ccm-73-03

FORTRAN Programming on the CDC 3300 Under OS-3

Notes to accompany the OSU videotape series

OSU
COMPUTER CENTER
Oregon State University
Corvallis, Oregon 97331

F O R T R A N   P R O G R A M M I N G
O N   T H E   C D C   3 3 0 0
U N D E R   O S - 3

Notes to accompany

the OSU videotape series

ccm 73-03

Computer Center

Oregon State University

Corvallis, Oregon 97331

July, 1973

PREFACE

The current OSU videotape series on FORTRAN consists of twenty half hour tapes. Together they comprise a basic introduction to the FORTRAN language. The tapes were completed in October of 1971. This manual is intended for use in conjunction with the videotapes. It can also be used, however, as a supporting text for any FORTRAN course which uses the CDC 3300 at Oregon State.

TABLE OF CONTENTS

1 - INTRODUCTION TO COMPUTERS

This series of lectures will introduce you to FORTRAN - the widely used computer programming language designed to give solutions to problems expressible in terms of the symbols and operations of algebra.

You will find it helpful to have some general knowledge of what happens inside a computer before you start your study of the FORTRAN language. So we will begin with a very brief history of computers and then proceed with a general description of a computer system.

1-1 History

The history of electronic digital computers is quite a short one. Major theories of computing and of computer design were developed in the 1930's and 40's. The first working computers were also built in this period. The modern era of computers began about 1950 with the advent of the first computers designed for commercial use. In the twenty plus years since those first commercial models, the computer industry has grown very rapidly. Improvements in computer hardware have occurred frequently with the aid of numerous developments in electronics, electromagnetics and microminiaturization. There have also been many advances in software technology, including the development of several problem-oriented programming languages designed to solve particular classes of problems of which FORTRAN was the first major algebraic-like language. Also, extremely sophisticated operating systems were developed to allow simultaneous execution of several unrelated programs.

1-2 Power

Just what features do computers have that make them so useful and powerful?

FIRST: SPEED - computers can do hundreds of thousands of operations every second.

SECOND: VERSATILITY - computers can work on many different, unrelated problems in areas ranging from scientific research to accounting to classroom teaching to control of manufacturing processes to information retrieval in a library. The list of areas in which computers can be used has been growing steadily and there is no end in sight.

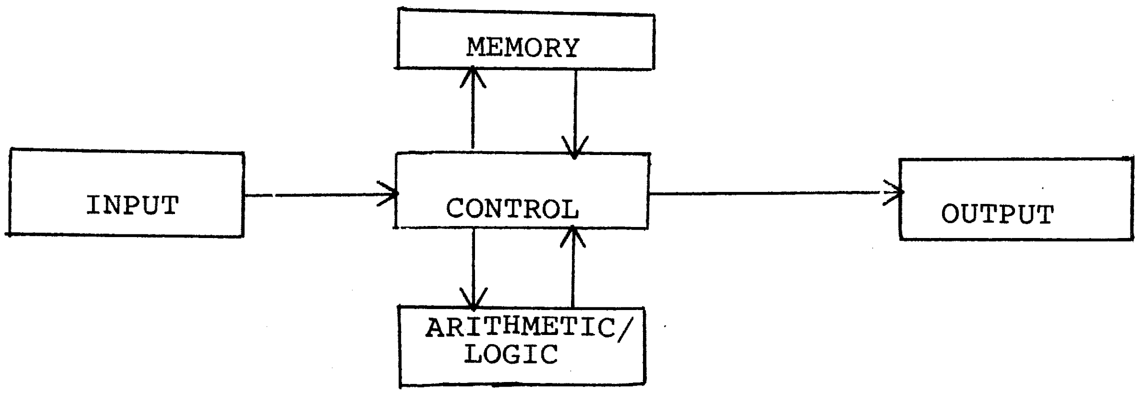
THIRD: SIMPLICITY - through languages like FORTRAN computers are easy to use, requiring a minimal amount of training.

FOURTH: PRECISION - computers do arithmetic operations on many digits at a time, always obtaining the same answer. This contrasts sharply with normal human execution of all but the simplest arithmetic.

FIFTH: RELIABILITY - computers are nearly error-free. Not that there are no errors associated with computers, but such errors are 99% human errors and are not due to machine failure.

1-3 Computer System

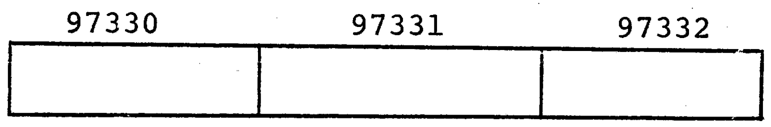
A computer system consists of five interconnected components which we can represent symbolically as follows:



The arrows represent the interconnections and indicate the flow of information.

CONTROL - usually called the Central Processing Unit or CPU, for short, it controls the flow of instructions and the movement of information.

MEMORY - the array of locations in which information is stored. Every location is given an address. A particular piece of information is located at a particular address. The MEMORY is somewhat analogous to a postal zip code sorter, which might be represented as follows:



The sorter consists of boxes (locations) with given zip codes (addresses) in which mail (information) is stored. The MEMORY unit is often called CORE or CORE STORAGE since the information is often stored by magnetizing small ring-shaped magnetic cores; these are placed in a lattice structure with each core at the intersection of several wires. When current is passed down certain wires, the core is magnetized in a clockwise direction and when the current is reversed, the core becomes magnetized in a counter-clockwise direction. This gives rise to a natural two state system and is the primary reason for representing all information in terms of binary numbers, a zero being represented by one of the directions, and a one by the other direction.

ARITHMETIC/LOGIC - the set of special locations, usually called registers, in which arithmetic is done and logical comparisons are made by the use of special electronic circuits.

INPUT - any of several devices which pass information from external sources into the computer's memory. The major input devices are:

CARD READER

TELETYPE (and other on-line terminals)

MAGNETIC TAPE

MAGNETIC DISK

In addition, several other input devices are used at various times.

OUTPUT - any of several devices which pass information from the computer's memory to external sources. The major output devices are:

LINE PRINTER

CARD PUNCH

TELETYPE (and other on-line terminals)

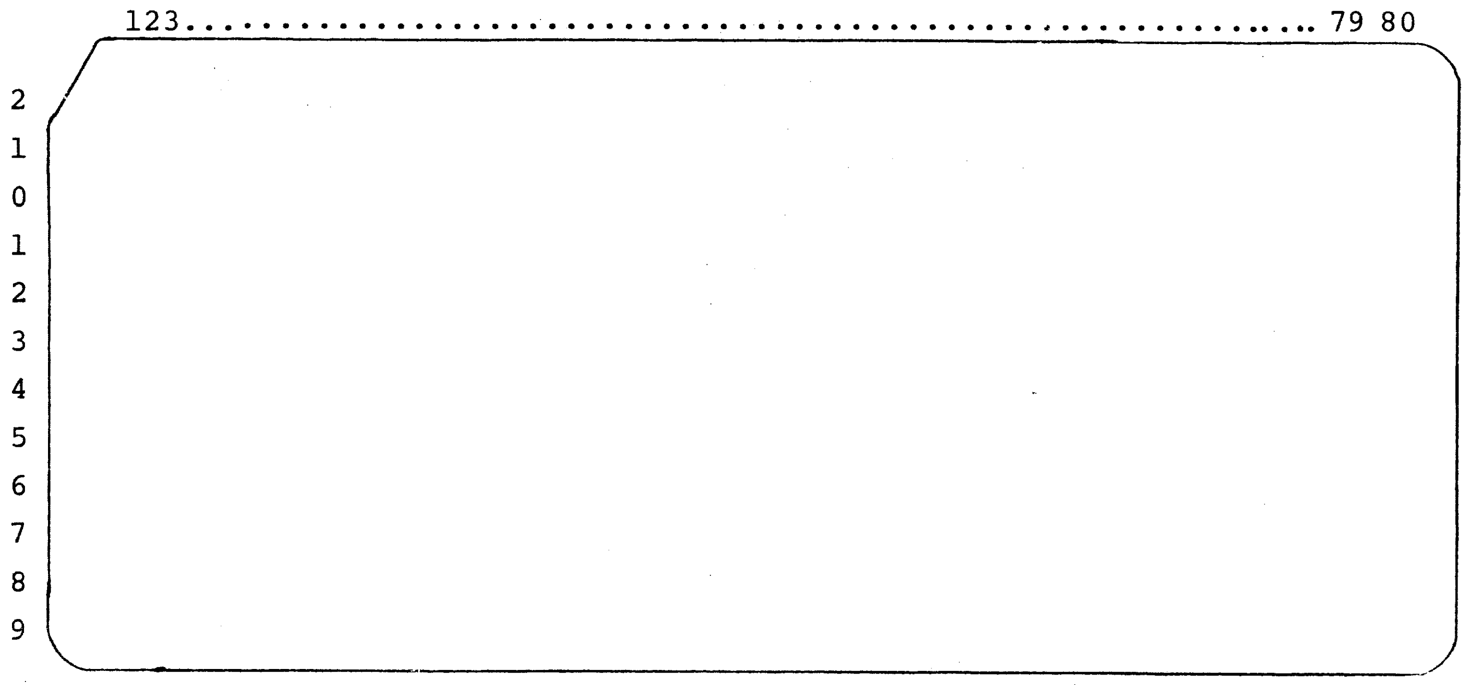
MAGNETIC TAPE

MAGNETIC DISK

1-4 External Sources

The external sources mentioned above are man-machine interfaces such as punched cards, punched paper tape, line printer paper, the keyboard and paper display of a teletype or the keyboard and screen of a CRT (cathode ray tube) display.

Punched cards play a particularly important role as external input sources. They were developed by Herman Hollerith to aid in the tabulation of data for the 1890 U.S. Census. The code he developed is still used today. The Hollerith card consists of 12 rows and 80 columns arranged like this:



Each of the characters of the computer alphabet consists of a unique combination of punches usually in 1, 2, or 3 rows of a single column. Thus there are a maximum of 80 characters per card. The numbers 0 through 9 are obtained with a single punch in the equivalent row. Combining a 12,11 or 0 punch with the punches in 1 to 9 gives the English alphabet. A is 12,1; B is 12,2; etc.

A	12,1
B	12,2
⋮	
I	12,9
J	11,1
⋮	
R	11,9
S	0,2
⋮	
Z	0,9

1-5 Batch Processing

Programs and data are punched on cards by the use of a key-punch and are submitted to the computer for batch processing in which the deck of cards is read into the computer's memory and the instructions of the program are executed. Any data cards are read into the memory when required by the program. This means that only one job at a time is processed.

1-6 On-Line Time-Sharing: OS-3

In recent years the use of teletypes and other keyboard devices has allowed on-line operation in which many jobs are processed simultaneously by sharing processing time among several users connected by wire or telephone line to the computer. The punching of cards is not required when teletypes are used. The information normally punched is sent directly to the computer and stored on magnetic disk for later use. Here at Oregon State we have an excellent time-sharing system - OS-3 for Oregon State Open Shop Operating System. This system typically handles 30 to 40 jobs simultaneously. There are several informative videotapes available dealing specifically with various aspects of OS-3. Questions concerning aspects of the system should be referred to Computer Center personnel.

1-7 Machine Language and Computer Programming

Every computer has a set of basic instructions it can execute; add, subtract, store, retrieve, move, compare, input and output, etc. This set of basic instructions forms the MACHINE LANGUAGE. Each instruction consists typically of a group of digits representing an operation and the address of a location in MEMORY, the contents of which is to be operated on. A computer program is a sequence of these basic machine language instructions specifically arranged to accomplish a given task. Computer programming is both an art and a science summarized in these six steps:

1. State the problem.
2. Formulate an algorithm for the solution of the problem.

   Algorithm - a complete, unambiguous procedure for the solution of a given problem in a finite number of steps - particular attention must be paid to starting, stopping and error checking.
3. Write a program based on the algorithm.
4. Enter the program into the computer and test execution on appropriate data.
5. Analyze the results.
6. If necessary, debug the program (that is, correct it) and repeat the test.

1-8 FORTRAN

Since the machine language is not easy to use without considerable training, we use the easier problem-oriented languages like FORTRAN whenever possible. FORTRAN was developed by IBM in 1955 and is an acronym for FORMULA TRANSLATION. It has undergone extensive modification since then. The current version, FORTRAN IV, has been standardized by the American National Standards Institute. The version available at Oregon State has many features not found in the standardized version and is missing some of the less useful standardized instructions. We will concern ourselves with the standard version whenever possible and will indicate any deviations in the OS-3 version when they first occur.

Since FORTRAN is not similar to machine language, the computer translates the FORTRAN program into an equivalent machine language program and then executes this translated program. Thus all FORTRAN programs must go through two phases.

1-9 Compilation and Execution

In phase one, the compilation phase, the computer's memory is loaded with a pre-written compiler program usually supplied by the computer manufacturer; at OSU we have systems programmers who write the compilers themselves. The FORTRAN source program is read as data and the output is in the form of a machine language object program. This program is usually stored temporarily in a work area on magnetic disk. During the second phase, the execution, this object program is loaded into memory and is executed with any data required being read from the appropriate input unit and the results being put on the selected output unit.

1-10 FORTRAN Alphabet

FORTRAN is an alphabetic language like English or Russian and unlike Chinese. There are 48 symbols in the alphabet: the 26 English letters A-Z, the ten digits 0-9, the 11 special characters + - * / , . () = ' $ and the blank or space, represented on a card with no holes punched. Since it is sometimes necessary to emphasize the existence of a space within a FORTRAN program, we will represent this character with ƀ or ␣.

1-11 Formulation of Statements

Whether you enter your program from cards or from teletype, the FORTRAN compiler treats each line as an 80 character record. When writing a program, the placing of the statements is very important. For a FORTRAN program follow these rules:

1. Put only one FORTRAN instruction on each record.
2. The body of the program must be in columns 7 through 72.
3. Columns 73 through 80 are ignored by the compiler but may be used for sequencing the statements to keep them in order.
4. Columns 1 through 5 may contain a statement number (restricted to the positive integers). This is used to label a statement which is referred to by another statement.
5. If column 1 contains a C, the entire record is treated as a comment and is not compiled.
6. If column 6 contains any character other than a blank or a zero, then columns 7-72 of that record are treated, not as a separate instruction, but as a continuation of the instruction on the previous record.

7. Blanks within a FORTRAN program are ignored with one exception which will be discussed later, e.g. READ is identical to R E A D.

1-12 Data Records

When the program requires data records, either from cards, teletype, or disk file, the above rules do <u>not</u> apply to the data.

For data records: 1) all 80 columns of a data record may be used and 2) if the data are numeric, blanks are treated as zeroes, while 3) if the data are alphabetic, blanks are treated as spaces. In either case blanks in a data record are not ignored unless they are specifically skipped. If data exist on magnetic tape or disk, there may be more than 80 columns in one record. These records must usually be handled by special instructions which we will not discuss further.

2 - BIT STRUCTURE

2-1 Binary Nature of Computers

As we have indicated, the hardware of a computer system consists of electronic and magnetic devices. Such devices are bi-stable or two state devices, that is off-on, high voltage-low voltage, clockwise magnetized-counterclockwise magnetized, etc. Because of the two state nature of these devices, the basic unit of information in the computer is not the familiar decimal digit but the binary digit or bit, for short. The two states of the hardware devices are equivalent to the two binary digits 0 and 1 from which we can construct this binary number table:

Binary	Decimal
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

2-2 Decimal and Binary Numbers

To determine the relationship between decimal and binary numbers we observe that $2503 = 2\text{x}10^3 + 5\text{x}10^2 + 0\text{x}10^1 + 3\text{x}10^0$. Thus, in the base 10 system the column position tells the power of 10 that the decimal digit in that column is multiplied by - first column to the left of the decimal point x $10^0 = 1$, second column to the left of the decimal point x $10^1 = 10$, third column x $10^2 = 100$, etc. All number systems work similarly so that in the base 2 system the column position tells the power of 2 that the binary digit in that column is multiplied by: Digit in the first column to the left of the binary point x $2^0 = 1$, digit in the second column to the left of the binary point x $2^1 = 2$, digit in the third column to the left of the binary point x $2^2 = 4$, digit in the fourth column to the left of the binary point x $2^3 = 8$, hence $1101_2 = 1x2^3 + 1x2^2 + 0x2^1 + 1x2^0 = 8 + 4 + 1 = 13$. Also, 100 111 000 111_2 (where the subscript indicates the base) = $1x2^{11} + 1x2^8 + 1x2^7 + 1x2^6 + 1x2^2 + 1x2^1 + 1x2^0 = 2048 + 256 + 128 + 64 + 4 + 2 + 1 = 2503_{10}$. It is clear that arithmetic in base 2 would be quite tedious for us since we have been programmed through life to work with the decimal system. Still, we need to refer to the binary representation within the computer from time to time. Fortunately we can use the less tedious octal (base 8) system for this purpose. Note the interesting relationship between the first nine numbers in the octal and binary systems.

Digits of base 8	Digits of base 2
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111
10	001000

Thus we see that we can generate each of the octal digits 0-7 from sets of three binary digits. One octal digit = three bits because $8 = 2^3$ so 100 111 000 $111_2 = 4707_8 = 4x8^3 + 7x8^2 + 0x8^1 + 7x8^0 = 2048 + 448 + 7 = 2503_{10}$.

2-3 Octal Numbers and The CDC 3300

The CDC 3300 uses the octal number system as a shorthand form for representing the binary system. From FORTRAN the CDC 3300 has 32768_{10} addressable locations with addresses from 00000 to 32767. In octal these are numbered from 00000_8 to 77777_8. We thus see that five octal (i.e., 15 binary) digits are required for each address in the computer's memory. Earlier we indicated that the form of a typical machine language instruction was a group of digits containing an operation and the address of a location, the contents of which were to be operated on. To get all the operation codes needed in the 3300's machine language, the computer requires three octal (i.e., 9 binary) digits. Hence eight octal (24 binary) digits are used to form a machine language instruction and thus 24 bits is the basic word size of the CDC 3300. Each of the 32768 addressable locations consists of one 24-bit word.

We need some knowledge of the binary representation in the computer to understand the causes for the limitations on the size of numbers we can work with.

2-4 Constants from Programmer's Point of View

What might be called the "nouns" of FORTRAN consist of constants and variables. Constants are fixed numeric quantities assigned by the programmer and stored in chosen memory locations by the compiler. Thus they are essentially the "contents" of memory locations. There are several types of constants, but for now we shall work with the two most important: integer (or fixed point) constants and real (or floating point) constants. The difference between the two types as seen by the programmer is determined solely by whether or not the constant has a decimal point. An _integer constant_ is a set of decimal digits _without_ a _decimal_ point. A _real constant_ is a set of decimal digits _with_ a _decimal point_. The (real) number may or may not have a decimal fraction part. Examples of integer constants: 37, -6, 0, 10, 3, 999002 and of real constants: 2.6, 18.471, -9.27, 3.1416, 4., 0.0, -8.

2-5 Constants as Stored in Memory

The difference from the computer's point of view is the form in which the numbers are stored internally. On the 3300 under OS-3 every integer constant uses one word of storage. The first bit is the sign (0 = +, 1 = -) and the remaining 23 bits form the number in binary; e.g. 2503 is an integer constant and is stored in binary as $000\ 000\ 000\ 000\ 100\ 111\ 000\ 111_2$; the octal equivalent is 00004707_8.

The largest integer constant is obtained when the sign is 0 and there are 23 1-bits. This number is $2^{23}-1 = 8388607$. Thus all integers on the CDC 3300 must be between -8388607 and 8388607, inclusive. This range changes from computer to computer depending on the particular computer's bit structure.

A real or floating point constant is stored in the computer in a modified form of binary scientific notation. Two consecutive 24-bit words are used to store one real constant. The first bit contains the sign of the number (0 for plus, 1 for minus), the next 11 bits contain a base 2 exponent which has been biased by having 2000_8 ($10\ 000\ 000\ 000_2$) added to it to allow for both negative and positive exponents. The remaining 36 bits form the normalized mantissa, a binary fraction less than one.

In the decimal system $281.37 = 2x10^2 + 8x10^1 + 1x10^0 + 3x10^{-1} + 7x10^{-2}$ which in base 10 normalized scientific notation would be $.28137x10^3$. In the binary system we might have $101.11_2 = 1x2^2 + 0x2^1 + 1x2^0 + 1x2^{-1} + 1x2^{-2} = 4 + 1 + 1/2 + 1/4 = 5.75$ and in binary normalized scientific notation this is $.10111x2^3$. This number would be stored as 2003560000000000_8 in the CDC 3300's memory. The 2003_8 is the biased exponent, $2000_8 + 3_8$; the 56_8 is 101110_2, the fractional part of the number above.

