Price: $5.50

TYMSHARE MANUALS
TYMCOM-IX

SUPER FORTRAN

A SUPERSET OF H LEVEL FORTRAN IV

April 1970

TYMSHARE, INC.
10340 BUBB ROAD
CUPERTINO, CALIFORNIA 95014

970, TYMSHARE, INC., Litho in U.S.A.

CONTENTS

TYMSHARE MANUALS
SYMBOL CONVENTIONS

The symbols used in this manual to indicate Carriage Return, Line Feed, and ALT MODE/ESCAPE are as follows:

Carriage Return: ↵

Line Feed: ↴

ALT MODE/ESCAPE: ⊕

NOTE: This symbol will be printed as many times as it is required to hit this key.

Action At The Terminal

To indicate clearly what is typed by the computer and what is typed by the user, the following color code convention is used.

Computer: Black **User: Red**

NOTE ON SPACING IN EXAMPLES

Because this manual is set in type with characters of varying width, the spacing in some of the examples may not appear exactly as on the terminal, where all characters are the same width. If the spacing in an example appears misleading, this general rule will be helpful:

The number of blanks or spaces printed can usually be determined by counting the print positions (characters) in the line or lines above.

Also, in some sections where the spacing is critical, blank spaces are indicated by a ■.

SECTION 1

INTRODUCTION

Tymshare SUPER FORTRAN is a comprehensive algebraic compiler which permits the programmer to concentrate on the problem to be solved rather than on formal syntax requirements.

SUPER FORTRAN is composed of 1) SUPER FORTRAN source language statements with which the actual programs are written, and which are compatible with IBM 360 H Level FORTRAN IV compilers; and 2) the Conversational Compiler System (CCS) commands which control the operating system under which SUPER FORTRAN language statements are used on the Tymshare system.

The Tymshare SUPER FORTRAN language is IBM 360 H Level compatible; in fact, it is a superset of standard FORTRAN IV. In addition, the Conversational Commands provide an easy, efficient tool for program development, testing, and debugging.

Each SUPER FORTRAN language statement is compiled and analyzed for errors as it is entered. If it is syntactically incorrect, a diagnostic is given immediately. The statement may then be corrected. After modification, the program can be listed on the terminal, saved on a disk file, and/or executed. The user can specify a single statement or a range of statements to be executed as well as execute the entire program. Program execution can be interrupted at any time, and direct statements can be entered for immediate execution. Program execution may be resumed at the point of interruption.

Source programs in SUPER FORTRAN can be entered directly from the terminal, from a paper tape, or from a file. The user with a source program punched on cards can request that the contents of the cards be put on a disk file at the computer center. He may then load the program from the file.

Data may be read from the terminal or from a file. Data on paper tape, magnetic tape, or cards may be written on a disk file and then used. Similarly, results can be printed on the terminal or saved on a file. At the user's request, large amounts of output can be printed on the high speed printer at the Tymshare computer center.

In addition to being conversational and H level compatible, Tymshare SUPER FORTRAN also includes the following features:

- Labelled COMMON, EQUIVALENCE, programmable error and end-of-file conditions, the EXTERNAL statement, BLOCK DATA subprograms, and dynamic formatting.
- Random access files.
- Fast loading binary program files.
- Program linking, preserving COMMON.
- Extensive string processing features.
- Programmable interrupts.
- CCS on-line editing and debugging capabilities.

SUPER FORTRAN can be used when any of the following characteristics are desired:

- When the user wishes to do complex arithmetic.
- When the user wishes to have double precision.
- When the user wants to use an existing FORTRAN IV program from other computers; or to debug a FORTRAN IV program to be run on other computers by taking advantage of the fact that FORTRAN IV is widely used.
- When the user wishes to use local names in subroutines.
- When the user wishes to use long names for greater readability.

ARRANGEMENT OF THE MANUAL

This manual can be used both as a tutorial guide and as a reference manual for Tymshare SUPER FORTRAN. Section 2—An Introduction to SUPER FORTRAN Language Programming—provides sufficient instruction for a beginning programmer to write complete programs in SUPER FORTRAN and to run them successfully on the Tymshare system.

The manual is organized so that an experienced programmer can skip Section 2 and proceed directly

to the other sections in which SUPER FORTRAN is described in detail. Sections 3 through 9 describe SUPER FORTRAN language syntax and statement elements. Section 3 contains a discussion of the fundamental SUPER FORTRAN statement elements; Section 4, replacement and control statements.

The ability to manipulate text with string variables and functions is an outstanding attribute of this language. Section 5 describes these string manipulation features.

Input and output statements are described in Section 6; declaration statements in Section 7. Subroutines and programmer defined functions are treated in Section 8. SUPER FORTRAN execution statements are discussed in Section 9.

Section 10 describes the CCS commands; that is, how programs are entered, stored, and executed on the Tymshare system. The section also includes the CCS commands that control running programs and that are used for debugging. Section 11 lists some sample programs written in SUPER FORTRAN and executed on the Tymshare system.

Appendix A contains instructions for planning efficient use of computer storage using memory allocation estimation. Appendix B is a table of the internal representation of ASCII codes. Appendices C, D, and E contain, respectively, summaries of EXECUTIVE commands, SUPER FORTRAN commands, and CCS commands. Appendix F describes some user aids such as abbreviations.

PROCEDURES FOR ENTERING AND LEAVING THE SYSTEM

The following is a summary of the log in and log out procedures for the Tymshare system. A complete description of the EXECUTIVE commands is found in the Tymshare *EXECUTIVE Manual, Reference Series*.

To gain access to the Tymshare time sharing system, you must first log in. As soon as the connection to the Tymshare computer is made, and the terminal identifying character is typed (*see Appendix C*), the system will type:

PLEASE LOG IN: ↵

Type a Carriage Return. The system replies with:

ACCOUNT: A3 ↵

Type your account number (A3 in this case) followed by a Carriage Return. The system then types:

PASSWORD: ↵

Type your password followed by a Carriage Return. The letters in the password do not print. The system next types:

USER NAME: JONES ↵

The user name JONES is followed by a Carriage Return. The system next asks for a project code.

PROJ CODE: K-123-K ↵

K-123-K is a project code. *NOTE: A project code is optional. If no project code is wanted, simply type a Carriage Return in response to the system's request.*

After you have entered the requested information correctly, the system will type the date and time. For example,

TYMSHARE 4/8 11:20

You are now in the EXECUTIVE and can call SUPER FORTRAN by typing SFORTRAN followed by a Carriage Return. SUPER FORTRAN will reply with a > when it is ready to accept a command.

To exit from the Tymshare system, you first must be in the EXECUTIVE, characterized by a dash in the left margin. To return to the EXECUTIVE from SUPER FORTRAN, type:

>QUIT ↵

The EXECUTIVE dash will appear in the left margin. Now type:

–LOGOUT ↵

followed by a Carriage Return. The system then will type:

CPU TIME: *n* **SECS.**

Number of computing seconds used.

TERMINAL TIME: 0:00:00

Number of minutes connected.

When the computer types

PLEASE LOG IN:

you may disconnect the line or let another user log in.

A SAMPLE SUPER FORTRAN PROGRAM

The following is a sample SUPER FORTRAN program entered and executed at the terminal.

```
PLEASE LOG IN: ↵
ACCOUNT:   A5 ↵
PASSWORD: ↵
USER NAME: FM ↵
PROJ CODE: KL-3-456 ↵

TYMSHARE  10/14  11:36

-SFORTRAN ↵

> 10 *THIS PROGRAM COMPUTES GAS MILEAGES. ↵
> 20  100 ACCEPT "INITIAL ODOMETER READING= ",ODOM1 ↵
> 30 ACCEPT "FINAL ODOMETER READING= ",ODOM2 ↵
> 40 ACCEPT "TOTAL GAS USED= ",GAS ↵
> 50 TOTAL=ODOM2-ODOM1 ↵
> 60 GASMILEAGE=TOTAL/GAS ↵
> 70 DISPLAY TOTAL," MILES",GASMILEAGE," MILES/GAL" ↵
> 80 GO TO 100 ↵
> 90 END ↵
> RUN ↵
INITIAL ODOMETER READING= 10147.6 ↵
FINAL ODOMETER READING= 10519.3 ↵
TOTAL GAS USED= 17.4 ↵
 371.7        MILES     21.362069  MILES/GAL
INITIAL ODOMETER READING= ⊕
```

The user typed an ALT MODE (or ESCAPE) to stop the program.

```
INTERRUPT
20  > QUIT ↵

-LOGOUT ↵

CPU TIME: 1 SECS.
TERMINAL TIME: 0:5:16

PLEASE LOG IN:
```

SECTION 2

INTRODUCTION TO SUPER FORTRAN PROGRAMMING

KEY STEPS IN PROGRAMMING

A computer program is a set of simple instructions written in a language the computer can understand which tells the computer how to solve a problem—how to accept data, how to process it, and how to return the results to the user.

The language used to write a program depends upon the problem to be solved and the computer to be used. The actual machine language of a computer is very tedious to learn and use. Therefore, a group of higher level computer languages has been developed. SUPER FORTRAN is one of these languages, and is essentially a combination of simple English and elementary algebra.

There are six key steps in writing any computer program:

1. Defining the problem.
2. Selecting a method for solution.
3. Analyzing the problem.
4. Writing the instructions.
5. Debugging and checking the program.
6. Documenting the program.

A brief discussion of each step follows.

DEFINING THE PROBLEM

Before you can write a program to solve a problem, you must know exactly what the problem is. A computer can only follow your instructions—it has no intuitive knowledge. You must first determine what answers are required, how these answers are to be given, and what accuracy is required in the answers. The results computed and reported by the computer are collectively referred to as output.

Next you must determine what information is given and in what form this original data is supplied. The original information supplied to the computer is called input.

Finally, study the problem to determine whether there are any special cases and define the alternatives in these cases.

SELECTING A METHOD FOR SOLUTION

There are usually several ways to solve a problem, and it will be necessary to select one which is best for your particular situation. You should first analyze the given data and the desired results to determine what computations must be made. You may prepare a brief step-by-step numerical solution to your problem using as many methods as you can. After some experience, it will be easier to see which method is best suited to computer solution.

ANALYZING THE PROBLEM

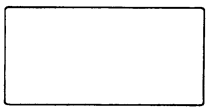
After the method of solution is selected, you should determine the steps required to solve the problem. A problem may be solved primarily by evaluating a series of formulas, or it may require more involved steps. Try to organize these steps into a logical sequence so the computer can perform the computations.

Often the clearest method of representing the logical sequence of a program is to picture it with a flowchart. A flowchart consists of a number of boxes connected by lines. Within each box is a brief statement of an operation to be performed. The interconnecting lines, with arrows attached, show the various paths the solution may take. If many decisions are to be made or many alternatives exist, a flowchart makes these alternate paths easier to follow.

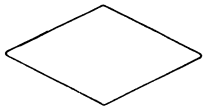
A flowchart may be simple, showing only the vaguest outline of the various alternatives, or very detailed. The greater the detail in the flowchart, the easier the actual programming will be.

Typically:

Rectangular boxes are used for input, output, and computations.



Diamond shaped boxes are used for decisions.



Circles are used as connectors when flowcharts become more complicated.

Figure 1 is an example of a simple flowchart for a program to find the sum of the reciprocals of N numbers.

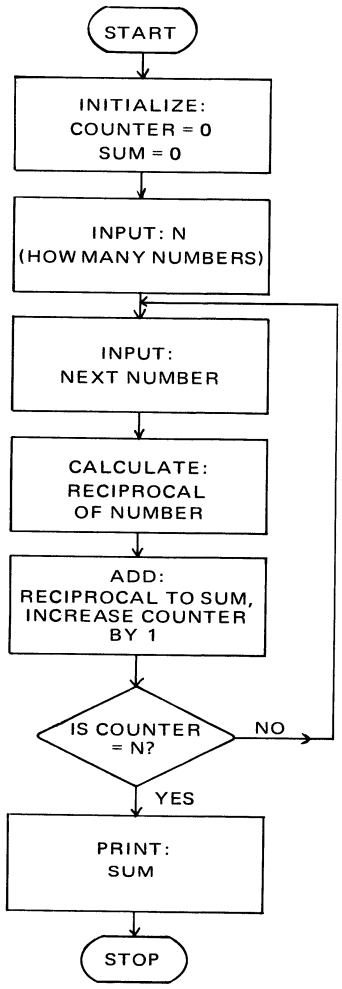


Figure 1 - Flowchart Example

It is usually worthwhile to prepare test cases which exercise all options; that is, follow all paths of the flowchart. A test case should contain input data for which correct answers are known.

WRITING THE INSTRUCTIONS

Writing the program instructions in a computer language is called coding. Since SUPER FORTRAN consists of a small number of statement types, each step in the problem solution usually corresponds to a single SUPER FORTRAN statement. In writing these statements, the correct order and proper syntax of the language must be used. You may write down all statements before beginning to type them into the computer from the terminal.

DEBUGGING

Debugging is testing and checking a program. As you enter the statements from the terminal, the interactive features of the Tymshare system can be a great help to you in finding and correcting errors in your program. Any syntax error will be detected and a message will be printed to indicate the nature of the error.

After SUPER FORTRAN syntax errors are corrected, you should try to run the program. If it runs, test it using data for which correct answers are known. The final checkout is, of course, continued satisfactory use of the program.

If the program does not run, SUPER FORTRAN will usually give you an error diagnostic telling you what is wrong. If your program runs, but the answers are incorrect, there are two options. One is to run the program part by part in sequential order. This will isolate the problem. The second is to work through the program as the computer would, using the simplest cases you can think of. Read each step in the program and execute it. Do only what you have told the computer to do, not what you know should be done. As you step through the program in this manner, you will probably find the errors.

DOCUMENTING THE PROGRAM

It is a good idea to include comments in the program to remind yourself of what the program does. After the program is debugged, document the program if you want to use it again.

INTRODUCTION TO SUPER FORTRAN LANGUAGE ELEMENTS

Now that we have introduced you to programming, we will proceed to a simple course in SUPER FORTRAN which will enable you to write and execute programs of your own on the Tymshare system. We suggest that you try to run some simple programs on the terminal to see how it works. The Tymshare conversational system is easy to use, and you will find that actual use of the computer is your most valuable learning experience and the fastest way to get acquainted with the system.

You recall that a computer program is a set of statements which tells the computer how to solve a problem. There are three basic types of SUPER FORTRAN statements: Input/Output (or I/O) statements; Replacement (or Assignment) statements; and Control statements.

Statements of each of these three types may contain the names of variables used in solving the problem. A SUPER FORTRAN variable name can be one or more alphabetic characters or numeric digits, but the first character must be alphabetic. It is a good programming practice to choose a variable name which has some meaningful relationship to the problem. This will make the program easier to read.

Some examples of legal SUPER FORTRAN variable names are:

A BALANCE CHARGE1 NUMBER X2ZY

INPUT/OUTPUT STATEMENTS

Input/output statements tell the computer how to get the data necessary to solve the problem and how to return the computed results to the user. Two fundamental input/output statements are ACCEPT and DISPLAY. The ACCEPT statement is an input statement to read data into the program from the terminal. The DISPLAY statement is an output statement which directs the computer to write on the terminal the current values of the variables listed in the statement.

ACCEPT A,B instructs the computer to read two numbers from the terminal. The first number is assigned to the variable A, the second to the variable B.

DISPLAY MONTH, CHARGE causes the computer to print on the terminal the current values of the variables MONTH and CHARGE.

REPLACEMENT STATEMENTS

A replacement statement is one which directs the computer to perform certain arithmetic operations on the variables in the program and to assign a new value to one of the variables.

Let us consider a simple example.

```
10 X=5.0
20 Y=X+2.0
30 X=X*3.0
```

Statement 10 assigns the value 5.0 to the variable X. Statement 20 adds 2.0 to the current value in X, 5.0, and assigns the value of the sum, 7.0, to Y. The value of X remains 5.0. Statement 30 multiplies the current value of X, 5.0, by 3.0. The product is then assigned to the variable X. The new value of X is 15.0.

The correct form of a replacement statement is one in which the left side is the name of a single variable. The equal sign means "is to be replaced by" rather than "is equal to". This is an important distinction. For example, X=X+1 is not a legitimate algebraic equation, but is a perfectly valid assignment statement. It directs the computer to add 1 to the current value of the variable X and to assign the value of the sum to X.

Replacement statements frequently involve arithmetic operations on the right side of the equal sign. For this reason they are sometimes referred to as arithmetic statements. The symbols for the arithmetic operations are + for addition, – for subtraction, * for multiplication, / for division, and ** or ↑ for exponentiation (raising a number to a power).

The computer can perform only one operation at a time. Therefore, since most expressions involve more than one arithmetic operation, some priority of computation must be established. All arithmetic expressions are scanned from left to right. If any exponentiation is encountered, it is executed first. Again the expression is scanned; multiplication and division are executed from left to right. The expression is scanned a third time, this time addition and subtraction being computed from left to right.

Parentheses may be used to alter the usual priority of computation. Any quantity within parentheses is evaluated before the scan for arithmetic operators begins. Evaluation of parentheses begins at the inner set of parentheses and proceeds to the outer set.

Within a given set of parentheses, arithmetic operations are performed according to the usual hierarchy.

Note the distinctions among the following expressions.

FORTRAN	Algebra
A-B/C↑2	$a - \frac{b}{c^2}$
(A-B)/C↑2	$\frac{a-b}{c^2}$
((A-B)/C)↑2	$\left(\frac{a-b}{c}\right)^2$

Example

Let us now write a simple program using only these two classes of statements:

Problem

Write a program to determine the monthly payment on a loan. The monthly payment, P, is given by the formula

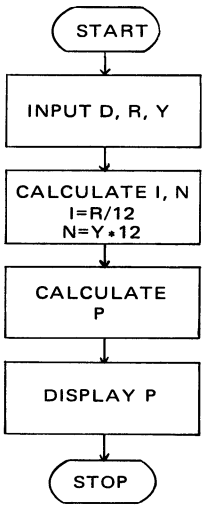
$$P = \frac{D \cdot I(I+1)^N}{(I+1)^N - 1}$$

where D is the original debt,
I is the monthly interest rate,
N is the number of months to pay off the loan.

The program should request values for the original debt, D, the annual interest rate, R, and the number of years to pay off the loan, Y, as input.

Flowchart

Here is a flowchart of the problem.



Program Coding

Here is the coding of the program.

```
REAL I
ACCEPT D,R,Y
I=R/12
N=Y*12
P=(D*I*(I+1)↑N)/((I+1)**N-1)
DISPLAY P
END
```

Note that ↑ and ** are equivalent representations of the exponential operator.

The last statement in every SUPER FORTRAN program must be END.

CONTROL STATEMENTS

The third major class of SUPER FORTRAN statements is the control statement. During the normal running of a program, statements are executed sequentially. That is, after one statement has been executed, the one immediately following it is executed. Control statements allow us to alter this normal sequence of execution. When such alteration becomes necessary, certain statements must be given labels by which they may be referred to in a control statement. A statement label (or statement number as it is sometimes called) can be any integer from one to five digits long.

Two basic control statements are GO TO and IF. When the GO TO statement is encountered, the program immediately executes the statement referred to in the GO TO command. For example, GO TO 100 directs the computer to execute next the statement labelled 100. After statement 100 is executed, sequential execution continues, beginning with the next statement after 100.

One form of the IF statement consists of a logical expression and an executable statement; for example,

IF (X.EQ.7) GO TO 200

This IF statement will transfer program execution to the statement labelled 200 if and only if the value of X is equal to 7. If X is not equal to 7, the statement immediately following the IF statement is executed.

There are six relational operators which may be used in a logical IF statement. They are: .EQ. (equal to); .NE. (not equal to); .LT. (less than); .LE. (less than or equal to); .GT. (greater than); .GE. (greater than or equal to). Note that the periods are an essential part of these operators.

PROGRAM ANALYSIS

Let us now analyze the program listed below, which contains examples of each of these statements. Every statement in this program has a line number (10, 20, 30, etc.). Each line number identifies a line, and line numbers keep the program statements in order. The statements are executed in the order in which they are numbered unless a transfer occurs as a result of a control statement. As we shall see later, line numbers are used to refer to the program statements when modifying and inserting statements. The choice of the line numbers is arbitrary; they may range from .001 to 999.999, assigned either explicitly by the programmer or implicitly by the system.

Within the program, there are two statements with statement labels in addition to line numbers. Lines 40 and 100 also have statement labels 1000 and 2000. Statement labels are part of the language and are used as reference points in the program. Statement labels can be any integer number between 1 and 99999, and need not be in numeric order.

This program is designed to compute monthly balances on a loan.

```
10          ACCEPT LOANNUMBER,↵
            PRINCIPAL, RATE, PAYMENT
20          DISPLAY "MONTH  BALANCE"
30          MONTH=1
40    1000  XINTEREST=PRINCIPAL↵
            *RATE/12[1]
50          PRINCIPAL=PRINCIPAL↵
            +XINTEREST-PAYMENT
60          IF (PRINCIPAL .LE. 0.)↵
            GO TO 2000
70          DISPLAY MONTH, PRINCIPAL
80          MONTH=MONTH+1
90          GO TO 1000
100   2000  STOP
110         END
```

Let us now analyze each statement.

```
10          ACCEPT LOANNUMBER,↵
            PRINCIPAL, RATE, PAYMENT
```

Line 10 is an ACCEPT statement which makes it possible to enter all the input data at one time. The ACCEPT statement is unique to Tymshare SUPER FORTRAN and allows data to be read in free form from the terminal. By free form we mean that the user simply types the data values separated by a comma or a Carriage Return. The data is not entered according to a specific format.

When the program is executed, the system will ring a bell and wait for the user to type the data values specified in the ACCEPT statement. In this case, the first value typed will be assigned to the variable LOANNUMBER, the second to the variable PRINCIPAL, etc. When all the values requested by a single ACCEPT statement have been typed, the user may type a final comma or press the Carriage Return.

```
20          DISPLAY "MONTH  BALANCE"
```

Line 20 allows us to put a heading over the columns of output data. Any message text inside a pair of double quote marks can be printed with a DISPLAY statement. When such a DISPLAY statement is encountered during program execution, the quote marks are not printed.

```
30          MONTH=1
```

Line 30 is a replacement statement which initializes the variable MONTH to the value 1.

```
40    1000  XINTEREST=PRINCIPAL*RATE/12
```

This statement is a replacement statement which computes the monthly interest. RATE is assumed to be the annual interest rate. Notice that this statement has a statement label (1000) in addition to a line number (40).

```
50          PRINCIPAL=PRINCIPAL↵
            +XINTEREST-PAYMENT
```

Line 50 is also a replacement statement. This statement is the heart of the program, for it is here that the new principal is computed by adding the monthly interest computed in line 40 to the current value of PRINCIPAL and subtracting the value of the monthly PAYMENT.

```
60          IF (PRINCIPAL .LE. 0.)↵
            GO TO 2000
```

The IF statement in line 60 tests the current value of the variable PRINCIPAL. If PRINCIPAL is less than or equal to zero, the loan has been repaid, and the problem solution is completed. If 'PRINCIPAL .LE. 0.' is true, program execution is transferred to the statement labelled 2000 and the program terminates. If 'PRINCIPAL .LE. 0.' is false, execution will not transfer to statement 2000, but rather will continue in normal sequence. A transfer using a control statement can be made only to a statement having a statement label.

```
70          DISPLAY MONTH, PRINCIPAL
```

Line 70, a DISPLAY statement, is used to print the answers. When this statement is encountered, the values of the variables MONTH and PRINCIPAL are

1 - The variable name XINTEREST is chosen instead of INTEREST because decimal accuracy is desired in calculations involving XINTEREST. For a complete discussion of this point, see *Variables*, Page 20.

printed on the terminal. The DISPLAY statement is used for free form output just as the ACCEPT statement is used for free form input. This means that the system supplies the format that is used for output. Free form input and output statements simplify the input and output procedures for the user.

Note carefully the difference between the DISPLAY statements in line 20 and line 70. In line 20 the names MONTH and BALANCE are printed. In line 70 the values of the variables MONTH and PRINCIPAL are printed.

```
80              MONTH=MONTH+1
```

This statement is a replacement statement which increments the value of the variable MONTH by 1.

```
90              GO TO 1000
```

Line 90 is another control statement. It is an unconditional GO TO statement which transfers program execution to statement 1000. We put in this transfer because, after displaying the current values of MONTH and PRINCIPAL, we wish to compute the balance after another interest charge and payment.

```
100  2000    STOP
```

The STOP statement stops execution of the program. Notice that it also has a statement number. Earlier in the program we asked for a transfer to this statement if the principal was less than or equal to zero.

```
110             END
```

The last statement is the END statement. Every Tymshare SUPER FORTRAN program must end with an END statement. The END statement may not be given a number; that is, we cannot transfer to this statement from any other part of the program.

To execute this program on the computer, log in and call SFORTRAN. When SUPER FORTRAN is ready to receive instructions, it prints the > symbol. Then you may begin typing the statements of your program with line numbers, ending each line with a Carriage Return. If you type a statement with a syntax error, the computer will print a message indicating the nature of the error. Retype the statement correctly. When you have finished typing all the statement, the computer will type a > and the program is ready to be executed.

To execute the program just written, type RUN or EXECUTE, followed by a Carriage Return.

Shown below is the sample program typed on line and executed.

(Terminal identification character may be requested here.)

```
PLEASE LOG IN:
ACCOUNT:   A3
PASSWORD:
USER NAME: MM
PROJ CODE: KL-3-456

TYMSHARE   10/14   11:49

-SFORTRAN

>10 ACCEPT LOANNUMBER,
PRINCIPAL,RATE,PAYMENT
>20 DISPLAY "MONTH   BALANCE"
>30 MONTH=1
>40 1000 XINTEREST=PRINCIPAL
*RATE/12
>50 PRINCIPAL=PRINCIPAL
+XINTEREST-PAYMENT
>60 IF (PRINCIPAL .LE. 0.)
GO TO 2000
>70 DISPLAY MONTH,PRINCIPAL
>80 MONTH=MONTH+1
>90 GO TO 1000
>100 2000 STOP
>110 END
>RUN
1336,1500.00,0.08,125.00
MONTH   BALANCE
  1       1385
  2       1269.2333
  3       1152.6949
  4       1035.3795
  5       917.28205
  6       798.39727
  7       678.71991
  8       558.24471
  9       436.96634
  10      314.87945
  11      191.97865
  12      68.258508

(@100 )>
```

When execution of the program is completed, the computer prints

```
(@100 )>
```

indicating that the STOP statement at line 100 was the last statement executed.

To save a debugged program for future use, type

SAVE file name ↵

The file name can be almost any name you want to give your program.[1] After you have executed the above program, you could do the following:

```
>SAVE LOANBALANCE ↵
TEXT ONLY?Y ↵
 NEW FILE ↵
```

Typing Y↵ in response to the question TEXT ONLY? creates a symbolic file, containing only the text of the program. Typing N↵ here creates a binary file containing both the text and the compiled version of the program. These options are treated in detail in Section 10 of this manual.

The computer then tells you whether the file is a NEW FILE or an OLD FILE. Saving the program on an old file will erase the contents of that file and replace them with the current program.

Type a Carriage Return after NEW FILE or OLD FILE if you want to save the program on that file. If you do not want to save the program on that file, press the ALT MODE/ESCAPE key and choose another file name.

The entire program, including line numbers, is now stored on the file named LOANBALANCE, and a > is printed.

To use the program in the future, type

```
>LOAD LOANBALANCE ↵
```

The computer will reply OK. and begin loading. When it has finished, it will type a >. Now the program can be executed again.

Suppose you want a listing of the program. After you have loaded the program, type LIST followed by a Carriage Return. The program, including line numbers, is printed neatly on the terminal; the statement labels are aligned and the statements indented.

Note the use of the LOAD and LIST commands in the following example.

```
>LOAD LOANBALANCE ↵
OK.
>LIST ↵
  10          ACCEPT LOANNUMBER,↓
              PRINCIPAL,RATE,PAYMENT
  20          DISPLAY "MONTH   BALANCE"
  30          MONTH=1
  40    1000  XINTEREST=PRINCIPAL ↓
              *RATE/12
  50          PRINCIPAL=PRINCIPAL ↓
              +XINTEREST-PAYMENT
  60          IF (PRINCIPAL .LE. 0.) ↓
              GO TO 2000
  70          DISPLAY MONTH,PRINCIPAL
  80          MONTH=MONTH+1
  90          GO TO 1000
 100    2000  STOP
 110          END
>
```

The commands RUN, EXECUTE, SAVE, LOAD, and LIST are Tymshare CCS SUPER FORTRAN commands. They are not part of the SUPER FORTRAN language but are operating features of the Tymshare Conversational Compiler System (CCS). You will learn about the other CCS commands in Section 10 of this manual.

MORE SUPER FORTRAN FEATURES

In this section, we shall introduce some additional SUPER FORTRAN language elements.

VARIABLES

There are several different types or modes of variables available in SUPER FORTRAN. The two most often used variable types are Integer and Real. An Integer variable represents an integer number without a decimal point. A Real variable represents a real number, including the decimal and any digits to the right of the decimal. SUPER FORTRAN arithmetic distinguishes between computations involving integers and those with real numbers. For example, 3/2=1, but 3./2.=1.5.

Unless explicitly specified otherwise, any variable name beginning with one of the letters I through N inclusive is treated as an integer variable. All other variable names are treated as real variables.

A scalar variable represents a single quantity, and may be of either Real or Integer type. All the variables under the heading *"Introduction To SUPER FORTRAN Language Elements"* are scalar variables.

1 - See *Rules For Naming Files, APPENDIX C*, Page 160.

A group of variables may be closely related, and it is frequently desirable to use subscript notation to identify them. Such a group is called an array and the variables belonging to the array are called array elements. For example A(1), A(2), A(3), and A(4) may refer to four observational values of a single phenomenon. They are elements of the array A.

DECLARATION STATEMENTS

If we wish a variable to be of a type other than that indicated by the spelling of its name, we must declare this explicitly in the program. To do this, we use a type declaration statement, REAL or INTEGER. For example, if we want the variable KILOS to represent a real decimal number, we write:

```
REAL KILOS,NAME(27)
```

Similarly, to use the variables A and C as integers, we write:

```
INTEGER A,C(5,70)
```

In order to use array variables, we must reserve space in the computer for them. The DIMENSION declaration does this. For example, if we wish to use a 15-element array N and a 31-element array TEMP, we write:

```
DIMENSION N(15),TEMP(31)
```

CONTROL STATEMENTS

One of the most valuable features of a computer is its ability to perform a given procedure repeatedly, making minor changes each time. The DO statement provides this ability.

Consider the following sequence of statements.

```
   DO 30 I=0,100,2
   J=I**2
   DISPLAY I,J
30 CONTINUE
```

DO 30 I=0,100,2 controls the repetition of all succeeding statements up to and including the statement labelled 30. Repetition is controlled by varying the index variable I from an initial value of 0 to a terminal value of 100 in increments of 2. The statements to be repeated comprise a DO loop. The DO loop above creates a listing of the numbers and their square for even integers from 0 to 100 inclusive.

The CONTINUE statement causes no action; it merely serves as a dummy statement to refer to the end of the loop.

LIBRARY FUNCTIONS

There are many built-in functions provided by the SUPER FORTRAN library. These are functions which are used frequently in computational work and include trigonometric functions, logarithmic functions, the square root function, and the absolute value function.

The library functions may be used simply by naming the function and placing the argument (or arguments) in parentheses.

For example:

```
SIN(X)
SQRT(A+3)
LOG(C/(D-1.))
```

FORMATTED OUTPUT

It is frequently desirable to have the computed results displayed on the terminal in a neater form than that generated by the free format DISPLAY statement. Therefore, a formatted output statement can be used to print the answers in columns with the decimal points aligned.

As an example, assume we wish to print computed values of A and B to four decimal places of accuracy. The values of A should be in one column and the values of B in a second column, the two columns separated by six spaces. The following statements accomplish this:

```
    WRITE (1,100) A, B
100 FORMAT (F12.4,6X,F12.4)
```

The 1 in the WRITE statement signifies that the values of the variables are to be written on the terminal. 100 is an arbitrary number, identifying the statement label of the FORMAT statement used. F12.4 specifies the format in which A is to be printed. The F format prints a real number, including the decimal. F12.4 causes a maximum of twelve characters (including the decimal point) to be printed, with a maximum of four digits after the decimal. 6X specifies six spaces between the two numbers.

Formatted input is also permitted. Formatted input and output are discussed in detail in Section 6 of this manual.

USING SUPER FORTRAN

We are going to take a sample problem and go through the programming steps to arrive at a solution. We will then show how to enter, debug, and execute the program on the terminal.

PROBLEM

Write a program to calculate the mean and standard deviation of sets of N numbers, where N is greater than 1, and to print the normalized data.

Input

Required are:

- N - the number of observations in the set being considered. (If a number less than or equal to 1 is entered for N, it should cause the program to stop.)
- A_i - the values of the observations, where i=1,2, ...,N.

Compute Mean And Standard Deviation

$$\text{Mean} = \frac{\sum_{i=1}^{N} A_i}{N}$$

$$\text{Standard Deviation} = \sqrt{\frac{\sum_{i=1}^{N} A_i^2 - \frac{\left(\sum_{i=1}^{N} A_i\right)^2}{N}}{N-1}}$$

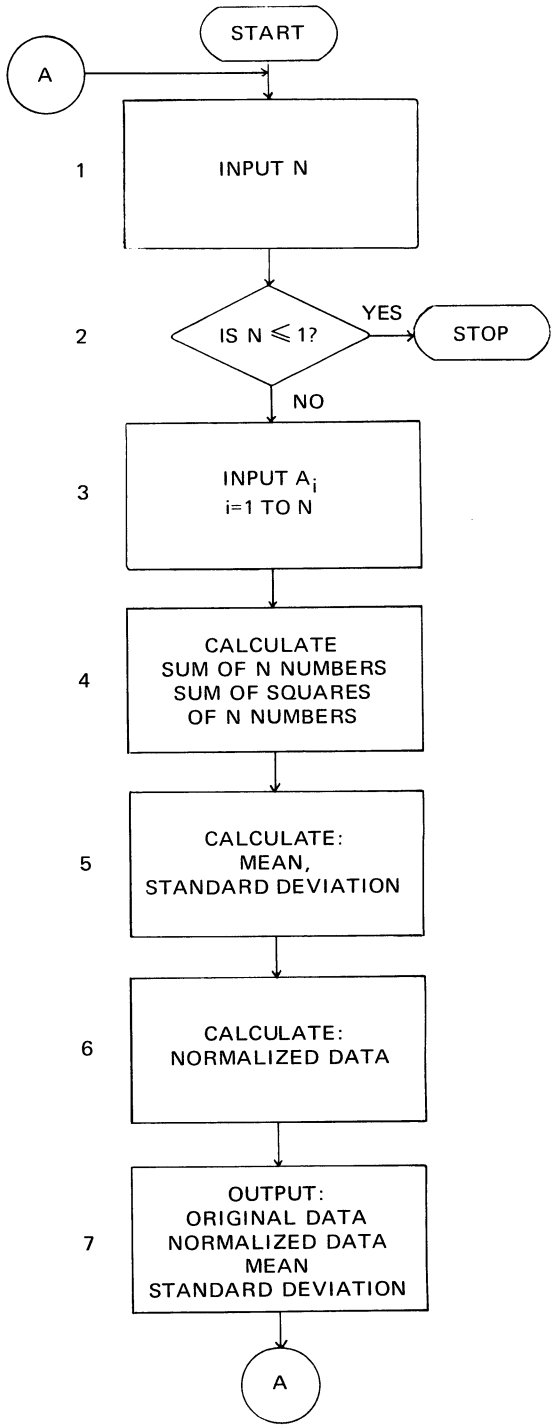
$$\text{Normalized Data} = \frac{A_i - \text{Mean}}{\text{Standard Deviation}} \quad i=1,2,\ldots,N$$

Output

Original data, normalized data, mean, and standard deviation.

FLOWCHART

Here is a flowchart of the problem:



PROGRAM CODING

Here is the coding of the program:

```
10        DIMENSION A(15)
20        REAL N, MEAN, NA(15)
30   40   ACCEPT "NUMBER OF ↵
          OBSERVATIONS: ",N
40        IF (N .LE. 1) GO TO 10
50        ACCEPT (A(I),I=1,N)
60        SUM=0.
70        SUMSQ=0.
80        DO 20 I=1,N
90        SUM=SUM+A(I)
100       SUMSQ=SUMSQ+A(I)**2
110  20   CONTINUE
120       MEAN=SUM/N
130       STD=SQRT((SUMSQ-SUM*SUM/N) ↵
          /(N-1))
140       DO 50 I=1,N
150       NA(I)=(A(I)-MEAN)/STD
160  50   CONTINUE
170       DISPLAY "INPUT DATA",(A(I),I=1,N)
180       DISPLAY "NORMALIZED DATA", ↵
          (NA(I),I=1,N)
190       DISPLAY "MEAN=",MEAN, ↵
          "STAND.DEV.=",STD
200       GO TO 40
210  10   STOP
220       END
```

This program reads the N data values into the array A. Thus, A(1) is the first data observation and A(N) the last data observation.

Line 10 reserves fifteen storage locations for array elements A(1) to A(15). If more than fifteen observations are anticipated in any set of data, the number of storage locations reserved must be large enough to accommodate them.

Line 20 is a type declaration statement. Without this statement, the variables N and MEAN and the array NA would be treated as integer variables. The array NA is implicitly dimensioned within this type declaration statement.

Line 30 is an ACCEPT statement with descriptive text within quote marks. The text will be printed on the terminal when the program is executed. The bell on the terminal will ring to signal that a value for N should be entered.

Line 50 reads the N values and stores them in array A. This is accomplished by means of an implied DO loop. This statement is equivalent to:

```
      DO 100 I=1,N
      ACCEPT A(I)
100   CONTINUE
```

Lines 60 to 110 correspond to box 4 of the flowchart. Lines 60 and 70 initialize the variables SUM and SUMSQ to zero. Lines 80 to 110 create a DO loop which sums the N numbers and the squares of the N numbers, each time adding the present value or square to the variables SUM and SUMSQ respectively.

After calculating the sum and the sum of the squares, we proceed to calculate the mean and standard deviation. Line 120 calculates the mean. In calculating the standard deviation in line 130, we use the library function SQRT to find the square root. Note that the argument, or expression, used with a library function is enclosed in parentheses.

Lines 140 through 160 correspond to box 6 in the flowchart. Another DO loop is used to normalize each of the N data values.

The next three lines are output statements. Text within quote marks is displayed describing the output in each case. Lines 170 and 180 contain implied DO loops. When line 190 is executed, the text MEAN= will appear on the terminal, followed by the value of the variable MEAN; then the text STAND.DEV.= and the value of the variable STD.

ENTERING THE PROGRAM

Tymshare SUPER FORTRAN provides commands which make it easy to create programs on the terminal. These commands include line prompting, debugging, and editing features.

If you do not want to type the line numbers when you are first entering a program, ask the computer to prompt you with line numbers. To do this, type a line number followed by an increment in parentheses. The computer will prompt you with line numbers starting with the number you specify and increasing each time by the increment specified. A Carriage Return is required at the end of each statement. After the last statement of the program has been entered, type a Control D (D^c) to terminate the entering phase.

NOTE: Throughout this manual, control characters are indicated by the superscript c; for example, D^c denotes Control D.

Many editing features are available when entering a statement. For example, Control A deletes the last character typed. If you attempt to enter a statement containing a syntax error, the computer prints a message indicating the nature of the error. This statement becomes the old line for editing purposes, and any of the control characters described in this manual may be used.

In creating the following program, some of these features will be demonstrated.

```
> 10(5) ↵
10 DIMENSION A(5) ↵
15 40 ACCEPT "NUMBER OF ↓
OBSERVATIONS: ",N ↵
20 IF (N .LE. 1) GO TO 10 ↵
25 ACCEPT (A(I),I=1,N) ↵
30 SUM=0. ↵
35 SUMMA^C←SQ=0. ↵
40 DO 20 I=1,N ↵
45 SUM=SUM+A(I) ↵
50 SUMSQ=SUMSQ+A(I)**2 ↵
55 20 CONTINUE ↵
60 MEAN=SUM/N ↵
65 STD=SQRT((SUMSQ-SUM*SUM/N) ↓
/(N-1)↵
MISSING OPERATOR
65 H^CSTD=SQRT((SUMSQ-SUM*SUM/N) ↓
/(N-1)) ↵
70 DO 50 I=1,N ↵
75 NA(I)=(A(I)-MEAN)/STD ↵
80 50 CONTINUE ↵
85 DISPLAY "INPUT DATA ",(A(I),I=1,N) ↵
90 DISPLAY "NORMALIZED DATA", ↓
(NA(I),I=1,N) ↵
95 DISPLAY "MEAN= ",MEAN, ↓
"STAND.DEV.= ",STD ↵
100 GO TO 40 ↵
105 10 STOP ↵
110 END ↵
115 D^C
> 12 REAL N,MEAN,NA(5) ↵
```

The command 10(5) causes the computer to prompt with line numbers beginning with line 10 in increments of 5.

In line 35, the user typed a second M instead of S. He typed a Control A to delete the M. A^C is acknowledged with a ←. The user then typed the rest of the line.

Line 65 contains an error. When the Carriage Return was typed to enter this statement, the computer responded with an error message and the old line number, 65. The text of the line is now available for editing. The user typed Control H to copy the line. The user then typed the missing final parenthesis.

When the computer prompted line 115, the user typed a Control D (D^C) to end the line prompt. The user had forgotten to declare N and MEAN as real variables, so he simply typed the statement as line 12. This statement will be inserted between lines 10 and 15.

EXECUTING THE PROGRAM

To execute the program, type RUN or EXECUTE. When an error is detected during execution, an error message is printed together with the statement which caused the error, and execution is terminated. At this point, you may enter direct statements for immediate execution to find out what caused the error. A direct statement may be any legal nondeclarative statement preceded by an @ sign.[1] A direct statement, once executed, is discarded and is not part of the program. Once the error is detected and corrected, the program may be executed again.

1 - See *Declaration Statements*, Page 79.

In the following examples, the program just entered is executed.

```
>RUN ↵
NUMBER OF OBSERVATIONS: 10 ↵
1,2,3,4,5,
SUBSCRIPT OUT OF RANGE
25 ACCEPT (A(I),I=1,N)
25 >@DISPLAY I ↵
 6

25 >DEFINITIONS A ↵
10 DIMENSION A(5)
75 NA(I)=(A(I)-MEAN)/STD

25 >10 DIMENSION A(15) ↵
> DEFINITIONS NA ↵
12 REAL N,MEAN,NA(5)
75 NA(I)=(A(I)-MEAN)/STD

> 12 REAL N,MEAN,NA(15) ↵

> RUN ↵

NUMBER OF OBSERVATIONS: 10 ↵
1,2,3,4,5,6,7,8,9,10 ↵
INPUT DATA 1      2      3      4      5      6      7      8      9      10
NORMALIZED DATA      -1.4863011      -1.156012      -.82572282
-.49543369      -.16514456      .16514456      .49543369
.82572282      1.156012      1.4863011
MEAN=      5.5      STAND.DEV.=      3.0276504
NUMBER OF OBSERVATIONS: 0 ↵

(@105 )>
```

The user tried to execute the program.

After five data values were entered, the computer typed an error message, the statement in error, and 25 >. The user typed an @ for immediate execution of the statement typed after it, DISPLAY I.

He then typed the CCS command DEFINITIONS A which listed the statement in which A is dimensioned. He found that he had not reserved enough space for the array A. The computer also printed statement 75, which reminded him to check the dimension of the array NA.

The computer prompted 25 >. The user typed line 10 again to redimension A. The old line 10 is replaced by this line. He then typed DEFINITIONS NA which listed the statement in which NA is dimensioned. He found that he had not reserved enough space for NA. He retyped line 12 to redimension NA.

He ran the program again, this time successfully.

After obtaining the answers, the user decides to have the output in a neater form. He decides to use formatted output. One format for writing the data and the normalized data is:

```
    WRITE  (1,200)(A(I),NA(I),I=1,N)
200 FORMAT  (F10.3,5X,F10.3)
```

Each element of the arrays A and NA is to be printed as a real number with three places after the decimal point. 5X specifies five spaces between the two numbers.

A formatted output for mean and standard deviation may be:

```
   WRITE (1,300) MEAN,STD
300 FORMAT ("MEAN=",F10.3,2X,"STAND.DEV.=",F6.3)
```

The program may be changed to use formatted output. The user types 85:95 which allows him to enter statements in that line range. The computer prompts with an @ sign at the beginning of each line. The statements entered will replace any old lines in the range and will be numbered beginning with 85 in increments of the first of 1, .1, .01, and .001 that allows the lines typed to fit in the range 85:95.

```
>85:95 ↵
@DISPLAY " INPUT DATA      NORMALIZED DATA" ↵
@WRITE (1,200) (A(I),NA(I),I=1,N) ↵
@200 FORMAT (F10.3,5X,F10.3) ↵
@WRITE (1,300) MEAN,STD ↵
@300 FORMAT ("MEAN=",F10.3,2X,"STAND.DEV.=",F6.3) ↵
@D^C
>LIST 85:95 ↵
 85            DISPLAY " INPUT DATA      NORMALIZED DATA"
 86            WRITE (1,200) (A(I),NA(I),I=1,N)
 87      200   FORMAT (F10.3,5X,F10.3)
 88            WRITE (1,300) MEAN,STD
 89      300   FORMAT ("MEAN=",F10.3,2X,"STAND.DEV.=",F6.3)
>
```

The user typed a Control D to indicate that he had finished entering statements. The computer prompted >. The user typed

```
LIST 85:95 ↵
```

and the computer then printed all the statements within this range. Notice that the statements just entered were numbered from 85 to 89 inclusive, in increments of 1.

Now, when the program is run, the output is aligned.

```
> RUN ↵
NUMBER OF OBSERVATIONS: 10 ↵
1,2,3,4,5,6,7,8,9,10 ↵
 INPUT DATA     NORMALIZED DATA
      1.000            -1.486
      2.000            -1.156
      3.000             -.826
      4.000             -.495
      5.000             -.165
      6.000              .165
      7.000              .495
      8.000              .826
      9.000             1.156
     10.000             1.486
MEAN=    5.500     STAND.DEV.= 3.028
NUMBER OF OBSERVATIONS: 0 ↵


(@105 )>
```

Here is the listing of the entire program:

```
> LIST ↵
  10            DIMENSION A(15)
  12            REAL N,MEAN,NA(15)
  15      40    ACCEPT "NUMBER OF OBSERVATIONS: ",N
  20            IF (N .LE. 1) GO TO 10
  25            ACCEPT (A(I),I=1,N)
  30            SUM=0.
  35            SUMSQ=0.
  40            DO 20 I=1,N
  45            SUM=SUM+A(I)
  50            SUMSQ=SUMSQ+A(I)**2
  55      20    CONTINUE
  60            MEAN=SUM/N
  65            STD=SQRT((SUMSQ-SUM*SUM/N)/(N-1))
  70            DO 50 I=1,N
  75            NA(I)=(A(I)-MEAN)/STD
  80      50    CONTINUE
  85            DISPLAY " INPUT DATA     NORMALIZED DATA"
  86            WRITE (1,200) (A(I),NA(I),I=1,N)
  87      200   FORMAT (F10.3,5X,F10.3)
  88            WRITE (1,300) MEAN,STD
  89      300   FORMAT ("MEAN=",F10.3,2X,"STAND.DEV.=",F6.3)
 100            GO TO 40
 105      10    STOP
 110            END
>
```

SECTION 3
SUPER FORTRAN STATEMENT ELEMENTS

SUPER FORTRAN statements may contain constants, variables, and functions which may be combined with arithmetic, logical, and relational operators to form expressions in much the same way as in ordinary mathematics. In this section we present the rules for forming these basic statement elements.

CONSTANTS

A constant is a quantity in a statement which cannot change during program execution. For example, the number 11 is a constant. There are six different types of constants in SUPER FORTRAN.

INTEGER CONSTANTS

An integer constant is a positive or negative whole number, or zero, such as:

0
-1245
3859437 *or +3859437*

A number written without a sign is always considered positive. No integer may have a value larger than 2^{23} -1 or smaller than -2^{23}. *NOTE: An integer constant does not contain a decimal point.*

REAL CONSTANTS

Real constants have two forms: decimal and exponential. In both cases, the magnitude (absolute value) of the constant can range from 10^{-75} to 10^{75}, or be zero. The computed accuracy of a real constant is eleven significant digits.

In decimal form, a real constant contains the sign of the number (optional for positive numbers and zero), one or more digits, and a decimal point. The decimal point must be included; it may appear anywhere in the number.

Examples

0.
3.1415926536
-0.07
.0000567

In exponential form, a real constant contains one or more digits with or without a decimal point, followed by the letter E and the exponent. The part of the number before the E is the mantissa; that part after the E is the exponent. The exponent is an integer and represents the power of ten by which the mantissa is to be multiplied.

Examples

Exponential Form	Value
5E-2	.05 (5×10^{-2})
1E7	10^7
-1.973E03	-1973.
.0271828E+2	2.71828
+25E3	25000

When a real constant is represented in exponential form, the exponent can range from -75 to +75; the mantissa may have up to 11 significant digits.

DOUBLE PRECISION CONSTANTS

A constant may have up to 17 significant digits if it is expressed in D exponential form. This form is just like the exponential form for real constants given above, except that a D is used in placed of an E between the mantissa and the exponent.

