available. When it is made available the run may be continued by giving the message:

GO $\#ZFA_j$

The peripheral will then be allocated.

6. $\#ZFA_j$ HALTED:- CE

When this occurs it indicates that the compiler has detected a check sum error while reading semi-compiled program.

If this occurs while paper tape is being read move the input tape back to the beginning of the block and type the message:

GO $\#ZFA_j$

If it happens again on the same block, mark the tape and terminate the compilation.

With #ZFAM, if there is a checksum error while reading the library from magnetic tape it is necessary to re-start the compilation from the beginning. This may be achieved by giving the message:

GO #ZFAM 27 (See note (iv) below)

and then continuing at step 2 of the operating instructions.

NOTES:

(i) If the compiler expects more input and none has been supplied by the programmer, suspend the compiler and re-activate by the message:

GO $\#ZFA_j$ 28

Action is then as for exception condition 1 above.

(ii) If the input is incomplete, consolidation may be forced by the message

GO $\#ZFA_j$ 29

Action is then as in step 5 of the operating instructions. If a list of blank cues is required, it should be obtained before the entry at 29.

(iii) If the compiler goes illegal or loops or otherwise comes to an unexplained stop it is recommended that the standard error procedure is carried out(See below).

Restart should then be possible by the message:

GO ##ZFA$_j$ 27 (See (iv) below)

and then continuing at step 2 of the operating instructions.

(iv) It is dangerous to 'take' or 'give' peripherals when using ##ZFA$_j$ compilers since this is likely to cause the compiler to go illegal. If at the end of a run it is desired to release any peripherals allocated to the compiler it is simply necessary to give the message:

GO ##ZFA$_j$ 27

STANDARD ERROR PROCEDURE

If illegal on instruction other than 150 - 157 then output the first 3,500 words of core store on the lineprinter. This information will be of use later to the compiler writer.

If illegal on instruction in the group 150 - 157 then the console log is usually sufficient to explain the cause.

OPERATING INSTRUCTIONS (FORTRAN Object Programs)

1. Load the program. If it is on paper tape, load a paper tape containing the FORTRAN library on the same reader.

2. Activate the program by the message

GO ##name AT 20

or as specified by the programmer. 'name' represents the name of the program.

3. If the program was compiled in trace mode, a steering list may be read at any time the program is suspended. Load the list on a card reader or paper tape reader and type the message

The list is read, the program is suspended and the message

##name HALTED:- ES

is output.

4. No trace information is output unless trace is switched on by the message

ON ##name 0

Trace may be switched off by

OFF ##name 0

5. If the program was compiled in trace mode, further information about the last 100 statements obeyed may be output if the program is suspended and then reactivated by the message

GO #name AT 29

The program will suspend and output the message

#name HALTER:- EP

This action should always be taken at the end of a run for a traced program, unless the program has deleted itself.

EXCEPTION CONDITION (FORTRAN Object Program)

N.B. This section applies only if the program was compiled in trace mode.

1. After any error stop, the program should be re-activated as in step 5 of the operating instructions.

One special type of error stop outputs the message:

#name HALTED:- AE

It should be treated as above.

18th Jan 1966. L.R. FAIRBROTHER.

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED

Programming Languages Division

FORTRAN NOTE 21
28.1.66

The FORTRAN IV Object Time Input/Output System

This note replaces FORTRAN NOTE 5. It gives a brief description of the input/output system used by FORTRAN IV object programs. The emphasis is on the methods of communication between the different parts of the system. Further information appears in Fortran notes 4, 7 and 18.

The FORTRAN IV Object Time Input/Output System

1. Outline of System

The FORTRAN IV input/output system is modular. At run time, only those parts that are needed will be present. The relationship between the different parts is shown diagramatically in Figure 1.

For each source language input/output statement the object program will contain one or more calls to %FINOUT. %FINOUT is a segment that carries out the bulk of the work of I/O in a way that is independent of the I/O medium involved. %FINOUT enters one of a series of peripheral routines to carry out actual I/O operations. Each peripheral routine is a segment.

For double precision variables a call is generated to %FIODP instead of %FINOUT, but %FIODP uses %FINOUT for all processing except numeric conversion.

Similarly %FIODA is called instead of %FIOTA for double precision arrays.

2. Communication between %FINOUT and the Peripheral Routines

%FINOUT does not know about the different I/O media but merely enters appropriate peripheral routines to perform the actual I/O operations. Communication takes place via three common areas: %FIOLIST, % FIOPER and %FIOLEN.

The program description for a FORTRAN IV program generates in the common area %FIOLIST a list of programmer's numbers that may be used by that program. Associated with these numbers are details of the actual devices and all auxiliary information required for the successful operation of these devices, including the addresses of the appropriate peripheral routines.

An initial entry to %FINOUT will search %FIOLIST for auxiliary information corresponding to a specified programmer's number. Then the one word area %FIOPER is set to point to this information for use by the peripheral routine.

The layout of the buffer is given in section 4. It is used to transmit information between %FINOUT and the peripheral routines during READ and WRITE operations. This information is transmitted in units of blocks, where one record may consist of one or more blocks. A formatted record is likely to consist of one block and an unformatted

record of several blocks. Formatted information will be in standard 6-bit code (without shifts). Unformatted information will be in binary form. The length of the buffer is the maximum of the lengths defined in the peripheral routines.

During a READ or WRITE operation the initial entry to the peripheral routine will set the 2 word common area %FIØLEN :

Word 1 (formatted) No. of characters of information in Buffer
(unformatted) " " words " "

Word 2 Start address of information

On reading, B0 of word 1 will be set to 1 by the peripheral routines when the last block of a record is read.

On writing, this bit is set to 1 by %FINØUT whenever a record is to be terminated.

3. Structure of %FIØLIST

%FIØLIST contains a list of information about the peripherals used by a program. It contains an entry for each of the INPUT, ØUTPUT, USE and CREATE statements in the Program Description. An entry consists of one or more pointers and an information block.

For each programmer's number in the statement there is a 2 word pointer:

1. Programmer's number
2. Address of information block

The first word of %FIØLIST contains the number of entries in the list.

For each occurrence of the word (MØNITOR) in a statement there is a similar pointer, except that the 1st word is n, n < 0. The value of n indicates whether the word is available for input or output being -1 for input and -2 for output. These pointers are for use by the object time diagnostic routines e.g. TRACE (%FMN4).

An information block contains the following items :

1. Address of the start of the Peripheral Routine (1 word).
2. Device details (2 words).
3. Zero word
4. Length in words of rest of block
5. Any further information (Optional.)

Before entering a peripheral routine, %FINOUT sets %FIOPER equal to the address of the information block.

Item 1 is used by % FINOUT.

Item 2 consists of 2 words as follows:

	Value of Word	Meaning
Word 1	Top bit = 1	Device Allocated
	Bits 1-8	Type of device (as in PERI control area)
	Rest of word = 1	INPUT
	= 2	OUTPUT
	= 3	USE
	= 7	CREATE
Word 2	Top bit = 1	Device not yet used.
	l.s. 12 bits = n	n is number in 1900 device name, e.g. 3 in MT3.

The top bits of these words are refered to and changed only within peripheral routines; some peripheral routines ignore them altogether.
Item 3 is used as working space for the peripheral routines, e.g. %FIOMT stores statement type in bits 0-5 and data block count in bits 6-23.
Item 4 is zero if there is no further information
Item 5 (if item 4 non-zero) contains file name plus any other information which may prove desirable. Only up to 12 characters of the file name are significant to executive, names less than 12 characters being space-filled on the right.

4. Structure of the buffer

For character peripherals, except the line printer, user information starts at the beginning of the first word; for the line printer, at character 3 of the first word.
On magnetic tape the first few words of the buffer are used as control words and user information starts at character 2 of word 2 for formatted tape (for compatability with #XQMP) and at word 5 for unformatted tape.

Although each peripheral routine may in theory use a separate buffer in practice all existing routines use the common area %FIOBUF.

Alison Finch

Diagram of Input/Output System

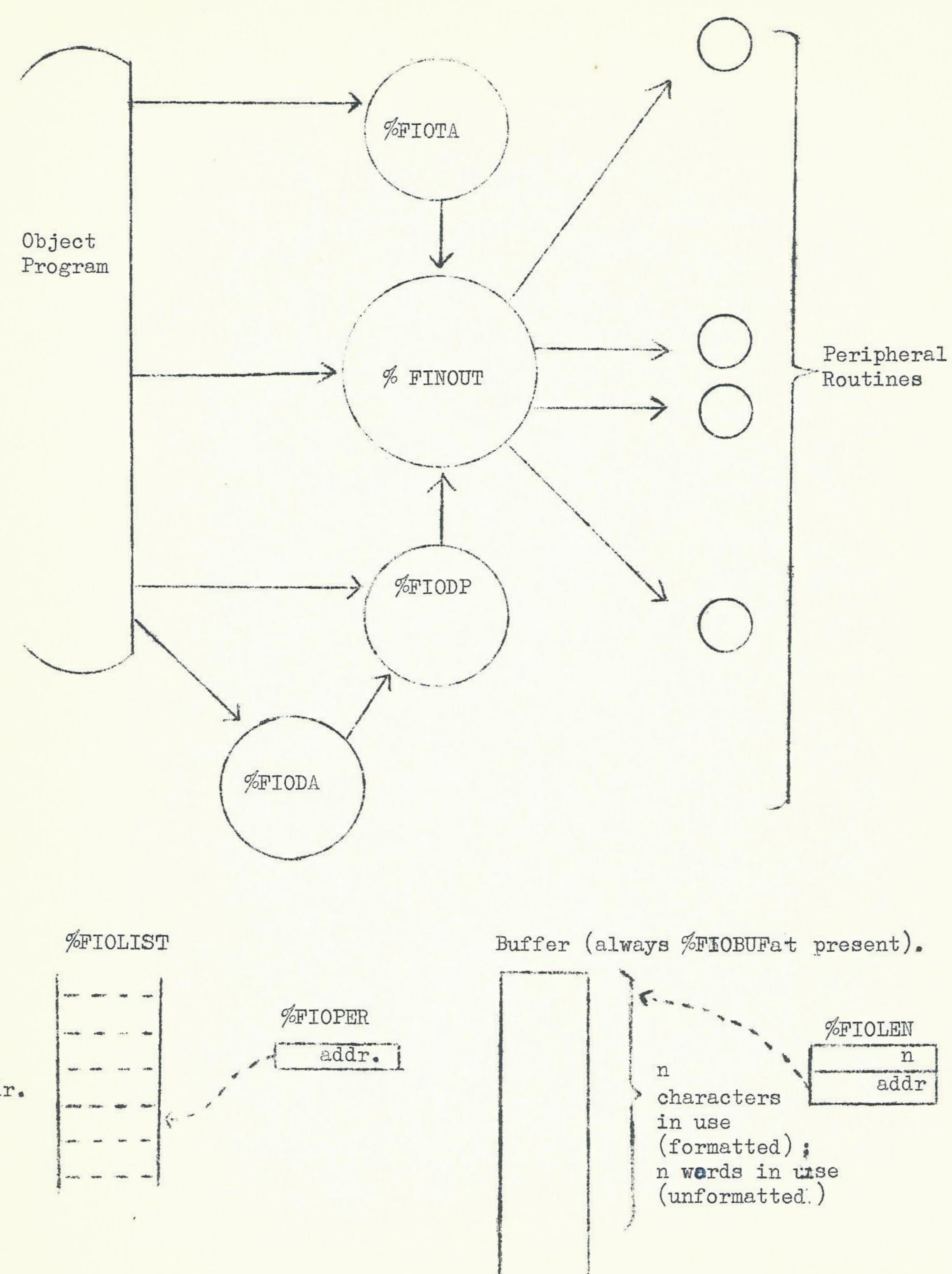


FIGURE 1

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED.

Programming Languages Division

FORTRAN NOTE 22

22/3/66

Dynamic Peripheral Allocation. (Compile time)

Dynamic Allocation System in Fortran IV compilers

Introduction: All peripherals are allocated and released dynamically by the compiler and a record is kept in a Lower Common area called 'TAPER' of peripherals in current use.

The area TAPER: This area is six words in length, each word referring to a particular type of peripheral, as follows.

Word 0:	Magnetic tape library
Word 1:	Magnetic tape input (source or s/c)
Word 2:	Slow peripheral input (card or paper tape)
Word 3:	Listing peripheral (printer or paper tape)
Word 4:	(a)Scratch magnetic tape OR (b) slow peripheral output.
Word 5:	Final output magnetic tape.

All these words are relevant to the magnetic tape compiler #XFAM, in which case alternative (a) for word 4 is the one applicable.

The paper tape compiler #XFAP and the card compiler #XFAC do not use words 0,1 and 5, and alternative (b) for word 4 is applicable.

Each word of TAPER is composed of the following elements.

Bit 0:	= 0 if not in current use (but see note (a) below)
	= 1 if in current use
Bits 1-8:	Indicates peripheral type (e.g. 3 for card reader)
Bits 9-23:	Indicates unit number (see note (b) below)

Notes: a) In all words of TAPER except word 1, a '0' in the sign bit implies that the peripheral is not allocated to the program. In the case of word 1, a '0' in the sign bit does not necessarily mean this, but indicates only that the peripheral is not in current use. This is necessary because it may be required that an input tape be left positioned at a certain point until it is known whether any further data is required from it.

In the case of word 1, therefore, a further indicator MTI is used: If MTI = 0, the peripheral is not allocated; if MTI = 1 it is allocated.

b) In #XFAM the following unit numbers are used for the magnetic tapes:-

Unit 0:	Tape containing library
Unit 1:	Final output tape
Unit 2:	Input tape
Unit 3:	Scratch Tape

For all three compilers, the slow input peripheral is unit 0 and the listing peripheral is always unit 1.

c) It is not of course possible for more than one of each kind of peripheral to be allocated to the compiler at the same time. Thus if a new input magnetic tape is requested, the old one is necessarily released first.

Release of Peripherals

The release of a peripheral is in all cases effected by entering the cued routine RPER, after setting X1 equal to the number of the appropriate word in TAPER. If RPER discovers that the peripheral is not allocated, it does nothing. Otherwise it removes the sign bit from the correct word of TAPER, then releases the peripheral.

If an input magnetic tape is to be released, RPER ascertains from the indicator MTI. whether the peripheral is allocated or not, instead of from TAPER+1. MTI and the sign bit of TAPER+1 are always left as zero.

Use of RPER:	The calling line is
	CALL 5 RPER
On input:	X1 = appropriate word of TAPER (0 to 5)
On exist:	X1 is unchanged.
Uses:	X2, X5, X6

Allocation of Peripherals

All peripherals are allocated dynamically by calling the cued routine APER.

