Program Product

IBM System/360 OS(TSO) ITF: BASIC Terminal User's Guide

Program Numbers: 5734-RC2
5734-RC4

This publication provides tutorial information and reference material for users of the BASIC language component of the Interactive Terminal Facility (ITF), a Program Product that operates under the Time Sharing Option (TSO) of the System/360 Operating System (OS).

The book tells how to use ITF in the TSO environment, and how to write programs in BASIC; it also includes detailed descriptions of the BASIC language elements, a subset of the TSO command language, as well as error recognition and correction. Sample programs and examples of the use of ITF:BASIC appear throughout the text.

This publication is intended for the TSO ITF:BASIC terminal user. No previous knowledge of programming or of the BASIC language is required.

IBM

This publication is being released prior to the availability of TSO ITF to permit installation planning.

FIRST EDITION (April, 1971)

This edition corresponds to Release 1 of TSO ITF.

Changes are continually made to the information herein; before using this publication in connection with the operation of IBM systems, consult the latest IBM System/360 SRL Newsletter, Order No. GN20-0360, for the editions that are applicable and current.

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for readers' comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Programming Publications, 1271 Avenue of the Americas, New York, New York 10020. Comments become the property of IBM.

© Copyright International Business Machines Corporation, 1971

Preface

This publication is both an introduction and a reference guide for users of the IBM System/360 Operating System, Time Sharing Option, Interactive Terminal Facility:BASIC language. There are no prerequisite publications.

IBM's ITF:BASIC is a terminal-oriented programming language which was developed for people who wish to solve problems of a mathematical nature but who do not wish or do not need to become experts in the field of modern computer programming.

The purpose of this publication is threefold: *first,* to lead the user with little or no programming experience simply and easily to the point where he can write elementary programs in ITF:BASIC; *second,* to provide him with enough information to allow him to write more and more advanced programs in ITF:BASIC; and *third,* to acquaint him with the TSO environment so that he can fulfill either or both of the first two purposes by actually using the terminal from the beginning. To aid the user of this publication, the material has been divided into three parts.

- Part I is tutorial. It explains what a program is, tells how to use the terminal, introduces the use of modes in TSO, shows step-by-step how to write a program, and introduces the more complex subjects of how to recognize errors, how to create and use files, subroutines, and functions, and how to modify and test programs. Sample programs appear throughout Part I.
- Part II is a source of reference material. It defines each element of an ITF:BASIC statement and the structure of an ITF:BASIC program. It also contains detailed rules, syntactical descriptions, and examples of each ITF:BASIC program statement.
- Part III is also a source of reference material. It contains detailed rules and syntactical descriptions for each command and subcommand introduced in Part I.

For those who are unfamiliar with the IBM syntax notation used in this book, Appendix A briefly describes the meaning of the format symbols. Other appendixes describe the EBCDIC collating sequence, file usage considerations, a summary of the use of the attention interruption in each mode, and the differences between OS ITF and TSO ITF.

A glossary and a complete set of ITF:BASIC error messages are contained at the end of this book. Each message is accompanied by a detailed explanation and the action required by the programmer to correct the error.

The following TSO publication describes the characteristics of all the terminals supported by TSO, and may be required for TSO users in installations that do not provide their own terminal usage instruction.

- *IBM System/360 Operating System, Time Sharing Option, Terminals,* Order Number GC28-6762

Three other TSO publications contain information that may be of interest to the TSO ITF:BASIC user, although they are not required for use of TSO ITF:BASIC. They are:

- *IBM System/360 Operating System, Time Sharing Option Guide,* Order Number GC28-6698
- *IBM System/360 Operating System, Time Sharing Option, Terminal User's Guide,* Order Number GC28-6763
- *IBM System/360 Operating System, Time Sharing Option, Command Language Reference,* Order Number GC28-6732

Contents

Introduction 9
Language 9

Part I. Using TSO ITF:BASIC

What is a Program? 13

Getting Started 15
Terminal Operating Procedures 15
  Starting and Ending a Session 15
    Logging On 15
    Logging Off 16
  The Attention Interruption 16
Keyboard Entry Procedures 17
  Your Entries 17
  Correcting Typing Errors 17
  Character Set 18
Mode Usage 19
  The Edit Mode 19
    Creating a Program 20
    Syntax Checking in the Edit Mode 22
      Using the SCAN Subcommand 23
    Executing a Program in the Edit Mode 23
      Test Execution of Your Programs 24
      Interrupting Execution 24
    Modifying Programs in the Edit Mode 24
    Saving Programs 24
    Terminating the Edit Mode 24
  The Command Mode 25
    Requesting Information on Commands and Subcommands 25
    Sending Messages to Other Terminal Users 26
    Executing a Program in the Command Mode 27
      Test Execution of Permanent Programs 27
      Interrupting Execution in the Command Mode 27
    Other Uses of the Command Mode 27
  The Test Mode 27

Writing a Program 29
Building a BASIC Statement 29
  Constants 29
  Variables 30
Assigning Values to Variables 30
    Varying the Input 32
Expressions and Calculations 33
    Arithmetic Expressions 33
    Character Expressions 35
    Relational Expressions 36
Printing Results 36
Loops 38
    Looping by FOR and NEXT 39
Arrays 40
    Arithmetic Arrays 40
    Character Arrays 42
    Input Values for Arrays 42
    Matrix Operations (MAT Statements) 43
Large and Small Numbers 45

Creating and Using Files 49
    Naming Files 49
      File Name Length 50

Creating a File 50
End-of-file Indicator 50
Activating and Deactivating Files 52
Repositioning Files 53
Using Files 54

Defining Your Own Functions and Subroutines 57
Functions 57
Subroutines 57

Errors and Corrections 61
Program Modification 61
Modifications in the Edit Mode 61
Deleting Statements 61
Inserting and Replacing Statements 62
Adding Statements to the End of Your Program 62
Changing Parts of Statements 63
Renumbering Statements 65
Displaying Statements 66
Modifications in the Command Mode 66
Renaming Programs and Data Files 66
File Name Warning 68
Deleting a Program or Data File 68
Displaying Names of Permanent Programs and Data Files 69
Messages 70
System Cues 70
Prompting Messages 71
Informational Messages 71
Broadcast Messages 71
ITF Error Messages 72
Using the Test Mode to Debug Your Programs 72
Initiating the Test Mode 72
Terminating the Test Mode 73
Test Mode Subcommands 73
Starting and Resuming Executions—GO Subcommand 73
Interrupting Execution—AT and OFF Subcommands 74
Attention Interruptions 74
Setting Breakpoints 74
Monitoring Program Execution—TRACE and NOTRACE Subcommands 75
Listing Values—LIST Subcommand 76
Changing Values of Arithmetic Variables—Assignment Statement 77

Part II. The BASIC Language

BASIC Program Structure 81
Statement Numbers 81
BASIC Statements 81
Statement Lines 81
BASIC Programs 81

Elements of BASIC Statements 83
BASIC Character Set 83
BASIC Short Form (External Representation) 83
BASIC Long Form (External Representation) 84
Identifiers 84
Numeric Constants 84
Internal Constants 85
Character Constants 85
Variables 85
Simple Variables 85
Array Variables 85
Array Declarations 86
Functions 86
Expressions and Operators 86
Character Expressions 86
Arithmetic Expressions and Operators 86
Unary Operators 87
Priority of Arithmetic Operators 87
Array Expressions 88
Relational Expressions 88

Program Statements 89
CLOSE Statement 89
DATA Statement 89
DEF Statement 90
DIM Statement 90
END Statement 91
FOR Statement 91
GET Statement 92
GOSUB Statement 93
GOTO Statement 94
IF Statement 94
Image Statement 95
INPUT Statement 96
LET Statement 97
NEXT Statement 98
PAUSE Statement 99
PRINT Statement 99
PRINT USING Statement 102
PUT Statement 104
READ Statement 104
REM Statement 105
RESET Statement 105
RESTORE Statement 105
RETURN Statement 106
STOP Statement 106

Array Operations (MAT Statements) 107
MAT Assignment (Simple) 107
MAT Assignment (Addition and Subtraction) 107
MAT Assignment (CON Function) 108
MAT Assignment (IDN Function) 108
MAT Assignment (Inversion) 109
MAT Assignment (Multiplication) 110
MAT Assignment (Scalar Multiplication) 110
MAT Assignment (Transpose) 111
MAT Assignment (ZER Function) 111
MAT GET Statement 111
MAT INPUT Statement 112
MAT PRINT Statement 113
MAT PRINT USING Statement 115
MAT PUT Statement 116
MAT READ Statement 117