Examples

Double Precision Constant	Value
5D-2	.05
.12345678901234567D5	12345.678901234567
-1.973D+3	-1973.
3.14159265359D0	3.14159265359

COMPLEX CONSTANTS

A complex constant is expressed as two real constants (each with eleven digits of accuracy) separated by a comma and enclosed in parentheses. The first number represents the real part of the complex number; the second represents the imaginary part of the complex number.

Examples

Complex Constant	Value
(3.,5.2)	3+5.2i
(-1.8,.16E2)	-1.8+16i
(2.4,0.)	2.4+0i
(0.,-6E-1)	-.6i

LOGICAL CONSTANTS

There are two logical constants, .TRUE. and .FALSE. .

STRING CONSTANTS

A string constant is a sequence of characters enclosed in double or single quote marks.

Examples

"XYZ"
'CODE 387'
"ISN'T THIS FUN?" *This string must be enclosed in double quotes since it contains a single quote.*

The value of a string constant is the string of characters inside the delimiting quote marks; thus, the value of "XYZ" is XYZ.

Unlike standard FORTRAN IV, Tymshare SUPER FORTRAN includes string variables and extensive features for string processing, as well as string constants. These features are discussed in *Strings*, Section 5.

VARIABLES

A variable is a quantity whose value can be changed throughout the program. For example, variables may be assigned new values in input statements and in replacement statements.

VARIABLE NAMES

In a SUPER FORTRAN program, a variable can be named with as many as 31 alphanumeric characters. The first character of the name must be alphabetic (A through Z). For example, the following are all valid variable names:

N
INDEX
ALPHA
X12

but

4Z1

is not, since it begins with a numeric character.

VARIABLE TYPES

There are six types of variables: integer, real, double precision, complex, logical, and string. The variable type corresponds to the type of data the variable represents. Thus, an integer variable represents integer data, a real variable represents real data, and so on.

Integer and real data types may be declared implicitly. If the first letter of a variable name is I, J, K, L, M, or N, the variable is integer. Any other variable name not appearing in any type declaration statement is real. However, explicit type declaration in a type declaration statement overrides implicit type declaration determined by spelling. Data types other than integer and real **must** be declared explicitly with a type declaration statement. (See *Declaration Statements*, Page 79.)

Example

In the statements

REAL MEAN
MEAN=SUM/N

SUM is a real variable and N is an integer variable (implicit type declaration) but MEAN is a real variable even though its name begins with an M, since its type is declared explicitly with the type declaration statement

REAL MEAN

NOTE: Certain valid variable names are given special meanings; for example, SIN is reserved for naming the mathematical function sine (see **Functions**, *Page 25). Such reserved words may not be used as variable names unless they are declared explicitly in a type declaration statement. Once a reserved word is so declared, it may not be used in its usual sense. For example, if SIN is declared to be a real variable with the declaration statement REAL SIN, the statement Y=SIN(X) could not be used to set Y equal to the sine of X in the same program.*

SCALAR VARIABLES

A scalar variable represents a single quantity, such as MEAN, N, and SUM in the above example.

ARRAYS AND SUBSCRIPTED VARIABLES

A group of variables which form or belong to a single class or collection may be related to one another by subscript notation. Such a collection is called an array, and the variables belonging to the array are called array elements.

A string of numbers in a single row or column is thought of as a one-dimensional array. In the following example, the entire array has the variable name A,

and each element of A is represented by a **subscripted variable** consisting of the variable name A followed by a subscript in parentheses:

Usual Notation	SUPER FORTRAN Notation
a_1	A(1)
a_2	A(2)
.	.
.	.
a_i	A(I)
.	.
.	.
.	.
a_n	A(N)

If two subscripts are used to identify the elements of an array, the array is a two-dimensional array. For example, if there are three rows and four columns in a table, A(2,3) could refer to the element in the second row and third column.

In SUPER FORTRAN there is no limit to the number of dimensions of an array. Whatever the number of dimensions, the entire array is represented by a single variable name, such as A above, and each element of the array is represented by a subscripted variable. A subscripted variable is denoted by the array name followed by a list of subscripts (one for each dimension) separated by commas and enclosed in parentheses. Each subscript can be any arithmetic expression (see *Expressions*, below).

Examples

A(3)
B(5,-5)
C(0)
Y(M,1,1,N+3)
VOLTAGE(2*N+1,L,L+1)

The array type may be integer, real, double precision, complex, logical, or string. As usual, the type must be declared in a type declaration statement unless it is integer or real; integer or real arrays may be declared implicitly.

Since an array is an entire collection of variables, the programmer must specify the maximum number of elements in all the arrays in his program to reserve storage for all the array elements. This can be done either in a DIMENSION statement or a type declaration statement (see Section 7, *Declaration Statements*).

VARIABLE INITIALIZATION

When a SUPER FORTRAN program is executed using the CCS command RUN,[1] all variables in the program are initialized to zero. Thus,

```
>1 ACCEPT A↵
>2 DISPLAY "A =",A,"BUT B =",B↵
>3 END↵
>RUN↵
13.5↵
A =    13.5     BUT B =    0
                          B has the value 0 since it
                          was not assigned a value
                          anywhere in the program.
(@3 )>
```

Variables are also initialized to zero when a binary program is executed with the CCS LINK command,[2] but are not so initialized when a binary program is executed with the SUPER FORTRAN statement LINK,[3] since this statement preserves COMMON.

EXPRESSIONS

There are three kinds of expressions in SUPER FORTRAN: arithmetic, logical, and string. Arithmetic and logical expressions are discussed in this section; the rules for forming string expressions are found in *Strings*, Page 40.

An expression always has a value. The value of an arithmetic expression is always an integer, real, double precision, or complex number; the value of a logical expression is either .TRUE. or .FALSE. .

ARITHMETIC EXPRESSIONS

A simple arithmetic expression may consist of a single basic element whose value is numeric; that is, a numeric constant, variable, or function.[4]

Examples

3.14
X
A(5)
SQRT(ALPHA)

More complicated arithmetic expressions may be formed from simple arithmetic expressions by using arithmetic operators which determine the computations to be performed.

Examples

A+5
3*(PI-2+B)
SQRT(X)-Y

1 - See *Executing A Program*, Page 113.
2 - See *The LINK Command*, Page 109.
3 - See *Program Linking*, Page 98.
4 - See *Functions*, Page 25.

Arithmetic Operators

The following binary operators (operators used with two operands) are available:

** *or* ↑	Exponentiation
/	Division
*	Multiplication
-	Subtraction
+	Addition

Unary arithmetic operations; that is, operations involving only one operand, also are available. There are two unary operators, + and −. For example,

-B	means negative of B
+A	means A

Order Of Operation

1. Parentheses may be used to specify the order of operation in an expression. When sets of parentheses appear within other sets of parentheses, the expression in the innermost set is evaluated first, then the expression in the next set, and so on.

2. Expressions not containing parentheses (including expressions within parentheses) are evaluated in the following order:

Exponentiation (** or ↑)

Unary minus (-)

Multiplication and Division (* and /)

Addition and Subtraction (+ and -)

3. Arithmetic expressions containing operators of equal priority (such as * and /) are evaluated from left to right. Thus, when two operators of the same precedence appear, the leftmost operation is performed first.

Examples

A+B*C	means	A+(B*C)
A/B/C	means	$\frac{A}{B \cdot C}$
A/B*C/D	means	$\frac{\frac{A}{B} * C}{D}$
A+B/C2**	means	$A+\frac{B}{C^2}$
((A+B)/C)2**	means	$\left(\frac{A+B}{C}\right)^2$
-C2**	means	$-C^2$

NOTE: Square brackets may be used in place of parentheses in arithmetic expressions. Complicated expressions can be made more readable by using both parentheses and brackets. Contrast

[(A-B)/(X+Y)] * [N↑4]

and the equivalent expression

((A-B)/(X+Y))*(N4)**

Modes Of Expressions: Mixed Expressions

The kind of arithmetic performed when an arithmetic expression is evaluated depends on the type, or mode, of the operands. For example, if both operands are integers, integer arithmetic is performed. Thus, 3/4 causes an integer division and has the value zero. But 3./4. gives a result of .75. Since both operands are real, real division is performed.

Arithmetic expressions containing constants or variables of more than one type are called mixed expressions. In a mixed expression, as each operation is performed, the types of the two operands are compared; the lower type is converted to the higher type and the result is of the higher type. The hierarchy of types is as follows:

Type Of Operand:

Complex — Highest
Double Precision
Real
Integer — Lowest

Examples[1]

A/I	The integer I is converted to real and real division is performed.
N/(I+2.)	The expression (I+2.) is first considered. Since 2. is real, the integer I is converted to real and the result of the addition is real. The integer N is then converted to real since it is to be divided by a real number. The result of the whole expression is real. If N=3 and I=4, the result is 0.5.
A*(N/2)	The result of the expression (N/2) is an integer since both operands are integers. (N/2) is then converted to a real number since it is to be multiplied by A which is real. If A=6. and N=3, the result is 6. .
A*(N/2.)	N is first converted to real since it is to be divided by a real number 2. . Real division is performed. Two real numbers are then multiplied together. If A=6. and N=3 the result is 9. . Remember, 3./2.=1.5 but 3/2=1.

1 - In these examples, implicit type declaration is assumed unless otherwise specified.

B+I If B is a complex variable with a value of (2.7,8.1) and I=1, I is converted to the complex number (1.,0). The result is (3.7,8.1).

D+B If D is a double precision variable and B a complex variable, the value of D is truncated to single precision, then treated as a complex number with zero as the imaginary part.

D*R If D is a double precision variable and R is a real variable, the value of R is converted to double precision and double precision multiplication is performed.

LOGICAL EXPRESSIONS

A logical expression may consist of a single logical constant or a logical variable. The value of a logical expression is always a truth value, .TRUE. or .FALSE. .

More complicated logical expressions may be formed by using logical and relational operators. These expressions may be one of the following:

1. Relational operators combined with arithmetic expressions.
2. Logical operators combined with logical constants or logical variables.
3. Logical operators combined with either or both forms of the logical expressions described in 1 and 2 above.

Relational Operators

A relational operator makes a comparison between arithmetic expressions. For example, the relational operator .GT. (greater than) may be used to compare the real variables X and Y in the logical expression

X.GT.Y

This expression has the value .TRUE. if X is greater than Y, and the value .FALSE. if X is not greater than Y (that is, if X is less than or equal to Y).

Comparisons may be made by means of any of the following relational operators:

Symbol	Mathematical Notation	Meaning
.EQ.	=	Equal to
.NE.	≠	Not equal to
.LT.	<	Less than
.LE.	≤	Less than or equal to
.GT.	>	Greater than
.GE.	≥	Greater than or equal to

Examples

A .EQ. B
X-5 .LT. Y+4
SQRT(BETA) .GE. ALPHA↑2

Unlike most FORTRAN IV compilers, SUPER FORTRAN can make comparisons between arithmetic expressions of different types. For example,

A .LE. J

is valid. J will be converted to a real number before the comparison is made. The hierarchy of types is the same as that used in mixed arithmetic expressions:

Type Of Expression Being Compared:

Complex — Highest
Double
Real
Integer — Lowest

When two arithmetic expressions of different types are being compared, the lower type is first converted to the higher type and then the comparison is made. For example, in the logical expression

D. EQ. R

if D is a double precision variable and R is a real variable, the value of R is first converted to double precision and then compared to the value of D. If R=1E0 and D=1D0 the value of this expression is .TRUE. . If R=1E0 and D=1.0000000000034D0 the value of this expression is .FALSE. .

When complex values are compared using .EQ. and .NE., both the real and the imaginary parts are compared.

EXAMPLES

(3.,4.) .EQ. (3.,4.) is .TRUE. since the real and imaginary parts of both operands are equal.

(1.,2.1) .EQ. 1 is .FALSE. . Here, the integer 1 is first converted to the complex number (1.,0.). Since the imaginary part of this number is not equal to 2.1 the expression is .FALSE.

However, when complex values are compared using .LT., .GT., .LE., and .GE., only the real parts of the numbers are compared. Thus,

(1.,9.) .LT. (2.,.8) is .TRUE. since the real part of the first operand (1.) is less than the real part of the second (2.).

(2.,-3.) .GT. (3.,-3.1) is .FALSE. since 2. is not greater than 3. .

Logical Operators

A logical operator operates on logical expressions. The result of a logical operation is always either .TRUE. or .FALSE. . For example, the logical operator .AND. may be used in the logical expression

X .AND. Y

where X and Y are logical variables. The value of this expression is .TRUE. only if both X and Y are .TRUE. .

There are six logical operators: .AND., .OR. (inclusive or), .EOR. (exclusive or), .NOT., .EQV. (equivalence), and .IMP. (implication). The following table shows the results of the logical operations on the logical expressions X and Y:

OPERATOR	X Y	.TRUE. .TRUE.	.TRUE. .FALSE.	.FALSE. .TRUE.	.FALSE. .FALSE.
AND	X .AND. Y	.TRUE.	.FALSE.	.FALSE.	.FALSE.
Inclusive OR	X .OR. Y	.TRUE.	.TRUE.	.TRUE.	.FALSE.
Exclusive OR	X .EOR. Y	.FALSE.	.TRUE.	.TRUE.	.FALSE.
NOT	.NOT. X	.FALSE.	.FALSE.	.TRUE.	.TRUE.
EQV	X .EQV. Y	.TRUE.	.FALSE.	.FALSE.	.TRUE.
IMP	X .IMP. Y	.TRUE.	.FALSE.	.TRUE.	.TRUE.

Some examples of logical expressions containing logical operators are:

X .EQ. 3 .OR. Y .LE. 4

.NOT. W .AND. .NOT. L

.NOT. (W .AND. .NOT. L)

(A .GT. 100 .EOR. B .GT. 200) .EQV. C

E*5 .GT. A-B .IMP. E .LE. 100

Order Of Operation In Logical Expressions

1. Just as in arithmetic expressions, parentheses may be used to specify the order of operation in logical expressions. When sets of parentheses appear within other sets of parentheses, the expression in the innermost set is evaluated first, then the expression in the next set, and so on.

2. Expressions not containing parentheses (including expressions within parentheses) are evaluated in the following order:

Evaluation of functions (for example, SQRT)

Exponentiation (** or ↑)

Unary minus (-)

Multiplication and Division (* and /)

Addition and Subtraction (+ and -)

The relational operators:
.EQ., .NE., .LT., .LE., .GT., .GE.

.NOT.

.AND.

.EQV. or .IMP.

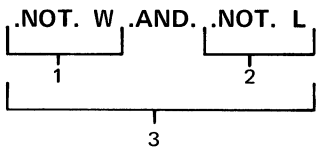
.OR.

.EOR.

3. Expressions containing operators of equal priority are evaluated from left to right.

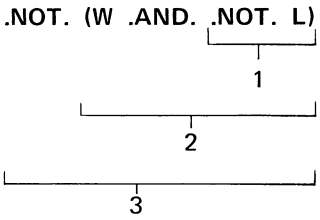
The following logical expressions are evaluated in the order indicated:

Example 1



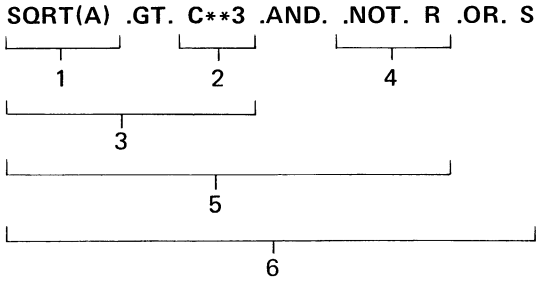
1. Leftmost logical operator .NOT.
2. Next logical operator .NOT.
3. Logical operator .AND.

Example 2



1. .NOT. in innermost parentheses
2. .AND. in innermost parentheses
3. Outermost .NOT.

Example 3



1. Functional evaluation
2. Exponentiation
3. Relational operator .GT.
4. Logical operator .NOT.
5. Logical operator .AND.
6. Logical operator .OR.

FUNCTIONS

In addition to constants and variables, there is another basic element which may be used in forming expressions, namely, the function reference. Functions are used in expressions in the form

function name $(a_1, a_2, \cdots a_n)$

where a_1 through a_n are the function arguments. These arguments can be constants, variables, or more complicated expressions. The number of arguments depends on the particular function. The arguments may be enclosed in square brackets as well as parentheses. For example,

SQRT(3.14)

may be used in an expression to return a value equal to the square root of 3.14. It may also be written as SQRT [3.14].

SUPER FORTRAN contains an extensive library of functions, such as the SQRT function above. In addition, the programmer may define his own functions; see *Subprograms: Programmer Defined Functions And Subroutines*, Page 89.

MATHEMATICAL FUNCTIONS

The SUPER FORTRAN function library contains a variety of functions, including mathematical functions, functions for converting variable types, string functions, and utility functions. The mathematical, conversion, and utility functions are discussed here; string functions are presented in Section 5 of this manual. A complete list of all the library functions may be found in the *SUPER FORTRAN Language Summary*, Page 161.

The mathematical functions described below all return a value of the same type as the argument used. For example, if R is a real variable and C is a complex variable,

SQRT(R) returns a real value, but

SQRT(C) returns a complex value.

This improvement over other versions of FORTRAN IV makes it unnecessary for the programmer to remember long lists of similar function names. For compatibility, the usual alternate names (such as DSQRT and CSQRT) are recognized by SUPER FORTRAN, but no distinction is made between them. Thus, if D is a double precision variable, SQRT(D), CSQRT(D), and DSQRT(D) all return the same double precision value. Unless otherwise specified in the following table, any type of argument may be used with any of the standard mathematical functions (integer arguments are converted to real).

STANDARD MATHEMATICAL FUNCTIONS				
Function	**Form**	**Mathematical Equivalent**	**Remarks**	**Alternate Names**
Exponential	EXP(a)	e^a		DEXP, CEXP
Natural Logarithm	LOG(a)	ln(a)		ALOG, DLOG, CLOG
Common Logarithm	LOG10(a)	$\log_{10}(a)$		ALOG10, DLOG10, CLOG10
Sine	SIN(a)	sin(a)	a in radians	DSIN, CSIN
Cosine	COS(a)	cos(a)	a in radians	DCOS, CCOS
Tangent	TAN(a)	tan(a)	a in radians	DTAN, CTAN
Arcsine	ASIN(a)	arcsin(a) or $\sin^{-1}(a)$	All inverse trigonometric functions (ASIN, ACOS, ATAN, ATAN2) return angle in radians	ARSIN, DARSIN
Arccosine	ACOS(a)	arccos(a) or $\cos^{-1}(a)$		ARCOS, DARCOS
Arctangent	ATAN(a)	arctan(a) or $\tan^{-1}(a)$		DATAN, CATAN
	ATAN2(a_1,a_2)	$\arctan\left(\frac{a^1}{a^2}\right)$	Complex arguments illegal	DATAN2
Hyperbolic Sine	SINH(a)	sinh(a)	Double precision argument illegal	CSINH
Hyperbolic Cosine	COSH(a)	cosh(a)	Double precision argument illegal	CCOSH
Hyperbolic Tangent	TANH(a)	tanh(a)	Real argument only	
Square Root	SQRT(a)	$\sqrt{a}$	Argument may not be negative unless it is also complex	CSQRT, DSQRT
Remaindering (Modular Arithmetic)	MOD(a_1,a_2)	$a_1 \pmod{a_2}$	Example: MOD(9,2)=1	AMOD, DMOD
Absolute Value	ABS(a)	|a| if a=b+ci, $\|a\| = \sqrt{b^2+c^2}$	Example: ABS(2) = 2 ABS(-7.1)=7.1 ABS((3.,4.))=5	IABS, DABS, CABS
Maximum Value	MAX(a_1,a_2...)	max(a_1,a_2...)	May have two or more arguments	AMAX0, AMAX1, MAX0, MAX1, DMAX1
Minimum Value	MIN(a_1,a_2...)	min(a_1,a_2...)	May have two or more arguments	AMIN0, AMIN1, MIN0, MIN1, DMIN1
Sign Function	SIGNUM(a)	sign of a	Value is: 1 if a>0 0 if a=0 -1 if a<0	

STANDARD MATHEMATICAL FUNCTIONS (Continued)				
Function	**Form**	**Mathematical Equivalent**	**Remarks**	**Alternate Names**
Transfer of Sign	SIGN(a_1,a_2)	sign of a_2 x $\|a_1\|$	Examples: SIGN(3,2)=3 SIGN(-3,-2)=-3 SIGN(3,-2)=-3 SIGN(6.1,0)=0	ISIGN, DSIGN
Greatest Integer Not Exceeding Value	ENTIER(a)	[a] (=greatest integer $\leqslant$ a)	Example: ENTIER(1.3)=1. ENTIER(-1.8)=-2.	
Truncation	TRUNC(a)	a truncated	Examples: TRUNC(1.3)=1. TRUNC(-1.8)=-1.	AINT
Rounding	ROUND(a)	a rounded to nearest integer	Examples: ROUND(3.56)=4. ROUND(-2.8)=-3.	
Fractional Part	FRACT(a)	fractional part of a (=a-ENTIER (a))	Examples: FRACT(1.63)=.63 FRACT(-1.8)=.2	
Positive Difference	DIM(a_1,a_2)	a_1-min(a_1,a_2)	Examples: DIM(4,3)=1 DIM(3,4)=0	IDIM

The following mathematical functions are used in complex arithmetic. These functions expect arguments of a particular type and return results of a particular type (not necessarily the same as that of the arguments).

COMPLEX FUNCTIONS				
General Form Of Function	**Meaning**	**Allowable Argument Types**	**Result Type**	**Acceptable Alternate Names**
CMPLX(a_1,a_2)	Creates a complex number equal to $a_1+a_2 i$	Real	Complex	
CONJG(a)	If a=b+ci then CONJG(a)=b-ci	Complex	Complex	
POLAR(a_1,a_2)	Creates a complex number with magnitude a_1 and phase angle a_2. (If PI=π, then POLAR (1,PI/2), is (0,1) or i).	Real	Complex	
IMAG(a)	Imaginary part of a (=c if a=b+ci)	Complex	Real	AIMAG

NOTE: The conversion function REAL discussed below may be used to obtain the real part of a complex number.

CONVERSION FUNCTIONS

The following functions are used for converting variable values from one type to another:

CONVERSION FUNCTIONS				
General Form Of Function	**Meaning**	**Allowable Argument Types**	**Result Type**	**Acceptable Alternate Names**
REAL(a)	Conversion of argument to real	Any arithmetic type	Real	FLOAT SINGL
INT(a)	Conversion of argument to integer (truncates)	Any arithmetic type	Integer	IFIX FIX IDINT
DBLE(a)	Conversion of argument to double precision	Any arithmetic type	Double Precision	DFLOAT

NOTE: The DBLE function should not be used to create double precision constants, since it is not as accurate as using D notation. Thus,

D=DBLE(1.63)

produces a double precision quantity with eleven accurate digits, whereas

D=1.63D0

is accurate to 17 digits.

UTILITY FUNCTIONS: DATE AND TIME

Two utility functions, DATE and TIME, are available in the SUPER FORTRAN function library. These functions both require a dummy argument, which may be of any type and which has no relation to the value returned.

The function

DATE(*a*)

where *a* is an argument of any type, returns a string value 12 characters long giving the date and time of day, such as

11/16 20:31

The function

TIME(*a*)

where *a* is an argument of any type, returns an integer value equal to the computer clock time in 60ths of a second. It can be used to determine the time required for a program to run, as in the following:

```
T0=TIME(X)
.
.     body of program
.
T1=TIME(X)
DISPLAY "TIME USED:",(T1-T0)/60,"SECONDS"
```

THE RANDOM NUMBER GENERATOR

SUPER FORTRAN contains a pseudo-random number generator which involves the use of both a function, RAND, and a statement, SETRAND.

The function

RAND(*a*)

where *a* is a dummy argument of any type, returns a real random number between 0 and 1, exclusive. If used alone, it will generate the same sequence of random numbers each time the program is run, no matter what argument is used. However, the statement

SETRAND (*arithmetic expression*)

may be used to control sequences of random numbers generated with RAND. The SETRAND statement initializes the random number generator according to the following rules:

1. If the arithmetic expression is zero, the same sequence of random numbers is initialized each time the program is run. However, SETRAND(0) used twice in a program initializes two different sequences of random numbers.

2. If the arithmetic expression is positive, the same sequence of random numbers is also initialized each time the program is run. If the same positive number is used twice in a program, the same sequence is initialized twice. However, different positive numbers yield different sequences. Thus,

```
SETRAND (16)
(DISPLAY RAND(X)), I=1,10
SETRAND (16)
(DISPLAY RAND(X)), I=1,10
```

print two identical sequences of random numbers, but

```
SETRAND (16)
(DISPLAY RAND(X)), I=1,10
SETRAND (103)
(DISPLAY RAND(X)), I=1,10
```

print two different sequences of random numbers.

3. If the arithmetic expression is negative, a sequence of random numbers subsequently generated with RAND will begin with a number set by reading the internal clock of the computer in milliseconds. The value of the negative argument has no relationship to the random numbers generated. For example, SETRAND(-1) used twice in a program initializes two different random number sequences. If a program using SETRAND with a negative expression is run twice, different sequences will be generated.

SECTION 4

REPLACEMENT AND CONTROL STATEMENTS

ARITHMETIC AND LOGICAL REPLACEMENT STATEMENTS

A replacement statement uses the replacement operator = to assign a value to a variable. All replacement statements have the form

variable=expression

indicating that the current value of the variable is to be replaced by the current value of the expression. The = means "is replaced by" rather than "is equivalent to".

There are three kinds of replacement statements in SUPER FORTRAN: arithmetic, logical, and string. Arithmetic and logical replacement statements are discussed here; string replacement is discussed in *Strings*, Page 40.

ARITHMETIC REPLACEMENT STATEMENTS

The general form of an arithmetic replacement statement is

variable=expression

where the variable is a scalar or subscripted variable and the expression is arithmetic. Execution of the statement causes the value of the variable to be replaced by the value of the expression. If the mode of the expression differs from the mode of the variable, the mode of the expression is converted to the mode of the variable before replacement occurs.

Examples

In the following, VAR, A, and B are real variables; I and J are integer variables; C is a complex variable; and D is a double precision variable.

VAR=A*B/5	The current value of VAR is replaced by the value of A*B/5. If A is 10. and B is 2., the value of VAR after execution of this statement is 4. .
I=5.66	The real constant 5.66 is converted to integer and then replaces the current value of I. Thus, I is set to 5.
A=I	The value of I is converted to a real value and then replaces the current value of A.
I=I+1	The value of I is replaced by the value of I+1.
A=C	The real part of the complex variable C replaces the value of A.
C=A	The value of A replaces the value of the real part of the complex variable C; the imaginary part of C is set to zero.
C=IJ**	I is raised to the power J and the result is converted to a real value which replaces the real part of the complex variable C. The imaginary part of C is set to zero.
D=J	The value of J is converted to double precision and stored in D.
D=3.65D-4	The double precision constant 3.65D-4 replaces the value of D.
C=(3.4,2.0)	The value of C is replaced by the complex constant (3.4,2.0). Note that C=(A,B) where A and B are real variables is not allowed. However, the same results may be achieved by using the CMPLX function, as in the next example.
C=CMPLX(A,B)	The value of the function CMPLX(A,B) is computed; this value (A+Bi) then replaces the value of C.

LOGICAL REPLACEMENT STATEMENTS

The general form of a logical replacement statement is

variable=expression

where the variable is a logical variable (scalar or subscripted) and the expression is a logical expression. Execution of the statement causes the variable to be replaced by a value of .TRUE. or .FALSE. depending on whether the expression is true or false.

Examples

In the following, H, K, and M are logical variables and I is an integer variable.

H=.TRUE.	The value of H is replaced by the logical value .TRUE. .
K=.NOT. M	If M is .TRUE., the value of K is replaced by the logical value .FALSE. . If M is .FALSE., the value of K is replaced by the logical value .TRUE. .
H=3. .EQ. I	The value of I is converted to a real value. If the real constant 3. is equal to this result, the logical value .TRUE. replaces the value of H. If 3. is not equal to this result, the logical value .FALSE. replaces the value of H.

NOTE: Replacement statements of the form

arithmetic variable=logical expression

. are allowed. Such expressions cause the value of the variable to be replaced by 1 if the value of the logical expression is .TRUE., and by 0 if the value of the logical expression is .FALSE. .

Examples

I=H	If H is a logical variable whose value is .TRUE., the value of the integer variable I is replaced by 1.
R=A .OR. B	If A .OR. B is .TRUE. and R is a real variable, the value of R is replaced by 1.

CONTROL STATEMENTS

SUPER FORTRAN statements are executed sequentially; that is, according to the order of their CCS line numbers.[1] Control statements allow the programmer to alter and control this sequence of execution.

STATEMENT LABELS

When control statements are used, certain statements may be labelled so that they may be referred to by the control statements. A statement label is always an integer; it may be from one to five digits long. Any executable statement can be labelled. The only nonexecutable statement that can be labelled is the FORMAT statement; however, control statements may not refer to this label.

Do not confuse SUPER FORTRAN statement labels with CCS line numbers.[1] Control statements can **never** refer to line numbers.

GO TO STATEMENTS

GO TO statements permit the user to alter the sequence in which statements are executed. A GO TO statement transfers execution to the statement whose label is specified in the GO TO statement. There are three types of GO TO statements:

1. Unconditional GO TO statements.
2. Computed GO TO statements.
3. Assigned GO TO statements.

Unconditional GO TO Statements

The unconditional GO TO statement causes an unconditional transfer. The form of the statement is

GO TO statement label

Example

GO TO 20

transfers to the statement labelled 20.

The GO TO statement transfers control unconditionally. However, a GO TO statement may be executed conditionally by using it in a logical IF statement.[2] For example, the statement

IF (X .EQ. Y) GO TO 30

causes transfer of control to the statement labelled 30 if the values of X and Y are equal. If X is not equal

1 - See Section 10, *Lines And Line Numbers*, Page 101.

2 - See *Logical IF Statements*, Page 33.

to Y, the GO TO statement is not executed; instead, execution continues with the next statement in the program.

The following statements illustrate using GO TO statements unconditionally and conditionally. These statements assume a one-dimensional integer array K containing at least N elements. The statements count the number of positive and negative elements among K(1), K(2), ..., K(N).

```
    I=1
    NPOS=0
    NNEG=0
10  IF (I .GT. N) GO TO 40
    IF (K(I) .GE. 0) GO TO 20
    NNEG=NNEG+1
    GO TO 30
20  NPOS=NPOS+1
30  I=I+1
    GO TO 10
40  DISPLAY "NO. OF POS. INTEGERS:", NPOS
    DISPLAY "NO. OF NEG. INTEGERS:", NNEG
```

Computed GO TO Statements

The computed GO TO allows transfer to one of several places in the program, depending on an integer value. The form of the statement is

GO TO (n_1,n_2,n_3, ...),v

where v is an integer expression and n_1,n_2, ... are statement labels. This statement transfers to the statement labelled n_1 if the value of v is 1; to n_2 if the value of v is 2, etc. For example,

```
GO TO (60,70,85),K
```

transfers to the statement labelled 60 if the value of K is 1, to the statement labelled 70 if the value of K is 2, and to the statement labelled 85 if the value of K is 3.

If the value of v is zero, negative, or greater than the number of statement labels in the computed GO TO, an error will be indicated.

ASSIGN And Assigned GO TO Statements

The ASSIGN statement allows the user to assign a statement label to a variable to be referred to in an assigned GO TO statement. It has the general form

ASSIGN n TO v

where n is the statement label of an executable statement and v is an integer variable name.

The assigned GO TO statement has the following form:

GO TO v, (n_1, n_2, ..., n_k)

where v is an integer variable which has had a statement label assigned to it by an ASSIGN statement, and the integers n_1, n_2, ..., n_k are a list of all possible statement labels which may be assigned to v.

Example

Consider the following partial program:

```
    INTEGER R
    ASSIGN 10 TO R
    .
    .
    .
    IF (A .LT. 100) ASSIGN 20 TO R
    .
    .
    .
    GO TO R, (10,20)
    .
    .
    .
10  X=A/2
    GO TO 50
    .
    .
    .
20  X=A**2+A/2
    GO TO 50
    .
    .
    .
50  ....
```

In this example, statement label 10 is assigned initially to R. Later, statement label 20 will be assigned to R if the value of A is less than 100. Thus, when the statement GO TO R is executed, control will be transferred to either the statement labelled 10 or the statement labelled 20, depending on the value of R.

IF STATEMENTS

Two kinds of IF statements are provided to make decisions, the arithmetic IF and the logical IF. With an arithmetic IF, decisions are based on an arithmetic quantity being less than zero, zero, or greater than

zero. With a logical IF, decisions are based on a logical quantity being true or false. In SUPER FORTRAN an .ELSE. clause may be appended to a logical IF statement so that the program will make one decision if a logical quantity is true and another if it is false.

Arithmetic IF Statements

The arithmetic IF statement has the general form

IF (expression) n_1,n_2,n_3

where the expression is any arithmetic expression and n_1, n_2, and n_3 are statement labels. This statement causes control to be transferred to the statement labelled n_1 if the value of the expression is negative, to n_2 if it is zero, and to n_3 if it is positive.

Examples

IF (K-N) 10, 10, 20

If the expression K-N has a value which is less than or equal to zero, the next statement executed is the statement labelled 10; if K is greater than N, the next executed is the statement labelled 20.

IF (Y-A(I)) 10, 15, 15

If the value of Y-A(I) is less than zero, the statement executed next is the statement labelled 10. If the value of Y-A(I) is zero or positive, the statement executed next is the statement labelled 15.

IF (X(I,J)N) 12, 5, 30**

*If the value of the expression X(I,J)**N is negative, the statement labelled 12 is executed next. If the value of the expression is zero, the statement labelled 5 is executed next. If the value of the expression is positive, the statement labelled 30 is executed next.*

Logical IF Statements

In SUPER FORTRAN, the logical IF statement may be used either with or without an .ELSE. clause.

Without an .ELSE. clause, the logical IF statement has the general form

IF (expression) statement

where the expression is any logical expression and the statement is any executable statement except a DO statement. When a logical IF statement is executed, FORTRAN first evaluates the logical expression. If the expression is true, the statement specified is executed; execution then continues with the next statement in the program (unless the statement in the IF caused a transfer of control). If the expression is false, the statement specified in the IF is not executed; execution continues with the next statement in the program.

Example

Execution of the statements

```
IF (X .GT. Y) A=B
C=A*5
```

causes the value of A to be replaced by B if X is greater than Y. If X is not greater than Y, the value of A will not be replaced by the value of B. In either case, C will be set to A*5. If the values of A, B, X, and Y before execution of these statements are

A=1.
B=2.
X=36.8
Y=1.

C will be set to 10. But if X were 0. instead of 36.8, C would be set to 5 since the value of A would not be changed by the IF statement.

The .ELSE. Clause

The logical IF statement may have an .ELSE. clause appended to it, taking the form

IF (expression) statement .ELSE. statement

This form causes the first statement to be executed if the logical expression is true; otherwise, the statement following the .ELSE. is executed. Unless control is transferred by one of these statements, execution continues with the next statement in the program. Like the statement preceding the .ELSE., the statement following the .ELSE. can be any executable statement except DO.

Examples

IF (X .EQ. 5) GO TO 10 .ELSE. GO TO 20

If X is equal to 5, control is transferred to statement label 10; otherwise, it is transferred to statement label 20.

IF (A .AND. B) X=Y .ELSE. X=SIN(Y)

If the logical expression A .AND. B is .TRUE., X is set equal to Y; if A .AND. B is .FALSE., X is set equal to SIN(Y).

DO STATEMENTS

The DO statement allows a sequence of statements to be executed repeatedly. For example, the statement

DO 30 J=1,5,1

causes all subsequent statements through the statement labelled 30 to be executed exactly five times. Repetition of these statements is controlled by varying the index variable J from an initial value of 1 to a final value of 5 in increments of 1, as follows: When the DO statement is executed, J is first set equal to 1. The statements following the DO through the statement labelled 30 are then executed. The value of J is then incremented by 1 and the statements are executed again. This process continues until incrementing J by 1 causes its value to be greater than 5; at this point, execution transfers to the statement following the statement labelled 30; the statements being repeated are not executed for J>5. Thus,

```
>1      DO 30 J=1,5,1 ↵
>2 30   DISPLAY J ↵
>3      END ↵
>RUN ↵
  1
  2
  3
  4
  5
(@3 )>
```

The general form of a DO statement is

DO *n v*=m_1,m_2,m_3

where

n is the statement label of the last step in the repetition.

v is the index variable whose value changes during the repetition. The variable must be nonsubscripted.

m_1 is the initial value for the index variable.

m_2 is the final value for the index variable; the index variable must exceed this value to terminate the repetition.

m_3 is the increment to the index variable for successive repetitions (m_3 may be omitted, in which case the increment is assumed to be 1).

Each of the indexing parameters m_1, m_2, and m_3 may be any arithmetic expression. Notice that the increment m_3 can be negative, as in Example 1 below. In such cases, the repetition terminates when the value of the index variable is less than m_2.

The group of statements from the DO statement through the statement labelled *n* is called a DO loop.

In SUPER FORTRAN, the index variable *v* can be of any arithmetic type (integer, real, double precision, or complex). Integer DO loops are the most efficient, however. In the case of a complex DO loop, the variable is incremented using complex mode arithmetic, but termination of the loop is determined by examining only the real parts of the indexing variable and final value.

Some examples of DO loops follow.

Example 1

In this example, the increment is negative. The value of I is printed beginning with 3, in steps of -1. The loop terminates when the value of I is less than 0.

```
>10      DO 30 I=3,0,-1 ↵
>20 30   DISPLAY I ↵
>30      END ↵
>RUN ↵
     3
     2
     1
     0
(@30 )>
```

Example 2

In this example, a real index variable X is used. The value of X is printed beginning with 1 in steps of .2. The loop terminates when the value of X exceeds 1.5.

```
>10      DO 100 X=1,1.5,.2 ↵
>20 100  DISPLAY X ↵
>30      END ↵
>RUN ↵
     1.
     1.2
     1.4
(@30 )>
```

Throughout a DO loop, the index variable may be used both as a subscript and as an ordinary integer variable. For example, the statements

```
    DO 100 I=3,20
100 A(I)=I
```

set A(3) to 3, A(4) to 4, and so on up to A(20)=20.

When a DO statement is executed, the values of m_1, m_2, and m_3 are permanently established as the initial value, final value, and increment for the loop. If the values of these parameters are changed inside the DO loop, the original values will be maintained. For example, the statements

```
   M=8
   DO 10 I=1,M
   M=3
   DISPLAY I
10 CONTINUE
```

will cause the numbers 1 through 8 to be printed on separate lines, rather than only the numbers 1, 2, and 3. Notice that the value of the variable M is actually changed by the statement M=3 even though the final value of the loop is not changed. Thus, adding the statement DISPLAY M to the above statements would cause a 3 to be printed after execution of the loop.

If the value of the index variable of a DO statement is changed inside the DO loop, the change will affect the number of times the loop is executed. For example, the loop

```
   DO 20 J=3,9
   J=J-1
   DISPLAY J
20 CONTINUE
```

will continue executing indefinitely since the value of J alternates between 3 and 2.

The CONTINUE statement used in the two preceding examples is a dummy statement used to mark the end of the loop. See *The CONTINUE Statement,* Page 36.

A DO loop must not end with a GO TO, PAUSE, STOP, RETURN, arithmetic IF, or another DO statement. The CONTINUE statement is a convenient way to terminate any DO loop.

Nested DO Loops

A DO loop may include other DO loops provided that the range of each inner loop is contained completely within the range of each outer loop. Such loops are called nested DO loops; the following skeleton examples illustrate their use:

Allowed

```
DO 10 I=1,N
 DO 20 J=1,M
  .
  .
  .
 20 CONTINUE
10 CONTINUE
```

Not Allowed

```
DO 10 I=1,N
 DO 20 J=1,M
  .
  .
  .
10 CONTINUE
20 CONTINUE
```

Allowed

```
DO 15 K=1,N
 .
 .
 .
 DO 5 A=1,7.5,.5
  .
  .
  .
 5 X=7+Y
  .
  .
  .
 DO 7 BAJ=2,3
  .
  .
  .
  DO 100 B=A*C,2
   .
   .
   .
  100 CONTINUE
   .
   .
   ...
 7 X=22
  .
  .
  .
15 CONTINUE
```

Nested DO loops may end with the same terminating statement. The following statements, which sum an array of two rows and three columns, illustrate this.

```
   TOTAL=0
   DO 15 I=1,2
   DO 15 J=1,3
   TOTAL=TOTAL+A(I,J)
15 CONTINUE
```

The elements of A are added row by row, thus:

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

Reversing the two DO statements above would cause the elements to be added column by column.

Transfer To And From A DO Loop

While transfer of control out of a DO loop is permitted, transfer of control into a loop is not. The following examples illustrate this:

Allowed	Not Allowed
DO 200 K=5,M,2	DO 10 I=1,N
C=K*P-Q	.
IF(C .GT. 1) GO TO 50	5 .
200 CONTINUE	10 CONTINUE
.	GO TO 5
.	
.	
50	

There is one exception to this rule: Transfer of control into a DO loop from an extended range of the DO loop is permitted. An extended range of a DO loop is a range of statements outside the DO loop to which a statement within the DO loop transfers. These statements could be included in the DO loop but are written outside, perhaps because they are to be executed again in the program. For example, consider the following statements:

```
    DO 100 K=1,10
    X=A*K
    Y=B*K↑2
    GO TO 50
60  C=X-Y
100 CONTINUE
    .
    .
    .
50  X=SQRT(2*X)
    Y=SQRT(3*Y)
    GO TO 60
    .
    .
    .
```

A DO loop

An extended range of the above loop.

Since the DO loop in the above example transfers to the statement labelled 50, we have an extended range of the DO loop. Thus, GO TO 60 is permitted since it returns control to the DO loop from an extended range. *NOTE: An extended range may include another DO loop.*

THE CONTINUE STATEMENT

The statement

CONTINUE

is used as a dummy statement which may be placed anywhere in the program. The statement causes no action. It is used primarily as a reference point for control statements, especially as the last statement of a DO loop.

A CONTINUE statement may be used to avoid ending a DO loop with a GO TO, PAUSE, STOP, RETURN, arithmetic IF, or another DO statement.

Example 1

The DO loop

```
    DO 25 I=1,N
25  SUM=SUM+X(I)
```

may also be written as

```
    DO 25 I=1,N
    SUM=SUM+X(I)
25  CONTINUE
```

Example 2

In the following statements, which count the number of non-zero elements of the array A, the CONTINUE statement is used to skip the count if an element is zero.

```
    NCOUNT=0
    DO 15 I=1,N
    IF (A(I))10,15,10
10  NCOUNT=NCOUNT+1
15  CONTINUE
```

Example 3

In this example, the CONTINUE statement is used as the last statement of the DO loop to avoid ending the loop with the statement GO TO 8.

```
    DO 40 I=1,20
8   IF (X(I) .GE. Y(I)) GO TO 40
    X(I)=X(I)+1
    Y(I)=Y(I)-2
    .
    .
    .
    GO TO 8
40  CONTINUE
```

IMPLIED DO LOOPS

SUPER FORTRAN allows two kinds of implied DO loops, the statement implied DO and the input/output list implied DO.

Statement Implied DO

A statement implied DO is a single statement of the form

(statement), $v=m_1,m_2,m_3$

It causes the repeated execution of the statement in parentheses, which must be an executable statement.

The parameters v, m_1, m_2, and m_3 are the index variable and its initial value, final value, and increment. These quantities follow the same rules as the analogous quantities in an explicit DO loop. In fact, the above form is equivalent to

```
DO n v=m1,m2,m3
n statement
```

where n is any statement label. Implied DO loops often enable the user to eliminate bothersome statement labels.

Example 1

```
(DISPLAY X), X=3.5,1,-.5
```

This implied DO loop is equivalent to

```
   DO 35 X=3.5,1,-.5
35 DISPLAY X
```

It causes the statement DISPLAY X to be executed repeatedly from the initial value of X to the final value, in increments of -.5.

Example 2

```
(SUM=SUM+X(J)), J=K*L, N
```

This statement sums the elements of the array X from X(K*L) to X(N), in increments of 1.

Implied DO loops may be nested. For example,

```
((TOTAL=TOTAL+A(I,J)),J=1,4),I=1,3
```

is equivalent to

```
    DO 100 I=1,3
    DO 100 J=1,4
100 TOTAL=TOTAL+A(I,J)
```

Notice that this example takes the usual form

```
(statement), I=1,3
```

where the statement is itself an implied DO.

The position of parentheses in a statement implied DO is important. For example,

```
(IF(A(I) .EQ. 0) A(I)=B(I)), I=1,3
```

causes the entire statement

```
IF(A(I) .EQ. 0) A(I)=B(I)
```

to be executed three times, varying the value of I from 1 to 3. But

```
IF(A(I) .EQ. 0) (A(I)=B(I)), I=1,3
```

first tests to see if A(I) is equal to zero, for some predefined value of I and then, if it is, executes the statement A(I)=B(I) three times, varying the value of I from 1 to 3. This statement is an IF statement containing a statement implied DO; the preceding statement is a statement implied DO containing an IF statement.

Input/Output List Implied Do

An implied DO may be included in the input/output list of the statements ACCEPT, DISPLAY, READ, WRITE, and DATA,[1] to assign values to all or part of an array.

Examples

```
ACCEPT A,B,(C(I),I=2,6,2)
```

This statement contains the implied DO (C(I),I=2,6,2). The statement accepts values for A, B, C(2), C(4), and C(6) from the terminal.

The statements

```
ACCEPT (A(I), I=1,10)
READ (0,200) (A(I), I=1,10)
```

read data values for A(1) through A(10).

The statements

```
DISPLAY (B(J), J=1,10,2)
WRITE (1,200) (B(J), J=1,10,2)
```

print the values of B(1), B(3), B(5), B(7), and B(9).

The statement

```
DATA (A(I),I=1,3)/4,5,6/
```

initializes A(1) to 4, A(2) to 5, and A(3) to 6.

MULTIPLE FORTRAN STATEMENTS

Tymshare's multiple FORTRAN statements greatly expand the power and brevity of the FORTRAN language by allowing the use of several statements anywhere a single statement is permitted.

Any number of executable statements may be joined to form a multiple statement by separating the statements with semicolons.

Example

```
A=B;C=D
```

Only the first statement in a multiple statement can be labelled. The partial statements in a multiple statement are executed from left to right.

A multiple statement may be given a CCS line number just like a simple statement, as in the following program:

```
LIST
 1          ACCEPT X,Y
 2          Z=X**2+Y**2;W=SQRT(Z)
 3          DISPLAY Z,W
 4          END
```

Multiple statements may be used in any statement where a single statement may be used, provided they are enclosed in parentheses.

1 - See *Input/Output Statements*, Page 45, and *DATA Statements*, Page 79, for complete descriptions of these statements.

Examples

```
IF (X .LE. 6) (ACCEPT Y; X=X+Y; DISPLAY X,Y)
```

If X is less than or equal to 6, the statements ACCEPT Y, X=X+Y, and DISPLAY X,Y are executed.

```
IF (X .LT. 0) (DISPLAY 'SINGULARITY'; GO ↵
TO 20) .ELSE. (Y=SQRT(X); DISPLAY ↵
'ANSWER =',Y
```

If X is less than zero, the statements DISPLAY 'SINGULARITY' and GO TO 20 will be executed. Otherwise, Y=SQRT(X) and DISPLAY 'ANSWER =',Y will be executed.

```
((Z(I)=X(I)+Y(I);ZSQ(I)=Z(I)**2)),I=1,N
```

This statement implied DO executes the multiple statement in parentheses for I=1 to N.

USER CONTROLLED INTERRUPTS

SUPER FORTRAN provides two statements for controlling program execution when an ALT MODE/ESCAPE is typed: the ON INTERRUPT and OFF INTERRUPT statement.

The ON INTERRUPT statement has the general form

ON INTERRUPT GO TO statement label

It causes the next ALT MODE/ESCAPE typed to abort whatever statement is being executed and transfer execution to the statement specified. For example,

```
ON INTERRUPT GO TO 100
```

causes a transfer to the statement labelled 100 when the next ALT MODE/ESCAPE is typed.

Only the first ALT MODE typed after execution of the ON INTERRUPT statement will cause a transfer. Any additional ALT MODE will simply interrupt the program in the usual manner, unless another ON INTERRUPT statement is executed.

The effect of the ON INTERRUPT statement may be nullified with the statement

```
OFF INTERRUPT
```

Example

```
> LIST ↵
   1          ON INTERRUPT GO TO 5
   2          DISPLAY "HIT ALT MODE
                TO TERMINATE PRINTOUT"
   3          DO 4 X=1,1000
   5     4    DISPLAY X**2
   7     5    OFF INTERRUPT
   8          DISPLAY "LOOP
                TERMINATED"
   9          END
> RUN ↵

HIT ALT MODE TO TERMINATE PRINTOUT
   1
   4
   9
   16 ⊕            The user types an ALT MODE.

LOOP TERMINATED    Control is transferred to
                   the statement labelled 5.
(@9 )>
```

THE PAUSE STATEMENT

This statement may take any of the following forms:

```
PAUSE
PAUSE "text"
PAUSE 'text'
PAUSE number
```

A PAUSE statement may be placed anywhere in a program. When this statement is executed, program execution is interrupted. The specified text or number, if any, is printed, and the user is returned to the CCS command mode. Direct statements and CCS commands may then be executed; program execution may be resumed at the point of interruption by typing the CCS command CONTINUE.[1]

Example

```
> FAST ↵
1 DISPLAY "BEGIN EXECUTION"
2 PAUSE "NOW, CONTINUE"
3 DISPLAY "EXECUTION CONTINUES"
4 END
> RUN ↵
BEGIN EXECUTION
NOW, CONTINUE
2 >CONTINUE ↵
EXECUTION CONTINUES
(@4 )>
```

NOTE: If a command file has been opened when a PAUSE is executed, the PAUSE causes commands to be taken from that file. See **Command Files**, *Page 99, for details.*

1 - Do not confuse the SUPER FORTRAN CONTINUE statement with the CCS CONTINUE command. See *Continuing Program Execution: CONTINUE*, Page 118, for a description of CCS CONTINUE.