The largest possible exponent obtainable from the 11 biased digits is $2^{10} - 1 = 1023$ and the smallest exponent is -1023. This means that the range of the real numbers the CDC 3300 can handle is between -2^{1023} and $+2^{1023}$ or approximately -10^{308} to $+10^{308}$.

The 36 bit mantissa can contain a maximum fraction of 36 1-bits. This number is 2^{36-1}, the 11 decimal digit number .68719476735. The mantissas of all real numbers are formed of these 36 bits; therefore every real number stored in the memory of the 3300 is precise to either 10 or 11 decimal digits, 11 digits for decimal fractions .10000000000 to .68719476735 and 10 digits for larger fractions. The range and precision depend entirely on the bit structure of the computer in question.

SUMMARY OF DEFINITIONS

Slide 1.

INTEGER CONSTANTS are programmed without a decimal point and stored in one 24 bit word, have an allowed range from -8388607 to 8388607 and may have up to 7 decimal digits of precision.

Slide 2.

REAL CONSTANTS are programmed with a decimal point, are stored in two 24 bit words with a sign, an exponent and a mantissa, have an allowed range from approximately -10^{308} to 10^{308}, and have either 10 or 11 decimal digits of precision.

Slide 3.

VARIABLES are alphanumeric addresses of memory locations chosen by the programmer, consist of 1 to 8 alphanumeric characters and must have a letter (A-Z) as their first character.

Slide 4.

INTEGER VARIABLES are programmed with the first letter either I,J,K,L,M or N and are addresses of locations, the contents of which are integer constants.

Slide 5.

REAL VARIABLES are programmed with the first letter from A through H or O through Z and are addresses of locations the contents of which are real constants.

3 - VARIABLES AND ARITHMETIC

3-1 Variables

Variables are very important quantities in FORTRAN. These are names given by the programmer to locations in the computer's memory in which information may be stored. The computer associates a name with a particular numeric address but the programmer works with the name only. We call this quantity a variable due to our ability to change the contents of the location and also due to its direct relationship with variables and unknowns in algebra. There is an equivalent type of variable for each type of constant. (1) On the 3300 a variable may consist of from 1 to 8 alphanumeric (i.e., alphabetic and/or numeric) characters. Standard FORTRAN allows only 1 to 6 characters. (2) The first character must be a letter. Since decimal points are not allowed in the names of variables, the distinction between integer and real variables is made as follows. (3) If the first letter of the name is I,J,K,L,M or N then the variable is implicitly type integer. A location is established in memory with this name. Every reference to this name refers to the contents of the location which will be a 24-bit word with an integer value, so I,J,K47, INDEX, NUMBER, N4KX will contain integer constant values. (4) If the first letter is not I,J,K,L,M or N, then it must be A-H or O-Z. The variable is implicitly type real and two consecutive 24-bit words in memory are associated with the variable name. Any reference to that name is to the contents of the 48-bit real location. Any number stored in that location is a real constant with a sign, an 11-bit biased exponent and a 36-bit mantissa: X,Y,Z, TABLE, DISTANCE, RATE, TIME.

3-2 Arithmetic Operations

If we loosely think of the constants and variables as the "nouns" of FORTRAN, then the operations on these nouns are the "verbs". Since we are working with an algebraic language, we need some way of doing arithmetic. There are 5 basic arithmetic operations allowed in FORTRAN. All algebraic problems and any problems in higher mathematics must be expressed in terms of these 5 operations.

Operation	Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/
Exponentiation	**

An arithmetic expression is defined as either (1) a constant or a variable standing alone, (2) a pre-written function such as a sine or a logarithm with an associated argument (a point at which the function is to be evaluated), or (3) any valid combination of constants, variables, functions and the 5 operations. Examples: A, 3, ALOG(X), A+B, 2*J, T1-T2, Y/Z, A+B-C*D/E**F-SIN(X).

3-3 Integer and Real Arithmetic

With each of the two types of variables and constants, there is associated a separate type of arithmetic. While integer and real arithmetic are similar for addition, subtraction and multiplication, in integer division what would be the decimal fraction part of the quotient is truncated since there is no room for it in the 24 bit integer word: 2*3 = 6; 2.*3. = 6. and 3./2. = 1.5 but 3/2 = 1 and 99/100 = 0.

3-4 Arithmetic Expressions

The formation of arithmetic expressions is governed by the usual rules of algebra with some special rules needed by the computer.

Rule 1: Arithmetic operations are done according to the natural hierarchy of algebra:

	1.	Function evaluation
Priority	2.	**
Class	3.	* and /
	4.	+ and -

In any arithmetic expression all functions are evaluated first and then all exponentiations are done, then all multiplications and divisions and finally all additions and subtractions.

A + B * C ** D = $a + bc^d$

Rule 2: When operations of equal priority exist in an arithmetic expression, these equal rank operations are done left to right.

$$A \overset{6}{+} B \overset{5}{-} C \overset{3}{*} D \overset{4}{/} E \overset{2}{**} F - \overset{1}{ALOG(X)}$$ operations done in order by number

$$B * C/D = \frac{bc}{d} \quad \text{but} \quad B/C*D = \frac{bd}{c}.$$

Rule 3: Parentheses may be used as in algebra to change the order of operations, in which case the expression inside the parentheses is evaluated first, according to rules 1 and 2.

$$A + B/C + D = a + \frac{b}{c} + d$$

$$A + B/(C+D) = a + \frac{b}{c+d}$$

$$(A+B)/C+D = \frac{a+b}{c} + d$$

$$(A+B)/(C+D) = \frac{a+b}{c+d}$$

Rule 4: When parentheses are nested, the innermost set is always completed first. Thus (A + (B+C) * D/E) is $a + \frac{(b+c)*d}{e}$ while ((A + (B-C) * D)/E) ** F is $\left(\frac{a+(b-c)d}{e}\right)^f$.

Rule 5: Arithmetic expressions should be written in a single mode entirely real or entirely integer. <u>Do not mix modes</u>. One exception: It is not mixed mode when real numbers are raised to an integer power; thus X ** 2 is preferable to X ** 2. because the second method takes much longer and is also less accurate.

The 3300 does allow mixed mode operations but they can be dangerous as this illustration shows:

Integer	3/2 * 5/4 = 1 * 5/4 = 5/4 = 1
Real	3./2. * 5./4. = 1.5 * 5./4. = 7.5/4. = 1.875
Mixed	3./2 * 5/4 = 1.5 * 5/4 = 7.5/4 = 1.875
Mixed	3/2 * 5/4. = 1 * 5/4. = 1.25 (not expected answer)

Rule 6A: An operation is always required and never implied, thus a(b+c)→A*(B+C). Rule 6B: Two operation symbols may not appear consecutively without an intervening non-operational symbol, so A * -B should be A * (-B). ** is allowed since it represents exponentiation.

3-5 Library Functions

All versions of FORTRAN include a set of pre-written library functions which varies somewhat from computer to computer. The following functions are always available:

SQRT(X)	- positive square root of real number X
SIN(X)	- sine of angle X, X in radians
COS(X)	- cosine of angle X, X in radians
ATAN(X)	- arctangent of X, ATAN in radians
ALOG(X)	- natural logarithm of X
ALOG10(X)	- base 10 logarithm of X
EXP(X)	- exponential of X, e^X natural antilogarithm
ABS(X)	- absolute value of X
FLOAT(I)	- floating point form of integer I
IFIX(X)	- integer (truncated) form of real X

The quantity in parentheses (X) is the argument, a point at which the function is being evaluated. It may be any arithmetic expression of the proper mode , which means according to rule 3 that it must be evaluated before the function itself can be evaluated.

For a more complete list of library functions available on the CDC 3300 see the FORTRAN Reference Manual available at the Computer Center.

4 - ASSIGNMENT STATEMENTS - INPUT/OUTPUT

4-1 The Assignment Statement

Once we understand how to form arithmetic expressions we can proceed to the formation of the first instruction in FORTRAN - the arithmetic assignment statement. It always has the form: variable name = arithmetic expression. Examples: DIST = RATE * TIME; AREA = 3.1415926536 * RADIUS ** 2; FPS = 88./60. * MPH; I = I + 1.

The assignment statement says to do the following: evaluate the arithmetic expression on the right of the equal sign and store the value in the appropriate mode in the location whose address is given by the variable on the left. The '=' is not an arithmetic equal but means 'is replaced by'. Thus the current value stored in location DIST <u>is replaced by</u> the product of the current values of RATE and TIME. Note I = I + 1 is meaningless in algebra but is valid in FORTRAN: the present value of I has 1 added to it and the answer put back in I. A = I + J computes I + J in integer and converts to real for storage in A. K = 3.1 * 5.2 computes the constant 16.12 but stores 16 in K, as the fractional part is truncated when the number is converted to integer.

4-2 A Sample Program

A typical FORTRAN program on the 3300:

```
N-EX     PROGRAM SIMPLE <-------------start
 EX      READ (60,1) X, Y  <----------input
N-EX   1 FORMAT (2F10.0)
 EX      SUM = X + Y <----------------arithmetic calculations
 EX      WRITE (61,2) X, Y, SUM  <----output
N-EX   2 FORMAT (3F12.2)
 EX      CALL EXIT                    phase 2 stop    stopping
N-EX     END                          phase 1 stop <- procedures
```

4-3 Executable and Non-Executable Statements (EX and N-EX)

The instructions in a FORTRAN program are executed in sequential order until a transfer of control statement is reached. Such statements transfer control from the normal sequence to any <u>executable</u> statement by referring to the statement number of that statement. There are two basically different types of statements:

Executable Statements

Statements which cause action within the computer:
- Assignment statements
- Transfer of control statements
- I/O statements

Non-executable statements

Statements which give the computer information but do not cause action:
- Specification statements
- Subprogram statements
- FORMAT statements

4-4 Input/Output

We now turn to the topics of INPUT and OUTPUT. Once an algorithm has been found to solve a given problem, the programmer can regulate the input and output according to the needs of the problem, the limitations of the algorithm and the actual I/O hardware devices. Beginning programmers usually work with the 80 columns of a card and the paper width of the computer's line printer (which on the 3300 is 136 columns). Thus up to 80 characters of information are available on each card (with additional restraints within a particular language such as FORTRAN). The teletype is more often about 64-72 columns per line, and when teletypes are used as I/O devices this limitation and their slow speed leads to the necessity for small amounts of I/O. Since the line printer and magnetic disk are easily accessible from teletype, much use is made of output to those devices and also to input from magnetic disk.

4-5 The READ Statement

The primary input statement in FORTRAN is READ and it assumes that the record is a card or card image (i.e., 80 column record from teletype, magnetic disk or tape). It is used in conjunction with an associated FORMAT:

```
  READ (60,1) X, Y
1 FORMAT (2F10.2)
```

The general form of the input statement is:

```
  READ (LUN,FMTNO) list
```

where LUN is the number of a logical unit, (LU) i.e., an input or output hardware device such as a card reader, a teletype, a magnetic tape or a magnetic disk. Such devices must have been EQUIPPED, i.e., the system must be aware of what number represents what piece of hardware.

4-6 Logical Units Under OS-3

The allowable values for logical unit numbers are 0-100 under OS-3. However since all of these are not allowed in FORTRAN a good working rule is the following. Use LUNs from 1 to 49 for magnetic tapes or magnetic disk files. Once your job is recognized to be valid, LUN = 60 is automatically equipped to the card reader or teletype, depending on whether the program is batch processed or on-line. LUNs from 50 to 59 should be used with care and only with knowledge of what the computer does with these units. Examples:

```
READ (1,40) I
READ (27,96) J, K, X
READ (60,27) RATE
```

4-7 Statement Numbers and Formats

FMTNO is the statement number of an associated FORMAT. FORMATs are non-executable statements and under OS-3 may appear anywhere in the program. They are not executed sequentially but are referred to when the READ is being executed. They may be referred to by more than one I/O statement. Like any statement number, that of a FORMAT must be positive and less than 32768. Every FORMAT must have a statement number.

4-8 Input Variable List

The list must be made up only of variable names separated by commas; no constants, no functions and no other arithmetic expressions are allowed. Numbers are read from a data record and stored in the locations whose addresses are the variable names. Thus consider:

```
          READ (60,490) A, B, TABLE, TAX, IK
      490 FORMAT (F8.2, F12.4, E13.4, F8.1, I7)
```

During compilation 4 real locations and 1 integer location are established with the given variable names as addresses. During execution this set of instructions reads five numbers from one record and stores them in locations A, B, TABLE, TAX, IK in the mode determined by the variable type. The FORMAT indicates the number of records to be read, where in the 80 columns each number is located, whether it is integer or real and the number of decimal places it has if it is real.

4-9 The WRITE Statement

The corresponding output instruction is:

```
          WRITE (61, 364) X, Y
      364 FORMAT ('b̸', F12.1, F12.1)
```

This statement writes the contents of locations X and Y on the line printer. General form:

```
          WRITE (LUN, FMTNO) list
```

with

LUN, FMTNO and list as in READ.

LUN = 61 is automatically equipped to the line printer or teletype; LUN = 62 is usually equipped to the card punch. In general a WRITE statement writes the contents of each variable name address in the list on the LUN according to the FORMAT with statement number FMTNO.

5 - INPUT/OUTPUT - FORMAT

5-1 Review of Input/Output Statements

We have seen that the general form for the input statement is:

```
READ (LUN, FMTNO) list
```

where LUN and FMTNO are the logical unit number and the statement number of an associated FORMAT statement, respectively, and list is a set of variable names separated by commas. Information from the LU is input according to the specifications of the FORMAT with statement number FMTNO and stored in the locations given by the variables in the list. The output statement is similar:

```
WRITE (LUN, FMTNO) list
```

This statement outputs information from the locations given in the list to the LU according to the specifications in the FORMAT, e.g.

```
WRITE (61,32) X, TAX, IJ
```

writes the contents of locations X, TAX, IJ on LU 61 (either line printer or teletype) according to the specifications in FORMAT 32.

5-2 The FORMAT Statement

FORMATs are complicated statements because of the many different specifications which they can contain. All FORMAT statements must have statement numbers since they are referenced by one or more READ or WRITE statements. Although some compilers require that all FORMATs be placed at the top or at the bottom of the program, OS-3 allows these non-executable statements to go anywhere in the body of the program. The specifications indicate the forms in which data exist on the input records and give the chosen appearance to the output records. They are used to indicate spacing, the types of quantities being input or output, the number of decimal places of decimal numbers, and the positioning on the I or O record. They are also used to generate headings and other explanatory information in the output. The general form of FORMAT is:

```
#FORMAT (specifications separated by commas).
```

e.g.

```
14 FORMAT (I5, 6X, F10.2)
```

5-3 Numeric Format Specifications

Numeric specifications are designed primarily to handle integer and real constants. Since READ is reading from a data record we must remember that all 80 columns are available and that blanks are treated as zeros.

To input/output integer constants use:

Iw$\Longrightarrow$integer field of w columns width. On input the data must be right-justified to eliminate blanks treated as zeros. On output the digits will be right-justified with blank fill to the left.

I2	XX, ƀX,	-X	X a digit
I3	XXX, ƀXX,	-XX	
I12	ƀƀƀƀƀXXXXXXX	$0\leq\#\leq 8388607$	
	ƀƀƀƀ-XXXXXXX	$-8388607\leq\#<0$	

To input/output real constants in decimal fraction form use:

Fw.d$\Longrightarrow$real decimal field of w columns width. On input if no decimal point exists in the w columns, then d is the number of places to the right of the decimal point when the number is stored in memory. If the decimal point is in the w columns, d is ignored. On output follow this rule: $w\geq d+3$. This leaves space for d digits to the right of the decimal point, the point itself, a leading digit and a sign.

Thus -1.487 must be output in F6.3 specification at least.

On output if the width w is considerably larger than the number of decimal places d, then the number is right-justified with blank fill to the left.

F6.2 XXX.XX or (on input only) XXXX$_{\wedge}$XX (the carat is an implied decimal point).

F5.0 XXXX. or XXXXX (on input and on output on the 3300); most computers add a decimal point on output when d = 0 but not the 3300.

Horizontal spacing: nX skips n columns, and is used on both the input and the output. Thus, 3X skips 3 spaces horizontally, i.e., reads or writes three blanks.

5-4 Input Sample - READ and FORMAT

Example of input from a card or teletype line:

```
Cols 4 5 6 7 8 9 10                   17 18 19 20
┌────────────────────────────────────────────────┐
│       1 2 3 . 4 5 6                 -  1  8  7 │
└────────────────────────────────────────────────┘
```

READ (60,5) X would read one number and FORMAT 5 should have one real specification, since X is real; if the specification is F10.0, F10.1, ..., F10.10, since a decimal point is there, X = 123.456 will be stored. If the specification is F7.0, F7.1, ..., F7.7, since a decimal point is in column 7, X would be stored as 123.000; since columns 8, 9, 10 are not part of w, .456 is not part of the number.

If the specification is F6.0, we would also get X = 123.000, having 6 digits with no places to the right and no decimal point in the w columns. But for F6.1 we get 12.3; for F6.2, 1.23; for F6.3, .123; for F6.4, .0123; for F6.5, .00123 and for F6.6, .000123. There is no decimal point present in the 6 columns, so the d part of the specification determines the magnitude of the number.

If we change the READ to:

```
      READ (60,6) X, I
    6 FORMAT (F10.3, 6X, I4)
```

we get X = 123.456, I = -187. If (F10.3, 6X, I3), then I = -18, and if (F10.3, I4) then I = 0. Note: to find the field on the record we add the w's and the n's together: e.g.

```
           (F10.3,    6X,    I4)
      cols  1-10    11-16  17-20
             10    +  6  +  4
              w       n     w
```

Since columns 11-16 were blank, we could also have:

```
    6 FORMAT (F10.3, I10)
```

or if

```
      READ (60,7) X, Y
    7 FORMAT (F10.3, 6X, F4.0)  or  (F10.3, F10.0)
```

then X = 123.456 and Y = -187.00 . . . . if F4.1 we get Y = -18.7; F4.2, Y = -1.87.

It may look as if there are only two numbers on the record, but we can use specifications to define more, e.g.

```
      READ (60,7) I, X, Y                    I = 123
    7 FORMAT (I6, F4.0, 6X, F4.0)            X = .456
                                             Y = -187.
```

5-5 Exponential Numbers and E Format

There is a third numeric specification Ew.d which is used to read or write a number in scientific notation rather than decimal fraction form. There are several shorthand forms for inputting such numbers but we shall only discuss the normal form which would require the number to be punched or typed right-justified as X.XXXX E b̸ XX for exponents -99 to +99 with E XXX for exponents 100 to 308 and with -XXX for exponents -308 to -100. The input specification would be E 11.4. The output is always in this form on the 3300, although this is not true in standard FORTRAN. The output rule to follow is thus $w \geq d + 7$, leaving one space for sign, leading digit, decimal point and four spaces for the exponent. The main purpose of Ew.d specification is the precise representation of very large and very small numbers, e.g. if X = .00000000012743 this number can be more easily represented on input by 1.2743 E-10, either READ using E format or set by X = 1.2743 E-10; full precision on output will be obtained only if we use Ew.d specification since X written in F12.4 gives .0000 while E12.4 gives 1.2743E-10. Similarly, the large number 1 000 000 000.0 is more easily represented as 1.E 09 and on output F12.4 would give 000 000.000*, the * indicating overflow of the FORMAT field, i.e., digits missing to the left, while E12.4 would give 1.0000E 09. By putting a number n in front of an F, E, or I specification that specification is repeated n times like the skipping specification, e.g. (F10.4, F10.4, I5, I5, I5) → (2F10.4, 3I5).

5-6 End-Of-Record Specification

As well as a horizontal spacing specification, there is a specification for vertical spacing: /. This specification essentially signals the end of a record and is used to start a new record within a single FORMAT: either a new card, a teletype line, or magnetic disk card image on input or a new line on the line printer or teletype on output.

Multiple slashes can be used to give multiple vertical spacing. Since / is an end-of-record indicator, it takes 2 slashes to skip one line vertically, and in general it takes n+1 slashes to skip n lines. The slash also is a delimiter, so no commas are needed to separate the slashes. In the FORMAT (F10.2//I5) the first data value will be processed in F10.2 from the first record with the first / indicating the end of that record. The second / indicates the end of the second record and that record is thus blank, effectively skipped. The second data value will be processed in I5 from the third record. It is important to note that the last right parenthesis of any FORMAT, if it is reached, also acts like a slash and indicates end-of-record.

