DUN-2400
UC-32, Mathematics
and Computers

FORTRAN PROGRAMMING

by

P. D. Gross, Supervisor
Computation and Control Unit
Research and Engineering Section
Douglas United Nuclear, Inc.

January, 1968

DOUGLAS UNITED NUCLEAR, INC.
RICHLAND, WASHINGTON

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

FORTRAN PROGRAMMING

PREFACE

This manual has been prepared to familiarize engineers within Douglas United Nuclear, Inc. with the computer programming language FORTRAN IV. FORTRAN IV is basically a "universal" programming language which may be used on most modern data processing computers. Although FORTRAN is only one of several such languages, it is particularly applicable to scientific and engineering applications due to its mathematical-based structure.

The first section of this manual serves to introduce how data are stored and transferred within the basic elements of the computer (including octal and binary notation). Although this information usually proves to be useful, it is not essential to learning the FORTRAN language.

Sections II through VI describe the structure and use of the various basic elements which make up the FORTRAN language itself. Examples are included to demonstrate or clarify usage of these elements wherever necessary.

The last section of this manual describes techniques in setting up an application for programming (flowcharting) and in debugging a program once it has been written.

Although most of the information contained herein applies directly to the FORTRAN IV system implemented on the UNIVAC 1107 and UNIVAC 1108 digital computers, it is, for the most part, also applicable to any other digital computer capable of utilizing the FORTRAN IV language.

P D Gross

TABLE OF CONTENTS

FORTRAN PROGRAMMING - AN INTRODUCTION

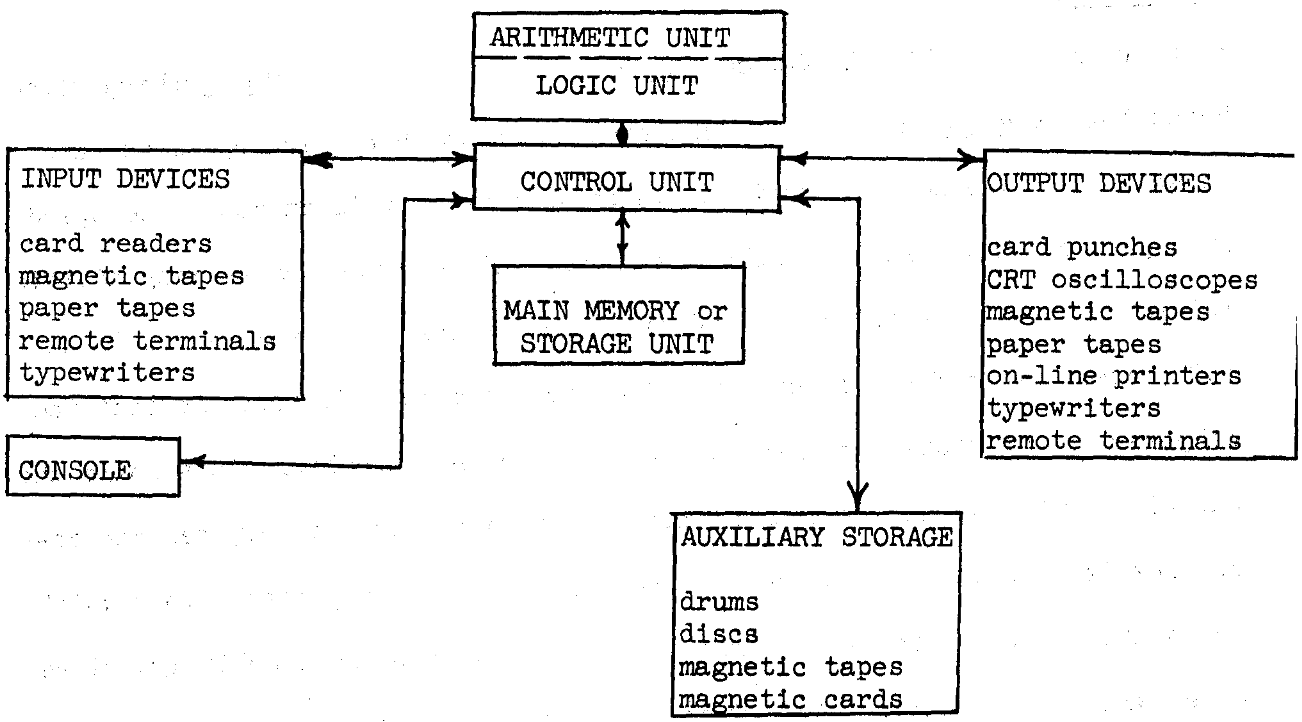
INTRODUCTION

A computer is stupid. It doesn't know anything. It can't do anything without being told exactly what to do. The question that then comes up is why use it at all? The answer is that it can do what it is told at about the speed of light, so that a machine like the UNIVAC 1108 can perform as many additions, multiplications, subtractions, and divisions in one hour as a man could in 150 years working 8 hours per day. Another prime consideration is that the computer will probably not make a single error in the calculations while the man will probably make several million. So much for what a computer can do--now how it does it. A computer program is a sequence of instructions which tells the computer what to do--read in data, add, subtract, multiply, store data, write out results, etc. There are two major problem areas concerned with programming a computer and these are:

1) The computer cannot "understand" English.
2) The computer does exactly what you tell it to and not what you want it to do.

Let's talk about number 1 first; actually the computer can't "understand" anything but is wired to perform certain operations under certain circumstances —that is, when it receives the instructions to perform those particular operations. The instructions are in the form of binary numbers stored in the computer's memory. Both instructions and data are stored and operated on as binary numbers.

A computer looks something like this:



The input devices (card readers, magnetic tapes, paper tapes, remote terminals, and/or typewriters) are used to provide the computer with both its instructions and the data to be operated upon. The control unit keeps track of what's going on and transfers information to and from different sections as required. The arithmetic-logic units are the places in which the actual mathematical and logical operations on the stored data occur. Storage is where both the program and data are kept during computer operation. These data and instructions pass from storage to the control unit and from there to the arithmetic-logic units or to the output devices. The input data and instructions are received from the input devices, the console, or the auxiliary storage and are then placed in main memory or storage before being utilized. The console is where the operator controls the overall operation of the computer--start, stop, execute the program, etc. Output devices may include paper cards (punched), cathode ray tube (CRT) oscilloscopes, magnetic tape, paper tape, on-line printers, typewriters, and remote terminals.

Main memory usually consists of millions of tiny ferrite rings wired together to store data by virtue of their being magnetized in one direction or another (depending upon the direction of flow of current in the wires connecting the rings). Thus, all information is stored as yes - no, or on - off since we are dealing with only bistate (flip-flop) devices. All information used in a computer is thus represented in a binary (on-off, yes-no, true-false) mode.*

* Binary means base two (or two-state) much as decimal means base ten. A binary digit thus may assume either one of two states (0 or 1) just as a decimal digit may assume any one of ten states (0, 1, 2, 3, 4, 5, 6, 7, 8, or 9). Since binary numbers are limited to 0 or 1 in each digit, they correspond to decimal numbers in the following manner

DECIMAL	NUMBER	=	BINARY	NUMBER				
10's	1's		32's	16's	8's	4's	2's	1's
0	0		0	0	0	0	0	0
0	1		0	0	0	0	0	1
0	2		0	0	0	0	1	0
0	3		0	0	0	0	1	1
0	4		0	0	0	1	0	0
0	5		0	0	0	1	0	1
.	.		.	.	.	.	.	.
.	.		.	.	.	.	.	.
.	.		.	.	.	.	.	.
0	9		0	0	1	0	0	1
1	0		0	0	1	0	1	0
1	1		0	0	1	0	1	1
.	.		.	.	.	.	.	.
.	.		.	.	.	.	.	.
.	.		.	.	.	.	.	.

The binary number system only consists of two numbers 0 and 1 to correspond to off and on, true and false, and yes and no. A clockwise magnetic field in a magnetic core may mean yes, on, true, or 1 and a counterclockwise field may represent a no, off, false, or 0. Flow of current in a wire could represent yes, on, true, or 1, and lack of flow of current would then represent no, off, false, or 0.

The value of a binary number is a function of its position; in other words, a binary number has "place" value. For example, in decimal notation the number 532 means five hundred and thirty two. The number 5 is in the hundreds place or position, the 3 is in the tens place or position, and the 2 is in the units place or position. If the 5 were in the thousands place or position and the 3 were in the hundreds place or position and the 2 were in the tens place or position the number no longer would be five hundred and thirty two, but five thousand three hundred and twenty. Thus the place or position of the digits of the number determines the value of the number. Also in the decimal system the number 532 is not equal to 632 even though they each have digits occupying the hundreds, tens, and units positions. In the binary system, however, only the place or position determines the value of the number. For example, 0000 equals 0, 0001 = 1, 0010 = 2, 0011 = 3, 0100 = 4, 0101 = 5, 0110 = 6, 0111 = 7, 1000 = 8, and 1001 = 9; thus the place values of bits are ... 64, 32, 16, 8, 4, 2, and 1. If a 1 bit is present in a place, the corresponding place value is part of the number, if a zero or "no" bit is present the place value is not part of the number. Thus 1 1 1 0 1 0 1 0 0 0 1 is equal to; starting from the right

$1 \times 2^0 \quad = \quad 1$

$0 \times 2^1 \quad = \quad 0$

0×2^2	=	0
0×2^3	=	0
1×2^4	=	16
0×2^5	=	0
1×2^6	=	64
0×2^7	=	0
1×2^8	=	256
1×2^9	=	512
1×2^{10}	=	1024
		1873

Thus, to convert a binary number to a decimal number, the following technique may be used:

```
    1 1 1 0 1 0 1 0 0 0 1

   x2
   --
    2+1
    ---
      3
     x2
     --
      6+1
      ---
        7
       x2
       --
       14+0
       ----
         14
         x2
         --
         28+1
         ----
           29
           x2
           --
           58+0
           ----
             58
             x2
             --
            116+1
            -----
              117
               x2
              ---
              234+0
              -----
                234
                 x2
                 --
                468+0
                -----
                  468
                   x2
                   --
                  936+0
                  -----
                    936
                     x2
                     --
                   1872+1
                   ------
                     1873
```

To convert a decimal number to a binary number the following technique may be used:

	Remainder	
2)1873	1	
2)936	0	
2)468	0	
2)234	0	
2)117	1	
2)58	0	
2)29	1	
2)14	0	
2)7	1	↑
2)3	1	
2)1	1	READ
0		ANSWER

As you can see, it is not very much fun to convert everything into binary before putting it on the computer. So, just to confuse the matter a bit more, let's use octal representation of binary data in the computer. In other words, let's use a system based upon the numbers 0 - 7 (octal) rather than 0 - 1 (binary) or 0 - 10 (decimal).

If you look at groups of only 3 binary places, it is at once apparent that their value can be equated exactly to an octal number and only exactly to an octal number. Three binary places may represent only the following numbers:

BINARY	OCTAL	DECIMAL
000 000	0	0
000 001	1	1
000 010	2	2
000 011	3	3
000 100	4	4
000 101	5	5
000 110	6	6

BINARY	OCTAL	DECIMAL
000 111	7	7
001 000	10	8
001 001	11	9
001 010	12	10
001 011	13	11
001 100	14	12
001 101	15	13
001 110	16	14
001 111	17	15
010 000	20	16
.	.	.
.	.	.
.	.	.

There are too many places to represent a base 7 number and too few for a base 9 number thus the number $110\ 101_2$ equals

$$6\quad 5_8$$

$$5\quad 3_{10}$$

(note the subscript 2 to represent a binary number, 8 to represent an octal number, and 10 to represent a decimal number) and you can see it's much easier to convert binary to octal than to decimal and vice versa. This really hasn't solved the problem of going from the decimal to binary and vice versa yet, however, since we still have the problem of converting octal to decimal and back, but we can attack that similarly to the binary to decimal conversions.

$$1873_{10} = \begin{array}{r} 8\,\overline{)1873} \\ 8\,\overline{)234} \\ 8\,\overline{)29} \\ 8\,\overline{)3} \\ 0 \end{array} \qquad \begin{array}{c} \underline{\text{Remainder}} \\ 1 \\ 2 \\ 5 \\ 3 \end{array}$$

↑ READ ANSWER

$= 3521_8$

$= 011\ 101\ 010\ 001_2$

$= \ 3 \quad 5 \quad 2 \quad 1_8$

$= \ 1 \quad 8 \quad 7 \quad 3_{10}$

In a similar manner:

$$3521_8 = \begin{array}{r} 3\ 5\ 2\ 1 \\ \underline{x8}\qquad\ \ \\ \underline{24+5}\quad\ \ \\ 29\qquad \\ \underline{x8}\qquad \\ \underline{232+2}\ \ \\ 234\ \ \\ \underline{x8}\ \ \\ \underline{1872+1} \\ 1873_{10} \end{array}$$

If we are dealing with fractions:

$$0.5692_{10} = \begin{array}{rr} & .5692 \\ & \underline{x8} \\ .4 & .5536 \\ & \underline{x8} \\ 4 & .4288 \\ & \underline{x8} \\ 3 & .4304 \\ & \underline{x8} \\ 3 & .4432 \\ & \underline{x8} \\ 3 & .5456 \\ + & \\ . & \\ . & \\ . & \end{array}$$

READ ANSWER ↓

$= .4433+_8$

$= .100\ 100\ 011\ 011+_2$

```
                  .5692
                     x2
             .1   .1384
READ                 x2
              0   .2768
ANSWER               x2
              0   .5536
   |                 x2
   ↓          1   .1072
                     x2
              0   .2144
                     x2
              0   .4288
                     x2
              0   .8576
                     x2
              1   .7152
                     x2
              1   .4304
              +
```

$= .100\ 100\ 011+_2$

Or $.100\ 100\ 011_2$ =, starting from the right,

$$\begin{array}{r} 0.5 \\ 2\overline{)1.0} \end{array} \quad + 1. = 1.5$$

$$\begin{array}{r} 0.75 \\ 2\overline{)1.5} \end{array} \quad + 0. = 0.75$$

$$\begin{array}{r} 0.375 \\ 2\overline{)0.75} \end{array} \quad + 0. = 0.375$$

$$\begin{array}{r} 0.1875 \\ 2\overline{)0.375} \end{array} \quad + 0. = 0.1875$$

$$\begin{array}{r} 0.09375 \\ 2\overline{)0.1875} \end{array} \quad + 1. = 1.09375$$

$$\begin{array}{r} 0.546875 \\ 2\overline{)1.09375} \end{array} \quad + 0 = 0.546875$$

$$\begin{array}{r} 0.2734375 \\ 2\overline{)0.546875} \end{array} \quad + 0 = 0.2734375$$

$$\begin{array}{r} 0.13671875 \\ 2\overline{)0.2734375} \end{array} \quad + 1. = 1.13671875$$

$$\begin{array}{r} .5683+. \\ 2\overline{)1.13671875} \end{array}$$

$= .5683+_{10}$

If you recall, we stated earlier that all numbers are represented as binary numbers in a computer. Similarly, all letters in the alphabet as well as several symbols may also be represented as binary numbers in a computer. As seen above each octal number occupies three binary positions. Letters and symbols, however, occupy 6 positions and the following standard representations are used:

CHARACTER	PUNCHED CARD CODE	BCD CODE (MEMORY or STORAGE)
A	12-1	110 001
B	12-2	110 010
C	12-3	110 011
D	12-4	110 100
E	12-5	110 101
F	12-6	110 110
G	12-7	110 111
H	12-8	111 000
I	12-9	111 001
J	11-1	100 001
K	11-2	100 010
L	11-3	100 011
M	11-4	100 100
N	11-5	100 101
O	11-6	100 110
P	11-7	100 111
Q	11-8	101 000
R	11-9	101 001

CHARACTER	PUNCHED CARD CODE	BCD CODE
S	0-2	010 010
T	0-3	010 011
U	0-4	010 100
V	0-5	010 101
W	0-6	010 110
X	0-7	010 111
Y	0-8	011 000
Z	0-9	011 001
.	12-3-8	111 011
)	12-4-8	111 100
(	0-4-8	011 100
+	12	110 000
-	11	100 000
=	3-8	001 011
*	11-4-8	101 100

This is called the BCD or binary coded decimal notation. In this notation, to be consistent, numbers also use 6 positions but for numbers one through seven, the uppermost three bits are blank or zero and 0 is represented as a 10.

CHARACTER	BCD CODE
0	001 010
1	000 001
2	000 010
3	000 011
4	000 100

CHARACTER	BCD CODE
5	000 101
6	000 110
7	000 111
8	001 000
9	001 001

In the UNIVAC 1107 as well as the UNIVAC 1108 and the IBM 7090, a computer word consists of 36 binary bits. This computer word may thus represent 12 octal numbers or 6 BCD characters: for example,

BCD	S		I		M		P		L		E	
BINARY	010	010	111	001	100	100	100	111	100	011	110	101
OCTAL	2	2	7	1	4	4	4	7	4	3	6	5

The computer does all of its addition, subtraction, multiplication, division and logical operations in binary. Input and output are usually converted to BCD so that the machines that punch our output cards and printouts can take the BCD and print or punch decimal numbers and alphabetic letters that we can easily understand.

So, as was mentioned earlier, the computer doesn't understand English--only binary or on-off numbers. Every program instruction and every piece of data is stored in the computer and operated on as either a binary or a BCD number.

In the earliest computers, if one wanted to add A to B and call the result C he had to do something like the following:

```
000 011 000 011 000 001 000 110 000 010
000 111 001 010 000 001 000 111 000 101
000 010 000 100 000 010 000 110 000 111
```

Where these are each three BCD machine instructions, the first 6 bits (left) contain the instruction and the remaining 24 contain the address that the instruction applies to. Thus if 3 means zero out and load the accumulator, 7 means add to the contents of the accumulator (the answer goes back in the accumulator) and 2 means store the contents of the accumulator; and if A, B, and C are stored in memory locations 3162, 0175, and 4267 respectively we will get C = A + B by the above three machine instructions.

It was rapidly apparent that there must be a faster and simpler way to program than this. As a result, a machine-oriented or "assembler" language was soon developed which replaced binary notation with mnemonics and let the computer itself do the translation.

The three above statements could then be written as something like this:

```
ZLA  3162
ADD  0175
STA  4267
```

The next step in the evolution of programming techniques was to formulate a language which had "macro" instructions composed of many of the above machine-oriented language instructions. The most common example of these currently in use in the United States are FORTRAN (FORmula TRANslation) and COBOL (COmmon Business Oriented Language). In FORTRAN, a scientifically oriented language, the above instructions may be written as

```
C = A + B
```

while in COBOL one might say

```
ADD A,  B GIVING C.
```

In this example we have replaced three machine instructions with a single "macro" instruction. When the FORTRAN or COBOL program is "compiled" by the computer, these macro instructions are broken down into the basic machine-oriented language instructions (SLEUTH for the UNIVAC 1108, FAP or MAP for the IBM 7090) and are then "assembled" to form the actual machine program using the binary instructions. The "assembler" also assigns the necessary storage locations for the program as well as for data.

FORTRAN then is not "machine-oriented" but is a "problem-oriented" language. The FORTRAN language statements are written on special FORTRAN coding forms which are then copied onto 80 column punched cards by keypunch operators. These cards containing the program instructions together with cards containing data to be operated upon are fed into the computer by inserting them into an on-line card reader or transferring the information from cards to magnetic tape which is then read into the computer.

PAGE ____ OF ____

FORTRAN CODING FORM 2

CODER

DATE

PROBLEM NO.

C ← FOR COMMENT

STATEMENT NUMBER

CONTINUATION

F O R T R A N S T A T E M E N T

IDENTIFICATION

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25

54-3000-560 (6-66) AEC RL RICHLAND, WASH.

Notice that the coding form, reproduced on the preceding page, also has 80 columns (like the punched cards) and is broken up into 4 major fields or areas. The first field is in columns 1-5 and it contains "statement" numbers which may be used to reference the FORTRAN statement appearing in the rest of the card. It is not necessary to have a number in the columns 1-5 unless you want to reference or go to that statement from somewhere else in the program. Statements may have any number from 1 to 32767. (One never includes comma's in FORTRAN numbers.) Column 6 is the "continuation" column and is used to tell the computer that the card contains a FΘRTRAN statement continued from the previous card (this is done by placing any non-zero characters in column 6). Usually the first continuation card is punched with a 1 in column 6, the second continuation card with a 2, etc. A maximum of 19 continuation cards is permitted per statement on the UNIVAC 1108.

Columns 7-72 contain the FORTRAN statement itself (which, as indicated above, may be continued to additional cards if necessary). It is usually good practice to use several short FORTRAN statements rather than one long one, to avoid mistakes. The computer ignores blank spaces in columns 7-72 except when they appear within a number (and then they <u>may</u> be considered to be zero, depending upon the computer).

Thus, as was mentioned earlier, the computer doesn't understand English, but FORTRAN comes rather close to the "language" of mathematics once you learn a few basic rules. At that time it was also mentioned that the computer does exactly what you tell it to do and not what you want it to do. This may seem facetious but it is very unfortunstely true.

If, for example, you are trying to find the volume of a sphere of a known diameter and you write that

$$\text{VOLUME} = \frac{3.14159\ (\text{DIAM})^3}{8.0}$$

Every volume calculated will be $\frac{\pi}{8} D^3$ rather than $\frac{\pi}{6} D^3$ as it should be. The program is definately not supplying the correct answer, but it certainly is doing what is "correct" in that it is doing exactly what you told it to do.

If, in averaging 23 numbers, you calculate only 22 of the numbers and divide their sum by 23, your answer is wrong but you are getting the "correct" solution to the problem that you have specified even though that is not the problem you want to solve.

Thus in writing your program your first step is to define exactly what it is you are attempting to do. This definition must include every equation and relationship involved in the correct sequence of their execution. You can't write a program to try one approach to a solution and then, if it doesn't look like you are getting the correct answer, have it try adding, subtracting, multiplying, or dividing by other numbers to get the "right" answer. You have to know exactly what it is that you want to do. This leads to the question "what should one put on the computer?"

A payroll may be calculated in a rather simple fashion:

GROSS = hours worked (dollars/hour) + overtime + shift differential + holiday pay, etc.

DEDUCTIONS = withholding + social security + insurance + savings plan + United Crusade + credit union + U. S. Savings Bonds, etc.

NET = GROSS - DEDUCTIONS

This calculation is really quite simple but it lends itself to the computer quite nicely since it must be repeated many hundreds or thousands of times per week. Thus a problem which has many sets of data to be processed is a good computer application.

Take a different type of calculation; in a very simple analysis, a nuclear reactor loading may be described in terms of the "buckling" of the materials present. The buckling is really the rate of change of the flux in that material. It is also known that the slope of the flux = 0 at the center of the reactor and that the flux = 0 at the outer edge of the reactor that is just critical. Thus to determine how far from critical a reactor loading is, one could "guess" a change in material buckling for each zone and from that calculate a flux distribution and see if it has zero slope at the center of the reactor and if it goes to zero at the outer edge. If it did, your problem is solved. But unless you are one heck of a good guesser it didn't and you will have to try new sets of bucklings. An iterative problem like this lends itself quite well to computer applications since if you start out with a zero slope at the center, and the flux is negative at the outer edge, the zone bucklings are too high and must be reduced. If you start with a zero slope at the center and flux at the outer edge is positive, you must increase the buckling in each zone to reduce flux at the edge. You keep changing the buckling and finally the flux is close enough to satisfy your error criteria. This may take 5, 50, or 500 iterations, but the equation will still be solved much more rapidly on the computer than by hand. Oddly enough, the real problem in this application is not to set up the mathematical model of the problem but to define for the computer how much to change the bucklings, to iterate to the "correct" solution as rapidly as possible. Thus problems which require many iterations to achieve a solution or problems containing many sets of data or many large and complex relationships are all quite applicable for computer usage. An additional class of problems that have wide computer applications are those that require taking very small time, distance or temperature steps to achieve the correct solution. These are usually combined simultaneously with iterative problems. For instance, calculations of transient temperatures during a reactor startup or scram requires consideration of many iterations and many short time steps as well as small physical volumes to achieve the correct results--there is also much feedback between power change and temperature from the standpoint of associated reactivity effects of change in fuel and moderator temperature. Solutions of problems of this type could not be attempted on a fine scale without the aid of a computer.

On the other hand, some calculations are best left off the computer. These are the simple single solutions with few complications.

Daily calculations of plant throughputs, for instance, might require more time to punch the input data on cards than for the user to solve the problem by hand. If all 365 sets of data were to be calculated at one time, and used as the basis for further calculations it might be reasonable for the computer to be used--particularly since no errors would be expected from the computer results (providing the program is correct and all the input data are transcribed correctly) while several dozen errors would probably occur in the hand calculations. However, a daily calculation of this type is probably not applicable.