THE STOP STATEMENT

This statement may take any of the following forms:

```
STOP
STOP "text"
STOP 'text'
STOP number
```

When a STOP statement is encountered, program execution is terminated. The specified text or number, if any, is printed, and the user is returned to the CCS command mode. The user may then enter direct statements, but execution may not be resumed with the CCS CONTINUE command.

Example

Execution of the line

```
10 STOP "END OF JOB"
```

causes the following to be printed at the terminal:

```
END OF JOB
(@10 )>
```

THE QUIT STATEMENT

A QUIT statement may be included in a program to return to the EXECUTIVE directly from a program, without first returning to the CCS command mode. This statement has the forms

```
QUIT
QUIT "text"
QUIT 'text'
QUIT number
```

Execution of the statement causes the specified text or number, if any, to be printed at the terminal. Immediately after this, the user is returned to the EXECUTIVE.

Example

Consider the following partial program:

```
   X=-3.5
20 Y=Y-X**2/2.
   .
   .
   .
   X=X+1
   .
   .
   .
   IF(X.GT.0) QUIT "DONE"
   GO TO 20
   END
```

Execution of these statements would yield the following:

```
>RUN
DONE

-
```

When X is greater than zero, DONE is printed and the user is returned to the EXECUTIVE.

The QUIT statement can be especially useful when executing command files that run programs in several Tymshare languages.

THE END STATEMENT

The statement

```
END
```

is required at the end of every SUPER FORTRAN program and subprogram. This statement is nonexecutable; it may never be given a statement label. Thus, a labelled STOP or CONTINUE statement should be used when transfer of control to the end of a program is desired as in the following:

```
   .
   .
   .
   IF (EPS .LE. 1E-7) GO TO 40
   .
   .
   .
40 STOP "ITERATION COMPLETE"
   END
```

SECTION 5
STRINGS

SUPER FORTRAN includes extensive string processing capabilities, including string constants, variables, and functions. String constants, such as 'P4-A95', were discussed in *Constants,* Page 19. In this section we discuss string variables and string manipulation in general.

STRING VARIABLES: THE STRING DECLARATION STATEMENT

Instead of assigning a numeric or logical value to a variable, the user may set a variable equal to a string of characters. String values may be assigned to variables in replacement statements, DATA statements, or input statements.[1]

Any valid variable name may be used as a string variable name. Both scalar string variables and string arrays are allowed.

String variables must be declared explicitly using the STRING declaration statement. This statement must specify not only the names of the string variables being declared, but also the maximum allowable number of characters in the string. It takes the form

STRING variable name(number of characters)

The variable name is followed by the maximum number of characters in parentheses. For example,

```
STRING S(72)
```

creates a simple string variable S which may contain up to 72 characters. Twenty-four words of storage will be permanently allocated to S with this declaration.

String arrays must be dimensioned in STRING declaration statements, and may not be dimensioned in DIMENSION or COMMON statements.[2] The form of the STRING array declaration is:

STRING array name (dimension$_1$,dimension$_2$,...)
(number of characters)

For example, the statement

```
STRING AB(20,10)(50)
```

declares AB to be a 200 element (20 by 10) string array, each element of which can accommodate 50 characters.

The STRING declaration statement may have a list of variables and arrays to be declared, separated by commas. For example,

```
STRING WORD(3),A1(30)(17),DIM(10,10)(8)
```

creates a simple string variable, WORD; a 30 element array, A1; and a 100 element array, DIM.

The STRING declaration statement, like all type declaration statements, is nonexecutable.[3]

STRING REPLACEMENT STATEMENTS

String variables and constants may be used in replacement statements in the usual way. For example,

```
STRING S(10)
S="CORNCOB"
```

sets the value of the string variable S to CORNCOB.

STRING COMPARISON

String values may be compared using any of the relational operators .EQ., .NE., .LT., .LE., .GT., and .GE. . Logical expressions formed using string values may be used in control statements in the usual manner. For example, the statements

```
STRING DECIDE(3)
.
.
.
IF (DECIDE .EQ. 'YES') GO TO 100
```

cause control to be transferred to the statement labelled 100 only if DECIDE has the string value YES.

Strings are compared internally using the ASCII character codes.[4] For example, the expression

```
"JUNE" .GT. "JULY"
```

is true. The first two characters of the strings match, but since N has a greater internal code than L, JUNE is greater than JULY.

If two strings being compared are of different lengths, the shorter string and the same number of characters from the longer string will be compared. If they match, the shorter string is taken to be the lesser of the two. Thus,

```
"SUN" .LT. "SUNDAY"
```

is true.

STRING CONCATENATION

The binary string operator + is used for string concatenation (joining strings together).

Example

The statements

```
STRING S1(3),S2(6)
S1='SUN'
S2=S1+'DAY'
```

set S2 equal to SUNDAY (S1+'DAY'='SUN'+'DAY').

1 - See *DATA Statements*, Page 79, and *Input/Output Of Strings*, Page 42.
2 - String variables may be placed in COMMON, however. See Page 84.
3 - Declaration statements are discussed in general in *Declaration Statements*, Page 79.
4 - See *Appendix B, Internal Representation of ASCII Codes*, Page 159.

STRING FUNCTIONS

The SUPER FORTRAN function library contains many functions for string manipulation. These are given in the table below. In the table,

S and T	are string arguments
I and N	are integer arguments
X	is an argument of any numeric type

SUMMARY OF STRING FUNCTIONS

Function Call	Value Returned (With Type)	Examples	
		Call	Value
LENGTH(S)	Number of characters in S. (integer)	LENGTH("CAT")	3
LEFT(S,N)	First N characters of S. (string)	LEFT('ABCD',2)	AB
RIGHT(S,N)	Last N characters of S. (string)	RIGHT('ABCD',3)	BCD
SUBSTR(S,I)	All of S from Ith character on. (string)	SUBSTR('ABCD',2)	BCD
SUBSTR3(S,I,N)	N characters of S starting from the Ith character. (string)	SUBSTR3('ABCD',2,2)	BC
INDEX(S,T)	Character position of first occurrence of T within S, if T is a substring of S; otherwise, 0. (integer)	INDEX('ABCD','BC') INDEX('ABAC','A') INDEX('ABCD','F')	2 1 0
INDEX3(S,T,I)	Character position of first occurrence of T within S beyond or including the Ith character; returns value of 0 if T does not occur in S between the Ith character and end of S. (integer)	INDEX3('ABAC','A',1) INDEX3('ABAC','A',2) INDEX3('ABAC','A',3) INDEX3('ABAC','A',4)	1 3 3 0
STR(X)	Arithmetic value of X is converted to a string value under free form output conventions. (string)	STR(123.1)	123.1 (123.1 preceded by a space; a 6 character string)

SUMMARY OF STRING FUNCTIONS (continued)

Function Call	Value Returned (With Type)	Examples	
		Call	Value
VAL(S)	The string S (consisting of numeric characters) is converted to a number. (numeric; depends on S)	VAL("133.1") VAL("133") VAL("1E4")	133.1 133 10000.
CHAR(N)	N must be an integer between 0 and 255 inclusive. CHAR returns a one-character string whose internal code in decimal is N. (string)	CHAR(5) CHAR(33) CHAR(35)	% A C
ASC(S)	Returns the integer value of the internal representation of the first three characters in S. If S has fewer than 3 characters, the internal representation is right justified. Thus, when S consists of a single character, ASC returns the internal code of that character. (integer)	ASC("C") ASC("CA") ASC("CAT") ASC("CATSUP")	35 8993 2302260 2302260

Note also that the mathematical functions MIN and MAX may be used with string arguments as well as with numeric arguments. For example,

MAX ("A","B","C12")

returns the value C12, and

MIN ('CAT','CATSUP')

returns the value CAT.

INPUT/OUTPUT OF STRINGS

String input and output can be either formatted or unformatted. In fact, all the I/O statements discussed in *Input/Output Statements*, Page 45, may be used for string I/O. Refer to this section for detailed information on these statements. Here, we discuss some special features of string input and output.

The usual forms of ACCEPT and DISPLAY may be used for free form terminal input/output of strings.

When typing strings in response to ACCEPT, the strings need not be surrounded with quotes unless they contain commas, Carriage Returns, control characters, or leading blanks.

Example

```
>LIST ↵
   1                    STRING S(5),T(5),U(5)
   2                    ACCEPT S,T,U
   3                    DISPLAY S
   4                    DISPLAY T,U
   5                    END
>RUN ↵
ABCDE,"  ABC",'AB,CD' ↵
ABCDE
   ABC      AB,CD
(@5 )>
```

The value for T must be enclosed in quotes since it contains two leading blanks; the value for U, since it contains a comma.

If a string typed in response to ACCEPT contains more characters than the declared length of the string, the right-hand characters are lost.

In addition, the free format READ and WRITE statements may be used for string I/O. These statements, discussed in *Free Format READ And WRITE,* Page 68, may be used for free format input from or output to a file. However, caution is advised when using unformatted output of strings to files. If the files are to be read later using unformatted input, the strings are most safely surrounded by quotes and/or separated by commas.

A special format specification, S, is available for string I/O. In addition, the A and G format specifications allow reading and writing of strings. All these features are discussed in *Formatted Input and Output,* Page 47.

SECTION 6

INPUT AND OUTPUT STATEMENTS

NOTE: To clarify the format of input and output values in this section, all tables demonstrating sample values are aligned so that the value (including blanks) begins at the appropriate border of the table and all significant blanks are indicated by the symbol ■.

Input and output statements are, as their name implies, the statements used to transfer data to and from the computer. Input statements are used to supply a program with the data needed to perform its computations, and output statements are used to return the results of these computations to the user.

Input to a program can be either from the terminal or from a disk file; output can be either printed on the terminal or stored on a disk file.

Terminal input and output is always symbolic; that is, the data is read or written in its usual character representation. Data may be read or written at the terminal in either a formatted or unformatted form.

Data stored on a disk file may be either symbolic or binary. (Binary form is the form in which information is stored internally, as opposed to the usual character representation on a symbolic file or the terminal.) Symbolic file input and output may be either formatted or unformatted. However, binary files must always be read or written in unformatted form.

Both random and sequential files are available. Either may be of symbolic or binary type.

In addition, special input statements are available which allow programmable error and end-of-file conditions.

In this section we first discuss unformatted terminal input and output. Formatted input and output is then discussed, followed by the rules for symbolic and binary disk file I/O with sequential and random files. Last, we discuss programmable error and end-of-file conditions.

FREE FORMAT TERMINAL INPUT AND OUTPUT

The simplest way to enter and print data on the terminal is to use the free format input/output statements ACCEPT and DISPLAY which are unique to the Tymshare system. These statements may be used for input and output of any type of data values, whether numeric, logical, or string.

THE ACCEPT STATEMENT

This statement is used for free format terminal input. It takes the general form

ACCEPT input list

Example

ACCEPT A,B,(C(I),I=1,9)

This statement causes values for the variables A, B, and C(1) through C(9) to be requested from the terminal. Notice that all entries in the input list must be separated by commas.

Input List

The input list in the ACCEPT statement may include any legal variable or array name. It may also include an implied DO loop, used to enter all or part of an array. In the above example, the implied DO loop (C(I),I=1,9) will cause values for C(1) through C(9) to be requested.

An entire array may be requested by its name alone in the input list. In this case, the exact number of elements specified when the array was dimensioned[1] will be requested. For example, if the array B has been dimensioned with

DIMENSION B(15)

then the statement

ACCEPT B

will request values for 15 elements of B, B(1) through B(15).

Literal text may be included in the input list; see *Literal Text In The I/O List*, Page 47.

1 - See *The Dimension Declaration*, Page 82, for details on array dimensioning.

Input In Response To ACCEPT

When the ACCEPT statement is executed, the system rings a bell and waits for the user to type values for the variables and/or arrays specified. The user must then type one value for each variable and/or array element specified in the input list. A Carriage Return, a comma, a Control D, or a space must terminate each value typed. The value supplied will assume the type of the variable or array in which it is stored. A bell will ring at the beginning of each line of values typed in response to a single ACCEPT statement.

Example

```
>1 DIMENSION A(5) ↵
>2 ACCEPT A,I ↵
>3 END ↵
>RUN ↵
12.5,13.4,16.35 ↵
19.21,11.1116,3 ↵

(@3 )>
```

In this example, the user types RUN followed by a Carriage Return and waits for the bell to ring. Then he types his data values. Since the array A was dimensioned as A(5), exactly 5 values must be entered for A. The user types these values, terminating the first two values typed with a comma, the value for A(3) with a Carriage Return, and the values for A(4) and A(5) with a comma. He then must enter the value for I, which he follows with a Carriage Return. Notice that any of the valid terminators (Carriage Return, space, comma, D^C) could be used after any of the values typed.

The ACCEPT statement ignores leading blanks, commas, Carriage Returns, and control characters (except for V^C).[1] In the example above the user could have entered the value for I by typing

```
■■3 ↵
```

instead of

```
3 ↵
```

since leading blanks are ignored. *NOTE: During free form input of strings, terminating characters and leading characters normally ignored will be accepted as part of the string value if the string is delimited by single or double quotes. This was discussed in* **Input/ Output Of Strings**, *Page 42.*

When logical values are entered in response to ACCEPT, any string of characters may be entered. The characters entered are scanned from left to right. If a T is found, the value assigned is true; if an F is found or if neither a T nor an F is found, the value assigned is false. See the example under *The DISPLAY Statement*, on the next page.

THE DISPLAY STATEMENT

The DISPLAY statement is used to print data at the terminal. It has the general form

DISPLAY output list

Example

```
DISPLAY A,B,(C(I),I=1,9),5*6+5,SQRT(Y)+Z
```

This statement causes the values of the variables A, B, and C(1) through C(9), and the values of the expressions 5*6+5 and SQRT(Y)+Z to be printed on the terminal.

Output List

All entries in the output list must be separated by commas. Constants, variables, arrays, expressions, and implied DO loops may be included. Execution of the DISPLAY statement causes the values stored in the variables and/or arrays specified to be printed on the terminal. Any expressions included in the list will be evaluated and their values returned. Real numbers are rounded to 10 significant digits; double precision numbers to 16 significant digits. An array can be referred to by only its name if the entire array is being written. For example, if the array ALPHA has been dimensioned using

```
DIMENSION ALPHA(10)
```

the statement

```
DISPLAY ALPHA
```

will cause the values of ALPHA(1) through ALPHA (10) to be printed on the terminal.

Values printed by separate DISPLAY statements are printed on separate lines. Thus,

```
>1 X=3 ↵
>2 Y=4 ↵
>3 DISPLAY X,Y ↵
>4 DISPLAY SQRT(X**2+Y**2) ↵
>5 END ↵
>RUN ↵
■3■■■■■4
■5

(@5 )>
```

1 - See *Editing Control Characters*, Page 126, for an explanation of Control V.

When more than one value is specified in a single DISPLAY statement (as in line 3 in the above example), the values are printed separated by spaces according to the following rule:

The printing area is divided into zones of three spaces each. After printing a value, SUPER FORTRAN skips two spaces and then, if not positioned at the beginning of a zone, moves to the beginning of the next zone. A leading space is always printed before a positive number.

Logical values are printed by DISPLAY as either T or F.

Example

```
> FAST ↵
1 LOGICAL A(5)
2 ACCEPT A
3 DISPLAY A
4 END
> RUN ↵
TURNIP,OFF,123,FAST,TURF ↵
T       F       F       F       T

(@4 ) >
```

LITERAL TEXT IN THE I/O LIST

The ACCEPT and DISPLAY statements may contain literal text in the input/output list, either alone or in conjunction with other values. The desired text must be enclosed in either single or double quotes. When an ACCEPT or DISPLAY statement containing literal text is executed, the text is printed on the terminal.

Examples

```
DISPLAY 'THIS PROGRAM COMPUTES
                INTEREST ON A LOAN'
```
Prints on the terminal the text enclosed in quotes.

```
ACCEPT "THE ORIGINAL VALUE IS", C
```
Prints THE ORIGINAL VALUE IS and then waits for the user to enter the value of C.

```
DISPLAY 'MAXIMUM CAPACITY =',T, "TONS"
```
If T is 5026.3, this statement prints

MAXIMUM CAPACITY = 5026.3 TONS

FORMATTED INPUT AND OUTPUT

Data also may be read and printed in formatted form using the ASA Standard FORTRAN IV statements READ, WRITE, and FORMAT. The READ and WRITE statements are executable input/output statements. The FORMAT statement is a nonexecutable reference statement which supplies certain information about the size and mode of the data values being read or written. The READ, WRITE, and FORMAT statements are used both for terminal input and output and for symbolic disk file input and output, both sequential and random. The formatting rules presented here apply equally well to terminal input/output and symbolic disk file input/output, except for fixed record length random files. In this case, the same field specifications are used, but certain special formatting rules must be followed.[1] Aside from these exceptions, the only difference between formatted terminal input/output and formatted symbolic disk file input/output is that symbolic disk files must be opened before use with an OPEN statement and closed after use with a CLOSE statement.[2]

In addition to the ASA standard FORMAT statement, dynamic formatting is allowed.[3] Both real arrays and string variables may be used to store formats to be used in reading or writing data.

THE FORMATTED READ AND WRITE STATEMENTS

The following forms of the READ and WRITE statements are used for formatted input and output:

READ (file number, format number) input list

WRITE (file number, format number) output list

The **file number** indicates the file from which the data is to be read or on which it is to be written. If a disk file is being used, the file number specified is the one used in the OPEN statement. The file number in this case may be any integer greater than 1. If the terminal is being used, the file number will be 0 if input is being performed and 1 if output is being performed.

1 - See *Special Rules For Fixed Record Length File I/O*, Page 72.
2 - See *Disk File Input And Output*, Page 68.
3 - See *Dynamic Formats*, Page 67.

The **format number** is the statement label of the FORMAT statement used to read or write the values of the variables and/or arrays in the list.

The **input list** may contain variable and array names and implied DO loops. In addition to these, the **output list** may contain expressions, which will be evaluated and their results printed upon execution of the WRITE. In either list, an array can be referred to by its name alone if the entire array is being read or written.

Examples

READ (0,7)A,I,(C(I),I=1,10)

Reads values for A, I, and C(1) through C(10) from the terminal according to the format specified in the FORMAT statement labelled 7.

WRITE (1,90)A,(B(J),J=200,300,5),"5TG",
SQRT(79)

Prints on the terminal the values of A, B(200), B(205), ..., B(300), the string constant "5TG", and the square root of 79. These values are printed in the format specified in FORMAT statement 90.

READ and WRITE may be used for input and output of any type variable (numeric, logical, or string). Information about the type of the data being read is contained in the FORMAT statement, where different specifications indicate different data types.

THE FORMAT STATEMENT

The FORMAT statement is a nonexecutable statement which supplies certain information about the size and type of the data values to be read or written and the form in which they are to be read or written. It takes the form

statement label **FORMAT** (**field spec.$_1$, field spec.$_2$, ...**)

Example

100 FORMAT (I5,2X,F7.2,3(I4,2F9.3),A3)

The field specifications supply the information about how data values are to be read or written. As an example, consider the specification

I5

which specifies that an integer value 5 characters long, such as 12345, be read or written.

The information supplied by a field specification is used somewhat differently during input than during output.

During input, the size specified is crucial since it indicates the exact number of characters read. For example, the statements

READ (0,100) INT

100 FORMAT (I5)

read **exactly** 5 characters from the terminal. If the user enters

12345 ↵

the value assigned to the variable INT is 12345. But if he enters

+12345 ↵

the value assigned to INT is 1234 since the + is counted as one of the characters read.

If the variable type specified in a field specification is different from the type of the variable into which the value is being read, the type of the variable overrides the type specified in the format. For example, if the statements

READ (5,200) X

200 FORMAT (I5)

are used to read the 5 digits

16934

from a file, the value assigned to X is the real value 16934.0, and not the value 16934, an integer.

During output, the type of the field specified always determines the type of the value printed. For example, the field specification I5 prints

11213.4 as **11213**

The size specified indicates the number of character positions used to print the value. For example, the field I5 prints 111 as

111 preceded by two spaces,

since five character positions are specified.

More detailed information on the differences between input and output is given in the descriptions of each field specification in the rest of this section.

SUPER FORTRAN includes numeric, nonnumeric, and utility field specifications. Numeric field specifications are used for reading and writing numeric values. Nonnumeric specifications are used for reading and writing alphanumeric, string, and logical values. The utility specifications are used for certain special purposes such as printing blanks and Carriage Returns, including literal text in a format, and scaling numeric I/O. All these field specifications are summarized below. Following the summary, detailed descriptions of each field specification are given.

FIELD SPECIFICATION SUMMARY

The tables below summarize all the Tymshare SUPER FORTRAN field specifications. In the tables, the following symbols are used:

w field width (the total number of characters read or written)

d number of decimal digits

i number of integer digits

s a string of characters

f a signed integer indicating a power of 10

n a positive integer

The following specifications are used to read and write numeric values only.

NUMERIC SPECIFICATIONS					
			Examples (Output)		
Specification	General Form	Type Of Value Read Or Written	Field	Prints	As
I	Iw	Integer	I4 I4	123 -123	■123 -123
F	Fw.d	Real (Decimal form)	F5.1 F6.1 F6.1	123.4 123.4 -123.4	123.4 ■123.4 -123.4
E	Ew.d	Real (E-exponential form)	E7.1 E7.1 E8.2	123.4 -123.4 123.4	■.1E+03 -.1E+03 ■.12E+03
D	Dw.d	Double Precision (D-exponential form)	D7.1 D7.1	123.4 -123.4	■.1D+03 -.1D+03

NOTE: The fields F, E, and D will accept input data in any form: integer, decimal, E-exponential or D-exponential. The I format field, however, will read only integer values. *See* **Input To Numeric Field Specifications**, *Page 54.*

The G, or generalized, field specification may be used for input or output of any variable type. This specification has the two forms summarized in the table below:

THE G (GENERALIZED) SPECIFICATION				
	Example (Output)			
General Form	Field	Prints	As	Remarks
Gw	G4	123 123.4	■123 ■123	Gw is equivalent to Iw for numeric values.
	G4	"ABC"	ABC■	Equivalent to Sw for string values.
	G3	.FALSE.	■■F	Equivalent to Lw for logical values.
Gw.d	G10.4	.01	■.1000E-01	Equivalent to Ew.d when absolute value printed is $\leq .1$ or $\geq 10^d$.
	G10.4	10. 100.	■10.00■■■■ ■100.0■■■■	When value printed lies between .1 and 10^d, exactly d digits are printed.

NOTE: Gw and Gw.d are exactly equivalent for string and logical values; the d specified is ignored.

The following specifications are used to read and write nonnumeric values only.

NONNUMERIC SPECIFICATIONS					
			Examples (Output)		
Specification	General Form	Function	Field	Prints	As
L	Lw	Reads or prints logical variables	L4	.TRUE.	■■■T
A	Aw	Reads or prints w characters. Allowable values for w depend on type of variable being read or written. (Integer, real, double, complex or string)	A6 A3 A9	FLS674 FLS674 FLS674	FLS674 FLS ■■■FLS674
S	Sw	Reads or prints declared string variables	S6 S3 S9	"FLS674" "FLS674" "FLS674"	FLS674 FLS FLS674■■■
	S (without field width)	Reads or prints declared string of any length as one record on specified file	S S	"ABC12" "$100.05"	ABC12↵ $100.05↵ (if file is not a fixed record length random file)

UTILITY SPECIFICATIONS				
			Examples (Output)	
Specification	General Forms	Function	Field	Action
H	wHs alternate forms: 's' "s" s	Allows inclusion of literal text in a format	3HEND 'END' "END" END	These four fields all print the text END on the output medium.
P	fP	Sets scaling to f^{th} power of 10 (affects only I/O with G,E, D, & F fields)	3P -1P	Sets scaling to 10^3 Sets scaling to 10^{-1}

UTILITY SPECIFICATIONS (Continued)				
			Examples (Output)	
Specification	General Forms	Function	Field	Action
T	Tw	Tabs	T20	Tabs to print position 20 (if output file is not random).
X	wX	Spacing	5X	Prints 5 spaces
/	/	Generates end of record action	/	Prints a Carriage Return (if output medium is not a fixed record length random file).
&	&	Suppresses normal end of record action caused by end of format	&	Suppresses Carriage Return generated at end of format (if output medium is not a fixed record length random file).
( )	$n\left(\begin{matrix}\text{field}\\ \text{specs}\end{matrix}\right)$	Allows field replication	2(I5,F6.2)	Is equivalent to (I5,F6.2,I5,F6.2).

NUMERIC FIELD SPECIFICATIONS

I FIELD SPECIFICATION

The I (integer) field specification has the general form

Iw

where the field width w specifies the number of characters to be read or written, including the minus sign if a negative number is to be read or written.

Input

During input, the field specification Iw causes w characters to be read. Only integers may be entered with the I field; attempting to read numbers in decimal or exponential form will cause an error message.

Examples

Using the field I3,

The Data	Is Read As
123	123
1234	123
-12	-12
-123	-12
■■2	2
■2■	20

Attempting to read the number 7.5 with an I3 field would result in an error message, even if it were being read into a real variable.

Since I3 reads exactly three characters, numbers longer than three characters cannot be read with this

specification. Thus, in the above examples, the numbers 1234 and -123 are read as 123 and -12. Notice that the fourth character is simply not read, and is thus available to be entered with another specification. For example, the statements

```
      READ (0,100) J,K
100 FORMAT (I3,I1)
```

read 1234 as two values: First, 123 is read and assigned to J using the field I3; then the 4 is read and assigned to K using the field I1.

Notice that blanks are converted to zeroes on input. Thus, 2 preceded by two blanks is read as 2 (002) using I3, but 2 preceded by a blank and followed by a blank is read as 20 (020).

Output

Any number printed with the I specification is printed as an integer; any decimal digits will be rounded. If the field width w is larger than the width of the number to be printed, leading blanks are supplied. If it is smaller, an error message is given.

Examples

Using the field I4,

The Number	Is Printed As
5678	5678
-567	-567
567	■567
-56	■-56
2584.6	2585
1.7	■■■2

However, attempting to use I4 to write 56789, or -5678, results in an error message since these numbers have a width of 5 characters.

F FIELD SPECIFICATION

The F (external fixed point, or decimal) field specification has the general form

Fw.d

where **w** specifies the field width and **d** indicates the number of digits to the right of the decimal point. The field width must include positions for the sign of the number, if negative, and for the decimal point.

Input

The field specification Fw.d reads w characters from the input medium.

If the input data does not include a decimal point, d digits to the right of the decimal point are assumed. For example, using F5.2,

The Data	Is Read As
12345	123.45
-1234	-12.34

On the other hand, if a decimal point is present, the d part of the specification is ignored. The total number of characters read is still determined by w, however. For example, F5.2 specifies that

123.4 be read as **123.4**

Like the I field specification, Fw.d never reads more than w characters. For example, the input data 156.4 requires a total field width of 5. If the specification F4.1 is used to read this data, only the characters 156. would be entered.

Blank are converted to zeroes on input; thus, using F5.2

The Data	Is Read As
■■■25	.25 (000.25)
■■25■	2.5 (002.50)

The F field specification also reads input data in E or D exponential form. In this case, it works just like the E and D specifications. For example, F10.4 reads -23345E+06 as -2.3345E+06.

Output

During output, the specification Fw.d causes the number to be printed in a field of width w with d digits to the right of the decimal point. The field width w must include one position for the decimal point and one position for the sign of the number if it is negative, as well as one position for each digit to be printed. If the field width w is larger than required to print the number, leading blanks will be supplied. If the number of decimal places specified by d is larger than required, trailing zeroes are printed. For example, using the specification F6.2,

The Number	Is Printed As
346.78	346.78
46.78	■46.78
25.	■25.00
25.725	■25.73
-12.346	-12.35

If insufficient total field width is specified, an error message will be given. For example, the field specification F6.2 cannot be used to print either

-346.78

or

1346.78

since each of these numbers requires a total field width of at least seven. Attempting to use F6.2 to print these numbers causes an error message. Notice, however, that if insufficient decimal places are specified by d, the decimal part of the number is rounded to the number of places specified. Thus, F6.1 could be used to print either of the above numbers; it prints

-346.78 as **-346.8**

1346.78 as **1346.8**

E FIELD SPECIFICATION

The E (exponential) field specification has the form

Ew.d

where **w** is the entire width of the field and **d** is the number of digits to the right of the decimal point.

Input

During input, the specification Ew.d reads w characters from the input medium. If the input data has no decimal point, d decimal digits are assigned to the mantissa. If the input data does have a decimal point, the d specified is ignored.

Examples

Using E10.2,

The Data	Is Read As
123456E+07	1234.56E+07
1.2345E+07	1.2345E+07
■1.234E+07	1.234E+07
-12345E+07	-123.45E+07
1234567E07	12345.67E+07

Notice that in the last example above, the sign of the exponent is not specified. This is unnecessary for positive exponents. A leading zero in the exponent may also be eliminated; for example, E10.2 reads

1234567E-7 as **12345.67E-07**

12345678E7 as **123456.78E+07**

Ew.d may be used to read input data in non-exponential form. In this case, it works just like Fw.d. Note also that it accepts input data in either E or D exponential form.

Output

A number printed using the Ew.d specification always has the form

■.xE±ee if the number is positive (note the leading space!)

or

-.xE±ee if the number is negative

where x consists of d decimal digits, and ee is a two digit exponent. A minimum of one decimal place is required; thus, a minimum total field width of 7 must be specified (E7.1). One print position is needed for the sign of the mantissa (or the leading space printed if the number is positive), one for the decimal point of the mantissa, one for the minimum one place after the decimal point, and four for the exponent (E, sign, and two digits).

In general, the total field width w must be greater than or equal to d+6. For example, E10.4 is a valid specification, but E10.5 is not. If w is insufficient to print the number (that is, if w is less than d+6) an error message will be given.

If the specified width is larger than needed to print the number, leading blanks are supplied. If the d specified is larger than necessary, trailing zeroes are printed. If too few decimal places are specified by d, output is rounded to the number of decimal places specified.

Examples

Specification	Prints	As
E9.3	479.	■.479E+03
	-479.	-.479E+03
	.005	■.500E-02
E10.3	479.	■■.479E+03
	-479.	■-.479E+03
E10.4	2145.65	■.2146E+04
	-6.5	-.6500E+01

D FIELD SPECIFICATION

The D (double precision) specification works exactly like the E specification except that, during output, a D is printed between the mantissa and the exponent instead of an E. Numbers read with this specification are not necessarily read as double pre-

cision values; they will be read as double precision values only if they are being read into a double precision variable.

THE G OR GENERALIZED FIELD SPECIFICATION

The G field specification can be used for input or output of any type of variable. It has two forms:

Gw and

Gw.d

In both forms, **w** is the field width; the **d** in the second form specifies information about the number of decimal places in arithmetic output or input.

Gw

During either input or output, the form Gw is equivalent to

Iw if arithmetic values are being read or written;

Lw if logical values are being read or written;

Sw if string values are being read or written.

Refer to the sections on the I, L, and S specifications for details.

Examples (Output)

Using G3,

The Value	Of Type	Is Printed As
12	integer	■12
-12.4	real	-12
AB6	string	AB6
AB	string	AB■
FALSE	logical	■■F
TRUE	logical	■■T

Gw.d

The form Gw.d functions exactly like Gw for input or output of string and logical values. The d specified is ignored. For example, the fields G3, G3.1, and G3.2 all print the string AB as AB followed by a space.

During arithmetic input, Gw.d is equivalent to Ew.d.

During output of arithmetic values, the form Gw.d prints the number n in decimal or exponential form depending on its size, as follows:

Let n be the number being printed. If $.1 < ABS(n) < 10^d$, the number n will be printed in decimal form in a field of width w. Exactly d digits will be printed, preceded by a space or minus sign (depending on the sign of the number) and followed by four spaces. The decimal point is printed in the appropriate position in relation to the d digits printed.

If the number being printed does not fall in the above range, it is printed just as if the specification Ew.d were being used.

Example

```
>LIST
    1           WRITE (1,100) (10.0**J,J=-2,5)
    2      100  FORMAT (G10.4)
    3           END
>RUN
 .1000E-01   ┐
 .1000E+00   ┘ 10^J ≤ .1
 1.000       ┐
 10.00       │
 100.0       ┘ .1 < 10^J < 10^4
 1000.       ┐
 .1000E+05   ┘ 10^4 ≤ 10^J
 .1000E+06
(@3 )>
```

If the field width is wider than needed to print the number, leading blanks are printed. If the specified d is too small, output is rounded. If d is too large, trailing zeroes are supplied.

The Gw.d specification is especially useful for printing values which are expected to lie within a certain range. Since it aligns the mantissas of the numbers printed, numbers outside the desired range can be detected easily.

INPUT TO NUMERIC FIELD SPECIFICATIONS

We shall now summarize some general rules for formatted input of numeric data.

As the reader may have noticed, the field specifications Fw.d, Ew.d, Dw.d, and Gw.d are all equivalent during numeric input. They all accept data in integer, decimal, E exponential, or D exponential form in exactly the same way. The rules in this section apply to numeric input using any of these specifications as well as to numeric input using the I and Gw specifications.

Input Data Fields

The term input data field refers to input characters. It is determined by the characters on the input file (or terminal) and the field specification used to read them. In general, the input data field consists of

the characters on the input medium that will be read with a given field specification. The number of characters in an input data field is usually determined by the field specification used to read them; for example, F6.2 reads an input data field of 6 characters. However, if the input data field is terminated early by a comma or Carriage Return, fewer than the specified number of characters are read, and the value read is assigned according to rules discussed below.

From the above, we see that an input data field may fill the field specification or not. For numeric input data fields, there are four cases:

1. Field filled: No decimal point
2. Field filled: With decimal point
3. Field not filled: Terminated with a comma
4. Field not filled: Terminated with a Carriage Return.

The following rules summarize the way values are assigned with these different input data fields.

1. Field Filled: No Decimal Point

In this case the data consists of a solid string of numeric characters not including a decimal point. (Blanks may be included, since they are converted to zeroes during numeric input.) The number of characters read is the same as the field width specified in the field specification. The value assigned to the variable being read is determined by the field specification.

Examples

Data Field	Field Specification	Value Assigned
12345	F5.2	123.45
-1234	F5.2	-12.34
123456E+04	E10.4	12.3456E+04
■12345E+04	E10.4	1.2345E+04
123	I3	123

2. Field Filled: With Decimal Point

In this case the data consists of a solid string of numeric characters which does include a decimal point. The decimal point in the input data field overrides any decimal position specified in the field specification. The number of characters read is the number specified as field width in the specification; the decimal point is counted as a character.

Data Field	Field Specification	Value Assigned
1.2345	F6.2	1.2345
123.E04	E7.1	123.E04
-1.234D-05	D10.4	-1.234D-05
■1.234D+06	G10.4	1.234D06

3. Field Not Filled: Terminated With A Comma

A comma may be used to terminate numeric input before the entire field specified has been filled. A field terminated by a comma will be read in free format; that is, the number will be stored exactly as it appears and the field specification will be ignored entirely. Whenever a comma is encountered in a data field, reading is discontinued. If a blank field is terminated by a comma, a value of 0.0 is assigned. Using the free format comma field terminator relieves the user of the time-consuming task of filling all the fields, thus minimizing the time required to create the file and also the space required to store the file.

This feature also allows the user to suppress insertion of a decimal point when none occurs in the data field; for example, F10.2 reads 11, as 11. and not .11 .

Examples

Data Field	Field Specification	Value Assigned
33.1,	F5.3	33.1
44.2,	I5	44.2 (if read into a real variable)
44,	F7.2	44.
	F5.3	0.0

NOTE: Comma termination also applies to input to the L (logical) field specification. However, input to the A (alphanumeric) and S (string) specifications is never terminated by a comma since a comma is considered a valid input character to such fields. Commas also have no effect on input data to the H and X field specifications. They do terminate data read with any form of the G specification, other than string values.

4. Field Not Filled: Terminated By A Carriage Return

In this case, trailing zeroes will be supplied to fill the field, and the value is assigned according to rules 1 and 2 above, depending on whether or not there is a decimal point in the input data field.

Examples

Data Field	Field Specification	Value Assigned
35↵	I6	350000
11↵	F5.2	110.00
356↵	F7.1	356000.0
35.6↵	E7.1	35.6 (=35.6000)

An exception to this rule occurs when the input data field is in E or D exponential form. (Recall that the F, E, D, and Gw.d specifications all read input data fields in the same way.) In this case, trailing zeroes are not added to the exponent field. For example,

E10.4 *(or F10.4 or D10.4)* reads **1234E3**↵ *(or 1234D3*↵*)* as **1234.E+03**

D11.3 reads **1.23E-6**↵ as **1.23E-06**

The following example illustrates the different input data fields.

Example

Suppose file number 3 contains the data

1,234567↵
1.23456↵

Then the statements

```
READ (3,7) A,B,C,D
7 FORMAT (4F5.2)
    or 7 FORMAT (F5.2,F5.2,F5.2,F5.2)
```

assign the following values:

A=1. *Data field terminated by comma*

B=234.56 *Field filled, no decimal point*

C=700.00 *Data field terminated by Carriage Return*

D=0.0 *Field filled with zeroes by previous Carriage Return*

Input Of Complex Numbers

Complex numbers may be read or written by any numeric field specifications. A separate field specification is required for the real and imaginary parts of the number. For example,

FORMAT(F4.1,E8.2)

Read	Into A Complex Variable As
123412345E-7	123.4 + (123.45E-7)i
123,-123.456	123 - 123.456i

SCALING: THE P SPECIFICATION

A scaling factor may be set in a FORMAT statement using the P specification. This specification has the form

fP

where the scaling factor **f** may be any integer: positive, negative, or zero.

The scaling factor affects numbers read and written by the F, E, D, and G field specifications only. Further, only the Gw.d form of the generalized field specification is affected by scaling; the form Gw is not.

The scaling factor indicates the power of 10 by which the values read or written are to be multiplied; that is,

External value = internal value $\times$ $10^{\text{scaling factor}}$

In some cases, the scaling factor changes the actual value read or written; in others, only the representation of the value is affected. Details are given under Input and Output below.

The scaling factor may be set and reset anywhere in a FORMAT statement.

Example

```
100  FORMAT(F6.2,1P,F5.2,F8.6,0P,F8.2,
                          -2PE12.1,F4.1)
```

When values are read or written with this FORMAT statement, the first value read or written (with F6.2) will not be scaled. Scaling by 10^1 will occur when the next two specifications (F5.2 and F8.6) are used. The specification 0P, indicating multiplication by 10^0 or 1, causes the specification F8.2 to be normal; in the last two specifications (E12.1 and F4.1), scaling by 10^{-2} will occur.

Note that the scaling specification need not be separated from the following specification with a comma. Thus,

```
500 FORMAT (3PD11.4)
```

and

```
500 FORMAT (3P,D11.4)
```

are equivalent.

Whenever a FORMAT statement is called, the scaling factor is automatically initialized to zero. Once scaling has been set in a FORMAT statement, it is used throughout that FORMAT unless reset. In the example below, the FORMAT statement is called once and the scaling factor is set to two after the first variable is read.

```
> LIST ↵
   1            READ(0,100) A,B,C,D
   2     100    FORMAT(F6.2,2PF6.2)
   3            DISPLAY A,B,C,D
   4            END
> RUN ↵
111111222222↵
333333444444↵
 1111.11      22.2222      33.3333      44.4444

(@4 ) >
```

When the FORMAT statement is called more than once, as in the next example, the scaling factor is reinitialized to zero each time a READ statement calls the FORMAT.

```
> LIST ↵
   1            READ(0,100) A,B
   2            READ(0,100) C,D
   3     100    FORMAT(F6.2,2PF6.2)
   4            DISPLAY A,B,C,D
   5            END
> RUN ↵
111111222222↵
333333444444↵
 1111.11      22.2222      3333.33      44.4444

(@5  ) >
```

Input

During input the effects of scaling on the F, E, D, and Gw.d specifications are all the same. The value read is affected only if the input data is not in E exponential or D exponential form. If the input data is in exponential form, scaling does not apply.

If the input data is not in exponential form, the effect of the scaling specification fP is given by

value stored = value entered $\times$ 10^{-f}

Examples

Value Entered	Scaling Factor	Value Stored
109.3	1P	10.93
123.4	2P	1.234
1.2E05	1P	1.2E05
113.1	-2P	11310.

Note that when the input data does not contain a decimal point the same relationship applies as when it does. Thus, FORMAT (F4.1) reads 1093 as 109.3, but FORMAT (1PF4.1) reads 1093 as 10.93.

NOTE: Input data that is terminated early by a comma is never scaled.

Output

During output, the effect of the scaling factor depends on the field specification used.

If the E or D field specification is used, the value of the number being printed is not altered; only the representation is affected, as follows: The mantissa is multiplied by 10^f and the value of f is subtracted from the exponent.

If the F field specification is used, the value printed is actually changed from the value stored, according to the relationship

value printed = value stored $\times$ 10^f

If the Gw.d field specification is used, the value printed is never changed. If the G specification prints the value in E format, the scaling factor has the same effect as it would if an E specification were used. If the G specification prints the value in decimal form, scaling does not apply.

Examples

Specification	Normal Output	Scaled Output	
		Scaling Factor	Number Printed
E11.5	■.15000E+02	1	■1.5000E+01
D11.5	■.15000D+02	-1	■.01500D+03
F4.0	■15.	1	150.
G10.4	■15.00■■■■	2	■15.00■■■■
G10.4	■.2641E+05	2	■26.41E+03

NON NUMERIC FIELD SPECIFICATIONS

L FIELD SPECIFICATION

The L field specification is used to read or write logical values only. It has the form

Lw

where **w** represents the total field width.

Input

The field specification Lw causes w characters to be read from the input medium and a truth value assigned to the variable being read according to the following rule: The w characters are scanned for the first occurrence of a T or an F. If a T is found, the value is true. If an F is found or if neither a T nor an F is found, the value is false.

Example

```
> LIST ↵
   1              LOGICAL LOG1,LOG2,LOG3
                        Line 1 is a LOGICAL
                        declaration statement.
   2              READ (0,100) LOG1, ↵*
                  LOG2,LOG3
   3      100     FORMAT (L1,L3,L5)
   4              DISPLAY LOG1,LOG2,LOG3
   5              END
> RUN ↵
TORFALSE? ↵
T       F       F

(@5 ) >
```

In this example, the logical variables LOG1, LOG2, and LOG3 are read using the field specifications L1, L3, and L5, respectively. First, LOG1 is read using L1; one character is read from the terminal. Since this character is a T, LOG1 is true. Then LOG2 is read using L3. The three characters ORF are read from the terminal; since the first logical character encountered is an F, the value assigned to LOG2 is false. Now, LOG3 is read using L5. The 5 characters read from the terminal are ALSE?; since these contain neither a T nor an F, LOG3 is false.

Notice that if LOG1 were read using L4, it would still be true. Even though the four characters read (TORF) contain an F, it is only the first occurrence of a T or F that determines the truth value assigned. If the characters FORT were read using L4, the value read would be false.

NOTE: Input to an L field specification may be terminated early with a comma just as may input to a numeric field specification. Thus, L3 reads T, as true.

Output

When a logical value is written using Lw, a T or F is always printed. If the field width is greater than one, leading blanks will be supplied.

Example

```
> LIST ↵
   1              LOGICAL A,B,C
   2              A=.TRUE.
   3              B=.FALSE.
   4              C= A .AND. B
   5              WRITE (1,10) A,B,C
   6      10      FORMAT (L1,L2,L3)
   7              END
> RUN ↵
T F  F

(@7 ) >
```

A FIELD SPECIFICATION

The A (alphanumeric) field specification allows the user to store any characters in a variable and to write the values of such variables. It has the general form

Aw

where **w** is the total field width.

The A specification may be used to read or write characters stored in integer, real, double precision, complex, or string variables. The maximum number of characters that may be stored in each type of variable is as follows:

Variable Type	Maximum Number Of Characters
Integer	3
Real	6
Double Precision	9
Complex	12
String	The length of the string.

Input

During input, the specification Aw causes w characters to be read into the specified variable. If w is greater than the character capacity of the variable, the right-hand characters are read and the left-hand characters are ignored. If w is less than the character capacity of the variable, the characters will be stored in the left-hand part of the variable and the rest of the variable will be filled with blanks.

Example

Suppose the specification

A6

is used to read the characters

P-1362

and to store them in a variable.

The characters that will be stored in different variable types are as follows:

Variable Type	Characters Stored
Integer	362
Real	P-1362
Double Precision	P-1362■■■
String, declared length=5	-1362
String, declared length=6	P-1362
String, declared length=8	P-1362■■

Note that only one A field specification is used to store characters in a complex variable, even though two numeric specifications are needed to read complex numbers. For example, the statements

```
      COMPLEX C
      READ (0,100)C
100   FORMAT (A12)
```

will read all the characters

TESTING12345

and store them in C, but FORMAT (A6,A6) will read only the first six characters of the input value.

NOTE: If input to an A field is terminated early by a Carriage Return, the rest of the field is filled with blanks.

Output

The specification Aw always prints w characters. If w is greater than the number of characters stored in the variable being printed, leading blanks are supplied. If w is less than the number of characters stored in the variable, the left-hand w characters will be printed.

Example

Using A6,

Variable Type	Characters Stored	Characters Printed
Integer	A12	■■■A12
Real	P-1362	P-1362
Double Precision	CO,MM,AS,	CO,MM,
String (Current Length=4)	TEST	■■TEST
String (Current Length=6)	STRING	STRING
String (Current Length=10)	1234566890	123456

Just as with input, only one A field specification may be used to write characters stored in a complex variable. For example, if the characters

SILLYTEST!!!

are stored in the complex variable COMPLEX, the statements

```
      WRITE (1,200) COMPLEX
200   FORMAT (A12)
```

will print all 12 characters stored in COMPLEX. FORMAT (A6,A6) would print only the first 6 characters stored in COMPLEX.

S FIELD SPECIFICATION

The S, or string, field specification can be used to read or write string constants or declared string data only. It may be used either with or without a field width w, in either of the following forms:

Sw

S

The **Sw** specification causes a string of length **w** to be read or written. To read or write a string of variable length, the **S** specification is used without a field width.

Sw Input And Output

Input

During input, the specification Sw causes exactly w characters to be read. If w is greater than the declared length of the string being read, the right-hand characters will be lost. (This is exactly opposite to the A specification.) If w is less than the declared length of the string, the characters will be stored in the left-hand part of the string and the rest of the string will be filled with blanks.

If input to an Sw specification is terminated early with a Carriage Return, the rest of the field will be filled with blanks.

Examples

Using S5,

If The Declared String Length Is	The Data	Is Read As
5	AB123	AB123
3	AB123	AB1
10	AB123	AB123■■■■■

Notice that when the declared string length is less than the full width, the entire number of characters specified is still read from the input medium; however, only the left-hand characters are actually assigned to the variable. The right-hand characters are lost; they are not available for input to another variable. Thus, the statements

```
    STRING STR1(3), STR2(3)
    READ (0,100) STR1, STR2
100 FORMAT (S5,S5)
```

read the 10 characters

```
ONE;1TWO;2
```

from the terminal and assign ONE to STR1 and TWO to STR2. If the format (S3,S3) were used instead, only the first six characters would be read; ONE would be assigned to STR1 and ;1T to STR2.

Output

During output, the specification Sw causes the first w characters of the string to be written. If w is greater than the length of the string, the string will be printed left justified in a field of width w. The rest of the field will be filled with blanks.

Examples

Using S5,

The String Value	Is Printed As
A	**A**
SP ACE	SP■AC
12345678	12345
ABC	ABC■■

S Input And Output

When the **S** specification is used without a field width, strings of any length may be read or written.

Input

During input, this form of the S specification reads a single record.[1] If the input medium is not a fixed record length random file, S reads all characters preceding the next Carriage Return and assigns them to the string variable being read. If the declared length of the variable is exceeded, the right-hand characters are dropped.

Example

```
>LIST ↵
   1            STRING X(5),Y(10),Z(5)
   2            READ (0,100) X,Y,Z
   3     100    FORMAT(S)
   3.5          DISPLAY X,Y,Z
   4            END
>RUN ↵
ABC ↵
1234567890 ↵
STRING ↵
ABC■■■1234567890■■STRIN

(@4 )>
```

If the input medium is a fixed record length random file, S reads all characters up to the end of the current record. (See *Random Files*, Page 70, for the definition of records on a fixed record length random file.)

Output

During output, the S specification simply writes the specified string, whatever its length.