6 - HOLLERITH ; STARTING, STOPPING, TRANSFER OF CONTROL

6-1 The Hollerith Specification and Carriage Control

On output whenever a new line is to be started, whether at the beginning of a FORMAT or by a / inside, the first character on that line is used as a carriage control symbol and is not printed. The carriage on the line printer or teletype is controlled by that character. A blank will cause single spacing before any printing: a 0 will cause double spacing and a 1 will cause a skip to a new page. A - causes triple spacing and a + overprinting on the current 3300 line printer. Single spacing is the most common type of control needed and can be done in several ways. The other CC symbols can be generated only by Hollerith specifications to be discussed next. But to get a blank in column one of the output record for single spacing we can use nX which will skip over n columns, e.g. (1X, F8.4) skips over column 1→ƀ in column 1 → single space before printing. Or we can use a large value for w in any of the numeric specifications, guaranteeing a blank in column 1 because of the blank fill by the computer, e.g. (F16.4) ƀƀƀƀƀƀXXXX.XXXX. To obtain the other CC symbols we need Hollerith specification, which is used primarily for output. This specification is most important because it gives us a means of printing out any string of characters as a heading or explanation of the output. Hollerith specification can be formed in either of two ways; (1) nH text where n is an exact count of the characters in the text or (2) 'text' where whatever is between the single quotes will be printed. This latter method is not available in all FORTRAN compilers, but is available on the 3300. It is preferred since miscounting n is a common error in the first form. This, by the way, is the one place within a FORTRAN program where a blank is not ignored. All blanks to be printed by Hollerith specification must be included in the count (nH form) and space must be left for them in the text in the quote form.

```
             WRITE (61, 12) X, Y
          12 FORMAT (1H0, 3X, 'X = ', F10.2, 3X, 'Y = ', F10.2)
```

X $\boxed{\quad 337.276}$ Y $\boxed{\quad -28.74}$ will generate, with the .276 rounded to two decimal points,

```
double space
X = 337.28   Y = -28.74
```

For a heading at the top of a page there may be no list, e.g.

```
            WRITE (61,98)
         98 FORMAT (1H1, 0X, 'THIS IS A HEADER FOR A NEW PAGE')
```

```
To center on the page:  136 characters total
                         -1 for CC
                        -31 for number in text
                        104 left
```

104/2 = 52, so use 52X to center text.

6-2 Starting The Program

The first statement in every FORTRAN program on CDC computers must be the non-executable statement: PROGRAM name. Name must have the same form as a variable but cannot be the same as any variable used in the program, e.g. PROGRAM SIMPLE. Standard FORTRAN does not use this statement, but it must be used on the 3300 and all CDC computers.

6-3 Ending The Program

The last statement in every FORTRAN program and also in every subprogram, i.e., the last card in a program deck or the last line on teletype in all FORTRANs is the non-executable END. This statement indicates to the compiler during the compilation phase the end of the set of related instructions to be translated, either in the program or in a separate subprogram. (FUNCTIONS are subprograms.)

During the execution phase the END no longer exists, so to stop execution a STOP or CALL EXIT is needed. These are executable statements. EXIT is a prewritten subprogram designed to make the return of control to the operating system easier than the plain STOP, designed for use on computers which actually halted. Either statement signals the end of the EXECUTION phase. There should be one or more STOP or CALL EXIT statements in every program. They may appear anywhere between PROGRAM name and END after the other non-executable statements we will discuss.

6-4 Transfer of Control - The GO TO Statement

A program designed from the instructions we have dealt with so far would be executed sequentially, i.e., the statements would be executed one at a time as they appeared in a listing of the program. Most FORTRAN programs at some time require a change from this normal sequential execution. We can make this change by using transfer of control statements. The simplest of these is the unconditional GO TO of the form: GO TO n, where n is the statement number of any executable statement, e.g. GO TO 7, GO TO 43.

```
          1 READ (60,3) X
            ...
            ...
            WRITE (61,17) X, XSQ, Y
            ...
            GO TO 1
```

6-5 The Computed GO TO Statement

Somewhat less useful is the computed GO TO which uses the value of an integer variable in standard FORTRAN (or any arithmetic expression under OS-3) as a switch. Depending on the value of the switch, the program may transfer in any of several possible directions.

GO TO (n_1, n_2, ... n_m), iswitch where the last comma is necessary in standard FORTRAN (optional on the 3300). The n_i, i=1, m are statement numbers of executable statements. The value of 'iswitch' determines which statement number to transfer to

$$\text{iswitch} = \begin{array}{ll} 1, \rightarrow n_1 & \text{if iswitch} < 1, \rightarrow n_1 \\ 2, \rightarrow n_2 & \\ \vdots & \\ m, \rightarrow n_m & \text{if iswitch} > m, \rightarrow n_m \end{array}$$

e.g. company payroll with bonus being given, bonus amount depends on years of service:

years	0-3	4-6	7-15	16-up
bonus	25	50	75	100
LCODE	1	2	3	4

If we can relate LCODE to years worked, then

```
   GO TO (6,81,97,4), LCODE
 6 PAY = PAY + 25.
   GO TO 15
81 PAY = PAY + 50.
   GO TO 15
97 PAY = PAY + 75.
   GO TO 15
 4 PAY = PAY + 100.
15 WRITE (61,13) PAY
```

7 - ARITHMETIC IF; OS-3 CONTROL CARDS

7-1 The Arithmetic IF Statement

The computer has the ability to compare numbers and we can take advantage of that fact by using another conditional transfer of control statement: the arithmetic IF statement. This statement is based on the Trichotomy Principle of mathematics: every real number is either negative, positive or zero. Thus it is a three way transfer of control statement of the form: IF (arithmetic expression) n_1, n_2, n_3 where n_1, n_2, n_3 are statement numbers of executable statements, two of which can be the same but not all three. This statement works as follows: the arithmetic expression is evaluated; the numeric result is either negative, positive or zero. If the arithmetic expression is negative, transfer to statement n_1. If the arithmetic expression is zero, transfer to statement n_2. If the arithmetic expression is positive, transfer to statement n_3. For example,

```
  IF(X-Y) 3,4,5
3 Z = 2. * X-Y
  GO TO 6
4 Z = X+Y
  GO TO 6
5 Z = X-Y
6 WRITE (61,36) Z
```

```
  if X = 3., Y = 4.  X-Y < 0  Z = 6-4 = 2.
  if X = 4., Y = 4.  X-Y = 0  Z = 4+4 = 8.
  if X = 5., Y = 4.  X-Y > 0  Z = 5-4 = 1.

  Note that IF(X-Y) 3,4,5 says
  if X-Y < 0 → 3 same as if X < Y → 3
  if X-Y = 0 → 4 same as if X = Y → 4
  if X-Y > 0 → 5 same as if X > Y → 5
```

So when we write down the arithmetic expression we can think of comparing it with 0, e.g. IF(A) 12,14,17, or if it consists of two or more parts we can compare the parts directly as we did above. Another example of that type of comparison is:

```
   IF(I + 2) 17, 17, 24
if I + 2 < 0 → 17    I < -2 → 17
   I + 2 = 0 → 17 ⟺ I = -2 → 17
   I + 2 > 0 → 24    I > -2 → 24
```

Common case to check for division by 0 before dividing, e.g. if X = A + B/(C + D) to be calculated in a program.

```
      DEN = C + D
      IF(DEN) 12,19,12
   12 X = A + B/DEN
      GO TO 16
   19 . . . no division
```

7-2 Requirements of the OS-3 Operating System

We now have enough instructions to write some simple but meaningful FORTRAN programs, but before we can submit any programs to the computer, we need to examine the requirements of OS-3. All operating systems require the use of control statements so that the system can keep track of valid users and their accounts. In addition certain general features of the computing system are accessible through these control statements. We will assume for now that all the programs we write are punched on cards and submitted as a deck to the 3300. Later we will demonstrate the requirements for running FORTRAN programs from teletype.

7-3 The Job Card

The first card of any deck submitted to the CDC 3300 must be

$^7_8$JOB, XXXXXX, AAAA, ID

where the 7 and the 8 are multi-punched in column 1, there being no single character on the keypunch which has a $^7_8$ code. The job number is a six digit number. The user code is four alpha-numeric characters unique to a particular user. Not all the commas are needed but for simplicity and consistency we will put them in. The ID is your name and any other identifying information which you may want. A specific example:

$^7_8$JOB, 748921, XYZQ, J. DOE

7-4 Job Numbers, Charges and File Blocks

When a job number/user code is created at the computer center, a fixed number of seconds of CPU time is established for that job, depending on the amount of money supplied by the user ($5/min is the CPU charge). As each run is made under the job number/user code, the cost of the run is subtracted from the previous balance. This cost includes card reading, printing and on-line charges as well as CPU charges, so the system keeps track of the total amount of money remaining in the account and updates the number of CPU seconds remaining accordingly, also at the end of each run. When the job number is created a fixed number of save and of scratch file blocks are established for the user. Save file blocks are permanent storage areas on magnetic disk in which decks of cards, either programs or data, may be stored rather than being in actual card form. One file block is 512 24-bit words. Programs and data may be stored permanently in save file blocks by entering them directly from teletype and storing them on disk, or by copying a deck of cards onto disk. In either case each related set of information is saved under a chosen 1 to 8 character file name and is retrievable by referencing the file name.

If the programs or the data are to be listed, or output is required in printed form or in an intermediate form for later inputting, then scratch file blocks are used. A scratch file is temporary magnetic disk storage available only while the job is being run. While standard output scratch files are sent to the appropriate output device and some scratch files may become save files during the run, all other scratch files are lost for further use at the end of the run.

7-5 The Time Card

When each job card is read by OS-3 and determined to be valid, the system automatically limits the job to 60 seconds of CPU time. Since this is the equivalent of $5, it is best for beginning programmers to change this limit in order to avoid using all $5 on one run. We use the time card $^{7}_{8}$ TIME = 5 which causes the computer to quit working on the job after 5 seconds, if it does not finish before. For jobs which require more than 60 seconds we can use $^{7}_{8}$ TIME = 3600, for example, to allow one hour of CPU time. Note that the cost is greater than $300 due to I/O charges.

7-6 Assigning Logical Units: The Equip Card

In some instances the data for a program will not be on cards but may be on a save or scratch file or on magnetic tape.

In these cases you will need to tell the computer where the data are by equipping a LUN to a particular piece of hardware.

${}^{7}_{8}$EQUIP, 9 = FILE establishes logical unit 9 as a scratch file. Data may be written on the file and, after it is rewound, they may be read from it again. Unless saved as a save file, the information on LUN 9 is lost at the end of the run.

${}^{7}_{8}$EQUIP, 27 = TEMPDATA establishes logical unit 27 as equivalent to the save file TEMPDATA. Data previously stored on TEMPDATA are then available on LUN 27.

The LUN in a READ must match the LUN in the EQUIP so that the READ inputs data from the correct device. Similarly for the LUN in a WRITE statement. Remember that 60/61 are automatically equipped to the card reader and line printer or to the teletype. From FORTRAN it is best to use LUNS 1-49 for save or scratch files or magnetic tapes. 50-59 are used by the OS-3 system and it is best to avoid these units.

7-7 The FORTRAN Compiler

To compile and execute a FORTRAN program from cards:

```
          7
          8 FORTRAN, L, R <--------  Followed by deck of FORTRAN
                ^       ^             cards
               /         \
Compiles FORTRAN   Lists on  If no compilation errors,
source program in- LUN = 61. loads object program and
to object program.           executes.
```

```
          7
          8 FORTRAN, L, R
                 PROGRAM NAME
                 -------
                 -------
                 END
          77
          88 end-of-file card for compiler, indicating last
             program or subprogram to compile



          77
          88
         { (data cards if necessary)
         {77
         {88 if end-of-file check is used in the FORTRAN program
```

7-8 The LOGOFF Card

The last card of every deck must be $\overset{7}{8}$ LOGOFF which completes the run and updates the user's account.

8 - PROGRAM EXAMPLES 1; SUMMATION AND COUNTING

8-1 Program Example

Example of a program to convert °F to °C.

(See sheet 8-#1)

Multiple sets of data on save file TEMPDATA

```
7
8JOB,738921,XYZQ,    J. DOE

7
8TIME=3

7
8EQUIP,13=TEMPDATA

7
8FORTRAN,L,R

       PROGRAM TEMPS
     1 READ (13,10) F
    10 FORMAT (F10.0)
       IF(F + 500.) 3,3,4    ←————————
     4 C = 5./9. * (F-32.)
       WRITE (61,7) F,C
     7 FORMAT (F10.1, . . .)
       GO TO 1
     3 CALL EXIT
       END

77
88

7
8LOGOFF
```

To stop, let the last piece of data have any temperature <-500, since absolute 0 is approximately -458°F. The check on the last card is inserted at the arrow above: IF(F + 500.) 3,3,4.

If the data on TEMPDATA are

```
  32.
 100.
 180.
 212.
-999.
```

the program will stop when F is READ as -999.

Program to calculate the distance of a point (X,Y) from the origin of a Cartesian coordinate system.

(See sheet 8-#2)

```
7
8JOB,748921,XYZQ,J.DOE
7
8TIME=3
7
8FORTRAN,L,R
      PROGRAM TEMPS
      READ(60,10)F
   10 FORMAT(F10.0)
      C=5./9.*(F-32.)
      WRITE(61,7)F,C
    7 FORMAT(F10.2,'DEG.
       F=',F10.2,'DEG. C')
      CALL EXIT
      END
77
88
   (1 DATA CARD WITH DEG.C)
7
8LOGOFF
```

8-#1

```
   PROGRAM DISTANCE
   READ(60,18)X,Y
18 FORMAT(2F5.1)
   DIST=SQRT(X**2+Y**2)
   WRITE(61,19)DIST
19 FORMAT('THE DISTANCE IS'
  1,F8.9)
   CALL EXIT
   END
```

8-#2

8-2 EOF Check

Rather than use F + 500. as the stopping criteria, we can use an end-of-file, a $\frac{77}{88}$, and check for its having been read. To check for EOF at the end of a data file we use the instruction in the form IF(EOF(LUN)) GO TO n or IF(EOF(LUN)) CALL EXIT.

We can modify our previous example to allow calculations of distance to many points with the use of this instruction.

(See sheet 8-#3)

Here is a simple problem to illustrate arithmetic if checking for division by 0 in X = A + B/ (C + D).

(See sheet 8-#4)

8-3 Summation

One of the most important topics in programming is the topic of summation. We have seen that we can add together two quantities stored in two locations in memory and store the sum in some other location, e.g. SUM = A + B. What we are referring to with the word summation is a mathematical sum, of not just two, but an arbitrary number of quantities. This is often represented in sigma notation as $S = \sum_{i=1}^{n} X_i$ where i is a subscript which takes on values from 1 to n. $S = \sum_{i=1}^{n} X_i$ <u>means</u> $X_1 + X_2 + X_3 + \ldots + X_n$. This notation is valid regardless of the value of n but if we were to write FORTRAN statements to represent this sum, e.g. $X_1 + X_2 + X_3 + X_4 + X_5 + X_6$ for n = 6, we would need a new statement every time n changed. We would like a technique which like sigma notation remains valid regardless of n. We take care of this problem by doing the following: consider

$$S = \sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n.$$

```
7
8FORTRAN,L,R
      PROGRAM DIST2
   10 READ(60,18)X,Y
   18 FORMAT(2F5.1)
      IF(EOF(60))CALL EXIT
      DIST=SQRT(X**2+Y**2)
      WRITE(61,19)DIST
   19 FORMAT('THE DISTANCE IS'
     1,F8.3)
      GO TO 10
      END
77
88
    (DATA CARDS)
77
88
7
8LOGOFF
```

8-#3

```
   PROGRAM CALC
   READ(60,13)A,B,C,D
13 FORMAT(4F5.1)
   DENOM=C+D
   IF(DENOM)18,27,18
18 X=A+B/DENOM
   GO TO 36
27 WRITE(61,12)
12 FORMAT(' DENOMINATOR=0')
   GO TO 34
36 WRITE(61,28)A,B,C,D,X
28 FORMAT(4F6.2,F10.2)
34 CALL EXIT
   END
```

8-#4

9 - SUMMATION AND COUNTING; LOGIC CONCEPTS

9-1 Summation Examples

In our previous discussion of summation we progressed to the point of considering $S = \sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n$. We'll now show a program that will do this operation.

(See sheet 9-#1)

Counter I is counted from 1 to N. X also goes from 1. to within a fraction of N (because of roundoff errors in binary representation of decimal fractions). X may never be $\equiv$ N (real form).

9-2 Counting

Counting and summing are essentially the same operation. In counting the same three steps always occur. The counter is initialized (usually to 0 or 1), then the counter is incremented by having 1, (or some other integer) added to it e.g. I = I + 1. Then the value is tested against a fixed value e.g. IF(I-N) 3,3,4. The testing and incrementing can also be done in the reverse order. When summing, we often use a test on a counter to tell us when we are through, but we do initialize and increment the sum location in the same way.

There are two basic types of counts we can do. They are illustrated as follows:

```
                  COUNT UP               COUNT DOWN
initialize        I = 1                      READ N
increment       3 I = I + 1                3 N = N - 1
test              IF(I-N) 3,3,4              IF(N) 4,4,3
                4 . . .                    4 . . .
```

$I \leq N \rightarrow 3$ $\qquad\qquad$ $N \leq 0 \rightarrow 4$

We will return to summation when we discuss arrays, subscripts and DO loops.

9-3 Three-Way and Two-Way IF Statements

Note that IF(N) 4,4,3 is really a three-way IF with two branches. Because this is not a very efficient statement the logical IF was developed with FORTRAN IV.

```
      PROGRAM SIGMA
      READ(60,10)N
   10 FORMAT(I3)
      SUM=0.
      X=1.
      I=1
    3 SUM=SUM+X
      X=X+1.
      I=I+1
      IF(I-N)3,3,4
    4 WRITE(61,6)N,SUM
    6 FORMAT('SUM FROM 1 TO'
     1,I4,'IS',F7.0)
      CALL EXIT
      END
```

9-#1

To understand this instruction we must first deal with relational operators and expressions and logical operators and expressions.

9-4 Relational Operators and Expressions

The relational operators are used to compare arithmetic expressions. There are six relational operations.

.LT.	.GT.	The periods before and after are
.LE.	.GE.	needed to distinguish these opera-
.EQ.	.NE.	tions from variable names.

Note: .EQ. is not the same as = in FORTRAN.

We use these operations as we did arithmetic operations, to form expressions: representing an expression as <exp>. <Rel. exp> = <arith exp><rel. op.><arith exp>. e.g. X * Y .LE. 47. - Z/Q. Now while an arithmetic expression has a numeric value, a relational expression has a logical value - it is either true or false, i.e.,

if $X * Y \leq 47. \quad -Z/Q$, the expression is T, or

if $X * Y > 47. \quad -Z/Q$, the expression is F.

9-5 Logical Operators and Expressions

A relational expression by itself is a simple logical expression. We may also combine two or more relational expressions with one or more of the three logical operations thus generating a compound logical expression. The three logical operations are .AND., .OR., .NOT.

<Compound logical expressions> = <Rel. exp><log. op><rel. exp> or <rel. exp> alone, such an expression also has a value T or F. The truth value of a compound logical expression is determined by the rules of Boolean algebra. The easiest way to present the rules we need is to introduce truth tables.

9-6 The .AND. Operator

Let P,Q, be two relational expressions, then since P has two possible values and Q has two, the truth table of $P \wedge Q$ (P .AND. Q) is:

P	Q	P .AND. Q
T	T	T
T	F	F
F	T	F
F	F	F

e.g.

X .LE. Y .AND. 2. * X .GT. Y + 3.	Whole exp.
let X = 6., Y = 7. then $6. \underset{T}{\leq} 7.$ and $12. \underset{T}{>} 10.$	T
let X = 3., Y = 7. then $3. \underset{T}{\leq} 7.$ but $6. \underset{F}{\ngtr} 10.$	F
let X = 9., Y = 7. then $9. \underset{F}{\nleq} 7.$ altho $18. \underset{T}{>} 10.$	F
let X = 2.1, Y = 2. then $2.1 \underset{F}{\nleq} 2.$ and $4.2 \underset{F}{\ngtr} 5.$	F

9-7 The .OR. Operator

A truth table for $P \vee Q$ (P .OR. Q) is:

P	Q	P .OR. Q
T	T	T
T	F	T
F	T	T
F	F	F

e.g.