SECTION II - ARITHMETIC STATEMENTS, CONSTANTS, AND VARIABLES

As mentioned in the previous section, FORTRAN stands for FORmula TRANslation and is a "problem-oriented" language--that is, it much more closely resembles the language of mathematics than the language of computers. FORTRAN is also a "machine-independent" language. In other words, a FORTRAN program, once written, may be run on virtually any computer having a FORTRAN "compiler". The main problem in running FORTRAN programs on different computers is that, even though the FORTRAN program says the same thing to both computers, the "compiler" (or translator to machine language) will probably not produce exactly the same machine language program for each computer. The reasons for this are severalfold. In the first place, there are many versions of FORTRAN --II, IV, V, etc. as well as different "versions" of each version, and a FORTRAN II program may not work out too well on a FORTRAN V compiler. Also, the compiler is really only a program itself which was designed by the computer manufacturer or a software firm to translate the FORTRAN program into an *efficient machine language program* for use on *that particular model* of computer. This sounds great, until you realize that different computers do things differently so that even adding 1 + 1 may be done quite differently--not only by computers built by different manufacturers, but even by different computer models of the same manufacturers. The second hooker mentioned above is the phrase "efficient machine language program". Compilers tend to *move* parts of your program around during translation to machine language to make it as efficient to run on the computer as possible. Thus, a really "sophisticated" compiler may do things very differently than a somewhat less sophisticated compiler, and the results, even when run on the exact same computer, could be startlingly different.

Thus anyone who does very much FORTRAN programming gets to learn a little about how "his" compiler works--or at least he gets to know someone else who knows this. Before we worry about compilers, however, let's get a little FORTRAN under our belt.

A FORTRAN program is merely a deck of cards (called a "symbolic" deck) which, when loaded into the computer with the appropriate FORTRAN compiler, will result in an "object" or machine language deck that can be operated on by the computer. The FORTRAN program can vary in length from a few FORTRAN "statements" to many thousands of FORTRAN "statements", where the FORTRAN "statement" is usually the equivalent of many machine language instructions.

Since FORTRAN is mathematically oriented, the simplest statements concern everyday addition, subtraction, multiplication, division, and exponentiation.

Each one of these five operations is represented in FORTRAN by a special symbol as follows:

+ means "add"

- means "subtract"

* means "multiply"

/ means "divide"

** means "raise to the power"

Thus if one wanted to write the following expression

$$x = \frac{y^{3.6} + z^{9.72}}{s + q \quad \sqrt[3]{r}}$$

in FORTRAN it would be as follows:

```
X = (Y**3.6 + Z**9.72) / (S + Q * R**(7./3.))
```

Note that only capital letters are used since FORTRAN has no provisions for lower case letters.

Notice also that parentheses were used to separate parts of the expression. It is allowable to use as many sets of parentheses as is necessary to define the equation and it is usually better to have too many parentheses than too few. Extra sets of parentheses will be ignored, but missing ones can't be put in by the computer since it won't know where they go. The computer does assign a "weight" to each of the five above operators and it is as follows:

OPERATOR	FUNCTION	WEIGHT
**	Exponentiation	3
*	Multiplication	2
/	Division	2
+	Addition	1
-	Subtraction	1

If statements contain no parentheses, the computer will evaluate the expression from left to right in the order the terms appear unless the operation to the right of the one being examined has a higher weight than the one being examined--in that case, the adjacent operation with the higher weight is performed first. For example:

```
X = Y - Z * R ** 3.2
```

is $x = y - z(r^{3.2})$

whereas

```
X = Y - (Z * R) ** 3.2
```

is $x = y - (zr)^{3.2}$

and X = Y - Z ** R * 3.2

is $x = y - 3.2z^r$

and X = Y ** Z - R * 3.2

is $x = y^z - 3.2r$

and X = (Y - Z) ** R / 3.2

is $x = \frac{(y - z)^r}{3.2}$

and X = Y / Z / R / 3.2

is $x = \frac{y}{3.2zr}$

whereas X = (Y / Z) / (R / 3.2)

is $x = \frac{3.2y}{rz}$

Note that inclusion of parentheses will overide the "built-in" weighting factors assigned to the operators.

The variables X, Y, Z, and R, above are representations used by the programmer; however the machine considers them merely as "storage locations". In other words, X is some location in memory, say 0967, and Y, Z, and R are also, as far as the computer is concerned, merely memory or storage locations.

A FORTRAN arithmetic relationship thus differs from mathematical equations in that FORTRAN statements may have only one variable or constant on the left side of the equal sign. Thus

X = Y + Z

is permitted but

Y + Z = X

is not allowed.

X = Y + Z means "add the contents of the storage location assigned to

Y, to the contents of storage location assigned to Z, and place the result in the storage location assigned to X".

Thus

X = X + 4.

is a valid FORTRAN expression even though it does not look "correct" mathematically speaking. In essence it says--"add 4. to X and store the answer in X".

Now that the mathematical operators are defined, let's spend a minute or two on defining variables and constants.

Since variables and constants are really only "storage locations" (to the computer) to which we have assigned "names" for our convenience of representation, we must adhere to certain "naming rules" so that the compiler knows how to handle such data.

The first rule is that the names used to represent variables or constants may not contain more than six characters (although they may have less--in fact a single alphabetic character may be a valid name.) The name is not permitted to start with anything except an alphabetic character. The name may contain alphabetic characters and numbers in any desired sequence provided they start only with an alphabetic character. Names may not include any symbol such as +, -, *, /, ,, ., ', (,), =, etc. since the compiler is not smart enough to realize you meant the single variable WA-RP and not the variable WA minus the variable RP.

Another rule is that any name starting with either an I, J, K, L, M, or N is considered to represent an integer variable whereas those names starting with alphabetic characters other than I, J, K, L, M, or N are considered to be real variables.

Integers are represented in memory as just that--integers. For example, if IX = 483_{10} (or 743_8) a 12 character octal representation of the 36 bit binary "word" in memory would appear as this

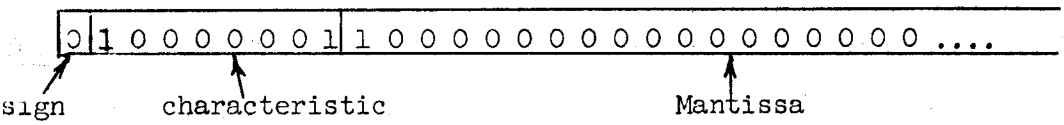
+	0	0	0	0	0	0	0	0	0	7	4	3

IX in Memory Dump (octal)

Real numbers are not stored as integers in memory--mainly because they are not integers and, since they may have values in the range of 10^{-38} to 10^{+38}, they could hardly fit in a 12 character octal integer "word". Instead, real numbers are stored as shown below

S	CHAR.	MANTISSA

where S, the left-most binary bit in the 36 bit binary (12 character octal) word represents the sign of the number (+ or -), the next 8 bits represent the characteristic, and the remaining 27 bits represent the mantissa. The mantissa is of the form $.XXXXXXXXX_8$ whereas the characteristic would be $n_8 + 200_8$ if the number is represented as $(.XXXXXXXXX)\ 2^n$. Thus, to represent the number 1_{10} in the computer we would get $1_{10} = 1_8 = 1_2 = (.1 \times 2^1)_2$ or the word in binary would have a characteristic of 200 + 1 = 10 000 001 and a mantissa of .100 000 000 ....



Since we can group three binary bits to form each octal character, the above word would appear as shown below in octal (for example in an octal memory dump) realizing that $.100_2 = .4_8$

$$2\,0\,1\,4\,0\,0\,0\,0\,0\,0\,0\,0_8$$

Similarly the number 4 could be represented as $4_{10} = 4_8 = 100_2 = (.1 \times 2^3)_2$. or 2 0 3 4 0 0 0 0 0 0 0 0 $_8$ in octal. The number 10 could be represented as $10_{10} = 12_8 = 001\ 010_2 = (.1010 \times 2^4)_2$ and since $.101_2 = .5_8$ the word would look like 2 0 4 5 0 0 0 0 0 0 0 0 $_8$ in octal.

This now raises the question of how one would represent a number like 7.4692×10^{17} in FORTRAN. It is obvious that you can't keypunch $X10^{17}$ since you can't shift up half a line on a FORTRAN card. Instead you merely replace $x\ 10^{17}$ by E17 or "exponent of 17" thus one could write $x = 7.4692 \times 10^{17}$ in FORTRAN as

```
X = 7.4692E17
```

or

```
X = 7.4692E+17
```

The + in the exponent as well as a + for the number is "understood" if it is not included. However, all - signs must be included, as

```
Y = -9.2436E-4
```

to represent $y = -0.00092436$.

Since a - is a mathematical operator as well as a "minus sign", one must be somewhat more careful in using it than might be expected (since two operators are not permitted to be adjacent). Thus to multiply -X by -17.6(Y) one must write it as (-X)*(-17.6)*Y using the parentheses to "attach" the sign to the variable X or the constant 17.6 and not mix it up with the operator*.

Real numbers are represented by their having a decimal point, whereas integers are conspicuous by the absence of a decimal point. Thus, even though <u>you</u> "know" that 7. is an integer, the FORTRAN compiler calls it a real number because it has a decimal. If one were to write

```
INK = 4.
```

the computer would first have to change the real number $2\,0\,3\,4\,0\,0\,0\,0\,0\,0\,0\,0_8$ to the integer $0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,4_8$ before it stored it in the location representing the variable INK. Thus the following are valid FORTRAN integers

```
IP = 9

J = -17394

MRS = 2741

KP194 = 2

N47B6R = -19
```

and the following are valid real numbers

```
ZETA = 1.9467322E-27

P = 1.0

UR = 1743.99

QM = -1.6E06

A = 99.32 / 7.6E-4
```

As mentioned above, real numbers must be in the range 10^{-38} to 10^{+38}. Notice that no commas are permitted in numbers represented in FORTRAN.

If one has integral powers in an expression it is much more efficient to use the integer than a real number for the exponent. In other words,

```
X = Y**9
```

is preferred to

```
X = Y**9.
```

of course if the exponent is a real number like 7.63 you don't want to use an 8 or a 7.

When real numbers are multiplied, added, divided, subtracted, or raised to a power, a real number is the result of the calculation. Likewise integer arithmetic will result in integer answers. The computer always truncates the results of integer division so that if one had IX = 7/2 the result would be IX = 3 likewise MRK4 = 1-1000*(99/100) would yield MRK4 = 1 since 99/100 truncates to 0; however MRK4 = 1-1000*99/100 would result in MRK4 = -989 since * and / have the same weight and 1000 * 99 would be performed before /100.

If you were to divide a real number by an integer or vice versa you would have problems called a "mixed expression" on the IBM 7090, but the UNIVAC 1108 FORTRAN compiler is clever enough to convert the integer to a real number before performing the operation. Even though the UNIVAC 1108 FORTRAN compiler will "look after you" it is best not to mix expressions when possible just to be safe (you might run the program on some other computer some day).

Another example of mixing integers and real numbers, but one that is commonly done, is the use of different modes on different sides of an equal sign, thus

```
             A = 19/4 + 5/4
               = 4 + 1
               = 5.
AND         IR = 19./4. + 5./4.
               = 4.75 + 1.25
               = 6
```

are both valid--the computer evaluates the expression in the appropriate mode and then converts the solution to the mode required for storage as the answer.

If you recall, any variable whose name starts with I, J, K, L, M, or N was defined to be an integer variable (or constant). There is a means of overriding this definition and that is to use a TYPE statement at the start of your program. For instance you may specify IMAX, J, KAY and MØ to be real variables by saying

```
REAL IMAX, J, KAY, MΘ
```

Note the line through the Θ in MΘ to differentiate the letter Θ from the number 0 as in the name MO.

Similarly, X, YY, ƵEL, and P may be specified to be integer variables by

```
INTEGER X, YY, ƵEL, P
```

If a name is <u>not</u> found in a TYPE statement, the I-N ruled for integers will still apply.

Besides the INTEGER and REAL Type statements there are four other Type statements. One of these is CΘMPLEX to define complex numbers of the type 7. + 9.4i (represented as (7., 9.4) in FΘRTRAN).

For instance the names C1, C2, ƵK, Q, and RP may be set up as complex variables by

```
CΘMPLEX C1, C2, ƵK, Q, RP
```

and one could then set zk = 9.642 + 1.1i by

```
ƵK = (9.642, 1.1)
```

or

```
ƵK = (0.9642E1,11.E-01)
```

Each complex variable uses two consecutive storage locations--the first for the real part (9.642 above) and the second for the imaginary part (1.1 above). Integers are <u>not</u> permitted as complex variables. All complex numbers <u>must</u> be defined by a CΘMPLEX Type statement.

There is another kind of number stored in the computer and defined by a Type statement--that is the DΘUBLE PRECISIΘN number. It may appear that 8 or 9 significant figures are a lot and that they give plenty of accuracy, but you can never please everybody, so to permit calculations having 16-18 significant figures, a double precision number is used. Every double precision number must be defined by a DØUBLE PRECISIØN Type statement as for Dl, RƵ, IV, NK below

DØUBLE PRECISIØN Dl, RƵ, IV, NK,

Each double precision number is stored in two consecutive storage locations in memory with the most significant figures in the first storage location and the least significant figures in the next location. The characteristic of the first location is as normally calculated, that of the second location is the first -27 (since the first 27 binary bits went into the first word and bits 28-54 will go in the second word). All double precision numbers are represented with a D rather than E to signify the exponent so

```
DI = 9.3D+6
RƵ = 1.446392117138D+0
IV = 6.333333333312D-4
NK = 1.2D+1
```

are all double precision numbers (note that they had to be defined in the DØUBLE PRECISIØN Type statement as well as have the D in the number).

Another Type statement is the LØGICAL Type statement to define logical variables. A logical variable is a variable which may assume the value true or false (represented by a 1 or 0 respectively in memory). Thus to make L1, JK, R7, M39, PIN36 logical variables we must use the following statement:

LØGICAL L1, JK, R7, M39, PIN36

and we may then set

```
L1 = .TRUE.
JK = .TRUE.
```

R7 = .FALSE.

etc.

Logical variables may not assume the values of numbers--only .TRUE. or .FALSE. Note the periods necessary on each side of the .TRUE. or .FALSE. datum.

There are three logical operators, .ΘR., .AND., and .NΘT. (which must also be set off by periods) as well as six logical relational operators, .EQ. (equal to), .NE. (not equal to), .GE. (greater than or equal to), .GT. (greater than), .LE. (less than or equal to), and .LT. (less than), again all set off by periods; which are used in logical manipulations just as the **, *, /, +, and - are used in arithmetic manipulations.

Now that you know all about numbers in FΘRTRAN, consider the problem you have if you want to operate on say 500 numbers, you certainly don't want to dream up 500 FΘRTRAN names, one for each number--you would then have to write 500 sets of all arithmetic expressions to operate on each name (or number). Thus you say, O.K., let's have 500 X's and we will call the first x-X(1), the 23rd x-X(23), and the 500th x-X(500). X is called a "subscripted variable" or an "array" and one merely decides which "X" he wants to use and that number (say the ith) is the subscript used.

A variable may have from 1 to 7 subscripts on the UNIVAC 1108. Thus one might have stated X(2,5,5,10) just as well as X(500) and still achieved the same result--500 X's all stored in sequence from X(1) or X(1,1,1,1) to X(500) or X(2,5,5,10) in memory. They would be stored in the sequence X(1), X(2), X(3), X(4)...X(500), or X(1,1,1,1), X(2,1,1,1), X(1,2,1,1), X(2,2,1,1)... X(2,5,5,10).

The subscript must be an integer and may not itself be subscripted. It may also be a product, sum, or difference of integers. The following subscripts are valid:

```
X(I)
R(K+93)
MILK(LL+91)
B3K92(6*M+9)
F6MIX(3+L, 4*N-2, K, IR)
```

but

```
RLP(M(71))
```

is not valid since the subscript itself is subscripted. Subscripts should not be zero or negative even though such subscripts are permitted (X(0) would be stored next to X(1) backwards in memory, X(-0) next to X(0), X(-1) next to X(-0), etc.). Each subscript is separated from the adjacent subscript by a comma. The group of subscripts are enclosed in a single set of parentheses.

Integer, Real, Double Precision, Complex, and Logical Variables may all be subscripted. The maximum size of each array must be defined. This may be done in a DIMENSIØN statement as follows:

```
DIMENSIØN X(9,3,210), Y(10), IX(2)
```

The dimensions of an array may also be specified in a TYPE statement if the variable name itself appears in a TYPE statement, for example,

```
DØUBLE PRECISIØN VM(7,32), R(6,2,11)
```

If a variable is dimensioned in a TYPE statement it must *not* appear in a DIMENSIØN statement; similarly if a variable is dimensioned in a DIMENSIØN statement, it may *not* appear *with dimensions* in a TYPE statement but it *may* appear in a TYPE statement without dimensions, for example:

```
DIMENSIØN X(73), KY(9), R(2), MM(9)
INTEGER X, P(93), V, W7
REAL KY, IB72, MM, MN(643,2)
```

are all valid statements.

Whenever a subscripted variable (dimensioned) is used, it may not be used within the program without a subscript to denote which element within the array is referenced. For example:

```
REAL IXK, MV(72), LK39(3,7,964), II, IJ, IK
DIMENSIØN KK(2), ILR(9), PPT(77)
IXK = 9.64
R = 6.942E-30
LK39(1,7,32) = -99.4
IL = 17
KK(1) = 19
PPT(3) = 2.7E4
```

are all valid statements but

```
MV = 6.2
```

and

```
ILR = 10
```

are not since these variables represent arrays and it is not stated which element of the array is referenced. (The compiler will assume that in these instances the first element is referenced so it will set MV(1) = 6.2 and ILR(1) = 10.)

To determine the location of a particular element within an array, the following formula may be used:

Location of X(I1, I2, I3, ...I7) within array X(D1, D2, D3,...D7) = location of X(1, 1, 1, 1, 1, 1, 1) + (I1-1) + (I2-1) * D1 + (I3-1) * D1 * D2 + (I4-1) * D1 * D2 * D3 + ... + (I7-1) * D1 * D2 * D3 * ... * D6. For example, the location of RL7(2,1,4,3,4) within the array RL7(5,5,5,10,10) may be calculated as the storage location of RL7(1,1,1,1,1) + (2-1) + (1-1) *5 + (4-1) * 5 * 5 + (3-1) * 5 * 5 * 5 + (4-1) * 5 * 5 * 5 * 10 = location of RL7(1,1,1,1,1) + 1 + 0 + 75 + 250 + 3750 = location of RL7(1,1,1,1,1) + 4076.

The above rule applies for all arrays except those of complex or double precision numbers. In the latter case, two adjacent storage locations are required for each number. Thus for double precision or complex numbers, the location of X(I1, I2, I3 ... I7) within array X(D1, D2, D3, ... D7) is the loaction of X(1,1,1,1,1,1,1) + 2 * (I1-1) + 2 * (I2-1) * D1 + 2 * (I3-1) * D1 * D2 + 2 * (I4-1) * D1 * D2 * D3 + ... + 2 * (I7-1) * D1 * D2 * D3 * ... * D6.

SECTION III LOOPING AND TRANSFER OF CONTROL

LOOPING

Now we know all about how to store many sets of data in large arrays. This leads to the next question--what good is the use of such an array (or arrays)?

It's true that if you had ten numbers stored in X(1) to X(10) and wanted to find $y = \sum_{i=1}^{10} x(i)$ you still have to add all ten numbers up whether they are called X(1), X(2), X(3), ... X(10), or X1, X2, X3, ... X 10. The advantage of the former notation lies in the use of "indexing" or selecting the particular variable of interest in the array.

Thus, although

$$Y = X1 + X2 + X3 + X4 + X5 + X6 + X7 + X8 + X9 + X10$$

as well as

$$Y = X(1) + X(2) + X(3) + X(4) + X(5) + X(6) + X(7) + X(8) + X(9) + X(10)$$

one has no choice but to calculate Y as above using X1, X2, etc. but there is a much more efficient method of adding in the latter case. That is

$$Y = X(I) + Y$$

provided Y = 0.0 before we start, and the index I assumes all values from 1 through 10 inclusive. The real question is how we do the latter--and the answer, strangely enough, is to DØ it - using a "DØ loop". A DØ loop says "do what follows as many times as necessary to satisfy the index requirements". The DØ statement is of the form

DØ N I = J, K, L

where the integer N tells the computer just what statements it is to "DØ", the integer I is the index, the integer J is the first value the index is to assume, the integer K is the last value the index is to assume, and the integer L is the increment of the index. If L is left out it is assumed to be equal to the integer 1.

The DΘ statement in effect says "do everything from here to statement N one time for each value of the index I specified by J, K, and L".

Thus to add the ten values of X above we would do it as follows

```
          DIMENSIΘN X(10)
          Y = 0.0
          DΘ 20  I = 1, 10
    20    Y = Y + X(I)
```

This may not seem like a tremendous savings, and in this instance it probably is not, but imagine if we had say 5000 different values of X to add--the exact same three instructions will accomplish the 5000 additions as it did for the 10 (provided we index I from 1 through 5000 rather than 10).

Note that the statement N (20 above) is executed for <u>each</u> value of I. After the statement numbered N is executed, control returns to the DΘ statement, the index I is incremented by L, and if I exceeds K, control is then sent to the statement <u>following</u> statement N.

As many statements as are desired may be placed in the "loop" (between DΘ N I = J, K, L and statement number N). It is most efficient, however, to place within the "loop" only those statements which <u>must</u> be executed for <u>each</u> value of the index I.

Statement N may be virtually any statement (arithmetic, logic, or input/output) except that it <u>cannot</u> be another DΘ statement or a transfer or test statement (more about these later).

One special statement is commonly used to end a DΘ loop and that is the CΘNTINUE statement. The CΘNTINUE statement says just that--*continue* or "ignore me". It doesn't cause anything to happen except control to be transferred to the next statement (or back to the DΘ statement if it is the end of a DΘ loop). The above problem could have thus also been written as

```
           DIMENSIΘN X(10)
           Y = 0.0
           DΘ 20  I = 1, 10
           Y = Y + X(I)
     20    CΘNTINUE
```

Within the loop from DΘ N I = J, K, L to statement N, neither I, J, K, nor L may be redefined. That is, one may *not* have *any one* of these integers on the left side of an arithmetic statement within the loop.

In DΘ N I = J, K, L it is normally assumed that L is positive, that is K > J; however, it *is* permitted that J > K and L < 0. If J > K, L must be set equal to -1, -2, etc.

Thus, the above problem could have been written

```
           DIMENSIΘN X(10)
           Y = 0.0
           DΘ 20  I = 10, 1, -1
           Y = Y + X(I)
     20    CΘNTINUE
```

and X(10) would have been added to X(9) to X(8) to X(7) ... X(1) to achieve the same answer as before.

The most common limits of DΘ's are from 1 to K (where the limits of a

DΘ are the numbers J and K in DΘ N I = J, K, L). The "range" of a DΘ is the set of statements between the DΘ statement and statement N.

The integers I, J, K, and L may be integer constants such as 1, 10, 1000, 325, etc. or they may be integer variables <u>provided</u> they are <u>not</u> subscripted variables.

One is <u>not</u> permitted to <u>transfer</u> into the middle of a DΘ loop but must enter through the DΘ statement itself. It is permitted, however to transfer out of a DΘ loop (provided it is not at statement N, the end of the loop). The reason for this is that one must set up all the index limits and the index itself (I, J, K, and L) <u>before</u> entering the loop, and the only way to do this is by passing through the DΘ statement. Once the index and limits are set up, it is permitted to transfer <u>out</u> of the loop before the loop has calculated all of the range for the limits of the index (provided one does <u>not</u> return to <u>that point of exit</u> from the loop but back to the DΘ statement if the loop must be used again).

DΘ loops may follow one another in a program or they may be "nested" within one another. Nested DΘ's must be <u>entirely</u> within each other.