The calling line is	CALL 5 APER
On input:	X1 = appropriate word of TAPER (0-5)
	X7 = counter modifier: Type/unit no.
On exit:	The allocation is complete.
	X1 and X7 are unaltered
	X3 = unit number
	X2 = Peripheral type
Uses:	All accumulators except X0.

On entry to APER, the counter/modifier in X7 is transferred immediately to the appropriate word in TAPER.

If the peripheral is not magnetic tape, APER will then attempt to allot it and if unsuccessful will type "Halted xx" where xx is CR, LP etc. When allotted, the appropriate sign bit in TAPER is set to '1'.

If a magnetic tape is to be allocated, more complex operations take place. APER looks at the common area MFIL to find out the name and generation number of the file required; it then sets up a control area accordingly and opens the tape. Finally the sign bit in the appropriate word of TAPER is set to '1' and if this word is word 1, MTI is also set to 1.

Initial allocation of peripherals

A listing peripheral is allocated if or when the user requests it by means of the intersegment statement LIST.

The subroutine STØUT makes the initial allocation of an output peripheral. In #-XFAM it allocates a scratch tape, writes a tape-mark to it and zeroises word 5 of TAPER. In # XFAP and # XFAC it allocates a tape punch or card punch respectively.

The subroutine STINP allocates the initial input peripheral, and this will always be a card reader or paper tape reader depending on whether the compiler is entered at word 21 or 22 or at word 20. Words 0 and 1 of TAPER are zeroised.

Change of Peripherals during compilation

This can be effected by means of the intersegment statement "READ FRØM".

For all three compilers, the following two forms are permissible:

(1) READ FRØM (TR)
(2) READ FRØM (CR)

For #XFAM (in which input may be from magnetic tape), the following are also permissible.

(3) READ FRØM (MT, Filename (G))
(4) READ FRØM (MT, Filename (G). Subfilename)
(5) READ FRØM (MT,. Subfilename)

Here, G is the generation number of the file and need not be specified. If not specified G = 0 is assumed. Type (3) can be used for obtaining input from a simple file. To obtain input from a composite file, types (4) or (5) must be used, and the subfile name must always be specified. Type (4) is used to open a tape and input a subfile. Type (5) can be used to input a subfile from a magnetic tape that is already opened; in this case the subfile requested must not be further back on the tape. Obviously it saves time to use type (5) rather than type (4) wherever possible; for type (4), if the tape is already open, it will first be released then re-opened.

Treatment of READ FROM by the compiler: The actual transfer from the current to a new input peripheral is effected by the cued routine RDFR.

The calling line is CALL 0 RDFR

On input: X1 = appropriate word of TAPER (0 to 5)
X7 = counter/modifer: Type/unit no
If magnetic tape, MFIL must be set up (see below)
On exit: X1 and X7 are unaltered.
Uses: All accumulators.

If reading from magnetic tape the Common Area MFIL is set up previously as follows:

Words 0, 1, 2: File name (word 0 = zero if not specified)
Word 3: Generation number (zero if not specified)
Words 4, 5, 6: Subfile name (word 4 = zero if not specified)

In the case of READ FRØM (CR) and READ FRØM (TR), RDFR merely releases the current peripheral (by calling RPER) and allocates the one requested (by calling APER).

The following, more complicated, operations are carried out if input from a magnetic tape has been requested:-

If a READ FRØM statement occurs while actually reading magnetic tape, it is simply ignored. Otherwise the first word of MFIL is examined. If it is not zero any magnetic tape input file currently allocated to the program is released and the one requested is allocated. If the first word of MFIL is zero, the compiler checks (from MTI) that a tape is indeed allocated and sets the appropriate sign bit in TAPER+1 to '1'.

The tape is then positioned ready for reading the first record to be input. In the case of a composite file, the compiler also notes from the subfile sentinel whether the contents of the subfile are source or semi-compiled. In the case of a simple file, only semi-compiled. In the case of a simple file, only semi-compiled input is acceptable.

The slow peripheral remains allocated to the program when a magnetic tape is being read, and control returns automatically to the slow peripheral in the following circumstances:-

(a) On detecting a tape-mark in the case of a simple file.
(b) On reading the end of the subfile in the case of a composite file.

In case (a), the tape is rewound and released, but in case (b) it is left positioned at the end of the subfile in case a further subfile is required from it at a later time (however, bit 0 of TAPER+1 is set to zero at this point).

Final Release of Peripherals

All peripherals which have been allocated during compilation are released before the final Halt is reached. Entry at word 27 of store will also cause all peripherals to be released followed by a typed message "Halted PF". This facility is included to make it possible to begin compilation again at any stage.

Halts in the Dynamic Allocation System

(a) The following halts indicate that a particular peripheral is not available to the compiler. To continue compilation the operating must put the appropriate peripheral on line and GØ:-

Halted TR	(paper tape reader)
Halted TP	(paper tape punch)
Halted LP	(line printer)
Halted CR	(card reader)
Halted CP	(card punch)

(b) The following halt occurs if entry has been made at word 27 to release all peripherals:-

Halted PF

This indicates that all peripherals have been released.

P. Girad.

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED

Programming Language Division

Fortran Note 23
22/3/66.

Compiler Magnetic Tape Input/Output System

The note describes the system developed in #XFAM for input from, and output to, magnetic tape in subfile format as specified in PDCC/171, issue 2. The document supersedes Fortran Note 17.

Magnetic Tape Input and Output for #XFAM

Introduction: All output from #XFAM is on magnetic tape. Input may be from cards, paper tape or magnetic tape, or from any combination of these (see description of the dynamic allocation system). The Fortran IV Library must be input from an MLT.

Magnetic Tape Input: Input may be from "simple files" or from "composite files", with the important restriction however that only semi-compiled programs or segments may be held on simple files.

A simple file consists of the following:-

(a) A Header label containing the file name, etc.

(b) TAPEMARK

(c) A semi-compiled program or semi-compiled segments.

(d) TAPEMARK

(e) Trailer label composed as follows:

Word 0 = #40000000
Word 1 = count of number of blocks in file
Words 2, 3 = Zero
Words 4 - 19 = Not used.

A composite file may hold source programs, source segments, semi-compiled programs and semi-compiled segments.

The tape is divided into subfiles each preceded by a subfile sentinel and terminated by end-of-subfile sentinel. At present a subfile may consist _either_ entirely of subfiles (these may be nested to any depth) _or_ entirely of program in one of the forms described above. No two subfiles sentinels on a file may contain the same subfile name.

Since a subfile sentinel contains one indicator recording the nature of its contents (i.e. source, s/c segments, or s/c program), a subfile containing program must contain it in only one of these forms. However this indicator is ignored in the case of subfiles composed of other subfiles; consequently a subfile can contain subfiles holding program material in any or all of the three forms.

A composite file is made up as follows:-

(a) A Header label containing the file name, etc.

(b) Subfiles (nested to any depth)

(c) End of composite file trailer label

A subfile is always preceded by a subfile sentinel having the form:-

Word 0 : #00000006

Words 2-4 : Subfile name (must start with a letter and must contain only letters, digits, spaces and hyphers)

Word 12 : B2F4 for source program or segments
A200 for consolidated semi-compiled program
A300 for semi-compiled segments
C100 for composite subfile

Word 14 : Maximum block size
(21 for source; 20 for semi-compiled).

Words 1,5 - 11, 13, 15- 19 : Not used at present

A subfile is always terminated by an end of subfile sentinel having the form:-

Word 0 : #40000000

Word 1 : Count of data blocks in subfile (excluding sentinels)

Words 2-3 : Zero

Words 4-19 : Not used.

An end of composite file trailer label is made up as follows:-

Word 0 : #00000007

Word 1-19 : Not used

Note : Each of the three types of block described above is preceded by a TAPEMARK.

Source and s/c record format

1) The maximum length of a semi-compiled record is 20 words. This is in the form of a simple "card image".

2) The maximum length of a source record is 21 words. The first word contains a count of the actual number of words in the record (including the first word). The other words are in the form of a simple "card image" of the source statement, though possibly containing fewer characters than a card.

Opening and positioning of magnetic tapes for input

This is described fully in connection with the "READ FRØM" statement of the dynamic allocation system. In all cases the tape is positioned ready to read the first data record.

Input of magnetic tape records

The record is initially read into a 21-word area called BUFF. Then, if it is source (this information is taken from the last subfile sentinel) the appropriate number of words are moved into CIMAGE starting of course at the second word of BUFF. If it is semi-compiled the appropriate number of words is moved into CIMAGE starting at the first word of BUFF.

Detection of a TAPEMARK is taken to indicate the end of the file in the case of simple files: The tape is released and reading continues immediately from the slow peripheral.

In the case of a composite file, reading continues until an end of subfile sentinel of the same level as the original subfile sentinel is detected. The tape is NOT released however, but remains positioned after the end of subfile sentinel in case the programmer requests further subfiles from it at a later stage. Reading continues from the slow peripheral meanwhile.

Halts in the magnetic tape input system

There are three possible halts. Each of these causes the same message to be output on both the console typewriter and on the listing peripheral

(a) Halted: ILLEGAL SENTINEL

This arises if a TAPEMARK is detected on a composite file, followed by a block other than a subfile sentinel, an end-of-subfile sentinel, or an end of composite file trailer label.

The compilation can be continued by typing 'GO' in which case reading continues as though no irregularity had been detected (i.e. ignoring the illegal sentinel)

(b) Halted : SUBFILE NOT FOUND

This arises if the request subfile is found to be not on the tape. The halt will so occur if the programmer has used a READ FROM of the type READ FRØM (MT,. Subfile Name), and the subfile is not between the point at which the tape was positioned and the end of file.

If required, compilation can be continued by typing GØ. This will cause reading from the slow peripheral to continue, so enabling the programmer to feed in another READ FRØM if he wishes.

(c) Halted : NO TAPE ALLOTTED

This arise if a READ FRØM (MT,. Subfile Name) is used, but no tape is currently under program control.

If GØ is typed, reading continues from the slow peripheral enabling the operator to read in another READ FRØM message if he wishes.

Listing in the Magnetic Tape Input System

Whenever reading takes place from a subfile, the contents are preceded on the listing (if listing is taking place) by the line

SUBFILE (Subfile Name)

Any other subfile detected within the main subfile are similarly listed. Also all end-of-subfile sentinels detected are recorded in the listing by the line

END OF SUBFILE

The abnormal halts, ILLEGAL SENTINEL, SUBFILE NOT FOUND, and NO TAPE ALLOWED are also recorded in the listing; these are described more fully in a separate section.

Magnetic Tape Output

All output in # XFAM is on magnetic tape. However there are several alternative types of magnetic tape output, which can be selected by preceding the program description by an appropriate "SEND TØ" statement.

The "SEND TØ"statement

The various possible SEND to statements can be classed under three main types:

Type 1 : SEND TØ (MT,'File Name'(G))

This would cause the output to be written as a simple file to a tape which is already named as indicated and whose generation number is G. G may be zero or omitted, in which case the generation number is not checked by Executive.

Type 2 (a) (SEND TØ statement omitted)
(b) SEND TØ (MT)
(c) SEND TØ (MT,'File name'(G), R)

Statements of this type cause the output to be written to a tape which is previously scratch. In cases (a) and (b) a retention period of 999 days is assumed. In case (c) R is the retention period and is compulsory unless it is required to write to a pre-named tape. In cases (a) and (b) the file name is taken from the following PRØGRAM or SEGMENTS statement. In case (c) it is taken to be as stated, the generation number (G) being optional.

Type 3 SEND TØ (MT,'File name'(G).'Subfile name')

This would cause the output to be written as a subfile (having the name specified) added to the end of an existing composite file. The generation number (G) of the composite file, need not be specified.

Treatment of SEND TØ by the compiler

When a SEND TØ statement is detected (or if it is absent) the compiler sets up a common area MFIL2 as follows:-

All three types:	Words 0 - 2	: File Name
	Word 3	: Generation Number (zero if unspecified)
	Word 4	: Type (i.e. 1,2 or 3)
	Word 8	: Not used at present
	Word 11	: Retention period (type 2 only)

For type 3, additional details are included:-

Words 5 - 7	: Subfile Name
Word 9	: A200 if consolidation required A300 if SEGMENTS only.

This information is not used further by the compiler until completion is complete.

If the output is 'Segments' only a no-consolidation switch is also set.

Summary of Output Operations

Initial Allocation

The routine STØUT is called soon after entering the compiler. This allocates a scratch tape and zeroises word 5 in TAPER.

Use of Scratch Tape

The compiler continues outputting to the scratch tape until either (a) errors are found in the input or (b) a FINISH is detected.

In case (a) the outputting of blocks in discontinued, and the scratch tape is released by calling the routine FERRR (from segment U)

Action on detecting a FINISH

1. If in error mode : Any peripherals still allocated are released and the compiler halts ZZ

2. If there are no errors, but no consolidation is required: The compiler enters a routine called TSØØ (from segment A). This terminates the scratch tape with two Tape Marks and rewinds it; the final output tape is then allocated and positioned ready for writing. The routine ENOUT finally copies the scratch tape to the final output tape, after which any remaining peripherals are released and the compiler halts EC

3. No errors; consolidation required: The compiler calls LIBSH which causes the necessary library routines to be found from an MLT and written to the scratch tape. If, after this, blank cues remain the compiler halts after printing a list of routines still needed. If all cues are satisfied the scratch tape is terminated and rewound, and the final output tape is allocated and positioned for writing. The request slip, loader, and consolidated leader are then written to the final output tape. The routine ENØUT then copies the contents of the scratch tape across, all remaining peripherals are released and the compiler halts EC.

P. Girad

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED

Programming Languages Division

Fortran Note 24

28th March, 1966

Utility Programs and the Production of Fortran Compilers

This note gives some general information concerning the production of the compilers and describes briefly the various utility programs that are used in this connection.

The Production of Fortran Compilers

Compiler Segments

The Fortran compiler is based on the 'compiler kit' philosophy. That is, in addition to the segments forming the 'heart' of the compiler there are a number of alternative 'system' segments. Different compilers may thus be produced by choice of different 'system' segments. There are currently 24 different types of segments, the majority having a name of the form:-

F x 2 y

where x is a letter of the alphabet

y is A - general
P - paper tape
C - cards
M - magnetic tape
L - limited version (Basic Fortran)

Details of each segment are given in appendix 1. In addition, the letter T may be appended to a segment to denote that it is under test.

Compilation

The segments are held in PLAN souce form on magnetic tape, amendments and compilations being made by using the COSY system (#XPMA + #XPLJ) on a four tape cyclic system.

At present three source files exist: FORTSOURCE which contains those segments needed to generate XFAM, XFAP, and XFAC; ZFOMSOURCE which holds the segments for the Basic Fortran compiler XFOM; and TESTSOURCE which basically contains XFAM segments, and is used to develop language enhancements and other off-line tests which, if included in the FORTSOURCE file, would make this too unwioldy to handle.