A .EQ. B .OR. B**2 - 4. * A .GE. 3.	Whole exp.
let A = 8., B = 8. then $A \underset{T}{=} B$ and $64 - 32 \underset{T}{\geq} 3.$	T
let A = 2, B = 2 then $A \underset{T}{=} B$, but $4 - 8 \underset{F}{\ngeq} 3$	T
let A = 1, B = 4 then $A \underset{F}{\neq} B$ but $16 - 4 \underset{T}{\geq} 3$	T
let A = 4, B = 2 then $A \underset{F}{\neq} B$ and $4 - 16 \underset{F}{\ngeq} 3$	F

10 - LOGICAL IF; PROGRAM EXAMPLES

10-1 The .NOT. Operation

For $\neg P$ (.NOT. P), the negation operator operates on one logical expression only; thus we have:

P	.NOT. P
T	F
F	T

.NOT. X .LE. Y

let X = 3., Y = 6. $X \leq Y$ so .NOT. X .LE. Y is False

let X = 9., Y = 6. $X > Y$ so .NOT. X .LE. Y is True

10-2 The Logical IF Statement

The <u>logical</u> <u>IF</u> is formed as follows:

```
IF(logical expr.) statement s
statement t
```

Statement s may not be another logical IF or a DO and it must be executable. It is executed as follows. The logical expression is evaluated. If it is T, then statement s is executed. If the logical expression is F, then the normal sequence is resumed with statement t next. If s is to be executed and s is a transfer of control statement, then the statement executed after s is entirely determined by the type of transfer. If s is executed but it is not a transfer of control, then normal sequencing is continued and t is executed after s.

```
IF(X .LE. Y .OR. Y .LE. Z) GO TO 37
WRITE (61,47) X,Y,Z
```

if $X \leq Y$ or if $Y \leq Z$, then statement 37 will be executed next; otherwise the WRITE is executed next.

```
IF(I * J .EQ. K**2) I = I + 1
WRITE (61,57) I,J,K
```

if $I * J = K^2$ then I is reset to I + 1 and then the WRITE is executed. If $I * J \neq K^2$ then just the WRITE is executed.

10-3 Sample Problem

Use of logical IF to find the largest element of a set of numbers. We read the numbers one at a time and compare each number with XMAX, which will contain the largest value up to the particular comparison being made.

(See sheet 10-#1)

To illustrate execution we introduce data cards:

```
 1.
 4.
 7.
 2.
77
88
```

XMAX is set to -1. X 10^{300}, X is read as 1., but it is not end-of-file so 1. is compared against -1. X 10^{300} and XMAX is set to 1. Then X is read as 4., again no EOF, so 4. is compared with 1. and XMAX is set to 4. Then X is read as 7., again no EOF, so 7. is compared with 4. and XMAX is set to 7. Then X is read as 2., again no EOF, so 2. is compared with 7. XMAX is not changed since 2. < 7. Then EOF is read and XMAX is written as 7.0 and the program stops.

10-4 The Complete Hierarchy Rules

We now have all the available operations and can form the final hierarchy rules which indicate the order in which operations are performed.

(1) Function evaluation

(2) **

(3) * and /

(4) + and -

```
      PROGRAM MAXFIND
      XMAX=-1.E300
    5 READ(60,1)X
    1 FORMAT(F5.1)
      IF(EOF(60))GO TO 6
      IF(X.GT.XMAX)XMAX=X
      GO TO 5
    6 WRITE(61,10)XMAX
   10 FORMAT('THE LARGEST IS'
     1,F6.1)
      CALL EXIT
      END
```

10-#1

(5) Relational operators: .LT., .LE., .EQ., .NE., .GE., .GT.

(6) .NOT.

(7) .AND.

(8) .OR.

10-5 Parentheses and Logical Expressions

While parentheses are allowed to change the order of arithmetic operations, on the 3300 they cannot be used to change the order of logical operations. If the natural order is not what is needed, two or more IF statements may be required to cause proper checking of all logical operations. If r,s and t represent logical expressions then

```
r .AND. .NOT. s .OR. t is evaluated as
.NOT. s followed by r .AND. (.NOT.s) and then
     (r .AND. .NOT. s)
.OR. t.
```

10-6 Example of Logical IF

A useful example of the use of logical IF comes in a quick visit to the dice tables in Nevada. Two dice are thrown and the numbers on the upper surface of the dice are added. The possible sums are 2,3,4,...,12. If the sum is 7 or 11, the game is won. If the sum is 2,3, or 12 the game is lost. Any other number becomes the player's "point" and he continues throwing until he throws his "point" again in which case he wins or until he throws a 7 in which case he loses. Some of the statements in a program involving such a game might include generation of the sum of the dice = NSUM, followed by

```
         IF(NSUM .EQ. 2 .OR. NSUM .EQ. 3 .OR. NSUM .EQ. 12) GO TO 16
         IF(NSUM .EQ. 7 .OR. NSUM .EQ. 11) GO TO 18
         NPOINT = NSUM
      10 GENERATE NEW NSUM
         IF(NSUM .EQ. 7) GO TO 16
         IF(NSUM .EQ. NPOINT) GO TO 18
         GO TO 10
      18 handles win
         . . . .
      16 handles loss
         . . . .
```

10-7 Two Linear Equations In Two Unknowns

Sample problem: System of two linear equations in two unknowns.

mathematical example	solution	check
$3X + 4Y = 10$	$X = 2$	$6 + 4 = 10$
$8X - 6Y = 10$	$Y = 1$	$16 - 6 = 10$

In general $AX + BY = C$ Two unknowns $\Longrightarrow$ cannot be
$DX + EY = F$ formulated in FORTRAN directly.

Algorithmic solution:

$$X = \frac{CE - BF}{AE - BD} \qquad Y = \frac{AF - CD}{AE - BD}$$

(See sheet 10-#2)

Sample data for linear equations problem.

```
     3.     4.     10.     8.     -6.     10.
     3.     4.     10.     6.      8.     10.
7 7
8 8
```

10-8 Quadratic Equations

Sample problem: Quadratic equation solutions.

mathematical example	solutions
$X^2 - 6X + 8 = 0$	$X = 4, X = 2$

In general $ax^2 + bx + c = 0$

```
      PROGRAM LINEQNS
    1 READ(60,7)A,B,C,D,E,F
    7 FORMAT(6F10.2)
      IF(EOF(60))CALL EXIT
      DENOM=A*E-B*D
      IF(DENOM.EQ.0.)GO TO 9
      X=(C*E-B*F)/DENOM
      Y=(A*F-C*D)/DENOM
      WRITE(61,37)X,Y
   37 FORMAT('0X=',F10.2,5X,
     1'Y=',F10.2)
      GO TO 1
    9 WRITE(61,45)
   45 FORMAT('0NO SOLUTION')
      GO TO 1
      END
```

10-#2

The solution is obtained from the quadratic formula:

$$X = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$b^2 - 4ac$ is the discriminant. The discriminant determines the type of roots.

If discriminant < 0, there are two complex roots: case (1)

If discriminant = 0, there are two equal real roots: case (2)

If discriminant > 0, there are two unequal real roots: case (3)

Case:

(2) if discriminant = 0, $X_1 = X_2 = \frac{-b}{2a}$

(3) if discriminant > 0, $X_1 = \frac{-b + \sqrt{\text{discr.}}}{2a}$

and $X_2 = \frac{-b - \sqrt{\text{discr.}}}{2a}$

(1) if discriminant < 0, complex conjugate roots

$$X_1 = \frac{-b}{2a} + \sqrt{\frac{-\text{discr.}}{2a}}\ i$$

$$X_2 = \frac{-b}{2a} - \sqrt{\frac{-\text{discr.}}{2a}}\ i$$

We can work with complex arithmetic on the 3300 but we will defer discussion until later. What we do instead is to work with the real and the imaginary parts separately:

$$\text{XREAL} = \frac{-b}{2a}$$

$$\text{XIMAG} = \frac{\sqrt{-\text{discr.}}}{|2a|}$$

(See sheet 10-#3)

```
      PROGRAM QUADEQNS
    1 READ(60,19)A,B,C
   19 FORMAT(3F10.2)
      IF(EOF(60))CALL EXIT
      TWOA=2.*A
      DISCR=B*B-4.*A*C
      IF(DISCR)30,40,50
   30 XR=-B/TWOA
      XI=ABS(SQRT(-DISCR)/TWOA)
      WRITE(61,37)XR,XI
   37 FORMAT('OTWO COMPLEX CON'
     1,'JUGATE ROOTS',F8.4,
     2'+OR-',F8.4,'I')
      GO TO 1
   40 X=-B/TWOA
      WRITE(61,38)X
   38 FORMAT('OTWO EQUAL REAL'
     1'ROOTS     BOTH',F8.4)
      GO TO 1
   50 ROOT=SQRT(DISCR)
      X1=(-B+ROOT)/TWOA
      X2=(-B-ROOT)/TWOA
      WRITE(61,39)X1,X2
   39 FORMAT('OTWO UNEQUAL',
     1'REAL ROOTS',5X,F8.4,
     2'AND',F8.4)
      GO TO 1
      END
```

10-#3

11 - ARRAYS AND SUBSCRIPTS

11-1 Arrays

Our next topics are arrays and subscripts. An array in FORTRAN is simply an interrelated set of constants stored in consecutive locations under one variable name. Each array element, also called a subscripted variable, is obtained by referring to the name of the array and a subscript telling which element of the array is under consideration: thus A(3) refers to the third element of A while TABLE (4,2) refers to the element in the 4th row, 2nd column of the two-dimensional (2D) array table. There are three types of valid arrays. 1D arrays can be thought of simply as lists of related items or, mathematically speaking, as vectors. 2D arrays can be thought of as tables with rows and columns, or mathematically as matrices. They are not stored in rows and columns, however, but are stored consecutively in a particular way which allows the computer to find each element easily through the use of a simple formula. We'll see how this works later on. 3D arrays are also used occasionally; they can be thought of as 3D lattices with rows, columns and planes or ranks. Thus THREED (2,3,1) is the element in the second row, third column and first plane of the array THREED.

11-2 The DIMENSION Statement

Any variable name may be an array, but if it is an array it cannot also be a simple variable within the same program. To define a variable name as an array we use a DIMENSION statement: form:

```
        DIMENSION VAR1(#), VAR2(#), etc.
```

This is a non-executable statement to tell the compiler a variable is an array; it is used by the computer to establish the maximum number of locations needed to store all the possible array elements. Every quantity treated as an array within a program must appear in a DIMENSION statement which, on the 3300, must come at the top of the program after PROGRAM name, but before any executable statements. With the name of each variable in the DIMENSION statement, there must be associated a number which represents the maximum number of elements the array may have. In the executable statements of the program the array name must always, with two exceptions, appear with a subscript which indicates which element of the array is being referenced. The two exceptions: the array name may be used with a subscript or by itself without a subscript in READ or WRITE statements and in the call to a subprogram; in every other executable statement the array name must be written with a subscript.

DIMENSION A(20), X(100), Z(3,4) sets up arrays A,X, and Z. A is a 1D array with a maximum of 20 elements. X is a 1D array with a maximum of 100 elements and Z is a 2D array with 12 elements arranged in 3 rows and 4 columns;

```
x x x x
x x x x
x x x x
```

```
READ(60,7) A(1) ← reads 1 element
HIGH = ABS(A(I)) ← absolute value of Ith element
Y = A(2) * A(3)
IF(A(3) - A(5) * A(4) .LE. A(7)) GO TO 6
```

<u>exceptions</u>

```
READ(60,7) A ← 20 elements READ
AMAX = FINDMAX(A,20) ← All of A sent to subprogram
```

<u>11-3 Subscript Forms</u>

Subscripts must always appear in parentheses. They do <u>not</u> count as one of the 1 to 8 characters of the variable name. In FORTRAN subscripts must have one of the following forms and they are always integer.

1. Integer constant	A (1)
2. Integer variable	A (I), I defined previously
3. Combinations:	All variables in these combinations must have been previously defined.
A (I + 1)	Integer variable + Integer constant
A (J - 2)	Integer variable - Integer constant
A (2 * K)	Integer constant * Integer variable
A (3 * IK + 1)	Integer constant * Integer variable + Integer constant
A (4 * L - 11)	Integer constant * Integer variable - Integer constant

In some cases it may be more advantageous to calculate one of the valid combinations and store it.

```
IP1 = I + 1
C(IP1) = A(IP1) + B(IP1)
```

may be cheaper than C(I + 1) = A(I + 1) + B(I + 1).

Any other form must be temporarily stored:

```
JK = J + K
 X = A(JK)
```

11-4 Exceeding DIMENSION Limits

If the value of the subscript exceeds the maximum limit of DIMENSION, strange things may happen, including the possibility of destroying part of your program or transferring to an illegal address. The newest version of the OS-3 compiler has a special option, the E parameter added to the FORTRAN control card, which checks for exceeding these limits, a very common error of all FORTRAN programmers.

11-5 Uses of Arrays

The primary advantages of arrays are: (1) the same name is used for all related elements, (2) all elements may be saved for future use; rather than requiring READ, calculate, READ, calculate with one value at a time, we can do all READing and then all calculating which is often a better way to program, (3) we can use the power of counting with an integer variable acting as a counter and also as a subscript. Thus:

```
  DIMENSION A(50)
  N = . . .
  I = 1
3 READ (60,1) A(I)
  I = I + 1
  IF(I - N) 3,3,4
4 . . .
```

All A's are saved and retrievable by referencing A with a subscript which has the appropriate value.

(4) we can do operations on and with vectors and matrices, both very important quantities in higher mathematics.

11-6 Examples Using Arrays

Consider this averaging problem with arrays which calculates the mean test score and then determines each student's deviation from the mean. Without arrays we would need to input the data again to allow those calculations of deviations which cannot be done before the mean is calculated. (See sheet 11-#1)

The answers would appear like this:

```
The average is          70.0

Student           Score          Deviation
   1               50              -20.0
   2               70                  0
   3               90               20.0
```

```
      PROGRAM AVERAGES
      DIMENSION SCORE(100)
      SUM=0.
      READ(13,7)N
    7 FORMAT(I3)
      I=1
    3 READ(13,8)SCORE(I)
    8 FORMAT(F5.0)
      SUM=SUM+SCORE(I)
      I=I+1
      IF(I-N)3,3,4
    4 XN=N
      AVG=SUM/XN
      WRITE(61,13)AVG
   13 FORMAT('0THE AVERAGE IS'
     1,F10.1//'0STUDENT',5X,
     2'SCORE  DEVIATION')
      I=1
   21 DEV=SCORE(I)-AVG
      WRITE(61,15)I,SCORE(I),DEV
   15 FORMAT(I6,F11.0,F11.1)
      I=I+1
      IF(I-N)21,21,22
   22 CALL EXIT
      END
```

11-#1

12 - OTHER TYPES OF CONSTANTS, VARIABLES & FORMAT SPECIFICATIONS

12-1 DECLARATION Statements

There are several other non-executable statements, which, if present, must also go at the top of the program with any DIMENSION statement. No particular order is required for these non-executable specification statements. Those statements we will discuss are used to change the normal modes of arithmetic. It is sometimes inconvenient to have variables beginning with I,J,K,L,M or N always being integer. Thus a programmer working with latitude and longitude would prefer to use variables LAT and LONG but because of their implicitly integer type it would appear he cannot do so. There is an easy way to get around this inconvenience and similar situations in which a variable starts with A-H or O-Z but it is wished to treat it as integer, e.g., if you wish to use the variable COUNT as counter.

12-2 The INTEGER and REAL Statements

We have explicit type DECLARATION statements which change implicit type and establish location in the mode defined. INTEGER list and REAL list use lists of variables and/or arrays separated by commas. You may define every variable as one or the other if you wish, but it is easier to use the implicit types when you are thoroughly familiar with them.

```
      PROGRAM CHECK
      DIMENSION X(50), A(20), TABLE(5,3), ISET(6)
      INTEGER TABLE, COUNT
      REAL LAT, LONG
      COUNT = 0
      READ (60,17) LAT, LONG            real
   17 FORMAT (2F10.2)
      COUNT = COUNT + 1                 integer
```

12-3 The DOUBLE PRECISION Statement

Occasionally the 10 or 11 precise digits automatically established for real variable locations on the 3300 are not sufficient for a particular application. Then the declaration statement DOUBLE PRECISION list may be used to change the variables in the list to 96 bit quantities with the 48 extra bits added to the mantissa. This gives 84 bits, which is equivalent to 25 or 26 decimal digits. If the 7 digits of an integer constant are not sufficient then DOUBLE(2) list changes all the variables in that list to DP integers with 48 bits - the largest number is then 15 digits long. All normal arithmetic is done in DP mode when mixed mode operations are present. DP constants cannot be defined easily on the 3300 so the contents of DP locations usually are the result of arithmetic expressions calculated in DP arithmetic. Also, unfortunately, OS-3 does not currently allow I/O of DP quantities. The usefulness of DP is thus somewhat curtailed.

12-4 The COMPLEX Statement

More useful than DP is the COMPLEX type statement COMPLEX list which establishes each variable in the list as 96 bits; 48 for the real part and 48 for the imaginary. Normal complex arithmetic is done by the five basic operations, but a series of special functions exists for forming complex constants, taking real and imaginary parts, square roots, logarithms, etc. For more information on the complex arithmetic package consult the Computer Center.

12-5 The FLOAT and IFIX Library Functions

As we have seen we can change the type of a variable for the entire program by using the type declaration statements. We may, however, wish a variable to be both integer and real at different times in the same program, particularly to avoid mixed mode, e.g.,

N sçores, mean = $\dfrac{\sum_{i=1}^{n} S_i}{n}$

```
    READ N
    I = 1
  3 READ X
    SUM = SUM + X
    I = I + 1
    IF(I-N) 3,3,4
    XMEAN = SUM/N
```

We count scores with integer N but need real N to divide without mixed mode. Without using mixed mode this can be done in two ways:

```
READ (3,7) N                              READ (3,7) N
.                                         .
.                                         .
.                                         .
XN = N                                    XMEAN = SUM/FLOAT (N)
XMEAN = SUM/XN
```

FLOAT is a library function to allow a change in variable type without actually changing the location N. There is also the opposite function IFIX with truncation of any fraction: J = K + IFIX(X). X is still real, only in integer form for this computation. When a mixed mode operation occurs, FLOAT is used by default to change integer values to real for that computation only.

12-6 Hollerith Constants

We have already discussed the bit structure of integer and real constants, but we can, on the 3300, also have a quantity called a Hollerith constant. This consists of one or two words filled with characters represented as binary coded decimals (BCD).

13 - HOLLERITH CONSTANTS; AW FORMAT; DO LOOPS

13-1 BCD Code

Integer or real numeric BCD notation requires 4 bits to represent the digits from values 0 to 9 and two extra bits to include all the other characters. Thus each BCD character is expressed with 6 bits, giving a total of 2^6 or 64 characters possible, the 48 of FORTRAN plus 16 others. 6 bits = 2 octal digits. There are 64 characters possible from 00_8 to 77_8. A is 21_8, B is 22_8 and so forth.

13-2 Forms for Integer and Real Hollerith Constants

The 24 bits of a word on the 3300 are separable into 4 sets of 6 and thus we can represent 4 BCD characters in every computer word. Hollerith constants are formed with nH as in the FORMAT specification, but unlike the specification $1 \leq n \leq 4$ defines an integer Hollerith constant, while $5 \leq n \leq 8$ defines a real Hollerith constant. Thus only n determines the type here. Characters are left justified and, if n < upper limit, the computer blank fills to the right.

```
NAME1 = 4HJOHN
NAME2 = 4H DOE
NAME3 = 3HJOE
```

stores	$\boxed{J}\boxed{O}\boxed{H}\boxed{N}$	in NAME1	24 bits of NAME1
	$\boxed{\ }\boxed{D}\boxed{O}\boxed{E}$	in NAME2	24 bits of NAME2
	$\boxed{J}\boxed{O}\boxed{E}\boxed{\ }$	in NAME3	24 bits of NAME3

while XNAME = 8HJOHN DOE

stores $\boxed{J}\boxed{O}\boxed{H}\boxed{N}\boxed{\ }\boxed{D}\boxed{O}\boxed{E}$ in 48 bits of XNAME

This type of variable has many uses as in grade reporting.

```
DIMENSION IGR (5)
IGR (1) = 1HF     stores     F
IGR (2) = 1HD                D
IGR (3) = 1HC                C
IGR (4) = 1HB                B
IGR (5) = 1HA                A
```

Note integer = integer and real = real for Hollerith constants or conversion from mode to mode may destroy some of the characters.

Integer Hollerith constant 1HA, 2HAB, 3HABC, 4HABCD

Real Hollerith constant 5HABCDE, 6H...,7H...8HABCDEF.

13-3 AW FORMAT for I/O of Hollerith Constants

To input or output BCD information we use either an integer or a real variable name in the list and AW FORMAT (or RW) which allows input or output of characters forming Hollerith constants into or from integer or real locations.