For instance, say you wanted to calculate

$$x = \sum_{i=1}^{10} y_i^2 + 17.3 z_i \quad \text{where}$$

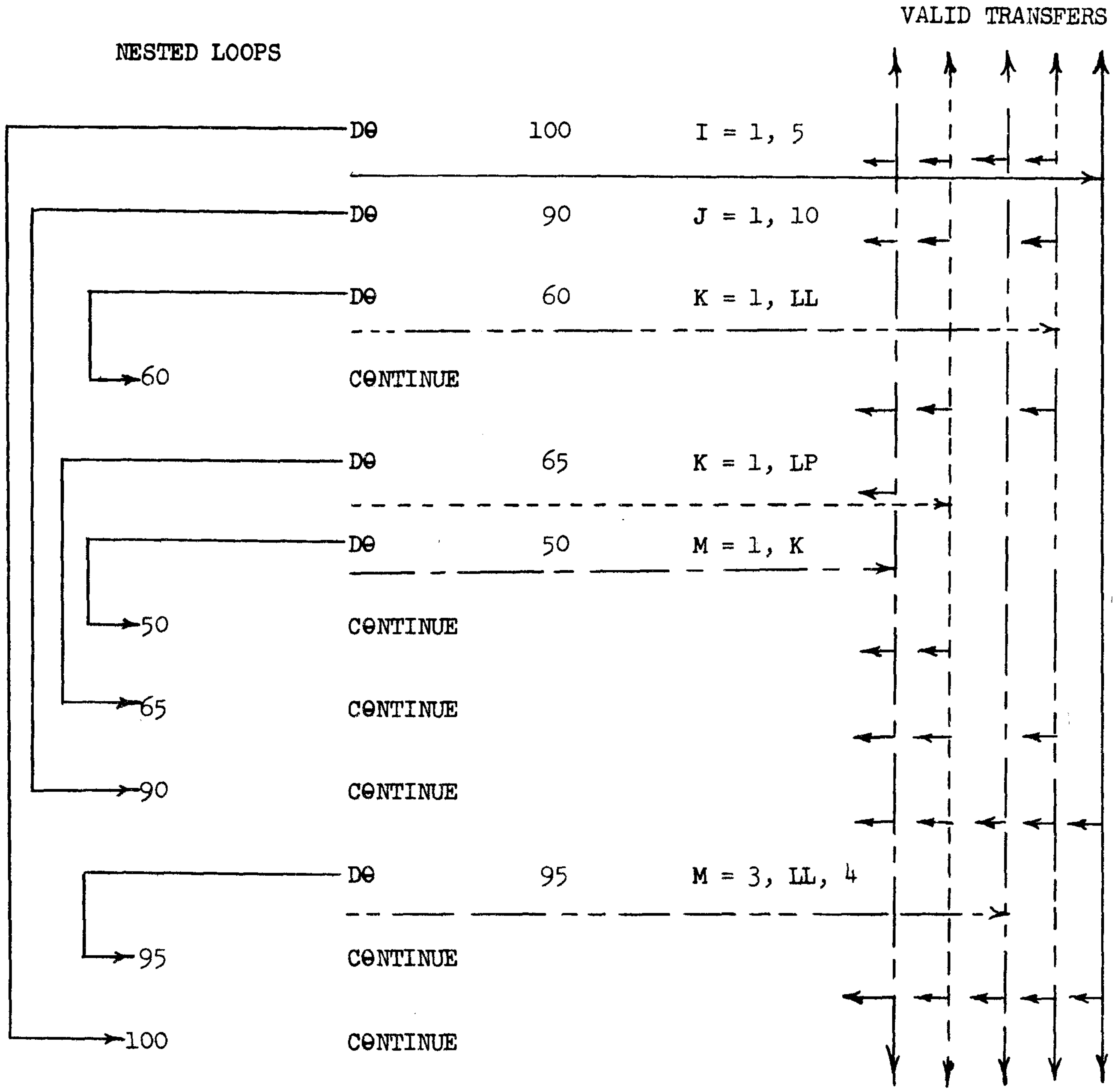
$$z_i = \sum_{j=1}^{15} (p_{i,j}) * (q_{i,j})$$

To calculate this, we must "nest" a DØ loop for calculating z within the loop for calculating x as follows

```
          DIMENSIØN Ƶ(10), P(10,15) Q(10,15), Y(10)
          X = 0.0
          DØ 100 I = 1, 10
          Ƶ(I) = 0.0
          DØ  50  J = 1, 15
     50   Ƶ(I) = Ƶ(I) + P(I, J)*Q(I,J)
    100   X = X + Y(I)**2 + 17.3*Ƶ(I)
```

Note that the inner loop on J lies <u>entirely</u> within the outer loop on I.

Although details of transferring will be covered in the next section, the following example will show valid transfers from nested DØ loops.



```
      DØ    100    I = 1, 5
      DØ     90    J = 1, 10
      DØ     60    K = 1, LL
60    CØNTINUE
      DØ     65    K = 1, LP
      DØ     50    M = 1, K
50    CØNTINUE
65    CØNTINUE
90    CØNTINUE
      DØ     95    M = 3, LL, 4
95    CØNTINUE
100   CØNTINUE
```

Vertical arrows indicate that transfer to anyplace in that direction is permitted except to <u>within</u> the range of a DØ <u>not</u> containing the DØ loop being left.

Note that the same indices may <u>not</u> be used on loops within loops, but that the limits of an inner DØ may be the same as the index of an outer DØ (i.e. DØ 50 M = 1, <u>K</u> where K is the index of an outer loop).

Note also that the nested DΘ's had to lie entirely within their outer loops. Note that the index K was used twice within the loop DΘ 90 but it was not permitted to be used in the loop DΘ 65 unless the loop DΘ 60 had already ended (as it did).

Several nested DΘ loops may end on the same statement. This may be illustrated in calculating the product of two 15 by 15 matrices.

Given

$$c_{ij} = \sum_{K=1}^{15} a_{i,k} \, b_{k,j} \qquad i,j = 1,2, \ldots 15$$

```
      DIMENSIΘN  C(15,15), A(15,15), B(15,15)
      DΘ  10  I = 1, 15
      DΘ  10  J = 1, 15
      C(I,J) = 0.0
      DΘ  10  K = 1, 15
10    C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

TRANSFER OF CONTROL

Now that you know that you are not permitted to transfer to within the range of a DΘ loop but must enter through the DΘ statement itself, and that you are permitted to transfer out of a DΘ loop, let's get into the mechanics of how to transfer. Let's say that you have just reached a point in your program at which you wish to go to statement number 105 to continue the calculation. One way of doing this is to use an "unconditional GΘ TΘ" which says merely

```
GΘ TΘ 105
```

or whatever statement number you wish to go to. When the above statement is encountered within your program, control will be transferred to statement number 105 and all statements between 105 and the "GØ TØ 105" statement will be skipped. Unconditional GØ TØ's may be transfers either forward or backward within a program.

A second type of transfer statement is the "conditional GØ TØ" of the form

```
GØ TØ IMY3
```

where the value of IMY3 has previously been specified by an ASSIGN statement which could say

```
ASSIGN 105 TØ IMY3
```

and the net result would be to transfer to statement number 105 from the "GØ TØ IMY3" statement.

A third form of transfer is to use the "assigned GØ TØ" which requires a previously defined ASSIGN statement as did the "conditional GØ TØ"; the difference being that in the "assigned GØ TØ", the statement numbers permitted for transfers are defined by the GØ TØ statement as well as assigned by an ASSIGN statement.

```
GO TO IMY3, (105, 1020, 2350, 5003)
```

is an example of an assigned GØ TØ. The number of statements one may transfer to in a conditional GØ TØ is virtually unlimited while that for the assigned GØ TØ is quite limited, even though <u>both</u> the assigned <u>and</u> the conditional GØ TØ require ASSIGN statements to specify the actual statement number to be transferred to.

The conditional GØ TØ should <u>not</u> be used unless it is essential, since the FORTRAN compiler cannot efficiently optimize the program if a conditional GΘ TΘ appears in the program. A further, although minor, reason for not using the conditional GΘ TΘ is that one may inadvertantly assign a statement number which does not exist in the program. The assigned GΘ TΘ would realize this since that statement number would not appear on the right side of the GΘ TΘ M, (I1, I2, I3, ..., IN) statement (as I1, I2, I3, ... or IN) and execution would be stopped; the conditional GΘ TΘ, however, would not catch the error and control *would* be tranferrred to *somewhere* (probably outside your program) and likely very bad things will occur (such as looping and not being able to get out, destroying your data or program, or destroying the resident which controls the computer's overall operation). In any event it is not a happy occurence and should be avoided at almost all costs.

Note that the statement

ASSIGN 10 TΘ JX

is *not* the same as the statement

JX = 10

since the ASSIGN statement is used *only* to assign statement numbers to variables which will later be used by control statements. The execution of the ASSIGN statement in this case presets to 10 the destination of all control statements pertaining to JX.

There is one last type of GΘ TΘ statement, probably the most frequently used (with the exception of the unconditional GΘ TΘ -- eg. GΘ TΘ 20) and that is the "computed GΘ TΘ" which says

GΘ TΘ (I1, I2, I3, ...), N

where N is a positive non-subscripted integer variable and I1, I2, ... are statement numbers to which the transfer is to be made. If N = 1, control is transferred to statement number J1, etc.

For example,

```
GΘ TΘ (100, 405, 160, 130, 190, 225), KX3
```

if KX3 = 1, control is transferred to statement 100; if KX3 = 6, control is transferred to statement 225; if KX3 = 7 you are in trouble (as with the conditional GΘ TΘ KY where you may have said ASSIGN 109 to KY and there is no 109 in the program) since you can only transfer to one of six locations in GΘ TΘ (100, 405, 160, 130, 190, 225), KX3 and you picked the seventh.

The most commonly used transfer statements are the computed GΘ TΘ and the unconditional GΘ TΘ.

We can now reexamine permitted transfers in nested DΘ loops using the transfer statements described above.

PERMITTED TRANSFERS WITHIN NESTED DO LOOPS

```
 30       CΘNTINUE
 33       DΘ     100          I = 1, 5
 35       GΘ TΘ  (30, 40, 92, 93, 98, 100, 110), IKY
 40       DΘ     90           J = 1, 10
 45       GΘ TΘ  (30, 35, 53, 66, 70, 90, 92, 93, 98, 100, 110), ILY
 53       DΘ     60           K = 1, 11
 55       GΘ TΘ  (30, 35, 45, 60, 66, 70, 90, 92, 93, 98, 100, 110), IMY
 60       CΘNTINUE
 66       DΘ     65           K = 1, LP
 67      *GΘ TΘ INY,  (30, 35, 45, 53, 68, 55, 65, 70, 90, 92, 93, 98, 100, 110)
 68       DΘ     50           M = 1, K
 48       GΘ TΘ  (30, 35, 45, 53, 67, 50, 55, 65, 70, 90, 92, 93, 98, 100, 110), I
 50       CΘNTINUE
 55      *GO TO IPY, (30, 35, 45, 53, 66, 67, 65, 70, 90, 92, 93, 98, 100, 110)
 65       CΘNTINUE
 70      *GΘ TΘ IQY, (30, 35, 45, 53, 66, 90, 92, 93, 98, 100, 110)
 90       CΘNTINUE
 92       GΘ TΘ (30, 35, 40, 93, 98, 100, 110), IRY
 93       DΘ     95           M = 3, LL, 4
 94       GΘ TΘ (30, 35, 40, 92, 93, 100, 110), ISY
 95       CΘNTINUE
 98       GΘ TΘ  (30, 35, 40, 92, 93, 100, 110), ITY
100       CΘNTINUE
110
```

* THESE STATEMENTS MUST BE PRECEEDED BY APPROPRIATE ASSIGN STATEMENT

There is another type of transfer statement which is usually associated with a test; that is the "arithmetic IF" statement. The arithmetic IF statement, used to test the value of an arithmetic expression, is of the form

```
IF (EXPRESSIØN) J, K, L
```

where J, K, and L are FØRTRAN statement numbers. If the value of EXPRESSIØN is negative control is transferred to statement J, if EXPRESSIØN is zero to statement K, and if EXPRESSIØN is positive to statement L. J, K, and L may be numbers corresponding to statement numbers, or names which have been previously defined as specific statement numbers by ASSIGN statements. The arithmetic EXPRESSIØN may be any expression involving arithmetic operators (+, -, *, /, and **), arithmetic built in or library functions (to be discussed later), and arithmetic variables or constants (complex numbers are not permitted, but integer, real, and double precision expressions are permitted). An example would be

```
IF (X**2 - 4.3*B/C) 101, 111, 140
```

If the value of $x^2 - \frac{4.3b}{c}$ is less than zero control will be transferred to statement number 101, if it is equal to zero control will be transferred to statement number 111, and if it is greater than zero control will be transferred to statement 140. Another way of looking at the above statement is that if $\frac{4.3b}{c} > x^2$ control will be transferred to statement number 101, if $x^2 = \frac{4.3b}{c}$ control will be transferred to statement number 111, and if $x^2 > \frac{4.3b}{c}$ control will be transferred to statement number 140.

Another example would be to calculate the sum of all positive numbers (called SUM) in an array of 20 numbers (called A). If a zero or negative number is encountered it must not be a part of the sum. All negative A's must be printed out with their location in the array (eg first, fifth, seventeenth, etc. number in the array). Assume the 20 values of A are already stored in memory. One program which would accomplish this task is

```
        DIMENSIΘN  A(20)
        SUM = 0.0
        DΘ  100  I = 1, 20
        IF  (A(I))  10, 100, 20
   10   PRINT OUT THE VALUES OF I AND A(I)*
        GΘ TΘ 100
   20   SUM = SUM + A(I)
  100   CΘNTINUE
```

Note in this example that the CΘNTINUE statement was required since if A is zero we do not want to print it out *or* add it to SUM but go to the end of the loop and calculate the next A. Note also that if A is negative we must go to statement 10 where we print out I, the position of that A in the array, and the value of A. After printing this information we must use the statement GΘ TΘ 100 so we do not add the negative A's to SUM.

A second type of IF statement is a "logical IF" which is of the form

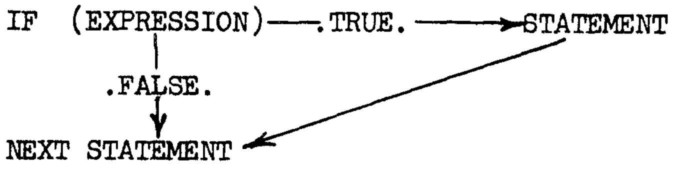
IF (EXPRESSIΘN) STATEMENT

where EXPRESSIΘN is a "logical" expression and STATEMENT is *any* FORTRAN statement *except* another logical IF statement or a DΘ statement.

If the logical EXPRESSIΘN is .TRUE., the STATEMENT will be executed and control will then pass to the next statement. If the logical EXPRESSIΘN is .FALSE., the STATEMENT will *not* be executed but control will pass directly

* more about how to do this later

to the next statement as is indicated below



An example would be: if you have an array, A, of 100 numbers ranging in value from more than 0. to less than 5000. and you wanted the actual minimum value of A, AMIN, and the actual maximum value of A, AMAX, it could be done as follows:

```
         DIMENSIØN   A(100)
         AMAX = 0.0
         AMIN = 5000.
         DØ 10   I = 1, 100
         IF (A(I) .LT. AMIN)     AMIN = A(I)
         IF (A(I) .GT. AMAX)     AMAX = A(I)
    10   CØNTINUE
```

Note, first AMAX and AMIN are initialized respectively to the smallest and largest "potential" values of A in the array. Then a DØ loop is set up to test all values of A in the array. The first logical IF statement tests to see if the current value of A(I) is less than the minimum value thus far calculated. If it is true that the current value of A is less than the smallest value yet examined, the IF statement is .TRUE. and the arithmetic statement AMIN = A(I) is executed (which stored the value of A(I) in AMIN). Control then goes to the next logical IF statement which tests for the maximum value of A in the same manner (When will both logical IF statements be .TRUE.?, when will both be .FALSE.?). After both IF statements are completed (and either, both, or neither of the arithmetic statements are executed) control goes to the CØNTINUE statement which passes back to the DØ loop starting point, increments the index by 1, and goes through the loop again.

Notice that one may compare arithmetic variables in a logical IF provided that logical operators are used. One may NOT say

1) A(I) .EQ. B(I) this should be A(I) = B(I)

2) IF(A(I) .EQ. B(I)) 100, 110, 120 this should be IF(A(I) - B(I)) 100, 110, 120

3) IF(A(I) = B(I)) X1 = X1 + 1 this should be IF(A(I) .EQ. B(I)) X1 = X1 + 1

4) IF(A(I) - B(I)) X1 = X1 + 1 this should be IF(A(I) .LT. B(I)) X1 = X1 + 1

since in the first instance you are trying to perform an arithmetic calculation with a logical operator, in the second case you are trying to perform an arithmetic test using a logical operator, in the third case you are trying to peform a logical test using an arithmetic operator, and in the fourth case you are again performing an arithmetic operation in a logical test but the answer is a number and not .TRUE. or .FALSE..

The following logical IF is permitted

```
IF (X*Y**2 .GT. 37.*Y-9.*X)  X = Y**2
```

since the result is either .TRUE. or .FALSE. depending upon the value of xy^2 compared to 37y-9X.

Examining arithmetic and logical operators, the order in which the computer will evaluate expressions containing these operators is as follows:

EVALUATED FIRST	**	arithmetic exponentiation
	* or /	arithmetic multiplication or division
	+ or -	arithmetic addition or subtraction
	.LT., .LE., .EQ., .NE., .GT., or .GE.	relational operators
	.NOT.	logical operator
	.AND.	logical operator
EVALUATED LAST	.OR.	logical operator

Inclusion of parentheses within an expression will override the "built-in" order given above. Thus the following IF statement

```
      IF (IX .GT. 9 .AND. IY .LE. 1)  GØ TØ 90
80    IX = IX + 3
      GØ TØ 100
90    IX = IX - 3
100   CØNTINUE
```

will go to 90 if and only if $ix > 9$ <u>and</u> $iy \leq 1$, otherwise control will pass to statement number 80.

The statement

```
IF (IX**2 .GT. 2*K .AND. MI .LE. MZ .ØR. .NOT. (IX .LT. MI))  GØ TØ 37
```

is a valid expression which will result in transfer to 37 if $ix \geq mi$ or if $(ix)^2 > 2(k)$ <u>and</u> $mi \leq mz$. The expression will be evaluated as if it were written as

```
IF (((IX**2 .GT. 2*K) .AND. (MI .LE. MZ)) .OR. (.NOT. (IX .LT. MI)))  GØ TØ 37
```

SECTION IV INPUT/OUTPUT

The previous sections have covered data representation, arithmetic statements, variables and arrays, looping, transfers, and arithmetic and logical tests. All of this is fine, and it is all an essential part of FØRTRAN programming, but once the computer has solved the desired problem it doesn't help you to know that the computer knows the answer -- you want the answer yourself. Obviously, to obtain the answer you will have to have the computer tell it to you. This is normally done with a "formatted WRITE" statement. What is a "formatted WRITE" statement? This might best be answered by first telling you what a "non-formatted" or "binary WRITE" statement is. If you recall, a computer "word" is a collection of 36 binary bits which represent a number, alphabetic characters, or symbols. To WRITE such a word out on magnetic tape one merely says

```
WRITE (N) WØRD
```

where N represents the number of the "logical unit" on which the word is to be written. On the UNIVAC 1108 this could be any number from zero through about 29 (where 0 represents the typewriter at the console, 5 is the card reader where most input is received, 6 is the printer where most output is written, and the remaining numbers represent magnetic tape units A - H or magnetic drum storage - see the table on the next page for specific logical unit assignments on the UNIVAC 1108). What the above binary WRITE statement does is copy on logical unit N the binary word (or bits) stored in the location in memory assigned to WØRD.

The binary WRITE is the most commonly used one (and the fastest) for input and output to and from the computer--for libraries, programs, records, etc., but it is not normally used to communicate with people since people

UNIVAC 1108

FORTRAN I/O TABLE ASSIGNMENTS - NTAB$

Logical Unit	Assignment
0	KTYPE$ (Typewriter)
1	Tape A
2	Tape E
3	Tape B
4	Tape F
5	Card Reader
6	Printer
7	Tape D
8	Tape H
9	Tape A
10	Tape E
11	Tape B
12	Tape F
13	Tape C
14	Tape G
15	Tape D
16	Tape H
17 through 24 are the same as 9 through 16	
25	Drum File 2,400,000 - (5,000,000-1)
26	" " " 400,000 - (2,400,000-1)
27	" " 2,400,000 - (3,400,000-1)
28	" " 3,400,000 - 4,400,000-1
29	" " 4,400,000 - 5,000,000-1

seem to dislike translating to and from binary. However, this method is used for all records that the program generates and uses as well as for temporary storage of data during the program execution.

For output records that are going to be used by people, the above-mentioned "formatted WRITE" would be used. This statement is of the form

WRITE (N, NF) WΘRD

Where N, as before, represents the logical unit on which the output will be written and NF is the number corresponding to the FΘRMAT under which the data (stored in the memory location assigned to WΘRD) will be written. In a formatted write, the data stored in WΘRD is not merely spewed forth and placed on tape, drum, or the printer as in a binary WRITE, but it is first "converted" to the mode specified by the FΘRMAT. There are about eight different modes or FΘRMAT types possible--they are: integer, real, exponential, double precision, G (a choice of real or exponential), octal, logical, and alphameric. Each of these modes is represented in a FΘRMAT as follows:

TYPE	REPRESENTATION	EXAMPLE
INTEGER	Iw	I14
REAL - FIXED POINT	Fw.d	F10.2
REAL - EXPONENTIAL-FLOATING POINT	Ew.d	E11.4
DOUBLE PRECISION	Dw.d	D19.8
E or F = G	Gw.d	G17.3
OCTAL	Θw	Θ13
LOGICAL	Lw	L6
ALPHAMERIC	Aw	A4

In each of the above examples, w represents the "field" width or the number of characters or consecutive output locations assigned to the word. Those FØRMATS having a w.d say there are w locations for that word and of them d are to the right of the decimal point. The decimal point itself is considered one of the locations. Thus the F10.2 could be a number like 6.23, 2100000.00, 1.03, 0.07, etc.

Since integers, octal, logical, and alphameric output have no decimal point as part of the word, none is provided for in the corresponding format type and they have only a w signifying the output field length.

The E, D, and G formats produce output of the type 0.12763E-19 where 0. and E± are provided by the computer and all numbers are represented as factors with an exponent. If a number is to be written as E10.4 the last four positions of the word will be used for the exponent designation E±XX, the first two for 0., and the remaining four positions will contain the number. Thus one requires ten positions to write four significant figures in E, D, and (sometimes) G formats, (also, if the number is negative, there will be no room for the - sign). On some computers the E or D is not printed but nevertheless a space is still provided for it. It is thus impossible to write E10.5 since one needs 11 positions for such a number (12 if the number itself is negative) and the output field width is only 10 characters long.

The G format is one in which the number is written as a fixed point number (without the exponent) if it will fit within the given field; if it won't fit as a fixed point number it will be written as an exponential. For example, the number 913762.4 would appear as 913762.4 in a G10.1 format but as 0.9138E+06 in a G10.4 format. Note that the number was rounded in the last significant place. Recall that in the computer integers are truncated, but for output all numbers are rounded.

Octal words are normally each 12 characters long (recall that a word is 36 binary bits, with each octal character equivalent to 3 binary bits) and an Θ13 or Θ14 will result in one or two blank spaces preceeding the 12 character octal number.

Logical words are represented as a T or an F in the right-most position in the field on the UNIVAC 1107. On some computers, the entire word TRUE or FALSE is printed out.

Alphameric words are alphabetic characters, symbols, or numbers. Each "computer word" (36 bits of binary information) could represent up to 6 alphameric characters (see the table on Page 14 of the first section). Thus the largest alphameric format usually is A6.

The way that the above formats are used is in a FΘRMAT statement. Recall how in the "formatted WRITE"

WRITE (N, NF) WΘRD

we said NF was the number corresponding to the format under which the data will be written. Using one of the newly learned formats we can now construct a FΘRMAT statement for the above WRITE statement as follows

NF FΘRMAT (FS)

where NF is the same number used as NF in the WRITE statement and FS is the format specification (one or more of the format types discussed above). If we wanted to write out WΘRD as an octal number FS could be Θ12, if WΘRD is a logical word we could have FS be L8. Likewise if WΘRD was merely a number FS could be F6.2, E10.3, G9.1, etc. A typical example would be

```
        WRITE (6, 45) IX, M, ZILCH
    45  FORMAT (I10, I4, F6.2)
```

in which we write on logical unit 6 (the printer--where most output is written on the UNIVAC 1107) according to FØRMAT number 45 the two integers IX and M and the real number ZILCH as a fixed point number. FØRMAT number 45 says that the first word written (IX) will be an integer number of up to ten characters, the second word (M) will also be an integer but it will have a maximum of 4 characters and the third word (ZILCH) will be a fixed point number of up to six characters with two of them to the right of the decimal point and up to three characters to the left of the decimal point (the decimal point itself took the sixth position). Note that each word specification in the FØRMAT statement is separated from its neighbor by commas. If fewer significant characters exist than the format calls for, the left side of the word will be filled in with blanks, for example, if IX was 136 it would appear as

```
        ,,,,,,,136
```

on the output page where , represents a blank space. If, on the other hand, ZILCH was a number like 3924.13 it is too big for an F6.2 format, so in the place of the number ZILCH, six *'s (on the UNIVAC 1108) would appear on the output page to signify that the word size exceeded the format specification.

The exact same FØRMAT as above could be used for many different WRITE statements, for instance one could also say