Fortran Library Tape (FLT)

Fortran Library Tapes are similar to PRE-MLT and MLT library tapes but contain only ASA Fortran compilers, the ASA Fortran library and utility programs that are commonly used by the fortran team. The contents of the FLT is shown in appendix 11. To guard against tape failures four tapes are

used in a cyclic system, being updated by the library maintenance program #XPMU.

Whenever proved versions of compilers or subroutines are produced they are incorporated into the FLT. There is, however, one complication. Since both XFAP and XFAC contain a segment giving library leader information, these compilers would normally be one generation behind any modified library routines. This is overcome by doing a 'dummy' FLT update run using both XPMU and LIBN to produce only the library leader segment, which is then compiled into XFAP and XFAC. It is then possible to include up-to-date versions of XFAP, XFAC and the modified library routines in the next genuine FLT update. This method ensures that FLT's do not exist containing compilers and library routines which are incompatible.

Testing and Issue

There is no guarantee that amendments made which correct a particular fault in a compiler do not have repercussions and causes other faults to appear. For this reason a diagnostic test program, #DIAG, has been written. It consists of many modular phases which are entered in turn at run time, each phase testing certain aspects of the compiler (not input/output), and, if any discrepancies are found, diagnostic messages are printed out. There are two versions of this program, one for Basic Fortran (XFOM) the other for ASA Fortran (XFAM, XFAP, XFAC). Since the size of the latter version is in excess of 18K, only XFAM is used to compile it. (both input and output being magnetic tape). XFAP and XFAC, which differ only in the system aspect, are then tested in this respect.

When a compiler and/or library subroutine have been amended and tested they are ready for issue. Immediately after issue the mark number is updated.

Description of Utility Programs

PF01

This routine is used to correct a general purpose loader held in binary on paper tape into a psuedo subroutine % F01 which is held in ≤ 20 word blocks at the end of the magnetic tape library. The subroutine looks like an ordinary semi-compiled program but in fact contains blocks of binary programs. The Fortran 11 compiler # XFLM requires the loader to be held in this manner since there is insufficient space available to either hold the loader in store or read the loader in from magnetic tape where it is held as an ordinary binary program in a type 5 block (≤ 512 words)

GPLC and # GPLL

These routines # GPLC card orientated and # GPLL paper tape oriented, convert a general purpose loader held in binary to an octal representation which is inserted as data (lower preset) into the output segment of the Fortran compiler. This is necessary for any version of the compiler which holds the loader in core store. Note that the loader cannot be in source form as binary output is required.

COND

The purpose of this routine is to eliminate all unnecessary run out, E blocks and pause blocks present in a semi-compiled library tape. This routine is now obselescent since # FLPM will produce a condensed semi-compiled library direct from magnetic tape.

LIBN and # LIBC

These routines will generate a plan source segment consisting primarily of data giving the essential information required by the consolidator about the library routines (i.e. a table of compacted leader information). ASA Fortran compilers require the use of # LIBN which generates a plan source segment FV2A with an internal cue LIBRY. Parameters are read from either paper tape or cards, the library being on magnetic tape in block SRF4. The routine # LIBC is used to create a segment called LIBRY, the library being on paper tape. This routine is used with the Fortran 11 compiler.

#MTPR

This is a magnetic tape to printer general listing program which prints the first 20 words of any block read. The advantage over other similar programs is that it will print for each block read only the number of words given in the second character of the block (or 30 words if this is zero or <20). Also it will not halt until it finds two tape marks.

#RENB

The purpose of this program is to copy only selected segments of semi-compiled program from an input magnetic tape to an output tapo which is opened as a scratch tape but given a specified name and a retention period of 50 days.

This program is used in conjunction with #ZPOT to overcome two difficiencies, in #ZPOT, namely.

a) program may not be renamed
b) segments cannot be deleted.

Since ASA Fortran compilers are at present compiled using the COSY system this program is of use only with Fortran 11 compilers.

#SCPR

This routine may be used whenever semi-compiled program held on paper tape, cards, or magnetic tape is to be analysed and printed on the line-printer.

It is in fact identical to #YKBP (see Software Testing Manual for specification) but with the addition that input may be from cards or magnetic tape thus enabling the output from all three compilers, XFAM XFAP and XFAC, to be examined.

#CDMT

This routine will transcribe cards in Fortran source language to magnetic tape in subfile format (see PDCC/171 issue 2) suitable for input to #XFAM. Parameter cards mixed with source cards are required to create start of subfile or end of subfile sentinels.

#FLPM

Whenever it is desired to produce the ASA Fortran library on paper tape this routine may be used. It assumes that the library is held on magnetic tape in program library format under group name SRF4 and produces two reels of paper tape in a compact form.

#CSDR

The ASA Fortran compilers are compiled using the COSY system. This system has the advantage of selecting specified segments to be compiled but also has the disadvantage of not being able to duplicate segments without going via a hard copy on cards or paper tape. For other than trivial modifications development of the compilers is achieved by creating duplicates of segments into which modifications are put and tested. When proved, the duplicate segment becomes the up-to-date version and the original is deleted.

The utility program #CSDR is capable of both deleting and duplicating segments, the duplicated segments being renamed in specified. Except for reading parameters from a slow peripheral the process is carried out entirely on magnetic tape, in a manner very similar to #XPMA. In the normal course of development #CSDR is used twice, once to duplicate and rename, once to delete the old segment and name the duplicate as the new segment.

#FLID

Three compiling systems are possible depending on the way in which leader information is supplied to the consolidator for a paper tape (or card) compiler such as #XFAP (or #XFAC)

In order of preference they are:-

(i) Leader information is held in store, (segment FV2A in XFAP or XFAC), semi-compiled information being read from the library supplied at load time.

(ii) As (i) but leader information is read from the library which is also supplied at compile time.

(iii) As (ii) but semi-compiled information is read and punched out (copy switch on) at compile time. Library is not then required at load time.

The utility program #FLID is a means to improve system (ii) by producing a paper tape containing the leaders only, copied from the paper tape library.

#CSRΘ

This routine is designed to re-order segments on a COSY source file as specified by a parameter list which may be read from cards or paper tape. Such re-ordering is necessary in order to reduce the core store required by XFOM to 5632 words, the pre-release PLAN compiler ZPLJ is used in place of XPLJ during the COSY run. However at present ZPLJ requires segments to be supplied in overlay unit order within overlay area order with permanent last. Thus, as XPMA is not capable of re-ordering segments CSRΘ is used as a preliminary whenever amendments made to XFOM alter the sequence of overlays.

Summary of Utility Programs

Name	Description
##PFOI	Creates Fortran II loader for ##XFLM.
##GPLC)	Conversion of binary GPL's to plan format.
##GPLL)	
##COND	Copies semi-compiled program on paper tape removing superfluous run out etc.
##LIBN)	Creates 'LIBRY' segment from the semi-compiled library.
##LIBC)	
##MTPR	Magnetic tape to printer (general).
##RENB	Renames input file to ##ZPOT.
##CSDR	Duplicates, renames and deletes segments held on magnetic tape in COSY format.
##SCPR	Prints and analysis semi-compiled programs.
##CDMT	Transcribes cards to magnetic tape in subfile format.
##FLPM	Creates paper tape library from FLT.
##FLID	Creates paper tape with leaders only.
##CSRO	Re-orders segments on a COSY source file.

L.R. FAIRBROTHER.

Appendix I

Compiler Segments

Name	Description
FA2A	Supervisor
FB2M	Input (magnetic tape, paper tape, cards)
FB2A	Input (paper tape or cards only)
FC2M	Output (magnetic tape)
FC2P	Output (paper tape)
FC2C	Output (cards)
FD2M	Consolidator (magnetic tape)
FD2A	Consolidator (paper tape or cards)
FE2A	Segment compiler, statement compiler, END
FF2A	MASTER, SUBROUTINE, FUNCTION, EXTERNAL, BLOCK DATA, RETURN, PAUSE, STOP.
FG2A	EQUIVALENCE, DATA.
FH2A	Arithmetic statements, IF, GO TO, CALL, ASSIGN.
FI2A	DO
FJ2A	Input/output statements.
FK2A	Type statements, DIMENSION, COMMON
FL2A	Listing segment.
FM2A	Expression analysis (Part I).
FN2A	Expression analysis (Part II) and compilation.
FP2A	Input (peripheral independant).
FQ2A	Output (peripheral independant).
FR2A	Instruction generator.
FS2A	List System.
FT2A	TRACE.
FU2A	Error segment.
FV2A	Library leaders (LIBRY).
INTFC	Intrinsic and standard functions.
FZ2A	Program Description.

NOTE: In addition to these segments there is one segment per compiler which supplies the compiler name and mark number.
There is also a set of segments FA2L to FZ2L which comprise the Basic Fortran compiler, XFOM

APPENDIX 11

FORTRAN 4 LIBRARY TAPE (FLT)

Binary Programs

XFAM/1D
XFAP/1D
XFAC/1D
XFCM/1
LIBN/1
MTPR/1
SCPR/1
GPLC/1
CDMT/1
FLPM/1

Library Subroutines (under group name SPF4).

%FAP4/4
%FMN4D/4
%FMN4/5
ALLØT/4
DISENG/4
RLEASE/4
RUNØUT/4
%FINCUT/9
%FIOPT/5
%FIOCARD/5
%FIOLP/5
%FINIL/6
%FIOTA/5
%FIOMT/7
%FIOMTF/6
%FINMT/6
%FIODA/4
%FIODP/6
%FARHD/5
%FEx4/4
REAL/4
AIMAG/4
CMPLX/4
CØNJG/4
CLOG/4
CSIN/4
CCOS/4
CEXP/4
CSQRT/4
%FC2/4
%FCP/4
CABS/4
DMINI/4
DMAXI/4
DSIGN/5
SNGL/4
DBLE/4
DSQRT/5
DLØG10/4
%FDR/4
%FDD/4
DMØD/5
DCØS/5
DSIN/5
DLØG/5
DEXP/4
DATAN/5
DATAN2/5
DABS/5
%FDPØLY/4
%FD2/4
%FDP/5
SQRT/4
COS/4
SIN/4
ABS/4
LABS/4
COT/4
TAN/4
EXIT/4
DATE/4
MØD/5
SWØFF/4
SWØN/4
INT/4
NINT/4
FLØAT/4
ACØS/4
ASIN/4
ACØT/4
ATAN2/4
ATAN/4
SINH/4
CØSH/4
ALØG10/4
EXP10/4
TANH/4
COTH/4
AMØD/5
ACØSH/4
ASINK/4
ATANH/4
ACØTH/4
ALØG2/4
AINT/4
ANINT/4
PLØT/4
SIGN/5
ISIGN/5
XTY/4
EXP/4
ALØG/4
IDATE/4
MAX0/4
AMAX0/4
MAX1/4
AMAX1/4
MIN0/4
AMIN0/4
MINI/4
AMINI/4
IDINT/4
IFIX/4
DIM/4
IDIM/4
SLITE/4
SLITET/4
SSWTCH/4
DVCHK/4
ØVERFL/4
%FØVL/4
%RØL/4

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATERS LIMITED

Programming Languages Division

FORTRAN NOTE 25

8.12.66.

Array addressing and communication

This note replaces Fortran Note 9 which should be marked as obsolete or destroyed.

Array Addressing

1. Calculation of the address of an array element

Associated with each array is an "array header" which contains information about the structure of the array. The array header is not stored adjacent to the array itself; array headers are stored in upper preset area, whereas arrays are held in upper variable area (or upper preset if reference is made to them in a DATA statement).

For arrays with a single subscript the address of the specified array element is calculated by code generated by the compiler. For example, if A(I) is encountered and A is a real array then the code generated would be:

```
LDX  6  I
SLL  6  1
LDX  3  Address of array header
LDX  3  0(3)
ADX  3  6
```

The second instruction becomes 'SLL 6 2' for double precision or complex arrays and is omitted for integer or logical arrays compiled in 'Compress Integer and Logical' mode. When the subscript is an expression then 'LDX 6 I' is replaced by a series of instructions to calculate, in X6, the subscript value. For arrays with more than one subscript, the address of the array element is calculated by entering one of several subroutines contained within the Fortran arithmetic package, %FAP4. Three separate routines exist in order to minimise execution time and their entry points are;-

%FAP4 + 25 for arrays with two subscripts and 1 or 2 words per element (i.e. not complex or double precision elements).

%FAP4 + 20 for arrays with two or more subscripts and 4 words per element.

%FAP4 + 6 for arrays with two or more subscripts and 1 or 2 words per element.

On entry to any one of these subroutines

X3 = address of the array header

X1 = link

and the CALL must be followed by one instruction per subscript which, if OBEYed, will place the address of the subscript in X3.

On exit, the address of the desired array element is in bits 6 - 23 of X3 and the top 6 bits will be non zero. The calling routine will be re-entered at the location following the last subscript instruction.

If the object program segment is being compiled in 'TRACE' mode then all array element calculation will be carried out by a subroutine contained within the Fortran trace package %FMN4 with the entry points.

%FARRY + 0 for arrays with 1 or 2 word elements

%FARRY + 2 for arrays with 4 word elements

This subroutine requires the same parameters as the %FAP4 subroutines and gives the same result but in addition checks that the array element address is within the bounds of the array.

The Basic.Fortran compilers generate a call to either %FAP4 + 6 or %FAP4 + 20, depending on the type of element, for array elements with one or more subscripts (unless in TRACE mode when the call is to %FARRY or %FARRY+2).

2. The Array Header Format (See Footnote *)

The array header format is compatible with that used by Algol. It is made up as follows:

1st word Sign bit = 0 for 1 or 4 word elements
= 1 for 2 word elements
Next 5 bits = No.of dimensions in array
Next 15 bits = Base address (address of element 0,0,---0)

2nd word Sign bit = 1 for 4 word elements
Next 5 bits = Zero
Next 18 bits = Address of first element - always element 1,1,1,----in Fortran, but may be dirrerent for Algol.

3rd word Sign bit = 0
Next 5 bits = 0
Next 18 bits = Address of first word past end of array.

This is the array header for a one-dimensional array. For an array of more than one dimension succeeding words give the partial products of dimensions as follows:

4th word = D_1 ≥ 2 dimensions
5th word = D_1 D_2 ≥ 3 dimensions

---

* The array header format is likely to be changed in the near future; details of the proposed change are given in the appendix.

last word = $D_1 . D_2$ --- D_{n-1} n dimensions

where D_i is the maximum value of the ith subscript may take.

3. Communication of array information between segments.

The 'calling' segment may specify as an actual argument in a CALL statement either an array name or an array element in order to transmit an array or part of an array to the 'called' segment. In the case of an array name it is the address of the array header that is passed on whereas for an array element it is the address of the array element that is passed on. The two types are distinguishable by inspection of the top 6 bits which are zero in the case of an array header and non-zero for an array element.