The limitations on w are $1 \le w \le 4$ for integer variables and $1 \le w \le 8$ for real variables; the variable type not the w indicates whether 4 or 8 characters can be stored. Again blank fill occurs to the right when w < upper limit .

```
      DIMENSION XNAME(3)
      READ (60,7) XNAME
    7 FORMAT (3A8)
```

reads a 24 character name into the 3 real words forming XNAME.

```
      WRITE (61,8) XNAME
    8 FORMAT ('0', 3A8)
```

writes 24 characters for XNAME.

13-4 A Return to Summation and Counting

We will now return to the topics of summation and counting. In our previous discussion we wrote a program to sum the integers from 1 to N. The heart of that program was:

```
      SUM = 0.
      X = 1.
      I = 1
    3 SUM = SUM + X
      X = X + 1.
      I = I + 1
      IF(I-N) 3,3,4
    4 . . . .
```

This could be improved in several ways:

(1) Count down

```
      SUM = 0.
      X = 1.
    3 SUM = SUM + X
      X = X + 1.
      N = N - 1
      IF(N) 4,4,3
```

(2) Integer arithmetic

```
      ISUM = 0
      I = 1
    3 ISUM = ISUM + I
      I = I + 1
      IF(I-N) 3,3,4
    4 . . . .
```

(3) Using XN = N as limit

```
      XN = N
      SUM = 0.
      X = 1.
    3 SUM = SUM + X
      X = X + 1.
      IF(X-XN) 3,3,4
    4 . . .
```

(4) Integer arithmetic count down

```
      ISUM = 0
    3 ISUM = ISUM + N
      N = N - 1
      IF(N) 4,4,3
    4 . . .
```

(5) Test first

```
  ISUM = 0
  I = 1
3 IF(I-N)2,2,4 or IF(I .GE. N) GO TO 4
2 ISUM = ISUM + I
  I = I + 1
  GO TO 3
4 . . .
```

13-5 Initializing, Incrementing and Testing: The DO Loop

This last form is very important.

```
  I = 1                       initialize
3 IF(I .GT. N) GO TO 4        test
  . . .
  I = I + 1                   increment
  GO TO 3                     loop
```

The steps labelled form the basis for the most difficult instruction in FORTRAN: DO loop. All are done automatically by DO which is the iteration instruction in FORTRAN. All the steps within a DO loop are repeated until some terminating condition is met. The values of the variables are usually changed on every pass through the loop.

Thus:

```
  ISUM = 0
  DO  1  I = 1,N
  ISUM = ISUM + I
1 CONTINUE
```

sums the integers from 1 to N without requiring individual steps on I being written.

14 - DO LOOPS CONTINUED

14-1 DO Loop Format

In DO loops the initializing, testing, incrementing and looping are all done automatically. The DO instruction is generally of the form:

$$\text{DO n index} = m_1, m_2, m_3$$

where n is the statement number of the last statement in the range of the DO, which is all the statements to be repeated, down to and including statement n. The index must be an integer variable. The value of this variable is defined only within the loop by the index parameters, m_1, m_2, m_3. These may only be integer variables or positive integer constants. m_1 is the initial value for the index. m_2 is the testing value and m_3 is the incrementing value. If the comma and the m_3 are omitted, m_3 is automatically set to 1.

14-2 Execution of DO Loops

During execution of a DO the index is initialized to m_1, then it is tested. If $\text{index} > m_2$, then we have a normal exit from the DO and transfer is made to the first executable statement following the statement numbered n. If the $\text{index} \leq m_2$, then all the statements in the range, down to and including statement n, are executed. When statement n has been executed, control is returned to the indexing section of the DO. The index is incremented by m_3 ($\text{index} = \text{index} + m_3$) and again tested against m_2. If there is a transfer of control statement within the range of the DO which causes transfer out of the range before the $\text{index} > m_2$, then this is said to be a non-normal exit. If no exit occurs we remain in the loop and repeat the steps within the loop, returning again to the indexing section when statement n is executed until a normal or non-normal exit does occur.

14-3 Rules for the Formulation of DO's

1. Control may not be passed from statements outside the DO to statements inside the DO; transfer only to the DO itself.

2. Control may be passed from inside the DO to outside (a non-normal exit). In this case the value of the index is available for further use. If we have instead a normal exit, the value of the index is not available. The index can then be reused in another DO or in any other way except as an array name, since it cannot be both an array and a simple variable.

Normal exit:

```
  DO 1 K = 1, N
  KSUM = KSUM + K
1 CONTINUE
  K = K + 1
```

Invalid since K undefined on normal exit

Non-normal exit:

```
  DO 1 I = 1, 15
  READ (60,2) X (I)
  IF(X(I) .LT. 0.) GO TO 9
1 CONTINUE
   ...
   ...
9 X(I) = -X(I)
```

3. Control may be passed from one statement inside the loop to another statement inside the loop, but all branches must ultimately reach the last statement in the range to keep the loop going to completion. Note that a statement reached from within a loop may appear to be physically outside the loop, but if it transfers back inside, it is still in the range.

4. The last statement in the range of a DO must be executable and since it signals the return to the indexing section of the loop, may not be a transfer of control statement, another DO or any other statement which would stop this looping procedure. A good rule to follow to avoid having to remember what cannot go at the end of a DO loop is to always use a CONTINUE statement as the last statement. This statement causes no overt action but allows continuation of normal operations. Thus:

```
   DO  17  JKLX = J1, K46B, M
17 CONTINUE
```

5. The index and its parameters must not be changed by statements in the range of the DO. Thus:

```
   DO  13  I = 1,N
   I = I + I
   N = N/2
13 CONTINUE
```

is illegal, but

```
             DO  13  I = 1,N
             . . .
          13 CONTINUE
             N = N - 1
             DO  14  I = 1,N
             . . .
          14 CONTINUE
```

is valid.

14-4 Sample Problem

Assume that the value of Y from the formula $Y = 41.298\sqrt{1+X^2+X^{1/3}E^X}$ is needed for values of X from 1.00 to 3.00 in steps of .02. We will need to generate a printed line for each X-Y pair. The solution is done most easily with a DO.

```
         PROGRAM FORMULA
         WRITE (61,28)
      28 ...
         DO  1  J = 100,300,2
         X = FLOAT (J) /100.  (Note J/100 gives wrong answer)
         Y = 41.298 * SQRT (1. + X * X) + X ** .333 * EXP (X)
         WRITE (61, 37) X,Y
      37 FORMAT (F14.2,F11.3)
       1 CONTINUE
         CALL EXIT
         END
```

15 - NESTED DO LOOPS; INPUT/OUTPUT OF ARRAYS

15-1 Nested DO Loops

DO loops may be nested, i.e., DO loops inside DO loops. When nested, the range of the inner DO must not exceed the range of the outer DO. The last statements may coincide however.

```
      DO  6  IJX = M1, M2, M3
      DO  8  K = 1, KMAS
      DO 10  I = 1,5
      DO 12  J = 2, N, 2
      DO 12  L = 2, M, 2
   12 -------
   10 -------
    8 -------
    6 -------
```

On the 3300 DO's may be nested up to 10 deep. In nested DO's the index of the outermost DO is initialized, then when the next DO is reached, its index is initialized and it retains control until a normal or non-normal exit is effected. The index of the outer DO is held constant. When statement n of the outer DO is reached, control returns to the indexing section of the outer DO. When the inner DO is reached again, its index is re-initialized and it is completed again with no change to the index of the outer loop. So each pass through the outer loop means that the entire inner loop is done.

15-2 Sample of Nested DO's

```
                         ISUM = 0
                         DO  10  I = 1,2
                         DO  10  J = 1,3
                         ISUM = ISUM + I + J
                      10 CONTINUE
                         WRITE ...
ISUM = 0; I = 1:   J = 1, ISUM = 0 + 1 + 1 = 2
                   J = 2, ISUM = 2 + 1 + 2 = 5
                   J = 3, ISUM = 5 + 1 + 3 = 9
                   J = 4>3 Complete with normal exit, return to
                           indexing of first DO.
          I = 2:   J = 1, ISUM = 9 + 2 + 1 = 12
                   J = 2, ISUM = 12 + 2 + 2 = 16
                   J = 3, ISUM = 16 + 2 + 3 = 21
                   J = 4>3 Complete (inner)
          I = 3>2  Complete (outer)
```

15-3 Use of DO Loops with Arrays

DO loops are used for iteration and counting. The most powerful feature of FORTRAN is the use of the index of the DO as the subscript of an array.

Sample program for finding the largest element of an array, assuming 10 elements. (See sheet 15-#1)

15-4 I/O and FORMAT Lists with Differing Numbers of Elements

As we have seen, arrays are very important quantities in FORTRAN. They are also somewhat awkward to work with and so special means exist for inputing and outputing arrays. Since the actual number of elements in an array may be less than the limit established in the DIMENSION and in many cases is variable, we need FORMAT rules to handle this variability. Previously, every I/O list had exactly the same number of variables as there were I,E,F and A specifications. If there are more I,E,F or A specifications in the FORMAT than elements in the list, the I/O of the variables in the list is completed with the left-most specifications and the remainder is ignored. Any Hollerith or spacing specifications between the last I,E,F,A and the first one ignored will be printed on output.

```
      WRITE (61,26) X,Y
   26 FORMAT (3H0X=,F6.1, 3X, 2HY=,F8.2, 3X, 2HI=,I5)
```

is executed as

```
      X = XXXX.X        Y = XXXXX.XX        I =
```

If, on the other hand, there are more items in the list than I,E,F or A specifications and the rightmost parenthesis is encountered with more items to be input or output, then a new record is automatically started and the computer continues the FORMAT from the rightmost left parenthesis and repeats from that point.

```
      WRITE (61,30) A,B
   30 FORMAT  (' ',F10.2)
```

A written in F10.2, right parenthesis encountered, new record started and B written on next line in F10.2.

```
      PROGRAM FINDMAXA
      DIMENSION A(10)
      READ(60,10)A
   10 FORMAT(10F8.2)
      AMAX=A(1)
      DO 1 I=2,10
      IF(A(I).GT.AMAX)AMAX=A(I)
    1 CONTINUE
      WRITE(61,12)AMAX
   12 FORMAT('OTHE LARGEST IS'  ,F10.2)
      CALL EXIT
      END
```

15-#1

15-5 Unlimited Groups

If, as indicated in 15-4, there are more items in the list than I,E,F or A specifications and the rightmost parenthesis is encountered with some list elements still to be input or output, then a new record is automatically started (the right parenthesis acts like /) and the computer repeats the FORMAT from the rightmost unlimited group, where an unlimited group is one or more FORMAT specifications inside a set of parentheses without a multiplier in front of the group. Thus in the FORMAT (1X,(F10.2),2(2F8.4,3X)) the (F10.2) is an unlimited group while the 2(2F8.4,3X) is not. The first three variables in a WRITE list will be printed as though the FORMAT were (1X,F10.2,2F8.4, 3X,2F8.4,3X), but after that, all the rest will be printed in (F10.2) and every time the right parenthesis of the unlimited group is reached now, a new record is automatically initiated. If no unlimited groups exist within the FORMAT, then the entire FORMAT is treated as an unlimited group. Consider this program segment:

```
     DIMENSION X(6)
     DO 1 I=1,6
   1 X(I)=I
     WRITE(6,10)X
  10 FORMAT(1X,(F10.2),F12.4)
```

this will print

```
     1.00    2.0000
    3.00
    4.00
    5.00
    6.00
```

```
     WRITE(6,11)X
  11 FORMAT(1X,2(F10.2),F12.4)
```

this will print

```
     1.00    2.00    3.0000
     4.00    5.00    6.0000
```

16 - INPUT/OUTPUT OF ARRAYS, CONT; PROGRAM EXAMPLES

16-1 Repeated Groups

We previously mentioned shorthand notation for specifications. (I5,I5)$\rightarrow$(2I5). We can also use this shorthand on repeated groups of specifications, with the use of extra parentheses.

```
(I10, F10.2, I5, F10.2, I5)→(I10, 2(F10.2, I5))
                                     rightmost
```

16-2 Input/Output of Arrays

There are three basic ways to input or output arrays. We can use the array name without subscripts.

```
      DIMENSION  A(10)
      READ (60,20)A
```

This reads all 10 elements of A. The FORMAT completely determines how many numbers per record and where on the records the numbers fall. We can also use a DO loop with N = number of elements. With N = 10

```
      DO  1  I = 1,N
      READ (60,20) A(I)
    1 CONTINUE
```

Since the READ is executed on every pass through the loop, we have N cards read and only one number per card, so the FORMAT is not in control. Finally we have the implied DO which may be used only in I/O statements.

```
      READ (60,20)  (A(I), I = 1,N)
```

Again the FORMAT is in complete control, because the READ is only executed once; in addition the value of N may change without requiring a change in the FORMAT. With the following FORMAT

```
     20 FORMAT (10F8.0)
```

ten numbers per card are read; if N>10, then the cards are read with ten numbers per card until exactly N numbers are input, depending only on the FORMAT rules we have just discussed. If the FORMAT is (F8.0), then 1 number per card is read, or if (5F8.0), then 5 numbers per card are read. General form of the implied DO:

READ
WRITE (array (sub), index = m_1, m_2, m_3)

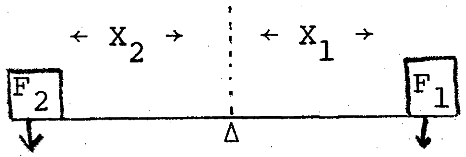
Rules on subscripts and on the index and its parameters are as before. Usually sub = index. Implied DOs are preferable because they allow flexibility.

```
        WRITE (61,7) (X(I), Y(I), I = 1,N)
      7 FORMAT (1H0, F10.2, F10.4)
```

writes answers $\underline{X_1}$ $\underline{Y_1}$ automatically, with double spacing.
$\underline{X_2}$ $\underline{Y_2}$
$\underline{X_3}$ $\underline{Y_3}$

16-3 Example 1 - Forces, Moments and Center of Gravity

Consider this problem from physics:



When the system is balanced, the sum of the moments = 0, where each moment = F_iX_i. The total moment = $\sum_{i=1}^{n} F_iX_i$.

Without arrays:

```
        PROGRAM TEST
        READ (5,7) N
      7 FORMAT (I5)
        AMOM = 0.
        DO  100  I = 1,N
        READ (5,8) F,X
      8 FORMAT (2F10.2)
        AMOM = AMOM + F * X
    100 CONTINUE
        WRITE (61,19) AMOM
     19 FORMAT (' TOTAL MOMENT =', F12.2)
        CALL EXIT
        END
```

The values of F and X are lost after being read, with only the last set available at the end. If further computations on F and X are wanted, then they would have to be reread - an expensive and slow process.
With arrays:

```
      PROGRAM TEST 2
      DIMENSION X(100), F(100), AM(100)
      READ (5,7) N
    7 FORMAT (I5)
      AMOM = 0.
      READ (5,8) (F(I), X(I), I = 1,N)
    8 FORMAT (8F10.2) or (2F10.2)
      DO  100  I = 1,N
      AM(I) = F(I) * X(I)
      AMOM = AMOM + AM(I)
  100 CONTINUE
      WRITE (61,19) AMOM
   19 FORMAT (....)
      WRITE (61,47) (F(I), X(I), AM(I), I = 1,N)
   47 FORMAT (3F14.2)
      CALL EXIT
      END
```

If the F_i are weights, the center of gravity C.G. = $\frac{\Sigma \text{ moments}}{\Sigma \text{ wts}}$

```
      SUMF = 0.
      AMOM = 0.
      DO 1 I = 1,N
      AMOM = AMOM + F(I) * X(I)
      SUMF = SUMF + F(I)
    1 CONTINUE
      CG = AMOM/SUMF
      WRITE (61,13) CG
```

After CG is computed, we redefine the value of X so the elements of X give the distances from the Center of Gravity rather than from the origin of the coordinate system.

```
      DO  2  I = 1,N
      X(I) = X(I) - CG
    2 CONTINUE
      WRITE (61,19)(F(I), X(I), I = 1,N)
```

Without arrays this last part would be impossible since all the X_i would not be available.

16-4 Example 2 - Means and Standard Deviations

A statistical example - mean and standard deviation of a set of test scores.

$$\mu = \sum_{i=1}^{n} X_i/n \qquad \sigma = \sqrt{\frac{\Sigma(Xi-\mu)^2}{n-1}}$$

which is computed with fewer operations by the formula

$$= \sqrt{\frac{\Sigma Xi^2 - (\Sigma Xi)^2/n}{n-1}}$$

where ΣXi^2 = sum of squares of the scores and $(\Sigma Xi)^2$ = the square of the sum of the scores. (See sheet 16-#1)

16-5 Example 3 - A Grading Problem with Many Special Features

The next program calculates individual grade point averages from total grade points and total units as calculated course by course. The important features are the use of INTEGER type declaration, the FLOAT library function, Aw FORMAT and Hollerith constants. (See sheet 16-#2)

```
      PROGRAM MEANSD
      DIMENSION SCORE(100)
      READ(60,13)N
   13 FORMAT(I3)
      READ(60,14) (SCORE(I),I=1,N)
   14 FORMAT(16F5.0)
      SUM=0.
      SUMSQ=0.
      DO 21 I=1,N
      SUM=SUM+SCORE(I)
      SUMSQ=SUMSQ+SCORE(I)**2
   21 CONTINUE
      XN=N
      SMU=SUM/XN
      SD= SQRT((SUMSQ-SUM**2/XN)
     1/(XN-1.))
      WRITE(61,19)XMU,SD
   19 FORMAT('OMEAN = ',F6.1,
     1'STANDARD DEVIATION = '
     2,F6.2)
      CALL EXIT
      END
```

16-#1

```
      PROGRAM GPAS
      DIMENSION NUN(99),GR(99)
     1,COURSNAM(99),SNAME(2)
      INTEGER GR,GP
      WRITE(61,20)
   20 FORMAT('1',3X,'NAME',17X,
     1'GP',5X,'UNITS',10X,'GPA')
   10 READ(6,15)SNAME,NC
   15 FORMAT(2A8,I4)
      IF(EOF(6))CALL EXIT
      READ(6,25) (COURSNAM(I),
     1NUN(I),GR(I),I=1,NC)
   25 FORMAT(A8,I2,1X,A1)
      TGP=0.
      TUN=0.
      DO 5 I=1,NC
      GP=4
      IF(GR(I).EQ.1HB)GP=3
      IF(GR(I).EQ.1HC)GP=2
      IF(GR(I).EQ.1HD)GP=1
      IF(GR(I).EQ.1HF)GP=0
      TGP=TGP+FLOAT(GP*NUN(I))
      TUN=TUN+FLOAT(NUN(I))
    5 CONTINUE
      GPA=TGP/TUN
      WRITE(12,35)SNAME,TGP,
     1TUN,GPA
   35 FORMAT('',2A8,2(5X,F5.0)
     1,8X,F6.2)
      GO TO 10
      END
```

16-#2

17 - TWO DIMENSIONAL ARRAYS; SUBPROGRAMS

17-1 Storage of Two Dimensional Arrays in Memory

As we mentioned briefly, 2D arrays are not actually stored in tabular form in the computer's memory, but rather in sequential locations so that an array such as is stored in consecutive locations in column order as 1,2,6,3,3,4,1,0,7,5,9,4

```
1  3  7
2  4  5
6  1  9
3  0  4
```

that is A(1,1), A(2,1), A(3,1), A(4,1), A(1,2), A(2,2), etc. - the first subscript always varies the fastest. We must be aware of the form in which 2D arrays are stored before we can input or output 2D arrays properly.

17-2 Calculation of Location of Array Element

The computer is able to determine which element is required by use of a simple formula. If array A is DIMENSIONed NR by NC where NR is the maximum number of rows and NC the maximum number of columns (both actually numbers), then the formula the computer uses to find the position of element A(I,J) in the NR * NC consecutive locations of array A is

LOC = I + (J-1) * NR where LOC is from 1 to NR * NC.

Thus DIMENSION A(4,3) has NR = 4, NC = 3. There are 12 elements and LOC goes from 1 to 12.

```
A(1,1) is in position    1 + (1-1) * 4 = 1
A(2,1) is in position    2 + (1-1) * 4 = 2
A(1,2) is in position    1 + (2-1) * 4 = 5
A(4,2) is in position    4 + (2-1) * 4 = 8
```

and so on.