```
        WRITE (6, 45) NV, L
        WRITE (6, 45) I, I1, PER3, M, M1, PER4, N
```

using the exact same FØRMAT (number 45). Impossible you say! The format calls for three words and the above examples are writing out two and seven words respectively. Well, that's true, but output (and input) is determined by both the FΘRMAT and the call list (NV and L or I, I1, PER3, M, M1, PER4, and N in the examples above) and if the call list specified fewer words than the format list contains, only the words specified by the WRITE statement will be written. Likewise, if there are more words in the call list of the WRITE statement than the FΘRMAT specified, the FΘRMAT is started over again after it is finished until all words have been written. Thus in the last example, I is written as I10, I1 as I4, PER3 as F6.3, and now we are out of format but there are more words in the list so the format is restarted and M is written as I10, M1 as I4, PER4 as F6.3 and once again we must restart the format so that N is an I10.

On the 1108 all formatted output is written in blocks of 132 characters (22 words) since that is how many characters fit on a printed line. What actually happens is that everything to be printed out except binary output is placed in an output buffer in octal representation (regardless of whether you specified an integer, logical, alphameric, etc. word). The following table shows the octal code corresponding to each character (and the code punched on cards) for the UNIVAC 1108.

Character	Octal Code	Card Code
@	00	7-8
[	01	12-5-8
]	02	11-5-8
#	03	12-7-8
△	04	11-7-8
Space	05	Blank
A	06	12-1
B	07	12-2
C	10	12-3
D	11	12-4
E	12	12-5
F	13	12-6
G	14	12-7
H	15	12-8
I	16	12-9
J	17	11-1
K	20	11-2
L	21	11-3
M	22	11-4
N	23	11-5
Θ	24	11-6
P	25	11-7
Q	26	11-8
R	27	11-9
S	30	0-2
T	31	0-3
U	32	0-4
V	33	0-5
W	34	0-6
X	35	0-7
Y	36	0-8
Z	37	0-9

Character	Octal Code	Card Code	Character	Octal Code	Card Code
0	60	0	)	40	12-4-8
1	61	1	-	41	11
2	62	2	+	42	12
3	63	3	<	43	12-6-8
4	64	4	=	44	3-8
5	65	5	>	45	6-8
6	66	6	&	46	2-8
7	67	7	$	47	11-3-8
8	70	8	*	50	11-4-8
9	71	9	(	51	0-4-8
'	72	4-8	%	52	0-5-8
;	73	11-6-8	:	53	5-8
/	74	0-1	?	54	12-0
.	75	12-3-8	!	55	11-0
⌗	76	0-7-8	,	56	0-3-8
Idle	77	0-2-8	\	57	0-6-8

Thus if we had

```
          IX = 632

          IY = 7

          FMIX = 119.96

          WRITE (6, 10) IX, IY, FMIX

       10 FØRMAT (I6, I6, E10.4)
```

the output buffer would contain in octal

$$\overbrace{050505666362}^{IX = 632}\overbrace{050505050567}^{IY = 7}\overbrace{60756162606012426063}^{FMIX = 0.1200E+03}05050505\ldots$$

since, from the above table, 05_8 is a blank, 66_8 is a 6_{10}, 63_8 is a 3_{10}, 62_8 is a 2_{10} etc. and 119.96 is rounded to 120.0 if only four significant figures are used; the above output would appear as

```
,,,632,,,,,70.1200E+03,,,,,,...
```

Note that if FMIX were -119.96 we would have gotten

```
,,,632,,,,,70.1200E+03,,,,,...
```

since if a - sign overrides the field specification the number will be printed in the specified field without the - sign. For this reason it is advisable to always be sure that a - sign is provided for. Thus, for instance, a F20.1 format is much preferred over a 15X F5.1 format to insure adequate room for all significant figures (the numbers will appear the same on the output page.

There are 22 six character words set up in the output buffer for each formatted line of output.

In the above example, instead of writing

```
          10 FØRMAT (I6, I6, E10.4)
```

we could have written

```
          10 FØRMAT (2I6, E10.4)
```

which would have accomplished the same thing. 2I6 says there are two consecutive integer words, each of six characters.

It is also permitted to have repetition of groups of words such as

```
18 FØRMAT (I4, 2F10.6, 3(F4.2, 2I9, A3), I6)
```

This FØRMAT says there is one I4, two F10.6, three sets of (one F4.2, two I9, and one A3) followed by an I6. It could have been written as

```
18   FØRMAT (I4, F10.6, F10.6, F4.2, I9, I9, A3, F4.2, I9, I9,
   1 A3, F4.2, I9, I9, A3, I6)
```

but this is obviously much less convenient.

Recall that earlier we said when a FØRMAT is used up and more words appear in the WRITE call list the FØRMAT is repeated--well that is not strictly true, what really happens is that control goes back to the next open left parenthesis and repeats the format from there. For example,

```
     WRITE (6, 25) B, ALE, IM, IX, I2, MM, MN, MZ
25   FØRMAT (F10.6, E12.4, 2(I1, I3))
                            ↑
```

the words IM, IX, I2, and MM complete the FØRMAT so that it must start again for MN and MZ but it goes back to and starts at the (inner) parenthesis labeled with an arrow.

Reading is exactly the same as writing; except that instead of transferring the data from the computer memory to tape, cards, or the printer, it is transferred from tape or cards to memory; a binary read is as follows

```
READ (N) LIST
```

where, again, N is the logical tape unit from which reading is to take place and LIST in this case is where the data to be read is stored in memory (in the location assigned to LIST).

A formatted READ is as follows

```
     READ (N, NF) WØRD
NF   FØRMAT (FS)
```

where we read from logical unit N according to format number NF the variable WØRD. The variable appears on logical unit N as a word of format specification FS.

For example, if the first six locations of a card contain the integer 196 (right adjusted so that the 1 is in column 4, the 9 in column 5, and the 6 in column 6 of the card) it could be read in and stored in the location assigned to IB2 by

```
       READ (5, 20) IB2
20     FØRMAT (I6)
```

Note that the integer 196 had to be right adjusted; if it had appeared on the card as ,,196, the number stored in IB2 would have been 1960 and not 196 since the computer "assumes" that all blanks encountered in reading are really zero's. (Actually minus 0 on the UNIVAC 1107 but the result would still have been 1960 and not 196 as desired.)

Input E and F fields also may appear as input but here the decimal may be considered to be "built-in". If you recall, 196.2 could not have been written under an F4.1 format since this provides for one character to the right of the decimal and only two to the left (the fourth character belonging to the decimal itself). However 196.2 could have been read from a card as 1962 under an F4.1 format since this format says one character is to the right of the decimal (the 2) and there are three other characters in the field (196). Thus there is a "built-in" decimal between the 6 and the 2. Believe it or not, the number 0.031 could also be read in by the F4.1 by writing on the card .031 where, in this case, by including your own decimal you "override" the built-in decimal.

You could not read, however, an A8 word since, as you recall, A says alphameric and each alphameric character requires 6 binary bits; in a 36 bit word you can only fit 36/6 or 6 alphameric characters so A6 is the largest A format permitted for input. One word of caution here--all data read in as alphameric (A format) should be stored as integer numbers or constants to prevent loss of significant digits in any testing of these data.

An input data card contains 80 columns so each input FØRMAT and corresponding READ statement can read up to 80 columns per card. If you were to

```
      READ (5, 10) A, B, C
      READ (5, 20) K, ZM
10    FØRMAT (3F10.6)
20    FØRMAT (I1, F9.2)
```

what would happen is that the first card in the card reader (logical unit 5) will have its first ten characters "transferred" to the memory location assigned to A, the next ten characters (column 11-20 of the card) "transferred" to the memory location assigned to B, and the characters in columns 21-30 will be "transferred" to the memory location assigned to the word C. (The data is not really "transferred" since it still remains on the cards but it is converted to the proper mode (integer, real, alphameric, logical, etc.) and stored.)

When the next READ statement is encountered, the NEXT CARD is used and the contents of column 1 of the second card are "transferred" to the memory location corresponding to the word K. The contents of columns 2-10 of the second card are then stored in the memory location assigned to ZM.

Note that when a new READ is encountered, it automatically starts with the next card in the card reader (or next record on tape, but more about records later). Had the first format been

```
        10    FΘRMAT (2F10.6)
```

after B was read in and the format restarted, the next card would have been read in columns 1-10 for C. Thus when reading in data, if a format is restarted the next card is read even though only one READ statement may be involved.

Now that we have seen how to read and write single variables at a time, the next thing we must discuss is input and output of entire arrays. One way to write out an array is as follows

```
              DIMENSIΘN A(5)
              WRITE (6, 10) A(1), A(2), A(3), A(4), A(5)
        10    FΘRMAT (5F10.3)
```

As you can see if this method is used and if you have several thousand or even several hundred elements to write, you will have to write a rather large WRITE statement. Therefore a simpler technique has been made available and that is to use an "implied DΘ" loop as follows

```
              WRITE (6, 10)(A(I), I = 1, 5)
```

This WRITE statement says, in effect, "write out on logical unit 6, using FΘRMAT 10, the array A(I) where I takes on the values 1-5 successively. This technique may be used to write out arrays having more than one subscript as well as portions of arrays.

An even greater simplification is possible; writing out the above array could also have been accomplished as follows

```
              WRITE (6, 10) A
```

In this instance the computer "knows" that A is a subscripted variable since it appears in a DIMENSION statement. Thus, since the computer was not told which A to write out it is clever enough to write out all five A's.

An example of a perfectly legitimate albeit somewhat complicated WRITE statement appears below

```
          DIMENSION A(100), B(2, 3, 4), C(50, 10), D(50)
          WRITE (6, 20) (A(I), I = 11, 30), B, ((C(I, J), J = 1, 10),
        1   D(I), I = 1, 10)
     20   FORMAT (20F5.0/24F5.1/ (11F10.2/))
```

Notice in the above WRITE statement that each array that is not written out as a complete array is written out using an index. An array and its index must contain parenthesis to set off that particular output grouping of the array. What happens in the above write statement is that elements 11-30 of the array A are first written out, then the entire array B is written out. The order in which B is written out is B(1,1,1), B(2,1,1), B(1,2,1), B(2,2,1), B(1,3,1),...,B(2,3,4). Whenever a multi-subscripted array is written out, the left-most subscript is varied most frequently. This is also the order in which the array is stored in memory. Note also that the arrays C and D are written out together. The order in which these are written out is C(1,1), C(1,2), C(1,3),...,C(1,10), D(1), C(2,1), C(2,2), C(2,3),...,D(2), C(3,1),... ,D(3), C(4,1),...,C(10,10), D(10). Note here that by including the index specifications in the WRITE statement you automatically override the order in which the computer naturally would have printed out the array C. Note also that by judicious use of subscripting it possible to intermingle several different arrays.

Something new has suddenly appeared in the FØRMAT, that is the / (or slash). The purpose of the slash is to tell the computer to start a new line of output. The use of n successive slashes will result in n-1 blank (skipped) lines appearing on your output page. Thus elements 11-30 of the array A were written out on the first line, each element having an F5.0 format. The array B was written out on the next line, each element having an F5.1 format. The arrays C and D were written out on the following 10 lines, each line corresponding to a different value of the subscript I. Note that since the (11F10.2/) appeared within an inner set of parentheses, only <u>this</u> format group was repeated for each line containing the arrays C and D. The output generated by the above WRITE and FØRMAT statements would appear as follows

```
A(11)     A(12)     A(13)     A(14)     A(15) ... A(30)
B(1,1,1)  B(2,1,1)  B(1,2,1)  B(2,2,1)  B(1,3,1) ... B(2,3,4)
C(1,1)    C(1,2)    C(1,3)    C(1,4)    C(1,5)  ... C(1,10)  D(1)
C(2,1)    C(2,2)    C(2,3)    C(2,4)    C(2,5)  ... C(2,10)  D(2)
.
.
.
C(10,1)   C(10,2)   C(10,3)   C(10,4)   C(10,5) ... C(10,10) D(10)
```

Before we get into any more specific examples, recall that an E10.4 format has the form 0.XXXXE±YY. If we desire to replace the 0. by a number we could accomplish this by using a "scale factor". For instance if Y = 6239.1 and we had

```
        WRITE (6, 27) Y
 27     FØRMAT (E10.4)
```

we would get

```
0.6239E+04
```

However, we could have placed a 1P scale factor in the format as follows:

```
27   FØRMAT (1PE10.4)
```

and in this case we would have gotten

```
6.2391E+03
```

The nP scale factor in effect says "for all formats following, take the number stored in memory and multiply its mantissa by 10^n and subtract n from its exponent before writing it out" (or, for input, take the number being read in and multiply its mantissa by 10^{-n} and add n to its exponent before storing it in memory). In effect what you are doing is shifting the decimal n places (to the right on output and to the left on input). Thus the external representations of the number (on cards, tape, printer, etc.) equals the internal representation in memory times 10^n.

Several facts must be stated about use of the scale factor before we use it indiscriminantly. One of these is that it affects numbers written in F formats as well as those in E formats. The problem here is that F formats have no exponent to adjust so if we have

```
  ZZ = -3.429600
  WRITE (6, 6) ZZ
6 FØRMAT (2PF10.4)
```

we would get

```
-342.9600
```

since the 2P scale factor adjusted ZZ by multiplying the mantissa by 10^2, we "see" ZZ as 100 times what it was when stored in memory. Negative scale factors are permitted, thus if we had

```
6 FØRMAT (-2PF10.4)
```

we would get FØR ZZ -0.0343

recalling that we *round* numbers on output rather than truncate them.

The other thing associated with use of scale factors is that *they apply to all number formats following the one in which they first appear*. Thus if we had

```
                   X1 = 10.658
                   X2 = 132.94000
                   X3 = 6.44970
                   WRITE (6, 10) X1, X2, X3
               10  FØRMAT (1PE16.4, 2F10.4)
```

we would get

```
          1.0658E+01         1329.4000          64.4970
```

even though the 2F10.4 format did *not* have a scale factor associated with it. To prevent such an occurrence, when we no longer want a scale factor we must "turn it off" or zero it out as follows

```
               10 FØRMAT (1PE16.4, 0P2F10.4)
```

to get

```
          1.0658E+01         132.9400           6.4497
```

Scale factors are commonly used to convert from "units" in the real world to "programmed" units. For instance, if data in the form of electrical readings in millivolts are input to a program requiring volts in the calculations it performs, they could easily be read in under a 3PF10.5 format which would take the input number in millivolts and store it as volts in memory. Likewise if microvolts were inputted to the same program they could be read as 6PF10.5 to automatically make the correct conversions. When these numbers are then written back out, the same format would convert the answer back to the input units by making the correct scale factor conversion.

There are several format fields we have not yet mentioned. One of these is a "blank field". To skip say n spaces, the format nX may be used. Thus if we have

```
           X = 1.0
           Y = 2.0
           Ƶ = 3.0
           WRITE (6, 10) X, Y, Ƶ
     10    FØRMAT (10X, 2F5.1///)
```

the output would appear as

```
           ,,,,,,,,,,,,1.0,,2.0
  ———————>
  ———————>
           ,,,,,,,,,,,,3.0
```

where , represents a blank space and ———————> represents a skipped line.

The 10X skipped the first 10 spaces and, since there was only one significant figure to the left of the decimal point for each number, only 3 characters were needed for each F5.1 field so that 2 additional blank spaces were filled in on the left side of each of the F5.1 fields. The first slash in effect said "go to the end of the line", the second slash said "go to the end of the next line", while the third slash skipped the second line or "went to the end of the third line". By then the format is used up, so control goes back to the next open left parenthesis and starts again with 10 blank spaces and writes out Ƶ as an F5.1.

Another very valuable format field is the "Hollerith" or "Alphameric" field nH which in effect says "print out the next n characters just as they appear".

An example is

```
                    10HXXXXXXXXXX
```

which would result in ten X's being printed out.

If we had

```
                    T11 = 7.35
                    WRITE (6, 23) T11
                23  FØRMAT (5X, 4HT11=,F5.2)
```

we would get

```
                    ,,,,,T11=,7.35
```

If we instead said

```
                    WRITE (6, 23)
```

with no list (T11 in this case) we would get

```
                    ,,,,,T11=
```

so you see that X and H fields are output list independent.

One last thing before we start going into more examples--the first (left-most) column on each page is the "carriage control" column to tell the printer what to do with what follows. If a 1 appears in the carriage control column, a new page will be started, a 0 will cause a line to be skipped (like 2 slashes), a blank space will cause a new line to be started, (like one slash) and a + will result in retarding the skipping of a line. Formats that start with X, I, E, F, G, etc. fields which leave the first column blank will thus result in the format printing on the next line.

As a typical example, if we wanted to read an array of 100 numbers (10 numbers on each of 10 cards with an F8.2 format) square each of them, and write out each number and its square, one per line with 40 lines per page we could do it as follows:

```
      DIMENSIØN A(100)
      READ (5, 10) A
10    FØRMAT (10F8.2)
      NPAGE = 1
      WRITE (6, 20) NPAGE
20    FØRMAT (1H1, 20X, 16HSQUARING PRØGRAM, 40X, 5HPAGE ,
    1 I3 ///   14X, 6HNUMBER, 14X, 6HSQUARE)
      NPAGE = NPAGE + 1
      DØ 50 I = 1, 100
      ASQ = A(I)*A(I)
      WRITE (6, 30) A(I), ASQ
30    FØRMAT  (2F20.4)
      IF ((I .NE. 40) .AND. (I .NE. 80))  GØ TØ 50
      WRITE (6, 20) NPAGE
      NPAGE = NPAGE + 1
50    CØNTINUE
      STØP
      END
```

Before we explain in too much detail what is happening in this program we should note that there are two new FØRTRAN statements never before encountered. One, the last one, says END; the END statement tells the compiler that the program it is now compiling has ended and what follows may be a new program, a new subroutine, or data. An END statement is required to be the last card of every subroutine or main program to be compiled.

The STΘP statement preceeding the END statement tells the <u>computer</u> that the program is finished when it reaches this point in executing the program and its execution should be terminated.

The above program is a complete FΘRTRAN program and if it is fed into a computer with a FORTRAN IV compiler and is followed by 10 cards containing data it should produce the square of the data and write them out.

We will now examine the program in more detail.

First we have our DIMENSIΘN statement which sets up the size of the array A as 100. Then we READ in the 100 values of A from logical unit 5 (the card reader) under FΘRMAT number 10. FΘRMAT 10 says there are 10 numbers per card, each F8.2 so that ten numbers per card are read and stored in the real array A for each of the 10 cards.

An alternate and valid approach would have been to read in ten numbers (one card) at a time, square them, and write them out before reading the next ten.

After the 100 values of A are read in and stored, an integer variable called NPAGE (for the page number of the output) is initialized to 1. Then a heading line is written out by FΘRMAT 20 which says "1H1 or go to a new page (by putting a 1 in the carriage control column), skip 21 spaces (1 for the carriage control column + 20 for the 20X), write out a 16 Hollerith field giving the name of the program, skip 40 spaces, write out PAGE and the page number, skip two lines (3 slashes) and write out NUMBER in columns 15-20 and SQUARE in columns 35-40.

This is what would appear on the first page:

```
,,,,,,,,,,,,,,,,,,,,SQUARING PROGRAM,....,,,,PAGE,,1
,,,,,,,,,,,,,NUMBER,,,,,,,,,,,,,,SQUARE
```

Note that we do not see the 1H1 since the first column is used for carriage control only and is not printed.

After we write out the heading, we add 1 to NPAGE so that the next time it is used it will say PAGE,,,2.

We then set up a DΘ loop to process the 100 values of A. First we calculate A squared (ASQ) and then we write out both the current A and A squared (ASQ) under FΘRMAT number 30. We then test to see if we are at the end of a page or not. If not we go to the end of the loop and calculate and write out the next A and A square. If we have processed exactly 40 or 80 values of A, we must go to a new page so we write out the heading again (FΘRMAT number 20) and increment the page counter by one.

When we have calculated and written out all 100 values of A we STΘP.

If we wanted to write the same program but have it read in <u>only one card at a time</u> and stop when it encountered a zero value of A, we could do it as follows:

```
C SQUARING PRΘGRAM
               DIMENSIΘN A(10)
               NPAGE = 1
C WRITE PAGE HEADING
               WRITE (6, 20) NPAGE
          20   FΘRMAT (1H1, 20X, 16HSQUARING,PROGRAM, 40X, 5HPAGE,, I3///
            1    14X, 6HNUMBER, 14X, 6HSQUARE)
               NPAGE = NPAGE + 1
          5    INDEX = 0
C READ INPUT DATA CARD
          8    READ (5, 10) A
```

```
     10   FΘRMAT (10F8.2)
          DΘ 50 I = 1, 10
          IF((A(I) .LT. 0.001) .AND. (-A(I) .LT. 0.001))  STΘP
C CALCULATE SQUARE AND WRITE OUT RESULTS
          ASQ = A(I)*A(I)
     50   WRITE (6, 30) A(I), ASQ
     30   FΘRMAT (2F20.4)
          INDEX = INDEX + 1
          IF(INDEX .LT. 4)  GΘ TΘ 8
C PREPARE NEW HEADING PAGE
          WRITE (6, 20) NPAGE
          NPAGE = NPAGE + 1
          GΘ TΘ 5
          END
```

Here again we have encountered a new card--the "CΘMMENT" card--with the alphabetic character C in column one and alphameric information in the remaining columns. This card is ignored by the computer (as are all cards with a C in column 1) and may be used to provide information to the programmer. The above program differs in several other ways from the previous one but it will still read in the 100 numbers and square them and write them out with their square. This program is more flexible than the previous one in that it will read in as many numbers as there are and square them whether there are 100, 10, 1, 1000, etc. they will all be processed until a zero is encountered; whereas the first program will only read in 10 cards of 10 numbers each.

Note that in this program the page initialization (NPAGE = 1) and the heading WRITE statement must be placed first (after the DIMENSIØN statement) since we loop back to the beginning and read a new card after every tenth number and we do *not* want to reinitialize the page number and go to a new page with a new heading after reading every card (each 10 values of A).

Note also that we have a new integer variable here, INDEX, which is used to signal when 40 lines have been printed on a page and a new page should be started.

We now have a DØ loop for only the 10 values of A currently in memory and we first test A to see if it is "zero" or not. If A is "zero" we stop; if not, we calculate A squared and print them out. Note that we don't *really* test to see if A = 0.000000000 since A is *not* an integer and roundoff errors may not say this is true even if a "zero" A were read in. Only *integers* should be tested for *equality* in logical tests. However, we do the same thing with

```
     IF((A(I)) .LT. 0.001) .AND. (-A(I) .LT. 0.001))  STØP
```

since A was read in as an F8.2 so *all* values of A that are *not* *zero* will be read in and stored as greater than or equal to 0.01 *or* less than or equal to -0.01 since that is the minimum size number that is consistent without built-in format. (It is true that one could override our built-in format by placing a positive number as small as .0000001 or a negative number as large as -.000001 in the F8.2 field by punching in the decimal point, but it was assumed that the F8.2 would be adhered to. If this assumption is not valid, one could test for ((A .LT. 0.0000001) .AND. (-A .LT. 0.000001)) to *assure* stopping for zero's and only zero's assuming only a 1 character maximum roundoff error.)

If A is not "zero" we proceed as before and square it and write it out. After we have processed each ten A's we test for the end of the page (INDEX = 4). If we have 30 or fewer A's on a page we read a new card (GØ TØ 8). If we have 40 A's on a page, we put a heading on a new page and reinitilize our page counter (GØ TØ 5).

If in the above program, we started the data pack with a card containing a case title which we wished reproduced as part of the page heading, and if the title was found in columns 1 - 60 of the first card we read, our program could start as follows

```
         DIMENSION A(10), NTITLE(10)
         NPAGE = 1
         READ (5, 1) NTITLE
     1   FØRMAT (10A6)
         WRITE (6, 20) NTITLE, NPAGE
    20   FØRMAT (1H1, 5X, 16HSQUARING PRØGRAM, 10X, 10A6, 10X, 5HPAGE I3///
       1 14X, 6HNUMBER, 14X, 6HSQUARE)
         .
         .
         .
```

Now whenever we write out under FØRMAT 20 we must include in the WRITE list not only the page number (NPAGE) but also the 10 word case title (NTITLE). Note that the maximum alphameric word size, A<u>6</u>, was used. We could also have written 12A5 and dimensioned NTITLE (12) but this required two more words of storage (NTITLE (11) and NTITLE (12)). If we did use the latter approach we would have to <u>both</u> <u>READ</u> <u>and</u> <u>WRITE</u> the title as 12A5. If we READ the title as 12A5, the right-most character of each word (the sixth position) will contain a blank (A formats are left-adjusted whereas F, E, G, I, L, or Ø formats are right-adjusted) so that if we then write out TITLE as 12A6 or 10A6 we would get an extra blank space printed out every sixth character and, in the latter case, only 10/12 of the title.

It is important to remember that for A formats, the unfilled portion of the field is placed on the right (the word is left-justified) whereas the reverse is true for other formats.

In the above example, we stored the case title in an array called NTITLE and this information was available in memory for use at any time. There is a second way of reading and storing alphameric data in memory and that is with the Hollerith field.

Although this approach is not commonly used, an example would be:

```
          READ (5, 20) X
      20  FØRMAT (10HAAABBBCCC=F10.2)
          X = X + 3.6
      30  WRITE (6, 20) X
```

where the card being read looks like

```
          ,,DELTA-X=,,,,103.62
```

the portion ,,DELTA-X= will be stored in 10HAAABBBCCC= and 103.62 will be stored in the memory location corresponding to the variable X. As long as nothing further is read in under FØRMAT 20, 10HAAABBBCCC= will contain the information ,,DELTA-X= and when statement 30 (WRITE (6, 20) X) is executed the output will appear as

```
          ,,DELTA-X=,,,,107.22
```

However, if after we said X = X + 3.6 we had also included

```
          READ (5, 20) Y
```

where it read

ZILCH/X+Y=,,,,439.16

and then wrote statement 30 we would have gotten out

ZILCH/X+Y=,,,,107.22

Thus, although this technique is available, the usual method for reading, storing, and writing alphameric information is with A formats and by storing data in arrays rather than in Hollerith fields.

One can WRITE or READ to or from, magnetic tapes, discs, and drums as well as the printer, card reader, and console typewriter by merely inserting the correct logical unit designation in the WRITE or READ statement. It is usually also possible to WRITE or READ formatted information to and from main memory and to "re-read" something using the software package available at your computer installation.

For instance, the 1108 operating system "knows" when a WRITE or READ referencing logical unit -4 is encountered, the user is talking about main memory or the I/Θ buffer (the intermediate storage "buffer" through which all input and output passes between memory and the I/Θ device). Thus if you read in a card and desire to reread that particular card under some other format, it may be done as follows:

```
      DIMENSION  X(13), IMY(14), XMY(7)
 10   FΘRMAT (I2, 13A6)
 20   FΘRMAT (I2, 8X, 7F10.0)
 30   FORMAT (I2, 8X, 14I4)
      KY = -4
 90   READ (5, 10) IX, X
```

```
        GØ TØ (100, 110), IX
100     READ (KY, 20) IX, XMY
        GØ TØ 120
110     READ (KY, 30) IX, IMY
120     .
        .
        .
```

Here the card is first read and the integer in columns 1 and 2 is tested to see if it = 1 (meaning the card contains 7 real variables and should be read by statement 100) or if it = 2 (meaning the card contains 14 integer variables and should be read by statement 110). The card is then "reread" by the proper statement utilizing the appropriate FØRMAT.

Another technique available, is that of storing the FØRMAT itself in an array in a DATA statement (more about DATA statements later) or reading it in just before you need it.

If you are not certain what format will be used to read in data, the format itself may be read in just before the data is. For example

```
    DIMENSION IX (12)
    READ (5, 10) IX
    READ (5, IX) X, Y
10  FØRMAT (12A6)
```

Here we first read in the format (under which we will then read X and Y) and store the format in the array called IX. The corresponding data cards could appear as follows

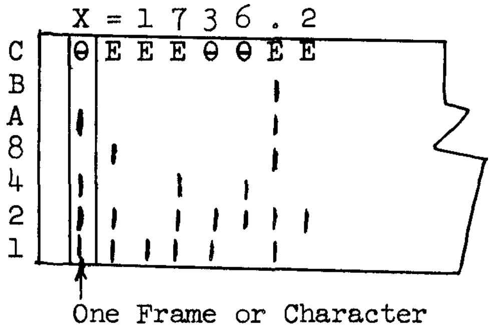
```
(F10.6, F5.0)
,,,.933216,1385
```

and X would contain 0.933216, Y = 1385.

This permits the user to choose his own format specifications each time he runs the program.

Before we leave the subject of input and output we should spend a little time talking about how the data is represented on the output medium. We already saw how input records on cards contain 80 characters for 80 columns as a maximum and how output for the printer contains a maximum of 132 columns or 22 words (each 6 characters). We also mentioned how all information is transferred as either binary or field data (Hollerith or alphameric) in the I/Θ buffer.

Magnetic tape usually contains seven "tracks" or channels. Six of these are used to record the BCD code (see the table on page 14 in section I) or the binary number and the seventh contains the "parity check" bit. For example



Where there is a 1-bit to represent the presence of a 1 in the table in section I. The parity channel contains a 1-bit if the tape is even parity and the number of 1-bit's in the other 6 channels is odd (eg. for =, 1, 7, ., and 2) it contains a 1-bit if the tape is odd parity and the number of 1 bits in the other six channels is even (eg. for X, 3, and 6). A tape is either EVEN or ODD parity but not both. Thus for even parity, only the E's would contain a 1-bit (the Θ's would be blank) and for odd parity the Θ's would contain 1-bits and the E's would be blank.

Each character or number is a FRAME on the tape and each FRAME is checked for correct parity upon reading to catch transmission errors.

A "logical record" is that which is read or written by one READ or WRITE statement. A logical record may contain many frames or characters. It may also contain many lines (at 22 words of output per line) of data or many cards but it is classed a "logical" record if it is read or written by a single statement (even if the statement contains many implied DΘ's).

A "physical" record is the way a record is grouped on tape or on the printer. Binary records are written out in blocks of 253 words (plus 2 control words and a checksum word) or less. There is a 3/4 inch gap of blank tape between each physical record. BCD or alphameric physical records are written out in blocks of 22 words since there are 22 words per line of output. Thus if you write out logical records of several hundred or thousand words each, the physical record(s) of which they consist are limited to 22 (if BCD or Binary Coded Decimal) or 253 (if binary) words, with each physical record separated from its neighbor by a 3/4 inch end of record gap (blank space) on tape.

There are two control words per binary physical record (256 words total maximum per block) as mentioned before. One control word is at the start of the block and the other is at the end of the block. The left half of the control word tells how many words are in the block and the right half of the control word contains the block number and a flag for the last block of the logical record.

The checksum says how many 1 bits were punched in that physical record.

A FILE is a group of one or more logical records concerning a particular subject on one or more tapes.

To complete the discussion of input-output, there are three more FORTRAN statements that should be discussed. These all deal with tapes, drums, or discs. They are REWIND, BACKSPACE, and END FILE.

REWIND means "position the tape at the load point or start". When your computer encounters the statement

REWIND N

it will rewind the tape on logical unit N to the starting point (marked by a piece of reflecting tape on the magnetic tape).

BACKSPACE N

means rewind logical unit N back one logical record (where a logical record was a record generated by one read or one write statement and could be composed of one or more physical records). Thus after you read or write a record, you may BACKSPACE and read or write the record again.

END FILE N

causes a logical END OF FILE to be placed on the tape on logical unit N; an END OF FILE is a gap that is about 3 inches long and it signals the end of a file. Every output tape must have an END FILE placed on it or a sentinel signalling that the file continues on another tape, so that upon reading or printing it you stop at the end of your data records and don't attempt to read or print whatever is on the tape following your data.

The usual procedure followed when using output tapes or input tapes is to first REWIND them to get them at the starting point. When output tapes are finished they are marked with an END FILE and then all tapes are rewound again so they may be removed by the computer operator.

If one is using a preselected area of disc or drum memory the REWIND instruction will usually position you at the start of that area. BACKSPACE does the same thing on drums or discs as it does on tape--it positions you at the start of the last logical record you have read or written.

SECTION V FUNCTIONS AND SUBROUTINES

One of the most important features available in FØRTRAN programming is that of being able to utilize previously written programs and routines without having to completely rewrite each one whenever it is needed in your particular program.

Many often-used mathematical functions already exist and are provided by the FØRTRAN compiler (or processor) or exist in a library. Other functions may be constructed by the programmer himself.

The simplest type of function is the BUILT-IN or INTRINSIC function which is part of the FØRTRAN processor and is automatically coded (in line) in your program by the compiler during the compilation process. There are about thirty such functions usually available in the FØRTRAN IV language. A table of the typical BUILT-IN functions appears on the next page (these are all available in the UNIVAC 1108 EXEC II FØRTRAN Processor).

Note first that the FØRTRAN function name obeys the same rules as a FØRTRAN variable name--it is limited to six characters, it must start with an alphabetic character; I, J, K, L, M, or N means the result is in the integer mode. Note also that functions are quite specialized with regard to input and output modes--special functions exist for real, integer, double precision, and complex numbers.

A typical example of the use of such an internal function is to recall an example in the last section in which we wished to test a variable, A(I), to see if it's absolute value was less than 0.001. To do this we used the logical IF statement

```
IF((A(I) .LT. 0.001) .AND. (-A(I) .LT. 0.001))    STØP
```

UNIVAC 1108 FORTRAN

FORTRAN	No. of		Mode of	
Name	Args.	Function	Argument	Function
ABS	1	Determine the absolute value of the argument.	Real	Real
IABS			Integer	Integer
DABS			D-P	D-P
AINT	1	Truncate: eliminate the fractional portion of the argument.	Real	Real
INT			Real	Integer
DINT			D-P	D-P
AMØD	2	The expression X-(X/Y)*Y is computed where X is the first and Y the second argument. (Z) denotes the integral part of Z.	Real	Real
MØD			Integer	Integer
AMAXO	≥ 2	Select the largest value.	Integer	Real
AMAX1			Real	Real
MAXO			Integer	Integer
MAX1			Real	Integer
DMAX1			D-P	D-P
AMINO	≥ 2	Select the smallest value.	Integer	Real
AMIN1			Real	Real
MINO			Integer	Integer
MIN1			Real	Integer
DMIN1			D-P	D-P
FLØAT	1	Convert from integer to real.	Integer	Real
IFIX	1	Convert from real to integer.	Real	Integer
DBLE	1	Convert from real to double-precision.	Real	D-P
CMPLX	2	Convert two real arguments to one complex number.	Real	Complex
SIGN	2	Replace the algebraic sign of the first argument by that of the second.	Real	Real
ISIGN			Integer	Integer
DSIGN			D-P	D-P
DIM	2	Positive difference: subtract the smallest of the two arguments from the first argument.	Real	Real
IDIM			Integer	Integer
SNGL	1	Obtain the most significant part of a double-precision argument.	D-P	Real

FORTRAN Name	No. of Args.	Function	Mode of Argument	Mode of Function
REAL	1	Obtain the real part of a complex argument.	Complex	Real
AIMAG	1	Obtain the imaginary part of a complex argument.	Complex	Real
CØNJG	1	Obtain the conjugate of a complex argument.	Complex	Complex

which in effect says if A(I) is less than 0.001 (all negative numbers, 0.0, and positive numbers smaller than 0.001 AND -A(I) is less than 0.001 (all positive numbers, 0.0, and negative numbers larger than 0.001) then STØP.

Had we known about BUILT-IN functions at that time we could have used the ABS function (which calculates the absolute value of its argument) and written the test as

```
IF(ABS(A(I)) .LT. 0.001)  STØP
```

In this example, the FUNCTIØN is ABS and the ARGUMENT is A(I). Both are in the real mode. The argument of BUILT-IN functions may be a variable name (as above), a constant, or any arithmetic expression (involving +, -, /, *, ** operating on arithmetic variables or constants).

For example,

```
IS = IFIX(A**C/D+7.43*(C+D)**3)-2
```

is a valid use of the IFIX function which converts its real argument (A**C/D+7.43*(C+D)**3) into integer form.

The FUNCTIØN is normally used as part of an arithmetic expression and it supplies a single valued solution to the argument(s) it is provided.

The BUILT-IN function names may not be used for variable or constant names in the same program in which they are referenced as functions; they *may* be used as variables or constants provided they *are* mentioned in a TYPE statement (see section II page 32) and are *not used as functions*. For example, the *variable* ABS could *not* be used in the program above which used the *function* ABS, but it would be perfectly valid to say

```
REAL AINT
.
.
.
AINT = ABS(A(I))
```

in the program where ABS is a BUILT-IN function and, although AINT is also on the list of BUILT-IN functions, AINT is a real variable defined by the REAL type statement (which tells the compiler AINT is *not* being used as a function but as a variable this instance). It is then *NOT* permitted to use the *function* AINT at any place in the above program.

A second type of function commonly used is the EXTERNAL or EXTRINSIC function. There are two types of EXTERNAL functions--those found in a library provided by the installation and those programmed by the programmer himself.

The table on the next page lists the EXTERNAL library functions available in the UNIVAC 1108 EXEC II FØRTRAN library.

Note again that names are limited to 6 alphameric characters and must start with an alphabetic character as for FØRTRAN variable names. Also, again, the evaluation or result of the function is a *single* answer provided in the place of the function name in an arithmetic statement.

EXTERNAL functions are *not* coded in-line (as are BUILT-IN functions) but are transferred to at the time of execution when control passes to them in a statement. As a result, execution of EXTERNAL functions is not as fast as execution of BUILT-IN functions.

UNIVAC 1108 FORTRAN

No. of			Type of	
Args.	Function Reference		Argument	Function
1	Trigonometric Sine:	SIN (X)	Real	Real
		DSIN (X)	D-P	*D-P
		CSIN (X)	Complex	*Complex
1	Trigonometric Cosine:	COS (X)	Real	Real
		DCOS (X)	D-P	*D-P
		CCOS (X)	Complex	*Complex
1	Trigonometric Tangent:	TAN (X)	Real	Real
		DTAN (X)	D-P	*D-P
		CTAN (X)	Complex	*Complex
1	Trigonometric Arcsine:	ASIN (X)	Real	Real
		DASIN (X)	D-P	*D-P
1	Trigonometric Arccosine:	ACOS (X)	Real	Real
		DACOS (X)	D-P	*D-P
1	Trigonometric Arctangent:	ATAN (X)	Real	Real
1		DATAN (X)	D-P	*D-P
2		ATAN2 (X_1,X_2)	Real	Real
2		DATAN2 (X_1,X_2)	D-P	*D-P
1	Hyperbolic Sine:	SINH (X)	Real	Real
		DSINH (X)	D-P	*D-P
		CSINH (X)	Complex	*Complex
1	Hyperbolic Cosine:	COSH (X)	Real	Real
		DCOSH (X)	D-P	*D-P
		CCOSH (X)	Complex	*Complex
1	Hyperbolic Tangent	TANH (X)	Real	Real
		DTANH (X)	D-P	*D-P
		CTANH (X)	Complex	*Complex
1	Exponential (e^X):	EXP (X)	Real	Real
		DEXP (X)	D-P	*D-P
		CEXP (X)	Complex	*Complex
1	Natural Logarithm ($\mathrm{LOG}_e{}^X$):	ALOG (X)	Real	Real
		DLOG (X)	D-P	*D-P
		CLOG (X)	Complex	*Complex

* NOTE: If the result of the function is double precision or complex the function name must be declared in a type statement.

No. of			Type of	
Args.	Function Reference		Argument	Function
1	Common Logarithm ($LOG_{10}x$):	ALOG10 (X) DLOG10 (X)	Real D-P	Real *D-P
1	Square Root $(X)^{1/2}$	SQRT (X) DSQRT (X) CSQRT (X)	Real D-P Complex	Real *D-P *Complex
1	Cube Root $(X)^{1/3}$	CBRT (X) DCBRT (X) CCBRT (X)	Real D-P Complex	Real *D-P *Complex
1	Absolute value of a complex number	CABS (X)	Complex	Real
2	The expression X_1-(X_1/X_2)* X_2 is computed, where(Z) denotes the integral part of Z.	DMOD(X_1X_2)	D-P	*D-P

* NOTE: If the result of the function is double precision or complex the function name must be declared in a type statement.

Since the FØRTRAN compiler itself does not provide the EXTERNAL function during compilation but provides only a transfer address to where the function will be located during execution, the compiler DOES NOT KNOW whether the function is CØMPLEX, DØUBLE PRECISIØN, or REAL so it assumes that it is REAL and provides for only one answer after the function is evaluated. Therefore, to avoid getting only half of the answer, EVERY TIME A DOUBLE PRECISION OR COMPLEX EXTERNAL FUNCTION IS USED, THE NAME OF THE FUNCTION MUST BE INCLUDED IN THE APPROPRIATE TYPE STATEMENT so that the compiler will provide for a two-word answer.

For example,

```
COMPLEX A, B
.
.
.
A = CSQRT(B)
```

will result in only the real part of B being placed in A even though BOTH A and B are defined as complex. To work properly, you must have

```
CØMPLEX A, B, CSQRT
.
.
.
A = CSQRT(B)
```

As for BUILT-IN functions, the argument of functions may be variables, constants, or arithmetic expressions (including other functions). The following statement is perfectly valid:

```
A = SQRT(ALØG(SIN(ABS(X))+CØS(REAL(B))))
```

where B was defined in a CØMPLEX type statement. (Note, for BUILT-IN or INTRINSIC functions, the name of the function need not be stated in a TYPE statement if it is complex or double precision since the compiler is providing the function and it knows whether the answer requires two words or not. It is only functions that are provided EXTERNAL to the compiler that must be defined in TYPE statements.)

The above statement will cause the cosine of the real part of B to be added to the sine of the absolute value of X; the square root of the log$_e$ of this sum will be stored in A.

A third type of function, the STATEMENT FUNCTIØN, is one constructed by the programmer as a part of his program.

As before, the FØRTRAN name of the STATEMENT FUNCTIØN must be limited to six alphameric characters, beginning with an alphabetic character. The function name may not be the same as any constant or variable name in the same program. A STATEMENT FUNCTIØN is limited to a single arithmetic or logical statement and only a single answer is provided. All logical STATEMENT FUNCTIØN names must appear in LØGICAL type statements.

The STATEMENT FUNCTIØN precedes the first executable statement of the program and it can reference any previously defined STATEMENT FUNCTIØN or a BUILT-IN or EXTERNAL function.

Examples of STATEMENT FUNCTIØNS are:

```
FXY(X,Y) = X**2 + EXP(Y*X)

R(S) = 6.48*S + 3.2E-5*S*S + 1.9E-11*S*S*S

BYK3(P,Q,B) = P*B*R(Q)**3
```

Note that the last STATEMENT FUNCTIØN (BYK3) referenced the second (R).

We could also have the following logical STATEMENT FUNCTIØNS:

```
LØGICAL LFI2, A, B, C, D, LFI3, L3

INTEGER E, F, G
```

```
LFI2(A, B, C, D) = ((.NOT. A .AND. C) .OR. (B .AND. D))
LFI3(A, B, E, F) = ((A .AND. B) .OR. (E .GT. F))
L3(E, F, G) = ((E .GT. F) .OR. (E .LE. G))
```

The arguments of the functions IN THE FUNCTION STATEMENT may *not* be *subscripted* variables even though the actual arguments used in the reference *may* be subscripted. Thus

```
AB(X, Y) = A**X + B*Y**2 + X*Y
```

may be referenced by

```
W73 = E*AB(F(3), G(I))*SQRT(G(I+1))
```

During execution, the value of F(3) is used for the dummy variable X and G(I) is used for the dummy variable Y in the FUNCTIΘN STATEMENT for the evaluation of AB. However, use of

```
AS(S(1), X(2)) = A*S(1) + B*X(2)**2 + S(1)*X(2)
```

as a FUNCTIΘN STATEMENT *is not permitted.*

There is one last function, and that is the FUNCTIΘN SUBPRΘGRAM which is *compiled exclusively* of your main or referencing program. It is *not* a part of the referencing program like STATEMENT FUNCTIΘNS, and it likewise is not supplied by the compiler (like BUILT-IN functions) or the allocator (like EXTERNAL functions). It is referenced in the referencing program exactly like EXTERNAL functions (if CΘMPLEX, DΘUBLE PRECISIΘN, or LΘGICAL, it's name must be placed in the corresponding TYPE statement in the referencing program). It differs from previously mentioned functions in that it's arguments may be any arithmetic or logical expression, array names, statement numbers preceeded by the character $, or nH....- a Hollerith (or alphameric) field.

The FUNCTIΘN SUBPRΘGRAM itself must have its first statement say

TYPE FUNCTIΘN F(A)

where TYPE is REAL, INTEGER, LΘGICAL, DΘUBLE PRECISIΘN, or CΘMPLEX. REAL and INTEGER need not be used if the naming rules (I, J, K, L, M, or N for integers) are adhered to, but the others are required. F is the *name* of the function and again it must be six or fewer alphameric characters starting with an alphabetic character. A is the argument(s) of the function--the arguments are not limited in number, but they must be separated by comma's; they may be array names or non-subscripted variable names. If any of the arguments are array names, they must appear in a DIMENSIΘN statement in the subprogram (the DIMENSIΘN statement must preceed any reference of the array name in an executable statement). For FUNCTIΘN (and SUBRΘUTINE) subprograms *only*, the DIMENSIΘN statement used may be of a special form in which the size of the array is defined not by an integer *constant* (as is usual) but by nonsubscripted integer *variables* (provided both the array name and all of the variable subscript names are *arguments* of the subprogram).

For example,

```
REAL FUNCTIΘN F1(A, I1, I2)
DIMENSIΘN A(I1, I2)
```

The program which references the FUNCTIΘN SUBPRΘGRAM F1 must also contain a DIMENSIΘN statement which specifies the maximum dimensions of the array A. The integer variables I1 and I2 cannot appear on the left side of an arithmetic or logical statement in the subprogram (i.e. they cannot be changed by the subprogram).

As before, only a single value is returned when the function is evaluated.

Thus, the function itself must appear at least once on the left side of a logical or an arithmetic statement.

An example of use of such a FUNCTIΘN SUBPRΘGRAM would be to evaluate a number factorial (e.g. 6! = 6*5*4*3*2*1) The complete FUNCTIΘN SUBPRΘGRAM to evaluate N! would be

```
        INTEGER FUNCTIΘN FACTRL(N)
        M = 1
        DΘ 10  I = 1, N
   10   M = M* I
        FACTRL = M
        RETURN
        END
```

In this example we again see the END statement which tells the compiler that the particular routine is finished. There is also a new statement introduced here for the first time and that is the RETURN statement. RETURN tells the computer to go back to the place in which it was referenced in the program which referenced the FUNCTIΘN. RETURN marks the logical end of the FUNCTIΘN subprogram, whereas the END statement marked the physical end of the FUNCTIΘN subprogram.

The factorial function may then be referenced by a main program by

```
        INTEGER FACTRL
        X = Y*Z(I)**2/FLΘAT(FACTRL(I))
```

note that we converted the integer answer provided by the FACTRL function subprogram to a real number by using the BUILT-IN function FLΘAT.

An additional example would be a main program which says

```
DIMENSIØN A(100, 100)
.
.
.
BX = PSUM(A, I, J, K, L)
.
.
.
END
```

and its FUNCTIØN SUBPRØGRAM for PSUM which is as follows

```
FUNCTIØN PSUM(X, M1, M2, I, J)
DIMENSIØN X(M1, M2)
XXX(Ƶ) = Ƶ*ALØG(Ƶ)
PSUM = XXX(X(I, J))
RETURN
END
```

Note here we used I and J in the main program (corresponding to M1 and M2 in the FUNCTIØN SUBPRØGRAM) to define the size of the array X (corresponding to A in the main program). The dimensions to be used for the array X were thus set up at "object time" or when the FUNCTIØN subprogram is executed by the computer. Note that the FUNCTIØN SUBPRØGRAM PSUM contains the STATEMENT FUNCTIØN

```
XXX(Ƶ) = Ƶ*ALØG(Ƶ)
```

Note also that we did not have to say

```
REAL FUNCTIØN PSUM(X, M1, M2, I, J)
```

since by the naming convention PSUM is a real name.

Besides alphameric information and logical variables, there is one additional argument possible for a FUNCTIØN SUBPRØGRAM; that is a FØRTRAN statement number (in the referencing program) preceeded by the character $.

For example,

```
        DIMENSIØN A(100, 100)
        .
        .
        .
        BX = PSUM(A, I, J, K, L, $100, $120)
        .
        .
        .
100     WRITE (6, 10)
        .
        .
        .
120     WRITE (6, 20)
        .
        .
        .
        END
```

Wherever a $N appears in the reference, that argument must be a $ in the subprogram. Thus we must have

```
FUNCTIØN PSUM(X, M1, M2, I, J, $, $)
```

The reason for this option is that in the event of an error; or for some other reason, you may not wish to return from the FUNCTIØN SUBPRØGRAM to the referencing program in the exact place you left (as you would when you reached the RETURN statement). Thus a special type of RETURN statement is permitted which looks like

```
RETURN N
```

where N is an integer constant or integer variable and corresponds to the Nth argument in the argument list. (The Nth argument must be a $ and it must correspond to a $NS in the referencing program where NS is a valid FØRTRAN statement number in the referencing program). Thus, in the above example we could have