It is also possible to transmit an array of Hollerith characters. The argument in the CALL statement takes the form of a TEXT constant (i.e. a string of hollerith characters preceded by nH) and the address of this string of characters is made to look like the address of an element by making the top 6 bits non zero.

The called segment will have as a corresponding argument in a FUNCTION or SUBROUTINE an actual array name. The ASA specification allows, and in fact insists on, the array being re-defined in a DIMENSION or 'type' statement. The subscripts may either be unsigned integers or integer variables in which case they must appear also as arguments of the subroutine or function. Thus the original array defined in the calling segment may be re-partitioned either statically or dynamically, the structure being defined by the values of the subscripts at the time that the subroutine is entered. An array must be given its maximum size in its original definition. Arrays may be repartioned in any manner but the type of actual anddummy array should correspond. It should be realised that an array, originally defined as A(10,10) when redefined as B(5,5) will not correspond to the top left hand corner of the original array but to the first 25 elements of the array taking in column order, i.e. B(5,5) $\equiv$ A(5,3).

Note that only arrays appearing as an argument in the FUNCTION or SUBROUTINE statement may have a variable structure.

4. Structuring of Dummy arrays.

The structuring of a dummy array is done by means of the FORTRAN subroutine %FARHD which on entry requires

(i) X3 to contain the address at which %FARHD is to generate a new array header.

(ii) X6 to co ntain the address of a word which contains the address of the original array header, or to contain the address of a word which contains the address of the first element of the array and the top 6 bits are non-zero.

(iii) X1 to contain the link.

%FARHD also requires a parameter list following the CALL instruction where;-

<u>1st word</u>: $b_0 - b_{11}$ = 0

$b_{12} - b_{17}$ = number of words per element (i.e.1,2 or 4)

$b_{18} - b_{23}$ = number of dimensions in new array structure

<u>following words</u>:- a series of instructions (one per dimension) which when OBEYed would place in X3 the address of a word containing that dimension.

On exit from %FARHD, a new array header will have been generated starting at the address originally specified in X3.

The word originally specified by the contents of X6 will now contain the address of the new array header.

L.R. Fairbrother.

Appendix to Fortran Note 25

1. Change of array header format

In the near future object programs will have to work in "Extended Data Mode" where the modifiable part of a word is the least significant 22 bits instead of 15 bits. When the array header format was originally designed allowance was made for extension only to 18 bits and therefore it will be necessary to change the array header format to work under "Extended Data Mode" conditions. Since there are a number of Fortran/PLAN users whose programs are dependant on the format of array headers it will be necessary to make the change in two stages:

(i) Provide subroutines which interrogate and generate array headers so that users may modify their programs to make use of these subroutines.

(ii) Change the array header format and at the same time modify these subroutines so that the user is unaffected.

2. Array header subroutines

Two subroutines are proposed for use in object programs at PLAN level:

a) GETAH to interrogate an array header, the calling sequence being;-

```
CALL    1  GETAH
LDX     3  'Address of array header'
LDX     3  'Address of information block'
```

where the information block consists of n + 4 words to receive details of the array header:

1st word	Number of dimensions (n)
2nd word	Number of words per element
3rd word	Base address
4th word	Address of first element
5th word	Address of first word past end of array
6th word	first partial product (D_1)
7th word	second partial product ($D_1 . D_2$)
(n+4)th word	last partial product ($D_1 . D_2 --- D_{n-1}$)

b) GENAH to generate an array header, the calling sequence being:-

```
CALL   1  GENAH
LDX    3  'Address of location at which array
           header is to be generated'
LDX    3  'Address of information block'
```

The information block must be set up as defined previously. On exit (return to link + 2) The array header will be generated at the address specified. If this address is the same as the address of the information block the effect will be to condense the information block into a standard array header. It should be realised that the number of words in an array header is likely to be increased when the array header is changed.

3. New array header format

Word	Bit	Content
0	B0	0 if 1 or 4 word element, 1 if 2 word element
	B1	1 (to distinguish array element from array header)
	B2-B23	Base address, i,e. address of element 0,0,0 ------
1	B0	0 if 1 or 2 word element, 1 if 4 word element
	B1	Undefined
	B2-B23	Address of first element - in Fortran always element 1,1,1,------
2	B0	1 (to distinguish from old type of header)
	B1	Undefined
	B2-B23	Limit address, i.e. address of first word past end of array.
3	B0	Undefined
	B1	Undefined
	B2-B23	Number of dimensions (maximum 15)
4,5 etc		Partial products as now

The above format should be taken as definitive and any previously proposed new array header format should be ignored.

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED.

Programming Languages Division.

FORTRAN NOTE 26

June, 1967.

The 16K EDS Compiler #XFAE

This Note is intended to describe the Systems segments of the E.D.S. version of the compiler, insofar as these differ from the magnetic tape version. However, sections which are common to both versions are also outlined where this seems necessary for an understanding of the whole.

The material is divided into two parts, which are complementary.

The first part describes the various routines, while the second part contains a numbered list of names of areas, markers, pointers and so forth, together with notes on their use in the compiler. Any mention of an item which is described in the second part of the Note is indicated by the appropriate number in square brackets, enabling the item to be looked up directly.

Descriptions of Routines

1. Initial entry to compiler

Entry is at word 20 for paper tape input, 21 for card input and 22 for ATLAS card input. Other entries are at word 27, which causes all peripherals to be released and the computer to halt PF, and word 28, which simulates the detection of a FINISH statement,

All entry points are in segment FA2D.

When entered at 20,21 or 22 the compiler sets words 30 to one of three values depending on the entry.

The routine DISM is next called. This does nothing in the normal case since NUMIN[1] = 0. If however the compiler has been re-entered after an abortive compilation, NUMIN will not be zero and a console DISPLAY message will be typed.

Having read in the appropriate overlay, the compiler then enters it at the cue FMAS1.

2. FMAS1

This routine performs that part of the initialisation of the compiler which is required once only, at the beginning.

The following markers are zeroised:

ΘVMMM[3], STINI [7], PDMK [8] SGMK [9], INPSW [10], and SCLM [12].

The following lower areas are zeroised:

LC1 common area; Part of LC3 common area; the first 511 words of LV.

The following initialisation is also performed:

ΘVNDX[2] is set to #40000000 (indicating no overlay list yet)
PRIΘ[4] is set to #500
CORE[6] is set to the number of words of store initially required by the compiler, minus 10.
MΘNI[5] is set to 1 indicating assumption of TRACE level 1.
CMΘD[11]: Bit 0 is set to 1.
Bits 1 - 23 are set to zero.

The routine PCK6 is called, to initialise the pointers CΘBP[13] and SCΘB [14]
The routine STINP is called to initialise TAPER [15]. TAPER is first zeroised (all 6 words). Bit 1 of word 30 is then tested to find out whether input is initially on cards or paper tape. The appropriate device is allotted by calling APER (described later).

The routine PLIE (outlined later) is called to read a line from the input device into the area CIMAGE[16].

The routine STSE (outlined later) is called to compare the contents of CIMAGE with a table of acceptable statements, setting X3 to a value depending on the results of the comparison. X3 is then used as a modifier to obey the appropriate instruction in a branch table. In this case, the only "acceptable" statements are LIST(TP),

SHORTLIST (TP), and NO LIST. The first two of these cause LISMK [17] to be set to 1 (tape punch); NO LIST causes it to be

set to 0; anything else causes it to be set to 2, (line printer assumed).

Note: The first line remains in CIMAGE and is examined again during the routine FMAS2.

The routine STLST is called. This calls APER to allot a listing device if required, and prepares to start listing.

The routine PLST3 is called, causing the first line to be actually listed (if there is a listing device).

IND1 [18] is set no-zero before entering the routine FMAS2.

3. FMAS2

This routine performs that part of the compiler initialisation which has to be done before every segment is dealt with.

SΘL [20] is set equal to the address in SΘLΘ [19] thereby causing the local lists for the segment to start at the beginning of the listspace.

The markers EOFM[21], SCLM[12], and EMARK[22] are zeroised.

If IND1[18] is zero, the routine PLIE (outlined later) is called to read a line from the input device into CIMAGE [16]. Otherwise a line is assumed to be already there.

The routine STSE (outlined later) is called to compare the contents of CIMAGE with a table of acceptable statements, setting X3 accordingly for use in obeying a branch table. Acceptable statements here are all intersegment statements, and any statement which introduces a Fortran segment or the Program Description. Anything else results in a branch to ER17, which flags an error 17.

FMAS2 is the usual return point after processing of intersegment statements or of whole segments.

4. SEGL1

The branch table used by FMAS2 results in a branch to SEGL1 for all statements which introduce Fortran Segments.

STIN1[7] and SGMK[9] are set non-zero.

The routine INE is now called. This completes the initialisation required for any Fortran segment, and includes the setting up of the 23 index words for the 23 threads of the main list. The moving list pointer NAVL[23] is set equal to SOL[20] initially. The local cue list index word is set up; the LC2 common area is zeroised; the routine PCH6 is called to initialise the output buffer pointers COBP[13] and SCOB[14]; and the first S/C record is begun by inserting the '3' needed for beginning a segment title block. A few other things, which have not been mentioned are also initialised.

SEGL1 finally branches to PRSL1 (see next section)

5. PRSL1

This routine constitutes the main path through the compiler for the processing of each Fortran statement.

When PRSL1 is entered, a line will already be in store waiting to be processed. However, the compiler has to find out whether there are any continuation lines, and to do this calls PSSE (outlined later). PSSE continues reading further lines until the first line which is not a continuation line is found, and builds up the statement in the area STATEMENT[24]

When the statement is complete the label field is dealt with. Discrimination as to type of statement is then carried out by calling STATE. This uses the routine STSE to compare the statement type with one of three tables of acceptable statements, setting X3 accordingly for use with a branch table. The table used for comparison is governed by the contents of SCLM[12]. If SCLM is zero only a statement which introduces a segment is acceptable; if it is 1, then any statement permissible is an ordinary Fortran segment is acceptable; if it is 2, only those statements permitted within a BLOCK DATA segment are acceptable. Intersegment statements are of course not allowed at all here.

Arithmetic statements are identified separately, and for these the central compiler is entered immediately.

If SCLM=1 (i.e. within an ordinary segment) or SCLM=2 (i.e. within a BLOCK DATA segment) the compiler branches straight to the appropriate part of the central compiler.

If SCLM is zero the only acceptable statements are MASTER, SUBROUTINE, BLOCK DATA, and the various types of FUNCTION statement. These are dealt with respectively by the routines MASE, SPSE, BLOCK, and (at least eventually) FPSE. In all these cases the first thing done is to set SCLM non-zero; it is set to 1 unless the statement is BLOCK DATA in which case it is set to 2. Having done that, processing of the statement itself begins,and that can reasonably be considered as part of the central compiler.

When a statement has been completely processed by the central compiler, control returns to CC which forms part of STATE.

The compiler then exits from STATE and continues through PRSL1. If the next statement (the first line of which will be already in store) is an END statement, it calls the routine ENDE (outlined later) to process it, and then branches back to FMAS2, since it is no longer within a Fortran segment. If the next line is not an END statement, control returns to the beginning of PRSL1 and the whole process is repeated for the new statement.

6. RPER

This routine deals with the release of peripherals or closing of files.

Calling sequence:

```
LDN  1  n
CALL 5  RPER
```

where n is the number of the relevant word in TAPER [15], and is therefore in the range 0 - 5.

Action: If the whole of the relevant TAPER word is zero, no device of that class has been allotted. In that case RPER merely exits. If the relevant TAPER word is non-zero, it is zeroised and the peripheral is released or the file closed as the case may be.

On exit, X1 and X7 will be the same as at the beginning. X7 sometimes contains useful information throughout.

Note: All release of peripherals or closing of files is done by this routine.

7. APER

This routine deals with the allocation of peripherals or opening of files, and all such operations are done by this routine.

Calling sequence:

```
LDX  7  'C/M'   (or equivalent)
LDN  1  n
CALL 5  APER
```

where n is the number of the relevant word in TAPER [15], and is therefore in the range 0 - 5. C/M is a counter modifier for the device or file required, in the form type/unit number, (B0 must be zero). This counter modifier is transferred by APER into bits 1-23 of the TAPER word. If the device is successfully allocated, or the file opened, bit 0 of the TAPER word is set to 1 before the exit.

In the case of a slow device the routine attempts to allot it by means of an ALLOT instruction; if successful it merely sets Bit 0 of the TAPER word to 1 and exits. If unsuccessful it halts TR, TP, LP, CR or CP as the case may be, and a further attempt can be made by typing GO.

In the case of magnetic tape or EDS the area MFIL[25] must have been previously set up as follows:

MFIL to MFIL+2: Name of file to be opened.
MFIL+3: File generation number for insertion in control area. If the device is magnetic tape this information is transferred to the control area and the file is then opened.

If the device is EDS, words 1 to 8 of the control area are first zeroised to get rid of any reply information from previous operations. The file details are then transferred from MFIL to the control area and the file is opened. Finally the routine TRW1 (see next section) is called to test the reply word. If the reply is not satisfactory a halt occurs depending on the exact reason for this. A further attempt to open the file can be made by typing GQ.

On exit from APER, X1 and X7 are the same as on entry. Also X2 contains the device type number and X3 the unit number.

Note: Whereas RPER can safely be entered even if there is nothing to release, APER must not be entered if the device concerned is already allocated.

8. TRW1

This routine tests the reply word after an attempt to open an EDS file. It uses X6 as the link and expects to find the reply in X2. If the reply is satisfactory it returns to link+1. Otherwise it returns to link+0 if GQ is typed after the halt.

Note: APER relies on TRW1 not to use any accumulators except X2 and X6.

9. PLIE

PSSE

PLNE

These are the three entries to the peripheral-independent part of the input system.

PLIE is entered when the compiler is between segments (or before the first segment) and wishes to input the next line. In that case PLIE will be called from FMAS2 and one of the following types of line would be expected:

(1) An intersegment statement
(2) A statement introducing a Fortran segment
(3) A type 3 semi-compiled record (or type +)

PLIE first sets SCIND[27] equal to zero then calls INPUT (described later) to read the next line into CIMAGE[16]. If the line is in fact semi-compiled (type 3 or +), this fact is detected before returning to PLIE and SCIND is set non-zero. In that case, since SCLM[12] is necessarily zero in these circumstances, PLIE immediately branches to the routine SEMI (described later). SEMI outputs the semi-compiled record, and calls PLNE. Since PLNE does the same as PLIE except that it does not zeroise SCIND, the next line is then input and the compiler branches back to SEMI. Semi-compiled is thus copied across to output until SEMI detects the end of the segment and branches to FMAS2 instead.