Intrinsic Functions 119

Part III. Command Language

Command Language for TSO ITF:BASIC 123
General Rules of Usage 123
Syntax of a Command 123
Positional Operands 123
Keyword Operands 123
Delimiters 123
Subcommands 123
How to Enter a Command or Subcommand 124
Continuations 124
AT Subcommand 125
BASIC Command 125
CHANGE Subcommand 126
CONVERT Command 127
DELETE Command (Also a Subcommand of EDIT) 129
EDIT Command 130
END Subcommand 130
GO Subcommand 130
HELP Command and Subcommand 130
INPUT Subcommand 131
LIST Subcommand 132
LISTCAT Command 132
LOGOFF Command 132
LOGON Command 133
NOTRACE Subcommand 133

OFF Subcommand 133
RENAME Command 134
RENUM Subcommand 134
RUN Command (Also a Subcommand of EDIT) 135
SAVE Subcommand 136
SCAN Subcommand 136
SEND Command 136
TRACE Subcommand 137

Appendixes

Appendix A. Syntax Conventions 141

Appendix B. Collating Sequence of the ITF:BASIC Character Set 143

Appendix C. Attention Interruption Summary 145

Appendix D. File Usage Considerations 147
File Maintenance 147
Using Files in *userid*.DATA 147

Appendix E. Differences Between OS ITF and TSO ITF 149
Terminological Differences 149
Visual Differences 149
Functional Differences 150

Glossary 153

Error Messages 161

Illustrations

Figures

Figure 1. Creation of an ITF:BASIC Program 21
Figure 2. Example of Program Creation (Including a Syntax Error) 22
Figure 3. Example of Executing a Program Created in an Earlier Session 23
Figure 4. Fixed-decimal Format (F-format) and Exponential Format (E-format) 45
Figure 5. Approximation of an Infinite Sum 46
Figure 6. The Output Data File 51
Figure 7. The Input Loop Used To Put Values into a File 52
Figure 8. Searching a File for a Single Value 54
Figure 9. Processing All Values in a File 55
Figure 10. Subroutine Example 58
Figure 11. Two Ways of Adding Statements to the End of a Program 63
Figure 12. Example of Modifying a Program As It Is Being Created 67

Tables

Table 1. Matrix Assignment Example 44
Table 2. BASIC Special Characters and Their Representation on Some TSO Terminals 83
Table 3. Carriage Positions in a PRINT Statement 101
Table 4. Arithmetic Expression Conversions in a PRINT USING Statement 103
Table 5. BASIC Intrinsic Functions 119
Table 6. Commands and Their Subcommands, ITF Test Mode Excluded 124
Table 7. ITF Test Mode Subcommands 125
Table 8. Attention Interruption Summary 145

Introduction

Time-sharing systems allow many users simultaneous access to the resources of a central computer. Users have their own typewriter-like terminals with which they communicate with the system. They type their instructions and the computer responds, all through the same medium. A type of conversation develops and the end result of that conversation is the accomplishment of work. One user's work does not interfere with another's, even though they are working at the same time with the same computer.

Many time-sharing systems are limited in scope and are usually aimed at one field of application. The IBM System/360 Operating System Time Sharing Option (TSO) is a time-sharing system that offers an unusually large range of computing capabilities to its users, namely, all of the numerous facilities of the IBM System/360 Operating System and several IBM Program Products, each with a particular application in mind.

ITF:BASIC is one of these program products. Its purpose is to meet your problem-solving needs quickly and efficiently without requiring you to learn complex computer techniques. Both TSO and ITF:BASIC have been designed for easy use and they mesh well. As far as you're concerned, TSO and ITF:BASIC are one and the same and this publication makes very little distinction between the two. TSO facilities are discussed only insofar as they apply to your ITF:BASIC needs.

If your work takes you beyond ITF:BASIC, there are other TSO publications that provide you with information about the full scope of TSO capabilities. A list of these publications is given in the preface.

Language

You communicate with TSO ITF:BASIC in two languages: a command language and a programming language. Both are English-like and easy to use. The command language is a set of commands and subcommands through which you control the system and direct its acceptance and execution of your work. RUN, DELETE, RENAME, and EDIT are examples of commands. A subset of the TSO full command language is discussed throughout Part I of this book. The exact formats of these commands and subcommands and rules for their use are given in Part III.

The programming language is called BASIC. It is used to write programs. BASIC statements always begin with a statement number (up to five digits) and are often mathematical in form, such as:

```
60 LET A = B + C
```

In BASIC, statement numbers determine the order in which statements are retained by the system. This order is always sequential.

The BASIC language is completely described in Part II of this book; examples and explanations of its use appear throughout Part I.

Part I. Using TSO ITF:BASIC

What Is a Program?

A program is a logical plan for obtaining a desired result for a particular problem —in physics, mathematics, finance, statistics, or any field. For example, if you wanted to find the average of four numbers—510, 371, 480, 791—you would first add the four figures together and then divide by four. Your logical plan would look something like this:

```
  510          538
  371       4/2152
  480         20
 +791          15
 2152          12
                32
                32
                 0
```

Here you have used a scratch-pad kind of mathematical notation that you, as the one performing the calculations, can understand. To have a computer perform the calculations, the program must be written in a notation (or language) that the computer understands and it must be sent to the computer by some convenient means. In our case, the language is called BASIC; the means of transmission is a terminal, which for all practical purposes is a typewriter connected by telephone lines to an IBM computer.

Written in BASIC, the same problem would look like this:

```
.
.
.
00010 let x = 510+371+480+791
00020 let y = x/4
00030 print y
00040 end
.
.
.
run
 538
.
.
.
```

For the present, do not be concerned with the information not shown (indicated by three vertical dots); you will learn about these things later. The program itself, the instruction to "run" it, and the results are all we need to know about now. The four BASIC statements (the actual program entered by the user) have statement numbers, which you will remember are a requirement. After we have typed the program, we instruct the computer to "run" (or execute) it. The RUN subcommand causes the computer to evaluate and perform each instruction in numerical order.

Now on to our program. The first statement, statement 10, is an assignment statement, that is, an instruction that causes the computer to assign the value on the right side of the equal sign to the variable on the left. The computer evaluates the expression (510+371+480+791) and assigns the sum (2152) to x. Statement

20 is another assignment statement; it tells the computer to divide the value of X by 4 (the slash symbol means division) and assign the result (538) to Y. Y now represents the average of the four values, but at this point, the average Y is known only to the computer. We will not know the results until the computer responds to the PRINT statement (statement 30) by printing the value of Y (538) at the terminal.

So far, we have not mentioned the END statement, statement 40. An END statement indicates the logical end of the program. Any statements numerically following an END statement are ignored in execution. END is optional, and if omitted it is assumed to follow the highest-numbered statement in the program.

Of course, there is much more to know about writing programs in BASIC—these things will be discussed as you proceed through the book. The first thing to know is how to get started at your terminal.

Getting Started

This section presents the information that you as a *terminal user* require to establish communication with the system.

Terminal Operating Procedures

ITF can be used with any of the terminals supported by TSO. All TSO terminals have a typewriter-like keyboard for entering information and either a typewriter-like printer or a display screen for recording your entries and system responses. The features of each keyboard vary from terminal to terminal; for example, one terminal may not have a backspace key, while another may not allow you to enter lower-case letters. The features of each terminal as they apply to TSO are described in the *TSO Terminal* publication mentioned in the preface.

Certain conventions apply to the use of all TSO terminals:

1. Any lower-case letters that you enter are received by the system as upper-case letters. For example, if you enter

   ```
   rem this is a comment
   ```

   the system receives it as

   ```
   REM THIS IS A COMMENT
   ```

   The only exceptions are certain text-handling applications, which are not within the scope of this book. Text-handling is described in the *TSO Terminal User's Guide* listed in the preface.
2. All messages or other responses sent to you by the system appear in upper-case letters. Again, the only exception is the output from special text-handling applications.

The examples in this publication appear as they would look on an IBM 2741 Communications Terminal having a PTTC/EBCD keyboard. Lower-case letters are used to illustrate your entries and upper-case letters are used for system responses. This convention provides a visual aid in differentiating what you type from what the system types.

Starting and Ending a Session

Before you can begin to do any work at the terminal, you must first establish a connection with the computer. Instructions for doing so will be provided by your installation. In many cases, you will find an instruction sheet attached to your terminal. If there is no instruction sheet, or if your installation has not provided the information in some other way, consult the *TSO Terminals* publication listed in the preface.

Logging On

After you have established a connection, you must identify yourself to the system by entering the LOGON command. In this command you must supply the *user identification code* assigned to you by your installation. Other information may be required, depending on the conventions established for ITF users at your installation. This optional information includes:

- *password:* further identification for additional security protection.

- *account number:* the account to which your work is charged.
- *log-on procedure name:* the name of an installation-written procedure that defines your scope of work (i.e., an ITF:BASIC user). This book assumes that ITF: BASIC users will enter the log-on procedure name PROC(ITFB).[1] If your installation has informed you otherwise, ignore references to this name.