17-3 Input/Output with 2D Arrays

There are several ways to do I/O with 2D arrays. Consider:

```
(1)         DIMENSION B(6,4)
            READ (60,17) B
         17 FORMAT (    )
```

This reads all 24 elements in <u>column</u> order with the number of records and the number of values per record completely determined by the FORMAT.

```
(2)      DO  1  I = 1,6
         DO  1  J = 1,4
         READ (60,19)  B(I,J)
      19 FORMAT (    )
       1 CONTINUE
```

This double DO loop inputs all the elements of B in the more natural <u>row</u> order but with only one number per record since the READ gets executed 24 times - one record per READ. For <u>column</u> order we simply reverse the order of the DOs.

```
(3)      DO  1  I = 1,6
         READ (61,21)  (B(I,J),  J = 1,4)
      21 FORMAT (    )
       1 CONTINUE
```

This implied DO-in-normal-DO reads the data in <u>row</u> order and the FORMAT would be best as (4F10.0), thus requiring 6 cards with one row on each card.

```
(4)      READ (60,23)  ((B(I,J), I=1,6), J = 1,4)
and      READ (60,24)  ((B(I,J), J=1,4), I = 1,6)
```

are the simplest and best forms for inputting B in <u>column</u> and <u>row</u> order respectively. The double implied DO allows either order with the FORMAT in complete control as to the number of records and the number of values on each record. <u>Row</u> order is more natural for English speaking people who read left to right. Hence the last form is the best.

<u>17-4 Example of 2D Arrays</u>

Here is an example dealing with a 2D array. Consider the following table in which the rows represent different brands of a product and the columns represent the cities in which the product was sold. The entries are sales reduced to a common population base - the sales index.

	SF	PORTLAND	SEATTLE
Brand W	88	89	91
X	67	75	78
Y	95	86	89
Z	90	83	83

```
      DIMENSION  TABLE (4,3)
      READ (60,10)   ((TABLE (I,J), J=1,3), I=1,4)
   10 FORMAT (3F5.0)
```

Possibility 1 - total and average index of Brand W. The total is the sum of the elements of row 1.

```
      X = 0.
      DO  11  J = 1,3
      X = X + TABLE (1,J)
   11 CONTINUE
      AVG = X/3.
```

Possibility 2 - total and average index of all brands sold in PORTLAND. The total is the sum of the elements of column 2.

```
      Y = 0.
      DO  12  I = 1,4
      Y = Y + TABLE (I,2)
   12 CONTINUE
      AVG = Y/4.
```

Possibility 3 - total indexes of each of the four brands in all cities. Each brand has a total - the sum of the elements in the brand-row. We use an array to store the four row sums.

```
      DIMENSION TABLE (4,3),  ZROW (4)
      DO  14  I = 1,4
      ZROW (I) = 0.
      DO  14  J = 1,3
      ZROW (I) = ZROW (I) + TABLE (I,J)
   14 CONTINUE
```

Possibility 4 - total indexes in each of the three cities of all brands - the three column sums.

```
      DIMENSION TABLE (4,3), ZCOL (3)
      DO  15  J = 1,3
      ZCOL (J) = 0.
      DO  15  I = 1,4
      ZCOL (J) = ZCOL (J) + TABLE (I,J)
   15 CONTINUE
```

Possibility 5 - total index and average of all brands in all cities - the sum of all the entries in the table.

```
      TOTAL = 0.
      DO  16  I = 1,4
      DO  16  J = 1,3
      TOTAL = TOTAL + TABLE (I,J)
   16 CONTINUE
      AVG = TOTAL/12.
```

17-5 Reasons for Use of Subprograms

We have mentioned the existence of pre-written library functions several times and have used some of them in our examples. We also have the capability of writing subprograms ourselves, both FUNCTIONs and SUBROUTINEs. There are several reasons for the use of subprograms. (1) To allow easy use of common mathematical operations such as SIN, EXP, etc. (2) To allow repetition of a series of steps needed at several places within one program. (3) To allow segmenting of a large program into smaller portions, each of which can be compiled and tested separately. (4) To allow programming by one installation for use by another without the worry of different input and output requirements.

17-6 Compilation and Execution of Subprograms

Usually the subprogram is a complete entity with its own starting and stopping statements. The set of statements which make up a subprogram are compiled together but can only be executed in conjunction with a main program, (starting with PROGRAM name and ending with END), which references the subprogram.

```
Main program                Subprogram

____________                __________

____________                __________

____________                RETURN

Call to subprogram          END

____________

____________

____________
```

17-7 The END and RETURN Statements

As we have mentioned, the last statement in every program or subprogram is END. Thus all FUNCTIONs and SUBROUTINEs must have END as the last card, again to signal the compiler the end of related statements to be compiled. The last executable statement must be RETURN - establishing linkage to the point from which the subprogram is called. There may be more than one RETURN statement.

17-8 The FUNCTION Name Statement and its Parameters

The first card of a FUNCTION must be:

FUNCTION name (parameters separated by commas)

thus FUNCTION FN(X,Y,N).

The parameters must be variable names which are used in the body of the FUNCTION. They are dummy variable names which are replaced by the addresses of the actual parameters to be used during the call to and the execution of the function.

```
      READ (60,10) X                      FUNCTION SIN(A)
      XRAD = X * 3.1415926536/180.        ----------
      Y = SIN(XRAD)                       ----------
      WRITE (61,11) X,Y                   SIN = some function of A
   11 ....                                RETURN
                                          END
```

18 - SUBPROGRAMS CONTINUED; OTHER SPECIFICATION STATEMENTS

18-1 FUNCTIONS

We have seen that (1) the first statement in every FUNCTION subprogram must be FUNCTION name (parameters), (2) the last statement must be END to signal the end of related statements to be translated, (3) there must be one or more RETURN statements. (The RETURN is always the last executable statement in each branching segment within the FUNCTION), and (4) the name of the function must appear on the left of an = sign or in a READ statement in order that the value of the function is stored in that location. Thus the name determines the type of function. An integer name ⇒ an integer function and a real name ⇒ a real function. Only one value is returned to the main program - that stored in the location given by the name of the function.

In the main program VAR = FUNCTION name (actual parameters) references the function with the given name with the actual parameters, which may be any arithmetic expression - they are not limited to variable names as are the dummy parameters. The function reference itself may be alone as above or as part of a compound arithmetic expression.

```
      XYZ = Z*B-SUMFN (X,Y,Z)
```

18-2 Sample Problem with FUNCTION

Here is a main program and the function it uses to evaluate the largest element of an array. (See sheet 18-#1)

18-3 SUBROUTINES

In contrast to FUNCTIONs which return only one value, we have SUBROUTINES which may return one or more values or none at all. As with FUNCTIONs, the last statement in a SUBROUTINE must be an END and there must be one or more RETURNs to generate the

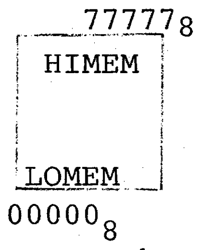
```
      PROGRAM MAIN
      DIMENSION X(500)
      READ(60,10)NUM,(X(I),I=1,NUM)
   10 FORMAT(I3/(10F8.0))
      BIG=XMAX(X,NUM)
      WRITE(61,30)BIG
   30 FORMAT(F16.7)
      CALL EXIT
      END

      FUNCTION XMAX(A,N)
      DIMENSION A(1)
      XMAX=A(1)
      DO  1  I=2,N
      IF(A(I).GT.XMAX)XMAX= A(I)
    1 CONTINUE
      RETURN
      END
```

18-#1

correct linkage for return to the main program. The first statement must be SUBROUTINE name (parameters). Again the parameters are dummy names which are replaced by the actual parameters when the calling program calls the SUBROUTINE. To reference a SUBROUTINE: since no value or more than one value may be returned, we cannot use an assignment statement, so we use a CALL such as: CALL SUBNAME (actual parameters). Thus the name of the SUBROUTINE does not determine the values returned; the argument list does. Again the actual parameters may be any arithmetic expression but the parameters in the statement SUBROUTINE name (parameters) must be variables only. Example: Mean and Standard deviation of a set of scores with a SUBROUTINE. (See sheet 18-#2)

18-4 Other Specification Statements

There are four other non-executable specification statements of importance. Each, when used, must appear at the top of the program after PROGRAM name along with any DIMENSION or type declaration statements. Symbolically, the computer's memory looks like this:



LOWMEM is the lower part of memory with addresses near 00000_8 and HIMEM is the upper part of memory with addresses near 77777_8.

FORTRAN programs when compiled correctly are loaded into HIMEM.

18-5 COMMON

The area starting at location 00020_8 is set aside as the COMMON area. The values of variables placed in COMMON are thus accessible from subprograms by referencing positions relative to the beginning of COMMON. Large programs unable to fit in available memory can be segmented and, with the use of COMMON, quantities determined in one segment will be available to the other segments. DIMENSION information may also be included in the COMMON list. The variables in the list, including any arrays, are stored in consec-

```
      PROGRAM SCORES
      DIMENSION X(100)
      READ(60,10)N,(X(I),I=1,N)
   10 FORMAT(I3/(10F8.0))
      CALL MEANSD(X,N,XBAR,SD)
      WRITE(61,12)XBAR,SD
   12 FORMAT('MEAN',F10.1,'SD'
     1,F10.2)
      CALL EXIT
      END

      SUBROUTINE MEANSD(A,N,XMU,SD)
      DIMENSION A(1)
      SUM=0.
      SUMSQ=0.
      DO 1 I=1,N
      SUM=SUM+A(I)
      SUMSQ=SUMSQ+A(I)**2
    1 CONTINUE
      XN=N
      XMU=SUM/XN
      SD=SQRT((SUMSQ-SUM**2/XN)
     1/(XN-1.0))
      RETURN
      END
```

18-#2

utive locations starting at the beginning of the COMMON area.

COMMON A, B, X(25), I, JK sets up 29 consecutive locations, 27 real and 2 integer, in the COMMON area. Values stored in those variables in a main program may be accessed in a subprogram without an argument list with a statement which matches in type but is not necessarily identical to that in the main program. Thus

```
          COMMON X,Y,Z(25), J,L or COMMON Z(27), I(2)
```

matches the COMMON above. A variable in COMMON need not be included in the argument list of a subprogram as long as both the main program and the subprogram have COMMON statements similar to the one above to allow both to access the same quantities. While the 3300 initializes all locations established in a program before execution to zero, those declared in the COMMON area are not zeroed.

18-6 EQUIVALENCE

The EQUIVALENCE statement is used to allow two or more variables to share a single location. It may be used to join programs written by different programmers or simply to save storage space. Thus, once a particular variable is no longer needed, its location can be used with a different name. EQUIVALENCE statements are often used in conjunction with arrays, particularly those in COMMON.

```
             COMMON  IN(150), B, N, X(35), A(20,10)
             EQUIVALENCE  (IN(1), IYR), (IN(2), MONTH), (IN(3), IDAY),
            1(IN(4), IHR)
```

IN(1) through IN(4) have alternate names IYR, MONTH, etc., either name may be used to refer to the contents of the first four elements in the COMMON area.

18-7 Labelled COMMON

The 3300 now allows labelled (or block) COMMON, one or more areas which immediately precede the program in HIMEM. COMMON/ blockname/list is the form. These blocks allow shorter COMMON statements in several parts; each part can be used or not in each subprogram - all blocks would probably be included in the main program.

```
          COMMON / XYBLOCK/ X(15), Y(15)
          COMMON / INTS / I, J, K, LM(47)
```

18-8 DATA

The last of the four statements is DATA. This statement allows constant values to be stored in variable locations at compilation time rather than at execution time. The CDC 3300 unfortunately has a non-standard form for this instruction.

```
          DIMENSION NAME (2), SUM (1500)
          DATA  (PI = 3.1415926536),  (E = 2.7181828184),
         1(NAME = 4HJOHN, 4H DOE),  (SUM = 1500(0.))
```

Thus at compilation time locations PI and E are established with their correct values. The integer array NAME has 2 Hollerith constants stored in it and the 1500 elements of SUM are zeroed quickly and efficiently. Care must be taken to avoid mixed mode across the equal sign in DATA statements since no conversion can take place. Variables in unlabelled COMMON should not be initialized with DATA statements. The DATA statement is most often used in conjunction with labelled COMMON blocks.

19 - ENTERING AND EDITING FORTRAN FROM TELETYPE

19-1 The OS-3 EDITOR

We will now finish our discussion of FORTRAN by entering and editing some FORTRAN programs directly from the teletype. In order to do this properly we need to spend some time discussing the OS-3 EDITOR. The EDITOR is really another language which in turn is translated into machine language. Since the EDITOR is a special purpose language designed to allow the entering and editing of programs and data directly from an on-line keyboard, relatively few instructions are needed and the task of using the EDITOR will not be as difficult as you might think. In any case we will not attempt to present all the available instructions for the EDITOR but rather those most commonly used. To use the EDITOR we must first log on a keyboard device. For our purposes we will assume this to be a teletype.

19-2 Logging On

```
                                     OFF
To log on a teletype, turn the                        switch to LINE.
                                  LINE      LOCAL
```

Hold down the CTRL key on the keyboard and hit A --[CTRL,A]. This signals the computer that you wish to log on. The computer responds with a # - the main symbol of control or command mode. You then have about 20 seconds to type your job number, a comma, and your user code, followed by a carriage return [CR]. If you take more than 20 seconds, you must repeat the steps.

```
[CTRL,A]
#703001, JSSI  [CR]
```

The computer then crosses out your number and prints the date, the time and the terminal number if your number is valid. It then returns another #.

#TIME = 10 establishes a 10 second limit (which can be changed later if necessary)

#*HI or #*SCOOP generates important information as to the amount of time remaining on the job number, the number of save file blocks currently in use and the number of users currently time-sharing.

#DIRECTORY gives a list of all save file block names as of the preceding night.

There are many other general features of the computing system available in this mode but we will introduce only those we need when we need them.

19-3 Entering Programs with the EDITOR

To enter a program directly from the teletype we proceed as follows:

#EDIT - this calls in the EDITOR language translator. The EDITOR has its own symbol,]. In EDIT mode we have available the EDITOR scratch pad, an area in memory in which we can create programs or data files or into which we can bring an already created program or data file and then make changes with the use of the EDIT instruction set.

]INPUT - this instruction allows us to use the scratch pad to enter our own program. Each line in the scratch pad is given a sequence number and the computer lists this sequence number next.

19-4 Program to Calculate Mean of a Set of N Test Scores

```
]INPUT
00001:       PROGRAM MEAN           skip to column 7
00002:       DIMENSION SCORE(100)
00003:       READ(60,10)N           read N from unit 60=TTY
00004:    10 FORMAT(I3)
00005:READ@        READ(60,11)      forgetting to skip to col. 7,
00006:      1SCORE(I),I=1,N)        @ wipes out line and we start
00007:    15 FORMAT(4F5.0)          again
00008:       SUM=0.
00009:       DO 1 I=1,N             continuation in column 6 on
00010:       SUM=SUM+SCORE(I)       line 6 was required for TV pic-
00011:     1 CONTINUE               ture
00012:       XMEAN=SUM/FLOAT(I)
00013:       WRITE(61,17)XMEAN
00014:    17 FORMAT('0MEAN =',F6.1)
00015:       CALL EXIT
00016:       END                    when completed, we return to
00017:                              EDIT mode with [ESC] or [ALT.
                                    MODE].
```

```
]OUT,SCORES
```

Save program by sending it to disk file named SCORES.

```
]FORTRAN,I=SCORES,R
```

Call FORTRAN compiler with input from SCORES: compile, load machine language program and execute if no errors.

```
ERRORS FOR MEAN

LABELS NOT REFERENCED
  15

ERROR 0501 AT    1 STATEMENT PAST  10

PARENTHESES DO NOT MATCH

LOADING DELETED
```

Since we had compilation errors, no loading can occur.

```
#EDIT
```

Back to the drawing board!

```
]FIN,SCORES
```

Bring the present version of SCORES back into the scratch pad area

```
]LIST
```

This lists the contents of the scratch pad but we will omit the listing for space considerations.

Line 6 had a missing parenthesis, so we insert the parenthesis with a special edit instruction, SARL, which searches for a given string of characters and replaces any occurrences of that string with another string.

```
]SARL,6,,/1S/,/1(S/
00006:     1(SCORE(I),I=1,N)
]OUT,SCORES
```

We again put our newest version on file.

```
]FORTRAN,I=SCORES,R
```

Again we attempt to compile and execute.

```
ERRORS FOR MEAN

LABELS NOT REFERENCED
  15

UNDEFINED STATEMENT LABELS
  11
```

But again we have compilation errors, even more serious than before; we need to carefully examine the error messages and completely correct all the errors, it's getting expensive!

```
ERROR 2700 AT    1 STATEMENT PAST  10
 THE FORMAT STATEMENT REFERENCED DOES NOT EXIST

LOADING DELETED
```

The first error message indicates that statement number 15 is not referred to by any other statements, while statement 11 has been referred to, according to the second message, but no statement 11 exists. The last error message indicates that what has probably happened is that a FORMAT statement with statement number 15 exists but is not referenced, while a FORMAT which is supposed to have statement number 11 is missing. The logical conclusion is that statement 15 should really be 11, or that the reference to 11 should be changed to a reference to 15.

```
#EDIT
```

So we edit again.

```
]FIN,SCORES
```

We bring SCORES back into the scratch pad.

```
]LIST,5,6
00005:        READ(60,11)
00006:       1(SCORE(I),I=1,N)
```

We list lines 5 and 6 since by referring back we know we have a READ using FORMAT 11.

```
]LIST,7
00007:    15 FORMAT(4F5.0)
```

We also need line 7.

```
]PXX2

#MI

]SARL,6,,/1(/,/1/
00006:       1SCORE(I),I=1,N)

]OUT,SCORES

]FORTRAN,I=SCORES,R
```

Feedback on the teletype line unexpected--returns us to control mode. MI sends us back to what we were doing before. If we forget what we are doing we may end up uncorrecting something which is correct now. Out it goes again; we really didn't pay much attention to the error messages.

Again we get error messages and again we will have to edit.

```
#EDIT

]FIN,SCORES

]LIST,5,7
00005:        READ(60,11)
00006:       1SCORE(I),I=1,N)
00007:    15 FORMAT(4F5.0)
```

List lines 5 through 7.

```
]SARL,6,,/1S/,/1(S/
```

We correct this again, hopefully for the last time.

After the SARL instruction is executed, line 6 is written again on the teletype:

```
00006:     1(SCORE(I),I=1,N)
```

```
]SARL,5,,/11/,/15/
00005:      READ(60,15)
```

At last we have corrected the statement number error!

```
]OUT,SCORES
```

We out the program to disk file

```
]FORTRAN,I=SCORES,R
```

```
NO ERRORS FOR MEAN
RUN
```

Hurray! No compilation errors. The computer prints RUN to show it is beginning execution.

```
  3
   50   60   70
```

We enter the number of scores, 3. We enter the values for each of the scores.

```
MEAN =  45.0
```

The answer (?) is printed in the form indicated by our FORMAT.

```
END OF FORTRAN EXECUTION
```

CALL EXIT has been executed.