1 - See *Data Records*, Page 62,

Example

```
>LIST ↵
   1              STRING X(7),Y(27),Z(3)
   2              X="TESTING"
   3              Y="THIS IS THE S ↵
                   SPECIFICATION"
   4              Z="END"
   5              WRITE (1,200) X,Y,Z
   6      200     FORMAT(S)
   7              END
>RUN ↵
TESTING
THIS IS THE S SPECIFICATION
END

(@7 )>
```

In this example, each string is printed followed by a Carriage Return, due to the end of record action generated by the end of FORMAT 200.[1] If the FORMAT were changed to

```
200 FORMAT (S,S,S)
```

the strings printed would all be concatenated, as follows:

```
>RUN ↵
TESTINGTHIS IS THE S SPECIFICATIONEND

(@7  )>
```

NOTE: Do not expect a FORMAT like (S,S,S) above to read three string fields on input since the entire string input would be read with the first S.

LITERAL TEXT IN A FORMAT: THE H SPECIFICATION

The H, or Hollerith, specification is used to include literal text in a format. It may take any of the following equivalent forms, where s is a string of characters and w is the number of characters in s:

Form	Example
wHs	3HEND
's'	'END'
"s"	"END"
s	END

OUTPUT

During output, a Hollerith specification in a format causes the specified text to be printed at the terminal.

Example

```
>LIST ↵
   1              WRITE (1,100)
   2      100     FORMAT (14HTHIS IS A ↵
                   TEST)
   3              WRITE(1,200) 25,SQRT(25)
   4      200     FORMAT($THE SQUARE ↵
                   ROOT OF$,I3,' IS',I2)
   5              END
>RUN ↵
THIS IS A TEST
THE SQUARE ROOT OF 25 IS 5

(@5 )>
```

Notice that Hollerith field specifications may be included in a FORMAT statement either alone or in conjunction with other specifications. When no other specifications are included, the text is printed using a WRITE statement with no output list, as in the first statement in the above example.

INPUT

During input, w characters from the input medium actually replace the characters of the string in the format. (w is the number of characters in the string in the Hollerith specification, no matter which form is used.) Consequently, if the same format is used subsequently for output, the characters read will be written on the output medium.

Example

Consider the program set up in part as

```
.
.
.
   WRITE (1,100)
100 FORMAT (7HTABLE 1)
.
.  statements to print first table
.
   READ (2,100)
   WRITE (1,100)
.
.  statements to print second table
.
```

1 - See *End Of Record Action*, Page 62, and *FORMAT Rescan*, Page 66.

If the next seven characters in file number 2 are

TABLE 2

when the statement READ (2,100) is encountered, execution of the above statements results in the following output:

TABLE 1

.
. *first table is printed*
.

TABLE 2

.
. *second table printed*
.

After the first WRITE statement prints the heading TABLE 1, the statement READ (2,100) causes the characters TABLE 2 to replace the characters TABLE 1 in FORMAT 100. Thus, the second WRITE statement prints the heading TABLE 2 when executed.

SPACING: THE X SPECIFICATION

The X specification, which has the form

wX

causes **w** blanks to be printed during output and **w** characters to be skipped during input. For example, if the statements

```
READ(0,5) A
5 FORMAT (3X,I2)
```

are used to read 15469, A will be assigned the value 69 since the 3X causes the first three characters to be skipped.

During output, the X field may be used to put blanks between the values printed. For example, if X=346.85 and Y=87.341, the statements

```
WRITE(1,15)X,Y
15 FORMAT(F6.2,4X,F6.3)
```

will print the two values with 4 blanks between them, thus:

346.85■■■■87.341

Before discussing the /, &, and T specifications, we introduce the concept of data records, to which these specifications apply.

DATA RECORDS

A data file (including the terminal) is composed of data records, which, in general, are groups of related data items. For example, a record on a payroll data file might consist of an employee number, hourly pay rate, and so on. In SUPER FORTRAN, the exact definition of a data record depends on the configuration of the data file. For symbolic sequential files, the terminal, and symbolic variable record length random files, a data record consists of any number of characters terminated by a Carriage Return, such as

1234,69 ABC ↵

For fixed record length random files, a record consists of the number of elements specified as record length in the OPEN statement.[1]

The concept of a data record for a random file is further clarified later in this section. Here, we are concerned only with rules governing the relationship between FORMAT statements and sequential I/O. These rules also apply to symbolic variable record length random files. However, formatted I/O with fixed record length random files uses different rules due to the different definition of a record in this case. These rules are discussed in *Special Rules For Fixed Record Length File I/O*, Page 72.

In the rest of this section on formatting, the term data record refers to data records consisting of a string of characters terminated by a Carriage Return unless specified otherwise.

END OF RECORD ACTION

FORMAT statements are record-oriented; that is, they are designed to read data that is organized into records. A single FORMAT statement is designed to read or write one record, in the following sense:

1 - See *Random Files*, Page 70.

- During input, the end of a FORMAT statement causes SUPER FORTRAN to seek a Carriage Return before input is continued.[1]
- During output, the end of a FORMAT statement causes a Carriage Return to be printed.[1]

Example 1: Input

If the statement

```
   READ (3,100) A,B
   READ (3,100) C,D
100 FORMAT (I4,I4)   (or FORMAT(2I4)
```

are used to read the data

1490006172 ↵
2223964819 ↵

A is read as 1490 and B is read as 61 (0061). Then the end of the FORMAT statement is reached, so SUPER FORTRAN seeks a Carriage Return, thus skipping the last two characters in the first record (72). The second record is then read; C is assigned the value 2223 and D, the value 9648. The end of the FORMAT again causes a record terminator to be sought, so the characters 19 at the end of the second record are skipped also.

Example 2: Output

The statements

```
   X=2.345
   Y=6.937
   ALPHA=16.491
   BETA=10.393
   WRITE (5,20) X,Y
   WRITE (5,20) ALPHA, BETA
20 FORMAT (F7.3,F7.3)
```

cause the two records

■■2.345■■6.937 ↵
■16.491■10.393 ↵

to be written on file 5. The FORMAT statement labelled 20 is used twice, and, each time the end of the FORMAT is reached, a Carriage Return is printed.

Example 3

The data on file 3

1234.56789AAC ↵
7965.3477,8RTF ↵
9773.996RMT ↵

is read by

```
   LOGICAL L
   READ (3,9)N1,A1,B1,R1
 9 FORMAT (I2,F5.2,F3.1,A3)
   READ (3,9)N2,A2,B2,R2
   READ (3,10)X,Y,Z,L
10 FORMAT (I3,4X,I1,A2,L1)
```

as:

N1=12 A1=34.56 B1=78.9 R1=AAC
N2=79 A2=65.34 B2=77.0 R2=8RT
X=977 Y=6 Z=RM L=T *(true)*

EARLY ENCOUNTER OF END OF RECORD

During input, if a record terminator (Carriage Return) is encountered before the entire variable list in the READ statement has been read, the remaining variables will be set to zero, until the FORMAT indicates that a new record should be processed.

Example 1

The data on file 2

3.6,2.1,2.5 ↵
6.7,3.3 ↵

is read by

```
   READ (2,10) A,B,C,D,E
10 FORMAT (5F10.5)
```

as

A=3.6 B=2.1 C=2.5 D=0. E=0.

The Carriage Return in the first line is encountered after reading the variable C, so the rest of the variable list (D and E) is set to zero.

Example 2

The data

564326.981 ↵
16,,69138 ↵

is read by

```
   READ (3,60) A,B,C,D,E,F
60 FORMAT (3F5.2) or FORMAT (F5.2,F5.2,F5.2)
```

as

A=564.32 B=6.981 C=0.
D=16. E=0. F=691.38

After reading values for A and B, a Carriage Return in the data file is encountered; hence, C is set to zero.

1 - This rule does not apply to fixed record length random files. See *Special Rules For Fixed Record Length File I/O*, Page 72.

However, since the end of the FORMAT is now encountered, D, E, and F are read from the next record. See *Format Rescan*, Page 66.

END OF RECORD SPECIFICATION: /

It is possible to read or write a number of data records using a single FORMAT statement by specifying a slash (/) in the FORMAT statement at the end of the field specifications for each record. Multiple slashes may be used in the FORMAT statement to skip records on input or generate blank records on output. The action of the / in a FORMAT statement is the same as the action of the end of the FORMAT; that is,

- During input, a / in a FORMAT statement causes SUPER FORTRAN to seek a record terminator (Carriage Return), thus skipping to the next record.[1]
- During output, the / causes a Carriage Return to be printed.[1]

Example 1

During input, for the following data

6677 ↵
15432 ↵
243 ↵
893 ↵

the statements

```
    READ(0,50)K,M,N
50  FORMAT(I4/I5//I3)
```

read the first record and assign the value 6677 to K, then read the next data record and assign 15432 to M, then skip the next record, and assign the value 893 to N.

Example 2

The following data on file 2

12345,6789 ↵
12345678 ↵

is read by

```
   READ (2,7) A,B,I,D,E
7  FORMAT (F4.2,F5.2,I2/F5.2,I3)
```

as

A=12.34 B=5. I=67 D=123.45 E=678.

Example 3

The statements

```
    A=111.
    B=2.135E06
    C=5.794
    WRITE (1,100) A,B,C,"ABCF"
100 FORMAT (I3,3X,E10.3/F5.3,5X,S4)
```

print the following

111 .214E+07 ↵ *record 1*
5.794 ABCF ↵ *record 2*

NOTE: When a slash is used between two field specifications, a comma need not be used to separate the specifications; for example, FORMAT (I5,F6.3/G12.6) is allowed.

SUPPRESSING NORMAL END OF RECORD ACTION: &

An & may be used at the end of a FORMAT statement to suppress the Carriage Return that is normally generated by the end of the FORMAT. This allows writing on or reading from a single record using more than one FORMAT statement. This feature also is very useful in documenting formatted terminal input as shown in the example below.

Example

The statements

```
    WRITE (1,100)
100 FORMAT ('VELOCITY =',&)
    READ (0,200)V
200 FORMAT (F6.2)
```

will print

VELOCITY =

and then wait on the same line for the user to enter the value of V. The Carriage Return that would normally be generated at the end of FORMAT 100 is suppressed by the &.

TAB POSITION WITHIN A RECORD: THE T SPECIFICATION

The T (tabs) specification has the form

Tw

It specifies that the next read or write operation start at the w^{th} position in the current record. For example, the statements

1 - This rule does not apply to fixed record length random files; see *Special Rules For Fixed Record Length File I/O*, Page 72.

```
    WRITE (1,100) "DEMONSTRATING T"
100 FORMAT (T10,S15)
```

print the following on the terminal:

```
■■■■■■■■■DEMONSTRATING T
          ↑
          character position 10
```

Input

During input, if the value of w is greater than the present character position, intervening characters are skipped. For example, the statements

```
    READ (3,15) M,N
15 FORMAT (I3,T10,I3)
```

read the data

```
111222ABC333↵
          ↑
          character position 10
```

as

```
M=111 N=333
```

After the value of M is read, the specification T10 causes all characters between positions 4 and 9 inclusive, to be skipped. The value of N is read beginning at position 10.

For sequential files, including the terminal, the value of w may not be less than the current position. However, in reading from a random file, w may be less than the current position, thus allowing the user to back up in the current record to read characters that have been read or skipped previously.

Output

If w is greater than the current character position, intervening positions will be skipped as follows:

- If the output medium is a sequential file (not including the terminal), blanks will be printed in the intervening positions. For example, if file 3 is not random, the statements

```
    WRITE (3,100) M,N
100 FORMAT (I3,T20,I3)
```

  print the value of M followed by 16 blanks followed by the value of N.

- If the output medium is the terminal or a random file, intervening character positions will be unaffected. For example, if file 3 were a random file and the current record were

```
111ABCDEF
```

then the statements

```
    M=222
    N=333
    WRITE (3,50) M,N
50 FORMAT (I3,T7,I3)
```

would change the above record to

```
222ABC333
```

If w is less than the current character position, output to a sequential file is illegal. Output to a random file and to the terminal is as follows:

- If the output medium is a random file, output will resume at the indicated position and any data previously written there will be replaced. For example, if file 4 is random, the statements

```
    X=34.19
    Y=69.5
    WRITE (4,200) X,Y
200 FORMAT (F5.2,T2,F4.1)
```

  print

```
369.5
```

  After the value of X is printed, the T2 specification causes backspacing to the second character position in the current record; thus, the last 4 characters of X are replaced by the 4 characters of Y.

- If the output medium is the terminal, a Carriage Return without a Line Feed is printed and the specified position will then be assumed. Thus, overprinting may result. For example,

```
>LIST↵
  1          STRING G(10),L(10)
  2          G='>>>>>>>>>>'
  3          L='<<<<<<<<<<'
  4          WRITE (1,100) G,L
  5      100 FORMAT (T6,S10,T1,S10)
  6          END
>RUN↵
<<<<<※※※※※>>>>>

(@6 )>
```

REPEATING A FIELD SPECIFICATION

If the same field specification is to be used a number of times, a repeat count may be used. The repeat count appears immediately preceding the field specification and specifies the number of times the field specification is to be read or written. For example, if the format statement

FORMAT (3I4,7F9.6)

is used to read or write data, the format I4 is used for each of the first three values, and F9.6 for the next seven values.

For the following data record

134895765897

the statements

```
    READ(0,15)(A(I),I=1,5)
15  FORMAT(3I2,2F3.1)
```

will assign the following values:

A(1) = 13.
A(2) = 48.
A(3) = 95.
A(4) = 76.5
A(5) = 89.7

Parentheses may be used to indicate the repetition of a series of field specifications. For example,

FORMAT (2(3I3,F12.4),E8.2)

is equivalent to

FORMAT (3I3,F12.4,3I3,F12.4,E8.2)

or

FORMAT (I3,I3,I3,F12.4,I3,I3,I3,F12.4,E8.2)

FORMAT RESCAN

A FORMAT statement will be rescanned automatically whenever the number of items specified in the input or output list exceeds the number of fields specified in the FORMAT statement. Each time the FORMAT statement is scanned, the usual end of FORMAT action occurs; that is, a Carriage Return is sought during input, and a Carriage Return is printed during output.[1] For example, if the FORMAT statement

100 FORMAT (F10.4)

is used with the statement

READ (2,100)(A(I),I=1,3)

the value for each of the elements is expected to be on a different line. (If all three data values are on the same line, use 3F10.4.)

Similarly, if the same FORMAT statement is used for

WRITE(1,100)(A(I),I=1,2)

the value of A(1) will be printed on one line, then a Carriage Return is printed and the same format will be used to print A(2) on the next line.

Rescan of a FORMAT statement always takes place starting with the last first level left parenthesis, if any. (If there is none, rescan takes place from the beginning of the FORMAT statement as above.)

Example 1

If the statement

100 FORMAT (I4/2(F6.8,I2))

is used to read or write more than five variables, rescan would begin with the F6.8 field with the repeat count of 2 in effect. If being used with a WRITE statement specifying 13 values in the output list, the FORMAT would print values in the following form:

I4 ↵
F6.8 I2 F6.8 I2 ↵
F6.8 I2 F6.8 I2 ↵
F6.8 I2 F6.8 I2 ↵

Example 2

The statement

100 FORMAT (3(F6.4,I2),2(I2,3(E10.4,E11.5)))
↑

if used to print more than 20 values, would be rescanned from the parenthesis marked with the arrow; that is, the last first level left parenthesis.

1 - Unless I/O is taking place in a fixed record length random file. See *Special Rules For Fixed Record Length File I/O*, Page 72.

DYNAMIC FORMATS

In SUPER FORTRAN, a format description may be entered into an array or a string variable and the array or string name may be used in subsequent input and output statements to specify the format of the data being transmitted. A DATA statement or a standard input statement may be used to enter a format description into an array or string variable. For example:

```
> LIST ↵
   5              INTEGER V(7)
  10              REAL A(4)
  20              READ(0,100)A
  30      100     FORMAT(4A6)
  40              K=10;L=11;M=12
  50              (V(I)=I),I=1,7
  60              WRITE(1,A)K,L,M,(V(I),I=1,7)
  70              END
> RUN ↵
(I5,1X,I2,I3,7(1X,F6.2))
   10 11 12    1.00    2.00    3.00    4.00    5.00    6.00    7.00

(@70 ) >
```

The format description is entered from the terminal in the READ statement at line 20. The name of the array containing this data is used as the format specification for the WRITE statement at line 60. Note that except for the word FORMAT, the entire format specification is read, including the parentheses.

STRING variables extend the scope of dynamic formatting considerably. Not only may a format description be read into a string variable, the format description may also be computed or altered during program execution. These features are demonstrated in the following example:

```
> FAST ↵
10 STRING S(10)
20 DISPLAY 'ENTER FORMAT'
30 ACCEPT S                    The format is entered
                               from the terminal.
40 DISPLAY 'ENTER NUMBER'
50 ACCEPT A
60 DISPLAY 'NUMBER APPEARS ↵↓
   AS FOLLOWS:'
70 WRITE(1,S) A
80 DO 20 M=3,5
90 S='(E11.'+STR(M)+')'        A different format is
                               specified on each pass
                               through the loop.
100 20 WRITE(1,S) A
110 END
> RUN ↵
ENTER FORMAT
(F14.2) ↵
ENTER NUMBER
666666 ↵
NUMBER APPEARS AS FOLLOWS:
■■■■666666.00
■■■.667E+06
■■.6667E+06
■.66667E+06

(@110 ) >
```

DISK FILE INPUT AND OUTPUT

In SUPER FORTRAN, both sequential and random access disk files are available. In both cases, file input and output may be either symbolic or binary. Symbolic disk file I/O may be read and written either in the formatted form discussed previously, or using the free format READ and WRITE statements discussed below. Binary disk file I/O must always be read and written with free format READ and WRITE statements. No matter what kind of disk file is being used, it must be opened before use with an OPEN statement and closed after use with a CLOSE statement. Up to four files may be open simultaneously.

FREE FORMAT READ AND WRITE

File input and output may be unformatted as well as formatted. Unformatted input and output uses the free format READ and WRITE statements

READ (file number) input list

WRITE (file number) output list

These statements may be used for either terminal or file I/O. However,

READ(0) input list

is equivalent to

ACCEPT input list

and

WRITE(1) output list

is equivalent to

DISPLAY output list

Examples

READ (3)X,Y,Z	*Reads values for X, Y, and Z from file 3.*
WRITE (4)(A(I),I=1,N)	*Writes values for A(1) through A(N) on file 4.*

The free format READ and WRITE statements follow the same conventions as ACCEPT and DISPLAY. When an unformatted READ statement is used for terminal input, a bell will ring to request data just as it does with ACCEPT.

Except for strings, data items on a file being read with free form input may be separated with blanks, commas, or Carriage Returns. String data items may be separated with commas, Carriage Returns, or Line Feeds, but not with blanks.

NOTE: Free format READ and WRITE may not be used for fixed record length random file I/O.

SEQUENTIAL FILES

A sequential file is a file in which reading and writing must take place in the sequence in which data is stored on the file. In other words, once the file is opened, the first data item in the file must be read or written, then the second, and so on. Sequential file I/O will thus prove much slower than random file I/O in many cases. However, sequential files have the advantage of requiring less program overhead than random files.

OPENING A SEQUENTIAL FILE

All disk files must be opened before input or output can take place. They are opened using the OPEN statement, which also assigns a file number to the file and specifies the file type (symbolic or binary). In addition, the mode specified in the OPEN statement specifies whether the file is to be sequential or random.

A sequential file may be opened in one of the following modes:

INPUT	*Sequential input only.*
OUTPUT	*Sequential output only.*

The following form of OPEN is used:

OPEN(**file number,** **"file name",** [**INPUT** *or* **OUTPUT**], [**SYMBOLIC** *or* **BINARY**], ec[1])

For example,

OPEN (3,"PROG1",INPUT,BINARY)

opens the binary file named PROG1 for sequential input and assigns it the file number 3.

File numbers 0 and 1 are reserved for terminal input and output and are not used in the OPEN statement.

1 - ec: The user may include an error condition of the form *ERR=statement label*. Control will be transferred to the statement labelled if there is an error in opening the file. Possible errors are: nonexistent file, incorrect file type, busy file, or inaccessible file.

A literal file name specified in an OPEN statement must be surrounded by single or double quotes. Any valid file name may be used.[1]

If INPUT or OUTPUT is not specified in the OPEN statement, INPUT is assumed. If SYMBOLIC or BINARY is not specified, SYMBOLIC is assumed.

NOTE: Opening a previously created file for OUTPUT causes whatever was already in the file to be erased.

Opening a sequential file in either the INPUT or OUTPUT mode initializes reading or writing at the beginning of the file. The only way to read or write a data item in the middle of a sequential file is to read or write all preceding data items first. This difficulty does not occur with random files. Thus, if the user wishes to read or write only a small part of the total data file, random files should be used.

The file name specified in the OPEN statement need not be specified using a string constant; a string variable or expression may also be used.

Examples

OPEN (5,A,OUTPUT,BINARY,ERR=10)

If A is a string variable whose value is DATA3, this statement opens DATA3 for binary sequential output as file 5 and goes to statement 10 if there is an error in opening the file.

OPEN (4,"/"+STR(I)+"/",OUTPUT)

If I=2, this statement opens a file named /2/ for symbolic sequential output.

Another user's file may be opened if it has been properly declared or if it has an @ in its name.[2] For example,

OPEN (3,"(A3R)@DATA",INPUT,SYMBOLIC)

opens the file named @DATA in account A3, user name R for INPUT.

CLOSING A FILE

A disk file should always be closed after use, using the CLOSE statement. This statement has the form

CLOSE (file number)

If four files are open simultaneously, any of them may be closed with a CLOSE statement so that other files can be opened. Once a file has been closed, the file number may be used later to designate another file.

Once a file has been closed, it must be reopened before it can be reused. With sequential files, this implies that reading or writing must begin anew at the beginning of the file when a file has been closed and reopened.

EXAMPLE: SEQUENTIAL FILE I/O

The program in the following example reads the data in the file ALPHA, squares every number read, and writes the original values and their squares on the file BETA. The values are read using a free format READ, but are printed using a formatted WRITE. The CCS command COPY is used to print the contents of the files on the terminal. Notice that the file BETA contains six records, since the FORMAT statement 100 specifies that each record printed contain two data values, and therefore twelve values are printed with the FORMAT statement.

```
>COPY ALPHA TO TEL
20, 3.44, 654
8.275, -15.7,1.28

>LIST
    1              OPEN (2,"ALPHA",INPUT)
    2              DIMENSION A(6),B(6)
    3              READ (2) A
    4              CLOSE (2)
    5              OPEN(3,"BETA",OUTPUT)
    6              DO 25 I=1,6
    7              B(I)=A(I)**2
    8              WRITE (3,100) A(I), B(I)
    9        25    CONTINUE
   10       100    FORMAT(2F11.2)
   11              CLOSE (3)
   12              END
>RUN
(@12 )>COPY BETA TO TEL
       20.00        400.00
        3.44         11.83
      654.00     427716.00
        8.28         68.48
      -15.70        246.49
        1.28          1.64

(@12 )>
```

1 - See *Rules For Naming Files in Appendix C*, Page 160.

2 - Other user's files may be both read from and written on if the user who owns the directory so desires. See the Tymshare *EXECUTIVE Manual, Reference Series*, for details.

BINARY FILES

Binary files may be read and written in either sequential or random mode. These files cannot be listed in the EXECUTIVE nor read into EDITOR since they are written in binary code, but they do have the advantage of requiring less storage space than a symbolic file, greater accuracy for numeric values, and in many cases are faster to use.

Binary sequential files are opened and closed using the OPEN and CLOSE statements discussed above. Binary files must be read and written using free format READ and WRITE statements; formatted I/O with binary files is never allowed.

Example

```
OPEN (3, "BIN",INPUT,BINARY)
READ (3)A,B,C,(R(I), I=1,9)
CLOSE (3)
```

Further details on binary random files are found in the following section.

RANDOM FILES

SUPER FORTRAN allows random access disk files of either symbolic or binary type in addition to sequential files. With random files, the user may read information beginning at any location in a file, write information on any part of a file without destroying the rest of the file, and erase selected parts of a file. These direct access features allow implementation of many applications that are impossible with sequential files, and provide dramatic increases in the performance speed of certain applications involving the use of data files.

For example: If a file contains descriptions of thousands of parts in an inventory and only one item needs to be updated, the record for that item could be read, changed, and rewritten without any other part of the file being affected.

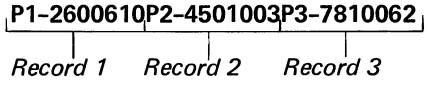
ELEMENTS

The elements of a random file are simply the "units" stored in the file:

- If a file is symbolic, an element is a character in the file.
- If a file is binary, an element is a word (24 bits).

RECORD LENGTH

The record length of a random file is the number of elements (characters or words) in a record. Any record length may be specified by the user when he opens a random file. If he does not specify a record length in the OPEN statement, his file is a variable record length file; that is, it may contain records of different lengths. If a record length is specified in the OPEN statement, the file is a fixed record length file and may contain only records of the specified length. *NOTE: In a symbolic fixed record length file, records need not be terminated with a Carriage Return.* Records are defined solely by the number of characters specified as the record length in the OPEN statement discussed below. For example, a fixed record length symbolic file with a record length of 10 could contain 3 records not separated by Carriage Returns, as follows:



POSITION

Each element in a variable record length file, and each record in a fixed record length file, is assigned a positive integer called the position of the element or record. If a random file has a fixed record length, the first record is at position 1, the second at position 2, and so on. In other words, a position is simply a record number. If a random file has variable record length, a position is an element number (character number if the file is symbolic, word number if it is binary).

Example

Suppose the following records are stored on a symbolic variable record length random file. Then

This Record	Begins At Position
THIS IS A STRING ↵	1
456789 ↵	18
1 ↵	25
END ↵	27

Note that Carriage Returns are counted as elements.

NOTE: If a symbolic random file is created in EDITOR, it should be written with the WRITE ↴ command. The WRITE ↵ command compresses multiple blanks which will cause the position count to be inaccurate.

CURRENT POSITION

The current position of a random file is defined as the next position to be affected by an input or output operation. This is always the position following the position most recently read or written, unless

- It is otherwise specified with an indexed READ or WRITE or POSITION statement (discussed below).
- The file has been opened but nothing has been read or written. In this case, the current position is position 1.

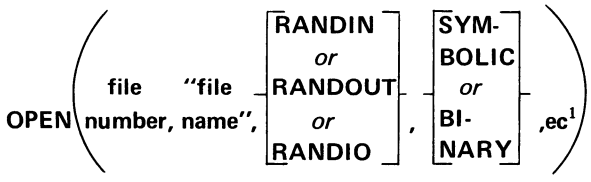
OPENING A RANDOM FILE

In addition to opening a file in the sequential INPUT and OUTPUT modes, the OPEN statement discussed under Sequential Files can open a file in the following random access modes:

Mode	Description
RANDIN	Random, read only
RANDOUT	Random, write only
RANDIO	Random, read/write access (essentially an update mode).

Opening a previously created random file in the RANDOUT mode does not erase the contents of the file as does opening a file in the sequential OUTPUT mode; neither does opening a file in the RANDIO mode.

To open a variable record length random file, use the following form of the OPEN statement.



For example, the statement

OPEN (5, "D49",RANDIN, SYMBOLIC)

opens the symbolic file named D49 for random input as file number 5.

To specify a fixed record length, the number of elements desired as record length is enclosed in parentheses and appended to the mode. For example,

OPEN (3, "DATA12",RANDIO (80), BINARY)

opens a fixed record length binary file for both input and output. Each record contains 80 binary words (80 integer values, 40 real values, etc). On the other hand, the statement

OPEN (4, "FILE1",RANDIO (80), SYMBOLIC)

specifies a fixed record length of 80 characters for the symbolic file FILE1.

THE POSITION FUNCTION AND STATEMENT

At any time, the current position within an opened random file can be found using the library function

POSITION(f)

where f is the file number specified in the OPEN statement. This function returns the current position on file number f as an integer value. For example, the statements

```
OPEN (4, "P68",RANDIO)
.
.
.
I=POSITION(4)
```

set I equal to the current position on the file P68. The next input or output operation performed on this file will take place at position I, unless otherwise specified in a POSITION statement or an indexed READ or WRITE, discussed below.

The POSITION statement should not be confused with the POSITION function. While the POSITION function is used to determine the current position, whatever it may be, the POSITION statement is used to change the current position. This statement takes the form

POSITION(f,p)

It sets the current position on file number f to position p. For example,

POSITION(3,N)

sets the current position on file number 3 to N.

1 - ec: The user may include an error condition of the form *ERR=statement label*. Control will be transferred to the statement labelled if there is an error in opening the file.

Example

```
>LIST ↵
   1        OPEN (5,"VRL",RANDIO)
   1.5      DISPLAY "CURRENT POSITION ¬↓
             IS", POSITION(5)
   2        POSITION (5,7)
   3        DISPLAY "CURRENT POSITION ¬↓
             IS", POSITION(5)
   4        CLOSE(5)
   5        END
>RUN ↵
CURRENT POSITION IS   1
CURRENT POSITION IS   7

(@5 )>
```

In this example, the function POSITION(5) used in line 1.5 returns a value of 1 since the file VRL has just been opened. Then the POSITION statement in line 2 sets the current position to 7, so the POSITION function in line 3 returns the value 7.

Do not forget that position numbers are record numbers for fixed record length files, but are element numbers for variable record length files.

RANDOM FILE READ AND WRITE STATEMENTS

The usual forms of the formatted READ and WRITE statements may be used for random file I/O. In this case, reading or writing begins at the current position. For example, the statements

```
POSITION(3,N)
WRITE(3,100)X,Y,Z
```

write the values of X, Y, and Z on file 3 beginning at position number N.

The free format READ and WRITE statements may be used for symbolic variable record length random file I/O, but not for symbolic fixed record length random file I/O. (Of course, the free format READ and WRITE must always be used for binary file I/O.)

In addition, an indexed READ/WRITE statement is available to specify the position to be affected. It has the formatted form

READ
or
WRITE (file number, format number) (position) I/O list

and the unformatted form

READ
or (file number)(position) I/O list
WRITE

Example

The statement

```
WRITE(3,100)(N)X,Y,Z
```

is equivalent to

```
POSITION(3,N)
WRITE(3,100)X,Y,Z
```

Both write the values of X, Y, and Z on file number 3 beginning at position N.

When data is written on a random file, the new data replaces whatever was previously at the elements written, element for element.

For example, if the following data is on file number 3 (a variable record length symbolic file)

```
12345 ↵
678 ↵
901726 ↵
```

the statements

```
      WRITE(3,200)(8)99
200   FORMAT(I2)
```

produce the file

```
12345 ↵
699 ↵
901726 ↵
```

This fact should be kept in mind especially when writing on variable record length files, since writing a record longer than an existing record will replace elements in the following record.

When reading from or writing on a variable record length file, the user has full responsibility for not violating his own structure for his file. However, he also is given an almost unlimited flexibility with which to solve his data retrieval problems.

When reading from or writing on fixed record length files, however, certain rules protect him from accidentally crossing record boundaries and destroying his record structure. These are discussed below.

SPECIAL RULES FOR FIXED RECORD LENGTH FILE I/O

Binary Fixed Record Length Files

If the file is binary, the size of the variables in the I/O list must exactly match the size of one record.

Thus, if A is a real array, and if file 4 is a binary, fixed record length file,

```
READ(4)(10)(A(I), I=1,20)
```

is illegal unless file 4 has a record length of 40 words.

Symbolic Fixed Record Length Files

Symbolic fixed record length file I/O must always be formatted. The free format READ and WRITE statements may not be used with such files, in either the indexed or non-indexed form.

Since records in a fixed record length file are defined solely by their record length, and not by any special record terminators, the normal end-of-record action caused by the end of a FORMAT or by a / in the FORMAT declaration simply positions the user at the beginning of the next record. It does not generate nor seek a Carriage Return. If the FORMAT used does not read or write an entire record, remaining character positions are skipped during input and filled with blanks during output.

Example

```
>COPY FRT TO TEL↵
P72-459861P73-901652P74-002391
>LIST↵
   1            OPEN (2,"FRT",RANDIO(10))
   2            STRING S(4)
   3            S="P65-"
   4            WRITE (2,100)(2) S
   5     100    FORMAT(S4)
   6            CLOSE (2)
   7            END
>RUN↵
(@7 )>COPY FRT TO TEL↵
P72-459861P65-■■■■■P74-002391
```

In this example, FORMAT(S4) is used to write four characters at position (record number) 2. The end of the FORMAT causes six blanks to be written as the rest of the record.

Suppressing the end of record action with an & allows the user to write part of a record without destroying the rest of the record. Thus, if line 5 in the above example were

```
5        100   FORMAT(S4,&)
```

the file written by the program would be

```
P72-459861P65-901652P74-002391
```

Another READ or WRITE statement subsequently executed in the same program would cause reading or writing to resume in mid-record; specifically, at the fifth element in record 2.

Formatted items must fit within one record on fixed record length symbolic files, unless a / is used to indicate that succeeding data items refer to the following record. Like the end of a FORMAT, the / positions reading or writing at the beginning of the next record and does not generate nor seek a Carriage Return. The / must be used when reading or writing more than one record. Thus,

```
    OPEN(3, "FRLDATA",RANDIO(10))
    WRITE(3,200)(A(I), I=1,10)
200 FORMAT(F10.2/F10.3)
```

is legal, but using

```
200 FORMAT(F10.2,F10.3)
```

instead causes an error message since the statements then attempt to write more than one record without specifying the end of the record. Note, however, that

```
200 FORMAT(F5.2,F5.3)
```

would be legal in the above (assuming it fit the data values) since this format fits within one record of length 10.

FILE SIZE

The size of a random file is defined as

- The position of the last element in the file for a variable record length file.
- The position of the last record in the file for a fixed record length file.

The size of an opened random file may be obtained using the library function

```
SIZE(f)
```

where f is the file number assigned in the OPEN statement. This function returns an integer value equal to the size of file f.

Example

Suppose the file SIZETEST contains the data

```
012479↵
653721↵
598743↵
```

then . . .

```
> LIST ↵
   1            OPEN (5,"SIZETEST",RANDIN(7))
   2            I=SIZE(5)
   3            DISPLAY "FIXED RECORD LENGTH: SIZE =",I
   4            CLOSE (5)
   5            OPEN (5,"SIZETEST",RANDIN)
   6            I=SIZE(5)
   7            DISPLAY "VARIABLE RECORD LENGTH: SIZE =",I
   8            CLOSE(5)
   9            END
> RUN ↵
FIXED RECORD LENGTH: SIZE =     3
VARIABLE RECORD LENGTH: SIZE =     21

(@9 )>
```

ERASING DATA FROM A FILE

Disk storage costs may be reduced by erasing unneeded records within a random file. This is done with the statement

ERASE(f)(i,j)

where f is the file number assigned in the OPEN statement. The ERASE statement erases data from the ith position to the jth position, inclusive. For example,

ERASE(3)(1000,1560)

erases the contents of file 3 from position 1000 to position 1560, inclusive.

The ERASE statement reduces the storage used but does not change the file size unless the last position in the file is erased. If it is, the new file size is set to the last position not erased. For example, if SIZE(2) is 3469 and the statement

ERASE(2)(3000,3469)

is executed, SIZE(2) will become 2999.

An asterisk or a minus 1 may be used to specify the end of a file. Hence either

ERASE(3)(N+1, *)

or

ERASE(3)(N+1, -1)

will reduce the size of file 3 to N.

ERASE never changes the position of any elements or records that have not been erased.

If data is read from erased positions on a file, the values read will be set to zero.

Data may not be erased from a file opened in the RANDIN mode.

EXAMPLE: RANDOM FILE I/O

The program in this example accepts a name from the terminal and assigns it to the string variable NAME1. It then determines which record in the fixed record length file ADDR contains the full name and address of this person, reads this record, and prints the name and address at the terminal. If the name entered is not in the file, a message is printed. Whether or not the name is found in the file, the program continues to request additional names until the user types an ALT MODE/ESCAPE, which transfers control to the statement 50 CLOSE(2) using the ON INTERRUPT statement.

The data in the file ADDR is organized in the following form:

NAME	STREET	CITY ↵
Positions 1-20	Positions 21-45	Positions 46-55 ↑ Position 56

```
>COPY ADDR TO TEL
MR. JOHN B. CAREY      285 COTTLE AVENUE          CAMPBELL■■
MRS. LESLIE FISHER     1964 HAMPTON DRIVE         DANVILLE■■
MR. CARL LARSON        985 SOUTH 9 STREET         SAN JOSE■■
MR. DALE MOSS          1650 SARATOGA AVENUE       SARATOGA■■
MR. JOHN REY           106 FORMAN STREET          CAMPBELL■■
MR. DANIEL TORRES      24 SCHARF AVENUE           LOS GATOS■
MISS DONNA WILKES      315 SOUTH 3 STREET         SAN JOSE■■
MR. MICHAEL YOUNG      60 WILSON ROAD             CHESTER■■■
MR. HENRY C. ZIMMER    15 JACKSON STREET          PALO ALTO■


>LIST
   1            STRING NAME1(10),NAME2(9)(10),NAME(20),STREET(25),CITY(10)
   2            OPEN (2,"ADDR",RANDIN(56))
   3            ON INTERRUPT GO TO 50
   4      5     ACCEPT "ADDRESS OF? ",NAME1
   5            DO 10 I=1,9
   6            IF (NAME1 .NE. NAME2(I)) GO TO 10
   7            READ (2,100) (I) NAME,STREET,CITY
   8            GO TO 30
   9     10     CONTINUE
  10            WRITE (1,200) "THE ADDRESS IS NOT LISTED HERE"
  11            GO TO 5
  12     30     WRITE (1,300) NAME,STREET,CITY
  13            GO TO 5
  14     50     CLOSE (2)
  15            DATA NAME2/CAREY,FISHER,LARSON,MOSS,REY,TORRES,
                WILKES,YOUNG,ZIMMER/
  16    100     FORMAT (S20,S25,S10)
  17    200     FORMAT (/S/)
  18    300     FORMAT (/S/S/S/"CALIFORNIA"/)
  19            END


>RUN
ADDRESS OF? YOUNG

MR. MICHAEL YOUNG
60 WILSON ROAD
CHESTER
CALIFORNIA

ADDRESS OF? CAREY

MR. JOHN B. CAREY
285 COTTLE AVENUE
CAMPBELL
CALIFORNIA
```

```
ADDRESS OF? MORRIS↵

THE ADDRESS IS NOT LISTED HERE

ADDRESS OF? WILKES↵

MISS DONNA WILKES
315 SOUTH 3 STREET
SAN JOSE
CALIFORNIA

ADDRESS OF? ⊕
(@19 )>
```

Note the following points about this program:

1. The symbolic data file ADDR contains nine records of length 56. This file is opened for random input as a fixed record length file in line 2 of the program. We chose to terminate each record in this file with a Carriage Return to facilitate listing the file on the terminal. However, as we discussed in *Record Length*, Page 70, it is unnecessary to terminate records with Carriage Returns when using fixed record length files. If the Carriage Returns were removed from the file ADDR and line 2 were changed to

   2 OPEN(2, "ADDR",RANDIN(55))

   the program would perform exactly as above. However, the COPY command would not produce a readable listing of ADDR since characters would overprint due to the absence of Carriage Returns.
2. The DO loop in lines 5 through 9 determines the record number for the requested address. This is accomplished by comparing NAME1 (the name entered from the terminal) to each element of the nine-element string array NAME2.
3. Values are assigned to NAME2 using the DATA statement in line 15.[1] Data statements are used to initialize variable values before execution of the first executable statement in a program. Thus, the DATA statement in line 15 assigns NAME2(1) the value CAREY, NAME2(2) the value FISHER, and so on up to NAME2(9) the value ZIMMER before execution of the DO loop in lines 5 through 9.
4. The indexed READ statement in line 7 reads record number I from the file ADDR. When NAME1 is equal to NAME2(I), I has a value equal to the record number of the desired record since the names in the DATA statement and the names in the file are in the same order.

PROGRAMMABLE ERROR AND END OF FILE CONDITIONS

END OF FILE PROCESSING

The READ statement incorporates a test on the end of file condition so that termination of input files can be detected and processed by the program. For example,

READ(3,100,END=50)X,Y,Z

If, during the execution of this statement, the end of file 3 is reached before the values of X, Y, and Z have been read, program control is immediately transferred to the statement labelled 50. The program may then process the end of file condition.

INPUT/OUTPUT ERROR PROCESSING

The READ and WRITE statements allow the user to specify a statement label to which control is passed if an error occurs during data transmission. The error condition appears in a READ or WRITE statement as follows:

READ
or **(3,100,ERR=10)(A(I), I=1,6)**
WRITE

The READ statement can employ the error condition and end of file condition at the same time. The following example illustrates this:

```
>LIST↵
  10          OPEN(3,"DATA",INPUT,SYMBOLIC)
  20          READ(3,100,END=10,ERR=50)(A(I),I=1,100)
  30    100   FORMAT(100(I2/))
  40    10    DISPLAY (A(N),N=1,I-1)
  50          CLOSE(3); STOP
```

1 - See *DATA Statements*, Page 79, for further information.

```
60      50      DISPLAY "CORRECT ERROR IN DATA FILE AT LOCATION",I
70              INTEGER A(100)
75              CLOSE(3)
80              END
```

Initially, the file DATA has an error. This will exercise the error condition and print the error message at statement 50:

```
>COPY DATA TO TEL↵
10
12
1%
20

>RUN↵
CORRECT ERROR IN DATA FILE AT LOCATION      3
```

The file DATA is corrected so that all of the numbers will be read to the end of the file and control is transferred to statement 10 which displays the values read.

```
(@80 )>COPY TEL TO DATA↵
 OLD FILE↵

BEGIN INPUT.
10↵
12↵
15↵
20↵
(@80 )>RUN↵
 10     12     15     20


(@50 )>
```

SECTION 7
DECLARATION STATEMENTS

It is essential in Tymshare SUPER FORTRAN, as in other versions of FORTRAN IV, that specific areas internal to the computer be reserved to store certain information relevant to program operation. This information is furnished to the compiler in nonexecutable statements called declaration statements, which are processed before the executable statements in the program are executed. At the time the user types RUN, all declarations in the program are processed. In particular, storage areas are set aside for each variable and array in the program. After the declarations are processed, program execution begins.

Three of the declaration statements, END, FORMAT, and the type declaration STRING, have been discussed. The FORMAT statement is the only declaration statement that can have a label. Eleven declaration statements will be discussed in this section: Comments, DATA statements, DIMENSION, COMMON, the type declarations INTEGER, REAL, COMPLEX, LOGICAL, and DOUBLE PRECISION, and the EQUIVALENCE and EXTERNAL statements. Three other declaration statements will be introduced in *Section 8: FUNCTION, SUBROUTINE, and BLOCK DATA.*

COMMENTS

Comments may be inserted into a program to document the program. They may contain any characters; however, they must begin with either a C: or an *.

Examples

```
C:THIS IS A COMMENT
*COMMENTS ARE USED FOR PROGRAM ↵
 DOCUMENTATION
C:LINES 30-60 COMPUTE STANDARD ↵
 DEVIATION
```

Comments are listed along with other program statements, but are ignored entirely when the program is executed. They are not listed when the program is executed. A comment may appear anywhere in a program except as the last statement, which must always be END.

DATA STATEMENTS

Data statements are used to initialize variable values before program execution. The general form of the DATA statement is

$$\text{DATA } v_1,v_2, \ldots v_m/C_1,C_2, \ldots C_j/,v_{m+1}, \ldots v_n/C_{J+1}, \ldots C_k/$$

where v_1 through v_n are scalar or subscripted variables or array names, and C_1 through C_k are values representing numeric, logical, string, or literal constants.

Examples

```
DATA A/1.1/,B/2E07/,C/T/,I/110/,S/STRING/
DATA A,B,C,I,S/1.1,2E07,T,110,STRING/
DATA A,B/1.1,2E07/,C,I/T,110/,S/STRING/
```

These three DATA statements are all equivalent. They initialize A to 1.1, B to 2E07, the logical variable C to .TRUE., the integer variable I to 110, and the string variable S to STRING, assuming S has a declared length of six or more.

In general, the variable list and constant list of a DATA statement are scanned from left to right. For each variable, a corresponding constant is selected in sequence until the variable list is exhausted. If the number of constants (including * replication discussed below) is exhausted and variables remain to be assigned values, the constant list is rescanned from left to right. For example,

DATA A,B,C /5/

initializes A, B, and C to 5, while

DATA A,B,C /5,10/

initializes A to 5, B to 10, and C to 5.

An implied DO loop may be included in a DATA statement. For example,

DATA (Z(I),I=1,5)/1,2,3,4,5/

sets Z(1) to 1, Z(2) to 2, and so on, up to Z(5) to 5.

An array may be referred to by its name alone in a DATA statement. Thus, if Z has been dimensioned as a one-dimensional 5 element array, the statement

DATA Z/1,2,3,4,5/

is equivalent to the preceding example.

Replication of a constant in a DATA statement may be expressed by preceding the constant with an integer indicating the desired number of repetitions and an *.

Example 1

DATA A,B,C/5,2*7/

is equivalent to

DATA A,B,C/5,7,7/

Example 2

If A has been dimensioned by

DIMENSION A(10)

the statement

DATA A/2*4,3*1/

assigns values to A in the order

4,4,1,1,1,4,4,1,1,1

The constant list is rescanned after values for A(1) through A(5) are assigned.

The * representation is the best method to assign values to arrays in DATA statements. It should prove considerably more efficient than other methods, particularly in the initialization of large arrays.

No matter what the line number of a DATA statement, it will assign values before execution of the first executable statement in the program, and will be executed only once. Thus, after execution of the statements

A=1.0

.

.

.

B=72.5

.

.

.

DATA A,B,C/55.6,100,99/

A has the value 1.0; B, the value 72.5; and C, the value 99. . Even though the DATA statement appears after the replacement statements, it assigns values before the replacement statements are executed.

The above rule applies even if the DATA statement occurs in a subprogram[1]. The DATA statement assigns values before the first executable statement in the entire program, not before the first executable statement in the subprogram. Consequently, dummy variables (such as the parameters in a FUNCTION declaration) may not be referred to in DATA statements.

Variables declared in COMMON[2] may be initialized with DATA statements, unlike many implementations of FORTRAN which require COMMON blocks to be initialized in special BLOCK DATA subprograms. SUPER FORTRAN however, not only allows initialization of COMMON variables in DATA statements, but also provides BLOCK DATA subprograms for compatibility with other versions.[3]

The following rules govern the way different types of constants are assigned to different variable types in DATA statements. Note that in all cases, leading blanks preceding a constant are ignored and constants are terminated with a comma or a /. Embedded blanks are not significant in numeric and logical constants, but are treated as characters in string and literal constants.

1. Numeric Constants

Numeric constants may be in either integer or real form, signed or unsigned. The type of the variable in the list determines the type of the value assigned. Thus, if the type of the variable in the list is real and a corresponding integer constant occurs in the constant list, a real internal value is assigned. If the type of the

1 - See *Subprograms: Programmer Defined Functions And Subroutines,* Page 89.
2 - See *The COMMON Declaration,* Page 84.
3 - See *BLOCK DATA Subprograms,* Page 97.

variable is integer and the corresponding constant contains a fractional part, only the integer part of the value will be assigned.

Example

DATA X,I/100,3.75/

assigns the real variable X the value 100., and the integer variable I the value 3.

Complex variables automatically use two numeric constants from the constant list. Double precision values are also generated automatically. For example, if DBL is a double precision variable and CPX is a complex variable,

DATA DBL/1.06D-13/,CPX/18,9.2/

assigns DBL the value 1.06D-13 (which has 17 digits of accuracy), and CPX the value 18.+9.2i.

2. Logical Constants

Logical constants in DATA statements may be represented by T, F, .TRUE., or .FALSE. However, very nearly any construction will be accepted, since the same scanning rule used with the L format field is used here: The constant will be scanned for the first occurrence of a T or an F. If a T is found, the value .TRUE. is assigned. If an F is found, or if neither a T nor an F is found, the value .FALSE. is assigned.

3. Literal Constants

Literal constants in DATA statements can be in any of the following equivalent forms:

Form	Example	
"string"	"ABC"	
'string'	'ABC'	
nHstring	3HABC	*n is the exact number of characters in the string.*

NOTE: Two characters may not appear in a literal constant: The Carriage Return and the /.

Notice that the DATA statement is the only statement in which Hollerith constants (literal constants of the form nHstring) may be assigned to variables. Hollerith constants are not allowed in replacement statements in SUPER FORTRAN; only quoted string constants may be assigned to variables in any executable statements.

When a literal constant is included in a DATA statement, the specified characters are stored in the corresponding list variable according to the following character capacities:

Variable Type	Maximum Character Capacity
Integer	3
Real	6
Double Precision	9
Complex	12
String	The declared length of the string.

If the number of characters in the constant exceeds the capacity of the variable, the righthand characters are omitted. If the number of characters is less than the capacity of the variable, the characters will be left-justified; the remaining character positions on the right will be filled with blanks.

Example

If I is an integer variable, R a real variable, C a complex variable, and S a string variable of declared length 10,

DATA I,R,C,S/3*'AB-123',7HTESTING/

assigns I the value AB-, R the value AB-123, C the value AB-123 followed by six blanks, and S the value TESTING followed by three blanks.

Note that only one constant is used to assign literals to complex variables, but two constants are needed to assign numeric values to complex variables.

4. String Constants

String constants in DATA statements are defined simply as a string of characters beginning with the first non-blank character in the current position in the constant list, and terminating with a comma or closing /. They are used to assign values to string variables. For example,

DATA STR1,STR2/DOG,CAT/

assigns the value DOG to STR1 and the value CAT to STR2 (assuming the string variables STR1 and STR2 each have a declared length of at least three).