If your user identification code were SMITH4, you would log on as follows:

```
logon smith4 proc(itfb)
```

One or more blanks must separate the three items from each other and the command is sent to the system by pressing the appropriate key (e.g., the RETURN key on IBM 2741 terminals) after the last item. Note that, for the sake of brevity, the notation CR or Ⓒ will be used henceforth in this book to represent this action on your part.

If you have typed the command correctly, the system will acknowledge you as an authorized user and you will be ready to begin your work. If you have made an error, TSO will require you to re-enter the command in part or entirely. A typical log-on sequence looks like this:

```
logon jv4 proc(itfb)
IKJ56455I JV4 LOGON IN PROGRESS AT 10:44 ON MAY 3, 1971
{Other TSO informational messages may appear here}
READY
```

After giving its acknowledgment, TSO types the "system cue" READY, positions the printing element at the beginning of the next line, and waits for you to type your next entry. At this point the system is said to be in the *command mode.* (This mode is always indicated by the system cue READY.) Only commands can be entered in this mode; no BASIC statements are allowed. Some commands will switch the system into other modes. The primary emphasis in this publication will be on one such command, EDIT, which initiates the edit mode. Most of the work that you, as an ITF:BASIC user, will be doing will be in the edit mode.

Logging Off

To end your session at the terminal, use the LOGOFF command. This command causes the system to:

1. display your user identification code,
2. display the current date and time of day, and
3. terminate your session.

The LOGOFF command must be entered in the command mode (i.e., in response to the system cue READY.) If you are currently in the edit mode and you wish to log off, you must first switch back to the command mode (you'll see how to do this later in this chapter under the heading "Terminating the Edit Mode").

The format of the LOGOFF command is simply:

```
logoff
```

The time you spend at the terminal from log on to log off is called a *session.*

The Attention Interruption

The attention interruption allows you to interrupt the current action of the system so you can enter a new command, subcommand, BASIC statement, etc. This ability to interrupt the system prevents you from being "locked out" by the system while a long-running program executes or while voluminous data is being displayed at your terminal. You can use the attention interruption for access to the system at any time.

[1] Details about log-on procedures for ITF are given in the publication *System/360 OS (TSO) ITF, Installation Reference Material,* Order Number SC28-6841.

In general, when you enter an attention interruption, the system cancels (or at least, interrupts) what it was doing and sends you a system cue. For example, if the system has been processing a command, the system cue it prints is READY, indicating that it is in the command mode and ready to accept another command.

There are two ways to cause an attention interruption:

1. Press the *attention key* at your terminal. This key is one of the following:
   a. ATTN, on an IBM 2741 terminal
   b. LINE RESET, on an IBM 1052 Printer-Keyboard
   c. BREAK, on a Teletype[1] terminal

   If the attention key is also the line-deletion character (described later in this chapter), you may have to press it twice to cause an attention interruption, depending on whether or not the first was interpreted as a line deletion.
2. If your terminal does not have an attention key (e.g., the IBM 2260 Display Station), use a simulated attention key, as instructed by your installation. If your installation has not provided this information, consult the *TSO Terminals* publication listed in the preface.

The attention interruption has many applications and it is discussed frequently in Part I. A summary is given in Appendix C.

Keyboard Entry Procedures

Normal communication between a terminal and the central computer is carried on by means of entries from the keyboard.

Your Entries

In general, an entry is a command, subcommand, or BASIC statement that you type at your terminal. You can never type more than one entry per line (e.g., you can't type two BASIC statements on the same line), but, in some cases, you can type multiple-line entries (e.g., you can continue an EDIT command over two, three, or more lines). BASIC statements must always be one-line entries.[2] Only certain commands and subcommands can be multiple-line entries. The multiple-line technique, and those commands and subcommands for which it can be used, is described in Part III. As an ITF user, you will rarely need more than one line for your commands and subcommands.

The character position at which you begin typing your entries depends on the command or subcommand currently in effect. In some cases, you can begin typing at the left-hand margin; in other cases, you can begin typing seven or more positions in from this margin. Where and when you can type should become quite obvious as you proceed through this book. You should note, however, that any references to character positions in this book are always relative to your left-hand margin setting. Thus, if your margin is set at position 10, for example, a reference to the ninth character position actually means position 18 on your terminal.

Correcting Typing Errors

You can correct typing errors in a line either before or after you have sent the line to the system, i.e., before or after you have ended the line by pressing the appropriate key. (On IBM 2741 terminals, for example, you press the RETURN key

---

[1] Trademark of Teletype Corp., Skokie, Illinois.

[2] The physical characteristics of some terminals, notably Teletypes, permit line continuations, regardless of the kind of entry being typed. Your *TSO Terminals* manual describes these characteristics. If you make use of this physical feature, please note that the maximum number of characters in a BASIC statement is 120. If you type more than 120 characters in such an entry, the entry will be rejected by ITF.

to end the line.) If you notice the error before you have completed the line, it is generally easier to correct it at that time. This is what we will describe here. Methods for correcting errors in completed lines are described in the chapter "Errors and Corrections."

There are two ways you can correct a line as you are typing it. You can request that the character you just typed be deleted, or you can request that all preceding characters in a line be deleted. These requests are made by using the *character-deletion character* and the *line-deletion character* selected for you by your installation.[1] For example, if the characters have been defined as ¢ for character deletion and % for line deletion, this line

```
170 let a,b,c = %170 let s,y+z =¢¢¢¢,z = 10
```

line deletion, deletes everything typed to this point (arrow at %)

character deletion, deletes "+z =" (arrow at ¢¢¢¢)

is received by the system as

```
170 let s,y,z = 10
```

This example illustrates how the character-deletion character can be used repetitively to delete more than one of the preceding characters. In this case, the characters "+z =" were in reverse order and replaced by ",z =".

Note that the blank space produced when you hit the space bar is always considered a character and should always be counted when using the character-deletion function.

If you are using an IBM 2741 Communications Terminal, then your installation will probably assign the character-deletion and line-deletion functions to the BACKSPACE and ATTN (attention) keys, respectively. If this is the case, you should be very careful in your use of the ATTN key. If you want to delete the line you're typing, press ATTN *once and only once.* The line will be deleted and the system will space to a new line for you to restart from. If you press ATTN more than once for a delete operation, the system will interpret your action in a way that you may not have intended (see Appendix C).

Note that BACKSPACE deletes all characters backspaced over, including the one backspaced to. Thus, for example, to change DECETE to DELETE after having typed the final E, you would backspace four times to the C and type LETE.

Character Set

The characters with which you compose your entries depend, of course, on the type of keyboard you have. All keyboards have the following:

1. *Alphabetic characters:* 29 alphabetic characters—26 letters (A,B,...,Z) and three *alphabetic extenders* ($,#,@).
2. *Digits:* ten digits: 0,1,...,9.
3. *Special characters:* all other characters on the keyboard.

Digits and alphabetic characters are often referred to as *alphameric* characters. Special characters vary from terminal to terminal and not all special characters have meaning to ITF:BASIC. The special characters that ITF:BASIC recognizes are defined in Part II under the heading "Elements of BASIC Statements."

You may notice when looking at this list that some characters are not physically present on your keyboard. If this is the case, you can represent these characters by others that are on your keyboard. For example, correspondence keyboards (9812 feature) of 2741 terminals don't have the "less than" (<) and "greater than"

[1] You can use the PROFILE command to establish your own character-deletion and line-deletion characters. This command is described in the *TSO Terminal User's Guide,* (see the preface).

(>) characters, but you can use the left bracket ([) and right bracket (]), respectively, in their place. It all depends on how the characters on your keyboard are translated by the system. Consult your *TSO Terminals* book (see the preface), to see what characters you can use in place of those that are not physically present on your keyboard.

Don't worry about character sets for now. Your need for certain characters will become obvious as you progress through this book and become familiar with rules for constructing statements and commands.

Mode Usage

This section contains a general overview of the TSO and ITF modes required by the ITF:BASIC user. It is intended to assist you in getting started and to allow you to learn ITF:BASIC at the terminal. The remainder of Part I contains a more complete discussion of the modes (and the commands and subcommands used in each mode) in the context of their usefulness to the ITF:BASIC user. Part III contains a detailed description of each command and subcommand, their functions, formats, and rules for use. For the present, you need only enough information to begin—the concepts of mode usage will become more clear to you as you proceed through the book.