```
RETURN 6
```

or

```
                    RETURN 7
```

where the first (RETURN 6) refers to FΘRTRAN statement number 100 and the second (RETURN 7) to statement number 120 in the main program.

A typical use in this example would be as follows:

```
        DIMENSIΘN A(100, 100)
        .
        .
        .
        BX = PSUM(A, I, J, K, L, $100, $120)
        .
        .
        .
  100   WRITE (6, 10)
   10   FΘRMAT (1H1, 10X, 40HRΘW'NUMBER'IN'ARRAY'A'IS'ΘUT'ΘF'SEQUENCE)
        STΘP
        .
        .
        .
  120   WRITE (6, 20)
   20   FΘRMAT (1H1, 10X, 43HCΘLUMN,NUMBER,IN,ARRAY,A,IS,ΘUT,ΘF,SEQUENCE)
        STΘP
        END
```

and the FUNCTIΘN SUBPRΘGRAM would be

```
        FUNCTIΘN PSUM(X, M1, M2, I, J, $, $)
        DIMENSIΘN X(M1, M2)
        XXX(Ƶ) = Ƶ*ALΘG(Ƶ)
        IF((I .GT. M1) .ΘR. (M1 .GT. 100))   RETURN 6
        IF((J .GT. M2) .ΘR. (M2 .GT. 100))   RETURN 7
        PSUM = XXX(X(I, J))
        RETURN
        END
```

Here if one of the integer variables corresponding to the row number of the matrix A is bad, we use the error return to statement 100 in the main program. Likewise, if a column variable is bad we use the error return to statement 120 of the main program. If both the row and column indices are valid, we calculate $X_{IJ} \log_e X_{IJ}$ and return to the place we left the main program (to store $A_{KL} \log_e A_{KL}$ in the memory location assigned to the variable BX).

There is one additional type of RETURN statement and that is

RETURN 0

where 0 is the integer constant zero. The execution of this statement results in a transfer to the system error program. The RETURN 0 is the only RETURN statement permitted in a main program.

FUNCTIØN SUBPRØGRAMS are one means of transferring control from the main program to another "homemade" routine. A second means of transferring control is by use of a SUBRØUTINE SUBPRØGRAM.

A SUBRØUTINE is very similar to a FUNCTIØN SUBPRØGRAM. The major difference is that the latter results in only a single value solution and SUBRØUTINES may produce many values for answers. The SUBRØUTINE returns the value(s) it calculates only through its arguments (or through variables in CØMMØN blocks, but more about that later).

No specific value or answer is associated with the subroutine name as is the case for all types of FUNCTIØNS.

A SUBRØUTINE is not referenced by being a part of an arithmetic or logical statement but only by the following FØRTRAN statement.

CALL S(A)

where S is the subroutine name (again--a maximum of six alphameric characters, the first being alphabetic) and A are the arguments. The arguments may be, as before for FUNCTIΘN subprograms, any arithmetic or logical expression, an array name, a statement number preceeded by the character $, or a group of Hollerith characters (nH...).

The actual subroutine must start with the statement

SUBRΘUTINE S(A)

and must contain at least one RETURN statement and end with an END statement as for FUNCTIΘN subprograms.

Error returns are permitted in subroutines as in FUNCTIΘN subprograms. Dimensions may be specified by variable integers for SUBRΘUTINES as well as for FUNCTIΘN subprograms.

A main program may call any number of SUBRΘUTINES and each SUBRΘUTINE itself may call any number of SUBRΘUTINES or FUNCTIΘN subprograms.

We could have easily used a SUBRΘUTINE in the place of the FUNCTIΘN PSUM in the earlier example. If we had done so the main program would appear as follows:

```
        DIMENSIΘN A(100, 100)
        .
        .
        .
        CALL PSUM(A, I, J, K, L, $100, $120, BX)
        .
        .
        .
  100   WRITE (6, 10)
   10   FΘRMAT  (1H1, 10X, 40HRΘW,NUMBER,IN,ARRAY, A,IS,ΘUT,ΘF,SEQUENCE)
        STΘP
```

```
            .
            .
            .
      120   WRITE (6, 20)
       20   FΘRMAT (1H1, 10X, 43HCΘLUMNˌNUMBERˌINˌARRAY ˌ AˌISˌΘUTˌΘFˌSEQUENCE)
            STΘP
            END
```

and the subroutine would appear as follows

```
            SUBRΘUTINE PSUM(X, M1, M2, I, J, $, $, Y)
            DIMENSIΘN X(M1, M2)
            XXX(Ƶ) = Ƶ*ALΘG(Z)
            IF((I .GT. M1) .ΘR. (M1 .GT. 100))   RETURN 6
            IF((J .GT. M2) .ΘR. (M2 .GT. 100))   RETURN 7
            Y = XXX(X(I, J))
            RETURN
            END
```

Note the difference between SUBRΘUTINES and FUNCTIΘN subprograms. SUBRΘUTINE names are not on the left side of any arithmetic statement within the subprogram; SUBRΘUTINE names are referenced only by the CALL S(A) statement and their names are not part of arithmetic or logical statements within the referencing program; to return the answer to the referencing program, the variable that contains the answer (BX in this case) must be in the calling list of arguments. We could have used SUBRΘUTINE PSUM to return not only the value of BX but also that of many other variables, whereas the FUNCTIΘN PSUM could only return a single answer stored as the name of the FUNCTIΘN (i.e. PSUM).

Until now, we have discussed only EXTERNAL FUNCTIØN and SUBRØUTINE subprograms (those which are compiled separately from the main program) in which the first line of the compilation must be a FUNCTIØN or a SUBRØUTINE statement. For the most part these are the most common type. However, it is possible to have INTERNAL FUNCTIØNS or SUBRØUTINES which are referenced in the usual manner by the main program, but which follow directly after the last statement of the main program.

```
C                 MAIN PROGRAM STARTS HERE
                  .
                  .
                  .
                  CALL X(Y, Z)
                  .
                  .
                  .
                  W = Y + F3(Z)
                  .
                  .
                  .
                  SUBRØUTINE  X(A, B)
                  .
                  .
                  .
                  CALL R(A)
                  .
                  .
                  .
                  SUBRØUTINE R(P)
                  .
                  .
                  .
                  FUNCTIØN F3(Q)
                  .
                  .
                  .
                  END
```

The compiler assumes that all statements between

```
                  SUBRØUTINE  X(A, B)
```

and

```
                  SUBRØUTINE  R(P)
```

belong to SUBRΘUTINE X. Likewise the statement

FUNCTIΘN F3(Q)

marks the end of SUBRΘUTINE R and

END

is the end of the *entire set* of programs and is the *only* END statement in this set of programs.

As mentioned earlier, this technique is not often used since INTERNAL SUBRΘUTINES and FUNCTIΘN subprograms may be referenced *only* by the main program or other internal subprograms and not by external subprograms. Also, in the event of an error, a single subroutine cannot be re-compiled separately.

Many programs are written having very small main programs which reference many small subroutines. This technique permits compilation and debugging of small segments of the problem at a time and facillitates changes in and proliferation of specialized routines. In many programs the main program *has no executable arithmetic, logical*, or *I/O statements*--it first calls a special SUBRΘUTINE to read in data, it then calls a SUBRΘUTINE to test the input data, then SUBRΘUTINES are called to perform the desired calculations, and finally a SUBRΘUTINE is called to write out the results.

SECTION VI SPECIFICATION AND DATA STATEMENTS

Specification statements tell the FØRTRAN compiler how data are to be stored in the computer. Since these statements do not result in actual instructions that the computer executes when it runs the program, these are called "non-executable" statements. We have already discussed two of the four specification statements, the FØRMAT and DIMENSIØN statements.

The FØRMAT statement instructs the computer how to prepare and decode the output and input information, respectively, that goes to and from the I/O (input/output) channels and main memory. The FØRMAT statement may appear at any place in the program.

The DIMENSIØN statement specifies the maximum size of each array so that the correct amount of storage for each variable is properly allocated. The DIMENSIØN statement must appear before any executable statement of a main program, function subprogram, or subroutine.

As mentioned earlier, the dimensions of a variable may also be specified in a TYPE statement (and, as we shall soon see, by a CØMMØN statement); however, the dimensions of a variable may only be specified once in either a DIMENSIØN, TYPE, or CØMMØN statement. A variable dimensioned more than once will be considered to be multiply-defined -- a condition the compiler will not tolerate even if both definitions say the same thing.

The EQUIVALENCE statement, the third type of specification statement, does just what it says, it makes two or more variables (or arrays) equivalent (but NOT equal). This permits the multiple use of storage locations within any separately compiled FØRTRAN program or subroutine. The EQUIVALENCE statement is of the form

```
EQUIVALENCE     (Variable Names),    (Variable Names),....
```

where the variable names within any one set of parentheses share the same storage locations. The variable names within the parentheses are separated by commas, and each pair of parentheses is separated by commas.

An example would be

```
DIMENSION  F(5), G(10), H(8)
EQUIVALENCE  (A,B,C), (F(3), G(7), H(1))
A = 0.0
B = A + 6.0
Z1 = (A + 2.0)
C = 4.0
Z2 = (A + 2.0)
```

Here we have A, B, and C all assigned to the SAME storage location in memory. We also have F(3), G(7), and H(1) all stored in the same location. (Note that the EQUIVALENCE statement containing the arrays F, G, and H <u>had to follow</u> the DIMENSIØN statement in which they were defined.) As a result, F(4), G(8), and H(2) are also stored in the same memory location. In fact, F, G, and H are stored as follows (assuming G(1) is in location x)

STORAGE LOCATION		
x		G(1)
x+1		G(2)
x+2		G(3)
x+3		G(4)
x+4	F(1)	G(5)

x+5	F(2)	G(6)	
x+6	F(3)	G(7)	H(1)
x+7	F(4)	G(8)	H(2)
x+8	F(5)	G(9)	H(3)
x+9		G(10)	H(4)
x+10			H(5)
x+11			H(6)
x+12			H(7)
x+13			H(8)

Note that if one element of an array is equivalenced to an element of another array, the entire arrays are equivalenced.

Interestingly enough, we need not have specified H(1) in the EQUIVALENCE statement since the compiler knows H is an array (because it appears in a DIMENSION statement) and if H appears without a subscript, it is assumed to be the first element in the array. However, if we forgot to specify F(3), we would have equivalenced F(1) automatically to G(7) and H(1).

Note that the maximum size of F was dimensioned as 5. However, F(6) does exist -- it is G(10) and H(4); likewise F(10) is H(8) and G(14). F(0) also exists and is G(4) as well as H(-1). What the DIMENSION statement does is allocate a certain amount of storage to a variable array but it does not limit you to using only that storage area. Thus if you are not careful and you exceed a dimension, you could get into considerable trouble by destroying information stored as another variable in the adjacent array.

If we look back at the example of the EQUIVALENCE statement, we can see one reason why variables that are equivalenced are not necessarily equal.

We said that A, B, and C were all equivalent and we then zeroed out their common storage location by saying A = 0.0. We then set B = A + 6.0 = 6.0 so that the common storage location now contains a 6.0.

Z1 is than defined as A + 2.0 or 8.0. Before we evaluate Z2 we set C = 4.0 so that A, B, and C are all 4.0. One would expect thus that Z2 = A + 2.0 = 6.0, but that is probably not true. Z2 probably is equal to 8.0 just like Z1. The reason for this is that in most cases the compiler will recognize that A + 2.0 appears in both the expression for Z1 and that for Z2. It will also note that the value of A is not changed between the execution of these two statements (even though it really is changed by virtue of its being equivalenced to C) so that it will optimize the program by storing A + 2.0 in Z1 and then setting Z2 equal not to A + 2.0 but to Z1 which already contains what it thinks is A + 2.0. If Z1 and Z2 were longer expressions that both contained A + 2.0, the A + 2.0 would probably still have only been calculated for evaluating Z1 and it would have been put into a temporary storage location for use in calculating Z2; it still would not be changed to account for the statement C = 4.0. Although this is not a usual occurrence, it is one of the ways to get in trouble if you don't know what the compiler is doing and if you assume EQUIVALENCE means EQUAL.

Normally, only variables of the same mode are made equivalent to avoid errors, since, for one reason, double precision and complex variable each require two adjacent words per element in their arrays. (References to complex or double precision variables in EQUIVALENCE statements are references to the first word of the pair.)

Thus if we had

```
COMPLEX  A(100), B(100)
DIMENSIΘN  C(100), D(200)
EQUIVALENCE   (A, B, C, D)
```

we in effect specify the storage of 200 words and A(2), B(2), C(3), D(3) are all stored together as are A(3), B(3), C(5), and D(5). Likewise C(99), D(99), A(50), and B(50) are stored together as are A(100), B(100), and D(199) since the arrays A and B are complex and require two adjacent memory locations for each "word".

The last type of specification statement is the CΘMMΘN statement which, as its name implies, makes certain areas common to subroutines, function, subprograms, and the main program. The CΘMMΘN statement is of the form:

$$\text{CΘMMΘN} \quad /BN_1/VN_1/BN_2/VN_2 \; \ldots\ldots$$

where BN represents the name of the common block to which the variable names VN belong. Block names must be six or less alphameric characters starting with an alphabetic character. Each variable name in a list must be separated by commas from its neighboring variable names. Common areas having block names are called "labeled" common. If the block name is omitted, the variable names following are given a "blank" name or are part of Blank common (in contrast to labeled common).

Normally there is no need for an area of Blank common so it is rarely used.

An example of the use of CΘMMΘN statements is as follows:

```
CΘMMΘN  /BLΘCK1 / S , Y, Ƶ(100) /BLΘCK2/ A, B
```

Note that Ƶ is DIMENSIΘNED 100 in the CΘMMΘN statement and thus Ƶ <u>cannot</u>

appear in a DIMENS1ØN statement or in a TYPE statement with its dimensions associated. If the above CØMMØN statement appears in a main program and

```
                    CØMMØN   /BLØCK1/ P(52), Q(50)
```

appears in one of its subroutines, P(1) is stored in the same location as S, P(2) the same as Y, P(3) the same as Ƶ(1) ..., and Q(50) is stored in the same location as Ƶ(100). Note that the variable name within a specific common block in two or more routines need not be the same, but the same common blocks must be the same size.

Data listed in CØMMØN are usually stored in a large block in upper memory of the computer, with all labeled common first and blank common (if there is any) last.

All additional CØMMØN statements appearing in a single main program or subroutine will extend the size of those CØMMØN blocks. The size of a CØMMØN block is equal to the sum of the storage requirements of its variables. The order of variables stored in CØMMØN is the same as the order in which they are listed in the CØMMØN statement(s).

If CØMMØN statements contain variables whose dimensions they define, these CØMMØN statement must precede any executable statements of the program containing those variables.

Normally, it is not a good idea to put variables in both CØMMØN and EQUIVALENCE statements, since the CØMMØN statement orders the locations of its variables in memory and so does the EQUIVALENCE statement, and conflicts may result. Variables belonging to CØMMØN which appear in EQUIVALENCE statements results in all variables in that equivalence class automatically being placed in CØMMØN.

EQUIVALENCE statements may not alter the order of CØMMØN storage except that they may extend a CØMMØN block beyond the last assignment made for that block by the CØMMØN statement.

For instance, if we have

```
        CØMMØN  /BLØCK1/ A, B, C(50), D(80)
```

we may also have

```
     DIMENSIØN X(50), Y(80), Ƶ(100)
     EQUIVALENCE  (A, X(1)), (B, Ƶ(2)),  (D(51), Y(1))
```

which in essence stored the variables as follows:

```
A     B     C(1)  C(2)...C(50) D(1)...D(51)   D(52)...D(80)
X(1)  X(2)  X(3)  X(4)
Ƶ(1)  Ƶ(2)  Ƶ(3)  Ƶ(4)  Ƶ(52) Ƶ(53)
                                   Y(1)       Y(2)...Y(30)  Y(31)...Y(50
```

and extends CØMMØN block BLØCK1 by 20 locations corresponding to Y(31) through Y(50).

However, the following EQUIVALENCE statement is not permitted with the above CØMMØN statement

```
        EQUIVALENCE  (A, X(2)), (C(2), X(4))
```

since equivalencing A and X(2) causes BLØCK1 to be extended backward in memory (to X(1) which precedes A) and start one statement before the CØMMØN statement says it starts (the location assigned to A must be the start of BLØCK1 in the example). Also, if A is equivalent to X(2) and if C(2) is equivalent to X(4) there is an error, since C(1) must be equivalent to X(3), and there is no place in the array X for the variable B which appears between A and C(1), (or, by the equivalence statement, between X(2) and X(3)).

CØMMØN statements are often used to communicate information between SUBRØUTINES or between SUBRØUTINES and the main program without having to include them in the subroutine call list.

Recall an example several sections back where we wished to read in ten data cards (each containing ten numbers), square the numbers, and write out the numbers and their squares. This could have been accomplished as follows:

MAIN PROGRAM

```
      CALL   READER
      CALL   CALC
      CALL   WRITER
      STØP
      END
```

SUBROUTINE READER

```
      SUBRØUTINE READER
      DIMENSIØN   B(100)
      CØMMØN  /DATA/  A(200)
      EQUIVALENCE   (A, B)
      READ   (5, 10)   B
 10   FØRMAT   (10F8.2)
      RETURN
      END
```

SUBROUTINE CALC

```
      SUBRØUTINE CALC
      CØMMØN  /DATA/  X(200)
      DØ  10   I = 1,100
   10 X(100+I) = X(I)*X(I)
      RETURN
      END
```

SUBROUTINE WRITER

```
      SUBRØUTINE WRITER
      DIMENSIØN  Y(100)
      CØMMØN  /DATA/   Ƶ(200)
      EQUIVALENCE   (Ƶ(101),  Y(1))
      NPAGE = 1
      WRITE  (6, 20)  NPAGE
   20 FØRMAT (1H1, 20X, 16HSQUARING,PRØGRAM,  40X, 5HPAGE, I3  ///
     1 14X, 6HNUMBER, 14X, 6HSQUARE)
      NPAGE = NPAGE + 1
      WRITE (6, 30) (Ƶ(I),  Y(I),  I = 1,40)
   30 FØRMAT  (2F20.4)
      WRITE  (6, 20)  NPAGE
      NPAGE = NPAGE + 1
      WRITE (6, 30)  (Ƶ(I),  Y(I),  I = 41,80)
      WRITE  (6, 20)  NPAGE
      WRITE (6, 30)  (Ƶ(I), Y(I), I = 81,100)
      RETURN
      END
```

Admitted, the above is not as "clear" as doing the calculation all in one program but it is a very simple example concerning only one variable and its only purpose is to demonstrate the use of the EQUIVALENCE and CΘMMΘN statements. The above example would not have to use the EQUIVALENCE statements if we had set up two CΘMMΘN blocks as follows:

```
        SUBRΘUTINE READER
        CΘMMΘN /DATA1/  B(100)
        .
        .
        .
        SUBRΘUTINE CALC
        CΘMMΘN  /DATA1/  X(100)  /DATA2/  W(100)
        DΘ  10  I = 1,100
   10   W(I) = X(I)*X(I)
        .
        .
        .
        SUBRΘUTINE WRITER
        CΘMMΘN  /DATA1/  Z(100)  /DATA2/  Y(100)
        .
        .
        .
```

In this instance different CΘMMΘN blocks supplied information to different subroutines as needed. If we had desired to write out only the squares of the numbers and not the numbers themselves, DATA2 need have been the only CΘMMΘN block present in SUBRΘUTINE WRITER.

There is one last non-executable statement yet to be discussed and that is the DATA statement. The DATA statement may be used to initialize variables or arrays or to set up constants at the time the program is loaded for execution. The DATA statement is of the form

```
DATA   LIST/VALUES/, LIST/VALUES/,...
```

A typical example would be

```
DATA  A, B, C, K/6.2, 7.3, 7.3, 4/
```

Here, 6.2 is stored in the memory location assigned to A, 7.3 in B and C, and 4 in K *when the program is loaded for execution*. The reason that emphasis was placed on the last part of the last sentence is that this is the *ONLY* time the variables or constants in the DATA statement are loaded with the values in the DATA statement. The reason that the DATA statement is non-executable is that it is used *only* when the program is *loaded* for execution and is ignored thereafter. Thus the DATA statement may appear *anyplace* (provided it *follows* any DIMENSION or TYPE statements referencing variables in the DATA list) in the program and the variables and constants it contains will be stored *prior* to program execution. If you had the arithmetic statement

```
100 X = 10.4
```

You would store the real number 10.4 in the memory location assigned to X *every* time you passed through statement number 100.

If you had the DATA statement

```
100    DATA  X/10.4/
```

you would store the real number 10.4 in the memory location assigned to X *prior to execution when the program is loaded* and statement 100 would have *no effect thereafter*. (Incidentally, you are *not* permitted to *transfer* to a DATA statement -- as by an IF or GØ TØ statement. You are also *not* permitted to transfer to a FØRMAT statement.)

If you had

```
          100    X = 10.4
                 DATA  X  /7.4/
```

you would set the real number 7.4 in the storage location assigned to the variable X when the program is loaded. X would remain 7.4 until statement 100 is reached for the first time, and then X would be 10.4. X would remain 10.4 thereafter (even though the very next statement is a DATA statement concerning X) until it again appeared on the left side of an arithmetic statement.

Recall out first example of the DATA statement where we had

```
DATA   A, B, C, K /6.2, 7.3, 7.3, 4/
```

Here we stored 7.3 in <u>both</u> B and C. We could have told the DATA statement that two (or more) consecutive values were identical by writing them <u>once</u> and preceding them by an asterisk and the (integer) number of them that is required. For example,

```
DATA  A, B, C, K /6.2, 2*7.3, 4/
```

It is possible to reference arrays in DATA statements as follows:

```
DIMENSIØN   W(10),  X(10, 2)
DATA  (W(I), X(I,1), X(I, 2), I = 1,10)/30*0.0/
```

This serves to zero out the arrays W and X prior to execution. Some computers permit not only the use of the implied DØ in DATA statements as above but a also the use of unsubscripted arrays to reference the <u>ENTIRE</u> array (not just the first element of the array as in arithmetic statements).

```
DIMENSIØN   W(10), X(10, 2)
DATA  W, X /30*0.0/
```

is thus equivalent to the above.

One may initialize double precision, logical and complex variables as well as real and integer numbers in DATA statements as follows

```
DΘUBLE PRECISIΘN  X, Y
LΘGICAL IS, IT
CΘMPLEX  V, W
DATA  I1, I2, V, W, X, Y, Z /2*1, (6.2, 1.4), (3.1, 1.2),  6.3D+0,
      9.3274112D-06, 1.0/, IS, IT/.TRUE., .FALSE./
```

Here we have initialized the two integers I1 and I2 as 1, the complex number V as 6.2+1.4i, W as 3.1+1.2i, the double precision numbers X and Y as 6.3 and 9.3274112X10^{-6}, respectively, the real number Z as 1.0, and the logical variable IS and IT as True and False respectively. It is possible to store an octal number in a DATA statement as follows

```
DATA ΘCT1 /Θ3247/
```

where we have to precede the value with the character Θ (for Θctal). Recall that an octal number contains 12 characters per word; we only used four for the word ΘCT1 so the computer automatically adjusts the 3247 to the rightmost four positions in the location assigned to ΘCT1 and fills in the left eight characters with zeros. In other words, octal words are right-justified in storage.

It is also possible to store alphameric information in DATA statements (with six or fewer characters per word).