Since the mean of the three scores we entered is definitely not 45.0, we must have made a logic error in our program. Since it is the mean that is wrong, let's return to the editor, bring our program back into the scratch pad area and find the mistake.

```
#EDIT
```

```
]FIN,SCORES
```

```
]SARL,,,/MEAN/
00001:      PROGRAM MEAN
00012:      XMEAN=SUM/FLOAT(I)
00013:      WRITE(61,17)XMEAN
00014:   17 FORMAT('0MEAN =',F6.1)
```

We look through the whole program for the string of characters MEAN and list all occurrences.

```
]
TIME CUT
#TIME=30
#GO
```

Oops! Our 10 second time limit has just been exceeded. We reset the limit and GO back to the editor.

```
]SARL,12,,/I/,/N/
00012:      XMEAN=SUM/FLOAT(N)
```

The culprit is the I in the denominator of line 12.

```
]FILE,SCORES
```

Another way of generating a file on disk.

```
]FORTRAN,I=SCORES,R

 NO ERRORS FOR MEAN
RUN
```

Again we have no compilation errors.

```
  3
   50   60   70
```

We enter the number of scores.
We enter the scores.

```
MEAN = 60.0
```

SUCCESS!!

```
END OF FORTRAN EXECUTION
```

20 - ENTERING AND EDITING FORTRAN FROM TELETYPE, PART II

20-1 A Sample FORTRAN Program

```
#FORTRAN,I=INPUT,R
```

FORTRAN program to illustrate power and convenience of on-line interaction of user with computer and at same time to list and explain major EDIT instructions.

```
 NO ERRORS FOR INPUT
RUN
```

```
 SOME USEFUL INSTRUCTIONS
 APPEND
 ERASE
 FIN
 INPUT
 INSERT
 LIST
 MOVE
 OUT
 REP
 RESEQ
 SARL
```

A list of the most useful instructions in the EDITOR.

```
 FOR EXPLANATIONS OF EACH
 FOLLOW DIRECTIONS
```

Directions for obtaining information (from the program) on any of the instructions or terminating.

```
 ENTER INSTRUCTION OR [CTL,W]
 APPEND
```

User enters instruction he wants explained.

```
 APPEND
 ADDS TEXT WHICH FOLLOWS
 AFTER THE LAST LINE
 IN THE SCRATCH PAD
```

Program prints format of instruction and explains what the instruction does.

```
 ENTER INSTRUCTION OR [CTL,W]
 ERASE
```

On to the next instruction to be explained.

```
 ERASE,N1   or   ERASE,N1,N2
 ERASES LINE N1 ONLY OR
 ALL LINES FROM N1 THRU N2

 ENTER INSTRUCTION OR [CTL,W]
 FIN

 FIN,NUMBER  OR  FIN,FILENAME
 ENTERS INFORMATION INTO
 THE SCRATCH PAD FROM EITHER
 SCRATCH FILE WITH LUN=NUMBER
 OR SAVE FILE FILENAME
```

```
ENTER INSTRUCTION OR [CTL,W]
INSERT

INSERT,N
INSERTS LINES WHICH FOLLOW
AFTER LINE NUMBER N

ENTER INSTRUCTION OR [CTL,W]
INPUT

INPUT
CLEARS SCRATCH PAD, PROVIDES
SEQUENCE NUMBER FOR EACH LINE
OF TEXT WHICH USER ENTERS
LINE BY LINE FROM KEYBOARD

ENTER INSTRUCTION OR [CTL,W]
LIST

LIST
LISTS ENTIRE SCRATCH PAD

LIST,N1
LISTS LINE N1 ONLY

LIST,N1,N2
LISTS ALL LINES N1 THRU N2

ENTER INSTRUCTION OR [CTL,W]
MOVE

MOVE,N1,N2,N3
MOVES LINE N1 THRU N2
AFTER LINE N3

ENTER INSTRUCTION OR [CTL,W]
OUT

OUT,NUMBER  OR  OUT,FILENAME
OUTPUTS INFORMATION FROM
THE SCRATCH PAD TO EITHER
SCRATCH FILE WITH LUN=NUMBER
OR TO SAVE FILE FILENAME

ENTER INSTRUCTION OR [CTL,W]
REP

ENTER
REP,N
REPLACES LINE N WITH LINES
WHICH FOLLOW
```

```
ENTER INSTRUCTION OR [CTL,W]
RESEQ

RESEQ
CAUSES RESEQUENCING
OF THE LINE NUMBERS

ENTER INSTRUCTION OR [CTL,W]
SARL

SARL,N,M,/STRING/,/REPLACE/
SEARCHES LINES N THRU M
FOR CHARACTERS IN STRING,
REPLACES ALL OCCURRENCES
WITH REPLACEMENT STRING,
THEN LISTS ALL CORRECTIONS

ENTER INSTRUCTION OR [CTL,W]
[CTL,W]
```

The CONTROL key is held down and the W is struck. [CTL,W] sends an End-of-file to the computer.

```
END OF FORTRAN EXECUTION
```

20-2 COPY

One of the most useful of the general features of the computer system is the COPY command. We COPY from an input unit, usually a save or scratch file or the card reader (if our input is on cards), to an output unit - another save or scratch file or the line printer or teletype (if just a listing is required).

```
#COPY, I=unit or file, O=unit or file
```

This is the most commonly used form. The input unit is 60 if the "I=" parameter is omitted, while the output unit is 61 if the "O=" parameter is omitted.

20-3 Combinations and Permutations

Now we will use the COPY command to print out the definitions and formulas pertaining to combinations and permutations which have been placed on files COMB, PERM and FORMULAS. Then we will write and execute a FORTRAN program with a SUBROUTINE and a FUNCTION to calculate permutations and combinations from the given formulas.

```
#COPY,I=COMB
   THE NUMBER OF COMBINATIONS
   OF N OBJECTS TAKEN R AT A TIME
   IS THE NUMBER OF WAYS IN WHICH
   R OF THE N OBJECTS CAN BE CHOSEN
   WITHOUT REGARD TO ORDER.

   FOR 4 OBJECTS A,B,C,D TAKEN
   3 AT A TIME WE HAVE

   ABC  ABD  ACD  BCD
```

COPY from file COMB to unit 61 - the teletype

```
#COPY,I=PERM
   THE NUMBER OF PERMUTATIONS
   OF N OBJECTS TAKEN R AT A TIME
   IS THE NUMBER OF ARRANGEMENTS
   THAT CAN BE MADE CONSIDERING
   THE ORDER IN WHICH THE OBJECTS
   ARE TAKEN.

   FOR 4 OBJECTS A,B,C,D TAKEN
   3 AT A TIME WE HAVE

   ABC ACB BAC BCA CAB CBA
   ABD ADB BAD BDA DAB DBA
   ACD ADC CAD CDA DAC DCA
   BCD BDC CBD CDB DBC DCB
```

COPY from file PERM

```
#COPY,I=FORMULAS
   THE ALGEBRAIC FORMULAS

   P(N,R)=N!/(N-R)!

   WHERE N!=1.2.3...(N-1).N

   C(N,R)=N!/(R!.(N-R)!)
    OR C(N,R)=P(N,R)/R!
```

20-4 Program Using SUBROUTINE and FUNCTION

```
#EDIT

]INPUT
00001:         PROGRAM MAIN
00002:         X=4.
00003:         R=3.
00004:         CALL PC(X,Y,PXY,CXY)
00005:         WRITE(61,10)PXY,CXY
00006:      10 FORMAT(2F8.0)
00007:         CALL EXIT
00008:         END
```

Now our program to calculate the number of permutations and combinations of 4 objects taken 3 at a time with a SUB-ROUTINE and a FUNCTION

```
00009: [ESC] or [ALT MODE]
]SARL,4,,/X,Y/,/X,R/
00004:       CALL PC(X,R,PXY,CXY)
```

We stop at end of main program to correct error.

```
]APPEND
00009:       SUBROUTINE PC(X,Y,PXY,CXY)
00010:       PXY=FACT(X)/FACT(X-Y)
00011:       CXY=PXY/FACT(Y)
00012:       RETURN
00013:       END
00014:       FUNCTION FACT(X)
00015:       N=X
00016:       FACT=1.
00017:       DO 1 I=2,N
00018:       FACT=FACT*FLOAT(I)
00019:     1 CONTINUE
00020:       RETURN
00021:       END
00022:
```

Append the SUBROUTINES on current content of scratch pad, i.e. the main program above.

```
]OUT,PERMCOMB
```

Program to disk

```
]FORTRAN,I=PERMCOMB,R
```

Compile and execute

```
 NO ERRORS FOR MAIN

 NO ERRORS FOR PC

 NO ERRORS FOR FACT
RUN
```

No compilation errors

```
      24        4
```

P(4,3)=24 and C(4,3)=4

```
 END OF FORTRAN EXECUTION

 #LOGOFF
 TIME 9.161 SECONDS MFBLKS 4 COST $1.70
```

#LOGOFF to terminate run and update account

APPENDIX A

COMMON ERROR MESSAGES

IN FORTRAN PROGRAMMING ON THE CDC 3300

AT

OREGON STATE UNIVERSITY

FORTRAN ERROR MESSAGES

FORTRAN programs go through three phases: compiling, loading and execution. Each phase can have errors associated with it. Normally in each phase some information is given about the errors involved. The FORTRAN compiler will list all the diagnostic errors in the syntax at the end of the program listing. Most such diagnostic errors are fatal and prohibit any further action on the program. Certain of these diagnostics will allow execution to continue. Most of the time, however, the existence of the error means something is wrong with the program or at least there is some redundancy: either extra statements which can never be reached or statement numbers not referenced by other statements in the program. The compiler messages are usually somewhat cryptic, but they do attempt to point to the statement in which an error has been detected. When an error is detected, the location of the error is given relative to the statement number of the closest previous statement with a number. Unfortunately, the error message pointing to a particular statement may actually be caused by an error in or omission of a previous statement; thus when the programmer examines the statement indicated by the error message, there may be no visible error. Be sure to understand all the error messages and make certain all the errors are corrected before you make any further attempt to run the program again.

We'll now look at some examples for finding statements relative to a given statement number.

```
ERROR AT     3 STATEMENTS PAST     13
ERROR AT STATEMENT    27
ERROR AT     5 STATEMENTS PAST      0
```

When counting statements past a given statement number, comment cards (C in column 1) are not counted, nor are continuation cards (non-zero, non-blank in column 6), which are part of the preceding statement and not new statements. In the above set of examples there are possible errors at three statements past the line with statement number 13, at the line with statement number 27, and at five statements past 0 where the 0 indicates that the error occurred before the first statement with a number. Statement number 0 is assumed to immediately precede the first statement in the program or subprogram being compiled.

The following program has been designed to generate many of the most common error messages. Below the listing of the program are the actual FORTRAN compiler diagnostics. The OS-3 EDITOR line number is given to the left of each line of the program as it would appear on being output on the line printer. We have added numbers to the error messages to make it easier to refer to them in the analysis of the errors which follows the list of errors as generated by the compiler.

```
    0+001       DIMENSION A(10)
    0+002       Y=Z+X
    0+003       DIMENSION T(25)
    0+004    13 Q=C+(X*(B+X*A)
    0+005    13 CONTINUE
    0+006       Y-B+C=X
    0+007       STUVWXYZOW=I+JK
    0+008       FORMXT
    0+009    12 FORMAT(F8,2,I3,)
    0+010       FORMAT(I16)
    0+011       DO 1 X=1,25
    0+012       Y=A(X)
    0+013     1 CONTINUE
    0+014       DO 10 I=X,Y,Z
    0+015       DO 14 J=1,2
    0+016    10 CONTINUE
    0+017    14 CONTINUE
    0+018       DO 11 I=1,N
    0+019       I=I+1
    0+020    11 N=2*N
    0+021       WRITE(61,12)(B(I),I=1,N)
    0+022       WRITE(61,16)1,SQRT(X)
    0+023       GO TO 27
    0+024       A=R+S
    0+025    37 IF(J=K) GO TO 10
    0+026       A=R+-S
    0+027       WRITE(61,12)T(J)
    0+028       STOP
    0+029         END
```

```
ERRORS FOR JOB
 UNDEFINED SIMPLE VARIABLES                                        (1)
   S            K            JK            Z            R
```

```
LABELS NOT REFERENCED                                                    (2)
  37         13

UNDEFINED STATEMENT LABELS                                               (3)
  27         16

ERROR AT      1 STATEMENTS PAST     0                                    (4)
 PROGRAM IDENTIFICATION NOT PRESENT

ERROR AT      3 STATEMENTS PAST     0                                    (5)
 DECLARATIVE STATEMENTS MAY NOT APPEAR AFTER EXECUTABLE STATEMENTS

ERROR AT STATEMENT    13                                                 (6)
 PARENTHESES DO NOT MATCH

ERROR AT      1 STATEMENTS PAST    13                                    (7)
 LABEL ON THIS STATEMENT HAS BEEN USED PREVIOUSLY

ERROR AT      2 STATEMENTS PAST    13                                    (8)
 LEFT SIDE OF REPLACEMENT STATEMENT NOT IN CORRECT FORMAT

ERROR AT      3 STATEMENTS PAST    13                                    (9)
 IDENTIFIER HAS MORE THAN EIGHT CHARACTERS

ERROR AT      4 STATEMENTS PAST    13                                   (10)
 CANNOT IDENTIFY STATEMENT TYPE

ERROR AT STATEMENT    12                                                (11)
 SYNTAX ERROR IN FORMAT SPECIFICATION

ERROR AT      1 STATEMENTS PAST    12                                   (12)
 FORMAT STATEMENT NOT LABELED

ERROR AT      2 STATEMENTS PAST    12                                   (13)
 RUNNING INDEX OF A DO NOT A SIMPLE INTEGER VARIABLE

ERROR AT      3 STATEMENTS PAST    12                                   (14)
 SUBSCRIPT IS NOT A SIMPLE INTEGER VARIABLE

ERROR AT      1 STATEMENTS PAST     1                                   (15)
 DO LOOP QUANTIFIER IS NOT A SIMPLE INTEGER VARIABLE OR CONSTANT

ERROR AT      2 STATEMENTS PAST    14                                   (16)
 THE RUNNING INDEX IN A DO MAY BE CHANGED WITHIN THE LOOP

ERROR AT STATEMENT    11                                                (17)
 THE UPPER LIMIT VARIABLE (M2) OF THE DO MAY BE CHANGED WITHIN THE LOOP

ERROR AT      1 STATEMENTS PAST    11                                   (18)
 FUNCTION OR UNDIMENSIONED ARRAY IS IN I/O LIST

ERROR AT      2 STATEMENTS PAST    11                                   (19)
 WRONG FORMAT OF I/O DATA LIST OR ILLEGAL ENTRY IN I/O DATA LIST

ERROR AT      2 STATEMENTS PAST    37                                   (20)
 FUNCTION OR UNDIMENSIONED ARRAY IS IN I/O LIST

ERROR AT      4 STATEMENTS PAST    37                                   (21)
 END LINE SUPPLIED BY COMPILER

ERROR AT      3 STATEMENTS PAST    11                                   (22)
 STATEMENT LABEL REFERENCED IS UNDEFINED
```

```
ERROR AT      4 STATEMENTS PAST    11                                   (23)
 STATEMENT CAN NOT BE EXECUTED
ERROR AT STATEMENT    37                                                (24)
 =    <OPERAND>      APPEARS IN ARITHMETIC EXPRESSION
ERROR AT      1 STATEMENTS PAST    37                                   (25)
 +    -    APPEARS IN ARITHMETIC EXPRESSION
```

An analysis of each of these errors follows with the identifying error message given before each error is explained to make it easier to use this set of notes when finding errors in your own programs. This is only a sample of the most common errors. There will be other messages which you may have to decipher yourself.

UNDEFINED SIMPLE VARIABLES

(1) K, M, JK, Z and R are UNDEFINED SIMPLE VARIABLES because they appear in the program without having been input or assigned a value through an assignment statement. They would be undefined and thus have zero values at execution time. This diagnostic is non-fatal.

LABELS NOT REFERENCED

(2) 37 and 13 are redundant or incorrect statement numbers. They are not referred to by any other statements. Either these numbers are incorrect or the numbers in the statements which are supposed to reference these statement numbers are incorrect. A third possibility is that these numbers are not needed at all. This diagnostic is non-fatal.

UNDEFINED STATEMENT LABELS

(3) 27 and 16 are UNDEFINED STATEMENT LABELS since no such statement numbers exist in the program but at least one statement references each of these numbers. It may be that the statement number 27 is incorrect here, that the entire statement is missing or that the statement is present but has the wrong statement number or none at all. Combining the information of this message with that from the last message suggests that perhaps 27 should be changed to 37 or that the 37 which does exist should be changed to 27.

PROGRAM IDENTIFICATION NOT PRESENT

(4) PROGRAM name must be the first card of every program on CDC computers. We need to insert such a statement here. For example: PROGRAM GARBAGE. This statement must be inserted before the first DIMENSION.

DECLARATIVE STATEMENTS MAY NOT APPEAR AFTER EXECUTABLE STATEMENTS

(5) Declarative statements are non-executable statements which are used by the compiler to define variables as arrays or to change the implicit nature of certain variables. DIMENSION, REAL, INTEGER, COMPLEX, DOUBLE PRECISION and COMMON are the most commonly used declaratives. All such statements must be placed immediately after the PROGRAM name and before any executable statements.

PARENTHESES DO NOT MATCH

(6) There must be the same number of left and right parentheses in any given arithmetic expression. Here parentheses must be balanced by adding a right paren after the single right paren already present or by deleting the first left paren.

LABEL ON THIS STATEMENT HAS BEEN USED PREVIOUSLY

(7) Since statement 13 is already present, this statement cannot be labelled 13. This is one statement past the only valid 13.

LEFT SIDE OF REPLACEMENT STATEMENT NOT IN CORRECT FORMAT

(8) The form of an assignment statement must be VARIABLE = EXPRESSION. Note that this is two past the only valid statement 13.

IDENTIFIER HAS MORE THAN EIGHT CHARACTERS

(9) A maximum of eight alphanumeric characters is allowed as a variable name.

CANNOT IDENTIFY STATEMENT TYPE

(10) FORMXT is not a valid statement and is unrecognizable to the compiler.

SYNTAX ERROR IN FORMAT SPECIFICATION

(11) This message can be caused by any error within a FORMAT, so it is necessary to analyze exactly what the FORMAT is supposed to do. A normal correction here might be 12 FORMAT (F8.2,I3) but (F8.2,I3,F7.1) could have been what was wanted. Since the Hollerith specification in the nHtext form requires an exact count of the number of characters in text, including all blanks, this error often occurs because of a miscount of n.

FORMAT STATEMENT NOT LABELED

(12) The statement immediately following 12 is a FORMAT with no statement number. It therefore cannot be referenced by a READ or WRITE.

RUNNING INDEX OF A DO NOT A SIMPLE INTEGER VARIABLE

(13) The index of a DO must be an integer variable. This statement would be made valid by inserting INTEGER X as a declarative statement after the PROGRAM name statement.

SUBSCRIPT IS NOT A SIMPLE INTEGER VARIABLE

(14) X is used as a subscript but is real. Subscripts must be integer. Again the INTEGER X statement would correct this error, as long as no attempt is made to treat X as a real in some other calculations.

DO LOOP QUANTIFIER IS NOT A SIMPLE INTEGER VARIABLE OR CONSTANT

(15) The index parameters (or quantifiers) must be integer variables or integer constants only.

THE RUNNING INDEX IN A DO MAY BE CHANGED WITHIN THE LOOP

(16) The index of the loop is I and the second statement in the loop attempts to redefine I. This cannot be allowed.

THE UPPER LIMIT VARIABLE (M2) OF THE DO MAY BE CHANGED WITHIN THE LOOP

(17) The limiting parameter N is changed within the loop. This cannot be allowed. In this loop the violation of these rules would generate an infinite loop if N were originally defined greater than 1.

FUNCTION OR UNDIMENSIONED ARRAY IS IN I/O LIST

(18) According to the form of the output statement, B is an array. But B does not appear in a DIMENSION statement and thus it is not an array. It is thus an UNDIMENSIONED ARRAY IN AN I/O LIST. The lack of a DIMENSION for B causes the error in a statement which is by itself perfectly valid.

WRONG FORMAT OF I/O DATA LIST OR ILLEGAL ENTRY IN I/O DATA LIST

(19) Only variables can be included in an I/O list. 1 is a constant and is thus an illegal entry.

FUNCTION OR UNDIMENSIONED ARRAY IS IN I/O LIST

(20) Only variables can appear in an I/O list. All arrays must have been DIMENSIONed and no FUNCTION names, such as SQRT(X), are allowed.

END LINE SUPPLIED BY COMPILER

(21) The last line in every program and in every subprogram must be END. The OS-3 compiler will put an END after the last card before the $\begin{smallmatrix}77\\88\end{smallmatrix}$ but this may indicate some cards are missing or that the end-of-file card is in the wrong place.

STATEMENT LABEL REFERENCED IS UNDEFINED

(22) A statement number has been used in a transfer of control or a DO and no such number currently exists in the program. In this case there is no statement number 27 which is referred to by the GO TO 27 (line 0+023). Note that this is the second error message about statement number 27.

STATEMENT CAN NOT BE EXECUTED

(23) This message is given whenever a statement follows a transfer of control (other than a logical IF) but the statement has no statement number. Without a number on it there is no way to get to such a statement during execution. Although non-fatal, this error usually means that one or more statements will not be executed even though they are part of the program.

= <OPERAND> APPEARS IN ARITHMETIC EXPRESSION

(24) An = has appeared in the logical expression part of a logical IF. Only .EQ. is valid in logical expressions. Note, however, that the 's' part of IF(logical expression)s may have an = if it is an assignment statement.

+ - APPEARS IN ARITHMETIC EXPRESSION

(25) This error is due to a violation of the arithmetic rule which states that two operators may not be consecutive. Parentheses must be used to separate the -S from the R: R+(-S).

Other errors like these last two are possible when the precedence rules, the parentheses rules, and the other rules on the formation of arithmetic and logical expressions are violated.

INPUT RECORD ERROR

Since there is no easy way to tell what column you are working in when you are using a teletype, it may happen that one of your statements goes beyond column 80. If this does happen in a FORTRAN program entered from the EDITOR, then when an attempt is made to compile the FORTRAN program, the message INPUT RECORD ERROR is given and no further compilation takes place. This cannot happen when working from cards.