If the line read in is an intersegment statement, SCIND [27] is left zero by the routine INPUT, and the path followed through PLIE is governed by the fact that SCLM [12] and SCIND will be zero and V will be unset. This path eventually goes through PLSCX, where the end of line marker # 74 is inserted in CIMAGE [16], to PLSL8 and exit. On exit from PLIE the line is compared with a table of acceptable statements by FMAS2, and the compiler then branches to process it. Exactly the same procedure is followed for lines which introduce source segments, since the statement is not identified until FMAS2 compares it with the table.

PSSE is entered from the routine PRSL1 (already described). This only occurs when the compiler knows it is dealing with a Fortran source segment. The first time it is called in a given segment, the line introducing the segment will be in store. However, the compiler must find out whether there are continuation lines always, and to do this calls PSSE. PSSE first sets V and zeroises SCIND[27]. SCLM[12] will be zero if this is the first time in the segment that PSSE has been called; otherwise it will be non-zero. Bearing these various points in mind, the path followed can be traced. Further lines are input (by PLS5, not by PLSCC which is by-passed when entry is at PSSE) until the first line which is not a continuation line is found. The complete statement is built up in the area STATEMENT[24]. The statement is then subjected to a "preliminary line scan" (PLSC7) which finds some of the most blatant types of error, and the compiler finally exits via PLSL8.

Note: Except when semi-compiled records are being dealt with, a pointer called POSI [28] is set up before the exit. If entry was at PLIE, it is set to point to the seventh position in CIMAGE [16] which is where the statement itself is expected to start. If the entry was at PSSE, POSI is set to point to the beginning of the area STATEMENT[24], which is the place where further processing will begin in that case.

10. INPUT

This routine forms the peripheral-dependent part of the input system. It is usually called in the course of PLIE, PSSE or PLNE (already described) but is also called or branched to from other places.

It first deals with the special case of Read From (ED,...) continuation lines by branching to TRSP8 if INPSW [10] is non-zero. Otherwise it space-fills the area CIMAGE[16], and then goes on to determine, by examining the contents of TAPER [15], which is the current input device. A slow peripheral will be present in all circumstances, but if TAPER indicates that EDS or magnetic tape is in current use also, the EDS or magnetic tape is selected rather than the slow device.
The compiler branches to one of four sections accordingly:

TRSPL:	Paper tape input
CRSPL:	Card input
EDSPL:	EDS input
MTSPL:	Magnetic tape input

TRSPL: The paper tape record is read initially into the area BUFF [29]. If SCLM [12] is zero, the routine SETSC is called, which sets SCIND [27] or leaves it unaltered as the case may be. The record is then moved to CIMAGE [16] after interpretation and removal of all shift characters. Finally NLIND [30] is set.

CRSPL: Cards are read into CIMAGE [16] directly. If SCLM[12] is zero, the routine SETSC is called, which sets SCIND [27] or leaves it unaltered as the case may be. If the card is in ATLAS code (ascertained from word 30 of store) it is translated into standard code in the CIMAGE area. The rest of the routine, from CRSP1 onwards is devoted to working out what should be stored in NLIND[30]. This last section is used at the end of MTSPL and EDSPL also.

EDSPL: The areas BEMPT[31], BKNUM [32], and BUCIN+4 [33] will have been initialized during the processing of the READ FROM (ED,...) statement. All EDSPL has to do is to continue inputting records from EDS until the requisite number of buckets, as indicated by BKNUM, have been dealt with. It does not therefore have to look out for an end sentinel of any kind. Extraction of records from successive buckets is accomplished with the aid of SCN1 [34] and SCN7[35]. The actual reading of buckets is done by the routine RBUCN. This also tests the reply word by calling TRW4 (described later) and sets SCN1 [34].

When BKNUM has been reduced to zero, the compiler exits via EDSP3 (see next section).

MTSPL: This deals with input of source and semi-compiled program (whether batched or unbatched) from magnetic tape. It is much larger than EDSPL because the formats are less simple and because the ends of subfiles have to be located by testing of sentinels rather than by a simple counter.

Blocks are read initially into the area BUFF [29] and records are then extracted one at a time into the area CIMAGE[16].

When the time comes to return to the slow input device the compiler branches to BRIN2 after releasing the tape or setting bit 0 of TAPER+1 [15] to zero. BRIN2 makes sure that the correct overlay is in store, and then branches to INPUT, thus effectively continuing the attempt to obtain the next line by reading from the slow device.

11. EDSP3

BRINP

When any READ FROM is first detected, INLINK [37] is zeroised. If the READ FROM applies to EDS, the Read From routine finds the position on EDS where the data starts, and knows from the directory how many data buckets there are in the subfile.

The main compiler is then entered by calling BRINP, whose first action is to store X0 in FMLINK [36] thereby preserving the link back to the Read From routine. It then tests INLINK [37] to see if it is zero. The first time it will be, and control passes to FMAS2. Thus the compiler begins reading EDS, expecting to find an intersegment statement or a statement introducing a segment.

Compilation continues from EDS until the requisite number of buckets have been read, in which case control leaves EDSPL via EDSP3. Now at this point X0 will contain the link back to PLIE or PSSE, which will have called INPUT for the purpose of reading the nect line. This link is stored in INLINK [37], and, after reading in the correct overlay, the compiler goes back to the Read From routine via the link preserved in FMLINK. The Read From routine ascertains whether any more subfiles are to be read from EDS. If so, it sets up BEMPT [31], BKNUM [32], and BUCIN+4 [33] ready to read the next subfile, and then calls BRINP again. After preserving the link in FMLINK once more, INLINK is tested and this time is non-zero. The compiler ensures that the correct overlay is in store and then branches to INPUT, thereby effectively continuing the search for the next line from where it left off, and ready to exit back to PLIE or PSSE having found it.

When there is no more EDS input to come, the Read From routine closes the EDS file, causing TAPER+5 [15] to be altered accordingly, and branches to BRINP. BRINP restores the link from INLINK and branches to INPUT after ensuring that the correct overlay is in store. The effect of this is to continue searching for the next line from a slow device, returning, via the link, to PLIE wher it is found.

12. TRW2

TRW3

TRW4

These routines test the reply word after an "extend" operation or a "read/write" operation on EDS.

TRW2 is used after an attempt to extend a file. It expects to find the reply in X7, and an irrecoverable halt occurs if the reply is abnormal. The routine is relied on not to use any accumulator except X7 and the link X6. (It uses X2 if the reply is abnormal but there is then an irrecoverable halt anyway).

TRW3 is used to test replies after a write instruction on unit 1. The reply is expected to be in BUCK+1. If entry is at TRW4, the reply must be already in X7: This entry is used for input files. The routine is relied on not to use any accumulators except X7 and the link X6, unless the reply is abnormal in which case an irrecoverable halt occurs anyway.

13. SEMI

This routine copies semi-compiled segments which are input to the compiler onto the output file. Its basic method of use has already been described in connection with the routine PLNE. It lists the name of the semi-compiled segment, moves it from CIMAGE [16] to the area pointed to by SCØB [14], check-sums it, and causes it to be output by calling INTR1 (described later). If the record it has just output is a terminating record (type 4 or comma) it branches to FMAS2. Otherwise it calls PLNE to obtain the next record, which it deals with likewise.

14. GFE1

GFN

Both of these routines are used for picking up the next field in an input record. GFE1 is used normally, but not when the next field is expected to be a file or subfile name. In that case GFN is used.

GFE1 is described in detail in Fortran Note 12.

GFN, like GFE1, expects POSI [28] to be pointing to the first character in the field. Leading spaces, if any, are ignored. Thereafter, any character other than a number, a letter, a space, %, or - is regarded as a field terminator. On exit, both X5 and TERM [38] contain this terminating character. If the terminator is the first character encountered (other than spaces possibly), then X4 is negative on exit, indicating that there is no field. Otherwise X4 is zero on exit. Also on exit, X2 is pointing to the first character beyond the terminator, and the area NAME [39] contains the field which has been picked up. GFN treats spaces (other than leading spaces) as significant, unlike GFE1.

15. INTR1

INTR2

This routine is called when the compiler has set up a complete semi-compiled record in the output buffer pointed to by SCOB [14] and wishes to output it. One of the two calls to this routine is made from the peripheral-independent output segment FQ2A; the other is made from SEMI (described earlier). The routine tests CMØD [11], and if this is non-zero, indicating either that there are errors or there is no output device anyway, avoids outputting anything and merely resets the pointers SCØB [14] and COBP [13]. If CMØD is zero the routine calls PCH1 (described later) which deals with the outputting to EDS of the S/C record, and also resets the pointers.

16. PCH1/PCH6

This routine is called by INTR2 in order to output semi-compiled information to EDS.

The output system works roughly as follows:

Semi-compiled is output to a permanent file in the form of a single data subfile. If a program is successfully compiled, bucket 1 of the output file should contain a header, a directory description (DD) record, and two subfile description records (SFD's) one of which describes the subfile containing semi-compiled, while the other describes any empty space remaining at the end of the file. This directory bucket is not output until the whole program has been compiled. Although in the present version of the compiler it is not possible for the directory to overflow the capacity of a bucket, it was originally thought that batches of programs would have to be dealt with, and so provision was made for overflow of the directory. This makes the output segment unnecessarily complicated.

As soon as the building up of a bucket in the output buffer begins, a bucket is reserved for it on EDS by storing the number of the first unreserved bucket in either BKN1[43] or BKN2[42]. CNT[44] always contains the number of the first unreserved bucket in the file.

Output of semi-compiled information is double-buffered, the buffers being BUFR2 and BUFR3[46]. These are pointed to alternately by NCΘB[45].

When PCH1 is called, DIR[40] is set to zero, and a word count is appended to the front of the semi-compiled record. If the record is a type 3 semi-compiled record, then (unless this is the very first time, in which case the current buffer will be empty) WRT2 is called, which has the effect of writing away the current buffer and starting a new one. Otherwise WRT1 is called, which has the effect of adding the semi-compiled record to the current buffer if there is room, or writing it away and starting another one if there is not. WRT1 and WRT2 are described below.

Finally PCH6 resets the pointers SCΘB[14] and CΘBP[13] ready for the next semi-compiled record.

17 WRT1/WRT2

These routines expect the following information to be set up beforehand:

DIR = 0 for semi-compiled output; DIR non-zero for directory output.
X1 points to the area from which the record is to be taken.
X4 points to a location containing the number of the bucket on EDS reserved for the current output buffer.
X3 points to the current output buffer (i.e. BUFR2 or BUFR3)

The accumulators X1, X4 and X3 are not altered in the course of the routine; this is important because the routine calls itself at one point and branches to itself at another.

If there is room for the new record in the current buffer, and if entry was at WRT1, then WRT2 is not entered. The routine merely moves the record into the buffer setting X6 to point to its position there (X6 is used after the exit).

If there is not enough room in the buffer WRT2 is entered. WRT6 is then called in order to output the buffer to EDS. A new bucket is reserved, buffers are switched and the routine HEDR (see below) is called, to set up a header in the new buffer. The routine then branches back to WRT1, causing the record to be output into the new buffer.

18. HEDR

This routine sets up a header in the buffer whose start address is held in X3. The routine is relied on not to alter X3 or any other accumulator except X7. The link is X0.

19. WRT6

This routine actually writes buckets to EDS. If the file is already full, it will be extended by 80 buckets and LASTB[48] will be altered accordingly. After any file extension, the reply word is immediately tested by calling TRW2 (already described). Before writing a bucket the reply word for the previous write operation is tested. This method of working arises from the fact that semi-compiled output is double-buffered.

WRT6 is relied on not to alter any accumulators except those which it alters in the present version.

20. FERRR

This routine is entered from the error segment when the first error in a program has been flagged. It merely releases the output file by calling RPER (already described). The compiler will be prevented from trying to write further buckets by the fact that CMΘD [11] will be non-zero from that time on.

21. Program Description

The Program Description segment is entered from FMAS2 on detection of a PROGRAM (, OVERLAY PROGRAM(, or SEGMENTS statement. The only difference between these at present is that the first two must be followed by the program name whereas SEGMENTS on its own is permissible in which case the compiler assumes a name of "NONM".

If the Program Description is not the first segment, an error 17 results because SGMK[9] will be non-zero. Otherwise the overlay list pointer OVLLL[49] is initialised by making it point to the beginning of the list-space (whose address is held in SΘLΘ[19]).

Since the overlay list is held in the list-space area and grows when OVERLAY statements occur in the Program Description, the local lists for the Program Description segment must be accommodated elsewhere. They are in fact accommodated in the Polish List area, which starts at PNPI in common area LC3. SΘL[20] is therefore initialised to that address in this case.

Various other indicators are set non-zero including STIN1[7], PDMK[8], SGMK[9], and EMARK[22]. The usual segment-initialisation routine INE is also called, and the program name is stored for safe keeping. The PROGRAM, OVERLAY PROGRAM or SEGMENTS line is then processed (the cue list is begun etc).

PDCCE is the return point for all Program Description statements except the END statement. In PDCCE, the next line is read by calling PLNE (already described), and the compiler then compares the line with a table of acceptable statements and branches accordingly.

If in the course of the Program Description OVERLAY statements are detected, OVMMM[3] is set non-zero and entries are made in the Overlay list.

22. ENPD

When the END statement is finally detected in the Program Description the compiler branches to ENPD. If OVMMM[3] is non-zero, an entry for the overlay package %EROL is made in the cue list. An entry is also made for either %FERROR or %FERLIM depending on the contents of MONI[5]. CMOD[11] is then tested to find out whether any errors have been flagged. If not the routine ENDE (outlined later) is called. This outputs the leader for the Program Description segment. Having done that the Overlay list is no longer required and can be got rid of by setting ΘVNDX[2] to #40000000 again. Finally EMARK[22] is set zero, indicating the end of Program Description processing, and control returns to FMAS2.

23. ENDE.

This routine processes END statements and outputs segment leaders. The processing is somewhat different for END statements in Program Description segments to that for other END statements. EMARK [22] is non-zero if the Program Description is being dealt with, and zero otherwise. If errors have been flagged previously, as indicated by CMØD[11], most of the processing is omitted and control returns to FMAS2 almost immediately.

24. STSE

This routine is used in connection with the various comparison and branch tables used for discriminating between the various types of statement. It is described more fully in Fortran Note 12.

25. Fortran List System

This is described in detail in Fortran Notes 11 and 12. It must be remembered however that in the EDS Compiler there is no Consolidated Cue List, and that the Overlay list can be dispensed with as soon as the Program Description leader has been output. This means that all local lists (except that for the Program Description, which is held in the Polish List area) can start at the beginning of the list-space.

26. READ FROM

The Read From as applied to cards, paper tape and magnetic tape is essentially the same as in the magnetic tape compiler, except that a statement of the form READ FROM (MT,-. SRF5) is not allowed and results in the flagging of an error.