The Edit Mode

With TSO, programs are always created, modified, and saved in the edit mode. You initiate the edit mode by using the EDIT command in the command mode. The general format of this command is:

EDIT *name* BASIC [NEW|OLD] [SCAN|NOSCAN]

You are required to specify EDIT, the name of your program, and BASIC. In general, the name that you specify for your program can contain from one to eight alphameric characters, and the first character must be alphabetic.[1] Some valid names are:

X3	DIVIDEA
LOOKUP	PRC
A17Z3	MAY24JOE

If your program already exists (i.e., it's "old"), you can specify OLD, but you don't have to. OLD is assumed if neither OLD nor NEW is specified. You specify NEW only for programs that don't already exist, for programs that you are going to build from scratch. The SCAN and NOSCAN options involve the syntax checking of your program's statements and they determine the way you will be notified of syntax errors. If you want to be notified of syntax errors after every statement you type, then you must specify SCAN. If you don't want such interruptions while you are building your program, you may specify NOSCAN, but, again, you don't have to; NOSCAN is assumed if neither SCAN nor NOSCAN is specified. After you have finished building your program, you can ask for notification of all syntax errors in it by using the SCAN *subcommand* of the edit mode.

So much for the format of the command. Let's consider two types of edit mode initiation. In the first type, we have an old ITF:BASIC program named ABC and we wish to change some statements in it. We also want immediate notification of syntax errors. Our EDIT command would look like this

```
edit abc basic old scan
```

or like this

```
edit abc basic scan
```

[1] Program names can be qualified according to the TSO data set naming conventions (see the *TSO Terminal User's Guide* for details). If qualified names are used, the descriptive qualifier should always be BASIC for ITF:BASIC programs.

Remember that OLD is assumed if neither OLD nor NEW is specified. For the second type of edit mode initiation, we want to create an ITF:BASIC program and name it CGA, but we don't want immediate notification of syntax errors. Now our EDIT command would look like this

```
edit cga basic new noscan
```

or like this

```
edit cga basic new
```

If neither SCAN nor NOSCAN is specified in an EDIT command, NOSCAN is assumed. For the time being we'll deal with "new" programs. Later on, you'll see what can be done with "old" programs.

Creating a Program

As you just saw, you initiate the creation of an ITF:BASIC program by specifying either

EDIT *name* BASIC NEW SCAN   or   EDIT *name* BASIC NEW NOSCAN

After you've given an EDIT command for a "new" program, TSO facilitates the building of the program by automatically supplying statement numbers before each BASIC statement you type. This is known as the *input phase* of the edit mode. Statement numbers are typed by TSO in the first five positions of a line and you type your statements beginning in the seventh position. (The system automatically skips to the seventh position for you.) The first statement number supplied by TSO is 00010, the second is 00020, and so on. The increment used by TSO for numbering statements is 10, which allows you to make later insertions between existing statements.

The input phase is discontinued when either of the following occurs:

1. A syntax error is detected in a statement and an error message is given. This can happen only if you have specified SCAN in the EDIT command.
2. You explicitly terminate the input phase. You can do this by (a) giving a CR in response to a statement number (a "null line") or (b) giving an attention interruption. In the latter case, you may have to press the attention key twice if the attention key is also your line-deletion key.[1]

You can re-initiate the input phase by typing INPUT. This subcommand causes numbering to resume from the highest statement number contained in your program. If you want to type your own statement numbers, don't use the INPUT subcommand; do either of the things listed in item 2 above and simply precede each statement with a statement number. You can discontinue and restart the input phase as often as you like.

With this in mind, look at Figure 1 to see what program creation looks like in practice. The program created here is the same as the program to average four numbers that you saw earlier in the chapter "What is a Program?" The braces have been added to further illustrate the use of modes in program creation and execution.

After T4 has logged on, he enters the edit mode to create the program he calls AVG. In response to this command, the system types "ITF INITIALIZATION PROCEEDING" and then, because this is a "new" program, it types the word INPUT on the next line indicating that the system will type the statement numbers at the beginning of each line.

---

[1] If your line-deletion key is the attention key, line deletions will always be accompanied by an automatic CR given by the system. The system will not type anything at the beginning of the next line. If you're in the input phase, *don't* retype the statement number of the statement just deleted. Just retype the contents of the statement. Remember that line deletion deletes only what *you have* typed, not what the system has typed. Since you don't type statement numbers in the input phase, the statement number is still there.

This is true provided you typed at least one character on the line before you pressed the attention key. If you didn't type anything on the line, pressing the attention key would cause an attention interruption (which would terminate the input phase).

```
command  logon t4 proc(itfb)
mode     IKJ56455I T4 LOGON IN PROGRESS AT 11:06:51 ON MAY 1, 1971
         READY
         edit avg basic new scan
         ITF INITIALIZATION PROCEEDING
         INPUT
         00010 let x = 510+371+480+791
         00020 let y = x/4                  input phase of edit mode
         00030 print y
         00040 end
edit     00050 (CR)
mode     EDIT
         run
          538
         EDIT
         save
         SAVED
         end
command  READY
mode     logoff
         IKJ56479I T4 LOGGED OFF TSO AT 11:10:07 ON MAY 1, 1971
```

Figure 1. Creation of an ITF:BASIC Program

After all statements have been typed, T4 ends the input phase by giving a CR in response to statement number 50. The system then types the system cue EDIT and gives T4 a new line in which to type his next entry (the RUN subcommand). The RUN subcommand causes the system to execute the program and to display the value of Y (538) at the terminal. Once again, the system types the system cue EDIT and gives T4 a new line for his next entry (the SAVE subcommand).

The SAVE subcommand causes the system to permanently retain the program. In this case, because no name is specified with SAVE, the program is saved under the name specified in the EDIT command. Had a name been specified with SAVE, the program would have been saved with that name instead of the name in the EDIT command.

If T4 had terminated the edit mode without having saved his program, the system would have reminded him of his possible oversight and would have given him the opportunity to do so immediately. If he did not save it then, the program would be irretrievably lost. (For details, see "Saving Programs" later in this chapter.)

Figure 2 shows how this session would look it T4 had made an error in statement 20. Because the SCAN option has been specified in the EDIT command, the system checks each statement for syntactical correctness and informs the user of syntax errors as they are encountered (as in statement 20). If SCAN had not been specified, the error in statement 20 would have gone unnoted until T4 requested that the program be executed (by the RUN subcommand).

When the error is discovered at statement 20, the system terminates the input phase and types an error message indicating the nature of the error. The message contains a message number (615), the number of the statement in which the error appears (20), and a brief summary of the type of error made. (Explanations of all numbered ITF:BASIC messages are given at the end of this book.) Error messages that end with a plus sign have a longer more comprehensive description, also.

```
logon t4 proc(itfb)
IKJ56455I T4 LOGON IN PROGRESS AT 11:06:51 ON MAY 1, 1971
READY
edit avg basic new scan
ITF INITIALIZATION PROCEEDING
INPUT
00010 let x = 510+371+480+791
00020 let y + x/4
 615  00000020 MSNG = OR UNID STM+
EDIT
20 let y=x/4
input
INPUT
00030 print y
00040 end
00050 (CR)
EDIT
save
SAVED
end
READY
logoff
IKJ56470I  T4 LOGGED OFF TSO AT 11:12:10 ON MAY 1, 1971
```

Figure 2. Example of Program Creation (Including a Syntax Error)

Message 615 ends in a plus sign, so, had T4 not understood the brief summary, he could have typed a question mark on the next line available to him (under the word EDIT) and the system would have typed this more comprehensive description of the error. In this case, however, T4 knows what's wrong (he typed a plus sign instead of an equal sign); so he doesn't request the longer message. Instead he types the number 20 and follows it with the corrected statement. This causes the erroneous statement 20 to be replaced with the new statement 20. Now, not wishing to type the rest of the statement numbers himself, T4 gives the INPUT subcommand and the system resumes numbering from the last statement number in the program (i.e., 20 plus the system-generated 10).

T4 didn't have to correct the error when he did. He could have waited until he had typed all of his statements. It makes no difference to the system, but it's a good practice to correct errors as soon as you're notified of them. If you delay making corrections, you may easily forget to make them, especially when you're creating large programs. If you save your program without having corrected the errors in it, the system saves it just as you created it, errors and all. Later, when you try to execute that program, you will be notified of all the errors in it and the program will not be executed.

Syntax Checking in the Edit Mode

Syntax checking is the process by which the system determines whether or not the BASIC statements in your program are constructed properly. It involves the verification of punctuation, spelling of statement keywords, placement of statement keywords and operands, etc. It does not involve the meaning or logic of your program. Syntax is checked as you construct each statement; meaning and logic are checked after you have requested that your program be executed.