Because of the * replication convention and the comma termination convention, cases may arise when literal constants must be used in place of string constants. For example, the string 4*ABC would be interpreted as ABC, ABC, ABC, ABC. If the user desires a single constant equal to 4*ABC, he should use '4*ABC', "4*ABC", or 5H4*ABC. Similarly,

DATA S1,S2/"AB,19"/

assigns AB,19 to both S1 and S2, but

DATA S1,S2/AB,19/

assigns AB to S1 and 19 to S2.

THE DIMENSION DECLARATION

For the computer to reserve sufficient space for an array, information about the dimensions of the array and the maximum number of elements in the array must be specified in a DIMENSION statement, a type declaration statement, or a COMMON statement. The DIMENSION statement is discussed here; dimensioning with type declaration statements and COMMON statements is discussed in *Type Declaration Statements*, Page 83, and *The COMMON Declaration*, Page 84.

The general form of the DIMENSION declaration statement is

DIMENSION array list

where the array list specifies the array name and the maximum and minimum allowable subscript values for each dimension of the array. Any number of arrays may be dimensioned in a DIMENSION statement.

NOTE: String arrays cannot be dimensioned in a DIMENSION statement. They must be dimensioned in a STRING declaration statement, as was discussed under **Strings**, *Page 40.*

If the minimum subscript value desired is 1, only the maximum subscript value need be specified.

Example 1

DIMENSION A(60),TABLE(20,100)

This statement declares A to be a one-dimensional array with a maximum subscript value of 60. It reserves space for 60 elements of A, A(1) through A(60). It declares TABLE to be a two-dimensional array, or matrix, with a maximum size of 20 by 100, reserving space for the 200 elements, TABLE (1,1) through TABLE (20,100).

Note that more space can be reserved for an array than is actually needed. For example, values could actually be assigned only to elements A(1) through A(30) in a program that dimensioned A as in the above example. However, when an array is called by its name alone in an input or DATA statement, values for every element specified when the array was dimensioned must be assigned. When an array is called by its name alone in an output statement, values for every dimensioned element will be printed. Elements not assigned values or not sharing values in COMMON storage will be set to zero.[1]

If it is desired that the lower subscript of the array be other than 1 (as is automatically assumed in Example 1 above), both the lower and upper limits can be specified by separating them with a colon in the DIMENSION statement.

Example 2

DIMENSION B(6:10)

reserves space for five elements of the array B; namely, B(6) through B(10).

Either or both subscript limits may be negative or zero.

Example 3

DIMENSION MAT1(-3:2),MAT2(-10:-5),↵
ALPHA(0:4,15)

reserves space for the six elements of MAT1, that is, MAT1(-3), MAT1(-2), MAT1(-1), MAT1(0), MAT1(1), and MAT1(2). Space is also reserved for six elements of MAT2, MAT2(-10) through MAT2(-5), and for the 75 elements of ALPHA, a two-dimensioned array with the first subscript ranging from 0 to 4 and the second from 1 to 15.

When the user types RUN, all declarations in the program are processed before the first executable statement is executed. This means that arrays must be dimensioned with integer constants. Arrays may never be dimensioned with variables in a main program or a subprogram. For example

DIMENSION A(10,20)

is allowed, but

DIMENSION A(N,M)

is not.

ARRANGEMENT OF ARRAYS IN STORAGE

The elements of an array of more than one dimension are stored so that the first subscript varies more rapidly than the second, the second more rapidly than the third, and so on. For example,

DIMENSION A(5,3)

reserves storage space for the elements of A in the following order:

A(1,1), A(2,1), A(3,1), A(4,1), A(5,1),
A(1,2), A(2,2), A(3,2), A(4,2), A(5,2),
A(1,3), A(2,3), A(3,3), A(4,3), A(5,3)

1 - See *Variable Initialization*, Page 21.

The statement

DIMENSION B(-1:0,3,0:2)

reserves space for the elements of B in the following order:

B(-1,1,0), B(0,1,0),
B(-1,2,0), B(0,2,0),
B(-1,3,0), B(0,3,0),
B(-1,1,1), B(0,1,1),
B(-1,2,1), B(0,2,1),
B(-1,3,1), B(0,3,1),
B(-1,1,2), B(0,1,2),
B(-1,2,2), B(0,2,2),
B(-1,3,2), B(0,3,2)

It is especially important to keep this order of arrangement in mind when referring to an array by its name alone in an input, output, or DATA statement, and when declaring arrays in COMMON.[1]

```
> LIST ↵
   1            DIMENSION A(3,3)
   2            DATA A/1,2,3,4,5,6,7,8,9/
   2.5          DISPLAY 'A PRINTED AS ARRANGED IN STORAGE:'
   3            DISPLAY A
   3.5          DISPLAY 'A PRINTED ROW BY ROW:'
   3.7          DO 10 I=1,3
   5     10     DISPLAY 'ROW',I,'IS:',(A(I,J),J=1,3)
   7            END
> RUN ↵
A PRINTED AS ARRANGED IN STORAGE:
 1     2     3     4     5     6     7     8     9
A PRINTED ROW BY ROW:
ROW    1     IS:     1     4     7
ROW    2     IS:     2     5     8
ROW    3     IS:     3     6     9
(@7 )>
```

In this example, the two dimensional array A is stored column by column, as usual. To print the values of A in a row by row arrangement, it is necessary to specify a different subscript order (line 5) than is used to determine the arrangement in storage.

TYPE DECLARATION STATEMENTS

Type declaration statements are used to declare the types of variables, arrays, and programmer defined functions. Any of the following types may be specified in a type declaration statement.

INTEGER

REAL

DOUBLE PRECISION (or LONG)

COMPLEX

LOGICAL

STRING

The general form of a type declaration statement is

Type **List of variables, arrays, or functions**

Examples

INTEGER A,POUND(10)

REAL MN,MX(15,20)

DOUBLE PRECISION MEAN (or LONG MEAN)

COMPLEX B,J,T,COM

LOGICAL FIR, SEC, TR, N(10,100)

1 - See *The COMMON Declaration*, Page 84.

The elements in the list following the type are all declared to be of the type specified; thus, in the above examples, A and POUND store integer values, FIR, SEC, TR, and N store logical values, etc.

A STRING type declaration statement must specify not only the variable name, but also the maximum allowable number of characters in each string declared, as was discussed in *Strings*, Page 40. For example,

STRING NAME(20),ADDR(50)

declares NAME to be a single string of maximum length 20, and ADDR to be a single string of maximum length 50.

When arrays are being declared STRING, they must be dimensioned in the STRING declaration statement. Thus, to declare a 10 by 10 string array, each element of which has maximum length 30, the declaration

STRING A(10,10)(30)

must be used. The statement

STRING A(30)

is not acceptable; it always declares A to be a scalar string variable.

All variables, arrays, and functions in a program must be declared. If they are not declared explicitly with a type declaration statement, they are automatically declared implicitly according to the following rule:

If the variable, array, or function name begins with I, J, K, L, M, or N, it is assumed to be integer. Otherwise, it is assumed to be real.

Type declarations, like other declarations, are processed before execution of the first executable statement in the program. Absolute storage locations are assigned to each variable, array element, or function value in the following amounts:

Type	Words Of Storage Assigned
Integer	1
Real	2
Double Precision	3
Complex	4
Logical	1/24
String	1/3 X declared length

Since storage is allocated before program execution begins, a type declaration may not be used to change a value type during program execution.

DIMENSIONING WITH TYPE DECLARATION STATEMENTS

A type declaration statement may be used to dimension an array, thus eliminating the need for a DIMENSION statement. For example, the statements

REAL MN,MX
DIMENSION MN(16,16),MX(0:20,0:20)

are equivalent to the single statement

REAL MN(16,16),MX(0:20,0:20)

String arrays must always be dimensioned in type declaration statements; no other statement may be used to dimension a string array. When dimensioning a string array with a type declaration statement, the subscript limits are specified first, followed by the maximum allowable number of characters per array element. For example,

STRING A12(20,10)(50)

declares A12 to be a 200 element (20 by 10) string array, each element of which can accommodate 50 characters.

Dimensioned arrays and other quantities can be included in the same type declaration statement. Thus,

COMPLEX B,J,COM(-1:3)

declares B and J to be complex, and declares COM to be a complex array, containing the elements C(-1) through C(3).

THE COMMON DECLARATION

A COMMON declaration is used to define an area of common storage which may be used by

- the main, or calling, program, and
- one or more subprograms (functions or subroutines), and
- one or more programs linked with the SUPER FORTRAN LINK statement.[1]

Each COMMON declaration in a main program, subprogram, or linked program specifies the names of variables and/or arrays which are to be placed in COMMON storage. COMMON declarations thus make

1 - See *Subprograms: Programmer Defined Functions And Subroutines*, Page 89, and *Program Linking*, Page 98.

it possible for variables and arrays in the main program to share storage locations with variables and arrays on subprograms or linked programs. Unless a variable or array is specifically declared to be in COMMON, it will be defined only in the main program, subprogram, or linked program in which it appears.[1]

Variables and arrays may be declared to be in blank (unnamed) COMMON storage, or to be in special labelled (named) COMMON blocks. Using labelled COMMON permits a main program to share one area of COMMON storage with one subprogram, and another area with another subprogram. Thus, each subprogram need only declare those common blocks which it uses; and common blocks that have the same name occupy the same storage space in memory.

BLANK COMMON

To declare variables or arrays to be in blank (unnamed) COMMON, use the following form of the COMMON declaration statement:

COMMON variable list

Example

```
COMMON A,B,I
```

The variable list in the above form contains the names of scalar variables or arrays which are assigned to a common storage area in the order in which they are listed. The storage area to which the variables are assigned with this form of COMMON is called a blank COMMON area; that is, no name is given to this area of COMMON storage.

The COMMON declaration is position oriented; that is, the variable name itself is not important (except that it can specify the type of the data). It is the variable position in the list that must be considered. For this reason, a common value need not have the same variable name in the subprogram that it does in the main program; the variables need not even be of the same type.

Example

```
COMMON A,B,C,      Main Program
COMMON X,Y,Z,      Subprogram
```

In this example A has the same value or values as X, B the same as Y, and C the same as Z.

It is important to keep in mind the arrangement of arrays discussed on Page 82, when declaring arrays in COMMON. For example, assuming spelling indicates type, the statements

```
DIMENSION TABLE(2,3)  ┐
COMMON TABLE          ┘ Main Program

DIMENSION X(6)        ┐
COMMON X              ┘ Subprogram
```

cause the following variables to have the same values:

```
TABLE (1,1) and X(1)
TABLE (2,1) and X(2)
TABLE (1,2) and X(3)
TABLE (2,2) and X(4)
TABLE (1,3) and X(5)
TABLE (2,3) and X(6)
```

If more than one COMMON declaration appears in the same program part (main program, subprogram, or LINK program) the effects of the declarations are cumulative. Thus, the statements

```
COMMON A,B
COMMON I,J,K
```

in the same program part are equivalent to

```
COMMON A,B,I,J,K
```

Since variables declared in blank COMMON in both the main program and a subprogram share the same storage locations, COMMON statements may be used implicitly to transmit data to and from the main program and the subprogram. For example, consider the following program:

```
Main Program          Subprogram
COMMON X,Y            SUBROUTINE BRANDX(N)
REAL X                COMMON A,B
INTEGER Y(10)         REAL A
  .                   INTEGER B(10)
  .                     .
  .                     .
CALL BRANDX(M)          .
```

Here, the COMMON statements cause X to have the same value as A, and Y(1) through Y(10) to have the same values as B(1) through B(10), since they have the same storage locations. Thus, values can be transmitted from the main program to the subroutine and

1 - Unless it is passed as a subprogram parameter. See *Subprograms: Programmer Defined Functions And Subroutines*, Page 89.

vice versa without the need to include the variables in the argument list of a CALL statement.

If an array is placed in COMMON using just the array name, it must be dimensioned elsewhere in the same program part.

Example

```
DIMENSION A(10,10)  ┐
COMMON A            ┘ Main Program

COMMON C            ┐
DIMENSION C(10,10)  ┘ Subprogram
```

LABELLED COMMON

The form of the labelled COMMON declaration statement is:

COMMON /name/variable list/name/variable list...

Each block name is enclosed in slashes and the variables to be included in the block follow, separated by slashes. An example of a labelled COMMON declaration is:

```
COMMON /FORM/L,W/COLORS/RED,WHITE
```

This statement assigns L and W to the common area named FORM, and RED and WHITE to the COMMON area named COLORS.

The same COMMON statement may be used to assign values to both blank and labelled COMMON. For example,

```
COMMON A,B/C1/I,X,Y/C2/THETA
```

assigns A and B to the blank COMMON area; I, X, and Y to the COMMON area named C1, and THETA to the COMMON area named C2.

Two consecutive slashes in a COMMON statement indicate that the following variables are to be assigned to blank COMMON. For example,

```
COMMON A,B/C1/ALPHA,BETA//X,Y
```

assigns A, B, X, and Y to blank COMMON (in that order), and ALPHA and BETA to the COMMON block labelled C1.

Labelled and blank COMMON entries are cumulative throughout a program. The following statements are equivalent to the COMMON statement above:

```
COMMON A,B
COMMON /C1/ALPHA
COMMON X,Y
COMMON /C1/BETA
```

A program whose COMMON is defined as:

```
COMMON ALPH,BET/S1/A,B/PROD/X,Y//GAM
```

might selectively declare COMMON blocks in its subprograms as follows:

```
SUBROUTINE SYS1
COMMON /S1/RES,FREQ
    .
    .
    .
END

SUBROUTINE MAT
COMMON /PROD/P1,P2
    .
    .
    .
END
```

This would cause A and B to occupy the same storage as RES and FREQ, and X and Y to occupy the same storage as P1 and P2.

To allow optimized LINK files, COMMON block names are not communicated between programs through linking. Hence, each LINK program must declare the entire COMMON area.

NOTE: A COMMON block may be lengthened either by a subprogram or by EQUIVALENCE.[1]

Dimensioning With COMMON Statements

An array may be dimensioned in a COMMON statement instead of in a DIMENSION or type declaration statement. For example, the statements

```
DIMENSION R(250)
COMMON R
```

are equivalent to

```
COMMON R(250)
```

Also, the statements

```
DIMENSION B(10,20),C(0:50)
COMMON /S1/A,B/HOLD/C
```

are equivalent to

```
COMMON /S1/A,B(10,20)/HOLD/C(0:50)
```

1 - See *The EQUIVALENCE Statement*, Page 87.

Note, however, that an array can be dimensioned only once in a program part (main program, subprogram, or LINK program). Thus,

Correct

```
DIMENSION R(250)  ┐
COMMON R          ┘  Main Program

COMMON X          ┐
DIMENSION X(250)  ┘  Subprogram
```

Incorrect

```
DIMENSION R(250)  ┐
COMMON R(250)     ┘  Within the same program part
```

A type declaration statement may be used instead of a DIMENSION statement to dimension an array used in a COMMON statement. For example, either

```
REAL I(5,5,3)
COMMON I
```

or

```
COMMON I(5,5,3)
REAL I
```

declares I as a real array of three dimensions assigned to blank common storage.

String arrays cannot be dimensioned with COMMON statements; they can be dimensioned only with STRING declaration statements. Thus

```
STRING S(20)(40)
COMMON S
```

are allowed, but

```
STRING S(40)
COMMON S(20)
```

are not.

THE EQUIVALENCE STATEMENT

The EQUIVALENCE statement allows the user to allocate the same storage to different variables (that may be subscripted) within a single program unit. The form of the EQUIVALENCE statement is:

EQUIVALENCE (a,b,c,...),(d,e,f,...),...

The variables separated by commas within a set of parentheses share the same storage locations.

To use EQUIVALENCE effectively, the user should be aware of the following internal storage conventions:

Data Type	Storage Used	Remarks
INTEGER	1 Word	
REAL	2 Words	
DOUBLE PRECISION	3 Words	
COMPLEX	4 Words	Real Part/Imaginary Part
STRING	3 Characters/Word	Begins on a Word Boundary
STRING ARRAY	3 Characters/Word	Each Element Begins on a Word Boundary
LOGICAL	1 Word	
LOGICAL ARRAY	24 Elements/Word	Array Originates on a Word Boundary

The EQUIVALENCE statement lets the user control the assignment of storage so that he may reduce the number of storage locations used in his program or arrange data in a convenient way. For example:

```
INTEGER X(10),Z(200)
EQUIVALENCE (A,B,C),(X(1),Z(99))
```

Variables A, B, and C share the same storage. Array elements X(1) and Z(99) share the same storage location. Furthermore, this EQUIVALENCE statement implies X(2) and Z(100) share the same location.

As a further example, the following statements reduce the number of storage locations and store both real and integer variables in the same array area.

```
DIMENSION ARRAY1(100,100)
INTEGER ARRAY2(200)
REAL ARRAY3(99,100)
EQUIVALENCE(ARRAY2(1),ARRAY1(1,1)),
                (ARRAY1(1,2),ARRAY3(1,1))
```

The first 100 elements of ARRAY1 occupy the same storage as ARRAY2 (200 integer variables take the same number of words as 100 real variables). ARRAY3 shares storage with elements (1,2) through (100,100) of ARRAY1.

An EQUIVALENCE statement must not contradict any previously established EQUIVALENCE conditions. In the example above, for instance, equivalencing ARRAY3(2,1) with any element of ARRAY1 other than ARRAY1(2,2) would be invalid.

A variable in a program or subprogram can be made equivalent to a variable in a COMMON block. However, two variables in COMMON cannot be made equivalent to each other. For example:

```
COMMON A,B,C
DIMENSION R(3)
EQUIVALENCE (B,R(1))
```

This EQUIVALENCE statement causes R(1) to share storage with B and R(2) to share storage with C. R(3) is stored sequentially after R(2) and this causes the size of blank COMMON to be extended. However, it is not permitted to extend the size of COMMON in the reverse direction. To equivalence R(3) with B would be invalid because it would place R(1) before the first location of COMMON storage.

String quantities may be made to share storage with numeric variables. The user should keep in mind that a word of storage can contain three string characters, and that each string variable or element of a string array begins at a word boundary. As an example:

```
STRING S1(8),S2(11)
DIMENSION X(100)
EQUIVALENCE (X(1),S1),(X(3),S2)
```

The EQUIVALENCE statement causes the first six characters of S1 to occupy the same storage locations as X(1), and the last two characters to share storage with X(2). Likewise, the first six characters of S2 occupy the same locations as X(3) and the last five characters occupy the same locations as X(4). If S1 contains 'DELIVER' and S2 contains 'MERCHANDISE', the EQUIVALENCE statement above would cause the following shared storage:

	WORD 1	WORD 2	
X(1)	D E L	I V E	S1
X(2)	R * *		S1
X(3)	M E R	C H A	S2
X(4)	N D I	S E *	S2

*These positions are undefined

The program below prints the equivalenced elements of array X in A format. Note that A1 is used to print X(2). Since only one character has been defined for this element, A2 or A3 would print R followed by undefined characters.

```
>LIST ↵
   1          STRING S1(8),S2(11)
   2          DIMENSION X(100)
   3          EQUIVALENCE (X(1),S1),(X(3),S2)
   4          S1='DELIVER';S2='MERCHANDISE'
   5          WRITE(1,30)(X(I),I=1,4)
   6    30    FORMAT(A6/A1/A6/A5)
   7          END
>RUN ↵
DELIVE
R
MERCHA
NDISE

(@7 )>
```

THE EXTERNAL STATEMENT

In batch implementations of FORTRAN, a subroutine used in a subprogram requires an EXTERNAL statement to allow the compiler to recognize that a particular subroutine is used in the calling sequence. SUPER FORTRAN does not require the EXTERNAL statement, but will accept it for reasons of compatibility. The form of the EXTERNAL statement is:

EXTERNAL a,b,c,...

where a, b, c,... are names of subprograms that are used as arguments in other subprograms. This statement will have no effect on the program in which it appears.

SECTION 8
SUBPROGRAMS:
PROGRAMMER DEFINED FUNCTIONS AND SUBROUTINES

A computational procedure that is to be used more than once in a program usually may be written most efficiently as a subprogram. The subprogram may then be called in the main program whenever the computation is wanted.

A subprogram consists of one or more statements which are stored together outside the main program and which may be called by a name assigned to the group. There are two general categories of subprograms, the function and the subroutine.

A function is designed to return a specific value; it is used in an expression as if it were a variable. There are three kinds of functions in SUPER FORTRAN: the statement function, the FUNCTION subprogram, and library functions. Statement functions and FUNCTION subprograms are defined by the programmer by the methods described in this section. Library functions are functions which are stored permanently on the system by Tymshare, and may be called simply by naming the function with an actual argument list as was described in *Functions*, Page 25.

A subroutine is not designed to return a specific value and cannot be used in an expression. A subroutine must be called explicitly by a CALL statement, and may or may not return values.

Two separate steps are necessary if a subprogram is to be used:

1. The function or subroutine must be defined (unless it is a library function which is defined internally).
2. The function or subroutine must be called.

STATEMENT FUNCTIONS

Statement functions are useful when a single expression is to be evaluated repeatedly.

Statement functions are defined with a single nonexecutable statement of the following general form:

function name (dummy argument list) = expression

Any valid name may be used as a function name. Like a variable, a function has a type, which may be integer, real, double precision, complex, or logical. Function type is determined in the same way as variable type; that is, by the first letter of the function name unless a specific type declaration is used. For example, in the function definition

```
SUM(X,Y,Z)=X+Y+Z
```

SUM is a real function since it begins with the letter S. SUM could be defined as an integer function using the two statements

```
INTEGER SUM
SUM(X,Y,Z)=X+Y+Z
```

The dummy arguments, or formal parameters, in the function definition must be nonsubscripted variable names. Their type is determined by the same rules that determine any variable type; that is, they assume the type indicated by spelling unless explicitly declared in the surrounding program. The dummy arguments are **local** to the statement function; that is, their variable names may be the same as the names of variables used in the surrounding program without assuming the values of these surrounding variables. When the function is called, the dummy arguments in the definition are replaced by the actual arguments, or actual parameters, used in the call (see *Calling A Statement Function* below). The type of the actual arguments must agree with the type of the dummy arguments. The types of the dummy arguments are determined by the type of the variable of the same name in the surrounding program.

The **expression** in the function definition expresses a computation using the dummy arguments. It may also contain variables from the program or subprogram in which the function is defined and called, and may refer to other previously defined statement functions or library functions.

Examples

```
CENT(A)=SIN(A*3.14)
EXTRAP(X,Y,Z)=2*SQRT(B)+X-Y/CENT(Z)
```

A variable used in a function definition which is not a formal parameter, such as B in the definition of EXTRAP above, is **global**; that is, it assumes the value it has in the surrounding program when the function is called.

Statement functions cannot be recursive; that is, the function name cannot appear in the defining expression.

CALLING A STATEMENT FUNCTION

Statement functions may be called only in the program or subprogram in which they are defined. The function is called by the appearance of the function name and the actual argument list in an expression. For example, if FUNT is defined as

```
FUNT(A,B)=A+B*2.-X/Y
```

it could be called in the statement

```
P=FUNT(D,3.).
```

which would be equivalent to

```
P=D+3.*2.-X/Y
```

The actual arguments used in calling a function may be constants, subscripted variables, or more complicated expressions. However, the actual arguments in the function call must agree in type with the dummy arguments used in the function definition.[1] Thus, assuming spelling indicates type,

```
P=FUNT(I,3.)
```
and
```
P=FUNT(X,4)
```

are illegal calls to the function FUNT.

When a statement function is called, the actual arguments replace the dummy arguments in the function definition, and the defining expression is then evaluated. This value is returned to the place where the function was called. Thus, when FUNT is called in the statement P=FUNT(D,3.), the current value of D replaces the argument A in the function definition, and 3. replaces B. The value returned and stored in P is thus the value of the expression D+3.*2.-X/Y. The global variables X and Y assume the values assigned in the surrounding program. Note, however, that A and B may be used in the surrounding program without being changed by the function call, since they are local to the function definition. Thus,

```
>LIST
  1             FUNT(A,B)=A+B*2.-X/Y
  2             X=4.
  3             Y=2.
  4             A=312.3
  5             B=400.08
  6             DISPLAY 'FUNT(7.,8.)
                IS',FUNT(7.,8.)
  7             DISPLAY 'BUT A AND
                B ARE',A,B
  8             END
>RUN
FUNT(7.,8.) IS        21
BUT A AND B ARE       312.3       400.08
(@8 )>
```

A function may be called anywhere that an expression may be used, such as on the right side of a replacement statement, or in a DISPLAY statement.

The value returned by a function call always assumes the type of the function. For example, consider the two function definitions

SR(A,B)=A/B	a real function
IR(A,B)=A/B	an integer function

The function calls in the following statements return the values indicated:

DISPLAY SR(3.,2.)	*Returns the value 1.5(3./2.)*
DISPLAY IR(3.,2.)	*Returns the value 1. Although real division (3./2.) is performed, an integer is returned since IR is an integer function.*

The following example illustrates the use of a statement function.

```
>LIST
  1             C: THIS PROGRAM ILLUS-
                TRATES STATEMENT
                FUNCTIONS
  2             HYP(A,B)=SQRT(A**2+B**2)
  3       10    ACCEPT 'ENTER A: ',A
  4             ANS=HYP(A,4.)
  5             DISPLAY 'HYPOTENUSE
                =',ANS
  6             GO TO 10
  7             END
>RUN
ENTER A: 3.
HYPOTENUSE =      5
ENTER A: 3.5
HYPOTENUSE =      5.3150729063
ENTER A: 4.
```

1 - See *Arguments*, Page 95, for some exceptions to this rule.

```
HYPOTENUSE =      5.6568542495
ENTER A: ⊕
INTERRUPT
3 >
```

The function HYP is defined in line 2 and called in line 4. Since its type is not explicitly declared, it is assumed to be real since the function name begins with H.

FUNCTION SUBPROGRAMS

If the result of a computational procedure is to be used in an expression, but the procedure cannot be written as a single statement function, a FUNCTION subprogram may be used. A FUNCTION subprogram may contain any number of statements, and is an independent subprogram. Variables and statement labels in a subprogram are local to the subprogram; that is, they do not refer to any other variables or statement labels in the main program or other subprograms.

A FUNCTION subprogram may call other FUNCTION subprograms or SUBROUTINE subprograms, but it may not call itself.

DEFINING A FUNCTION SUBPROGRAM

A FUNCTION subprogram definition has the general form

```
FUNCTION function name (dummy argument list)
.
.
.
END
```

The nonexecutable FUNCTION statement must be the first statement of a function subprogram, and the END statement the last. There may be one or more RETURN statements,[1] but it is unnecessary to include one. The function name may be any valid variable name.

Any nonsubscripted variable or array name may be used as a dummy argument. These arguments are merely place holders and will be replaced by the actual arguments specified when the function is called. At least one dummy argument must be specified.

The dummy arguments assume the type indicated by their spelling unless specifically declared in a type declaration statement within the FUNCTION subprogram. When the FUNCTION subprogram is called, the types of the actual arguments must agree with the types of the dummy arguments.[2]

Example: A FUNCTION Subprogram

```
      FUNCTION HYP(A,B)
      IF ((A .EQ. 0) .OR. (B .EQ. 0)) ↵
      GO TO 100
      HYP = SQRT(A**2+B**2)
      RETURN
100   DISPLAY 'ERROR: ONE OF THE ↵
      SIDES IS ZERO'
      HYP = 0
      END
```

When a FUNCTION subprogram is called, the computations indicated in the definition are performed. The value to be returned should be assigned to the function name somewhere in the function definition. When a RETURN or END statement is encountered, the value assigned to the function name is returned to the calling point. In the above example, HYP is set equal to SQRT(A**2+B**2) if neither A nor B is zero, and is set equal to zero otherwise. When the RETURN or END is encountered, the value stored in HYP will be returned to the point at which the function is called.

The dummy arguments in a FUNCTION subprogram are local to the subprogram, just like the dummy arguments of a statement function. However, any other variable names used in the FUNCTION subprogram are also local to the subprogram. Such variables do not assume their values from the main program.

Statement labels are also local to the subprogram in which they are used. Thus, in the above example, there could also be a statement labelled 100 in the main program; transfer of control in the second line of the subprogram would still transfer to the subprogram statement

```
100 DISPLAY 'ERROR: ONE OF THE SIDES ↵
    IS ZERO'
```

1 - See *The RETURN Statement*, Page 96.

2 - See *Arguments*, Page 95, for some exceptions to this rule.

TYPE SPECIFICATION OF A FUNCTION SUBPROGRAM

Like variables and statement functions, all FUNCTION subprograms have a type. The type of the function determines the type of the value returned. The function type is determined by the spelling of the function name, as usual, unless specified in a type declaration statement in the calling program, or in the FUNCTION statement.

If the type of the function subprogram is to be specified in the FUNCTION declaration, the following form is used:

type FUNCTION function name (dummy argument list)

The type specified using this form may be INTEGER, REAL, DOUBLE PRECISION (or LONG), COMPLEX, or LOGICAL. For example, the FUNCTION definition

```
REAL FUNCTION INT(A,B,I,R)
INT=A+B*I/R
END
```

declares INT to be real; that is, INT will return a real value when called. If a type had not been specified, INT would have been assumed to be integer.

The function type can also be specified in a type declaration statement in the calling program. Thus, the following statements are equivalent to the above example:

```
REAL INT
.
.                    Calling Program
.
END
FUNCTION INT(A,B,I,R)
INT=A+B*I/R          Function Subprogram
END
```

To define a string function (a function that will return a string value) the function and type must be declared in a FUNCTION statement of the following form:

STRING FUNCTION function name (dummy argument list) (length)

The type may not be declared in a separate type declaration statement.

Example

```
STRING FUNCTION F(X,Y)(100)
.
.
.
END
```

define a string function named F with two dummy arguments X and Y. This function will return a string value of maximum length 100.

CALLING A FUNCTION SUBPROGRAM

A FUNCTION subprogram is called automatically whenever its name appears followed by the actual arguments required. When the function is called, the actual arguments replace the dummy arguments in the function definition. Then, control is transferred to the statement following the FUNCTION declaration. The function subprogram will be executed step by step until a RETURN or END statement is encountered. Then the value of the function will be returned to the calling program and execution of the calling program will continue.

A function call is treated as an arithmetic expression and may appear anywhere that an arithmetic expression is legal. The actual arguments may be specified as constants, predefined subscripted or nonsubscripted variables, array names, or arithmetic expressions. The type of the actual arguments must agree with the type of the dummy arguments in most cases. (Exceptions are discussed under *Arguments*, Page 95.)

Example

The function HYP defined earlier in this section might be called in a replacement statement, as in line 1.5 of the following:

```
>LIST ↵
   1      ACCEPT 'ENTER SIDES: ',X,Y
   1.5    ANS=HYP(X,Y)
   2      DISPLAY 'HYPOTENUSE IS',ANS
   3      END
   4      FUNCTION HYP(A,B)
```

```
    5          IF ((A .EQ. 0) .OR. (B .EQ. 0)) ↵
               GO TO 100
    6          HYP=SQRT(A**2+B**2)
    7          RETURN
    8     100  DISPLAY 'ERROR: ONE OF ↵
               THE SIDES IS ZERO'
    9          HYP = 0
   10          END
>RUN ↵
ENTER SIDES: 3,0 ↵
ERROR: ONE OF THE SIDES IS ZERO
HYPOTENUSE IS    0

(@3 )>RUN ↵
ENTER SIDES: 3,4 ↵
HYPOTENUSE IS    5

(@3 )>
```

NOTE: All variables are local to the main program or subprogram in which they are defined. However, variable values may be shared both by a main program and a subprogram in one of two ways: Variable values may be passed as parameters in the argument list of the subprogram call (as is done with X and Y in the above example), or they may be declared in COMMON using a COMMON declaration in both the main program and the subprogram, as was discussed in **The COMMON Declaration**, *Page 84.*

Example

The function MEAN in the following example computes the mean of a series of numbers. The number of values in the series is passed as an actual parameter to the subprogram when MEAN is called, as are the values in the series.

```
>LIST ↵
    1          C: MAIN PROGRAM
    2          DIMENSION R(200)
    3          ACCEPT 'TYPE NUMBER OF ↵
               VALUES: ',NVAL
  3.1          DISPLAY 'ENTER VALUES:'
  3.2          ACCEPT (R(I),I=1,NVAL)
    4          RM=MEAN(R,NVAL)
    5          DISPLAY 'MEAN =',RM
    6          END
    7          C: FUNCTION DEFINITION
    8          REAL FUNCTION MEAN(S,N)
    9          DIMENSION S(*)
   10          Z=0
   11          (Z=Z+S(I)),I=1,N
   12          MEAN=Z/N
   13          RETURN
   14          END
>RUN ↵
TYPE NUMBER OF VALUES: 10 ↵
ENTER VALUES:
2.334,4.55,2.368,3.972,4.33,5.11,2.88,3.55,4.01 ↵
4.778 ↵
MEAN =      3.7882
(@6 )>
```

Formal array parameters, like all arrays, must be dimensioned. They are dimensioned in the subprogram definition using either constants or asterisks. These two methods are discussed in detail under *Arguments*, Page 95. The example above uses an asterisk array in the line

```
9 DIMENSION S(*)
```

This kind of dimensioning allows the formal array dynamically to assume the bounds of the actual array passed as a parameter. Thus, when the array R is passed in the call

```
RM=MEAN(R,NVAL)
```

S assumes the bounds of R; that is, it becomes a 200 element array with the same values as R.

SUBROUTINE SUBPROGRAMS

A SUBROUTINE subprogram, or subroutine, differs from a FUNCTION subprogram in that a SUBROUTINE subprogram is not designed to return one specific value to the calling program. For this reason, a subroutine cannot be called in an expression. It must be called using a CALL statement, described below. A SUBROUTINE subprogram may or may not return values. It returns values, if any, through its arguments or through COMMON variables.

The general form of a SUBROUTINE definition is

```
SUBROUTINE subroutine name (argument list)
.
.
.
END
```

A subroutine need not have any arguments. To define a subroutine without arguments, the form

SUBROUTINE subroutine name

.
.
.

END

is used.

The first statement of a subroutine must be a SUBROUTINE statement and the last an END statement. Like function subprograms, subroutines may contain one or more RETURN statements, but a RETURN statement is not required.

The subroutine name may be any convenient name; it is not considered to be of any type. If dummy arguments are used in the SUBROUTINE statement, they must be nonsubscripted variables or array names. The dummy arguments assume the type indicated by their spelling unless specifically declared in a type declaration statement within the subprogram. When the subprogram is called, the types of the actual arguments must agree with the types of the dummy arguments.[1]

As in FUNCTION subprograms, all variables (including dummy arguments) and statement labels used in a subroutine are local to the subroutine.

Example: A Subroutine With Dummy Arguments

```
SUBROUTINE TOTAL(A,N,SUM)
DIMENSION A(*)
SUM=0.
DO 10 I=1,N
SUM=SUM+A(I)
10 CONTINUE
END
```

This subroutine sums N numbers. A, N, and SUM are dummy arguments which will be replaced by actual arguments in a CALL statement. A is an array name, and must be dimensioned in the subroutine. Using DIMENSION A(*) to do this allows A to assume the bounds of the actual argument.[2]

Example: A Subroutine With No Dummy Arguments

```
SUBROUTINE HZD
DISPLAY "PROGRAM NEEDS TEST DATA"
COMMON X,Y,Z
Z=X+Y
END
```

CALLING A SUBROUTINE SUBPROGRAM: THE CALL STATEMENT

A SUBROUTINE subprogram must be called explicitly with a CALL statement. Unlike a FUNCTION subprogram, it may not be called merely by the appearance of its name. The general form of the CALL statement is

CALL subroutine name (actual argument list)

When the CALL statement is executed, the dummy arguments are associated with values of the actual arguments and control is transferred to the statement following the SUBROUTINE declaration. The subroutine statements will then be executed in the order indicated until a RETURN or END statement is encountered which will transfer control out of the subroutine to the statement following the CALL statement. The actual arguments used must agree in order and number, as well as in type, with the dummy arguments in the SUBROUTINE statement.

Example

The subroutine TOTAL defined above could be called with the statement

```
CALL TOTAL (B,5,ANSWER)
```

as is done in the following program:

```
    C: MAIN PROGRAM
    DIMENSION B(10)
    ACCEPT B
    CALL TOTAL(B,10,ANSWER)
    DISPLAY 'THE SUM IS',ANSWER
    END
    C: SUBROUTINE
    SUBROUTINE TOTAL(A,N,SUM)
    DIMENSION A(*)
    SUM = 0.
    DO 10 I=1,N
    SUM = SUM + A(I)
10  CONTINUE
    END
```

When this program is executed, the CALL statement causes the actual arguments B, 10, and ANSWER to replace the dummy arguments A, N, and

1 - See *Arguments*, Page 95, for some exceptions to this rule.

2 - See *Arguments*, Page 95, for detailed rules regarding dimensioning of formal array parameters.

SUM in the subroutine. Thus, the ten elements of the array B are summed and the result is stored in the variable ANSWER. In this example the value of SUM is returned to ANSWER in the main program. This is one way of returning data values from the main program to the subroutine. The other way is to use COMMON statements to declare variables that share the same value. The following program is equivalent to the program above:

```
C: MAIN PROGRAM
DIMENSION B(10)
COMMON ANSWER
ACCEPT B
CALL TOTAL2(B,10)
DISPLAY 'THE SUM IS',ANSWER
END
C: SUBROUTINE
SUBROUTINE TOTAL2(A,N)
COMMON SUM
DIMENSION A(*)
SUM = 0.
DO 10 I=1,N
SUM = SUM + A(I)
10  CONTINUE
END
```

A SUBROUTINE subprogram may be called from the main program or from any subprogram except itself. Subroutines may not be recursive. The SUBROUTINE statement is nonexecutable.

ARGUMENTS

Two types of arguments are found in subprograms, the dummy arguments or formal parameters, specified in the definition of the subprogram; and the actual arguments, or actual parameters, specified when the subprogram is called. Subprogram arguments are a way of passing information (variable values, array values, etc.) between the calling program and the subprogram.

Dummy arguments are merely place holders. No space is reserved for them in memory; however, they do have type, and dummy arrays must be dimensioned using special dimensioning methods discussed below. Only nonsubscripted variables and array names may be used as dummy arguments. When a subprogram is called, the dummy arguments will be identified, one for one, with the actual arguments listed in the calling statement according to their order in the list. Because this replacement process is position-oriented, the order and number of arguments specified must be the same in both the actual and the dummy argument list. However, the names of the variables and/or arrays used are usually different.

The actual arguments which are given when the subprogram is called may be variables, array elements, array names, or arithmetic expressions. For example, the subroutine TOTAL(A,N,SUM) defined previously *(See Page 94)* could be called with any of the following:

```
CALL TOTAL(A,5,X)
CALL TOTAL(B,N(2),Y)
CALL TOTAL(C,R+2,Z)
```

A, B, C are previously dimensioned real arrays. The first CALL will cause the first five elements of the array to be summed and the result stored in X. The second CALL will cause N(2) (a subscripted integer variable previously assigned a value) elements of the array B to be summed and the result stored in Y. The third CALL will cause R+2 (an expression of any type; see below) elements of the array C to be summed and the result stored in Z. A constant or arithmetic expression is used only to initialize the dummy variables. In this case, information may be passed only to the subprogram; any values assigned in the subprogram to the corresponding dummy variable will not be transferred back to the calling program. If the actual argument is specified as a variable, array element, or array name, information may be passed both to and from the subprogram. In this case, if the value of the corresponding dummy variable is changed in the subprogram, the value stored in the actual argument will be changed.

All formal parameters within a subprogram have a fixed type. If they are not specifically declared within the subprogram, they implicitly assume the type implied by their spelling. When the subprogram is called, the types of the actual parameters must, with some exceptions, exactly match the types of the formal parameters.

Thus, if a subroutine header appears:

```
SUBROUTINE S(I,J,A)
```

and there are no internal type declarations, then

```
CALL S(M,N,X)
```

is valid, whereas

```
CALL S(X,M,N)
```

would be an error and would interrupt execution, assuming spelling indicates type.

The following exceptions should be noted:

A. Expressions

The statement

```
CALL S(M,N,N+1)
```

where S is defined as above is allowed. SUPER FORTRAN will automatically convert expressions (other than constants and variables) to the type required by the formal parameter.

B. Strings

A formal string must be matched by an actual string, but the length specification of the formal string is ignored. The length of the actual string is dynamically adopted.

All formal array parameters of a subprogram must be dimensioned. There are two valid ways of doing this; specifically, using constant bounds and asterisks.

1. Constant-Bound Arrays

With this method of dimensioning formal array parameters, the dimensioning statement appears in its usual form, such as

```
DIMENSION A(200)
REAL ALPHA (20,40)
```

However, dimensioning a formal parameter this way, as in

```
SUBROUTINE S(X)
INTEGER X(5,7)
.
.
.
```

is understood to be a local mapping of an externally supplied array. When an actual array is passed to the formal parameter X in the above example, the actual array is checked to see that it contains at least 35 words of storage. But the arrays need not agree in type or number of dimensions. For example,

```
DIMENSION I(60)
.
.
CALL S(I)
.
.
.
END
SUBROUTINE S(X)
INTEGER X(5,7)
.
.
.
END
```

is valid. The first 35 elements of I will be passed to the formal name X when the subroutine is called. I(1) will be passed to X(1,1), I(2) to X(2,1), etc. (See *Arrangement of Arrays In Storage*, Page 82).

When constant-bound arrays are used in dimensioning formal parameters, subscript out of range checking is performed recognizing the local limits ((5,7) in the above).

2. Asterisk Arrays

A formal array can be declared which dynamically assumes the bounds of an actual array. The formal specifications

```
REAL A(*,*)
DIMENSION SIGMA(*)
```

are two examples. When an actual array is passed to the formal name A, above, it must be real and have precisely two dimensions. The bounds of the actual array are adopted. Range checking is on those actual bounds. This is the recommended method to dimension formal parameters, since it allows control over the type and number of dimensions of arrays used as actual parameters. At the same time, maximum flexibility and storage economy are assured.

THE RETURN STATEMENT

The statement

```
RETURN
```

in a FUNCTION subprogram or a SUBROUTINE subprogram returns control to the calling program. In a SUBROUTINE subprogram, the RETURN statement causes a transfer back to the statement following the CALL statement; in a FUNCTION subprogram, control returns to the evaluation of the expression in which the function reference appeared. In Tymshare SUPER FORTRAN it is unnecessary to include a RETURN in a subprogram. If not included, program control will return to the main program at the end of the subprogram. However, there may be one or more RETURN statements for conditional return to the calling program.

BLOCK DATA SUBPROGRAMS

Many implementations of FORTRAN do not allow the user to initialize, by means of a normal DATA statement, variables which appear in COMMON. Such implementations require that the initialization of variables appearing in COMMON be done in a special BLOCK DATA subprogram.

A BLOCK DATA subprogram contains no executable statements. The first statement of the program is BLOCK DATA. It is followed by the COMMON statement. The DATA, DIMENSION, EQUIVALENCE, and TYPE statements associated with the data being defined follow. An example of a BLOCK DATA subprogram is:

```
BLOCK DATA
COMMON /DIMS/X,Y,Z,R/ALPHA/A,B,C
DIMENSION A(10), B(2,2)
DATA X/2.1/,R(3)/3*1.0/
END
```

Note that all elements of the COMMON block must be listed whether or not they are initialized.

Since SUPER FORTRAN makes no restrictions on including COMMON variables in a DATA statement, the user would normally use a DATA statement rather than a BLOCK DATA subprogram to initialize these variables. However, the BLOCK DATA subprogram is incorporated into SUPER FORTRAN for compatibility.

SECTION 9
EXECUTION STATEMENTS

PROGRAM LINKING

SUPER FORTRAN allows very rapid linking, preserving COMMON, between separately compiled binary programs. This feature allows the user to increase his total effective program size indefinitely by linking any number of binary programs.

Program linking is accomplished with the statement

LINK "file name"

which causes the binary program on the specified file to be loaded rapidly with its COMMON area automatically initialized to the values currently present in COMMON storage.

The binary program file specified in the LINK statement must be created with the CCS SAVE command, discussed in *Storing A Program On A Disk File: SAVE*, Page 107. Such binary program files contain the binary object program as well as the symbolic source text. However, when such a file is linked, only the binary program is brought into storage; the symbolic source text is not. Thus, the text of the previous program is retained when a LINK statement is executed.

Example

Suppose the file AREA contains the program

```
1 COMMON AREA
2 ACCEPT "ENTER BASE AND HEIGHT: ",↴
BASE,HEIGHT
3 AREA=BASE*HEIGHT/2
4 DISPLAY "AREA OF TRIANGLE IS:",AREA
5 LINK "VOLUME"
6 END
```

and the binary program file VOLUME contains the program whose symbolics are

```
1 COMMON ARBASE
2 ACCEPT "ENTER HEIGHT: ",HEIGHT
3 VOLUME=ARBASE*HEIGHT
4 DISPLAY "VOLUME OF REGULAR PRISM ↴
  IS:",VOLUME
5 END
```

Then ...

```
>LOAD AREA↵
OK.
>RUN↵
ENTER BASE AND HEIGHT: 3.1,5.2↵
AREA OF TRIANGLE IS:      8.06
ENTER HEIGHT: 9.↵
VOLUME OF REGULAR PRISM IS:    72.54
```

Here, the statement LINK "VOLUME" causes the binary object code in the file named VOLUME to be loaded and executed. The value of the variable ARBASE is initialized to the value of AREA computed in the first program since both these values are declared in COMMON.

Any interruptions of a linked program (including the end of the program) return the user to the command mode indicated by the >. At this point, all CCS commands which operate on the program text apply to the text of the program present when the LINK was executed (AREA in the above example).[1] In particular,

- RUN runs the previous program only; to re-run the linked program, another LINK statement (or a CCS LINK command) must be executed.
- LIST (and FAST and COPY) will list the previous program only.
- SAVE will save the previous program only. If SAVE is used to create a binary program file, the binary program stored will be the previous program and not the linked program.

No CCS commands which apply to a running program are allowed after a linked program is interrupted, except for NEXT and CONTINUE, which apply to the binary program. In particular, no direct statements or breakpoints will be accepted.

There is no limit to the number of programs that may be linked. For example, a LINK statement could have been included in the binary program file VOLUME, above, to execute another binary program file.

The SUPER FORTRAN statement LINK should be distinguished from the CCS command LINK which may also be used to execute binary program files. The CCS command LINK works just like the SUPER FORTRAN statement LINK except that the former initializes COMMON storage to zero instead of to the current values. When in the command mode, a direct LINK statement may be used to link a binary program file and still preserve COMMON. For example,

```
50 >@LINK "BFILE"
```
Distinguished from >LINK BFILE

1 - See *Section 10 - CCS SUPER FORTRAN Commands*, Page 101, for descriptions of the commands mentioned here.

COMMAND FILES

It is possible to instruct SUPER FORTRAN to take commands and/or data from a file instead of from the terminal. Such a file is called a command file; it may be created in EDITOR, the EXECUTIVE, or in SUPER FORTRAN with the CCS **COPY TEL TO file name** command.[1]

When creating a command file, the commands should be typed into the file exactly as they would normally be given from the terminal. For example, the command file COM may be created as follows:

```
>COPY TEL TO COM↵
 NEW FILE↵
BEGIN INPUT.
COPY DATA1 TO TEL↵
COPY DATA2 TO TEL↵
LOAD PROG↵
LIST↵
RUN↵
QUIT↵
LOGOUT↵
Dc
>
```

Note that the file COM contains an EXECUTIVE command (LOGOUT) as well as CCS commands. Any commands that can be typed at the terminal can be included in a command file. Since SUPER FORTRAN program lines are simply one form of the CCS ENTER command, program statements may also be entered from a command file.

The commands stored in a command file can be executed either with the CCS COMMANDS command discussed in *Executing Command Files In CCS*, Page 123, or with the following form of the OPEN statement:

OPEN (*,"file name")

For example, the statement

OPEN (*,"COMS")

opens COMS as a command file. Each time a command is sought, it will be taken from the file COMS. Thus if a PAUSE or a STOP statement is executed, CCS commands are taken from COMS rather than entered at the terminal. Encountering the end of the program will also cause commands to be taken from COMS.

The statement

CLOSE (*)

closes a command file. When a command file is closed, the source of commands again becomes the terminal.

Example

Suppose the file TEST contains the commands

```
@DISPLAY "WE ARE IN TEST"
@DISPLAY "NOW, THE CONTINUE COMMAND:"
CONTINUE
@DISPLAY "BACK IN TEST!"
@DISPLAY "NOW, END OF COMMAND FILE:"
```

The following program opens TEST and executes the commands in it.

```
>FAST↵
1 OPEN (*,"TEST")
2 DISPLAY "MAIN PROGRAM"
3 PAUSE "COMMANDS NOW TAKEN FROM↵
  TEST:"
4 DISPLAY "BACK IN MAIN PROGRAM!"
5 DISPLAY "NOW, THE PROGRAM END:"
6 END
>RUN↵
MAIN PROGRAM
COMMANDS NOW TAKEN FROM TEST:

WE ARE IN TEST

NOW, THE CONTINUE COMMAND:

BACK IN MAIN PROGRAM!
NOW, THE PROGRAM END:

BACK IN TEST!

NOW, END OF COMMAND FILE:

(@6 )>
```

First, a PAUSE in the main program causes commands to be taken from TEST. A CONTINUE in the command file causes execution to resume at line 4. Then the program end is encountered and commands are again taken from TEST. When the end of the command file is reached, control is returned to the terminal.