Any READ FROM detected while reading magnetic tape or EDS is ignored.

Any READ FROM causes INLINK [37] to be zeroised although this is irrelevant unless it is a READ FROM (ED,....).

All READ FROM's except the READ FROM (ED,...) eventually call RDFR to carry out the actual switching of peripherals. The calling sequence for RDFR is:

```
 LDX  7  'C/M' (or equivalent)
 LDN  1  n
CALL  0  RDFR
```

where n is the number of the relevant word in TAPER [15] and C/M is a counter modifier in the form Type/Unit no. This calling sequence is basically the same as that for APER (already described) and is in fact used by RDFR for calling APER.

READ FROM (CR/S1900) :
READ FROM (CR/ATLAS) : Bit 7 of word 30 is unset or set respectively. Otherwise these are treated in the same

way as READ FROM (CR), which merely sets up X1 and X7 and calls RDFR.

READ FROM (TR) is treated in a similar way.

In the case of a READ FROM (MT,...) the area MFIL[25] has to be set up. Details are given in connection with MFIL. Having done that, X1 and X7 are set up and RDFR is called.

Action of RDFR: If the device requested is a slow device, RDFR calls RPER to release the current device and APER to allot the new one. It then exits. If the device is magnetic tape and the first word of MFIL[25] is zero, RDFR expects a tape to be already allotted. Otherwise it opens one, taking the file name from MFIL. In either case it now tests word 1 of TAPER[15] and flags an error if this is zero. It then sets Bit 0 of TAPER+1 equal to 1. If MFIL+4 is zero, indicating a simple file, SIC[50] and SIMF[51] are set non-zero. SCT[52] is also set to zero before the exit. If MFIL+4 is non-zero, SIMF [51] is set to zero and the compiler scans the magnetic tape looking for the requested subfile, and positions the tape ready to begin reading.

On exit from RDFR control returns to FMAS2.

READ FROM (ED,....) : On EDS, unlike on magnetic tape, subfile names cannot be relied on to be unique, except within a particular directory subfile. It is therefore necessary to be able to follow a definite path down to a particular subfile, by specifying one subfile name for each level in the structure down to the level of the subfile required. The subfile is then located by following the appropriate pointer at each level.

A necessary consequence of this is that the READ FROM must contain a file name and an indefinite number of subfile names, the first of which applies to level 0, the second to level 1 and so forth. For this reason continuation lines are permitted as a special case, but subject to certain rules. These are that the first line must contain at least one subfile name, that lines may only be broken before a full-stop, and that no interspersed blank or comment lines are allowed. The format is therefore, for example,

```
    READ FROM (ED,Filename(G),S/F1(G).S/F2(G)
   1.S/F3(G).S/F4(G))
```

The generation numbers are all optional. If the generation number for the file is not specified, Executive assumes that the file with the highest generation number is required. If the other generation numbers are omitted, it is assumed that only one subfile of that name is present.

As the READ FROM is of indefinite length, it is not possible to read it all in before beginning to act on it. Thus as soon as the file name is known, an attempt is made to open the file, and as each subfile name is detected, the directory is scanned to find it. If an error is detected in the Read From after the file has been opened, it is released again at once so that input can continue from the slow peripheral.

Scanning of directories is carried out by the routine called SCAN. This expects to find the first bucket of the directory subfile in the area BUFF[29], and also expects the number of useful words in this bucket to be stored in SCN1 [34]. If the subfile name is found, the bucket containing it is left in BUFF, while SCN3 [53] points to the beginning of the required SFD.

When continuation lines are present, the difficulty arises that the slow peripheral must be re-activated to read in successive lines and also that the information currently in BUFF will be overwritten if the slow device happens to be a tape reader.

To avoid these difficulties the contents of BUFF are stored temporarily in the list space area (which contains nothing useful between segments), while the slow device is re-activated by removing bit 0 of TAPER+5[15]. The routine INPUT is then called to read the next line in (without listing it yet), and the end of line mark is afterwards inserted (INPUT itself does not do this; it is usually done in PLIE). The contents of BUFF and bit 0 of TAPER+5 are then restored.

When the SFD for the last-mentioned subfile has been located, the compiler branches to RDF13, where it prepares to find and read in all data subfiles implicit in the SFD. That is to say, the last-mentioned subfile may be composite, in which case it is assumed that all data subfiles in the structure based on it are required. To this end the compiler works systematically through the structure. Whenever a data subfile is located, the number of its first bucket is placed in BUCIN+4 [33], the number of buckets in it is placed in BKNUM [32] and BEMPT[31] is set to zero. The compiler then calls BRINP (already described) and returns only when the whole subfile has been input.

When everything implicit in the READ FROM has been input, the file is released and the compiler branches back to BRINP.

Further note on continuation lines: A continuation line is expected if the previous line appears to the compiler to be incomplete. If an expected continuation line is not forthcoming, the <u>previous</u> line is flagged as an error, and the new line is then treated as a new statement. For this reason, i.e: in case an intervening error flag has to be output, the continuation line is not listed as it is read in. INPSW[10] is used in this connection.

27. <u>SEND TO</u> (SNTBS)

This section is rudimentary at present because the compiler accepts only one form of SEND TO, and assumes that no output is required if no SEND TO is given.

The acceptable form is

SEND TO (ED,File Name(G).Subfile name(G)) where the generation numbers are optional. The compiler always sets up a new composite file in the file specified; consequently anything initially in the file is destroyed.

Bucket number 1 (the directory) is not output until the whole program has been successfully compiled. This is done by the routine CLEO1 (see below).

If this is not the first SEND TO detected, then STIN1[7] will be non-zero. In that case the new SEND TO is ignored and control returns to FMAS2. If STIN1 is zero, it is set non-zero at this point.

The file name and generation number are stored in MFIL to MFIL+3[25] and the subfile name and generation number in MFIL2[26]. If no errors are detected, bit 0 in CMOD[11] is set to zero and the file is then opened by calling APER (already described). The number of buckets in the file is then determined and stored in LASTB [48].

Bucket 1 is reserved for the directory by putting 1 in BKN1[43], and Bucket 2 is reserved for the first bucket of S/C by putting 2 in BKN2[42]. BCNT[54] is also set to 2. CNT[44] is set to 3.

A header is then set up in the first output buffer (pointed to by NCOB[45]), and control returns to FMAS2.

28. FINISH

An Error 36 is signalled if PDMK[8] is zero, indicating that no Program Description was supplied. CMOD[11] is then tested to find out if there is an output device; this will be the case only if CMOD is zero. If there is an output device, the routine CLEO1 (see below) is called. This outputs the directory. The compiler then halts ZZ or EC depending on whether there are errors or not.

29. CLEO1

The current output buffer is first written away by setting up the control area and calling WRT6. The routine then prepares to output the directory. The link is stored in DIR[40] thereby both preserving the link and setting DIR non-zero as in necessary for outputting directory information. X1 and X3 are then set up in the manner required by the routines HEDR and WRT1 (already described). HEDR is called, causing a header to be set up in the buffer. WRT1 is called causing a DD record to be moved into the buffer, after which the word "COMPOFILE" is placed in its name field. X3 will still contain the address of the buffer at this point. X4 and X1 are set up also as required by WRT1, which is then called to output an SFD for the semi-compiled information. The subfile name and other details are then inserted in this SFD. Finally another SFD is output, covering any unused buckets left over at the end of the file. The buffer itself is finally written away by calling WRT6.

List of Names of Areas, Markers, Pointers, etc.

1. NUMIN Contains the number of instructions so far compiled. This is output as part of a DISPLAY message on the console at the end of a program by the routine DISM.

2. OVNDX Overlay list index word. This is a counter-modifier pointing to the last item in the overlay list. If the overlay list is empty, OVNDX is set to #40000000

3. OVMMM If non-zero, indicates that at least one overlay statement has been found in the Program Description.

4. PRIO Indicates the priority of the object program. Is set to #500 (indicating priority of 50) if no PRIORITY statement is given.

5. MONI Indicates the TRACE level. Is set to 1 by default.

6. CORE Contains the number of words of core store occupied by the compiler, minus 10. If the list space becomes exhausted, the compiler attempts to obtain more core and adjusts CORE accordingly.

7. STIN1 If non-zero causes any SEND TO statement which occurs to be ignored. It is set non-zero when a SEND TO has been detected already, or when the start of any segment has been detected.

8. PDMK If non-zero, indicates that the Program Description segment has been previously detected.

9. SGMK If non-zero, indicates that a segment of some kind (Program Description or Fortran) has been detected previously.

10. INPSW If non-zero, indicates that a READ FROM (ED,...) continuation line is being processed.

11. CMOD Bit 0 : Set to 1 if no output device (EDS) has yet been allotted.

Bits 1-23: The number of errors so far flagged by the compiler.

12. SCLM SCLM=0. Compiler is not currently dealing with material inside a Fortran source segment.

SCLM=1 Compiler is currently processing a Fortran source segment, other than a BLOCK DATA segment.

SCLM=2 Compiler is dealing with a BLOCK DATA segment.

13. COBP "Current output buffer pointer". A counter-modifier pointing to the area into which the compiler places semi-compiled records to be output. It is a moving pointer, used to insert successive fields; it starts at 67/BUFF1+1.

14. SCOB "Start of current output buffer". Points to the same area as COBP above, but is fixed and always points to the start of the semi-compiled record, i.e. to BUFF1+1.

15. TAPER A six-word area in the common block CSLCØM, which controls the use of peripherals at compile time.

Word 0 : Unused at present. It was intended to use it for an EDS scratch file as part of an improved method of peripheral switching (see separate write-ups). There is no reason why this should not be implemented eventually.

Word 1 : Magnetic tape input
Word 2 : Slow input (card or paper tape)
Word 3 : Listing device (line printer or tape punch)
Word 4 : EDS S/C output (permanent file)
Word 5 : EDS input (permanent file)

Each word of TAPER contains information as described below.

1) Whole word zero : No device of this type is currently allotted.
2) Word non-zero:

Bit 0:

Words except word 1 : Bit 0 is equal to 1 if the device is currently allotted to the program.
Word 1 : Bit 0 is equal to 1 if the device is currently allotted and is being actively used. If bit 0 is zero, the tape is merely being retained in case it is required again later.

Bits 1-8: Device type (e.g.: 5 for magnetic tape.)
Bits 9 -23:Unit number.

Device types must be as follows: Word 1 may only be 5. Words 4 and 5 may only be 6. Word 2 may be 0 or 3. Word 3 may be 1 or 2.
Unit numbers are as follows: Magnetic tape input is unit 2, slow input is unit 0, listing device is unit 1, EDS output is unit 1, and EDS input is unit 2.

All allocation and release of peripherals is carried out by the routines APER and RPER . Nevertheless, words in TAPER are sometimes altered temporarily in other places for special purposes, as far instance when dealing with READ FROM (ED,.....) continuation lines.

16. CIMAGE An 80 word area in common block LCO into which individual lines of input are placed in the form of a "card image" (i.e. with shift characters removed etc).

17. LISMK Listing device indicator. LISMK=0 means no listing device; LISMK=1 means tape punch; LISMK=2 means line printer.

18. IND1 Set non-zero by routine FMAS1, before entering FMAS2. FMAS2 tests it and then zeroises it. If it was non-zero, FMAS2 avoids inputting a new line because the one input by FMAS1 has not yet been fully processed. Normally, of course, FMAS2 is not entered from FMAS1 but from somewhere else.

19. SØLØ A constant in LP containing the start address of the compiler's list space. The list space is in Upper Common Variable block UC1.

20. SØL "Start of List". No consolidated cue list is built up by the EDS compiler; consequently the local lists for each segment can always start at the beginning of the list space, whose address is held in SØLØ [19]. SØL is therefore initialised to this address by the routine FMAS2.

21. EOFM. "End of Fortran Marker." Believed to be redundant, but its complete removal would involve a change to the error segment of the compiler, which tests it.

22. EMARK. If non-zero, indicates that the Program Description segment (or its leader) is currently being dealt with.

23. NAVL. Local lists pointer (not fixed) which points to the location immediately after the last item in the lists.

24. STATEMENT. An area 331 words long in which Fortran statements (including continuation lines) are built up before being processed by the central compiler. Intersegment statements are not processed in STATEMENT however.

25. MFIL. An 8-word area in the common block CSLCOM, used for several purposes connected with the storage of file and subfile names taken from READ FROM and SEND TO statements:-

(1) READ FROM magnetic tape:
MFIL to MFIL+2 : File Name (MFIL set zero if none specified)
MFIL+3 : File generation number (zero if unspecified)
MFIL+4 to MFIL+6 : Subfile name (MFIL+4 set zero if no name specified, indicating simple file).
MFIL+7 : Unused at present but will be needed if subfiles are given generation numbers on mag. tape in future.

(2) READ FROM EDS:
The READ FROM (ED,...), unlike the READ FROM (MT,...), is acted upon as each field is read in. This is necessary because the EDS 'READ FROM' is of indefinite length. However this enables the first four words of MFIL to be used first for the File name and generation number, and then for all subfile names and generation numbers in succession, since each piece of information has been acted upon before it is over-written by the next field. MFIL+3 receives the file or subfile generation number if present; if none is specified it is set negative. MFIL+4 to MFIL+7 are not used in connection with EDS.

(3) SEND TO statements:
MFIL to MFIL+2 : EDS File Name
MFIL+3 : Generation number (set negative if not specified).
MFIL+4 to MFIL+7 : Not used. The subfile name and generation number from a SEND TO statement are stored elsewhere in the area MFIL2 [26]. It has to be preserved until the end of the compilation to be output in the directory; if it had been stored in MFIL+4 to MFIL+7 it would be in danger of being destroyed by the processing of a READ FROM (MT,...) statement.

26. MFIL2.

A 4-word area used for storing the subfile name and generation number from a SEND TO statement.
MFIL2 to MFIL2+2 : Subfile name
MFIL2+3 : Subfile generation number (set negative if not specified).

27. SCIND. "Semi-compiled indicator".

Set equal to 2 if a type 3 semi-compiled record is fed in. Remains equal to 2 until the compiler is about to input the first line following the type 4 terminating record; SCIND is then zeroised by the routine PLIE, which is called from FMAS2.

SCIND is set equal to 1 if a type + semi-compiled leader record is fed in, and remains so until after the type (comma) terminating block, when it is again zeroised by PLIE.

Except when reading semi-compiled, SCIND is zero.

28. POSI

A moving pointer which normally points to the next character to be examined in a field. The input routine (PLIE or PSSE) sets it initially to point to the first character of the statement which has just been input. In the case of PLIE this is the 7th character of CIMAGE [16]; in the case of PSSE it is the beginning of STATEMENT [24]. POSI is used by the routines which analyse input fields, namely GFE1 and GFN.