In the edit mode, notification of syntax errors is controlled by the SCAN/NOSCAN options of EDIT and by the SCAN subcommand. If you specify the SCAN option, you will be notified of syntax errors immediately after each statement you type. If you omit the SCAN option, or if you explicitly say NOSCAN, the syntax checking will be performed but you will not be notified of syntax errors until either:

1. You request that the program be executed, *or*

2. You explicitly request notification via the SCAN subcommand.

In the first case, all syntax errors will be listed immediately below your execution request (which could be in another mode). You should then correct all errors (first returning to the edit mode, if you have to) and then re-execute. Note that semantic and logic errors cannot be discovered until the program is free of syntax errors.

Using the SCAN Subcommand

The SCAN subcommand is used to obtain notification of syntax errors, either in existing statements or in future statements you will type in the edit mode. If you specify

```
scan
```

you will be immediately notified of all syntax errors found in existing statements. If you specify

```
scan on
```

you will be notified of syntax errors in subsequent statements you type, but not those found in existing statements. To discontinue these notifications, you can type

```
scan off
```

at any time.

You can selectively apply the SCAN subcommand to existing statements by specifying statement numbers in the subcommand. For example, assuming that the last statement number in your program is 380.

```
scan 140
```

causes the system to notify you of syntax errors in statement 140 and nothing more. Similarly,

```
scan 20 260
```

causes the system to notify you of all syntax errors existing in the range of statements defined by 20 and 260. Any syntax errors existing outside this range will not be noted at this time.

Executing a Program in the Edit Mode

You can execute the program specified in the EDIT command at any time in the edit mode (except in the input phase). The edit mode subcommand for doing so is RUN. Figure 3 shows how the program created in Figure 1 can be executed in a later session. Because we saved AVG when we created it (in Figure 1) it is known to the system and it is therefore "old."

```
logon t4 proc(itfb)
IKJ56455I  T4 LOGON IN PROGRESS AT 10:09:34 ON MAY 2, 1971
READY
edit avg basic old
ITF INITIALIZATION PROCEEDING
EDIT
run
 538
EDIT
end
READY
logoff
IKJ56470I T4 LOGGED OFF TSO AT 10:11:50 ON MAY 2, 1971
```

Figure 3. Example of Executing a Program Created in an Earlier Session

Test Execution of Your Programs

Through the edit mode, you can initiate the ITF test mode, where you can use special subcommands to find errors of logic in your edited program. The ITF test mode is initiated in the edit mode by using the TEST option of the RUN subcommand. In other words, just type RUN TEST and all of the testing and debugging subcommands of the ITF test mode will be available to you. The ITF test mode can be initiated from the command mode as well as from the edit mode. A complete description of the test mode is given in the chapter "Errors and Corrections."

Interrupting Execution

You can interrupt and, as a result, cancel the execution of your program in the edit mode at any time by entering an attention interruption. You cannot resume execution from the point of interruption (as you can in the ITF test mode), but you can, of course, re-execute the program by a subsequent RUN subcommand.

Press the attention key just once to interrupt execution. Two successive "attentions" will terminate the edit mode and return you to the command mode.

Modifying Programs in the Edit Mode

It is possible to modify programs while in the edit mode. Through the use of edit mode subcommands (i.e., LIST, DELETE, RENUM, etc.) and, in some instances, statement numbers, you can insert, replace, or delete statements; you can change portions of one or more statements; you can renumber all or part of the statements in your program; and you can display the contents of one or more statement lines. As you'll soon see, it is timesaving and very useful to be able to modify and update programs without entirely retyping them each time a change must be made. A complete discussion of program modification is given in the chapter "Errors and Corrections."

Saving Programs

The program you just created or the changes you made to a previously existing program are retained by the system only as long as you remain in the edit mode. That is, as soon as you return to the command mode, your newly created program (or your new set of changes) is discarded. If you want the system to make your program a permanent one, or if you want the system to incorporate your changes into the existing program, you must use the SAVE subcommand of the edit mode.

The format of the SAVE subcommand is:

SAVE [*name*]

If you don't specify *name,* the program is saved under the name you specified in the EDIT command. If the program is an existing one (i.e., it's "old"), the old program in permanent storage is replaced by the changed one. If your program is "new," then this newly-created program is made permanent.

If you do specify *name,* the program is saved under that name; the name specified in the EDIT command is ignored. If *name* is the name of one of your existing programs, that program is replaced by the one currently in the edit mode. Otherwise, no replacement occurs and the one currently in the edit mode is made permanent with the name used in the SAVE subcommand.

When you are updating an "old" program, you may want to retain the "old" program and save the updated one too. Whenever this is the case, you should specify SAVE with a name that is different from the name of the old program you wish to retain. Then you'll have two permanent versions of the program: the old version without the changes and the updated version.

The name that can be specified with SAVE is constructed according to the rules for naming programs; that is, it can contain from one to eight alphameric characters, the first of which must be alphabetic.

Terminating the Edit Mode

The edit mode can be terminated in either of two ways:

1. By the edit mode END subcommand.
2. By an attention interruption.

In the first case, you merely type END whenever you can type a subcommand. If

you have not saved your program, the system will give you the option of entering a SAVE subcommand at this point.

In the second case, give the attention interruption at the beginning of the line. In some cases, a second attention interruption may be required (i.e., if the first attention interruption was given during the input phase or during program execution). When you end the edit mode by an attention interruption, the system will not give you the option of saving your program, if you haven't already done so.

The Command Mode

By now, you should be familiar with some aspects of the command mode. For instance, earlier in this chapter, you saw that a terminal session is begun by typing LOGON and that it is terminated by typing LOGOFF. Both LOGON and LOGOFF are commands that are used in the command mode. The EDIT command, which initiates the edit mode, is also issued in the command mode. The discussion here will focus on other aspects of the command mode—on such things as requesting information from the system about the format or function of a command or subcommand, sending messages to other terminal users, executing a program in the command mode, making an existing OS ITF:BASIC program acceptable to TSO ITF:BASIC, etc.

Requesting Information on Commands and Subcommands

The HELP command can be used to request information about the function, syntax, or operands of any command or subcommand available in TSO. For instance, if in response to the system cue READY, you typed

```
help
```

you would receive an immediate list of *all* the TSO commands[1] (e.g., BASIC, DELETE, EDIT, etc.) and a brief explanation of their function. This list is quite long, so it would be more efficient to select the command you want information about and specify that command name after HELP. For example, the command

```
help edit
```

will cause the system to respond with the specified command name (EDIT), its function, its syntax, and a list of its operands. The *function* tells you what the command does, *syntax* describes the format of the command, and the *operand list* indicates which operands are optional and which are required when you use the specified command.

If all you require is information on the function of the EDIT command, simply request

```
help edit function
```

and the system will respond with the function only. Similarly, if you only want information about the syntax of EDIT, you could type

```
help edit syntax
```

and only the syntax would be displayed. And, of course, the command

```
help edit operands
```

would cause the system to list only the operands of the EDIT command.

You can specify syntax, function, and operands (in any combination or order) in a single HELP command. For example,

```
help basic operands syntax
```

would cause the syntax as well as the list of operands to be displayed for the BASIC command.

[1] This list will include *all* TSO commands (not just the subset included in this book). Unless you are actually using the full TSO command language, it is not recommended that you request this full list because it is quite extensive and not all of the information is pertinent to the ITF user.

You can also use HELP in any other mode to request information about any of its subcommands. For instance, if you want to know something about RENUM in the edit mode, you could, in response to the system cue EDIT, type the following

```
help renum
```

If you want to know only the function, syntax, or operands of the subcommand, you could enter one of the following:

```
help renum function
help renum syntax
help renum operands
```

There is one restriction on using the HELP command; you cannot use it before you use the LOGON command. As was pointed out in the section "Starting and Ending a Session" earlier in this chapter, LOGON must be the first command used in your session because it identifies you as an authorized user of the system.

Sending Messages to Other Terminal Users

It is possible, in the command mode, to send messages to any other terminal user or to a system operator. The command used to do so is the SEND command. SEND can be used at any time after you log on.

If you want to send a message to the console operator at the central computer location, you would type SEND and follow it by the text of your message (enclosed in *single* quotation marks). For example,

```
send 'effective 5/1—account #48 changed to #42'
```

The text of the message must be enclosed in single quotation marks, as shown, and the message length (including blanks) cannot exceed 115 characters.

To send a message to another user, you must know his user identification code. For example, the command

```
send 'terminal 5 not available til 13:00 2/1' user(djv,ram)
```

will send the message enclosed in quotation marks to the two users whose identifications are DJV and RAM.

Generally, when you send a message to another user, he will receive it immediately, provided that he is logged on. If he is not logged on, you are notified by the system and your message is deleted. By using the LOGON operand of the SEND command, you can request the system to save your message until the user you sent it to logs on.[1] For example, if you enter

```
send 'cost estimate is due 11/30' user(tad2) logon
```

TAD2 will receive your message when he logs on.