After it has been opened, data can be read from a command file by using an asterisk as the file number; for example,

READ (*,100) N,(A(I),I=1,N)

If file number 0 is specified, data is always read from the terminal, as it is with ACCEPT. However, the free form READ statement may be used to read data from a command file; for example:

READ (*)X,Y,Z

1 - See *Section 10 - CCS SUPER FORTRAN Commands*, for descriptions of commands mentioned here.

If a file opened by the CCS command COMMANDS is still open during program execution, it may be referred to as file * in the program being executed.

For example, consider the command file named COM which contains

```
LOAD DEM ↵
LIST ↵
RUN ↵
7 ↵
```

This file is used as follows:

```
                                                    Command Taken
                                                      From COM:
>COMMANDS COM ↵
OK.                                          ←———— LOAD DEM
  1              ACCEPT "A IS ",A            ←———— LIST
  2              READ(*) B
  3              DISPLAY "THIS IS B:",B
  4              END
A IS 15 ↵                                    ←———— RUN
THIS IS B:       7                           A was read from terminal,
                                             but B was read from COM,
(@4 )>                                       referred to as file * in line
                                             2 above.
```

SECTION 10

CCS SUPER FORTRAN COMMANDS

CCS commands are not part of the FORTRAN IV language but are special commands in the Tymshare SUPER FORTRAN system allowing easy, efficient program manipulation. CCS includes commands for creating, storing, retrieving, editing, executing, and debugging SUPER FORTRAN programs.

Whenever CCS types a >, the user is in the command mode. He then may use any of the CCS commands described in this section.

LINES AND LINE NUMBERS

Each line of a SUPER FORTRAN program has a line number. The numbered line may contain one statement or several executable statements separated by semicolons; the line is terminated by a Carriage Return. Line numbers may range from .001 to 999.999, inclusive.

Examples

35 10 CONTINUE *Has line number 35, but statement label 10.*

40.99 I=J+2;Z(I)=X(I)+Y(I) *Has line number 40.99; it contains two executable statements.*

Line numbers are independent of any statement label, and may not be referred to in control statements. Thus,

DO 10 I=1,N

would be a legal reference to the first line above, but

DO 35 I=1,N

would not.

There is no limit to the number of statements that may be contained in the same line (except that imposed by available program storage). However, the following restrictions must be observed when more than one statement is included in a line:

- Only the first statement in the line may be labelled.
- Only executable statements may be contained in the same line.

The Line Feed may be used to continue a CCS line.

All statements are compiled and executed according to the order of their line numbers. When more than one statement is contained on the same line, they are executed from left to right and listed as created, with semicolons separating the statements.

The maximum number of lines in a SUPER FORTRAN program is 1023; there is no limit to the number of statements.

LISTING A PROGRAM

All or part of a program can be listed in a formatted form for easy reading, or in an unformatted form for quicker printing. It may be listed either on the terminal or on a file.

THE LIST COMMAND: FORMATTED LISTINGS

This command lists all or part of a program in a formatted form with line numbers, statement labels, and statements aligned vertically. It has the general form

> LIST {**line numbers or ranges**} {**TO file name**}↵

where everything in braces is optional.

The command

> LIST ↵

lists the entire program on the terminal.

Example

```
> LIST ↵
  1          C: PROGRAM TO COMPUTE ↓
             THE AREA OF A TRIANGLE
  3          C:
  5          ACCEPT "ENTER VALUES ↓
             OF A,B,C",A,B,C
  7          S=(A+B+C)/2.
  9          AREA=SQRT(S*(S-A)*(S-B) ↓
             *(S-C))
 11          WRITE (1,200) A,B,C,AREA
 13    200   FORMAT (4F16.8)
 15          STOP
 17          END

>
```

The command

> LIST line number ↵

types the line with the specified line number (if one exists) together with its line number.

Example

```
> LIST 9 ↵
  9          AREA=SQRT(S*(S-A)*(S-B) ↓
             *(S-C))
```

The command

> LIST starting line number:terminating line number ↵

lists on the terminal all lines in the specified range (if any exist) together with their line numbers. The range consists of all lines from the starting line number to the terminating line number, inclusive.

Example

```
> LIST 11:15 ↵
 11          WRITE(1,200) A,B,C,AREA
 13    200   FORMAT(4F16.8)
 15          STOP
```

Several line numbers and/or ranges may be specified in the LIST command by separating the line numbers and ranges with commas.

Example

```
> LIST 1:4,7,13:15 ↵
  1          C: PROGRAM TO COMPUTE ↓
             THE AREA OF A TRIANGLE
  3          C:
  7          S=(A+B+C)/2.
 13    200   FORMAT (4F16.8)
 15          STOP
```

To list all or part of a program to a file, use any of the above forms with the **TO file name** option. For example,

```
> LIST TO PRETTYFILE ↵        Lists entire program
                              on the file named
                              PRETTYFILE.
 NEW FILE ↵
OK.
> LIST 1,3:5,49:60 TO @F1 ↵   Lists the lines 1, 3
                              through 5, and 49
                              through 60 on the
                              file named @F1.
 NEW FILE ↵
OK.
```

When lines are listed to a file, the computer responds with OLD FILE or NEW FILE depending on whether or not the file specified already exists in the user's directory. These responses are discussed in detail on Page 107.

THE FAST COMMAND: QUICK LISTINGS

An unformatted quick listing can be obtained with the FAST command. FAST has the general form

> FAST {line numbers or ranges} {TO file name} ↵

where everything in braces is optional.

The various forms of FAST are analogous to the forms of LIST. For example, a quick listing of the previous program could be obtained as follows:

```
> FAST ↵
1 C: PROGRAM TO COMPUTE THE AREA ↓
    OF A TRIANGLE
3 C:
5 ACCEPT "ENTER VALUES OF A,B,C", ↓
    A,B,C
7 S=(A+B+C)/2.
9 AREA=SQRT(S*(S-A)*(S-B)*(S-C))
11 WRITE (1,200) A,B,C,AREA
13 200 FORMAT(4F16.8)
15 STOP
17 END
```

Other examples are:

```
> FAST 3 ↵
> FAST 5:15 ↵
> FAST 1,7,11:17 ↵
> FAST TO QUICKFILE ↵
> FAST 5:17 TO TRI ↵
```

LINE ADDRESSING

Like LIST and FAST, many other CCS commands allow the user to specify the line or lines upon which the command is to operate. This may be done by using any of the various methods of line addressing provided by CCS to specify the address of a single line or range of lines. The address of a range of lines always takes the form

$a_1:a_2$

where a_1 is the address of the first line in the range and a_2 is the address of the last line in the range.

To illustrate the various methods of line addressing, we assume that the following program has been entered into CCS:

```
10 A=2.7;B=7.1
20 ACCEPT C
25 D=A*C+B
30 DISPLAY D
35 END
```

ADDRESSING A LINE BY ITS LINE NUMBER

The easiest way to address a line is by its line number, as we did in the previous section on LIST and FAST.

Example

```
> FAST 10 ↵
10 A=2.7;B=7.1
>
```

ASTERISK ADDRESSES

An address of the form

$*n$

where n is an integer, refers to the nth line in the program.

Examples

```
> FAST *1 ↵
10 A=2.7;B=7.1
>

> LIST *2:*4 ↵
  20            ACCEPT C
  25            D=A*C+B
  30            DISPLAY D
>
```

ADDRESSING THE CURRENT LINE

The current line, or line upon which CCS has most recently operated, may be addressed by a period. For example, since we just listed the fourth line in the program, we may list it again as follows:

```
> LIST . ↵
  30            DISPLAY D
>
```

The command

```
> FAST .,10:25 TO FILE2 ↵
```

writes the current line and lines 10 through 25 on the file named FILE2.

ADDRESSING THE LAST LINE IN A PROGRAM

A dollar sign may be used to address the last line in a program.

Example 1

```
> FAST $ ↵
35 END
>
```

Example 2

```
> LIST *3:$ ↵
  25            D=A*C+B
  30            DISPLAY D
  35            END
>
```

RELATIVE ADDRESSING

The user may address any line by indicating its position relative to another line. Such relative addresses may be specified as follows:

$a+n$	Refers to the nth line after the line addressed by a.
$a-n$	Refers to the nth line preceding the line addressed by a.

Here, a is a line address and n is an integer.

Example

Lines 20 through 30 in the above program may be addressed as in the following:

```
> LIST 10+1:$-1 ↵
  20            ACCEPT C
  25            D=A*C+B
  30            DISPLAY D
```

These lines could also be listed using any of the following:

```
>LIST 10+1:10+3 ↵
>LIST 10+1:35-1 ↵
>LIST *2:35-1 ↵
>LIST 20:.-1 ↵
```

If the current line is the last line in the program when this command is executed, lines 20 through the line before the last line will be listed.

ENTERING, STORING, AND RETRIEVING A PROGRAM

ENTERING STATEMENTS FROM THE TERMINAL

Statements may be entered from the terminal by using any one of three forms of the ENTER command. In all cases, the word ENTER is optional.

ENTERING STATEMENTS BY LINE NUMBER

A single line may be entered from the terminal when in the command mode by using either

>line number statement or statements ↵

or

>ENTER line number statement or statements ↵

A Line Feed may be used to continue the line on the next physical line as it appears on the terminal. The entire line must be terminated by a Carriage Return. If the line number specified in the command is already the line number of a statement in the program, the line typed replaces the original line. If the line number specified in the command is not already given to a line in the program, the line typed is inserted into the program according to line number order. Remember that more than one statement may be included in the same line only if all statements in the line are executable.

Example

If this program has already been entered,

```
1 DIMENSION X(10),Y(10)
2 INTEGER Z
3 Z(I)=X(I)+Y(I);W(I)=(X(I)+Y(I))↑2
4 END
```

the commands

```
>2 INTEGER Z(10) ↵
>2.5 DO 10 I=1,10 ↵
>ENTER 3.5 10 CONTINUE ↵
>3 Z(I)=X(I)+Y(I);W(I)=Z(I)↑2 ↵
```

produce the program

```
1 DIMENSION X(10),Y(10)
2 INTEGER Z(10)
2.5 DO 10 I=1,10
3 Z(I)=X(I)+Y(I);W(I)=Z(I)↑2
3.5 10 CONTINUE
4 END
```

Since the word ENTER is optional, the user can create a program by typing each line preceded by its intended line number. Lines can be changed, or new lines inserted, simply by using line numbers which are identical to, or fall between, line numbers previously used in the program.

ENTER WITH A LINE NUMBER RANGE

A line or range of lines may be entered without typing line numbers by using ENTER in either of the following forms:

>ENTER line number:line number ↵
lines to be entered
D^C

or

>line number:line number ↵
lines to be entered
D^C

After the Carriage Return following the command, CCS prompts the user with an @ sign. The user may now type his lines without line numbers; CCS prompts with the @ sign at the beginning of each line. To terminate the command, type a Control D immediately after the @. CCS assigns line numbers to the lines typed, beginning with the first number in the range and choosing as increment the first of 1, .1, .01, and .001 that will allow the lines typed to fit in the range specified. All statements already in the program with line numbers in the range specified in the ENTER command are deleted; then the lines typed are entered into the range.

Example

If this program has already been entered,

```
1 DIMENSION X(10),Y(10)
2 INTEGER Z
3 Z(I)=X(I)+Y(I);W(I)=(X(I)+Y(I))↑2
4 END
```

the following ENTER command

```
>2:3 ↵
@INTEGER Z(10) ↵
@DO 10 I=1,10 ↵
@Z(I)=X(I)+Y(I);W(I)=Z(I)↑2 ↵
@10 CONTINUE ↵
@D^C
>
```

produces the program

```
1 DIMENSION X(10),Y(10)
2 INTEGER Z(10)
2.1 DO 10 I=1,10
2.2 Z(I)=X(I)+Y(I);W(I)=Z(I)↑2
2.3 10 CONTINUE
4 END
```

The four statements typed are assigned line numbers between 2 and 3 in increments of .1. All old lines in the range 2:3 are deleted.

ENTER WITH PROMPTED LINE NUMBERS

The third form of ENTER allows the user to enter a line or range of lines by specifying a starting line number, an increment, and, if desired, a terminating line number. The command may be used in either of the following forms:

```
>ENTER line number(increment)line number ↵
```

or

```
>line number(increment)line number ↵
```

During this form of ENTER, CCS prompts the user by typing, at the beginning of each line, the line number to be assigned to the next line entered.

The terminating line number in the command is optional. When it is specified, CCS responds to the command by first deleting all lines from the starting line number to the terminating line number, inclusive; then it begins prompting the user as he types his lines. When the terminating line number is reached, CCS terminates the command. However, the user may terminate the command at any time before the terminating line number is reached by typing a Control D just after a line number is printed.

Example

```
>10(1)100 ↵
10 A=5;B=2*A ↵
11 X=SQRT(A+B) ↵
12 Y=SQRT(A-B) ↵
13 D^C
```

In response to this command, CCS first deletes lines 10 through 100 and then enters the new lines typed. If a D^C were not used to terminate the command, CCS would terminate it after line 100 was typed.

When the terminating line number is omitted from the ENTER command, statements will be accepted until the user types a Control D after a prompted line number. Thus, the above program could also be entered as follows:

```
>10(1) ↵
10 A=5;B=2*A ↵
11 X=SQRT(A+B) ↵
12 Y=SQRT(A-B) ↵
13 D^C
>
```

When using ENTER in this form, the user is protected against accidentally changing or interleaving program lines. If the next entry would cause changing or interleaving of lines, CCS automatically terminates the command instead of giving a line number prompt.

Example

```
>FAST ↵
20 DISPLAY Z
>1(10) ↵
1 ACCEPT X,Y ↵
11 Z=SQRT(X**2+Y**2) ↵
>
```

CCS terminates the command, since the next entry (line 21) would be entered after the existing line 20.

SYNTAX ERRORS DURING ENTER

In any of the three forms of ENTER, a syntactically incorrect statement causes an error message to be typed. Control is returned to the user as follows:

If a *single line* is being entered, CCS types a >. The user may then correct the statement or type an entirely different line with a different line number.

Example

```
> 10 A=B+*D ↵

* MISUSED
> 10 A=B+C*D ↵
>
```

If lines are being entered with the @ prompt, CCS reprompts with an @ after printing the error message. The statement may then be typed correctly. The corrected statement is entered into the program in place of the incorrect statement.

Example

```
> 10:20 ↵
@ A=B+*D ↵

* MISUSED
@ A=B+C*D ↵
@
```

If lines are being entered with line number prompts, CCS reprompts with the same line number and the statement may be corrected.

Example

```
>10(2) ↵
10 A=B+*D ↵

* MISUSED
10 A=B+C*D ↵
12
```

ENTERING A PROGRAM FROM PAPER TAPE

If the user's program is on paper tape, either of two commands may be used to enter it from paper tape.

If each line of the program on paper tape has a line number, use the command

> LOAD TEL ↵

The system will type

EACH LINE MUST HAVE A LINE NUMBER.
BEGIN INPUT.

Now turn on the paper tape reader. After the tape is read, turn off the paper tape reader and type a Control D.

The LOAD TEL command deletes any program lines already in CCS before it loads the new program lines. Thus LOAD cannot be used to enter corrections from paper tape.

If the lines of the program on paper tape do not have line numbers, any of the following forms of the COPY command may be used to enter the program:

> COPY TEL TO line number:line number ↵

> COPY TEL TO line number(increment) line number ↵

> COPY TEL TO line number(increment) ↵

After all three forms, the system will type

NO LINE MAY HAVE A LINE NUMBER.
BEGIN INPUT.

Now turn on the paper tape reader. After the tape is read, turn off the reader and type a Control D. Statements are assigned line numbers as specified in the COPY command used.[1]

These two commands should not be used for terminal input because syntactically incorrect statements are not printed until Control D is pressed, and are then discarded, thus preventing immediate keyboard correction.

STORING A PROGRAM ON PAPER TAPE

A program entered into CCS may be stored on paper tape either with or without line numbers.

To store the entire program with line numbers, use either

> LIST TO TEL — *To store a formatted listing of the program on paper tape.*

or

> FAST TO TEL — *To store a quick listing of the program on paper tape.*

Then turn on the paper tape punch and type a Carriage Return. The program will be punched on paper tape and printed on the terminal. After punching is completed, turn the tape punch off.

The commands LIST and FAST may also be used to store part of the program on paper tape using the usual forms of these commands with TEL specified as the output file.

1 - The rules for assigning line numbers during COPY are like those for the various forms of ENTER. For details, see *The COPY Command*, Page 111.

Examples

>LIST 10:55 TO TEL

>FAST *1:*100,500:$-1 TO TEL

Programs stored on paper tape with LIST or FAST may be read back into CCS using the LOAD TEL command discussed in the preceding section.

To store all or part of the program on paper tape without line numbers, use the following form of the COPY command:

>COPY line list TO TEL

After typing the command, turn on the paper tape punch and type a Carriage Return. The program will be punched on paper tape and printed on the terminal without line numbers. After the program is punched, turn the tape punch off.

This form of COPY may be used to save either all or part of the program depending on which lines are specified in the line list. The line list may contain any valid addresses of single lines or line ranges, separated by commas.

Examples

>COPY 10:500 TO TEL

>COPY 10,15:20,$ TO TEL

>COPY *16:.-5 TO TEL

Programs stored on paper tape with COPY may be read back into CCS using the COPY command in the forms discussed in the preceding section.

STORING A PROGRAM ON A DISK FILE: SAVE

The command SAVE allows the user to store his program on a disk file, either as

- the symbolic source text alone, *or*
- the symbolic text and the binary object program.

The symbolic version offers more economical use of disk storage; the symbolic text and binary program together allow more rapid loading and execution by means of the LINK command.[1]

The symbolic text stored on a file with SAVE always includes all program lines, together with their line numbers.

The general form of SAVE is

>SAVE file name ↵

After typing the Carriage Return following the command, the system responds with

TEXT ONLY?

The user may now type either a Y to create a symbolic program file, or an N to create a binary program file. He then types a Carriage Return. If the response is Y, CCS types

NEW FILE *If there is no file with the chosen name in the user's directory.*

or

OLD FILE *If there is a file with the chosen name in the user's directory.*

In either case, the user may type a Carriage Return to confirm the command, or an ALT MODE or ESCAPE to abort it.

Example

>SAVE PROG ↵
TEXT ONLY?Y ↵
OLD FILE⊕ *ALT MODE aborts this command, so no new text is saved on the old file.*

>SAVE NPROG ↵
TEXT ONLY?Y ↵
NEW FILE ↵ *This command is confirmed; CCS prints OK. and saves a symbolic version of the program on the file NPROG.*

OK.
>

If the symbolic option is taken and the OLD/NEW FILE response is confirmed, CCS saves the program text on the specified file. However, if the binary option is taken, the OLD/NEW FILE message is not typed until CCS checks the program for faulty structure (such as a missing END statement, duplicate declaration statements, etc.). If the program structure is faulty, a diagnostic is typed. Otherwise, the OLD/NEW FILE message is typed, the user confirms it, and the binary, compiled program is stored on the file, followed by the symbolic text.

Example: Creating A Binary Program File

>10 ACCEPT A,B,C ↵
>20 D=A+B*C ↵
>30 DISPLAY D ↵
>SAVE APROG ↵ *The user writes a program and attempts to save a binary version of it on the file named APROG.*

TEXT ONLY?N ↵

MISSING FINAL ↵
"END" *CCS detects a structural error and types the appropriate reference line in the program. The user corrects the error and attempts the SAVE again.*
30 DISPLAY D
>40 END ↵
>SAVE APROG ↵

1 - See *The LINK Command*, Page 109.

```
TEXT ONLY?N ↵
 NEW FILE ↵
OK.
```

This time, the program structure is sound. CCS saves the compiled program followed by the symbolic program on the file named APROG.

```
>
```

Symbolic program files may be entered into CCS using either the LOAD or the MERGE command, discussed on Page 109. They are much slower to enter and execute than binary program files since CCS must compile the symbolic text each time the program is run. However, they do have the following advantages:

- Symbolic program files use less storage space than binary program files.
- Program statements on symbolic program files may be merged with program statements already entered into CCS by using the MERGE command; the symbolics stored on binary program files may not be so merged.
- The user may type only part of his program and then save it on a symbolic file to be retrieved and completed later. This is not possible with binary program files, since structurally incomplete programs abort the SAVE command with the N option, as discussed above.

The compiled program stored on a binary program file may be loaded and executed extremely rapidly by using either the SUPER FORTRAN link statement[1] or the CCS LINK command.[2]

The symbolic text stored at the end of the file may also be loaded into CCS using the LOAD command; however, MERGE will not enter the symbolics of a binary program file.

The primary advantages of binary program files are that they:

- May be loaded and executed extremely rapidly.
- May be linked to other program files, thus allowing a larger total program size.[1]
- Help provide complete program security, since linked programs may not be listed.[2]

Binary files are thus very useful for debugged production programs that are to be run quite often, while symbolic files are more useful during program development and for smaller programs that are to be run only occasionally.

RETRIEVING A STORED PROGRAM

A program stored on a file with the SAVE command may be retrieved at any time by using the appropriate command from the following table:

SUMMARY OF COMMANDS FOR RETRIEVING A SAVED PROGRAM

Command Model	Effect On Symbolic Program File	Effect On Binary Program File	Remarks
LOAD file name ↵	Clears current program (and all other information) from CCS, then loads program from specified file.	Clears CCS, then loads symbolic text only.	Each line in the file must have a line number; the lines must be stored in line number order.
MERGE file name ↵	Merges by line number the contents of the specified file with any current program. Lines from the file replace any current program lines with the same line number.	May not be used on binary program files.	Each line must have a line number; the lines may be stored in any order. Much slower than LOAD.
LINK file name ↵	May not be used on symbolic program files.	Loads binary version of program and executes it immediately. Does not load symbolic text.	Current program text is retained. COMMON area of binary program is initialized to zero.

1 - See *Program Linking*, Page 98.

2 - See *The LINK Command*, Page 109.

The LOAD Command

This command allows rapid loading of symbolic program text from either a symbolic program file or a binary file. It has the form

>LOAD file name ↵

After the Carriage Return is typed, CCS prints OK. It then completely clears any current program (including variable values and all other information) and loads the program text from the specified file. Each line in the file must have a line number; the lines in the file must be stored in ascending line number order.

During program loading, syntactically incorrect statements are printed with error messages and then discarded. These statements should be entered correctly before the program is executed.

An ALT MODE or ESCAPE during LOAD aborts the command. No information is retained.

Example

```
>LOAD TRI ↵
OK.
>
```

After a program is loaded, it is ready for execution or modification. If the user modifies his program and wishes to keep the modifications, he should save the program again under either the same file name or a different file name.

Since LOAD completely clears any current program, it can be used for initial program entry only. The MERGE command may be used to merge program statements from a file with the current program statements.

The MERGE Command

MERGE allows the user to insert corrections and additions from a file into a program currently in CCS. It should not be used for initial program loading since it is much slower than LOAD. The form of this command is

>MERGE file name ↵

After the Carriage Return, CCS loads the program from the specified file. Each line in the file must have a line number; however, the lines may be stored in any order. If any line on the file has the same line number as a line in the current program, it replaces the line in the current program.

Just as during LOAD, syntactically incorrect statements are printed with error messages and then discarded. These should be entered correctly before the program is executed.

NOTE: MERGE may not be used to merge the symbolic text from a binary program file (created with the N option to the SAVE command).

The following example uses the LOAD command to enter a program stored on the file PROG, and then uses the MERGE command to merge corrections from the file CORR. The file PROG contains the text

```
1 ACCEPT X,Y
2 Z=SQRT(X**2+Y)
3 THETA=ATAN2(X,Y)
4 END
```

and the file CORR contains

```
3.5 DISPLAY X,Y,Z,THETA
2 Z=SQRT(X**2+Y**2)
```

thus,

```
>LOAD PROG ↵
OK.
>MERGE CORR ↵
>LIST ↵
   1          ACCEPT   X,Y
   2          Z=SQRT(X**2+Y**2)
                  Line 2 from CORR replaced
                  line 2 from PROG.
   3          THETA=ATAN2(X,Y)
   3.5        DISPLAY X,Y,Z,THETA
                  Line 3.5 from CORR was
                  merged between lines 3 and 4
                  from PROG.
   4          END
>
```

Note that the line numbers in the merged file CORR are not ordered. This file could not be entered with LOAD.

THE LINK COMMAND

The CCS command

>LINK file name ↵

has very nearly the same effect as the SUPER FORTRAN statement LINK. It loads the binary program from the specified file (created with the N option to the SAVE command) and begins executing the program immediately. The text of any current program is retained; the text on the file is **not** brought into CCS. When LINK is executed, the binary program

is loaded with its COMMON area and local storage automatically initialized to zero.

Example

A binary program on the file named CROOT is rapidly loaded and executed as follows:

```
>LINK CROOT ↵
TYPE THE NUMBER: 8.352 ↵
CUBE ROOT:       2.028913

>
```

Any interruptions of the linked program (including the end of the program) return the user to the command mode indicated by the >. At this point, all CCS commands which operate on the program text apply to the text of the program present before the LINK was executed. In particular,

- RUN runs the previous program only; to rerun the linked program, another LINK command (or a SUPER FORTRAN LINK statement) must be executed.
- LIST, FAST, and COPY will list the previous program only. SAVE saves the previous program only. This ensures program security, as discussed below.

No CCS commands which apply to the running program are allowed after a program executed with the LINK command is interrupted, except for NEXT and CONTINUE, which apply to the binary program. In particular, no direct statements or breakpoints will be accepted.[1]

Example

```
>LOAD CROOT ↵                          Loads symbolic program on file CROOT.
OK.
>RUN ↵                                 Executes this program.
TYPE THE NUMBER: 7777 ↵
CUBE ROOT:       19.812413

(@7 )>LINK CHECK ↵                     Loads binary program on file CHECK.
ROOT TO BE CHECKED IS 19.812413 ↵
   19.812413 CUBED = 7777.000

>LIST ↵
   1            *CROOT:  CUBE ROOT PROGRAM      Symbolics of CROOT
   2            ACCEPT "TYPE THE NUMBER: ",X    are retained. They are
   2.5          IF (X .EQ. 0) (APPROX1=0;GO TO 20)  listed with LIST, and ....
   3            APPROX0=X/3
   4      10    APPROX1=(2*APPROX0**3+X) /
                 (3*APPROX0**2)
   5            IF (ABS(APPROX1-APPROX0) .GE. 1E-8)
                 (APPROX0=APPROX1;GO TO 10)
   6      20    WRITE (1,100) APPROX1
   6.5    100   FORMAT("CUBE ROOT: ",F11.6)
   7            END
>RUN ↵                                                 ... executed with RUN.
TYPE THE NUMBER: -45.9 ↵
CUBE ROOT:       -3.580450

(@7 )>LINK CHECK ↵                     LINK must be used to
ROOT TO BE CHECKED IS -3.580450 ↵      execute CHECK again.
   -3.580450 CUBED = -45.900

>
```

1 - See *Executing A Program*, Page 113, and *Program Control And Debugging Aids*, Page 116, for details on the features discussed in this paragraph.

Since linked binary program files cannot be listed in CCS, the user may ensure security for files he wishes to share by storing them with the N option to the SAVE command. To prevent other users from knowing the name of the link file (so that they cannot LOAD the symbolics and then list them), the link file may be shared by running it from a remote, proprietary command file, which can never be copied or listed. (*See the Tymshare EXECUTIVE Manual, Reference Series, for details.)*

In the rest of this section we discuss two commands, COPY and MOVE, which may be used for program storage and retrieval as well as for manipulating lines within CCS.

THE COPY COMMAND

This command allows the user to copy all or part of a program. It may be used to

- Enter program lines from a file, provided the lines on the file do not have line numbers.
- Store program lines on a file without line numbers.
- Enter program lines from paper tape, and store program lines on paper tape (as was discussed on Pages 106 and 107).
- Copy one file to another file, no matter what the file type.
- Copy lines from one part of a program to another.

The general form of the command is

>COPY source TO destination ↵

It copies the **source** to the **destination**; the source is never deleted by this command.

The **source** can be either a file name (including TEL for the terminal) or a line list (a list of one or more line or range addresses, separated by commas). If the source is a line list, the destination may be omitted to list the source lines on the terminal without line numbers. For example,

>COPY *1,50:100,$ ↵

lists the first line in the program, lines 50 through 100, and the last line in the program on the terminal without line numbers. This command is equivalent to

>COPY *1,50:100,$ TO TEL ↵

The **destination** can be either a file name (including TEL) or one of the following:

line number:line number

line number(increment)line number

line number(increment)

The form

>COPY source TO line number:line number ↵

first deletes the lines in the destination range and then inserts a copy of the source lines into that range, numbered beginning with the first line number specified and choosing as increment the first of 1, .1, .01, and .001 which will fit the specified range. If the source is a file, no line in the file may have a line number.

Example

Suppose the file AA contains the lines

```
ACCEPT X,Y
Z=ABS(X-Y)
DISPLAY Z
```

then

```
>COPY AA TO 50:70 ↵
OK.
>FAST ↵
50 ACCEPT X,Y
51 Z=ABS(X-Y)
52 DISPLAY Z
>
```

The form

>COPY source TO line number(increment)line number ↵

is the same as the previous form except that the increment is specified by the user. For example, using the file AA above.

```
>COPY AA TO 50(10)70 ↵
OK.
>FAST ↵
50 ACCEPT X,Y
60 Z=ABS(X-Y)
70 DISPLAY Z
```

Note that this command deletes any lines between 50 and 70 before inserting the lines from AA.

The form

>COPY source TO line number(increment) ↵

inserts a copy of the source lines into the program, numbered beginning with the line number specified in the destination, in the increment specified. If the source is a file, no line on the file may have a line number.

Example

```
>COPY LINES TO TEL ↵
Z1=ABS(X+Y)
Z2=ABS(X-Y)

>FAST ↵
10 ACCEPT X,Y
50 DISPLAY Z1,Z2
>COPY LINES TO 20(10) ↵
OK.
>FAST ↵
10 ACCEPT X,Y
20 Z1=ABS(X+Y)
30 Z2=ABS(X-Y)
50 DISPLAY Z1,Z2
>
```

When this form is used, the user is protected from deleting or interleaving program lines. If a line being copied would be assigned a line number that would cause deletion or interleaving, a message is printed and the command is terminated. All lines up to, but not including, the offending line number are copied. Thus,

```
>FAST ↵
10 ACCEPT X,Y
50 DISPLAY Z1,Z2
>COPY LINES TO 35(15) ↵
OK.

TOO MANY LINES FOR SPECIFIED RANGE

Z2=ABS(X-Y)
>FAST ↵
10 ACCEPT X,Y
35 Z1=ABS(X+Y)
50 DISPLAY Z1,Z2
>
```

The second line in LINES is not copied since it would be assigned line number 50.

NOTE: The CCS command

COPY file name TO file name ↵

is equivalent to the EXECUTIVE command of the same form. Any file name (including TEL) may be used as source or destination; the files specified need not contain SUPER FORTRAN statements. The source file may contain SUPER FORTRAN statements with line numbers; if it does, the statements will be copied to the destination file with line numbers.

Additional Examples Using COPY

>COPY 10:25 TO 75:100 ↵

Deletes lines 75 through 100 and then inserts lines 10 through 25 renumbered in the range 75 to 100. Lines 10 to 25 are undisturbed.

>COPY 1:100 TO EE ↵

Copies lines 1 through 100 to file EE without line numbers. Lines 1 through 100 are undisturbed.

>COPY EE TO 200:300 ↵

Deletes lines 200 through 300, then inserts the contents of the file EE numbered in the range 200 to 300. The lines on file EE must not have line numbers.

>COPY A TO B ↵

Copies the contents of the file A to the file B. Files A and B may be any type.

>COPY TEL TO 40:50 ↵

Deletes lines 40 through 50 and inserts the lines typed on the terminal (or read from paper tape) during this command, numbered in the range 40 to 50. Statements typed on the terminal must not have line numbers. A D^C terminates the command.

>COPY TEL TO TEXT ↵

Copies the lines typed on the terminal (or read from paper tape) during this command to the file named TEXT. Lines copied may comprise arbitrary text. A D^C terminates the command.

>COPY ABC TO 100(5)200 ↵

Deletes lines 100 through 200 and inserts the lines on the file ABC numbered in the range 100 to 200 beginning with line number 100 in increments of 5. The lines in ABC must not have line numbers.

>COPY 3:10,*500:$ TO PROG ↵

Copies lines numbered 3 through 10, and the 500th line in the program through the last line, to the file named PROG without line numbers.

>COPY @T2 TO TEL ↵

Copies contents of the file named @T2 to the terminal. @T2 may be any symbolic file. If the file contains a SUPER FORTRAN program, the entire program, **including** *any line numbers, is printed.*

THE MOVE COMMAND

Program lines may be moved to a file or line range using the MOVE command. The general form of this command is

>MOVE line list TO destination ↵

The **destination** can be either a file name (including TEL for the terminal) or one of the following:

line number:line number

line number(increment)line number

line number(increment)

The source line list is deleted when MOVE is executed. Otherwise, MOVE works exactly like COPY.

Examples

>MOVE 1,20:50 TO SAVE ↵

Moves lines numbered 1 and 20 through 50 to the file named SAVE without line numbers, then deletes them from the program.

>MOVE *1:$-50 TO A ↵

Moves the first line in the program through line $-50 to the file named A without line numbers, then deletes them from the program.

>MOVE 1:10 TO 150:160 ↵

Deletes lines 150 through 160, inserts lines 1 through 10 into the range 150:160 (renumbered beginning with 150 in increments of the first of 1, .1, .01, and .001 that fits this range), then deletes lines 1 through 10.

>MOVE 1:10 TO 150(2)170 ↵

Deletes lines 150 through 170, inserts lines 1 through 10 into the range 150 to 170 (renumbered beginning with 150 in increments of 2), then deletes lines 1 through 10.

EXECUTING A PROGRAM

To begin execution of a symbolic program after it is entered, use the command

>EXECUTE ↵

or

>RUN ↵

Program execution will start with the first executable statement.

When a free form terminal input statement is executed, a bell will ring, signaling that data is to be typed from the terminal. One value must be typed for each variable in the input list.

Execution will terminate at the end of a program, when a PAUSE or STOP statement is executed, or when an error is detected. In addition, the user may interrupt execution by pressing the ALT MODE or ESCAPE key, or by setting breakpoints with the BREAK command before executing the program. These features are discussed in detail in *Program Control And Debugging Aids*, Page 116. Whenever execution is terminated, CCS prints a message (such as an error diagnostic when an error is encountered, or PAUSE when a PAUSE statement is encountered). It then prints either

>

or

(@line number)>

or

line number >

SUMMARY OF COMMAND MODELS FOR PROGRAM FILE STORAGE AND RETRIEVAL

Kind Of Program File	Can Be Created With	Can Be Retrieved With	Remarks
Symbolic file with line numbers	SAVE file name ↵ TEXT ONLY?Y ↵	LOAD file name ↵	LOAD deletes previous program; statements must be in line number order.
	FAST -[line list]- TO file name ↵ (quick listing) LIST-[line list]-TO file name ↵ (formatted listing)	MERGE file name ↵	MERGE merges statements on file with existing program; statements may be in any order. LOAD is much faster than MERGE.
Symbolic file without line numbers	COPY line list TO file name ↵ MOVE line list TO file name ↵	COPY file name TO line number range ↵	MOVE deletes the source. COPY never deletes the source. An increment may be specified in the destination range; for example, COPY Q TO 10(5)100 ↵ MOVE 20:50 TO 1(1) ↵
LINK or BINARY program file	SAVE file name ↵ TEXT ONLY?N ↵ *Saves binary version of program followed by symbolic text, with line numbers.*	LINK file name ↵ LOAD file name ↵	LINK brings in binary version of program and starts execution; text of any previous program is retained. LOAD brings in symbolics of program just as with a symbolic file.

NOTE: Example of a line list: *1:*50,100:$-10,$

The line list in FAST and LIST is optional; if omitted, the entire program is stored on the file.

where the line number typed in the last two cases indicates the line in which program execution was interrupted.

If only a > is typed after an interruption, the user should correct any errors and re-execute the program from the beginning (by typing RUN or EXECUTE). This situation occurs after certain error messages.

Example 1

```
>1 ACCEPT A,B↵
>2 C=A+B↵
>3 DISPLAY C↵
>RUN↵

MISSING FINAL "END"
3 DISPLAY C
>4 END↵
>RUN↵
4,8↵
 12

(@4 )>
```

A two-line error diagnostic followed by a > is typed. The user corrects the error by adding line 4 to the program. He then reruns the program, which now runs correctly.

Example 2

```
>LIST↵
   1            ACCEPT I
   2            J=2*I
   3            K=J**2
   4            L=J+K
   5            WRITE (1,100) K,L
   6      100   FORMAT (2N5)
   7            END
>RUN↵
11↵

FORMAT ERROR:
(2N5)
ILLEGAL CHARACTER IN FORMAT
5 WRITE (1,100) K,L
5>6 100 FORMAT (2I5)↵
>RUN↵
11↵
  484  506

(@7 )>
```

The computer prints an error diagnostic with a reference line from the program.

Here the error is corrected by retyping the incorrect line.

If (@line number)> is typed after an interruption, the user may enter SUPER FORTRAN language statements preceded by an @ for immediate execution. Such statements are called direct statements[1]. This situation occurs when the end of the program or a STOP statement is encountered. In the previous example, the user could print the value of J after execution of his program as follows:

```
>RUN↵
11↵
  484  506

(@7 )>@DISPLAY J↵
 22

(@7 )>
```

If any changes in the program text are made (such as changing a statement in the program, inserting statements, or deleting statements), direct statements may no longer be entered. CCS will stop printing (@line number)> and print only a > if any such change is made.

If CCS prints

line number >

after a program interruption, the user may not only enter direct statements but also may continue the program from the point of interruption by typing CONTINUE. This feature allows the user to change his program temporarily and continue execution without having to re-execute his program from the beginning. This is extremely useful in debugging programs.

Example

```
>FAST↵
2 DOUBLE PRECISION X
3 Y=1/X
4 DISPLAY "Y=",Y
4.1 DISPLAY "THE REST IS EXECUTED"
5 END
>RUN↵

ATTEMPT TO DIVIDE
        BY ZERO
3 Y=1/X
3 >@DISPLAY X↵
 0

3 >@X=3.↵

3 >CONTINUE↵
Y=        .33333333
THE REST IS EXECUTED

(@5 )>
```

Execution is halted at line 3 when the computer attempts to divide by zero. The user types a direct statement for immediate execution.

The user gives X a value other than zero by typing another direct statement. Program execution resumes at line 3 when the user types CONTINUE.

1 - See *Immediate Execution Of Statements: Direct Statements*, Page 118, for further details on this feature.

The direct statement @X=3. was executed and discarded. If the user wishes to make this a permanent change in the program, he must now type

2 X=3. ↵

and save the program on a file.

If the user makes changes in a statement, inserts a new statement, or deletes a statement, he will not be able to continue execution using CONTINUE, nor will he be able to execute direct statements. Execution must be started from the beginning. When any such changes to the program text are made, CCS stops printing a line number followed by a > and prints only a >.

If either **line number** > or **(@line number)**> is printed when a program is interrupted, the user may refer to parts of the program other than the current program block with direct statements. This is accomplished using the AT command. (See *Referring To Different Parts Of A Program: AT*, Page 119.)

To execute a binary program file, use the CCS LINK command (discussed on Page 109 or the SUPER FORTRAN LINK statement (discussed in *Program Linking*, Page 98). These commands both load and execute a binary program file. Program interruptions result in different command prompts.

When execution of a binary program is interrupted, CCS prints one of the following prompts: (Recall that direct statements cannot be entered when a running binary program is interrupted.)

Prompt	Meaning
>	*Neither direct statements nor CONTINUE may be typed.*
*n >	*Execution of binary program may be continued at the nth line in the program.*

After the linked program is interrupted

- Commands operating on the program text (such as LIST and RUN) apply to any symbolic program present before the LINK.
- Commands applying to the running program are illegal, except for NEXT and CONTINUE.

RETURNING TO THE EXECUTIVE: QUIT

The user may return to the EXECUTIVE by using the command

```
> QUIT ↵
-
```

The dash typed by the computer means that the user is in the EXECUTIVE.

PROGRAM CONTROL AND DEBUGGING AIDS

PROGRAM INTERRUPTION

Execution of a running program can be interrupted by the various situations summarized in the table below. After a program has been interrupted, certain debugging aids (such as the ability to execute direct statements or to continue execution from the point of interruption) are available. The kinds of debugging features available after an interruption depend on the type of interruption; the user can determine which features are available by the kind of command prompt CCS prints after an interruption. This will always be one of the prompts discussed in *Executing A Program*, Page 113.

SUMMARY OF PROGRAM INTERRUPTIONS

l = line number of line in which interruption occurred.
**n* indicates that interruption occurred in nth line of program.

Type Of Interruption	Message And Prompt Printed		Direct Statements?		Continuable?
	If Program Is Symbolic	If Program Is Binary	Symbolic	Binary	
ALT MODE/ ESCAPE	INTERRUPT *l* >	INTERRUPT **n* >	Yes	No	Yes
Breakpoint	BREAK *l* >	Breakpoints not allowed in binary program	Yes	No	Yes
End of Program	(@*l*)>	>	Yes	No	No
Error *Two alternatives possible*	Diagnostic > Diagnostic *l* >	Diagnostic > Diagnostic **n* >	No Yes	No No	No Yes
PAUSE Statement in Program	comment[1] *l* >	comment[1] **n* >	Yes	No	Yes
STOP Statement in Program	comment[1] *(@l)*>	comment[1] >	Yes	No	No
QUIT Statement in Program	comment[1] –	comment[1] –	Not applicable; user is now in EXECUTIVE.		No

Whenever a program has been interrupted, CCS commands such as LIST, DELETE, RUN, etc., may be executed. However, if any changes in the program text are made after an interruption, the program is not continuable; nor may direct statements then be executed.

Notice that whenever a symbolic program is continuable, direct statements may be executed. However, the converse is not true; when a STOP statement or the end of the program is encountered, direct statements may be executed but the program is not continuable. Direct statements may not be entered after interrupting a binary program.

BREAKPOINTS

Breakpoints can be set in a program using the command

>SET list of breakpoints ↵

or the equivalent command

>BREAK list of breakpoints ↵

The list of breakpoints consists of the line numbers and/or line number ranges at which breakpoints are to be set, separated by commas. For example,

>SET 5,8,10:14 ↵ *(or* **BREAK 5,8,10:14** ↵*)*

sets a breakpoint at line 5, line 8, and at every line between 10 and 14, inclusive. There is no limit to the number of breakpoints that may be set.

When a breakpoint is encountered during program execution, the program is interrupted just **before** executing the statement(s) in the line where the breakpoint was set. CCS prints

BREAK
l >

1 - Optional comment specified by user in statement.

where / is the number of the line where the breakpoint was set. Now, direct statements may be entered; execution may be continued with the CONTINUE command.

To clear breakpoints, use the command

>RESET list of breakpoints ↵

For example, if the BREAK command above has been executed, the command

>RESET 5,10:13 ↵

clears the breakpoints at lines 5 and 10 through 13, having active breakpoints at lines 8 and 14.

The command

>RESET ↵

clears all active breakpoints.

The command

>BREAK ↵

lists the line numbers of all active breakpoints. For example,

```
>SET 5,10,100:150 ↵
>BREAK ↵
100 : 150
10 : 10
5 : 5
>
```

CONTINUING PROGRAM EXECUTION: CONTINUE

Whenever CCS prints

line number >

after an interruption of a running program, execution of the program may be continued from the point of interruption using the command

line number >CONTINUE ↵

The line number printed by CCS, called the CONTINUE point, is the number of the line in which the program was interrupted; CONTINUE causes execution to resume beginning with the following line.

A program is continuable after certain kinds of error interrupts, after a PAUSE, when a breakpoint is reached, or when the user interrupts the program by pressing ALT MODE/ESCAPE once. Whenever a program is continuable, direct statements may be executed.

```
>FAST ↵
1  DISPLAY "DEMONSTRATING CONTINUE."
2  PAUSE "THIS IS A PAUSE"
3  DISPLAY "THE REST IS EXECUTED."
4  END
>RUN ↵
DEMONSTRATING CONTINUE.

THIS IS A PAUSE
2 >@DISPLAY "NOW WE CONTINUE..." ↵
                          Note the direct statement.
NOW WE CONTINUE...

2 >CONTINUE ↵
THE REST IS EXECUTED.

(@4  )>
```

IMMEDIATE EXECUTION OF STATEMENTS: DIRECT STATEMENTS

Whenever CCS prints either

(@line number)>

or

line number >

SUPER FORTRAN language statements may be entered for immediate execution. The statements entered must be executable statements and must be preceded by an @. As soon as a Carriage Return following the direct statement is typed, the statement is compiled and, if syntactically correct, executed. Then the statement is discarded.

A direct statement is considered to be temporarily inserted into the current running program at the line number printed in the command prompt (that is, when CCS prints **line number >**). This point is called the CONTINUE point. When a direct statement is executed, variables used in the statement assume values dependent upon the location of the CONTINUE point in the program. For example, for the program set up in part as

```
10 ACCEPT X
.
.
.
25 Y=X↑2
25.1 PAUSE
25.2 DISPLAY "EXECUTION CONTINUES"
.
.
.
```

```
47 Y=Y+6
48 STOP
```

the following may occur:

```
> RUN ↵
12. ↵
```

The value for X is accepted.

```
25.1 >@DISPLAY Y ↵
 144
```

CONTINUE point is 25.1, so $Y=X^2=144$.

```
25.1 >CONTINUE ↵
EXECUTION CONTINUES
```

```
(@48 )>@DISPLAY Y ↵
 150
```

Now, all the statements have been executed, so Y=Y+6=150.

```
(@48 )>
```

Direct statements are particularly useful debugging aids. For example, after a program interruption they may be used to

- Print the current values of variables, as in

  `15 >@DISPLAY A,B,C(5) ↵`

- Assign new values to variables, as in

  `35.1 >@ALPHA = 3.11 ↵`

- Skip parts of a program, as in

  `(@100 )>@GO TO 40 ↵`

  This direct statement causes execution to be resumed starting at the statement labelled 40.

REFERRING TO DIFFERENT PARTS OF A PROGRAM: AT

In the preceding discussion of direct statements, we defined the @ point as the point at which a direct statement is considered to be temporarily inserted into the program. Usually, the @ point is the point at which the program was interrupted, and is equal to the CONTINUE point if the program is continuable. However, the user may change the @ point at any time direct statements are executable by typing

AT line number ↵

This command changes the @ point (but not the CONTINUE point) to the line number typed. It is useful when the user wishes to refer to variables not defined in the current program block with direct statements. (A program block is a group of program statements constituting either the main program or a subprogram.)

Example

```
> LIST ↵
   2          DIMENSION R(10)
   3          ACCEPT R
   4          RM=MEAN(R,10)
   5          PAUSE
   6          DISPLAY "MEAN=",RM
   7          END
   9          REAL FUNCTION MEAN(S,N)
   9.1        DIMENSION S(*)
  10          Z=0.
  11          DO 100 I=1,N
  12          Z=Z+S(I)
  13     100  CONTINUE
  14          MEAN=Z/N
  15          RETURN
  16          END
```

Lines 2-7 comprise one program block, the main program.

Lines 9-16 comprise another, the function subprogram.

```
>RUN ↵
2.2,2.15,1.98,3.001,1.99,2.14,2.12,2.67,2.55,2.71 ↵

5 >@DISPLAY "MEAN IS",RM ↵          Here, @ point = CONTINUE
MEAN IS     2.3511                  point=5.

5 >@DISPLAY "SUM IS",Z ↵

"Z" UNDEFINED                       Z is not defined in the main
5 >AT 15 ↵                          program.
(@15 )5 >@DISPLAY "SUM IS",Z ↵      Here, @ point is changed to
SUM IS      23.511                  15, a line in the function sub-
                                    program. Now Z is defined,
                                    and its current value is printed.
(@15 )5 >CONTINUE ↵                 CONTINUE resumes execu-
MEAN=       2.3511                  tion after line 5, not line 15.

(@7 )>
```

In this program, a PAUSE at line 5 occurs after the function subprogram MEAN is called. During this pause, the user wishes to print both the value of the mean of the ten elements of R and the value of the sum of these elements. However, since the @ point is in the main program (line 5), the sum Z, a variable local to the function subprogram, is not defined here. To print the value of Z using a direct statement, he must change the @ point to some point in the function subprogram. He does this using the command

```
>AT 15 ↵
```

CCS responds with

```
(@15 )5 >
```

indicating that the @ point is now 15, but the CONTINUE point is still 5.

Notice that any line number in the subprogram could have been used in place of 15 in the AT command above. The value of a variable printed by a direct statement is **always** the value most recently assigned to the variable when the line at the @ point was executed. Thus, in the above example, the user could have set the @ point to 10 (where Z was set to zero) and Z would still have the value 23.511 as in the following:

```
5 >AT 10 ↵
(@10 )5 >@ DISPLAY "SUM IS", Z ↵
SUM IS      23.511
(@10 )5 >
```

The purpose of changing the @ point is to refer to variables not defined in the current program block, and **not** to restore values of variables assigned earlier in the program and subsequently assigned new values. The latter cannot be done.

Certain restrictions on the use of direct statements apply when the @ point has been changed using AT:

- Direct statements causing transfer of control to a labelled statement (such as @GO TO 20) may not be executed.
- Dummy arguments of a subprogram may not be displayed.