```
DIMENSIΘN   K(2)
DATA  K/10HEND,ΘF,JΘB/
```

Here we have the two word array K. The letters END,ΘF are stored in K(1) and ,JΘB in K(2). Notice that K(2) only contains four of its six characters (blank, J, Θ, and B). These four characters are left-justified and the two rightmost positions of K(2) are filled in with blanks. In other words, alphameric information is left-justified in storage.

If you recall, we mentioned earlier that FØRMATS could be stored in DATA statements. For example

```
              DIMENSIØN  KFMT(2)  A(9)
              DATA KFMT/11H(10F6.4,I2)/
        100   READ (5, KFMT)  A, B, L, Z
```

Here we stored the FØRMAT (10F6.4, I2) in the array KFMT. When we execute statement 100 we READ in the array A and B as the first ten F6.4 real numbers, L as an I2 integer, and then restart the FØRMAT to read in Z as an F6.4 real number.

This type of usage permits modifying FØRMATS during program execution as follows: Suppose we were writing out matrices which varied in size from 2x2 to 12x12. Each element of the matrix is a real number and is to be written out under an F10.2 format. If we wanted to write out each matrix as an n by n array (n varying from 2 to 12 depending upon the particular matrix) and it was stored in the array A. It could be done as follows:

```
     DIMENSION  A(12,12), IFMT(3), KFMT(12)
     DATA  IFMT /18H(/,,,,,,,,,,F10.2)/,
   1 KFMT/1H1, 1H2, 1H3, 1H4, 1H5, 1H6, 1H7, 1H8, 1H9, 2H10, 2H11, 2H12/
     .
     .
     .
C CALCULATION OF MATRIX A(N,N) IS PERFORMED HERE
     .
     .
     .
     IFMT(2) = KFMT(N)
     WRITE (6, IFMT)  ((A(I,J), J = 1, N), I = 1,N)
     .
     .
     END
```

Here we set up the FØRMAT (for writing out the matrix) as array IFMT. We did not know how many F10.2's per line to expect, so rather than have eleven separate FØRMATS for A(2,2) through A(12,12), and eleven separate WRITE statement to choose from we merely modify IFMT to suit the matrix size. This is done by placing the integer N in IFMT(2) which says we write out under format N F10.2. Note that N had to be stored as an alphameric number in the array KFMT and we had to place KFMT(N) into IFMT(2) by using the arithmetic statement

```
IFMT(2) = KFMT(N)
```

Had we said

```
IFMT(2) = N
```

we would have had N stored in IFMT(2) but as an integer number and not in the alphameric form required for FØRMAT's.

Note also that we were required to use the indices I and J in the WRITE statement for two reasons. First, the omission of indices would have written out the entire 12x12 array A even if we had only a 2x2 matrix. Secondly, even if it was a 12x12 matrix, the leftmost index, I, is the row number of the element and J is the column number--had we omitted the index, the computer would have automatically varied I most frequently (for each J) and the output matrix would have A(1,2) where you expected A(2,1) -- that is, the rows and columns would be reversed.

One particular use of the DATA statement that is not permitted is introducing data into variables or constants which are listed in labeled CΘMMΘN. To enter data into such variables, a special form of subprogram must be used, the BLΘCK DATA subprogram.

Only non-executable statements may appear in the BLΘCK DATA subprogram (BLΘCK DATA, CΘMMΘN DIMENSIΘN, Type statements, DATA, and END).

A typical example would be to enter the values 10.3 and 146 as the variables X and IX and to zero out the array Y where X, IX, and Y all appear in a CΘMMΘN statement. The main program and one or more subroutines may have the same CΘMMΘN statement as that appearing in the BLΘCK DATA subprogram which follows:

```
BLΘCK DATA
CΘMMΘN  /BLΘK1/X,IX,Y(34),M,R3,S
DATA  X,IX,Y/10.3,146,34*0.0/
END
```

Note that the BLΘCK DATA subprogram is *not called or referenced* by any other routines so that it does not require a RETURN statement. It is used only during program loading to permit introduction of the desired initial values of X, IX, and Y. Note also that *all* the elements in the CΘMMΘN block (BLΘK1) must be listed in the CΘMMΘN statement even though only three of them are affected. (Whenever a CΘMMΘN statement is in a program, *all* of the variables stored in that common block must be accounted for.)

SECTION VII FLOWCHARTING AND PROGRAM DEBUGGING

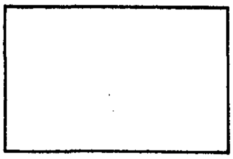
Although the topic of FLØWCHARTING was left until the end of this manual, it is actually one of the first things that a programmer does before he starts to write any FØRTRAN statements.

Once it has been decided that a particular problem or application is suitable for computer solution or processing, the programmer must perform a detailed analysis of the particular problem. He must decide exactly how he intends to provide a solution and this is usually done by breaking the problem down into discrete parts. Once a detailed solution has been determined, the programmer flowcharts his solution before attempting to code it for the computer. There are actually two "levels" of flowcharting. The "system" flowchart is used to visually represent the flow of data and the gross sequence of computer (and peripheral device) operations. It serves to indicate the overall happenings associated with the program while it avoids most programming details and calculations.

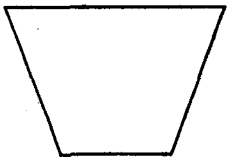
The "program" flowchart (which is usually prepared after the system flowchart) indicates the detailed workings of the program. The program flowchart serves as 1) an aid to program development (it permits working out the detailed logic involved and assuring that there are "no paths left untried"), 2) a guide in the actual coding and debugging of the problem, and 3) a means of documenting the program. This latter aspect may seem unnecessary but it is possibly the most important function of the three. After you have completed your program and it is operating satisfactorily, you will probably rush off and start working on something else. If, six or nine months later, someone wants you to modify your old program (or even worse, if someone else wants to modify it himself) you will be amazed at how many details (which you at one time probably lost some sleep over) you have forgotten. This is where an up-to-date flowchart pays for itself severalfold. It is much easier to look at a picture of what's happening than it is to try to untangle all of the logic in the code itself. It is essential, for this reason, that any modifications made to the program are always noted on the flowchart to keep it updated.

The following list of symbols are currently used in system flowcharting

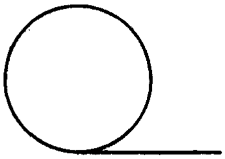
SYSTEM FLOWCHART SYMBOLS



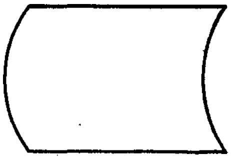
Processing - this symbol may contain an entire section of your program flowchart



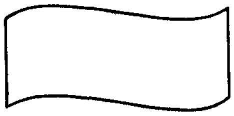
Input/Output



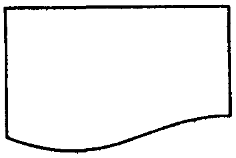
Magnetic Tape (input or output)



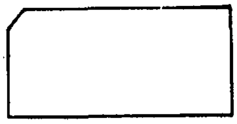
Drum, Disc, or Random Access unit (input or output)



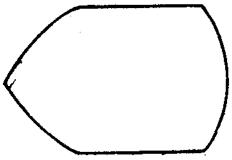
Paper Tape (input or output)



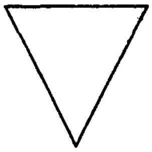
Printed report or document (output)



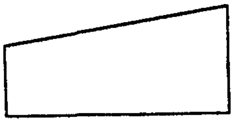
Punched Card (input or output)



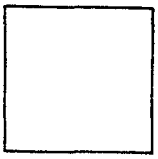
Display - Plotter, CRT, etc. (output)



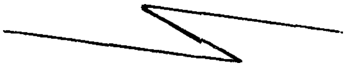
Off-line storage - magnetic or paper tape, cards, etc. (input or output)



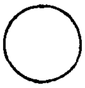
On-line keyboard



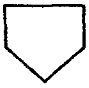
Auxiliary operation



Communications link

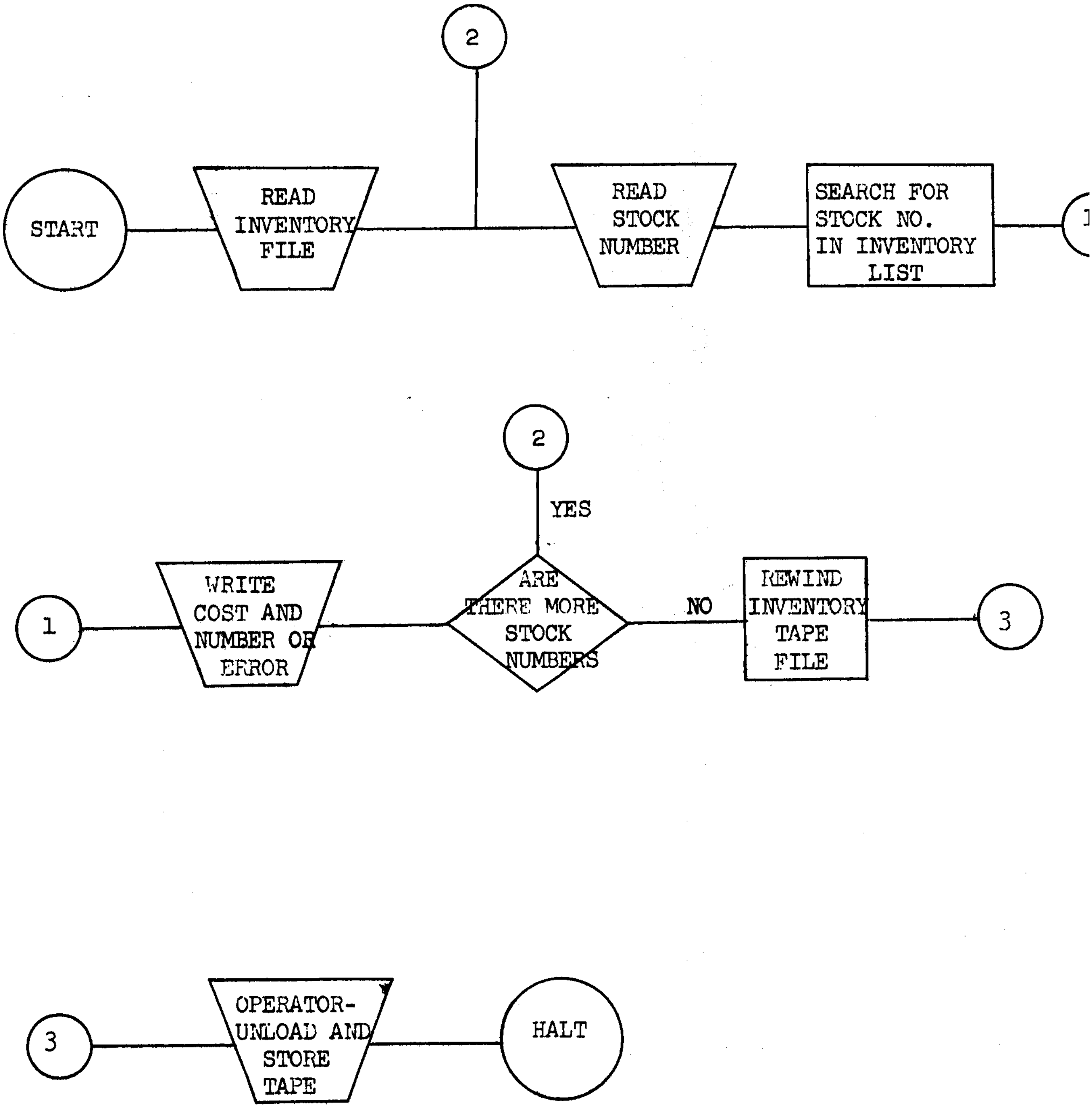


Connector - to reference a point on the same page



Off-page connector to reference a point on another page.

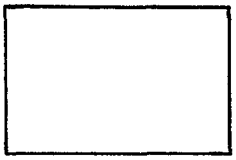
A typical example of the use of a system flowchart would be that of reading in an inventory file tape, searching for particular stock numbers, writing out the current number of items in inventory and cost of the items with the associated stock numbers (or stating that it is not on tape if it could not be found), rewinding the inventory tape, and writing a message to the operator telling him to remove the tape and place it in a special tape storage location. The system flowchart for this particular application could appear as follows (flow is from left to right and top to bottom.)


2
START
READ INVENTORY FILE
READ STOCK NUMBER
SEARCH FOR STOCK NO. IN INVENTORY LIST
1
2
YES
1
WRITE COST AND NUMBER OR ERROR
ARE THERE MORE STOCK NUMBERS
NO
REWIND INVENTORY TAPE FILE
3
3
OPERATOR-UNLOAD AND STORE TAPE
HALT

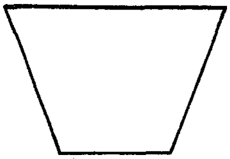
Note that in the above flowchart we described the "high points" in the application and we did not get into programming details such as what is on the file, how it is read and stored, or how we search for the stock number in the inventory list. It is possible to see the gross characteristics of the program, however, and what the program is intended to do (for instance if we forgot to find out if there was more than one stock number to be searched for or if we forgot to tell the operator where to store the inventory tape it should be obvious on the system flowchart).

In the above flowchart there was one symbol that we did not mention previously, and that was the diamond-shaped question "Are there more stock numbers?" This "decision" symbol is really one of the "programming" flowchart symbols but, as you will see from the following list, several symbols have identical meanings for both system and program flowcharts.

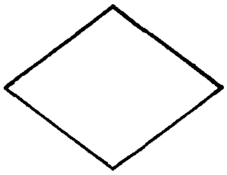
PROGRAM FLOWCHART SYMBOLS



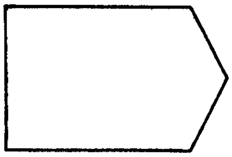
Processing - one or more program processing instructions



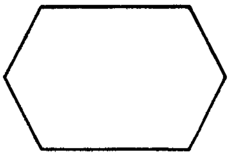
Input/Output



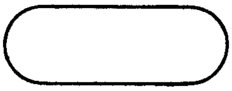
Decision - branching occurs here



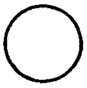
Modification or initialization



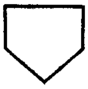
Predefined Process - operations not detailed - eg call a subroutine



Terminal - the beginning or end in a program



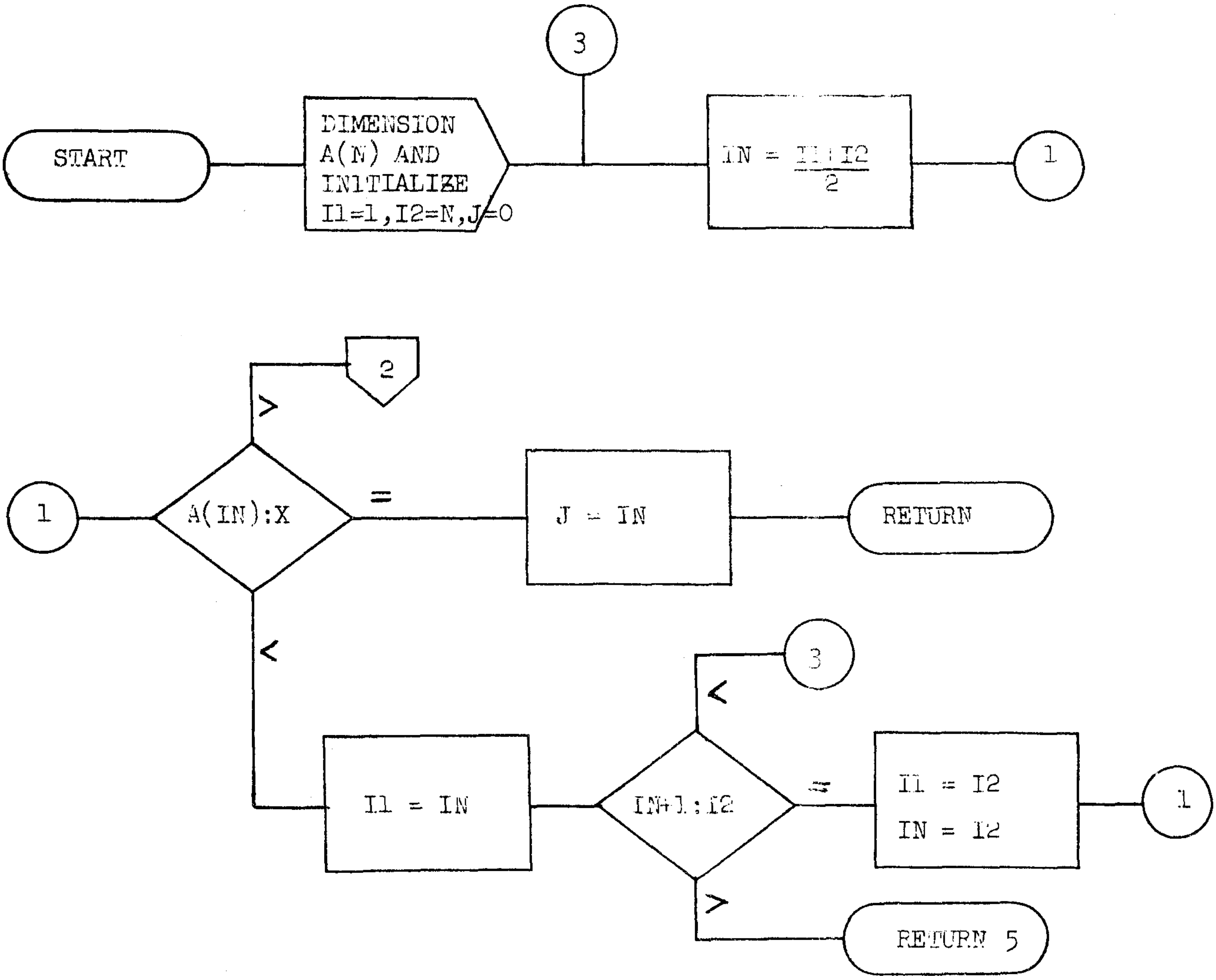
Connector - to reference a point on the same page

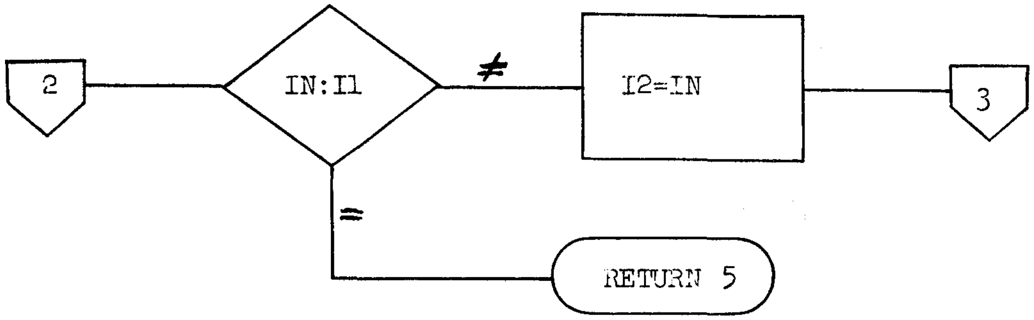


Off-Page Connector - to reference a point on another page

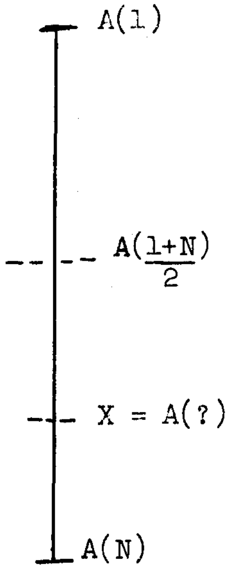
As an example of the use of program flowchart symbols, we could take the one processing box in the previous example, "search for stock number in inventory list" and program the search as an "artillery" (or binary) search subroutine. Assume the stock numbers are stored in an array, A, and that the call list of the subroutine contains the array, a variable X consisting of the stock number of interest, and an integer J into which we will store the array index corresponding to the stock number sought. If the desired stock number does not appear in the array, a message to this effect will be printed out, J will be set to zero, and an error RETURN will be made.

An artillery search is implemented by testing to see if X is equal to $A(\frac{N+1}{2})$ (the element in the array midway between the first and last) and if it is not, test the element midway between $A(\frac{N+1}{2})$ and the appropriate end. Keep testing elements at midpoints until the answer is found. The only requirement of such a search is that the array A is a monotonically increasing sequence. This type of search is usually considerably faster than testing each element of the array in sequence.



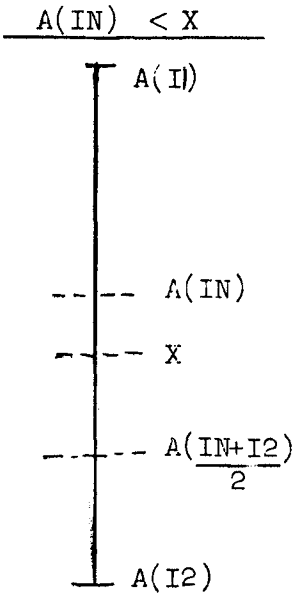


Note that the entire program flowchart above was necessary to describe what occurred in the one symbol "search for stock number in inventory list" in the system flowchart. To clarify what is happening in the program flowchart, a diagram plus a little descriptive paragraph might help.



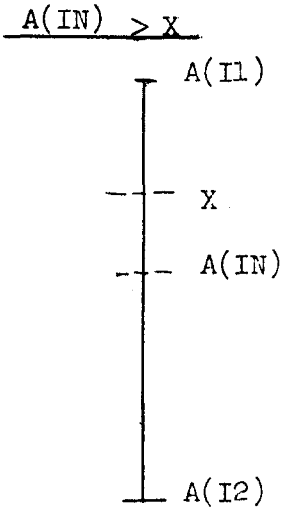
We started out with an array A of N elements and we are looking for the position of X in the array. We find the position of X by first testing $A(\frac{1+N}{2})$. On the first line of the flowchart we set up the indices I1 = 1 and I2 = N as well as the error return for J (J = 0) in the event that we do not find the desired solution. We then calculate the location of the element in A halfway between A(1) and A(N) as A(IN) where IN = $\frac{I1+I2}{2}$ (= $\frac{1+N}{2}$ for the first iteration).

On the second line of the flowchart (starting at #1) we test to see if A(IN) is greater than, equal to, or less than X. If A(IN) is equal to X we have found our answer and we set J = IN and return. If A(IN) is less than



X we know that X must lie between A(IN) and A(I2) so we set I1 = IN and search between A(IN) and A(I2). However, before we do this we must test to see 1) if IN+1 is equal to I2 (if it is, I2 is the only value of A that could be equal to X as we know X > A (IN) so we set I1 = I2 and IN = I2 and check for A(I2) $\overset{?}{=}$ X), 2) if IN+1 is greater than I2 (if it is, IN must equal I2 and we have failed to find the desired value so we use the error RETURN 5), or 3) if IN+1 is less than I2 (if it is, we go to 3, update IN, and make another iteration).

Going back to the start of the second line (#1) we must then see if A(IN) is greater than X. If it is, we go to the third line (#2) and test to see if if IN equals I1 (if it does, we have failed to find the desired value so we go to #4 and use the error RETURN 5) or if IN is not equal to I1 (if not, we set I2 = IN, go to #3, update IN, and make another iteration).



If you think that constructing and following the logic in the flowchart was tough, try following the logic in the subroutine itself (which follows) without the aid of the flowchart.

```
      SUBRØUTINE SEARCH (A,N,X,J,$)
      DIMENSIØN  A(N)
      I1 = 1
      I2 = N
      J = 0
10    IN = (I1+I2)/2
20    IF (A(IN)-X)   30, 60, 70
30    I1 = IN
      IF(IN+1-I2)   10, 40, 50
40    I1 = I2
      IN = I2
      GØ TØ 20
50    RETURN 5
60    J = IN
      RETURN
70    IF(IN .EQ. I1)    RETURN 5
      I2 = IN
      GØ TØ 10
      END
```

As you can see, the flowchart will be a tremendous aid in following the logic of the above subroutine during debugging. If you happened to make a mistake in one of the above IF statements, to say that it would be a chore attempting to discover the error merely by looking at the FØRTRAN statements without a flowchart is probably a rather superb understatement.

As you can see, even though flowcharting is one of the first things the programmer does before actually writing his program and although debugging is one of the last things he does before the program is set up for production use, the two phases are closely related.

If your flowchart is carefully worked out to cover all possible situations and is very detailed, it will greatly simplify the task of tracing through the program to locate errors. Once you find out where the error is occurring (from the funny looking output) you can trace backwards in your flowchart to see what would cause that or those particular things to occur. If the flowchart indicates that area of the program to be apparently error free, then a close comparison between the flowchart and the program listing might locate the problem.

The basic reason for debugging a program is that they almost never work correctly at first. The cause goes back to a statement made in the INTRODUCTION that the computer does exactly what you tell it to do and not necessarily what you want it to do. This leads to two sources of error -- 1) you told it to do the wrong thing and 2) you really didn't tell it to do anything for a particular situation. One of the seemingly impossible tasks of the programmer is for him to attempt to map out all possible logical paths in his program even though some (if not most) combinations of logic will not normally, if ever, occur. You saw an example of the logical complexity of the above simple subroutine which contained only one logical and two arithmetic IF statements. Picture the case where there are 50 or 100 input, calculational, or output choices available--you can never expect to be sure that your program does what's expected for all combinations of the logical choices permitted in such a situation (and 50 or 100 choices are still a relatively few). This leads to debugging.

First, the FΘRTRAN compiler will catch many of the errors of syntax in your program (you meant GΘ TΘ but said GΘ TO with a zero rather than an Θ). It may also catch much of your poor spelling and many logical problems (you set up the variable XARC in a DATA statement and then said W = XARK or you transferred to statement number 103 which just happens not to exist in your program, etc.). It is when you stop getting snide compiler--generated diagnostic messages that the real trouble starts--now you are on your own.

One of the first things you will probably do in debugging a problem is to run a test case made up of simple round numbers like 100, 1000, etc. and specify the least number of options available. First try it on a desk calculator following exactly the path the program would follow, statement by statement. If you got the answer you expected, try the <u>same</u> problem on the program. When it fails, you can do several things. A memory dump will probably help to find lost variables or constants that were not updated. Extra WRITE statements to give you debug output at crucial points in the calculations (if special DEBUG DUMPING routines are not a part of your software) are helpful at this stage. Showing your program to a knowledgeable friend who is unfamiliar with what you are trying to do may permit his telling you what you <u>are</u> doing (compared to what you <u>want</u> to be doing).

One prime catch is to never <u>assume</u> you are doing something in a part of the program and gloss over it because it worked once before--read it and you may find that the exact logic is slightly different than you "remembered".

If possible try to check out one section of the program at a time. This is the advantage of having many small subroutines rather than one large program. Each subroutine may be tested with "TYPICAL" data by using "special" main programs to provide the desired input. Try to test both positive and negative data covering the entire range of interest to catch any limits you might have (log and exponential routines, etc.)

After you have gotten your simple case debugged - try different logical options one or two at a time until you are fairly sure that for most typical cases the program works. (You can never hope to try all possible paths since for a set of only 20 yes-no decisions there are about one million possible paths.) Then you can turn it over for production runs--but keep the flowcharts--some day someone will try a series of options that you never tested.

Before we leave the area of debugging it would be worthwhile to discuss how to avoid building "bugs" to the user into your programs.

In the first place, you must assume that the basic law that "the series of events having the least probability of occurrence which cause the most undesirable results are certain to happen" has particular significance in the area of computer programs. In other words, "don't trust anybody" but set up your programs to check all input data for errors wherever possible. For instance, if you have an index of a computed GO TO as an input constant and it may assume the values one through six, some day it will be entered as ten through sixty or one hundred through six hundred, etc., if for no other reason than it was keypunched incorrectly. To save much machine time and grief on the part of the user, it is thus worth taking the time to have your program check all input data for "reasonableness" if possible. When an error is found, only that particular case may be bypassed from execution, or all remaining cases may also be bypassed (but all input data should be tested and all errors listed for the user, whether all cases are executed or not).

The easiest way to test input data is with a series of logical IF statements. One way may be as follows. Assume you have 83 separate input numbers.