You can send a message to only one operator at a time. With the SEND command, you must identify an operator by a number. For example,

```
send 'contact smith on ext.7943' operator(3)
```

If there is only one operator at your installation, you can omit the number. For example,

```
send 'contact smith on ext.7043' operator
```

If there are several operators and you omit the number, your message is sent to the main operator.

[1] It is possible for a user to suppress the printing of messages at his terminal. This feature is described in the *TSO Terminal User's Guide* (see the preface).

Executing a Program in the Command Mode

You can execute a permanent program (any program that has been saved as a part of permanent storage) at any time in the command mode. There are two commands for doing so: RUN and BASIC.

Returning to our now very familiar program to average four numbers (AVG), which you'll remember, was saved in an earlier session, the two ways to execute it in the command mode would look as follows:

USING RUN COMMAND	USING BASIC COMMAND
READY	READY
run avg basic	basic avg
ITF INITIALIZATION PROCEEDING	ITF INITIALIZATION PROCEEDING
538	538
READY	READY

In both cases, the program name must be specified immediately after the command name. The RUN command must further include the word BASIC to indicate that the program is written in the BASIC language. The BASIC command makes this known immediately, as you can see. In fact, the BASIC command is really a shorter way to specify program execution in the command mode.

Test Execution of Permanent Programs

By specifying the TEST option of the RUN or BASIC command, you can initiate the ITF test mode, where you can use special subcommands to find errors in logic in your permanent program. In other words, to test the program named AVG, just specify either of the following

```
run avg basic test        or        basic avg test
```

and all of the testing and debugging subcommands of the ITF test mode will be available to you. A complete description of the test mode is given in the chapter "Errors and Corrections."

Interrupting Execution in the Command Mode

As in the edit mode, you can enter an attention interruption and cancel the execution of your program in the command mode. You cannot resume execution from the point of interruption (as you can in the ITF test mode), but you can, of course, re-execute the program by a subsequent RUN or BASIC command.

Other Uses of the Command Mode

While you are in the command mode, you can change the name of any of your permanent programs or data files. You can also remove any program or data file from permanent storage, or list the names of all the programs and data files in permanent storage. These functions are performed through correspondingly named commands—RENAME, DELETE, and LISTCAT, respectively. A thorough discussion of these commands is contained in the chapter "Errors and Corrections" under the heading "Modifications in the Command Mode."

The command mode has one other command, CONVERT, which is designed as an aid for users who are moving from OS ITF to TSO ITF. By using CONVERT, you can make your OS ITF:BASIC programs acceptable for use in the TSO environment. This command is described thoroughly in Part III of this book.

The Test Mode [1]

You can use the test mode provided by ITF to help find any errors that could not be detected by the syntax scan. The test mode can be entered both from the edit mode and the command mode. In the edit mode, the subcommand that initiates the test mode is RUN TEST. In the command mode, there are two ways to initiate the test mode: (1) by the RUN command with the TEST option, or (2) by the BASIC

[1] This mode is not to be confused with TSO's test mode. It is entirely different and is only available with ITF.

command with the TEST option. Once in the test mode, you can use several subcommands to control the execution of your program—tracing changes to variables and program flow, stopping at various points in the execution to examine the current values of variables, etc. The test mode and the subcommands that can be used in the testing environment are discussed more fully in the chapter "Errors and Corrections."

Now that you have a general understanding of how to begin and how to use the modes in TSO, it is time to get acquainted with the ITF:BASIC language and to actually begin to use the system.

Writing a Program

BASIC statements are similar to handwritten mathematical notation except that in BASIC, statements must be typed on a single line and, of course, each statement must have a statement number. The expression

$$f = G \frac{m_1 m_2}{d^2}$$

for example, could be typed like this in BASIC:

```
100 LET F = G * (M1 * M2) / D**2
```

We have already seen that the slash (/) indicates division. Here we see that the asterisk (*) denotes multiplication and the double asterisk (**) denotes exponentiation. These symbols are called *operators* in BASIC—they cause some action to be performed. You'll learn more about operators later in this chapter.

Building a BASIC Statement

BASIC statements are made up of operators and *identifiers*. Identifiers are values which are either *constant* or *variable*. In our example above, the numeral 2 is a constant—its value never changes.

Constants

BASIC has three types of constants:

1. *Numeric*—decimal numbers such as 2, 10, 17.3, 4.19E-3 (the latter is expressed in the E, or "exponential," format which is discussed later in this chapter under "Large and Small Numbers").
2. *Character*—any character or group of characters on the keyboard enclosed in single or double quotes, such as 'ANSWER', "NAME CITY", etc.
3. *Internal (or system-supplied)*—BASIC provides the frequently needed values of π, $\sqrt{2}$, and e as shown below:

Meaning	BASIC Name	Value (in short form)	Value (in long form)
e	&E	2.718282	2.71828182845904
π	&PI	3.141593	3.14159265358979
$\sqrt{2}$	&SQR2	1.414214	1.41421356237309

In your programs, constants can be used for such things as multipliers, divisors, increments, and headings for your printouts. We'll see how to use character and internal constants later. Right now, let's concentrate on numeric constants. In BASIC, numeric constants can be signed or unsigned and they can have up to 7 significant digits (long-form BASIC constants can have up to 15 significant digits). If you supply a constant larger than 7 (or in long form, 15) digits, it will be truncated by the computer. The following numbers are all valid as constants:

Short Form	Long Form
376	123456700000000
+10	.000000000000001
−1234567	−765.4321

Variables

Identifiers can also be variable in nature. In our example

```
100 LET F = G * (M1 * M2) / D**2
```

the identifiers F, G, M1, M2, and D represent numeric values which can vary each time they are used. For this reason they are called *arithmetic variables.*

In BASIC, arithmetic variables are indicated by a single alphabetic character or by an alphabetic character followed by a digit. Examples of valid arithmetic variable names are: Z, D3, A1, @4, #7, $. An important thing to remember is that arithmetic variable names can never begin with a digit and the second character (if there is one) must always be a digit.

Later you will see that you can use a variable to represent "character data" whose value may change (i.e., an inventory item, an account name, etc.). These are called *character variables.* In BASIC, each character variable name must be an alphabetic character followed by the character "$." This naming scheme makes the character variable easily recognizable to the computer. Examples of valid character variable names are: C$, A$, $$, #$.

Assigning Values to Variables

Now that we understand that the primary elements of BASIC statements are operators (+, −, *, etc.) and identifiers (constants and variables), let's see how to use them in a program.

Initially, the system sets all arithmetic variables to zero and all character variables to 18 blanks. Now, we need to know how to assign other values to variables. We have already seen the easiest way to do this—by using the LET statement.

```
10  let x = 40
20  let y = 172
30  let t = x + y
40  z = t + 120
```

Statements 10 and 20 are rather straightforward. The computer takes the numeric constant on the right side of the equal sign and assigns that value to the arithmetic variable on the left. In statement 30, we've used variables (X and Y) on the right side of the equal sign and we've assigned the sum of their values to T. Because we've already given X the value of 40 and Y the value of 172 (in statements 10 and 20), the computer adds these known values together and assigns the result (212) to T. Statement 40 shows that the word LET is not required in an assignment statement. The computer treats this statement the same as statements 10 through 30; it adds 120 to the new value of T and assigns that value to Z.

The paragraph above illustrates how to assign values to arithmetic variables. Similarly, character variables can be given values through LET statements. For example:

```
10  let a$ = 'city'
20  let b$ = 'state'
30  $$ = 'zip code'
```

The computer takes the character constant (enclosed in quotes) on the right side of the equal sign, and assigns that character value to the variable on the left.

The LET statement can also be used to assign a value to more than one variable. This is called the multiple LET statement. Examples are:

```
100  let x,y,z = 12
200  let a1, m = 100
```

Statement 100 tells the computer to assign the value of 12 to X, to Y, and to Z. Statement 200 instructs it to give A1 and M each the value of 100.

The simple program we wrote to average four numbers used the LET statement to assign values to variables. It looked like this:

```
10  let x = 510 + 371 + 480 + 791
20  let y = x/4
30  print y
40  end
```

Perhaps you can see that this program can be used to average only these four numbers (510, 371, 480, 791). By using another method of assigning values to variables, the DATA statement, we can write this program so that it will average any four numbers supplied in the DATA statement.

```
10  read a,b,c,d
20  data 510,371,480,791
30  let x =  a+b+c+d
40  let y = x/4
50  print y
60  end
```

When the computer executes the READ statement, it "reads" the values given in the DATA statement and assigns them (in order) to the variables A, B, C, and D. When all the data is consolidated in a single statement, you can change the data by changing only the DATA statement. The advantages of this method of assigning values (known as "program input") are more evident in the sample program shown below.