NOTE: The @ point may be restored to the CONTINUE point.

PARTIAL PROGRAM EXECUTION

Step Execution: NEXT

Whenever program execution is continuable, the user may single step through his program, executing one statement at a time. This is accomplished by typing

```
NEXT ↵
```

to execute the next statement. After a NEXT command is executed, the program is continuable at the line following the line just executed.

Example

```
>FAST ↵
1 DISPLAY "FIRST, A BREAKPOINT..."
2 DISPLAY "LINE 2 IS EXECUTED."
3 DISPLAY "NOW, LINE 3 IS EXECUTED."
4 DISPLAY "THE REST IS EXECUTED."
5 END
>SET 2 ↵          A breakpoint is set at line 2.
>RUN ↵
FIRST, A BREAKPOINT...

BREAK
2 >NEXT ↵
LINE 2 IS EXECUTED.

3 >NEXT ↵
NOW, LINE 3 IS EXECUTED.

4 >CONTINUE ↵
THE REST IS EXECUTED.

(@5 )>
```

Executing A Specified Range Of Lines

We have already seen a method of executing a specified range of lines; that is, setting a breakpoint and then typing RUN or CONTINUE. For example,

```
>RESET ↵
>SET 25.1 ↵          Executes all lines in the
>RUN ↵               range *1-25.1-1, inclusive.

BREAK
25.1 >RESET ↵
25.1 >SET 100 ↵      Executes all lines in the
25.1 >CONTINUE ↵     range 25.1:100-1, inclusive.
```

However, CCS provides special forms of RUN and CONTINUE which make procedures such as the above more convenient.

The command

>RUN TO list of breakpoints ↵

is equivalent to

>RESET ↵
>SET list of breakpoints ↵ (*or* **BREAK list of breakpoints ↵**)
>RUN ↵

For example,

>RUN TO 5,10,50:70 ↵

is equivalent to

>RESET ↵
>SET 5,10,50:70 ↵
>RUN ↵

Similarly, the command

line number >CONTINUE TO list of breakpoints ↵

is equivalent to

line number >RESET ↵
line number >SET list of breakpoints ↵
line number >CONTINUE ↵

Thus, the example given at the beginning of this section could also be accomplished as follows:

```
>RUN TO 25.1 ↵
BREAK.

25.1 >CONTINUE TO 100 ↵
```

LOCATING LABEL AND VARIABLE REFERENCES AND DEFINITIONS

The REFERENCES Command

The command

>REFERENCES [**variable name** / *or* / **statement label**] **line list ↵**

prints lines with their line numbers which contain the variable name or statement label specified. The line list is optional. If included, only occurrences of the variable or label in the lines specified are printed. If omitted, occurrences of the variables or label throughout the program are printed. In both cases, only nondeclarative statements are printed; declarative statements (such as DIMENSION statements, type declaration statements, subroutine declarations, etc.) may be printed using the DEFINITIONS command discussed below.

Examples

>REFERENCES 30 ↵

Prints the statement labelled 30 and all nondeclarative statements in the program which refer to the statement labelled 30 (such as GO TO 30 and IF (A) 10,20,30).

>REFERENCES Z 10:50 ↵

Prints all nondeclarative statements containing the variable name Z in the line range 10:50.

>REFERENCES X *1:*50,200:400 ↵

*Prints all nondeclarative statements containing the variable name X which are found in either of the ranges *1:*50 or 200:400.*

The DEFINITIONS Command

The command

>DEFINITIONS [variable name / *or* / statement label] line list ↵

prints declarative statements (with line numbers) which contain the variable name or statement label specified. The line list is optional. If included, only declarative occurrences of the variable or label in the lines specified are printed. If omitted, all declarative occurrences of the variable or label throughout the program are printed.

Examples

>DEFINITIONS Z ↵

Prints all declarative occurrences of the variable Z in the program, such as DIMENSION Z(10).

>DEFINITIONS SUM 10:50 ↵

Prints all declarative references to the variable SUM in the range 10:50.

>DEFINITIONS SUM 1:50,.:$ ↵

Prints all declarative references to SUM found in either of the ranges 1:50 or .:$.

>DEFINITIONS 20 ↵

Prints any statement labelled 20, such as
20 CONTINUE or
20 FORMAT (F16.8).

Examples Using REFERENCES And DEFINITIONS In Program Debugging

Example 1

Suppose that during the execution of a program, the user gets the error message

```
"Z" DECLARED TWICE
3.7 REAL X,Y,Z
```

If he then uses the DEFINITIONS command as follows,

```
>DEFINITONS Z ↵
1.4 SUBROUTINE S(L,M,Z)
3.2 INTEGER I,L,T,Z
3.7 REAL X,Y,Z
```

he can see immediately that he has declared Z to be both integer and real. He may then change either statement 3.2 or 3.7.

Suppose he decides that Z should be real and changes statement 3.2 to

```
3.2 INTEGER I,T,L
```

keeping statement 3.7. He then runs the program again and gets the error message

```
ACTUAL ARGUMENT DOES NOT MATCH
                        FORMAL PARAMETER
12.2 CALL S(5,7,I)
```

The integer argument I does not match the real variable Z.

Using the REFERENCE command, he checks all calls to subroutine S as follows:

```
>REFERENCES S ↵
9.1 CALL S(5,6,3.9)
12.2 CALL S(5,7,I)
14.5 CALL S(11,2,SQRT(17.4))
>
```

The user learns there is only one subroutine call conflicting with the definition of Z. He may now correct his error by declaring I real or by using a different argument, and then rerun the program.

Example 2

Suppose the user has a subroutine in his program called GEM and wishes to add a parameter to all references to the subroutine. He may locate these references by simply using the command

>REFERENCES GEM ↵

which prints all subroutine calls to GEM. He then may modify these lines.

Example 3

Suppose the user encounters the error message

```
SUBSCRIPT OUT OF RANGE
30 X=A(35)
```

He then uses the command

30 >DEFINITIONS A ↵

which prints

```
10 DIMENSION A(20)
```

He now knows he must change line 10 to reserve more space for A.

VERIFYING PROGRAM EXECUTABILITY

The commands CHECK and INITIALIZE allow easy verification of program executability without actually running the program.

The CHECK Command

The command

>CHECK ↵

tests the current program for structural soundness, detecting such errors as missing END statements in subprograms or the main program, labelling or declaration errors, calls to undefined subroutines, etc. If a structural error is found, an error message is printed and control is returned to the user. If not, CCS prints OK. and returns control to the user.

Example

```
>CHECK ↵

"100" DECLARED TWICE          An error is detected.
36.1 100 A=3
>36.1 200 A=3 ↵               The error is corrected.

>CHECK ↵
OK.                           Now the program is structurally sound.

>
```

The INITIALIZE Command

The command

>INITIALIZE ↵

first checks the structural soundness of the current program, just like the CHECK command. If a structural error is found, a message is printed and control is returned to the user. However, if a structural error is not found, this command allocates data storage and begins execution of the program, breaking just before the first executable statement in the program. CCS prints the line number of this statement and a >, indicating that direct statements may be entered and that the program is continuable.

Example

Using INITIALIZE instead of CHECK in the previous example yields the following:

```
>INITIALIZE ↵

"100" DECLARED TWICE
36.1 100 A=3
>36.1 200 A=3 ↵          When the program is structurally sound, INITIALIZE breaks before the first executable statement (line 1 in this case). CONTINUE would now start executing the program at line 1.

>INITIALIZE ↵
1 >
```

EXECUTING COMMAND FILES IN CCS

In Section 9 we discussed command files and how to execute them with the SUPER FORTRAN language statements OPEN and PAUSE.[1] Command files may also be executed in the command mode using the CCS command

>COMMANDS file name ↵

When this command is executed, commands are taken from the specified file immediately.

As an example, consider the file C1 which contains the commands

```
LIST ↵
RUN ↵
@DISPLAY "NOW, QUIT AND LOGOUT" ↵
QUIT ↵
LOGOUT ↵
```

and may be used as follows:

1 - See *Command Files,* Page 99, for information on the structure and creation of a command file.

```
                                                          Command Taken
                                                            From C1:
-SFORTRAN ↵
>LOAD SUM ↵
OK.
>COMMANDS C1 ↵
   1            SUM=0.                                        LIST
   2            ACCEPT "HOW MANY NUMBERS? ",N
   3            DISPLAY "ENTER NUMBERS TO BE ADDED:"
   4            DO 10 I=1,N
   5            ACCEPT X
   6            SUM=SUM+X
   7      10    CONTINUE
   8            DISPLAY "THE SUM IS",SUM
   9            END
HOW MANY NUMBERS? 5 ↵                                        RUN
ENTER NUMBERS TO BE ADDED:
23 ↵
17 ↵
98 ↵
-54 ↵
41 ↵
THE SUM IS      125

NOW QUIT AND LOGOUT                                          @ DISPLAY
                                                             "NOW QUIT AND LOGOUT"


                                                             QUIT
CPU TIME: 2 SECS.                                            LOGOUT
TERMINAL TIME: 0:2:05
PLEASE LOG IN:
```

Notice that the command file in the above example includes an EXECUTIVE command (LOGOUT) as well as CCS commands. Any commands that can be typed at the terminal can be included in a command file. This means that SUPER FORTRAN statements (with line numbers) may also be included in a command file, since they are simply implied ENTER commands.

If a file opened by COMMANDS is still open during program execution, it may be referred to as file * in the program being executed (see *Command Files*, Page 99).

PROGRAM EDITING

One of the most powerful features of Tymshare SUPER FORTRAN is the ability to edit both program statements and CCS commands without calling EDITOR. Editing features to insert, delete, and renumber program statements are available. In addition, commands and statements may be edited while they are being typed, and program statements may be changed using the editing control characters which SUPER FORTRAN shares with Tymshare's text editing language EDITOR.

INSERTING PROGRAM LINES

Program lines may be inserted into a program simply by entering the lines from the terminal with line numbers that fall between the line numbers of the

existing program. (See *Entering Statements From The Terminal*, Page 104.) For example, the statement

Z=X2+Y**2**

may be inserted into the program

```
1 ACCEPT X,Y
2 DISPLAY Z
```

by typing

> 1.3 Z=X2+Y**2** ↵

Statements may also be inserted from a file by using the MERGE or COPY commands, and may be copied or moved from one part of a program to another using the commands COPY and MOVE. These commands have all been discussed in *Entering, Storing, And Retrieving A Program*, Pages 104–113.

DELETING PROGRAM LINES

The commands DELETE and CLEAR are used to delete part or all of a program.

The DELETE Command

This command deletes any line or group of lines from a program. It has the general form

> DELETE line list ↵

where the line list consists of one or more line or range addresses separated by commas. DELETE deletes all lines addressed in the line list.

Examples

> DELETE 35 ↵ *Deletes the line numbered 35.*

> DELETE *12 ↵ *Deletes the 12th line in the program.*

> DELETE 20:$ ↵ *Deletes all lines from line 20 through the last line in the program.*

> DELETE .,200:200+10,*400 ↵ *Deletes the current line, the range of lines addressed by 200:200+10, and the 400th line in the program.*

In the last example above, the address *400 refers to the 400th line before execution of the DELETE command, **not** the 400th line after the lines addressed by . and 200:200+10 are deleted.

The CLEAR Command

This command erases the entire program. It has the form

> CLEAR ↵

After the Carriage Return is typed, CCS replies with

ERASE PROGRAM?

This question may be answered by typing a Y (for Yes) or an N (for No) followed by a Carriage Return. If a Y is typed, CCS prints OK. and erases the program. If an N is typed, the command is aborted.

Example

```
> CLEAR ↵
ERASE PROGRAM?Y ↵
OK.

>
```

CHANGING PROGRAM LINES

The easiest way to change a line in a program is to retype the line with the same line number when in the command mode. The line typed replaces the old line with the same line number. For example, line 2 in the following program

```
1 ACCEPT A,B,C
2 X=A+B
3 DISPLAY X,Y
```

may be changed using

> 2 X=A+B;Y=ABS(X2-C**2)** ↵

to produce the program

```
1 ACCEPT A,B,C
2 X=A+B;Y=ABS(X**2-C**2)
3 DISPLAY X,Y
```

Whole ranges of program lines may be changed using the following forms of ENTER (discussed in *Entering Statements From The Terminal*, Page 104).

> ENTER line number:line number ↵

> ENTER line number(increment)line number ↵

In addition, the control characters explained in the following section may be used for more efficient editing of program lines.

Editing With Control Characters

Although the user may change program lines by retyping the entire line that needs changing, SUPER FORTRAN provides control characters for more efficient editing of program lines. These control characters, which are the same as those available in Tymshare's EDITOR, are summarized in the table below. Certain of these control characters may also be used to edit commands or statements while they are being

typed, and to edit input data being typed from the terminal. Details on the use of the various characters follow the table.

EDITING CONTROL CHARACTERS		
Control Character	**Symbol Printed**	**Action**
For Deleting		
A^C	←	Deletes the preceding character in the line being typed. Repeated use deletes several characters.
Q^C	↑	Deletes the entire line being typed.
W^C	\	Deletes the preceding word in the line being typed.
P^C *and a character*	%	Deletes characters from old line up to but not including the character typed after it.
X^C *and a character*	%	Deletes characters from old line up to and including the character typed after it.
K^C		Deletes the next character in the old line; prints the character it deletes.
S^C	%	Deletes the next character in the old line.
For Copying		
D^C		Copies and prints rest of old line and ends the edit.
F^C		Copies but does not print rest of old line and ends the edit.
H^C		Copies and prints rest of old line; edit continues at end of new line.
Y^C		Copies but does not print rest of old line; edit continues with the new line acting as the old line.
R^C		Prints rest of old line plus new line up to the point where R^C was typed; edit continues at this point.
T^C		Same as R^C except that it aligns old and new lines.

EDITING CONTROL CHARACTERS (Cont)		
Control Character	**Symbol Printed**	**Action**
For Copying (Cont)		
C^C		Copies the next character in the old line.
O^C *and a character*		Copies the old line up to but not including the character typed after it, printing the characters copied.
Z^C *and a character*		Copies the old line up to and including the character typed after, printing the characters copied.
U^C		Copies characters from old line up to the next tab stop in the new line, printing the characters copied.[1]
For Inserting		
E^C	< >	Inserts text into old line; first E^C prints <, second E^C prints >.
Others		
I^C		Spaces up to the next tab stop.[1]
Line Feed		Continues line being typed.
Carriage Return		Ends the new line and the edit.
N^C	←	Backspaces in the old and new lines.
V^C *and a control character*		Indicates that the control character that follows is to be accepted as any other character instead of performing its usual editing function.

Notice that many of the control characters in the above table (such as Z^C) operate on the **old line** to form the **new line**. When these characters are being used to change program lines, the **old line** is the line being changed; the **new line** is the edited line that replaces the old line in the program.

Editing The Line Being Typed

Whenever CCS commands or SUPER FORTRAN statements are being typed at the terminal, the control characters A^C, Q^C, and W^C are available.

1 - CCS has tab stops at print position 7, and every 7th print position thereafter.

A^C deletes the preceding character in the line being typed.

W^C deletes the preceding word in the line being typed.

Q^C deletes the entire line being typed.

Both A^C and W^C may be used repeatedly to delete more than one preceding character or word in the line being typed.

Example

```
>20 IF (XGRA^C←A^C←.GT.Y)↓
  THEN W^C\GO TO 100↵
>FAST 20↵
20 IF (X.GT.Y) GO TO 100
```

Note that W^C deletes any spaces following the word deleted, but does not delete any spaces preceding the word.

A Control Y may be typed while typing a command or statement to allow the line being typed to become the old line for editing purposes. All the editing control characters summarized above may then be used.

Example 1

```
>LIFT 1:10,Y^C
Z^C LISH^CT 1:10,$-10:$↵
```

Note that CCS returns the carriage when Y^C is typed.

Lines 1 through 10 and the last 11 lines in the program are listed.

```
>
```

In this example, the Y^C returns the carriage and allows edit of the command being typed. The user types Z^CI to copy the old line up to and including the I. He types an S to replace the F in the old line; then types an H^C to copy the rest of the old line (out to the comma). He then finishes typing the command. As soon as the terminating Carriage Return is typed, the command (LIST 1:10,$-10:$) is executed.

CAUTION: If Control Y is used to edit a line that follows a longer line, Control Y will copy the excess characters from the longer line.

Example

```
>14 C: THIS IS A TEST OF CONTROL Y.↵
>15 A=SQRT(X+Y)Y^C
>Z^C+15 A=SQRT(X+ZD^C)↓
A TEST OF CONTROL Y.
```

Note that Y^C copied the excess characters in line 14.

```
= MISUSED
>Z^C+15 A=SQRT(X+Z)↵
```

Editing A Line Already Typed

Editing The Previous Line

Immediately after a line of a program has been entered from the terminal, it becomes the old line for editing purposes; all the editing control characters may then be used.

Example

```
>35 X=SQRT(A); IF (X.LT.X) GO TO 60↵
>Z^C.35 X=SQRT(A); IF (X.LT.AD^C) GO TO 60
>FAST 35↵
35 X=SQRT(A); IF (X.LT.A) GO TO 60
>
```

In this example, control characters are used to edit the previous line (line 35). After line 35 has been entered, the user types Z^C. to copy characters up to and including the first . in line 35. He then types LT .A followed by a D^C, which copies the rest of the old line and ends the edit. The new line 35 replaces the old line 35 in the program.

Example 2

```
>17 X=SQRT(A-B)↵
>18 YO^C-=SQRT(A+B)↵
>FAST 17:18↵
17 X=SQRT(A-B)
18 Y=SQRT(A+B)
>
```

The O^C- copies characters up to but not including the -. The user types the rest of the new line.

In this example, the previous line (line 17) is used as an edit image in creating line 18. Here, both the old line (line 17) and the new line (line 18) become part of the current program.

The previous line is also available as the old line when a syntax error occurs after it is entered. Thus,

```
>10 A=3-*SQRT(3.14159)↵

* MISUSED
>Z^C-10 A=3-E^C<AE^C>D^C*SQRT(3.14159)
>FAST 10↵
10 A=3-A*SQRT(3.14159)
>
```

In this example a syntax error occurs when line 10 is entered. CCS prints an error message followed by a >. To correct the error, the user types a Z^C- which copies characters from the old line (the line in which the syntax error occurred) up to and including the -. He then uses E^C to insert the variable name A by typing E^CAE^C. The first E^C prints a <; the second, a >. A D^C then copies the rest of the old line and ends

the edit. The correct line 10 is now entered into the program.

Control characters may also be used to edit the previous line during ENTER with @ prompts and during ENTER with line number prompts, whether or not a syntax error occurs.

Example 1

```
>1:10 ↵
@ACCEPT A,B ↵
@Z=SQRT(A-B) ↵
@O^C-Z=SQRT(A+B) ↵          O^C- copies the previous
@DISPLAY Z ↵                line up to but not in-
@D^C                        cluding the -.
>FAST ↵
1 ACCEPT A,B
2 Z=SQRT(A-B)
3 Z=SQRT(A+B)
4 DISPLAY Z
>
```

Notice that a D^C typed immediately after an @ or line number prompt terminates the ENTER command, but a D^C typed anywhere else copies the rest of the previous line and ends the new line.

Example 2

```
>10(2)20 ↵
10 ACCEPT B,C,D ↵
12 A=B+*D ↵

* MISUSED
12 Z^C+A=B+E^C<CE^C>D^C*D     Z^C+ copies char-
14 DISPLAY A ↵                acters up to and
16 D^C                        including the +,
>FAST ↵                       E^CCE^C inserts a C,
10 ACCEPT B,C,D               D^C copies rest of
12 A=B+C*D                    old line (but D^C
14 DISPLAY A                  in line 16 termi-
>                             nates command.)
```

The EDIT And MODIFY Commands

The EDIT and MODIFY commands may be used to edit any line or group of lines in CCS. All the editing control characters may be used during these commands.

The EDIT command has the form

```
>EDIT line list ↵
```

where the line list consists of one or more line or range addresses, separated by commas.

If only one line is addressed in the line list, the command causes the line addressed to be printed on the terminal; the line is then available as the old line for editing purposes.

Example

```
>EDIT 2 ↵
2 S=A-B/2                  CCS prints the old line 2
                           and the number 2 on the
2 S=(A+B+C)/2. ↵           next line. The user types
>                          the text of the new line 2.
```

If more than one line is addressed in the line list, CCS prints the lines addressed one at a time. After printing a line, it waits for the user to edit it and then goes on to the next line. When the last line has been edited, control is returned to the user. The process can be terminated early by typing D^C at the beginning of a line.

Example

```
>LIST ↵
   1          ACCEPT A,B
   2          S=(A+B+C)/2.
   3          AREA=SQRT(S*(S+A)*S*
                        (S+B)*S*(S-C))
   4          WRITE (1,200) A,B,C, AREA
   5    200   FORMAT(4F10.2)
   6          STOP
   7          STOP
>EDIT 1,3:5,$ ↵            This command allows
                           edit of lines 1, 3 through
                           5, and the last line.
1 ACCEPT A,B
1 H^CACCEPT A,B,C ↵        H^C copies old line, user
                           types ,C ↵.
3 AREA=SQRT(S*(S+A)*S*(S+B)*S*(S-C))
3 AREA=SQRT(S*(S-A)*(S-B)*P^C(%%%%D^C(S-C))
                           User types new line up
                           to the * after (S-B),
                           then a P^C( to delete
                           characters up to but not
                           including the (. D^C cop-
                           ies rest of old line.
4 WRITE (1,200) A,B,C,AREA
4 F^C                      This line is OK; F^C cop-
                           ies it without printing
                           it.
5 200 FORMAT (4F10.2)
5 Z^C1200 FORMAT(4F16.8) ↵
                           User changes the format
                           specification.
7 STOP
7 END ↵                    The last line is edited.
```

```
>LIST ↵
   1            ACCEPT A,B,C
   2            S=(A+B+C)/2.
   3            AREA=SQRT(S*(S-A)*(S-B)
                                  *(S-C))
   4            WRITE (1,200) A,B,C,AREA
   5   200      FORMAT(4F16.8)
   6            STOP
   7            END
>
```

The MODIFY command has the form

>MODIFY line list ↵

It is the same as the EDIT command except that the lines addressed in the line list are not printed on the terminal.

Editing Input Data

Data typed into a running program may be edited with the following editing control characters:

A^C deletes the preceding character unless that character is one of the terminators used to separate data items (comma, space, Line Feed, Carriage Return).

W^C deletes the current data item. It has no effect on the characters A^C does not delete.

Q^C restarts the entire input statement from the beginning.

Example 1

```
>10(10) ↵
10 DIMENSION A(5) ↵
20 ACCEPT A ↵
30 DISPLAY A ↵
40 END ↵
50 D^C
>RUN ↵
2354A^C←,34.6A^C←78,23.4,56.7W^C\46.8,10.1 ↵
 235      34.78      23.4      46.8      10.1

(@40 )>
```

Example 2

```
>FAST ↵
10 DIMENSION A(5)
20 ACCEPT A
30 DISPLAY A
40 END
>RUN ↵
11.17,33.9,46.1,39Q^C↑
11.7,85,33.9,46.1,39 ↵
 11.7      85.    33.9      46.1      39.

(@40 )>
```

RENUMBERING A PROGRAM

The user may change the line numbers of some or all of the lines in his program by using the RENUMBER command. This command may be used in any of the following forms, where l_1, l_2, l_3, and l_4 are all line numbers:

>RENUMBER ↵

>RENUMBER l_1:l_2 ↵

>RENUMBER l_1:l_2 AS l_3:l_4 ↵

>RENUMBER l_1:l_2 AS l_3 (increment) l_4 ↵

>RENUMBER l_1:l_2 AS l_3 (increment) ↵

The form

>RENUMBER ↵

reassigns line numbers to all lines in the program beginning with 1 in increments of 1.

Example

```
>LIST ↵
   .1           C: THIS PROGRAM COM-
                PUTES THE AREA OF A
                TRIANGLE
   .2           C: A,B,C REPRESENT SIDES
                OF THE TRIANGLE
   3            ACCEPT "ENTER VALUES OF
                A,B,C ",A,B,C
   3.5          S=(A+B+C)/2.
   4            AREA=SQRT(S*(S-A)*(S-B)*
                (S-C))
   5            WRITE (1,200) A,B,C,AREA
   6      200   FORMAT (4F16.8)
   7            STOP
 298            END
>RENUMBER ↵

>LIST ↵
   1            C: THIS PROGRAM COM-
                PUTES THE AREA OF A
                TRIANGLE
   2            C: A,B,C REPRESENT SIDES
                OF THE TRIANGLE
   3            ACCEPT "ENTER VALUES OF
                A,B,C ",A,B,C
```

```
   4          S=(A+B+C)/2.
   5          AREA=SQRT(S*(S-A)*(S-B)*
              (S-C))
   6          WRITE (1,200) A,B,C,AREA
   7     200  FORMAT (4F16.8)
   8          STOP
   9          END
>
```

The form

> RENUMBER $/_1$:$/_2$ ↵

reassigns line numbers to the lines in the range specified beginning with the first line number in the range ($/_1$) in increments of the first of 1, .1, .01, and .001 which fits the range specified. For example, if the above program had line numbers as in the following,

```
   1          C: THIS PROGRAM COM-
              PUTES THE AREA OF A
              TRIANGLE
   2          C: A,B,C REPRESENT SIDES
              OF THE TRIANGLE
   3          ACCEPT "ENTER VALUES OF
              A,B,C ",A,B,C
   3.1        S=(A+B+C)/2.
   4          AREA=SQRT(S*(S-A)*(S-B)*
              (S-C))
   4.6        WRITE (1,200) A,B,C,AREA
   5     200  FORMAT (4F16.8)
   8          STOP
   9          END
```

the command

> RENUMBER 3:5 ↵

reassigns line numbers as follows:

```
   1          C: THIS PROGRAM COM-
              PUTES THE AREA OF A
              TRIANGLE
   2          C: A,B,C REPRESENT SIDES
              OF THE TRIANGLE
   3          ACCEPT "ENTER VALUES OF
              A,B,C ",A,B,C
   3.1        S=(A+B+C)/2.
   3.2        AREA=SQRT(S*(S-A)*(S-B)*
              (S-C))
   3.3        WRITE (1,200) A,B,C,AREA
   3.4   200  FORMAT (4F16.8)
   8          STOP
   9          END
```

The form

> RENUMBER $/_1$:$/_2$ AS $/_3$:$/_4$ ↵

reassigns line numbers to the lines in the range $/_1$:$/_2$, beginning with line number $/_3$ and choosing as increment the first of 1, .1, .01, and .001 that will allow the lines to fit the range $/_3$:$/_4$. Any lines in the range $/_3$:$/_4$ before the command is given are deleted when the command is given. Thus, the user should be sure that he does not accidentally delete program lines that he wishes to keep when using this form of RENUMBER. For example, if the command

> RENUMBER 3:4 AS 8:13 ↵

is applied to the revised version of the preceding program, the last two lines in the program are lost, producing the program

```
   1          C: THIS PROGRAM COM-
              PUTES THE AREA OF A
              TRIANGLE
   2          C: A,B,C REPRESENT SIDES
              OF THE TRIANGLE
   8          ACCEPT "ENTER VALUES OF
              A,B,C ",A,B,C
   9          S=(A+B+C)/2.
  10          AREA=SQRT(S*(S-A)*(S-B)*
              (S-C))
  11          WRITE (1,200) A,B,C,AREA
  12     200  FORMAT (4F16.8)
```

The form

RENUMBER $/_1$:$/_2$ AS $/_3$(increment)$/_4$ ↵

reassigns line numbers to the lines in the range $/_1$:$/_2$, beginning with line number $/_3$ in the increment specified in the command. Any lines in the range $/_3$ through $/_4$ before the command is given are deleted when the command is given.

Example

If the following program is in CCS:

```
   1          ACCEPT "ENTER VALUES
              OF A,B,C ",A,B,C
   2          S=(A+B+C)/2.
   2.1        AREA=SQRT(S*(S-A)*(S-B)*
              (S-C))
   2.25       WRITE (1,200) A,B,C,AREA
   2.3   200  FORMAT (4F16.8)
   9          STOP
  10          END
```

the command

> RENUMBER 2.1:2.3 AS 4(2)8 ↵

produces the program

```
1          ACCEPT "ENTER VALUES OF
           A,B,C ",A,B,C
2          S=(A+B+C)/2.
4          AREA=SQRT(S*(S-A)*(S-B)*
           (S-C))
6          WRITE (1,200) A,B,C,AREA
8     200  FORMAT (4F16.8)
9          STOP
10         END
```

The last form of RENUMBER,

> **RENUMBER $/_1$:$/_2$ AS $/_3$(increment)** ↵

reassigns line numbers to the lines in the range $/_1$:$/_2$, beginning with line number $/_3$ in the increment specified. In this form of the command, the user is protected against accidentally deleting or interleaving program lines. No lines, other than $/_3$ itself, are deleted when the command is executed; if any lines in $/_1$:$/_2$ would cause deletion or interleaving when renumbered, the lines are not renumbered.

Example

If this program is in CCS:

```
1          C: THIS PROGRAM COM-
           PUTES THE AREA OF A
           TRIANGLE
2          ACCEPT "ENTER VALUES OF
           A,B,C ",A,B,C
3          S=(A+B+C)/2.
3.1        AREA=SQRT(S*(S-A)*(S-B)*
           (S-C))
3.2        WRITE (1,200) A,B,C,AREA
3.9   200  FORMAT (4F16.8)
10         STOP
13         END
```

the command

> **RENUMBER 3.1:3.9 AS 4(2)** ↵

produces the following program:

```
1          C: THIS PROGRAM COM-
           PUTES THE AREA OF A
           TRIANGLE
2          ACCEPT "ENTER VALUES OF
           A,B,C ",A,B,C
3          S=(A+B+C)/2.
4          AREA=SQRT(S*(S-A)*(S-B)*
           (S-C))
6          WRITE (1,200) A,B,C,AREA
8     200  FORMAT (4F16.8)
10         STOP
13         END
```

SECTION 11

SAMPLE PROGRAMS

This section contains programs written in SUPER FORTRAN and executed on the Tymshare system. The problems are presented in an increasing degree of complexity.

MONTHLY PAYMENT PROGRAM

DEFINE THE PROBLEM

Compute the monthly interest and payment on a debt, and then print the amount of the payment.

Input

1. Original debt (P)
2. Annual interest (I)
3. Number of monthly payments to be made (N)

Compute

Monthly interest (I=I/12)

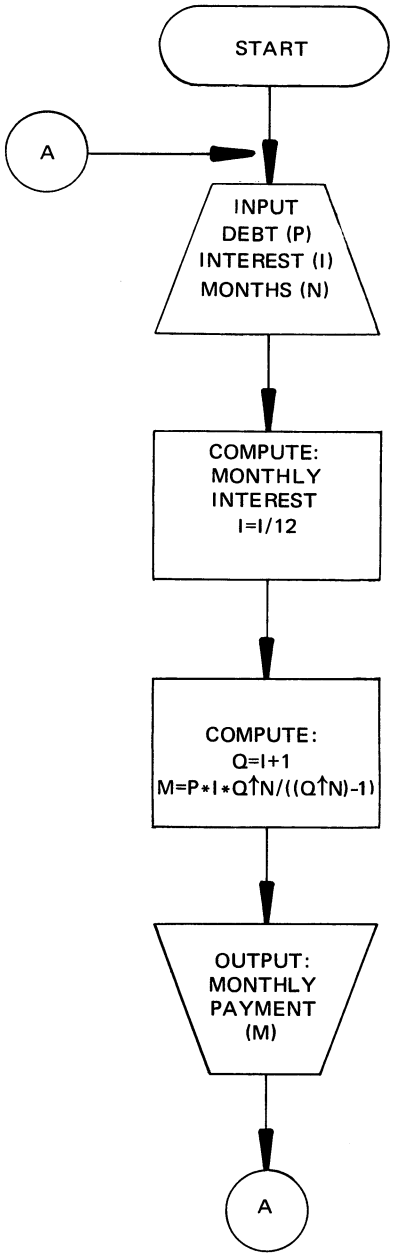
Monthly payment (M)

$$M = \frac{P \cdot I(I+1)^N}{(I+1)^N - 1}$$

Output

Monthly payment, M.

FLOWCHART


START
A
INPUT
DEBT (P)
INTEREST (I)
MONTHS (N)
COMPUTE:
MONTHLY
INTEREST
I=I/12
COMPUTE:
Q=I+1
M=P*I*Q↑N/((Q↑N)-1)
OUTPUT:
MONTHLY
PAYMENT
(M)
A

SUPER FORTRAN CODE AND SAMPLE EXECUTION

```
>5(5)
5 REAL I,N,M
10 7 DISPLAY "***"
15 ACCEPT "PRINCIPAL= $",P,"INTEREST= ",I,"NO.MONTHS= ",N
20 I=I/12
25 Q=I+1
30 M=P*I*Q↑N/((Q↑N)-1)
35 DISPLAY "MONTHLY PAYMENT = $",M
40 GO TO 7
45 END
50
>RUN
***
PRINCIPAL= $1200
INTEREST= .06
NO.MONTHS= 12
MONTHLY PAYMENT = $    103.27972
***
PRINCIPAL= $1200
INTEREST= .06
NO.MONTHS= 6
MONTHLY PAYMENT = $    203.51455
***
PRINCIPAL= $1200
INTEREST= .06
NO.MONTHS= 18
MONTHLY PAYMENT = $    69.878077
***
PRINCIPAL= $
INTERRUPT
15 >
```

DOUBLE DECLINING BALANCE DEPRECIATION

DEFINE THE PROBLEM

Compute the double declining balance depreciation on any given asset over any specified number of years using formatted I/O.

Input

1. Cost of the asset (C).
2. Estimated useful lifetime (U).

Compute

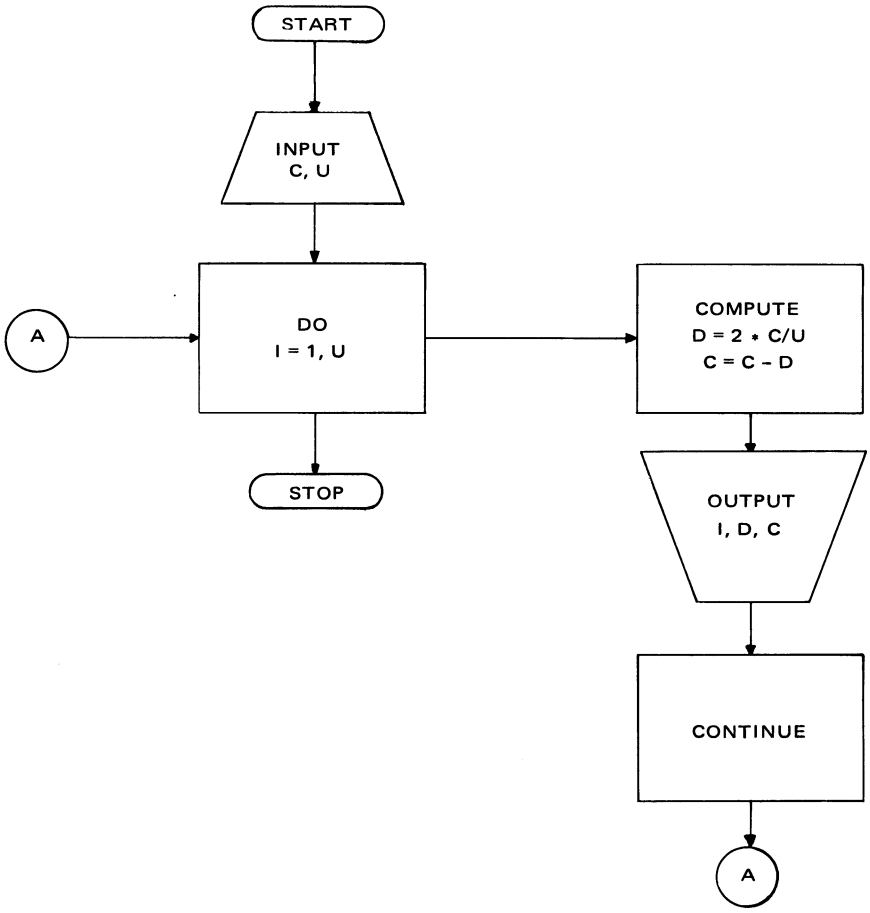
1. Depreciation $D = 2\frac{C}{U}$
2. Book values $C = C - D$

Output

For the entire range of years:

1. Year (I)
2. Amount of depreciation (D)
3. Book value (C)

FLOWCHART



SUPER FORTRAN CODE AND SAMPLE EXECUTION

```
>LIST
 10            C: DOUBLE DECLINING BALANCE DEPRECIATION PROGRAM
 20            WRITE(1,2) "COST OF ASSET= $"
 30      2     FORMAT(//S,&)
 40            READ(0,3) C
 50      3     FORMAT(F12.2)
 60            WRITE(1,4) 'ESTIMATED USEFUL LIFETIME= '
 70      4     FORMAT(S,&)
 80            READ(0,3) U
 90            WRITE(1,5) 'YEAR','DEPRECIATION','BOOK VALUE'
100      5     FORMAT(/S,8X,S,8X,S)
110            DO 100 I=1,U
120            D=2*C/U
130            C=C-D
140            WRITE(1,7) I,D,C
150      7     FORMAT(I4,8X,'$',F10.2,8X,'$',F8.2)
160      100   CONTINUE
170            END
>RUN


COST OF ASSET= $3500.00
ESTIMATED USEFUL LIFETIME= 7.

YEAR         DEPRECIATION         BOOK VALUE
   1         $    1000.00        $ 2500.00
   2         $     714.29        $ 1785.71
   3         $     510.20        $ 1275.51
   4         $     364.43        $  911.08
   5         $     260.31        $  650.77
   6         $     185.93        $  464.84
   7         $     132.81        $  332.03

(@170 )>
```

LEAST SQUARE LINE

DEFINE THE PROBLEM

Fit a least square line of the form Y=A+BX to the following set of data, where X is the independent and Y the dependent variable.

X	1	3	4	6	8	9	11	14
Y	1	2	4	4	5	7	8	9

Solution

Compute first the sums shown below.

X	Y	X^2	XY
1	1	1	1
3	2	9	6
4	4	16	16
6	4	36	24
8	5	64	40
9	7	81	63
11	8	121	88
14	9	196	126
$\Sigma X=56$	$\Sigma Y=40$	$\Sigma X^2=524$	$\Sigma XY=364$

Then compute the regression coefficients A and B.

$$A = \frac{(\Sigma Y)\ (\Sigma X^2)\ -\ (\Sigma X)\ (\Sigma XY)}{N\ \Sigma X^2\ -\ (\Sigma X)^2}$$

$$B = \frac{N\ \Sigma XY\ -\ (\Sigma X)\ (\Sigma Y)}{N\ \Sigma X^2\ -\ (\Sigma X)^2}$$

Then

Y=A+BX

Input

1. Number of data points (N)
2. The X values (X(1) - X(8))
3. The Y values (Y(1) - Y(8))

Compute

Equations given above:

1. Sums needed
2. Coefficients A and B

Output

The equation of the least square line

SUPER FORTRAN CODE AND SAMPLE EXECUTION

```
>FAST
10 *  THIS PROGRAM FITS A LEAST SQUARE LINE OF FORM: Y=A+BX TO
A SET OF DATA (X,Y) WHERE X IS THE INDEPENDENT VARIABLE.
15
20 STRING FMT(90)
25 DIMENSION X(8),Y(8),XX(8),XY(8)
30 ACCEPT "NUMBER OF DATA POINTS= ",N
35 ACCEPT "THE X VALUES ARE ",X
40 ACCEPT "THE Y VALUES ARE ",Y
45
50 *  LOOP TO CALCULATE XX AND XY
55
60 DO 100 I=1,N
65 XX(I)=X(I)↑2
70 XY(I)=X(I)*Y(I)
75 100 CONTINUE
80
85 *  LOOP TO CALCULATE TOTALS
90
95 DO 200 I=1,N
100 TX=TX+X(I)
105 TY=TY+Y(I)
110 TXX=TXX+XX(I)
115 TXY=TXY+XY(I)
120 200 CONTINUE
125
130 *  TO CALCULATE COEFFICIENTS A,B
135
140 A=(TY*TXX-TX*TXY)/(N*TXX-TX*TX)
145 B=(N*TXY-TX*TY)/(N*TXX-TX*TX)
150
155 *  TO COMPUTE OUTPUT FORMAT
160
165 I=TRUNC(LOG10(A)+.5)+4
170 J=TRUNC(LOG10(B)+.5)+4
175 FMT="('THE LEAST SQUARE LINE IS: Y=',F"+STR(I)+".3,'+',F"
+STR(J)+".3,'X')"
180 WRITE(1,FMT) A,B
185 END
>RUN
NUMBER OF DATA POINTS= 8
THE X VALUES ARE 1,3,4,6,8,9,11,14
THE Y VALUES ARE 1,2,4,4,5,7,8,9
THE LEAST SQUARE LINE IS: Y=.545+.636X

(@185 )>
```

COST OF PAINTING A BOX

DEFINE THE PROBLEM

Input

Enter from the terminal the dimensions of the box, cost of paint, and area that a gallon of paint will cover.

Compute

The cost and amount of paint needed to paint the box.

Output

The surface area of the box, gallons of paint needed, and cost of the paint needed.

NOTE: This program demonstrates the use of labelled COMMON.

SUPER FORTRAN CODE AND SAMPLE EXECUTION

```
>LIST
 10            COMMON /SUB1/ D1,D2,D3//A/SUB2/S,C1
 20            DISPLAY 'ENTER THE DIMENSIONS OF THE BOX'
 30            ACCEPT D1,D2,D3
 40            WRITE(1,10) 'COST OF PAINT = $'
 50            ACCEPT C1
 60            WRITE(1,10) 'SQ FEET PER GALLON ='
 70            ACCEPT S
 80            CALL SA
 90            CALL CA
100      10    FORMAT(S,&)
110            END
120            SUBROUTINE SA
130            COMMON /SUB1/DIM1,DIM2,DIM3//A
140            A=2*(DIM1*DIM2+DIM1*DIM3+DIM2*DIM3)
150            WRITE(1,20) 'SURFACE AREA =',A
160      20    FORMAT(S,F8.4)
170            RETURN
180            END
190            SUBROUTINE CA
200            COMMON AREA /SUB2/ SQFT,COST
210            WRITE(1,30) 'GALLONS OF PAINT NEEDED =',AREA/SQFT,
               'COST TO PAINT BOX = $', AREA*COST/SQFT
220      30    FORMAT(S,F6.2)
230            RETURN
240            END
>RUN
ENTER THE DIMENSIONS OF THE BOX
5.49,7.67,4.98
COST OF PAINT = $4.89
SQ FEET PER GALLON =312.
SURFACE AREA =215.2902
GALLONS OF PAINT NEEDED =    .69
COST TO PAINT BOX = $  3.37

(@110 )>
```

PAYROLL CHECKS

(FILE I/O)

DEFINE THE PROBLEM

Input

1. Read from file PRF each employee's number (EMP(I)), pay rate (RATE(I)), and hours worked (HRS(I)).
2. Enter from the terminal any changes to an employee's pay rate and/or hours worked.

Compute

Gross pay for each employee.

Output

1. Updated file of pay rates and hours. The file PRF is in the form:

   employee number$_1$ pay rate$_1$ hours$_1$

   employee number$_2$ pay rate$_2$ hours$_2$

   . . .
   . . .
   . . .

   Before execution it appears as:

```
>COPY PRF TO TEL
 99   2.00 48.00
 77   2.50 40.00
 88   3.75 40.00
 55   5.50 40.00
 66   3.35 40.00
 44   2.10 48.00
```

2. Gross pay file.

   The file GPF has the form:

   employee number$_1$ pay$_1$

   employee number$_2$ pay$_2$

   . .
   . .
   . .

NOTE: This program demonstrates random file input/output.

SUPER FORTRAN CODE AND SAMPLE EXECUTION

```
>LIST
 10            DIMENSION EMP(100),RATE(100),HRS(100),PAY(100)
 15            OPEN(3,'PRF',RANDIO,SYMBOLIC)
 25      7     FORMAT(I2)
 30            READ(3,8,END=33)(EMP(I),RATE(I),HRS(I),I=1,20)
 33      33    N=I-1
 35      35    DISPLAY 'ENTER ANY UPDATES'
 40      15    ACCEPT EMPNO,RT,HR
 42            IF (EMPNO .EQ. 0) GO TO 25
 45            DO 20 I=1,100
 50            IREC=I
 55            IF (EMPNO .EQ. EMP(I)) GO TO 30
 60      20    CONTINUE
 65      25    CLOSE(3)
 70            OPEN(4,"GPF",OUTPUT,SYMBOLIC)
 75            DO 200 I=1,N
 80            J=EMP(I)
 85            PAY(J)=HRS(I)*RATE(I)
 90            WRITE(4,5,ERR=210) EMP(I),PAY(J)
 95      200   CONTINUE
100      5     FORMAT(I3,2X,F7.2)
105      210   CLOSE(4)
110            STOP
115      30    POSITION(3,1+(IREC-1)*16)
120            RATE(IREC)=RT; HRS(IREC)=HR
125            WRITE(3,9) EMPNO,RT,HR
130            GO TO 35
135      8     FORMAT(I3,F6.2,F6.2)
140      9     FORMAT(I3,F6.2,F6.2,&)
145            END
>RUN
ENTER ANY UPDATES
77,2.50,48.
ENTER ANY UPDATES
55,6.00,40.
ENTER ANY UPDATES
0,0,0

STOP
(@110 )>COPY PRF TO TEL
 99   2.00 48.00
 77   2.50 48.00
 88   3.75 40.00
 55   6.00 40.00
 66   3.35 40.00
 44   2.10 48.00
```

```
(@110 )>COPY GPF TO TEL
 99     96.00
 77    120.00
 88    150.00
 55    240.00
 66    134.00
 44    100.80

(@110 )>QUIT

-
```

The file PRF now contains the updated values, and the gross pay has been computed and stored in the file GPF.

CHECKING ACCOUNT SERVICE CHARGES

In this problem, we wish to compute the monthly service charge for a regular checking account. The amount of the service charge is based on the average monthly balance and the number of checks written. The charge may be computed from the following table.

AVERAGE MONTHLY BALANCE

NUMBER OF CHECKS	UNDER $200	$200 to $299	$300 to $399	$400 to $499	$500 to $599	$600 to $699	$700 to $799	$800 to $899	$900 to $999	$1000 to $1099	$1100 to $1199	$1200 to $1299	$1300 to $1399	$1400 to $1499	$1500 to $1599
0	$ .75	$ .47	$ .33	$	$	$	$	$	$	$	$	$	$	$	$
1	.82	.54	.40												
2	.89	.61	.47	.33											
3	.96	.68	.54	.40											
4	1.03	.75	.61	.47											
5	1.10	.82	.68	.54											
6	1.17	.89	.75	.61											
7	1.24	.96	.82	.68	.54										
8	1.31	1.03	.89	.75	.61										
9	1.38	1.10	.96	.82	.68	.54									
10	1.45	1.17	1.03	.89	.75	.61									
11	1.52	1.24	1.10	.96	.82	.68	.54								
12	1.59	1.31	1.17	1.03	.89	.75	.61								
13	1.66	1.38	1.24	1.10	.96	.82	.68	.54							
14	1.73	1.45	1.31	1.17	1.03	.89	.75	.61							
15	1.80	1.52	1.38	1.24	1.10	.96	.82	.68	.54						
16	1.87	1.59	1.45	1.31	1.17	1.03	.89	.75	.61						
17	1.94	1.66	1.52	1.38	1.24	1.10	.96	.82	.68	.54					
18	2.01	1.73	1.59	1.45	1.31	1.17	1.03	.89	.75	.61					
19	2.08	1.80	1.66	1.52	1.38	1.24	1.10	.96	.82	.68	.54				
20	2.15	1.87	1.73	1.59	1.45	1.31	1.17	1.03	.89	.75	.61				
21	2.22	1.94	1.80	1.66	1.52	1.38	1.24	1.10	.96	.82	.68	.54			
22	2.29	2.01	1.87	1.73	1.59	1.45	1.31	1.17	1.03	.89	.75	.61			
23	2.36	2.08	1.94	1.80	1.66	1.52	1.38	1.24	1.10	.96	.82	.68	.54		
24	2.43	2.15	2.01	1.87	1.73	1.59	1.45	1.31	1.17	1.03	.89	.75	.61		
25	2.50	2.22	2.08	1.94	1.80	1.66	1.52	1.38	1.24	1.10	.96	.82	.68	.54	.00

Input

1. Number of checks written (NUMCHKS)
2. Average monthly balance (AVGBAL)
3. Current monthly balance (CURBAL)

Compute

1. This month's service charge (MATRIX (I,J))
2. New balance (CURBAL)

Output

1. This month's service charge
2. New balance

SUPER FORTRAN CODE AND SAMPLE EXECUTION

```
>LIST
 10            C:       CHECKING ACCOUNT
 20            C:
 30            WRITE(1,10) 'THIS IS A PROGRAM TO COMPUTE THE MO
               NTHLY SERVICE CHARGE FOR A REGULAR','CHECKING ACCOUN
               T AT A COMMERCIAL BANK.'
 40      10    FORMAT(S)
 50            C:
 60            REAL MATRIX(15,26),NUMCHKS
 70            WRITE(1,50) '     CURRENT BALANCE = '
 80      50    FORMAT(//S,&)
 90      75    FORMAT(/S,&)
100            ACCEPT CURBAL
110            DO 100 I=1,15
120            IF(I.LT.15) MATRIX(I,26)=2.50-(I*.14)
130            IF(I.EQ.15) MATRIX(I,26)=0.00
140            IF(I.EQ.1) MATRIX(I,26)=2.50
150     100    CONTINUE
160            DO 200 I=1,15
170            DO 200 J=1,25
180            MATRIX(I,J) = MATRIX(I,26)-((26-J)*.07)
190     200    CONTINUE
200            WRITE(1,75) '     AVERAGE BALANCE FOR THIS MONTH = '
210            ACCEPT AVGBAL
220            WRITE(1,75) '     NUMBER OF CHECKS THIS MONTH = '
230            ACCEPT NUMCHKS
240            I=AVGBAL/100
250            J=NUMCHKS + 1
260            IF (MATRIX(I,J).GE.0.54) GO TO 400
270            MATRIX(I,J) = 0.00
280            IF ((I.EQ.2.0.AND.J.EQ.1.0).OR.(I.EQ.3.0.AND.J.EQ.3.0)
               .OR.(I.EQ.4.0.AND.J.EQ.5.0)) MATRIX(I,J) = 0.47
290            IF ((I.EQ.3.0.AND.J.EQ.1.0).OR.(I.EQ.4.0.AND.J.EQ.4.0))
               MATRIX(I,J) = 0.40
300            IF ((I.EQ.3.0.AND.J.EQ.1.0).OR.(I.EQ.4.0.AND.J.EQ.3.0))
               MATRIX(I,J) = 0.33
310     400    DISPLAY 'THIS MONTHS SERVICE CHARGE =',MATRIX(I,J)
320            CURBAL = CURBAL - MATRIX(I,J)
330            DISPLAY 'THE NEW CURRENT BALANCE =',CURBAL
340            END
>RUN
THIS IS A PROGRAM TO COMPUTE THE MONTHLY SERVICE CHARGE FOR A REGULAR
CHECKING ACCOUNT AT A COMMERCIAL BANK.


     CURRENT BALANCE = 513.67

     AVERAGE BALANCE FOR THIS MONTH = 336.71

     NUMBER OF CHECKS THIS MONTH = 13
THIS MONTHS SERVICE CHARGE =   1.24
THE NEW CURRENT BALANCE =   512.43

(@340 )>
```

UPDATING FILES

DEFINE THE PROBLEM

There are three files that must be kept current: the EMPLOYEE file, DISTRICT file, and JOB file. The program allows additions and corrections to the JOB and DISTRICT files. It allows additions, corrections, and deletions to the EMPLOYEE file and alphabetizes by employee name.