```
      DIMENSION  XHI(83), XLØ(83), NAME(83), XINPUT(83), ISET(83)
      DATA  XHI, XLØ, NAME  /...../
      EQUIVALENCE  (XINPUT(1),...),  (XINPUT(2),...),...
   50 IX = 0
C     READ INPUT DATA HERE
      DØ 100  I = 1, 83
      ISET(I) = 0
      IF((XINPUT(I) .GT. XHI(I)) .ØR. (XINPUT(I) .LT. XLØ(I)))  ISET(I) = 1
  100 IX = IX + ISET(I)
      IF(IX .EQ. 0)  GØ TØ 300
      DØ 200  I = 1, 83
      IF(ISET(I) .EQ. 1)  WRITE (6,10) NAME(I), XINPUT(I), XHI(I), XLØ(I)
  200 CØNTINUE
      GØ TØ 50
   10 FØRMAT  (15HOINPUTˌERRORˌ-ˌ5X,A6,3Hˌ=ˌE15.5,10X,
     1 19HPERMITTEDˌRANGEˌISˌ,E15.5,4HˌTØˌ,E15.5)
  300     .
          .
          .
          .
```

Note in the above example we first initialized three arrays in a DATA statement - XHI, the high limits on each of the 83 input variables; XLΘ, the low limits on each of the 83 input variables; and NAME, the names of the 83 input variables. We then set up an EQUIVALENCE statement so that all real input variables are ordered in the array XINPUT (integer input variables may be stored in the array XINPUT by using arithmetic statements to equate them to the appropriate XINPUT after the input is read in). After the 83 input numbers are read in, they are each tested to assure that they are within the appropriate range XHI to XLΘ (if they are not, the index ISET is set equal to 1). If there are no errors, control is transferred to statement 300 and the remainder of the program is executed, if there are one or more errors, all of the errors are written out and the next set of input data are read.

A further aid to the user is to set up all input data so that it is easy to keypunch and so that the keypunched cards facillitate manual checking for errors. It is thus a good idea to end all data fields in columns 10, 20, 30, 40,... (which are defined quite clearly on 80-80 forms) to facillitate ease of data preparation.

One additional item in this area--it is a good idea to make the input as simple as possible. Let the program itself manipulate the data wherever possible and not the user. The greatest benefits from the use of the computer probably come from simplification in its use.

INDEX

DISTRIBUTION

No. of
Copies

1 Office of the Secretary of Defense

Advanced Research Projects Agency
Washington, D. C. 20301
(Attn.: Dr. Robert Taylor)

1 U. S. Atomic Energy Commission

Albuquerque Operations Office
P. O. Box 5400
Albuquerque, New Mexico 87115
(Attn.: L. P. Grise)

1 U. S. Atomic Energy Commission

Chicago Operations Office
9800 South Cass Avenue
Argonne, Illinois 60439
(Attn.: George H. Lee)

1 U. S. Atomic Energy Commission

AEC Library
Mail Station G-017
Washington, D. C. 20545

1 U. S. Atomic Energy Commission

New York Operations Office
376 Hudson Street
New York, New York 10014
(Attn.: Reports Librarian)

1 U. S. Atomic Energy Commission

Office of Assistant General Counsel for Patents
Washington, D. C. 20545
(Attn.: Roland A. Anderson)

1 U. S. AEC Scientific Representative

American Embassy
APO New York, New York 09777

1 <u>Aerojet-General Corporation</u>

San Ramon Plant
P. O. Box 78
San Ramon, California 94583
(Attn.: Document Custodian)

1 <u>Aerospace Corporation, San Bernardino (AF)</u>

San Bernardino Operations
P. O. Box 1308
San Bernardino, California 92402
(Attn.: SBO Library)

1 <u>Air Force Aero Propulsion Laboratory</u>

Wright-Patterson Air Force Base, Ohio 45433
(Attn.: APE/STINFO Office)

1 <u>Systems Engineering Group (RTD)</u>

Wright-Patterson Air Force Base, Ohio 45433
(Attn.: A. Daniels, SEPIR)

1 <u>Air Force Flight Dynamics Laboratory</u>

FDCL
Wright-Patterson AFB, Ohio 45433
(Attn.: Dr. Paul Polishuk)

1 <u>Air Force Institute of Technology</u>

Library
Air University, USAF
Wright-Patterson Air Force Base, Ohio 45433
(Attn.: AFIT-LIB)

1 <u>USAF School of Aerospace Medicine</u>

Aeromedical Library (SMSDL)
Bldg. 155
Brooks Air Force Base, Texas 78235
(Attn.: Chief Librarian)

1 <u>Air Force Weapons Laboratory (WLIL)</u>

Kirtland Air Force Base, New Mexico 87117
(Attn.: M. F. Canova)

2 Ames Laboratory

Iowa State University
Ames, Iowa 50010
(Attn.: Dr. F. H. Spedding)

1 Argonne Cancer Research Hospital

950 E. 59th Street
Chicago, Illinois 60637
(Attn.: Frances J. Skozen)

10 Argonne National Laboratory

Library Services Department
Report Section, Bldg. 203, Rm. CE-125
9700 South Cass Avenue
Argonne, Illinois 60439

8 Commanding Officer

Aberdeen Proving Ground, Maryland 21005
(Attn.: Technical Library, Bldg. 313)

1 Institute for Exploratory Research

U. S. Army Electronics Command
Fort Monmouth, New Jersey 07703
(Attn.: AMSEL-XL-S, Dr. W. J. Ramm)

1 Director

U. S. Army Engineer Nuclear Cratering Group
P. O. Box 808
Livermore, California 94550

1 U. S. Army Foreign Science and Technology Center

Munitions Building
Washington, D. C. 20315
(Attn.: AMXST-SD-TD)

2 Commanding Officer

Harry Diamond Laboratories
Washington, D. C. 20438
(Attn.: Stuart M. Marcus)

1 Medical Field Service School

Brooke Army Medical Center
Fort Sam Houston, Texas 78234
(Attn.: Stimson Library)

1 Commanding Officer

U. S. Army Medical Research Unit - Presidio
San Francisco, California 94129
(Attn.: Librarian, Letterman General Hospital)

1 Commanding Officer

U. S. Army Nuclear Defense Laboratory
Edgewood Arsenal, Maryland 21010
(Attn.: Librarian)

1 Commanding Officer

Picatinny Arsenal
Dover, New Jersey 07801
(Attn.: Technical Information Library)

1 U. S. Army Research Office-Durham

Box CM, Duke Station
Durham, North Carolina 27706
(Attn.: CRDARD-IP)

1 Division of Nuclear Medicine

Walter Reed Army Institute of Research
Walter Reed Army Medical Center
Washington, D. C. 20012

1 Atomic Bomb Casualty Commission

U. S. Marine Corps Air Station
FPO San Francisco, California 96664
(Attn.: Librarian)

1 Atomic Power Development Associates, Inc.

1911 First Street
Detroit, Michigan 48226
(Attn.: Document Librarian, for AT(11-1)-476,-865)

4 Atomics International

P. O. Box 309
Canoga Park, California 91304
(Attn.: Library)

1 The Babcock and Wilcox Company

Atomic Energy Division
P. O. Box 1260
Lynchburg, Virginia 24505
(Attn.: Information Services)

2 Battelle Memorial Institute

Columbus Laboratories
505 King Avenue
Columbus, Ohio 43201
(Attn.: John E. Davis)

6 Battelle Memorial Institute

Pacific Northwest Laboratory
P. O. Box 999
Richland, Washington 99352
(Attn.: Technical Information Section)

4 Westinghouse Electric Corporation

Bettis Atomic Power Laboratory
P. O. Box 79
West Mifflin, Pennsylvania 15122
(Attn.: Virginia Sternberg, Librarian)

2 Brookhaven National Laboratory

Information Division
Upton, Long Island, New York 11973
(Attn.: Research Library)

1 Clarkson College of Technology

Department of Physics
Potsdam, New York 13676
(Attn.: Dr. Richard Madey)

1 Columbia University

Pegram Nuclear Physics Laboratories
538 West 120th Street
New York, New York 10027
(Attn.: Dr. W. W. Havens, Jr.)

1 Combustion Engineering, Inc.

Nuclear Division
Prospect Hill Road
Windsor, Connecticut 06095
(Attn.: Nuclear Division Library)

1 Combustion Engineering, Inc.

Naval Reactors Division
P. O. Box 400
Windsor, Connecticut 06095
(Attn.: Document Custodian)

1 Chief, Livermore Division

Field Command
Defense Atomic Support Agency
Lawrence Radiation Laboratory
P. O. Box 808
Livermore, California 94550

1 Armed Forces Radiobiology Research Institute

Defense Atomic Support Agency
NNMC
Bethesda, Maryland 20014
(Attn.: Library)

2 E. I. du Pont de Nemours and Company

Savannah River Laboratory
Technical Information Service-773A
Aiken, South Carolina 29801

1 E. I. du Pont de Nemours and Company

Explosives Department
Atomic Energy Division
Wilmington, Delaware 19898
(Attn.: Document Custodian)

1 EG&G, Inc.

P. O. Box 8346
Albuquerque, New Mexico 87108
(Attn.: J. Frinkman or William J. Jones)

1 EG&G, Inc.

P. O. Box 1912
Las Vegas, Nevada 89101
(Attn.: Librarian)

1 Environmental Research Corporation

P. O. Box 1061
Alexandra, Virginia 22313
(Attn.: Francie G. Binion)

2 Air Resources Field Research Office

Environmental Science Services Administration
P. O. Box 2136
Las Vegas, Nevada 89101
(Attn.: P. W. Allen)

1 L. Machta, Director

Air Resources Laboratory
Environmental Science Services Administration
8060 13th Street
Silver Spring, Maryland, 20910

2 Federal Aviation Agency

Information Retrieval Branch, HQ-630
Washington, D. C. 20553

1 Commanding Officer

Pitman-Dunn Laboratories
Frankford Arsenal
Philadelphia, Pennsylvania 19137
(Attn.: S. Berk, L8400, Bldg. 312)

1 Fundamental Methods Association

31 Union Square West
New York, New York 10003
(Attn.: Dr. Carl N. Klahr)

2 General Atomic Division

General Dynamics Corporation
P. O. Box 1111
San Diego, California 92112
(Attn.: Chief, Tech. Information Services)

1 General Dynamics/Fort Worth

P. O. Box 748
Fort Worth, Texas 76101
(Attn.: Keith G. Brown or B. S. Fain)

2 General Electric Company

Nuclear Materials and Propulsion Operation
P. O. Box 132
Cincinnati, Ohio 45215
(Attn.: J. W. Stephenson)

2 General Electric Company

Atomic Power Equipment Department
P. O. Box 1131
San Jose, California 95108
(Attn.: Alleen Thompson)

1 U. S. Geological Survey

Building 25, Denver Federal Center
Denver, Colorado 80225
(Attn.: Library)

1 U. S. Geological Survey

Room 1033, General Services Administration Building
Washington, D. C. 20242
(Attn.: Librarian)

1 Goodyear Atomic Corporation

P. O. Box 628
Piketon, Ohio 45661
(Attn.: Department 423)

1 Hughes Aircraft Company

P. O. Box 3310
Building 600, Mail Station F-131
Fullerton, California 92634
(Attn.: Dr. A. M. Liebschutz)

1 IIT Research Institute

10 West 35th Street
Chicago, Illinois 60616
(Attn.: Document Library)

1 Isotopes, Inc.

Palo Alto Laboratories
4062 Fabian Street
Palo Alto, California 94303

1 Jet Propulsion Laboratory

California Institute of Technology
4800 Oak Grove Drive
Pasadena, California 91103
(Attn.: N. E. Devereus, Library Supv.)

1 Johns Hopkins University

Applied Physics Laboratory
8621 Georgia Avenue
Silver Spring, Maryland 20910
(Attn.: Boris W. Kuvshinoff)

3 Knolls Atomic Power Laboratory

P. O. Box 1072
Schenectady, New York 12301
(Attn.: Document Librarian)

1 University of California

Lawrence Radiation Laboratory
Technical Information Division
Berkeley, California 94720
(Attn.: Dr. R. K. Wakerling)

4 University of California

Lawrence Radiation Laboratory
P. O. Box 808
Livermore, California 94550
(Attn.: Technical Information Dept.)

1 Lockheed-Georgia Company

Division of Lockheed Aircraft Corporation
Marietta, Georgia 30060
(Attn.: Charles K. Bauer, Manager, Scientific and Technical
Information Department)

2 Los Alamos Scientific Laboratory

P. O. Box 1663
Los Alamos, New Mexico 87544
(Attn.: Report Librarian)

5 Lovelace Foundation

4800 Gibson Boulevard
Albuquerque, New Mexico 87108
(Attn.: Dr. Clayton S. White, Director of Research)

1 Martin-Marietta Corporation

Martin Company
Nuclear Products
P. O. Box 5042
Middle River, Maryland 21220
(Attn.: AEC Document Custodian)

1 Monsanto Research Corporation

Mound Laboratory
P. O. Box 32
Miamisburg, Ohio 45342
(Attn.: Library)

1 National Aeronautics and Space Administration

John F. Kennedy Space Center
Kennedy Space Center, Florida 32899
(Attn.: Mrs. L. B. Russell, Librarian)

1 National Aeronautics and Space Administration

Lewis Research Center
21000 Brookpark Road
Cleveland, Ohio 44135
(Attn.: Dorothy Morris)

1 Scientific and Technical Information Facility

P. O. Box 33
College Park, Maryland 20740
(Attn.: Acquisitions Branch, S-AK/DL)

2 National Aeronautics and Space Administration

(USS-10)
Washington, D. C. 20546
(Attn.: Document Control Officer)

2 National Accelerator Lab.

1301 W. 22nd Street
Oak Brook, Ill. 60521
Attn.: Librarian

1 National Bureau of Standards

Room E-01 Administration Building
Washington, D. C. 20234
(Attn.: Library)

1 Library

National Institutes of Health
Bldg. 10, Room 5N115
Bethesda, Maryland 20014
(Attn.: Acquisitions Unit)

1 National Lead Company of Ohio

Post Office Box 39158
Cincinnati, Ohio 45239
(Attn.: Reports Library)

4 Idaho Nuclear Corporation

NRTS Technical Library
P. O. Box 1845
Idaho Falls, Idaho 83401

2 David Taylor Model Basin

Applied Mathematics Laboratory
Carderock, Maryland 20007
(Attn.: Code 800)

1 Naval Facilities Engineering Command

Department of the Navy
Washington, D. C. 20390
(Attn.: Code 042)

1 Office of Naval Research Branch Office

Box 39, FPO
New York, New York 09510

2 Commander

U. S. Naval Ordnance Laboratory
White Oak
Silver Spring, Maryland 20910
(Attn.: Library)

1 Commander, Naval Ordnance Systems Command

Department of the Navy
Washington, D. C. 20360
Attn.: Code ORD-0332

1 U. S. Naval Postgraduate School

Monterey, California 93940
(Attn.: George R. Luckett, Director of Libraries)

2 Commanding Officer and Director

U. S. Naval Radiological Defense Laboratory
San Francisco, California 94135
(Attn.: T. J. Mathews)

1 Naval Ship Systems Command Headquarters

Navships 08
Navy Department
Washington, D. C. 20360
(Attn.: Irene P. White)

1 New York University

AEC Computing and Applied Mathematics Center
251 Mercer Street
New York, New York 10012
(Attn.: Director)

1 Environmental Medicine Library

New York University Medical Center
Long Meadow Road, Sterling Forest
Tuxedo, New York 10987
(Attn.: L. P. Zipin, Research Division)

1 NRA, Inc.

3501 Queens Boulevard
Long Island City, New York 11101
(Attn.: Seymour L. Goldblatt)

1 Nuclear Materials and Equipment Corporation

609 North Warren Avenue
Apollo, Pennsylvania 15613
(Attn.: Library)

1 Nuclear Technology Corporation

116 Main Street
White Plains, New York 10601

5 Union Carbide Corporation

Nuclear Division
X-10 Laboratory Records Department
P. O. Box X
Oak Ridge, Tennessee 37830

1 Oceanographic Services, Inc.

5375 Overpass Road
Santa Barbara, California 93105
(Attn.: N. R. Wallace)

1 Oregon State University

Corvallis, Oregon 97331
(Attn.: Arvid T. Lonseth)

1 Picker X-Ray Corporation

Waite Manufacturing Division, Inc.
1020 London Road
Cleveland, Ohio 44110
(Attn.: Research Center Library)

1 Cyclotron Laboratory

Princeton University
Department of Physics
Princeton, New Jersey 08540
(Attn.: Professor Rubby Sherr)

1 Officer in Charge

U. S. Public Health Service
Southeastern Radiological Health Facility
P. O. Box 61
Montgomery, Alabama 36101

1 Officer in Charge

U. S. Public Health Service
Northeastern Radiological Health Laboratory
109 Holton Street
Winchester, Massachusetts 01890

1 Puerto Rico Water Resources Authority

P. O. Box 4267
San Juan, Puerto Rico 00905
(Attn.: Executive Director)

1 Purdue University

Department of Nuclear Engineering
Lafayette, Indiana 47907
(Attn.: Prof. Alexander Sesonske)

1 Radioptics, Inc.

10 Du Pont Street
Plainview, Long Island, New York 11803

1 Rand Corporation

1700 Main Street
Santa Monica, California 90406
(Attn.: Dr. Mario L. Juncosa)

1 Reactive Metals, Inc.

Extrusion Plant
P. O. Box 579
Ashtabula, Ohio 44004
(Attn.: P. L. Bean, AEC Contract Manager)

1 James R. Crockett, General Manager

Reynolds Electrical and Engineering Company, Inc.
P. O. Box 1360
Las Vegas, Nevada 89101
(Attn.: Management Engineering Dept.)

1 Rice University

Houston, Texas 77001
(Attn.: Walter Orvedahl)

1 Sandia Corporation

P. O. Box 5800
Albuquerque, New Mexico 87115
(Attn.: Technical Library)

1 Sandia Corporation Livermore Laboratory

P. O. Box 969
Livermore, California 94550
(Attn.: Technical Library)

1 Southwest Research Institute

8500 Culebra Road
San Antonio, Texas 78206
(Attn.: Librarian)

1 Stanford University

Stanford Linear Accelerator Center
Stanford, California 94305
(Attn.: Librarian)

1 The Library

State University of New York at Binghamton
Binghamton, New York 13901

2 Stevens Institute of Technology

Department of Physics
Hoboken, New Jersey 07030
(Attn.: Dr. Winston Bostick)

1 Tennessee Valley Authority

Chattanooga, Tennessee 37401
(Attn.: Harold L. Falkenberry)

1 Texas Nuclear Corporation

Box 9267
Austin, Texas 78756
(Attn.: Dr. John B. Ashe, Director of Research)

1 Todd Shipyards Corporation

Nuclear Division
P. O. Box 1600
Galveston, Texas 77550
(Attn.: Central File)

1 VESIAC

University of Michigan
P. O. Box 618
Ann Arbor, Michigan 48107

1 University of Puerto Rico

Puerto Rico Nuclear Center
College Station
Mayaguez, Puerto Rico 00708

1 University of Rochester

Department of Physics and Astronomy
Rochester, New York 14627
(Attn.: Dr. M. F. Kaplon)

1 Department of Physics

University of Washington
Seattle, Washington 98105
(Attn.: Prof. Ronald Geballe)

1 Virginia Associated Research Center

12070 Jefferson Avenue
Newport News, Virginia 23606

1 Washington University

St. Louis, Missouri 63130
(Attn.: Leon Cooper)

1 TRACOR, Inc.

6500 Tracor Lane
Austin, Texas 78721

2 Union Carbide Corporation

Nuclear Division
ORGDP Records Department
P. O. Box P
Oak Ridge, Tennessee 37830

1 University of Chicago

5630 Ellis Avenue
Chicago, Illinois 60637
(Attn.: Richard Miller)

1 University of Maryland

College Park, Maryland 20742
(Attn.: Dr. B. E. Hubbard)

2 Westinghouse Electric Corporation

Atomic Power Division
P. O. Box 355
Pittsburgh, Pennsylvania 15230
(Attn.: Document Custodian)

1 Westinghouse Electric Corporation

Astronuclear Laboratory
P. O. Box 10864
Pittsburgh, Pennsylvania 15236
(Attn.: Florence M. McKenna)

1 Director

U. S. Army Engineer Waterways Experiment Station
P. O. Box 631
Vicksburgh, Mississippi 39180
(Attn.: Library)

1 The Library

U. S. Geological Survey
Branch of Astrogeology
601 East Cedar Avenue
Flagstaff, Arizona 86002
(Attn.: Librarian)

1 Officer in Charge

U. S. Public Health Service
Southwestern Radiological Health Laboratory
P. O. Box 684
Las Vegas, Nevada 89101

1 United Nuclear Corporation

Research and Engineering Center
Grasslands Road
Elmsford, New York 10523
Attn.: Library

74 AEC Division of Technical Information Extension

25 Clearinghouse for Federal Scientific and Technical Information

7 AEC-RL

J. P. Derouin
W. Devine, Jr.
J. E. Goodwin (2)
P. M. Midkiff
R. L. Plum
C. R. Qualheim

9 Atlantic Richfield Hanford Co.

J. W. Fillmore
G. L. Gurwell
M. K. Harmon
R. J. Kofoed
B. J. McMurray
H. P. Shaw
R. E. Smith
R. J. Sloat
R. E. Tomlinson

14 Battelle Memorial Institute Pacific Northwest Laboratories

R. D. Benham
G. J. Busselman
E. D. Clayton
R. Y. Dean
J. L. Deichman
D. E. Deonigi
G. E. Driver
E. A. Eschbach
P. L. Hofmann
J. L. Jaech
D. D. Lanning
R. C. Liikala
C. W. Lindenmeier
W. I. Neef

6 Computer Sciences Corporation Northwest Operations

Z. E. Carey
J. E. Farmer
R. J. Gurth
J. D. Orton
G. L. Otterbein
J. L. Peterson

4 Donald W. Douglas Laboratories

H. M. Busey
R. Cooper
J. Greenborg
W. E. Matheson

35 Douglas United Nuclear, Inc.

T. W. Ambrose
R. S. Bell
G. M. Blanchard
R. E. Dunn
G. C. Fullmer
P. D. Gross (20)
R. W. Hallet, Jr.
C. D. Harrington
R. T. Jessen
S. Koepcke
C. W. Kuhlman
W. M. Mathis
J. S. McMahon
R. Nilson
J. W. Riches
O. C. Schroeder

9 Hanford Engineering Services

R. D. Duncan
W. S. Graves
A. J. Hutzelman
G. Kligfield
R. Lysher
M. O. Rothwell
G. Salzano
E. E. Smith
C. L. Taylor

1 Sandvik Special Metals Corporation

S. M. Graves

2 Washington State University

O. W. Rechard (2)