LOADING DELETED

If fatal compiler diagnostics are generated and no object program is created, and the R parameter has been used, then the message LOADING DELETED is given and the job is terminated.

Once all compilation errors have been corrected, the machine language object program, which is normally placed on logical unit 56, can be loaded into memory with all appropriate subprograms needed being loaded from the system library. In the LOADER phase errors are usually unlikely unless the programming is relatively sophisticated. Some errors which might occur follow. All are fatal, causing the job to be terminated.

PROG DUPLICATE SYMBOL IN PROG

This is caused by having attempted to load two copies of PROG. This happens on-line when running a corrected version of a program right after a previous version without having rewound or released the LUN on which the previous object program resided. The use of the R parameter on the FORTRAN control card will eliminate this problem, but it is not always possible to use this parameter.

XQRT UNDEFINED SYMBOL IN ZAP

This is caused by having omitted to include a subprogram called XQRT with your main program ZAP. This could be a simple omission of the card deck for XQRT; it could be a mispunch of SQRT; or it could be that you had intended XQRT to be an array and did not DIMENSION it. In any case the LOADER looks for such a subprogram and it is missing from the object program and also from the system library.

0 TRANSFER SYMBOLS

After loading all of the object program and the library routines, no main program has been found. This occurs when using the R parameter on the FORTRAN control card when there are diagnostic errors in the main program but there are also some subprograms included by the programmer which have no diagnostics. An object program gets generated consisting of only those subprograms which have no fatal compiler diagnostics. An attempt to load this object program and the library causes difficulty since no main program exists.

SUBPROG MEMORY OVERFLOW

The total number of memory locations of all programs and subprograms loaded to this point exceeds 32768. This error usually occurs when large arrays are being used.

If neither compiler nor loader diagnostics have terminated the run, then the object program and the various subprograms it requires from the system library, as loaded by the LOADER, can be executed. We then have the possibility of execution errors which in turn can cause abnormal termination of the job. Such errors occur due to errors in the logic of the program when, for example, transfer is made to an illegal address or when a record is input which is illegal under a given FORMAT specification or when an attempt has been made to read beyond the end-of-file on a given file.

The following partial list of arithmetic errors indicates the types of checks that are kept on the various library functions. These errors are not fatal but usually end up giving one or more incorrect results.

```
ERROR IN SQRT     CALLED FROM 77676 NEGATIVE ARGUMENT
```

An attempt to take the square root of a negative number has been made. The error was detected in the SQRT library function which was referenced from location 77676 (octal) in memory.

```
ERROR IN EXP      CALLED FROM 77724 X NOT LT/GT 709.089/-709.78
```

An attempt has been made to take the exponential of a number which would exceed the largest real number allowed. The error was detected in EXP which was referenced from location 77724 in memory.

```
ERROR IN ALOG     CALLED FROM 77735 X LE 0
```

An attempt has been made to take the natural logarithm of a number less than or equal to 0 but such a logarithm is undefined. The error was detected in ALOG while execution occurred at 77735.

It is interesting to note that dividing by zero and exceeding the maximum or minimum limits on the size of an integer or real number (other than as above in the EXP example) do not lead to error messages under the present version of OS-3. Division by zero gives a result of 2 or 2., depending on the mode. Exceeding the limits of valid integers or reals leads to erroneous results which may be extremely difficult to detect. For example 8388607 + 8388607 = -1.

To eliminate these problems there are LIBRARY subprograms available to allow checks on the existence of these conditions. For more information refer to the FORTRAN reference manual sections on DIVIDE FAULT, EXPONENT FAULT or OVERFLOW FAULT.

If an execution error occurs during an I/O operation, a message is given and the job is terminated. For example

```
ERROR IN  BCD IN  CALLED FROM 76277 ILLEGAL CHARACTER LUN 23
  INPUT RECORD  1.00.
```

An input error has been detected in the BCD IN routine called by a READ in the source program and loaded at location 76277. An attempt was made to read a record from logical unit 23 but a character exists in one of the data fields on that record which is illegal under the FORMAT specification being used. This could be two decimal points in the same field, a comma or alphabetic character in a numeric field or simply the wrong FORMAT being used by the programmer. The record in which an error has been detected is listed after the message INPUT RECORD.

```
ERROR IN  BCD IN  CALLED FROM 75251 UNCHECKED EOF LUN 14
```

An attempt has been made to read beyond the end-of-file record on this logical unit. If the program has no end-of-file check, it may need one. If it already has one, it may be in the wrong place or more likely, the input FORMAT and the READ list do not match and the wrong number of data cards has been read.

```
ERROR IN  BCD OUT  CALLED FROM 76653 RECORD OVERFLOW LUN 61
```

An attempt has been made to output a record greater than 136 characters. This error is detected in the BCD OUT routine which is called by a WRITE at location 76653.

If any of these or certain other fatal execution diagnostics occur the job will be terminated. You will usually get the additional message ABNORMAL TERMINATION OF FORTRAN EXECUTION.

If the E parameter is added to the FORTRAN compile control card, then some additional useful information is given when an execution error occurs and in addition subscript limit error checking code is added to your object program during compilation so that the values of subscripts can be checked against the limits automatically established in the DIMENSION statement. The additional message might look like this:

```
WHICH WAS AFTER LABEL 00013 IN PROGNAME
```

This extra information relates the execution error back to the statement numbers used in the program PROGNAME, excluding FORMAT statements. Thus this error occurred at or after the statement with statement number 13 and before the next statement in the program with a statement number. FORMATs are excluded because they are non-executable statements which are not really executed in-line but are used by READ and WRITE statements to indicate how things should be input or output.

```
ERROR IN SQRT     CALLED FROM 77676 NEGATIVE ARGUMENT
WHICH WAS AFTER LABEL 00037 IN ZYZZLE
```

This illustrates the message given when the E parameter is used and an attempt has been made to take the square root of a negative number. The error occurs in program or subprogram ZYZZLE and can be traced to some statement at or after statement number 37 and before the next statement with a number. Since this message occurs during execution, it is possible that the attempt was made to take the square root on each pass through a loop of statements which include at least one SQRT reference. This means that the first occurrence of an error will be the one which causes the first occurrence of the message. This, of course, does not have to be on the first pass through the loop. Note that simply modifying the call to the SQRT function so that it takes the absolute value of the argument before taking the root does not really solve the problem. The question to be answered is what is actually causing the program to get a negative argument.

If the E parameter is not used, it is possible to have abnormal termination after a message like

```
FORMAT ERROR 3
```

The number indicates which type of FORMAT error has occurred. There is a list of these in the FORTRAN reference manual, but it is usually unnecessary to know which error has occurred since these messages almost always occur when part of an existing FORMAT statement is wiped out during execution by inadvertently assigning a value to a memory location in which part of a FORMAT resides. This is usually done when using an array with a subscript which takes on a zero or negative value by acident. The arrays are located immediately after the FORMATs in memory when the object program is loaded, so a zero or negative subscript on an array actually may reference an area in memory which contains part of a FORMAT statement.

When the E parameter is used, the actual value of the subscript is compared with 1 and with the maximum permissible value as established in the DIMENSION; if the subscript is outside this range, then the following diagnostic is given and the program is abnormally terminated.

```
ERROR IN ERRCHK   CALLED FROM 77731 SUBSCRIPT LIMIT ERROR
WHICH WAS AFTER LABEL 00006 IN PROGNAME
```

A search after statement 6 in PROGNAME should yield a zero or negative value for some subscript or possibly a value greater than the maximum allowed. The error was detected in the ERRCHK subprogram which is only included in the object program when the E parameter is used.

Execution errors give the subprogram in which the error was detected, the absolute address in octal at which the error was detected and a message as to the type of error. The octal address can be very useful if the programmer knows assembly language or can generate a FORTRAN compiler map, showing where all the variables and all the statement numbers used in the program are relative to the start of the program or subprogram being compiled, and a LOADER map, giving the absolute addresses for all subprograms and their alternate entry points. To generate these two maps use the following sequence of control cards.

--------------------	JOB card, TIME card, etc.
$^{7}_{8}$FORTRAN,L,M,X	Call FORTRAN compiler; generate list
program deck	on 61, compiler map on 61, object program on unit 56
77 88	End-of-file for compiler
$^{7}_{8}$LOAD,56	LOAD needed when X option used
MAP	Generate LOADER map on unit 61
$^{7}_{8}$LOGOFF	

ILLEGAL INSTRUCTION READ LUN 1 AT 076554
FILE AT END OF DATA

This is a system error message which usually means serious problems. In this instance the FILE AT END OF DATA message indicates that an attempt was made to read beyond the end-of-file and there was an end-of-file check in the program. The check was either in the wrong place or the program was allowed to continue reading data records after the EOF was encountered.

ILLEGAL INSTRUCTION HALT 00004 AT 077751

This is also a system error message which usually means even worse trouble. It will occur, for example, when a program references a subprogram but sends the wrong number of arguments to the subprogram.

There are a number of other system errors which may occur from time to time.

TIME CUT

This message indicates that the amount of time you established in your TIME=number control card has been exceeded or that the job number is completely out of time. The simple solution of increasing the time and rerunning the job may be dangerous. Find out why the program used the time it did before resubmitting the job.

INSUFFICIENT FILE SPACE

This message is given when all the currently available scratch file space on the job number has been used. When the job is initiated, the system establishes a scratch file limit of 100 blocks. The job will terminate with this message if that limit is reached or if the lower limit normally established on student jobs is reached. If a lower limit is desired or the job has a higher limit which you do not want to approach, then use the MFBLKS=number control card to change the limit up to the maximum available on the job number.

INSUFFICIENT SAVE FILE SPACE

This message is given when all the available magnetic disk save file space under a particular job number has been used. The limit is established at the Computer Center when the job number is created and can only be changed with special permission. To make use of the save file space when this message is given, some current save files must be destroyed with the DESTROY,name control card.

After some of these system error messages and in certain other instances the system generates a report on the status of the special purpose registers in the control unit of the computer. This information is of use to a programmer familiar with assembly language and the uses to which the various registers are put. To most programmers, then, the following is useless information. The status report looks something like this:

```
P        077747
LJA      077757
A      00000001
Q      46332262
B1        77757
B2        00010
B3        77776
```

```
EU  20035600
EL  00000000
other information added usually at this point
```

FORTRAN programmers do not need to concern themselves with these status reports unless/until they become familiar with the assembly language for the 3300.

If, through hard work and a little good luck, you manage to get through all the phases without running into a fatal error, then the computer will print the message

```
END OF FORTRAN EXECUTION
```

and terminate the job (or actually, look for the next control card should the job contain more than one major task). You must still check your answers to satisfy yourself that there are really no errors in the logic. Such undetectable errors as an incorrectly matching FORMAT or an incorrectly coded formula or a logical comparison which transfers in the wrong direction may still exist but not violate any of the rules of the three phases. These may be hard to track down, but at least the lack of any error messages means everything else is valid.

Obviously there are many other errors which can occur which cannot be listed here; hopefully this set of notes will give you some insight into how and where to look for your errors and perhaps even help you decide what to do about them.

APPENDIX B

EXPANDED CONTENTS OF INDIVIDUAL VIDEOTAPES

CONTENTS OF VIDEOTAPE LECTURES IN FORTRAN

1. Introduction to Computers -- (a) general concepts of a computer system; (b) definitions of hardware and software; (c) general idea of computer programming; (d) specific discussions on FORTRAN alphabet and rules for forming instructions.

2. Bit Structure, Part I -- (a) introduction to binary numbers and analogs with the decimal system; (b) specific form of information storage in CDC 3300 computer-groups of 24 binary digits, each group processed as a single unit (a word); (c) constants: fixed numeric quantities stored in the computer's memory -- (1) integer (or fixed point) constants, (2) real (or floating point) constants, (3) how constants are stored in memory in two different forms, (4) how constants are programmed.

3. Bit Structure, Part II -- (a) variables - names assigned to locations in computer's memory at which constants are or can be stored; (b) how to form variable names; (c) arithmetic operations in FORTRAN; (d) algebraic and computer rules for formulating arithmetic expressions; (e) integer and real arithmetics and their differences; (f) library functions.

4. Arithmetic Assignment Statements; Input/Output -- (a) assigning values to variables through use of arithmetic expressions; (b) sample program - (1) executable statements, those which cause action, (2) non-executable statements, those which supply information; (c) data records; (d) input of data from records by READ statements; (d) logical unit numbers; (e) output of data stored in computer's memory by WRITE statements.

5. Input/Output - FORMAT -- (a) FORMAT statements - (1) non-executable statements supplying information as to number of pieces of data on data record, type of each, number of decimal places, if any, and location on record, (2) used in conjunction with READ and WRITE to input information to and to output information from the computer's memory; (b) discussion of specific FORMAT specifications - (1) integer, (2) real decimal, (3) exponential, (4) spacing specifications.

subscripts; (f) reasons for use of arrays; (g) example of use of arrays in averaging problem.

12. Other Types of Constants, Variables and FORMAT Specifications -- (a) REAL and INTEGER explicit type declaration statements; (b) DOUBLE PRECISION and COMPLEX type statements and associated arithmetic; (c) library functions for converting from integer to real and vice versa.

13. DO Loops -- (a) Hollerith constants, which contain letters instead of numbers; (b) FORMAT specification for Hollerith constants; (c) return to summation - several ways to sum the integers from 1 to N; (d) DO statements - iterative statements which cause a series of other steps to be repeated in a loop.

14. DO Loops Cont. -- (a) the strict limitations on the form of the DO; (b) action which occurs in a typical loop; (c) rules for using DO's to iterate a series of steps.

15. Nested DO Loops; Input/Output of Arrays -- (a) loops inside loops; (b) example of doubly nested DO loops for double iteration; (c) examples of use of DO; (e) use of DO loops with arrays; (f) FORMAT rules if more or fewer elements in the argument list than numeric and alphanumeric FORMAT specifications.

16. Input/Output of Arrays Cont.; Program Examples -- (a) I/O of arrays with and without subscripts; (b) use of DO loops for I/O of arrays; (c) implied DO's; (d) examples of programs with arrays - (1) test scoring and averaging (2) use of other types of constants, variables and FORMAT specifications with arrays and DO's.

17. Two Dimensional Arrays; Subprograms -- (a) reasons for use of 2D arrays - (1) matrices, (2) tables; (b) examples of operations on a table of numbers; (c) definition of subprograms; (d) reasons for use of subprograms; (e) FUNCTION subprograms to define a single-valued function of several variables.

18. Subprograms Cont.; Other Specification Statements -- (a) formation of FUNCTION subprograms - (1) starting, (2) stopping, and (3) function definition requirements; (b) example of FUNCTION subprogram with accompanying main program; (c) SUBROUTINE subprograms -

(1) definition and (2) requirements; (d) referencing a SUBROUTINE from a main program - the CALL statement; (e) example of SUBROUTINE subprogram with accompanying main program; (f) other types of non-executable specification statements - (1) COMMON, (2) EQUIVALENCE, (3) COMMON/DATA/, (4) DATA statements and their uses.

19. Entering and Editing FORTRAN from Teletype -- (a) the OS-3 EDITOR language for entering and editing both programs and data directly from an on-line keyboard; (b) use of teletypes - logging on; (c) some useful features of OS-3; (d) input and running of a FORTRAN program from teletype; (e) correcting errors with the EDITOR - (1) compilation errors and (2) execution errors due to incorrect logic.

20. Entering and Editing FORTRAN, Part II -- (a) some useful EDITOR instructions - their general form and purpose; (b) direct on-line, real-time communication with computer; (c) OS-3 COPY command; (d) definitions of permutations and combinations through use of COPY and stored data files; (e) program example - permutations and combinations; (f) LOGOFF.

APPENDIX C

TAPE NUMBER, TITLES AND TIMES

1. Introduction to Computers (30:00)
2. Bit Structure, Part I (29:35)
3. Bit Structure, Part II (31:25)
4. Assignment Statements; Input/Output (31:10)
5. Input/Output - FORMAT (30:30)
6. Starting, Stopping, Transfer of Control (30:00)
7. Arithmetic IF; OS-3 Control Statements (32:00)
8. Program Ex. 1; Summation and Counting (30:00)
9. Summation and Counting; Logic Concepts (32:00)
10. Logical IF; Program Ex. 2 (32:00)
11. Arrays and Subscripts (26:52)
12. Other Types of Constants, Variables & FORMAT Specifications (27:17)
13. DO Loops (32:00)
14. DO Loops Continued (31:30)
15. Nested DO Loops; Input/Output of Arrays (31:00)
16. Input/Output of Arrays Continued; Program Examples (33:00)
17. Two Dimensional Arrays; Subprograms (31:05)
18. Subprograms Continued; Other Specification Statements (33:10)
19. Entering and Editing FORTRAN from Teletype (30:00)
20. Entering and Editing FORTRAN from Teletype, Part II (30:30)

APPENDIX D

CDC 3300 Character Codes

CDC 3300 Character Codes

BCD CODE	CARD CODE	KEY PNCH	LINE PRNT	TELE TYPE	ASCII CODE	BCD CODE	CARD CODE	KEY PNCH	LINE PRNT	TELE TYPE	ASCII CODE
00	0	0	0	0	260	40	11	-	-	-	255
01	1	1	1	1	261	41	11,1	J	J	J	312
02	2	2	2	2	262	42	11,2	K	K	K	313
03	3	3	3	3	263	43	11,3	L	L	L	314
04	4	4	4	4	264	44	11,4	M	M	M	315
05	5	5	5	5	265	45	11,5	N	N	N	316
06	6	6	6	6	266	46	11,6	O	O	O	317
07	7	7	7	7	267	47	11,7	P	P	P	320
10	8	8	8	8	270	50	11,8	Q	Q	Q	321
11	9	9	9	9	271	51	11,9	R	R	R	322
12	2,8		:	:	272	52	11,0			!	241
13	3,8	=	=	=	275	53	11,3,8	$	$	$	244
14	4,8	'	≠	'	247	54	11,4,8	*	*	*	252
15	5,8		<	&	246	55	11,5,8		↑	↑	336
16	6,8		$\overline{\%}$	%	245	56	11,6,8		↓	@	300
17	7,8		[	[	333	57	11,7,8		>	>	276
20	12	+	+	+	253	60	Blnk	Blnk	Blnk	Spce	240
21	12,1	A	A	A	301	61	0,1	/	/	/	257
22	12,2	B	B	B	302	62	0,2	S	S	S	323
23	12,3	C	C	C	303	63	0,3	T	T	T	324
24	12,4	D	D	D	304	64	0,4	U	U	U	325
25	12,5	E	E	E	305	65	0,5	V	V	V	326
26	12,6	F	F	F	306	66	0,6	W	W	W	327
27	12,7	G	G	G	307	67	0,7	X	X	X	330
30	12,8	H	H	H	310	70	0,8	Y	Y	Y	331
31	12,9	I	I	I	311	71	0,9	Z	Z	Z	332
32	12,0		<	<	274	72	0,2,8		]	]	335
33	12,3,8	.	.	.	256	73	0,3,8	,	,	,	254
34	12,4,8	)	)	)	251	74	0,4,8	(	(	(	250
35	12,5,8		≥	#	243	75	0,5,8		↱	\	334
36	12,6,8		¬	"	242	76	0,6,8		≡	←	337
37	12,7,8		;	;	273	77	0,7,8		∧	?	277

Other teletype characters and their ASCII codes:

Bell	207	Horizontal Tab	211
Line Feed	212	Vertical Tab	213
Return	215	Form Feed	214
Rubout	377	All Mode & Escape	233,374 375,376

Display unit uses BCD codes, with line printer characters, except:

36	-	(carriage return
37	▲	(send)
75	■	(parity error)
76	'	(print)

APPENDIX E

Line Printer Carriage Control

The character in column 1 of information sent to a line printer specifies control of paper movement during printing. This character is never printed. Listed below are the more commonly used carriage control symbols and the action of the paper carriage on the current CDC 3300 line printer when these characters are used in column 1.

Character	Action
ƀ (blank)	Single space before printing.
0 (zero)	Double space before printing.
-	Triple space before printing.
+	No space before printing, overprint.
1	Skip to a new page before printing, page eject.
A	Single space, print, skip to new page.
Q	Clear automatic page eject.
R	Reset automatic page eject.
S	Reset to six lines per inch vertical spacing.
T	Select eight lines per inch vertical spacing.

B,X,Y,Z,2,3,4,8,9 are used as special purpose carriage control symbols, but their function will not be explained here. The remaining characters will cause single spacing before printing.

Q,R,S,T when used as carriage control symbols, inhibit printing of anything else within the FORMAT. Automatic page eject and 6 lines per inch are standard default options, so R and S are used only when Q and T have been used previously in the same program and the program needs to return to the standard options again.

INDEX