29. BUFF

A 128-word buffer in the common area AREA. Paper tape, magnetic tape and EDS blocks are read into it initially and then transferred, a record at a time, to CIMAGE [16]. It is also used for processing directory buckets in an EDS input file.

30. NLIND

A character counter/modifier which is set to point to the character after the last significant character which has been read into CIMAGE [16] by the routine INPUT. The counter part is set equal to 72-n where n is the number of significant characters present. The character address part is in the form l.m starting with 0.0 for the first character position in the area.

31. BEMPT

Used when inputting data from E.D.S. It is set zero when the compiler is ready to read in the first bucket of data, indicating to the routine INPUT that no data bucket is yet in store. As soon as the first bucket has been input, it is set non-zero, so that the compiler will not read a new bucket each time it wants to input a record.

32. BKNUM

A decreasing counter containing the number of buckets in an EDS input file, which have not yet been input.

33. BUCIN

Control area for EDS input. BUCIN+4 is incremented directly whenever the next bucket is to be read.

34. SCN1

The bucket-input routine RBUCN sets SCN1 equal to the number of useful words in the bucket whenever a bucket is read from EDS. It is never set to any number greater than 128.

35. SCN7

Used in the EDS input routine. Points to the position immediately beyond the end of the present record in the input buffer, taking the first record to start at position 2.

36. FMLINK

Used for storing the link to the Read From (ED,.....) routine when the compiler is reading from EDS.

37. INLINK

Used for storing the link to PLIE or PSSE when the compiler returns to the Read From (ED,...) routine to find out if there are any continuation subfiles to be input. Set zero on detection of a READ FROM.

38. TERM

On exit from the field analysis routines GFE1 and GFN, TERM always contains the field terminator.

39. NAME

An 11-word area used for holding fields picked up by the field analysis routines GFE1 and GFN.

40. DIR

Used in connection with the output routines. Set zero if a non-directory record is being output. Set non-zero if a directory record is being output, it is sometimes used to hold the link in that case.

41. BLENG

Holds the length of a semi-compiled record as set up by the compiler.

42. BKN2

Holds the bucket number reserved for the next s/c bucket to be written.

43. BKN1

Holds the bucket number reserved for the next directory bucket to be written.

44. CNT

Holds the bucket number of the first un-reserved bucket on the output file.

45. NCØB

Output of S/C buckets (though not of directory buckets) is double-buffered. NCOB points to the buffer which is currently being built up.

46. BUFR2/BUFR3

The two output buffers (128 words each) one of which is pointed to by NCØB [45] at any time. Writing of S/C buckets takes place alternately from these.

47. BUFF1

A 21-word area. The compiler puts semi-compiled records into it starting at BUFF1+1. The output routine puts a word-count on the front before moving it to the output buffer. BUFF1+1 is always pointed to by SCØB [14].

48. LASTB

Contains the bucket number of the last bucket in the file. It is set up initially during the SEND TO routine, but may be altered by the routine WRT6 in the event of file extensions being necessary.

49. OVLLL

Overlay list pointer. This starts by pointing to the beginning of the overlay list, but moves as the list grows, such that it always points to the word after the last item in the list.

50. <u>SIC</u>

If non-zero indicates that the magnetic tape input file contains semi-compiled rather than source. The semi-compiled may be on a simple file or a may be in a subfile on a composite file.

51. <u>SIMF</u>

If non-zero indicates that the magnetic tape input file is a simple file. A simple file can contain semi-compiled only.

52. <u>SCT</u>

Indicates the depth within the subfile structure of the magnetic tape subfile currently being dealth with.

53. <u>SCN3</u>

Used when inputting from EDS. Points to the beginning of an SFD in a directory bucket relative to the start of the buffer BUFF[29]

54. <u>BCNT</u>

Set equal to the first bucket of semi-compiled information on the output file. It is used at the end of the program for calculating the number of buckets in the subfile for insertion in the directory.

P.M.Girard

V. Taylor

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED

Programming Languages Division

FORTRAN NOTE 26A

July, 1967.

Since the issue of Fortran Note 26, #XFAE has been re-organised fairly drastically, and some other modifications have been made. Except where otherwise stated, Fortran Note 26 is still applicable to the new version however. This Note describes the principal changes.

1. General

The purpose of re-organising #XFAE was to reduce its overall size from 11K to not more than 10K. This was made necessary by the discovery that on some 16K machines, Executive might require 6K of store.

The 10K version will necessarily be slower than the original, because normally it will be necessary to read an overlay twice per segment of program compiled.

2. Overlays in the 11K version

There were two overlays as follows:

(1/1) FZ2D, FY2D

(1/2) FG2A, FK2A, FM2A, FN2A, FR2A

Permanent program contained the following segments:-

NAMEXFAE, FA2D, FB2D, FC2D, FE2A, FF2A, FH2A, FI2A, FJ2A, FL2D, FP2A, FQ2A, FS2A, FT2A, FU2A, INTFC.

Where the segment name ends in A, this implies that the segment is identical to the corresponding segment in the magnetic tape compiler.

Overlay (1/1) : FZ2D dealt with the Program Description.

FY2D dealt with basic initialisation, allocation and release of peripherals, READ FROM and SEND TO statements, and the FINISH statement.

Overlay (1/2) : FG2A dealt with EQUIVALENCE and DATA

FK2A dealt with 'Type' statements, and with DIMENSION and COMMON statements.

FM2A, FN2A and FR2A formed the heart of the compiler, dealing with such things as the Polish list and the generation of instructions.

3. Operation of the 11K version

Overlay (1/1) was read in initially and would normally remain in store throughout the processing of the SEND TO (if any), the Program Description, and any initial READ FROM's. At the beginning of each Fortran segment, overlay (1/2) would be read in unless already in store. In practice it would normally remain in store until the FINISH statement was found, unless peripheral switching (which requires overlay (1/1)) occurred meanwhile.

If a SEND TO statement or a statement introducing the Program Description occurred out of context, overlay (1/1) would be read in to deal with it. In fact a SEND TO out of context would be ignored; anything else out of context would result in an error flag.

4. Overlays in the 10K version

This version contains four overlays as follows:

(1/1) FW2D

(1/2) FZ2D

(1/3) FG2A, FK2A, FX2D, FY2D

(1/4) FH2D, FI2A, FJ2D, FM2D, FN2D

Permanent program contains the following segments:

NAMEXFAE, FA2D, FB2D, FC2D, FE2D, FF2D, FL2D, FP2A, FQ2A, FR2A, FS2A, FT2A, FU2A, FV2D, INTFC.

The practice has been continued of changing the last letter of a segment name to D when the segment is no longer identical to the corresponding segment in the magnetic tape compiler. However segments with names ending in D are not necessarily the same in the 10K and 11K versions of the EDS compiler.

Overlay (1/1) : FW2D is a new segment containing the compiler initialisation routine, and the processing of the SEND TO statement. As this is the smallest overlay, there is plenty of room for the SEND TO processing to be expanded, which will almost certainly be necessary in due course.

Overlay (1/2) : FZ2D deals with the Program Description. As it contains a large number of Lower Presets, it was found necessary to put it in a separate overlay to save space.

Overlay (1/3) : FG2A and FK2A are, as their names imply, the same as the corresponding segments in the 11K version. Between them they deal with EQUIVALENCE, DATA, DIMENSION, COMMON and all 'Type' statements.

FX2D is a new segment containing some of the material originally to be found in segment FF2A of the 11K version. It deals with MASTER, BLOCK DATA, FUNCTION and SUB-ROUTINE statements.

FY2D deals with READ FROM and FINISH statements.

Overlay (1/4) : FH2D, FI2A and FJ2D contain processing that was done in permanent program in the 11K version by the corresponding segments FH2A, FI2A, and FJ2A. FH2D deals in addition with PAUSE, STOP and RETURN statements, which were originally part of segment FF2A.

FM2D and FN2D are basically the same as the corresponding segments in the 11K version. However, like FH2D and FJ2D, they have lost a number of routines which had to be put in permanent because they were required by more than one overlay.

Permanent program : This includes segment FR2A which was originally overlaid, but is now used by more than one of the overlays. Segment FV2D is a new segment containing miscellaneous routines which are used by more than one overlay in the 10K version.
Segments FH2D, FI2A, and FJ2D are no longer in permanent program, and segment FF2D is somewhat smaller, containing only the EXTERNAL and FORMAT statements and the generation of Function and Subroutine prologues.

5. Communication between overlays

Reading of overlays is accomplished by the standard overlay package %EROL which is incorporated in the compiler.

For each of the four overlays (1/1), (1/2), (1/3), (1/4) the compiler contains a cued routine. These are GET1, GET2, GET3 and GET4 respectively. Each contains a BRING instruction which causes %EROL to ascertain whether the relevant overlay is already in store or not, and to read it in if it is not. An overlay which is already in store is never read in on top of itself.

GET1, GET2, GET3 and GET4 are used in the following circumstances:

(1) GET1 is used immediately after entering the compiler so that the compiler initialisation can be performed. It is also used when a SEND TO is detected by the routine FMAS2; however normally overlay (1/1) will still be in store when the SEND TO is detected.

(2) GET2 is used when one of the statements which can introduce the Program Description is detected by FMAS2.

(3) GET3 is used in the following cases:

a) On detection of a READ FROM by FMAS2.

b) On detection of a FINISH by FMAS2.

c) On detection by FMAS2 of a statement which can introduce a Fortran segment.

d) On detection by the routine STATE in FE2D of any of the Fortran statements dealt with by segments FG2A or FK2A.

e) When the compiler has been entered at word 27.

f) When a data subfile has just been read from EDS and the compiler wishes to return to segment FY2D to see whether the READ FROM has been fully processed.

(4) GET4 is used in the following cases:

a) On detection by the routine STATE of any statement in the 'Compare' table other than an EXTERNAL or FORMAT statement or other than a statement which requires overlay (1/3).

b) On detection by STATE of any arithmetic type statement.

6. Operation of the 10K version

It is clear from the above that the compiler always checks that the correct overlay is in store before beginning to process any statement which may require that overlay. The new system does not therefore impose any new restrictions on the ordering of statements, over and above those imposed by any of the Fortran compilers.

On the other hand, statements appearing out of context might well cause wastage of time, since it might be necessary to read in another overlay in order to process them.

It is particularly important to adhere to the rule laid down in the Fortran manual that all 'Type', DIMENSION, COMMON, DATA and EQUIVALENCE statements in a segment should be grouped together at the beginning. EXTERNAL and FORMAT statements, however, may appear anywhere in the segment without loss of efficiency.

If these rules are followed, the compiler will normally read (1/3) at the start of each segment, and (1/4) in the course of processing the segment. Two overlays would thus be read for every segment compiled, once the SEND TO and Program Description have been disposed of.

The time required to read an overlay is uncertain at present, but should be much less than one second.

7. Notes on miscellaneous points

1) It was found necessary to duplicate the routine APER by placing identical versions of it in the two overlays (1/1) and (1/3). This was preferable to putting it in permanent, which would have required another 90 words or so of store. Both routines are called APER within their own segments, but the version in (1/1) has been given the cue APERW since it is called from outside.

2) It was found necessary to alter the order of the entries in the tables TAB22, MST2 and MIT2 in segment FE2D so that statements requiring overlay (1/3) could be distinguished from those requiring (1/4) by testing the modifier.

3) A new marker DOIND has been introduced in segment FE2D, whose function is to avoid the call to DO2E (in segment FI2A) unless the correct overlay (1/4) is in store. Since DO loops should never end on non-executable statements it is quite correct to omit testing for the end of a DO loop in that case.

4) Segment FR2A in permanent program branches at one point to a cue in overlay (1/4). Investigation has shown however that this branch is never executed in practice unless overlay (1/4) is already in store.

5) The routine STLST in segment FL2D calls APERW which is in (1/1). However STLST is used only by overlay (1/1) so the correct overlay must be in store.

6) Segment FU2A branches to overlay (1/2). However the correct overlay is necessarily in store when this is done since the marker EMARK is tested first.

8. Approximate sizes of the overlays at time of writing

	PROGRAM	LP and LT
(1/1)	178	42
(1/2)	384	148
(1/3)	1699	62
(1/4)	2131	146

Hence : Overlaid LP area = 148 words.

Overlaid Program area = 2131 words.

9. Other modifications introduced in the 10K version

1) When reading a data subfile from EDS,some checks are made on the format of the input. If the bucket header indicates that there are more than 128 words in the bucket, an error 44 is flagged and the compiler assumes that there are 128 words to be dealt with. If a record has a word count of zero or a number greater than 21, an error 43 is flagged and the rest of the bucket is ignored. If a record has a word count of 1, the record is interpreted as a blank line and no error is flagged.

2) It is now possible to specify a subfile generation number in a READ FROM (MT, ...) statement.

3) If the file name in a READ FROM (MT, ...) statement corresponds with that of a tape which is already allotted, the tape is rewound, rather than released and re-allotted.

4) The compiler will work with the new double-buffered paper tape and card output routines.

5) The processing of the EXTERNAL statement has been re-written as in the magnetic tape compiler.

P.M. Girard.

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED

Programming Languages Division

FORTRAN NOTE 27

August 1967

The Fortran Batch Monitor System

This note is intended to describe the interface between the Fortran compiler XFAS and the monitor program XFBM from the point of view of the compiler. There is also a description of the systems segments which have been changed and some new common areas. Finally there is a brief description of the loader XPO8 and of the new peripheral routines and their interface both with %FINOUT and the system.

© International Computers and Tabulators Limited 1967

1. Introduction

XFBM is a 'trusted' program which monitors the compilation and execution of a series of Fortran source programs, each one being termed a job. To do this it must know what is happening all the time, i.e. whether the object program is being compiled, loaded or run. There are 4 main phases to each job:

1) Job found and compilation started.

2) Compilation completed, object program ready to be loaded.

3) Loading completed, object program ready to run.

4) Run complete, go on to find the next job.

Other phases or combinations thereof may occur, especially under error conditions. For example, if there are errors in compilation phases 2) and 3) are omitted and the compiler goes straight on to the next job.

2. Interface

The interface between XFAS and XFBM consists of a series of DISP instructions. DISP was chosen in preference to SUSWT simply because it is easier to test XFAS apart from XFBM if it does not halt whenever a message has to be given to XFBM. The DISPs fall into two categories:

a) those which give information to XFBM about the progress of a job in the batch.

b) those which require some action from the operator, such as slow peripheral required.