The files are in the following forms:

@JOB file

A_1, B_1 where A represents the job code, and B represents the job name.
A_2, B_2
.
.
.
A_n, B_n

@DISTRICT file

A_1, B_1 where A represents the district code, and B the district name.
A_2, B_2
.
.
.
A_n, B_n

@EMPLOYEE file

$A_1, B_1, C_1, D_1, E_1, F_1, G_1, H_1$ where A through H represent last name, first name,
$A_2, B_2, C_2, D_2, E_2, F_2, G_2, H_2$ district code, employee number, job code, home address, city and state, and telephone number.
.
.
.
$A_n, B_n, C_n, D_n, E_n, F_n, G_n, H_n$

Input

New data for the file

Output

Updated file

Execution

Initially the @DISTRICT file contains

```
>COPY @DISTRICT TO TEL
DC,WASHINGTON DC
PA,PALO ALTO
BO,BOSTON
NY,NEW YORK
ME,METROPOLITAN
DA,DALLAS
LA,LOS ANGELES
SF,SAN FRANCISCO
SD,SAN DIEGO

>
```

Initially the @JOB file contains

```
>COPY @JOB TO TEL
SEC,SECRETARY
VP,VICE PRESIDENT
PW,PROGRAMMER WRITER
PR,PROGRAMMER
SR,SALES REPRESENTATIVE
DM,DISTRICT MANAGER
CO,COMPUTER OPERATOR
ME,MAINTENANCE ENGINEER

>
```

Initially the @EMPLOYEE file contains

```
>COPY @EMPLOYEE TO TEL
ADAMS,ANSEL,DC,111,VP,211 HAVEN AVE,WASHINGTON DC,399-8474
EVERS,EVERETT,PA,222,PW,890 RAND ST,PALO ALTO CA,983-3333
INGLES,INGRID,BO,333,SEC,38 FREEDOM WAY,BOSTON MASS,233-3355
OLSON,OLLIE,NY,444,PR,1010 EIGHTH AVE,NEW YORK NY,626-3554
UPDIKE,URSULA,ME,555,SR,881 BROADWAY,NEW YORK NY,211-5467

>
```

SUPER FORTRAN CODE

UPDATE PROGRAM

```
>FAST
1 STRING FNAME(9)
2 STRING DJ(2,50)(40)
3 STRING FI(2,50)(40)
4 DISPLAY "WHAT FILE DO YOU WISH TO UPDATE?"
5 ACCEPT FNAME
6 IF (FNAME.NE."@EMPLOYEE") GO TO 3
7
8 *EMPLOYEE UPDATE
9
10 LINK "@BALPH"                                  ┐ Note that @BALPH is called as
11 3 OPEN(4,FNAME,INPUT,SYMBOLIC)                 ┘ a LINK file.
12
13 *JOB OR DISTRICT UPDATE
14
15 READ(4,END=18)((FI(I,J),I=1,2),J=1,50)
16 18 NFI=J-1
17 CLOSE(4)
18 OPEN(4,FNAME,OUTPUT,SYMBOLIC)
19 IF (FNAME.EQ.'@JOB') GO TO 9
20 DISPLAY "ENTER UPDATES TO THE @DISTRICT FILE IN THE FOLLOW
ING FORM:"
21 DISPLAY "DISTRICT CODE, DISTRICT NAME"
22 GO TO 50
23 9 DISPLAY "ENTER UPDATES TO THE @JOB FILE IN THE FOLLOWING
 FORM:"
24 DISPLAY "JOB CODE, JOB NAME"
25 50 DISPLAY "TO TERMINATE INPUT, TYPE AN *"
26 ICNT=0
27
28 *ACCEPT DATA FROM THE TERMINAL
29
30 89 DO 55 J=1,100
31 DO 660 I=1,2
32 ACCEPT DJ(I,J)
33 IF (DJ(I,J).EQ."*") GO TO 30
34 660 CONTINUE
35 DO 666 L=1,NFI
36 IF (DJ(1,J).EQ.FI(1,L)) GO TO 56
37 666 CONTINUE
38 ICNT=ICNT+1
39 (FI(I,ICNT+NFI)=DJ(I,J)),I=1,2
40 55 CONTINUE
41 30 CONTINUE
42 *
43 WRITE(4,27)(((FI(I,L)),I=1,2),L=1,NFI+ICNT)
44 29 CLOSE(4)
45 27 FORMAT(S,',',S)
46 STOP
47 56 (FI(K,L)=DJ(K,J)),K=1,2
48 GO TO 89
49 END
>
```

BINARY PROGRAM ON FILE @BALPH

```
>FAST
1 STRING EDAT(8,50)(30),DUMMY(8)(30)
3 OPEN(4,"@EMPLOYEE",INPUT,SYMBOLIC)
4 OPEN(5,"ALPHEM",OUTPUT,SYMBOLIC)
5 DISPLAY "ENTER UPDATES TO THE @EMPLOYEE FILE IN THE FOLLOWING
 FORM:"
6 DISPLAY "LAST NAME, FIRST NAME, DISTRICT CODE, EMPLOYEE
NUMBER, "
7 DISPLAY "JOB CODE, HOME ADDRESS, CITY AND STATE, PHONE NUM
BER"
8 DISPLAY ""
9 DISPLAY "TO DELETE TYPE: LAST NAME, FIRST NAME,"
10 DISPLAY " EMPLOYEE NUMBER, DELETE"
11 DISPLAY ''
11.5 DISPLAY "TO RETYPE LINE, TYPE AN ALTMODE"
12 DISPLAY "TO TERMINATE INPUT, TYPE AN *"
13
14 *ACCEPT DATA FROM TERMINAL
15
16 16 DO 134 K=1,50
16.5 17 DISPLAY "ENTRY: "
17   DO 133 I=1,8
18 ACCEPT EDAT(I,K)
19.5 ON INTERRUPT GO TO 222
20 IF (EDAT(1,K).EQ."*") GO TO 30
20.5 IF (EDAT(I,K).EQ.'*') (DISPLAY 'ERROR IN INPUT--START AGAIN';
GO TO 16)
20.9 IF (EDAT(I,K).EQ."DELETE")(IF(I.NE.4)(DISPLAY"ERROR: TO
DELETE TYPE LAST NAME, FIRST NAME, EMP.NO., DELETE"; GO TO 17)
.ELSE.GO TO 134)
21 133 CONTINUE
22 134 CONTINUE
23 DISPLAY "MORE THAN 50 UPDATES. ONLY 50 ACCEPTED"
24 30 NUP=K-1
25
26 *ALPHABETIZE
27
28 IF (NUP.LT.2) GO TO 40
29 DO 15 I=1,NUP
30 DO 15 M=NUP,2,-1
31 IF ( (EDAT(1,M)+EDAT(2,M)).GE.(EDAT(1,M-1)+EDAT(2,M-1)))GO TO 15
32 (DUMMY(J)=EDAT(J,M)),J=1,8
33 (EDAT(J,M)=EDAT(J,M-1)),J=1,8
34 (EDAT(J,M-1)=DUMMY(J)),J=1,8
35 15 CONTINUE
36
37 *COMPARE TO @EMPLOYEE FILE
38
39 IOUT=0
40 40 READ(4,END=102)(DUMMY(I),I=1,8)
41 IF (IOUT.GE.NUP) GO TO 20
```

```
42 DO 202 I=IOUT+1,NUP
43 IF (PACK(DUMMY(1)+DUMMY(2)).EQ.PACK(EDAT(1,I)+EDAT(2,I)))
GO TO 10
44 IF ((DUMMY(1)+DUMMY(2)).LT.(EDAT(1,I)+EDAT(2,I))) GO TO 20
45 *EDAT < DUMMY
46 IF (EDAT(4,I).NE."DELETE") WRITE(5,70)(EDAT(K,I)
,K=1,8)
47 IOUT=I
48 202 CONTINUE
49 20 WRITE(5,70)(DUMMY(K),K=1,8)
50 GO TO 40
50.5 222 OFF INTERRUPT; GO TO 17
51
52 *IDENTICAL NAMES
53
54 10 IF (PACK(DUMMY(4)).EQ.PACK(EDAT(4,I))) GO TO 103
55 IF (PACK(DUMMY(4)).EQ.PACK(EDAT(3,I))) GO TO 40
56 WRITE(5,70)(DUMMY(K),K=1,8)
57 103 IF (EDAT(4,I).NE."DELETE") WRITE(5,70)(EDAT(K,I),K=1,8)
58 IOUT=I
59 GO TO 40
60 102 IF (IOUT.LT.NUP) WRITE(5,70)((EDAT(M,N),M=1,8),
N=IOUT+1,NUP)
61 CLOSE(4);CLOSE(5)
62 *
63 OPEN(*,"COMALF")
64 PAUSE
65 CLOSE(*)
66 70 FORMAT(S)
66.5 END
66.8
66.9 *FUNCTION TO ELIMINATE BLANKS
66.95
67 STRING FUNCTION PACK(STR)(30)
68 STRING STR(*),S3(30)
68.5 S3=STR
69 1 I=INDEX(S3,' ')
70 IF (I)[S3=LEFT(S3,I-1)+SUBSTR(S3,I+1);GO TO 1]
71 PACK =S3
71.5 RETURN
72 END
>
```

Note that a Command File is called to copy ALPHEM to EMPLOYEE:

```
>COPY COMALF TO TEL
COPY ALPHEM TO EMPLOYEE

>
```

SAMPLES OF PROGRAM EXECUTION

```
>RUN
WHAT FILE DO YOU WISH TO UPDATE?
@DISTRICT
ENTER UPDATES TO THE @DISTRICT FILE IN THE FOLLOWING FORM:
DISTRICT CODE, DISTRICT NAME
TO TERMINATE INPUT, TYPE AN *
SE,SEATTLE
SD,SAN DIEGO/ORANGE CTY
*


(@46 )>COPY @DISTRICT TO TEL
DC,WASHINGTON DC
PA,PALO ALTO
BO,BOSTON
NY,NEW YORK
ME,METROPOLITAN
DA,DALLAS
LA,LOS ANGELES
SF,SAN FRANCISCO
SE,SEATTLE
SD,SAN DIEGO/ORANGE CTY

(@46 )>RUN
WHAT FILE DO YOU WISH TO UPDATE?
@JOB
ENTER UPDATES TO THE @JOB FILE IN THE FOLLOWING FORM:
JOB CODE, JOB NAME
TO TERMINATE INPUT, TYPE AN *
TA,TECHNICAL ANALYST
TY,TYPIST
TS,TYPE SETTER
*


(@46 )>COPY @JOB TO TEL
SEC,SECRETARY
VP,VICE PRESIDENT
PW,PROGRAMMER WRITER
PR,PROGRAMMER
SR,SALES REPRESENTATIVE
DM,DISTRICT MANAGER
CO,COMPUTER OPERATOR
ME,MAINTENANCE ENGINEER
TA,TECHNICAL ANALYST
TY,TYPIST
TS,TYPE SETTER

(@46 )>RUN
WHAT FILE DO YOU WISH TO UPDATE?
@EMPLOYEE
ENTER UPDATES TO THE @EMPLOYEE FILE IN THE FOLLOWING FORM:
LAST NAME, FIRST NAME, DISTRICT CODE, EMPLOYEE NUMBER,
JOB CODE, HOME ADDRESS, CITY AND STATE, PHONE NUMBER
```

```
TO DELETE TYPE: LAST NAME, FIRST NAME,
 EMPLOYEE NUMBER, DELETE

TO RETYPE LINE, TYPE AN ALTMODE
TO TERMINATE INPUT, TYPE AN *
ENTRY:
VALE,SALLY,BO,286,SEC,934 INLAND DR,BOSTON MASS,343-9886
ENTRY:
JOHNSON,JON,BO,143,SR,877 ALTA WAY APT. 2,BOSTON MASS,543-9855
ENTRY:
BAKER,ROBERT,BO,135,SR,34 WEST ALVIN ST.,BOSTON MASS,984-6545
ENTRY:
KRUMMET,L.L.,LA,322,PR,832 AVENUE I,VENICE CA,999-3598
ENTRY:
ROLIM,E.A.,LA,874,PR,113 VALLEY ST,LOS ANGELES CA,877-9656
ENTRY:
CARDOS,JOSE,SF,283,PR,18 MARINA DR,OAKLAND CA,349-4345
ENTRY:
OLSON,OLLIE,NY,444,PR,564 NINTH AVE,NEW YORK NY,748-9478
ENTRY:
*


 OLD FILEOK.
*69>
```

DISPLAYING EMPLOYEE INFORMATION

DEFINE THE PROBLEM

The program must display data according to the option specified by the program user. The data is found on three files: @JOB, @DISTRICT, and @EMPLOYEE. The formats for these files are given in the previous sample problem, UPDATING FILES.

The output options available to the program user are ROSTER, NAME, DISTRICT, and JOB. With the last two, the user may specify particular districts or jobs, or he may request ALL. With the NAME option, the user may specify any name or names, or he may request ALL. For all options except ROSTER, the user may request a COMPLETE or PARTIAL list of data for each employee.

SUPER FORTRAN CODE

```
>FAST
5 INTEGER DCNT
10 STRING NM(50)(30)
15 STRING FMT(70), JB(50,2)(25)
20 STRING DST(30,2)(40)
25 STRING OPT(4)(8),LIST(8),OPTION(8)
,EDATA(8,50)(30)
30 STRING OPT2(8),JOB(30)(25),DSIN(20)(40)
35 DATA (OPT(K),K=1,4)/ROSTER,NAME,DISTRICT,JOB/
40 FMT ='(S,", ",S,T24,S4,S4,S3,S/,T35,S/,T35,S27)'
45 ICNT=300
50
55 * PROCESS OPTION
60
65 ACCEPT "OPTION=",OPTION
70 N1=2; N2=6
75 IF (OPTION.EQ.OPT(1))(FMT=LEFT(FMT,14)+'S,", ",S/,T24,S27)';
GO TO 1)
80 N1=5
90 DO 10 K=2,4
95 IF (OPTION.EQ.OPT(K)) KODE=KODE+K
105 10 CONTINUE
110 IF (ISK) GO TO 105
115 ISK=1
120 WRITE(1) ""
125 105 ACCEPT "COMPLETE OR PARTIAL LIST?",LIST
130 WRITE(1) ""
135 IF (LIST.EQ.'PARTIAL')(N2=8;FMT=LEFT(FMT,23)+RIGHT(FMT,
4))
140 IF (KODE.EQ.2) GO TO 2
145 IF (KODE.EQ.3) GO TO 3
150 IF (KODE.EQ.4) GO TO 4
155
160 * PRINT BY DISTRICT
165
169 3 DISPLAY "TYPE IN DISTRICT CODES "
```

```
170 DO 5 M=1,20
172 ACCEPT DSIN(M)
174 IF ((DSIN(M).EQ."ALL").OR.(DSIN(M).EQ."END"))(ND=
M-1;GO TO 1)
175 5 CONTINUE
185 1  OPEN(4,"@DISTRICT",INPUT)
190 READ(4,END=71)((DST(I,J),J=1,2),I=1,30)
195 71 DCNT=I-1
200 CLOSE(4)
205 145  DO 502 N=1,DCNT
210 IF (KODE.EQ.0) GO TO 555
215 DO 73 M=1,ND
220 IF (DSIN(M).EQ.DST(N,1)) GO TO 555
225 73 CONTINUE
230 IF (DSIN(1).NE."ALL") GO TO 502
235 555 WRITE(1) ""
240 WRITE(1,200) DST(N,2)
245 WRITE(1) ""
255 OPEN(2,'@EMPLOYEE',INPUT)
260 144 DO 500 L=1,ICNT
265 I=1
270 READ(2,END=501)(EDATA(J,I),J=1,8)
275 IF (EDATA(3,I).NE.DST(N,1)) GO TO 500
280 WRITE(1,FMT)(EDATA(K,I),K=1,N1),(EDATA(K,I),K=N2,8)
285 500 CONTINUE
290 501 CLOSE(2)
295 502 CONTINUE
300 STOP 1
305
445
450 * PRINT BY NAME
455
460 2 DISPLAY "LAST NAME(S) REQUIRED; "
465 DO 20 L=1,50
470 ACCEPT NM(L)
475 IF ((NM(L).EQ."ALL").OR.(NM(L).EQ."END"))(LLL=L-1;GO TO 600)
480 20 CONTINUE
485
490 * READ FROM @EMPLOYEE FILE
495
500 600 OPEN(2,"@EMPLOYEE",INPUT)
505 601 MORE=0
510 READ(2,END=603)((EDATA(I,J),I=1,8),J=1,50)
515 MORE=1
520 603 ICNT=J-1
525 DO 123 L=1,LLL
530 607 DO 122 J=1,ICNT
535 IF ((NM(1).NE."ALL").AND.(NM(L).NE.EDATA(1,J))) GO TO 122
540 WRITE(1,FMT)(EDATA(I,J),I=1,5),(EDATA(I,J),I=N2,8)
545 122 CONTINUE
550 123 CONTINUE
555 IF (MORE) GO TO 601
560 CLOSE(2)
565 STOP 2
```

```
570
575 * PRINT BY JOB
580
585 4 OPEN(3,"@JOB",INPUT,SYMBOLIC)
590 READ(3,END=30)((JB(I,J),J=1,2),I=1,50)
595 30 JCNT=I-1
600 CLOSE(3)
605 DISPLAY "TYPE IN JOB CODES"
610 DO 25 N=1,25
615 ACCEPT JOB(N)
620 IF ((JOB(N).EQ."ALL").OR.(JOB(N).EQ."END"))(NP=N-1;GO TO 15)
622 25 CONTINUE
625 15 DO 335 L=1,JCNT
630 DO 332 N=1,NP
635 IF(JOB(N).EQ.JB(L,1)) GO TO 99
640 332 CONTINUE
645 IF (JOB(1).NE."ALL") GO TO 335
650 99 WRITE(1) ""
655 WRITE(1,200) "POSITION:",JB(L,2)
660 WRITE(1) ""
665 OPEN(2,'@EMPLOYEE',INPUT)
670 DO 333 I=1,ICNT
675 M=1
680 READ(2,END=334)(EDATA(J,M),J=1,8)
685 IF (EDATA(5,M).NE.JB(L,1)) GO TO 333
690 WRITE(1,FMT)(EDATA(K,M),K=1,5),(EDATA(K,M),K=N2,8)
695 333 CONTINUE
700 334 CLOSE(2)
705 335 CONTINUE
710 STOP "END"
712 200 FORMAT(S,1X,S)
715 END
>
```

SAMPLES OF PROGRAM EXECUTION

```
>RUN
OPTION=ROSTER

WASHINGTON DC

ADAMS, ANSEL            211 HAVEN AVE, WASHINGTON DC
                        399-8474

PALO ALTO

EVERS, EVERETT          890 RAND ST, PALO ALTO CA
                        983-3333

BOSTON

BAKER, ROBERT           34 WEST ALVIN ST., BOSTON MASS
                        984-6545
INGLES, INGRID          38 FREEDOM WAY, BOSTON MASS
                        233-3355
JOHNSON, JON            877 ALTA WAY APT. 2, BOSTON MASS
                        543-9855
VALE, SALLY             934 INLAND DR, BOSTON MASS
                        343-9886

NEW YORK

OLSON, OLLIE            564 NINTH AVE, NEW YORK NY
                        748-9478

METROPOLITAN

UPDIKE, URSULA          881 BROADWAY, NEW YORK NY
                        211-5467

DALLAS

LOS ANGELES

KRUMMET, L.L.           832 AVENUE I, VENICE CA
                        999-3598
ROLIM, E.A.             113 VALLEY ST, LOS ANGELES CA
                        877-9656

SAN FRANCISCO

CARDOS, JOSE            18 MARINA DR, OAKLAND CA
                        349-4345

SEATTLE

SAN DIEGO/ORANGE CTY

1
(@300 )>
```

```
>RUN
OPTION=NAME

COMPLETE OR PARTIAL LIST?COMPLETE

LAST NAME(S) REQUIRED;
JOHNSON
BAKER
END
JOHNSON, JON            BO   143 SR 877 ALTA WAY APT. 2
                                    BOSTON MASS
                                    543-9855
BAKER, ROBERT           BO   135 SR 34 WEST ALVIN ST.
                                    BOSTON MASS
                                    984-6545

2
(@565 )>


>RUN
OPTION=JOB

COMPLETE OR PARTIAL LIST?PARTIAL

TYPE IN JOB CODES
PR
END

POSITION: PROGRAMMER

CARDOS, JOSE            SF   283 PR 349-4345
KRUMMET, L.L.           LA   322 PR 999-3598
OLSON, OLLIE            NY   444 PR 748-9478
ROLIM, E.A.             LA   874 PR 877-9656

END
(@710 )>
```

```
>RUN
OPTION=DISTRICT

COMPLETE OR PARTIAL LIST?PARTIAL

TYPE IN DISTRICT CODES
BO
NY
END

BOSTON

BAKER, ROBERT              BO   135 SR 984-6545
INGLES, INGRID             BO   333 SEC233-3355
JOHNSON, JON               BO   143 SR 543-9855
VALE, SALLY                BO   286 SEC343-9886

NEW YORK

OLSON, OLLIE               NY   444 PR 748-9478

1
(0300 )>
```

APPENDIX A

STORAGE ALLOCATION

The computer has a specified amount of memory available for both program and data. The total number of statements and amount of data that can be used can be greatly increased by linking programs together; that is, running one program, loading another binary program from a file and running it, and so forth.

Approximately 8000 words of memory are available for each program and its data. The amount of storage required for a program and for data may be found using the CCS command MAP.

The following values show the allocation of storage:

1 Real Number = 2 Words

1 Integer Number = 1 Word

1 Complex Number = 4 Words

1 Double Precision Number = 3 Words

1 String = 1/3 Word times the declared length of the string (rounded up to a whole number)

Thus, if a program contains the real array A(130,20), 5200 words would be required for data storage, leaving 2800 words for program storage.

The actual program size and storage used for data may be determined by using the CCS command MAP. The MAP command prints a table as shown in the example below.

```
>MAP↵
SOURCE PROGRAM:
13 LINES
TEXT = 206 CHARS (OF 24000)
COMPILATION = 148 BYTES (OF 18000)
NAMES = 6
(0,0)

OBJECT PROGRAM:
SIZE = 103 WORDS
COMMON = 0 WORDS
DATA STORAGE = 8 WORDS
7988 WORDS UNUSED

>
```

If a program exceeds the maximum size allowed, it must be reduced in size before it will run. The program size can be reduced either by deleting statements from the program or by reducing the bounds of dimensioned variables.

APPENDIX B

INTERNAL REPRESENTATION OF ASCII CODE

Representation		Character	Representation		Character
Decimal	Octal		Decimal	Octal	
00	00	Space	32	40	@
01	01	!	33	41	A
02	02	"	34	42	B
03	03	#	35	43	C
04	04	$	36	44	D
05	05	%	37	45	E
06	06	&	38	46	F
07	07	'	39	47	G
08	10	(	40	50	H
09	11	)	41	51	I
10	12	*	42	52	J
11	13	+	43	53	K
12	14	,	44	54	L
13	15	–	45	55	M
14	16	.	46	56	N
15	17	/	47	57	O
16	20	0	48	60	P
17	21	1	49	61	Q
18	22	2	50	62	R
19	23	3	51	63	S
20	24	4	52	64	T
21	25	5	53	65	U
22	26	6	54	66	V
23	27	7	55	67	W
24	30	8	56	70	X
25	31	9	57	71	Y
26	32	:	58	72	Z
27	33	;	59	73	[
28	34	<	60	74	\
29	35	=	61	75	]
30	36	>	62	76	↑
31	37	?	63	77	←

NOTE 1) The ASCII codes for control characters can be obtained by adding the decimal number 64 or the octal number 100 to the appropriate representation for the specific alphabetic character. For example, since the code for A is decimal 33 or octal 41, the code for Control A is decimal 97 or octal 141.

2) A Line Feed followed by a Carriage Return may be generated by a Control J. A Carriage Return followed by a Line Feed may be generated by a Control M.

APPENDIX C
EXECUTIVE SUMMARY

ENTERING THE SYSTEM

To gain access to the Tymshare system, the user must log in.

As soon as the computer has answered, place the telephone handset into the MARK V data modem, press the ORIGINATE button on the MARK V, and turn on your terminal. If your terminal is a Teletype Model 33 or 35, the system will ask that the identifying character D be typed. *NOTE: For the identification character for other terminals, consult your local Tymshare representative.* The system will then type:

PLEASE LOG IN:

The user may respond with: *Optional* ↓

account number password; user name; project code

to log in quickly, or he may respond with a Carriage Return to be prompted by the system. The system replies with:

ACCOUNT: A3 ↵

The user types his account number and a Carriage Return. The system responds with:

PASSWORD: ↵

The user types his password followed by a Carriage Return. The password does not print. The system types:

USER NAME: JONES ↵

The user types his user name followed by a Carriage Return. The system then types:

PROJ CODE: S122 ↵

The user types his project code and a Carriage Return. The project code is optional. The user may simply respond with a Carriage Return.

The system will now type:

TYMSHARE 12/8 11:20

–

The dash indicates that the user is in the EXECUTIVE and may give any EXECUTIVE command. Calling a language is an EXECUTIVE function. Thus, typing

–SFORTRAN ↵ *(or SFOR* ↵*)*

>

calls SUPER FORTRAN which acknowledges with a > indicating it is ready to accept a command.

RULES FOR NAMING FILES

1. File names that are not surrounded by slashes or quotes may contain only A through Z, 0 through 9, and @. Thus, @Z1 is a valid file name.
2. A file name may begin and end with single quote marks. Inside the quote marks can be any characters (including control characters) except a single quote mark.
3. A file name may begin and end with a slash, and may contain any characters (including control characters) except a slash. For example, /@Z1/ or /G^{C};/ are valid file names.

METHODS OF CREATING SYMBOLIC DATA FILES

1. Read from paper tape.
2. Using the EXECUTIVE COPY command.

–COPY TEL TO /#'S/ ↵
NEW FILE ↵
NO. 1, 1.87 ↵
NO. 2, 3.15 ↵
D^{C}

3. Using the EDITOR commands READ, WRITE, and APPEND.

***READ /#'S/** ↵
24 CHARACTERS
***APPEND** ↵
NO. 3, 4.22 ↵
NO. 4, 2.53 ↵
D^{C}
***WRITE** ↵↓
TO 'NO/S' ↵
NEW FILE ↵
48 CHARACTERS

4. Using the SUPER FORTRAN commands READ, WRITE, and OPEN.

```
OPEN(3,"F13",RANDOUT,SYMBOLIC)
WRITE(3,100)X,Y,Z
```

or

```
WRITE(3)X,Y,Z
```

for free form output.

5. Using the CCS command, COPY.

>COPY 5:100 TO NUMS

APPENDIX D

SUPER FORTRAN LANGUAGE SUMMARY

CONSTANTS

Type	Examples
Integer	0, -15, 425
Real	0., 1.9, -.127, 1.7E2, -5E+4, 1.2E-3
Complex	(-3, 5.2), (1.8, .5E2)
Logical	.TRUE. .FALSE.
Hollerith	3HYES, 6HAB CDE
String	"MEAN", 'CODE4'

VARIABLES

Types: Integer, Real, Double Precision, Complex, Logical, String

Names: No more than 31 characters beginning with a letter of the alphabet.

Scalar Variable: N, ALPHA, XX, R1

Subscripted Variable: A(2), B(-2,4), C(1,1,1)

ARITHMETIC OPERATORS
(In Order Of Priority)

** or ↑	Exponentiation
–	Unary Minus
* and /	Multiplication and Division
+ and -	Addition and Subtraction

RELATIONAL OPERATORS

.EQ.	Equal to
.NE.	Not equal to
.LT.	Less than
.LE.	Less than or equal to
.GT.	Greater than
.GE.	Greater than or equal to

LOGICAL OPERATORS
(In Order Of Priority)

.NOT.	
.AND.	
.EQV.	Equivalence
.IMP.	Implication
.OR.	Inclusive OR
.EOR.	Exclusive OR

EXPRESSIONS

Arithmetic Expressions	Logical Expressions
45	A .EQ. B
X	X-5 .LT. C*10
A(4)	(G .GT. 5) .AND. (Y .LE. 4)
SQRT(Y)	
FUNT(3,4)	
N/(I+2.)	
D-B*C	

REPLACEMENT STATEMENTS

Model	Examples
variable=expression	X=(Y+Z)/HEIGHT
	AREA(I)=(I-1)*SQRT(B)
	A=.TRUE.
	B(2)=A .AND. C
	(SUM=SUM+A(I)),I=1,20

CONTROL STATEMENTS

statement label = 1 to 99999

Models	Examples
IF (logical expression) statement	IF (A .GT. 100) (P=1;GO TO 10)
IF (arithmetic expression) n_1, n_2, n_3	IF (C-D) 10,10,20
where n_1, n_2, n_3 are statement labels.	
IF (expression) statement .ELSE. statement	IF (A .EQ. 5) GO TO 10 .ELSE. GO TO 20
DO statement label variable=initial value, final value, increment	DO 10 I=1,10,.2 DO 35 J=9,3,-1 DO 20 M=J,10
GO TO statement label	GO TO 25
GO TO (n_1,n_2,n_3), expression	GO TO (10,25,40),K
where n_1, n_2, n_3 are statement labels	
ASSIGN statement number TO variable	ASSIGN 10 TO L
GO TO variable, (n_1, n_2, ..., n_k)	GO TO L,(10,20,30)
where n_1 to n_k are a list of statement labels.	
ON INTERRUPT GO TO statement label	ON INTERRUPT GO TO 50
OFF INTERRUPT	OFF INTERRUPT
CONTINUE	10 CONTINUE
PAUSE "text" *or* number	PAUSE "NOW, CONTINUE"
STOP "text" *or* number	25 STOP 4
QUIT "text" *or* number	QUIT "DONE"
END	END

INPUT/OUTPUT STATEMENTS

Free Format Input/Output

Model	Examples
ACCEPT variable list or literal text	ACCEPT "X=",X, 'NUMBER OF VALUES=',N
DISPLAY variable list or literal text	DISPLAY "TOTAL IS",S,R+5,(S(I),I=1,K)
READ (file number) variable list	READ (0) AB,SOL READ (2) DATA
WRITE (file number) variable list	WRITE (1) (A(I),I=1,N) WRITE (5) STS, RDS

Formatted Input/Output

Field Specification

Type	Example	Sample Value
I - Integer	I3	123
F - Real (fixed point)	F7.2	1487.25
E - Real (floating point)	E8.1	.3E+08
G - Real (general)	G12.3	.244E+4
L - Logical	L1	T or F
A - Alphanumeric	A5	JONES
H - Hollerith	4HTIME	TIME
S - String variables	S5	AB123
X - Spacing	5X	spaces 5 times
T - Tabs	T20	tabs to print position 20
P - Scaling factor	3PF5.2	scales value by factor of 10^3
/ - Generates a Carriage Return		
& - Suppresses a Carriage Return		

FORMAT statement

Model	Examples
statement number FORMAT (S_1 ,S_2 ,. . . ,S_k) where S_1 to S_k are field specifications.	100 FORMAT (I2,F8.3,E10.1/4(I4,G9.2)) 25 FORMAT (S4,F12.3,6HMETERS)

Terminal Input/Output

File number 0 refers to the terminal in the READ statement; file number 1 to the terminal in the WRITE statement.

Models	Examples
READ(0,format number, error condition) variable list	READ(0,100,ERR=5)N,(A(I),I=1,100) 100 FORMAT(I5/F10.3)
WRITE(1,format number, error condition) variable list	WRITE(1,35)I,R+5,DEC 35 FORMAT(I6,F12.4,A5)

The error condition is optional.

File Input/Output

Opening A File

Four files can be open at a time. File numbers 0 and 1 are reserved for terminal input/output.

Model

OPEN (file number, "file name", [INPUT / OUTPUT / RANDIN / RANDOUT / RANDIO] , [SYMBOLIC / BINARY] , error condition)

Examples

```
OPEN(3,"QUAD", RANDIN, SYMBOLIC)
OPEN(5,"ROOTS", OUTPUT, SYMBOLIC)
OPEN(4,"STOR", RANDIO, BINARY,ERR=100)
```

Closing A File

Model	**Example**
CLOSE (file number)	CLOSE (3)

Symbolic File Input/Output

Models

READ(file number, format number, end of file and/or error condition) variable list

Examples

```
READ(3,25,END=100,ERR=5) A,(BC(I),I=1,N)
READ(2,FMT)S1,S2
```

where FMT may be entered as input.

WRITE(file number, format number, error condition) variable list

```
WRITE (2,10,ERR=15)R1,R2+4
WRITE (5,50) ARAY1
```

The end of file and error conditions are optional. The format statement label is omitted for free format I/O.

Binary File Input/Output

Models	**Examples**
READ(file number) variable list	READ(4)(Y(I),I=1,1000)
WRITE(file number) variable list	WRITE(3) AGE, VOL

Random File Functions

Models	**Examples**
POSITION(file number) *Finds position in an opened random file.*	K=POSITION(2)
SIZE(file number) *Finds size of an opened random file.*	NR=SIZE(5)-20

Random File Statements

Models	Examples
POSITION(file number, position number) *Sets position in an opened random file*	POSITION(5,SIZE(5)-10) POSITION(2,1)
ERASE(file number) (first position, last position)	ERASE(3) (1,100) ERASE(4) (256,*) *Erases from position 256 to the end of file 4.*

DECLARATION STATEMENTS

C: THIS IS A COMMENT
* THIS IS ANOTHER COMMENT

Models	Examples
DATA $variable_1$ /$value_1$ /,$variable_2$ /$value_2$ /,...,$variable_n$/$value_n$/	DATA A/4.1/,B/2E7/,I/100/
DATA $variable_1$, $variable_2$, . . . , $variable_n$/$value_1$, $value_2$, . . . , $value_n$/	DATA A,B,I/4.1,2E7,100/ DATA (Z(I),I=1,5)/1,2,3,4,5/
DIMENSION array name (maximum size of each subscript)	DIMENSION HT(60),BP(15,20)
DIMENSION array name (lower limit:upper limit of each subscript)	DIMENSION A(-2:10, 0:20),CAM(3, 0:4, -2:5)
COMMON variable list	COMMON H, P(20,20)/LABL/R,T//A
INTEGER variable list	INTEGER A, POUND, GAL, R(5)
REAL variable list	REAL I(10), MIN, MAX(-2:4)
DOUBLE PRECISION *or* LONG variable list	DOUBLE PRECISION AR, X(20) LONG DPX
COMPLEX variable list	COMPLEX X,BAT,COM(25),J
LOGICAL variable list	LOGICAL FIR,SEC,TR(4)
EQUIVALENCE (variable list), (variable list) . . .	EQUIVALENCE (A(1), LIST(155)),(A,B,C)
STRING variable list	STRING NAME(15), ADD(30), SARAY(3,5)(10)

SUBPROGRAMS

Statement Function

Models	Examples
Function name (dummy arguments)=expression	SRT (A,B)=A**2-4*B *Reference in main program:* Y=SRT(3,4) CHIS (X,Y,Z)=X+Y-Z+SQRT(M) *Reference in main program:* DISPLAY CHIS(3.1,4.2,5)

FUNCTION Subprogram

Models

```
FUNCTION function name (dummy arguments)
function name=expression
  .
  .
  .
END
```

Examples

```
FUNCTION TAG(A,B)
IF (A .LT. 200) GO TO 50
TAG=SQRT(A+B)
RETURN
50 INT=0
RETURN
END
DISPLAY TAG(R,X)        Call in main program.
```

```
type FUNCTION name (arguments)
function name=expression
  .
  .
  .
END
```

```
REAL FUNCTION I(X)
I=X**2
END
```

The function type may be any variable type: REAL, DOUBLE PRECISION, and so on.

Subroutine Subprogram

Models

```
SUBROUTINE name
  .
  .
  .
END
```

Examples

```
SUBROUTINE PRINT
DISPLAY "CHECK DATA"
END
```

```
SUBROUTINE name (dummy arguments)
  .
  .
  .
END
```

```
SUBROUTINE TR(N,A,TOTAL,MEAN)
  .
  .
  .
END
```

```
CALL subroutine name
CALL subroutine name (actual arguments)
```

```
CALL PRINT
CALL TR(10,B,SUM,MN)
```

LINKING

Model

Direct:

(@line number)>@LINK "file name"

Indirect:

>line number LINK "file name"

Example

(@100)>@LINK "K2"

>100 LINK "BPROG"

COMMAND FILE IN A FORTRAN PROGRAM

Model	Example
CLOSE(*)	CLOSE(*)
OPEN(*,"file name")	OPEN(*,"TEST")
READ(*,format number) variable list	READ(*,100) N,(A(I),I=1,N)

APPENDIX E

CCS SUPER FORTRAN COMMANDS SUMMARY

Line Number = .001 to 999.999

ENTERING A PROGRAM

From The Terminal

Single Statement Entry

```
>ENTER 10.2 Y=Z+1 ↵
>12 DISPLAY X,Y ↵
```

Multiple Statement Entry

```
>1:10 ↵
@DO 25 I=1,5 ↵
@A(I)=0 ↵
@B(I)=C(I)-1 ↵
@25 CONTINUE ↵
@D^C
>
```

Multiple Statement Entry With Prompted Line Numbers

```
>10(10) ↵
10 A=3 ↵
20 B=A+SQRT(X) ↵
30 D^C
>
```

From A Symbolic File Without Line Numbers

```
>COPY file name TO line range ↵
```

Example

```
>COPY ABC TO 1(5)800 ↵
```

From A Symbolic File With Line Numbers

```
>LOAD file name ↵
```

Clears current program and then loads file.

Example

```
>LOAD CHI1 ↵
```

```
>MERGE file name ↵
```

Merges by line number the contents of the symbolic file with any current program.

Example

```
>MERGE MON@ ↵
```

From A Binary File

> LINK file name ↵ — *Loads binary version of program and executes it immediately. COMMON is set to zero.*

> LOAD file name ↵ — *Loads source and binary programs.*

Example

> LINK /ACCT#/ ↵

From Paper Tape

> LOAD TEL ↵ — *Statements must have line numbers.*

> COPY TEL TO line number range ↵ — *Statements must not have line numbers.*

Example

> COPY TEL TO 1:200 ↵

SAVING A PROGRAM

> SAVE file name ↵

TEXT ONLY?Y ↵

Saves only symbolic version of program.

OLD / *or* / NEW — FILE ↵

OK.
>

> SAVE file name ↵

TEXT ONLY?N ↵

Saves symbolic and binary versions.

OLD / *or* / NEW — FILE ↵

OK.
>

COPYING

> COPY file name *or* line list TO file name *or* line range ↵ — *File name can be TEL.*

Examples

> COPY XYZ TO 10(5)200 ↵ — *Copies file XYZ to lines 10 through 200 in increments of 5.*

> COPY *1:100 TO 'RT/FL' ↵

> COPY 5:$ TO 75:90 ↵

The command MOVE is the same as COPY except it deletes the source after it is moved.

Examples

> MOVE 1:100 TO /1'ST/ ↵

> MOVE5:20 TO 75:90 ↵ — *A space is unnecessary between the command and the line range.*

> MOVE .:$ TO CR@ ↵

LISTING

Formatted Listing

> LIST ↵

Quick Listing

> FAST ↵

Lists entire program on terminal.

> LIST line list ↵ **> FAST line list ↵**

Lists specified lines on terminal.

> LIST line list TO file name ↵ **> FAST line list TO file name ↵**

Lists specified lines on file named.

Examples

> LIST 10:100 ↵

> FAST 25 ↵

> LIST 50:$ TO LFIL ↵

> FAST*1 ↵

RENUMBERING

> RENUMBER ↵ *Reassigns line numbers starting at 1 in steps of 1.*

> RENUMBER line number range ↵ *Reassigns line numbers in the range specified in increments of the first of 1, .1, .01, and .001 which fits the specified range.*

> RENUMBER line number range AS line number range ↵

Examples

> RENUMBER 2.1 AS 25 ↵ *or* **> RENUMBER 2.1,25 ↵**

> RENUMBER 1:15 AS 10(5)50 ↵ *or* **> RENUMBER 1:15,10(5)50 ↵**

DELETING A PROGRAM OR STATEMENTS

> DELETE line list ↵ *Deletes the lines specified.*

> CLEAR ↵ *or* **DELETE ↵** *Deletes the entire program.*

ERASE PROGRAM?Y ↵
OK.
>

Examples

> DELETE 5.3 ↵

> DELETE *1:$-2 ↵

EXECUTING A PROGRAM

> RUN ↵ *or* **> EXECUTE ↵** *Executes a symbolic program.*

> LINK binary file name ↵ *Enters and executes binary program on file named.*

DEBUGGING

> REFERENCES [variable name / *or* / statement label] line list ↵

The line list is optional. Prints lines containing variable name or statement label specified in the range specified.

Examples

> REFERENCES Z 10:50 ↵

> REFERENCES 30 ↵

> DEFINITIONS [variable name / *or* / statement label] line list ↵

Prints declarative statements which contain the variable name or statement label specified. The line list is optional.

Examples

> DEFINITIONS ↵ — *Prints all declarative statements.*

> DEFINITIONS 20 100:$ ↵

> DEFINITIONS CARGO *1:200 ↵

> SET list of breakpoints ↵
or
> BREAK list of breakpoints ↵

Sets breakpoints as specified in the list.

Examples

> SET ↵ — *Lists all breakpoints in the program.*

> SET 8,10:12 ↵

> BREAK 15,22,100 ↵

> RESET ↵ — *Clears all breakpoints.*

> RESET list of breakpoints ↵ — *Clears breakpoints specified in the list.*

Example

> RESET 20,45:50 ↵

line number >CONTINUE ↵ — *Execution resumes at line number.*

line number >@ statement ↵ — *The direct statement is executed immediately.*

Example

(@30)>@X(1)=0 ↵

line number$_1$ >AT line number$_2$ ↵ — *Moves orientation of direct statements to block containing line number$_2$.*

Example

15 >AT 55 ↵ — *The direct statement @DISPLAY VALUE refers to the variable value as it is defined in the subprogram containing line 55.*

(@55)15 >@DISPLAY VALUE ↵

line number >NEXT ↵ — *Executes statement with the given line number.*

> RUN TO list of breakpoints ↵ — *Clears and sets new breakpoints as specified in the list, then begins execution.*

Example

> RUN TO 5,10 ↵

line number >CONTINUE TO list of breakpoints ↵

Clears and sets new breakpoints as specified in the list, then continues execution at line number.

Example

25 >CONTINUE TO *12:50,$-3 ↵

EDITING

> EDIT line list ↵ — *Prints and allows editing, line by line.*

> MODIFY line list ↵ — *Allows line by line editing, but does not print the line to be edited.*

Examples

> EDIT25 ↵

> EDIT *10,55:60,$-1 ↵

> MODIFY 25,70 ↵

OTHER COMMANDS

> CHECK ↵ — *Checks to see whether or not program is executable.*

> INITIALIZE ↵ — *Prepares program for execution. Performs all declarations and data statements. Can be followed by direct statements.*

> COMMANDS file name ↵ — *Takes commands from the file named.*

> MAP ↵ — *Describes program resource utilization.*

> QUIT ↵ — *Returns to the EXECUTIVE.*

APPENDIX F

USER AIDS

ABBREVIATED COMMANDS

To save time, any CCS command may be abbreviated to as few characters as necessary to identify the command uniquely. Thus the command FAST may be abbreviated simply as

>F ↵

since there are no other commands that start with the letter F. But to list a program, at least

>LIS ↵

is required since the command LINK also starts with the letters LI.

If insufficient identification is given to a command, the message

AMBIGUOUS COMMAND, PLEASE TYPE MORE CHARACTERS

will be printed.

ELIMINATING AS AND TO FROM COMMANDS

A comma may be used in place of the words AS and TO in any CCS command as long as no ambiguity results. For example,

>COPY /A/,/B/

may be used instead of

>COPY /A/ TO /B/

but

>COPY 1,10,15,/A/

is not allowed since it is not evident which comma replaces the TO.

SPACES IN CCS COMMANDS

A space is usually required between parts of a command. For example, typing

LOADPROG ↵

to load the file named PROG causes an error diagnostic; the user must type

LOAD PROG ↵

However, spaces may be omitted between alphabetic and nonalphabetic characters. For example,

>LIST5,15,20:100 ↵

is legal.

Spaces are not allowed in the actual line list, since a space terminates a line list. For example,

>LIST 5,10:17,20:30 ↵

is allowed, but

>LIST 5,10:17, 20:30 ↵

is not.

NOTE: Spaces are not allowed in the variable name in either the REFERENCES or DEFINITIONS commands. For example,

>REFERENCES ALPHA1 1:100 ↵

is allowed, but

>REFERENCES ALPHA 1 1:100 ↵

is not.

COMMAND MODELS

If the user is not certain how to use a particular command, he may type a question mark after the command name. A model of the command is then typed. For example,

```
>COPY ? ↵
SHOULD BE:
COPY <FILE OR LINES> [ TO <FILE OR LINE-RANGE> ]

>
```

A REVIEW OF CCS PROMPTS

The following is a summary of the various prompts that may be printed by CCS. In the table,

l_1 and l_2 are line numbers.

*n indicates that an interruption occurred just before the nth line of the program.

i is an increment.

PROMPT	WHEN PRINTED	MEANING
>	Just after SUPER FORTRAN is called; after certain kinds of errors, after various CCS commands, such as DELETE; after binary program execution is interrupted by a STOP statement, the end of the program, or by certain errors.	CCS is ready to accept a command (but not a direct statement).
(@line number)>	When symbolic program execution is interrupted by a STOP statement or the end of the program.	CCS is ready to accept a command or a direct statement (a SUPER FORTRAN statement preceded by an @).
line number >	When symbolic program execution is interrupted by certain kinds of errors, ALT MODE, a breakpoint, or a PAUSE; after the commands NEXT and INITIALIZE.	CCS will accept a CCS command or a direct statement; in addition the CONTINUE command may be used to resume execution at the line number printed in the prompt.
*n >	After a continuable interruption of a linked binary program.	The CONTINUE command will continue execution at nth line in program; NEXT will execute nth line. No direct statements or breakpoints are accepted.
(@l_1)l_2 >	After the @ point has been changed with the AT command.	CCS will accept a command or a direct statement. Variables defined in the program block containing line l_1 may be referred to with direct statements. Typing CONTINUE resumes execution at line l_2.
@	During the command **> ENTER l_1:l_2** ↵	CCS will accept a SUPER FORTRAN statement[1]. The statement will be placed in the current program with a line number in the range l_1:l_2.
line number	During the command **> ENTER l_1 (i) l_2** ↵ or **> ENTER l_1 (i)** ↵	CCS will accept a SUPER FORTRAN statement[1]. The statement will be placed in the current program with the **line number** prompted.

1 - Or a group of executable SUPER FORTRAN statements separated by semicolons.

INDEX

NOTE: Page numbers which appear in **bold** *face type refer to those pages where the listed item receives the most detailed discussion.*