USING LET STATEMENTS

```
10  let p = 1000.00
20  let r = 5
30  let t = 10
40  let a = p*(1+r/100)**t
50  print a
60  end
```

USING DATA STATEMENT

```
10  read p,r,t
20  data 1000.00, 5, 10
30  let a = p*(1+r/100)**t
40  print a
50  end
```

Using LET statements, we would have to replace three statements (10, 20, and 30) to change the data. If we used the program with the DATA statement, we could change the data by replacing one statement (20). You will learn how to replace program statements later in the chapter "Errors and Corrections" under the heading "Modifications in the Edit Mode."

Character variables also can be given values using READ and DATA statements. In fact, arithmetic variables and character variables can be interspersed in the same READ and DATA statements. Of course, the DATA statement must supply values for these variables in the order in which they appear in the READ statement. For example:

```
10  read a,b$,x,#$
20  data 2507,"john doe",33,"new york"
```

The values would be assigned as follows:

A = 2507
B$ = JOHN DOE
X = 33
#$ = NEW YORK

You can actually have more than one DATA statement in a program; the effect is cumulative. For example:

```
10  read a,b,c,d,e,f,g
20  data 10012, -73621, 4308.973, 7.2, 15.0, -3.7
30  data 10
```

The values would be assigned like this:

A = 10012
B = −73621
C = 4308.973
D = 7.2
E = 15.0
F = −3.7
G = 10

You might want to do this when working with many numbers, or whenever some numbers are more subject to change than others. Placing the "changing value" in a separate statement permits you to change it without retyping every other number or value. This is timesaving and it might prevent you from making an error when typing a long list of numbers. However, take care to see that the number of variables given in the READ statement is not greater than the total number of values supplied by your DATA statements and that they are in a corresponding order. Should you have even one variable too many in the READ statement, your program will be terminated. However, if you supply more values in DATA statements than you have matching variables in the READ statement, your program will run, but those values without a matching variable will be ignored. A subsequent READ statement would continue where the previous READ left off. If, however, you want to ignore the remaining values in the DATA statement and start again with the first value, you could use the RESTORE statement which would cause a subsequent READ statement to start at the beginning of the list of values. For example:

```
10  read a,b,c,d
20  data 10012,-73621,4308.973,7.2,15.0,-3.7,10
30  restore
40  read e,f,g,h
50  read x,y,z
```

The values would be assigned like this:

A = 10012
B = −73621
C = 4308.973
D = 7.2
E = 10012
F = −73621
G = 4308.973
H = 7.2
X = 15.0
Y = −3.7
Z = 10

There are similarities in assigning values by the LET statement and by the DATA statement. Both methods actually supply the numbers to be averaged within the program itself. In each case, we would have to retype one or more statements to supply new data for our program. You can see that it would be advantageous in many instances to have more flexibility.

Varying the Input

A third means of assigning values to variables, the INPUT statement, provides this flexibility. Using the INPUT statement, our program to average four numbers would now look like this:

```
READY
edit avg basic new scan
ITF INITIALIZATION PROCEEDING
INPUT
00010 input a,b,c,d
00020 let x = a+b+c+d
00030 let y = x/4
00040 print y
00050 end
00060 (CR)
run
?     30,50,75,29
 46
EDIT
```

After we've typed our program, we signal the computer to run it (by the RUN subcommand). The computer will then move to the first statement in the program and attempt to execute it. Since our first statement (10) is an INPUT statement, however, the computer knows that four values are to be supplied from the terminal. It stops execution and prints a question mark (?) at the terminal. You respond by typing any four signed or unsigned numeric constants (in this case 30, 50, 75, 29) on the same line. The computer takes these values, assigns them to A, B, C, and D in the order they are typed, performs the remaining operations on them, and prints the answer (46). Note that if you do not supply a value for a variable listed in the INPUT statement, or if you enter more values than there are variables in the INPUT statement, the system will issue a message and allow you to retype the line.

The INPUT statement is very useful for handling varying input, and, as you'll see, it can be very handy in writing programs which can be used many times. Supplying data by means of the INPUT statement is known as "terminal-oriented input."

Expressions and Calculations

That part of the assignment statement that is to the right of the equal sign is called an *expression*. It is the expression that specifies the value to be assigned to the variable on the left of the equal sign. An expression can be very simple, involving no calculations, or it can be quite complicated, involving many variables and arithmetic operations. All of the following assignment statements contain valid forms of arithmetic expressions (character expressions and relational expressions are discussed later in this chapter):

```
40  let b = 5
50  let a = b
60  let z = 23.71
70  let y = 2**6 + (z/13.2)*a - a**3
```

Arithmetic Expressions

An arithmetic expression is composed of an arithmetic variable, an internal constant, a numeric constant, a subscripted arithmetic array reference, a function reference (array and function references are discussed later in this chapter), or a series of the above separated by operators and parentheses. Some examples of arithmetic expressions are:

```
A1          X+Y+Z
-6.4        X3/(-6)
SIN(R)      -(X-X**2/2+X**(Y*Z))
```

The symbols (arithmetic operators) used in expressions to specify mathematical operations are defined as follows:

OPERATOR	MEANING
+	addition (also the unary plus sign to indicate a positive value)
−	subtraction (also the unary minus sign to indicate sign reversal, e.g., the expression −A means multiply the value of A by −1)
*	multiplication (A*B means A multiplied by B)
/	division (A/B means A divided by B)
↑ or **	exponentiation or raise to the power (A**B or A↑B means A raised to the power B, or A^B)

In general, expressions are evaluated as follows:

1. Exponentiation is performed first; thus, it is said to have the highest priority. If two or more of these operators appear in the same expression, they are evaluated in the order they appear (from *left* to *right*).
2. Unary operations have the second priority and are evaluated from *left* to *right.*
3. Multiplication and division have the third priority. These operations are evaluated from *left* to *right.*
4. Finally, addition and subtraction have the lowest priority; they, too, are evaluated from *left* to *right.*

For example, in the expression:

```
-A**2 + B/C * 2.5
```

the evaluation process follows this sequence:

1. A**2 is evaluated first.
2. The unary minus sign is applied to the result of A**2 (i.e., the sign of A**2 is reversed).
3. B is divided by C.
4. The result of B/C is multiplied by 2.5.
5. Finally, the result of item 4 is added to the result of item 2.

Unary operators may be used in only two situations in a BASIC program:

1. Following a left parenthesis and preceding an arithmetic expression, or
2. As the leftmost character in an entire expression which is not preceded by an operator.

Parentheses may be used in an expression to alter the order in which the expression is evaluated by the computer. Any part of an expression enclosed in parentheses is evaluated before any other part of the expression. For example, the expression

```
A-B/C
```

is always evaluated as follows: divide B by C and then subtract the result from A. Spacing is ignored, i.e., even if the expression were written:

```
A-B   /C
```

the division operation would be performed before the subtraction operation. However, by using parentheses, this order of evaluation can be altered. For example, the expression

```
(A-B)/C
```

is evaluated as follows: first subtract B from A and then divide the result by C. Note that A−(B/C) is the same as A−B/C.

Thus, the use of parentheses in expressions is quite similar to the use of parentheses in algebra; that is, parentheses group operations and indicate which operations should be performed first.

With this in mind, let us reconsider an expression we evaluated earlier. Substituting the values 4, 6 and 2 for A, B, and C, respectively, notice how parentheses affect the order of evaluation and change the result.

Expression	Evaluation and Result
−A**2+B/C*2.5	−4**2+6/2*2.5
	−16 +6/2*2.5
	−16 + 3 *2.5
	−16 + 7.5
	−8.5
(−A)**2+B/C*2.5	(−4)**2+6/2*2.5
	16 +6/2*2.5
	16 +3 *2.5
	16 + 7.5
	23.5
−A**(2+B/C)*2.5	−4**(2+6/2)*2.5
	−4**(2+3) *2.5
	−4**5 *2.5
	−1024 *2.5
	−2560
−A**((2+B)/C)*2.5	−4**((2+6)/2)*2.5
	−4** (8/2) *2.5
	−4**4 *2.5
	−256 *2.5
	−640

This illustrates only four of the possibilities in this one expression. As you can see, the use of parentheses can have drastic effects on the result. Thus, you must understand the rules completely and apply them carefully. The last example illustrates the *nesting* of parenthesized expressions. In such cases, the expression within the innermost set of parentheses is always evaluated first; the expression within the next innermost set is evaluated next, and so on until the outermost level is reached. Thus, in this example, 2+B is evaluated first and then the result of 2+B is divided by C.

As a further illustration, evaluation of (2**3)**2 or 2**3**2 gives 64, whereas, evaluation of 2**(3**2) gives 512.