The first character of each DISP is ! and the second ranges from 0 to #14. XFBM uses the ! as identifier and the 2nd character as a modifier to a branch table. Their meanings are as follows with the equivalent HALT in brackets afterwards (if appropriate):

a)	! 5	End of successful compilation	(EC)
	! 6	Object program loaded	(LD)
	! 7	*FOR read	
	! 8	**** read	
	! 9	//// read	
	! :	End of batch	(PF)
	! ;	Overlays loaded	
b)	! 0	CR wanted	(CR)
	! 1	TR "	(TR)
	! 2	LP "	(LP)
	! 3	Incorrect batch parameters	
	! 4	More store required	(ST)
	! <	No SRF6 block	(NL)

DISPs of type a) cause XFBM to test and set various markers, and if directives appear in the wrong order, e.g. two lots of *'s running, messages will be printed on the output device to indicate possible errors in the batch or jobs incorrectly set up. XFBM will normally re-enter the compiler at the next instruction.

DISPs of type b) cause XFBM to halt with a message to the operator. For ease of testing XFAS alone, the next instruction in the compiler is a SUSWT e.g. HALTED CR. XFBM thus re-enters the compiler, when the operator has rectified the condition, at the next instruction after the SUSWT. The only exceptions are ! 3 and ! < when the batch must be restarted from the beginning.

The system is designed to use the system tape as efficiently as possible. The compiler remains in store until there is an object program to be run. The order on the tape

XFAS
SRF6 library block
XPO8 loader

is such that there is at most one scan for each job to be run.

If a job is not to be run, i.e. errors in compilation or NORUN (not a published facility, possibly redundant) then the tape is not moved at all and the compiler goes straight on to skip to the next job. Eventually it is hoped to reorder the compiler so that permanent area comes before overlays on the tape. This will save the initial backspace to find the overlays, and will involve writing a special overlay package.

XFAS is never deleted, as far as executive knows, until the end of a batch. XFBH overwrites the area containing XFAS first with the compiler and then with the loader and object program. Consequently, after the first entry to the compiler, peripherals remain allocated and, in the case of magnetic tape, positioned, throughout the execution of a batch.

3. Description of Segments

Overlay structure

Area 1	Unit 1	FBCT	slow input
Area 1	Unit 2	FBMT	magnetic tape input
Area 2	Unit 1	FG2A, FK2A, FM2A, FN2A, FR2A	Main part of compiler
Area 2	Unit 2	FW2S	Consolidation phase

The segments that have been altered are FW2S, FA2S, FB2S, FD2S, FL2S. The two segments FZ2A and FC2B have been deleted and FBCT, FBMT are new segments. Instead of %TROL, the overlay package is the same as the one in XFOM. The changes to the segments are now described in more detail. Various markers are mentioned in the following descriptions. They are more fully described in the section on common area LC21.

Segment A

This is the supervisor segment as it is all compilers and most sections are the same as in XFAM. All the parts concerned with the final stages of consolidation have been moved to Segment W, some intersegment statements have been removed altogether and minor changes have been made to FINIS and STØP. There is a new section for PAUSE which generates the instructions required to print the message on the system output.

The main change is to the initial entry to the compiler. The first time through for each batch it reads and analyses the batch parameters, and sets markers MARK1 and MARK2 (see section on common area LC21) indicating system input and output devices. Subsequently it discovers system input and output by a series of dummy allot instructions, in fact this is also how it discovers whether it is the first time through or not.

When the correct overlays for the job are in store it enters XFBM, DISP !; , and returns to skip to the start of a job, i.e. *FØRTRAN. The first record presented after the skip entry is listed under the heading ** NEXT DATA RECØRD** unless it is *FØR, **** or ////. This is only relevant when the previous job failed at object time without reading all its data, when this becomes equivalent to 'last card read reversed'.

ENTRY 4

FMAS1

Positions scratch tape, i.e. rewinds and skips to tape mark.

Resets core size to 11008 in case previous compilation or object program required more core.

Initiates storage areas and starts compiling complete program.

This entry is used by XFBM if object program reads *FØR parameter. The compiler is reloaded and started at ENTRY0,1,2 until markers are set and overlays loaded. Then ENTRY 3 is missed out and direct entry made to start compiling.

ENTRY 7

STØP

Terminates offlined output with end of file sentinel.

Enters XFBM | DISP ! : |

Halted PF

No peripherals are released, as if the object program reads ////, on entering the compiler at this point no markers are correctly set. It was thought unnecessary to do a complicated system of allots and releases when the next action of XFBM is to delete XFAS and thus release all peripherals.

Segment B

This contains the cue INPUT plus two new cues INPUT1, INPUT2. It uses MARK1 as a modifier to branch to the correct input section, paper tape, cards or magnetic tape in FBCT or FBMT. On exit, the next record is presented in CIMAGE, and various checks are made.

Entry at INPUT1.

This is entry to read batch parameters. Initial blank records are ignored, otherwise no checks made.

Entry at INPUT2.

This is entry for skip mode. Check for *FØR, **** and ////.

Normal entry at INPUT.

This checks for *FØR, ****, ////, *DAT and *STE.

SETSC (check for S/C) and PCH1 (S/C output section) are also in this segment.

Action of Segment B on finding

1) *FØR : If already within job (tests JMK) an error message is printed, then parameters are handed to XFBM with DISP !7. Return is to FMAS1 (Entry 4) to start compiling.

2) **** : If not within a job, enter monitor DISP !8. Return is to look for *FØR record. If within job, unset JMK, set marker for no data, NØIN. In skip mode (enter at INPUT2) terminate listing and enter monitor as above, otherwise treat as FINISH.

3) //// : If within job, print error message, then and otherwise enter XFBM with DISP !9. Return is to STØP.

Segment D

The only change is the removal of CSLC which builds up consolidated leader and forms request slip.

Segment L

References to tape punch have been removed, and magnetic tape output been included.

As it is not possible to test reply word for 'page full' condition on MT, there is now a line count kept, and up to 60 lines are allowed to one page.

Magnetic tape output is in blocks up to 128 words long in XQMP - compatible format.

Segment FBCT

This is paper tape and card input, TRSPL and CRSPL virtually unchanged.

Card input uses CDMK as ATLAS or 1900 code indicator. It also checks for a record *ATL or *190, changes CDMK when found and returns for the next record.

Segment FBMT

This is magnetic tape input, MTSPL but rewritten. The main section presents the next record in CIMAGE, and there are two subsidiary section: one checks sentinels when a tape mark is read, the other skips to the start of the next job when the compiler is in skip mode.

Sentinel checks

Start of job sentinel (name PRØG). If word 9 of the sentinel is negative or second two characters are not FB skip to the next job. Otherwise zeroize skip marker, ENTMK, and return for next record.

Start of Fortran sentinel. Return to read next record.

Start of Semicompiled sentinel. Set S/C marker, return for next record.

Start of Data or Steer sentinel. Set *DATA or *STEER in CIMAGE and return to process.

End of subfile sentinel. Ignore it in skip mode. Otherwise subtract 1 from subfile level count SCT and if non-zero read next block, if zero treat as *'s read.

End of composile file sentinel. Treat as //// read.

Unrecognized sentinels are ignored and the action is to skip to the start of the next job.

Segment W

This is brought into store and entered as soon as a program requires consolidation. It performs the following actions:

1) Outputs program description segment %%%F containing cues for system input (if data or steering information) and output routines. Also outputs 1st word of %FIØLIST which contains count or entries in list. Its value is 2 if no input data, 4 otherwise.

2) Scans the library block, passes each record through CSLA (consolidator) and outputs routines called for, in batched form if the library is batched.

3) Prints blank cues if there are any.

4) Positions library tape at end of SRF6 block ready for XFBM to read in loader.

5) Forms request slip (section CSLC of consolidator contained in this seg.)

6) Lists leaders if switch 10 on, (not a published facility but useful for testing purposes).

7) Terminates listing.

8) Tests core size of object program and if > current core size of compiler does GIVE for more core.

9) Sets up table of offset, address of consolidated cue list and relativisor settings.

Enters XFBM with DISP !5, with address of this table in X1 and the entry point for the object program in X0, (#200 if GO 20, normal entry; #210 if GO 28, trace steering list entry).

4. Common area LC21

8 wds.

MARK1	Input marker :	Paper tape - 0 Cards (1900) - 1 Cards (Atlas) - 2 Magnetic tape - 3
MARK2	Output marker :	Line printer - 0 Magnetic tape - 1

Above two words set up at start of Segment A as each job is begun. Used in FB2S, FL2S.

GØMK Entry point of object program.

Normal entry #200 i.e. GØ 20
Trace steer #210 i.e. GØ 28

Set up in Segment A when *DATA, *STEER or **** read.
Used in Segment W to pass across to monitor.

JMK Job mark.

Set zero in Segment A before start of each job.
Set non-zero when *FØR read.
Zeroized again in reading *DAT, *STE, ****, ////.

CØRE Current core size, normally 11008.

Set up in Segment A and increased for list size if required.
Used in Segment W to check if room for object program.

NØIN No input data for object program.

Set zero at start of each job.
Set non-zero when **** read at end of compilation indicating no object program data. No input routine will be incorporated in the object program.

RDMK Double buffer mark for cards.

Used only in FBCT, but zeroized initially in Segment A.

CDMK Card code marker.

Set up at start of each job to value of MARK1.
Used in FBCT as card code indicator. Needs to be distinct from MARK1 as it must be re-set at start of each job even if compiler not reloaded. Re-set whenever *ATL or *190 card read (detected in FBCT).

Common area MTUSE preset.
2 wds

LSTRE   Pointer to message   **NEXT DATA RECORD**

Used in Segment A and FBMT.

ENTMK   Set zero for normal mode input in FB2S
Set non-zero for skip mode.

Used in FB2S and FBMT.

5. XPØ8

Loader for Fortran Batch Monitor System.

This loader is based on XPØ1 with only minor modifications. It is always run under the name XFAS, being read into store by XFBM using a CONT.

On entry, X0 contains the address of a table set up by the compiler as follows:

Wd 0	SHIFT i.e. offset
Wd 1	Address of consolidated cue list as set up by the compiler.
Wds 2-11	Relativisor settings.

Thus relativisors and cue list are set up directly from store instead of being read in from MT.

Semi-compiled is input from MT3 which is already allocated and in a rewound position. It positions this tape initially by skipping to tape mark. S/c segments are in both batched and unbatched form: o/p from the compiler being unbatched, library s/rs being batched.

When the program has been loaded, XFBM is re-entered by a DISP ! 6 message in place of the HALTED LD.

6. Peripheral routines

There are 6 new peripheral routines and these are the only ones used by the system. The 4 input ones are %FIBTR, %FIBCR, %FIBCRA and %FIBMTI corresponding to paper tape, cards (1900), cards (ATLAS) and magnetic tape. The 2 output routines are %FIBLP and %FIBMTO, line printer and magnetic tape. All the routines are more straightforward than their counterparts in XFAM. This is for a number of reasons:

a) The peripheral is already allocated and has a fixed unit number.

b) The magnetic tape routines are essentially equivalent to slow input and so process the tape only in one direction, no backspace or rewinding is allowed.

c) Initial blank lines or cards are not ignored. There is an extra check built into the input routines for records *FØR, **** and ////. When these are found, common area %FBMK is set up with two chars ! 7, ! 8 or ! 9 and execution error 6 reported. %FERROR, after giving trace printout, enters XFBM with a DISP of the value in %FBMK.

A consequence of a) is that the interface with %FINØUT is of a fixed form; thus %FIØLIST and %FIØINF are preset within each routine. For a standard job, with input and output statements, %FIØINF and %FIØLIST contain the following values:

%FIØLIST

Wd 0	4	No. of entries in list. Set up by compiler	
Wd 1	2	Peripheral no.	Set up in output peripheral routine
Wd 2	Addr of Wd 0 of %FIØINF		
Wd 3	6	Peripheral no.	
Wd 4	as Wd 2		
Wd 5	1	Peripheral no.	Set up by input peripheral routine
Wd 6	Addr of wd 2 of %FIØINF		
Wd 7	5	Peripheral no.	
Wd 8	as Wd 6		

If there is no data, then wd 0 is 2 and wds 5-8 are omitted.

%FIØINF

Wd 0	Addr of system output routine	Set up by output routine
Wd 1	2 (indicates output)	
Wd 2	Addr of system input routine	Set up by input routine.
Wd 3	1 (indicates input)	

This interface was worked out without considering the possibility of using magnetic tapes at run time and there may be some difficulty in fitting that in to the existing structure. A possible solution would be to revert to compiling %FIØLIST in Segment W.

The slow peripheral routines are straightforward.

%FIBMTØ. This outputs single record blocks in XQMP-compatible format.

%FIBMTI. This assumes input tape positioned just before first record: either data or steer. Each record is presented to %FINØUT in turn, those of less than 80 characters being space-filled up to 80 characters. When a sentinel is read 'start of subfile' is assumed to be DATA subfile and is ignored, level is set to zero
'end of subfile' at level zero is ignored and level set to 1; at level 1 is treated as **** read.
Other sentinels should not occur.

A.S. FINCH.

INTERNAL USE ONL[illegible]

INTERNATIONAL COMPUTERS AND TABULATORS L[illegible]

Scientific Programming Dept.

I.C.T. 1900 Series

FORTRAN [illegible] INDEX
26.1.[illegible]6.

These notes are intended principally for the compiler writers. They are not necessarily kept up to date and should not be taken as definitive.

Fortran Note	Title	Remarks
1	ASA Fortran Arithmetic	See also FN 9
2	A Proposed Method of Describing a FORTRAN Program to the FORTRAN IV (ICT) Compiler	Superseded by FN 19
3	A Proposed Method of Describing and Implementing Program Overlays for the FORTRAN IV (ICT) Compiler	Superseded by FN 10
4	FORTRAN IV (ICT) Input and Output Operations	
5	The FORTRAN IV Object Time Input/Output System	Superseded by FN 21
6	Trace System for FORTRAN 4 and Current 1900 FORTRAN	General Distribution (reproduced in Basic FORTRAN Manual)
7	The FORTRAN IV Object Time Peripheral Routines	See also FN 2, 4 and 5
8	FORTRAN IV Data Storage	
9	Array Addressing in FORTRAN IV	
10	A Proposed Method of Describing and Implementing Program Overlays for the FORTRAN IV (ICT) Compiler	
10a	Amendment to FORTRAN Note 10	
11	FORTRAN IV List System Structure	
12	General Analysis Routines in the FORTRAN IV Compiler	
13	Storage Areas in FORTRAN IV	

Fortran Note	Title	Remarks
14	Some Aspects of Expression Evaluation in the FORTRAN IV Compiler	
15	The Statement Function and IF Mechanism in the ICT FORTRAN IV Compiler	
16	Consolidated Leader Listing and Private Error Codes	
17	Source and Semi-compiled Program Formats on Magnetic Tape	
18	Peripheral Routines	See also FN 7 and 5
19	Program Description and Inter-segment Statements	General Distribution
20	Provisional specification of the 1900 Fortran Compiler	General Distribution
21	The Fortran IV Object Time Input/Output System	See also FN 4, 7 and 18