In addition to the five basic arithmetic operations, many familiar mathematical functions such as sine (SIN), cosine (COS), square root (SQR), and natural logarithm (LOG) are available. A list of these functions (called "intrinsic functions" in BASIC) is given in Part II of this manual. Some examples of their use are shown below:

```
70 let v = cos(y)
80 let z = 1 + sqr(x**3)
90 let w = 1 - sqr(cos(a))
```

The quantity in parentheses immediately following the name of the function is an argument (e.g., X**3). An *argument* is merely an expression representing a value that the function is to act upon. The expression can be as simple or as complicated as any of the expressions we've encountered so far, and it is evaluated according to the same rules. Thus, in the second example, if the value of X is 4, then the value of X**3 is 64, and the value of SQR(X**3)—or the square root of 64—is 8. The last example shows nested function references, which are evaluated like nested expressions. Thus, the cosine of A is found first and the square root of that cosine value is found next.

Character Expressions

A character expression is composed of a character variable, a character constant, or a subscripted character array reference (arrays are discussed later in this chapter). Character expressions may be used in assignment statements (LET, READ and DATA, INPUT, and GET statements), in IF statements (to test relational

conditions—equal to, greater than, etc.), and in output statements (PRINT, PRINT USING, and PUT). A few examples of character expressions used in these statements are:

```
let a$ = 'abcdefg'
if d$ = c$ then 20
print z$,'total',#$(10)
```

In all operations with character expressions, except for output using the PRINT and PRINT USING statements, character constants containing more than 18 characters will be truncated on the right to 18 characters. Character constants containing less than 18 characters will be padded with blanks, on the right, to 18 characters. Character constants containing no characters (two adjacent quotation marks, known as a *null* character string) will be interpreted as 18 blank characters.

Relational Expressions

Relational expressions are used to test the relationship between two expressions. Two forms of relational expressions are allowed in BASIC, arithmetic and character. The general format of a relational expression is:

expression-1 relational-operator expression-2

The relational operators and their representation on some of the terminals supported by TSO are:

	OPERATOR		
DEFINITION	2741 (#9571) PTTC/EBCD	2741 (#9812) CORRESPONDENCE	TELETYPE MODELS 33/35
equal to	=	=	=
not equal to	<>	[]	<>
greater than	>	]	>
less than	<	[	<
greater than or equal to	>=	]=	>=
less than or equal to	<=	[=	<=

When a relational expression is encountered in a program, the computer evaluates *expression-1* and then *expression-2*. Their values are compared according to the definition of the relational operator used. The evaluation of the entire relational expression results in the expression being either satisfied (relation is true) or unsatisfied (relation is false).

Printing Results

Now that we've seen how values are assigned to variables (by the LET, READ and DATA, and INPUT statements) and how certain calculations can be performed with those values, let's explore the PRINT and PRINT USING statements and see how we get the results of the calculations out of the computer.

You've probably noticed that we've been using the PRINT statement in examples earlier in this book. For example:

```
edit avg basic new scan
ITF INITIALIZATION PROCEEDING
INPUT
00010 input a,b,c,d
00020 let x = a+b+c+d
00030 let y = x/4
00040 print y
00050 end
00060 (CR)
run
?     30,50,75,29
 46
```

As you can see, the computer prints the value of the variable Y (46) in response to the PRINT statement. The PRINT statement can also contain arithmetic and character expressions, character constants, and format control items (which are described later). If, for some reason, we wanted to add 10 to the average found in the example above, and to print that result following the phrase "average + 10 is:", we would type:

```
40 print 'average + 10 is:', y+10
```

The result would look like this:

```
AVERAGE + 10 IS:  56
```

It is often very helpful to use character constants in the PRINT statement in order to label output, particularly if the program is printing several values.

In general, each PRINT statement causes the computer to begin a new line. Therefore, a PRINT statement with nothing after it won't cause anything to be printed, but it will cause a carriage return. This is a useful technique for improving the appearance of your printed output.

Horizontally, the page is divided into *full print zones,* each zone having 18 print positions. Assuming that the left-hand margin has been set at position 0 on the IBM 2741 terminal, the zones would begin in positions 0, 18, 36, 54, 72, etc. A comma is used as a signal to the computer to move across the page to the next full print zone. For example, if we use the following statement:

```
70 print  a,b,c
```

the computer would start at the left edge of the page and print the value of the variable A. Then it would skip over to print position 18 and print the value of B. The value of C would be printed beginning in print position 36.

It is possible to increase the number of print zones on a line. A semicolon or a *null delimiter* (a blank or no separation at all between data items) indicates to the computer to use a *packed print zone* rather than a full print zone. A null delimiter may be used when one, and only one, of the data items is a character constant. The size of the packed zone for arithmetic data is determined by the length of the field to be printed, as follows:

Length of Print Field	Length of Packed Print Zone	Examples (*x* represents a blank)
2-4 characters	6 characters	*x*173*xx*
5-7 characters	9 characters	*x*173576*xx*
8-10 characters	12 characters	−45.63927*xxx*
11-13 characters	15 characters	*x*1.735790E−23*xx*
14-17 characters	18 characters	−892270409311563*xx*

If the item to be printed is a character variable or a subscripted character array reference, the size of the packed print zone is 18 characters minus any trailing blanks. If the data item is a character constant, the size of the packed print zone equals the length of the character string enclosed in quotes.

The PRINT USING statement is quite similar, but is much more useful for controlling the format of the answer to be printed. PRINT USING is used in conjunction with an Image statement to print values according to the format specified in the Image statement. The PRINT USING statement includes the values to be printed and the statement number of the Image statement to be used; the Image statement specifies the format of the print line. For example:

```
edit int basic new scan
ITF INITIALIZATION PROCEEDING
INPUT
00010 input p,i,n
00020 let a = p*(1+i/100)**n
00030 print using 40, n,a
00040 :in ## yrs amt = $####.##
00050 end
00060 (CR)
run
? 1000.00, 5, 10
IN 10 YRS AMT = $1628.88
```

Statement 30 directs the computer to print the values of N and A using statement 40 as the image. The colon beginning statement 40 identifies it as the Image statement. The alphabetic characters are printed as they appear in the statement, the value of N replaces the first set of #'s, and the value of A replaces the final set of symbols. Note that the decimal point in the value of A is aligned on the decimal point in the image specification.

Details about PRINT and PRINT USING statements are given in Part II of this publication.

Loops

Certain problems require that one specific operation or sequence of operations be performed repeatedly over a set of values. Such calculations are generally done most efficiently by a simple programming device known as a "loop." For example, consider the problem of printing an integer along with its square. Without using a loop, you could do this by writing the following statements:

```
10 print 1,1**2
20 print 2,2**2
30 print 3,3**2
40 print 4,4**2
```

and so on, ending with:

```
490 print 49,49**2
500 print 50,50**2
510 end
```

Obviously, this method is very time consuming and tedious. A loop provides a concise method. Consider this solution:

```
edit rtb basic new scan
ITF INITIALIZATION PROCEEDING
INPUT
00010 let x = 1
00020 print x,x**2
00030 let x = x + 1
00040 go to 20
00050 end
00060 (CR)
```

In RTB we have created a loop in statements 20 through 40 so that when the program is run the PRINT statement will be executed once each time the value of x increases by 1. The statement that makes the loop possible is the GOTO statement. It alters the normal sequence of execution by actually specifying the next statement to be executed. It does this by referring to the number of that statement.

There is one problem with the loop we have just shown: there is no provision for ending the loop. Consequently, not only will we get results for values from 1 to 50, but also for 51, 52, and so on, unless some action is taken to stop the execution once the requirements of the problem have been satisfied.

Either of two actions could be taken:

1. We could press the attention key, thereby causing the computer to cancel execution at the point the "attention" signal is recognized.
2. Or, better still (since the first action cancels our program and requires our physical intervention every time the program is executed), we could build into the loop a test for some condition, so that when the condition was met the loop would end automatically.

Taking the second action, we want the loop to end as soon as the value of X becomes greater than 50, or put another way, we want the loop to continue as long as X is less than or equal to 50. An IF statement says it quite concisely:

```
IF X <= 5 THEN 20    or    IF X <= 50 GOTO 20
```

In the IF statement, THEN and GOTO have the same meaning—they are interchangeable. This IF statement, when inserted in RTB, would provide the test needed to end the loop. RTB should now consist of this sequence:

```
10 let x = 1
20 print x,x**2
30 let x = x + 1
40 if x <= 50 then 20
50 end
```

As long as X satisfies the condition "X less than or equal to 50," execution will loop back to the PRINT statement. However, when X no longer satisfies the condition, then the loop will end automatically and the execution will "fall through" the IF statement to the statement on the next line, which in this case is an END statement signifying the end of the program.

The IF statement has many applications, some of which can be quite sophisticated, depending on the condition tested in the statement. For example, conditions such as the following can be tested:

```
IF A = 0 THEN 60
IF A <> 0 GOTO 40
IF B-X/Y < Z**2 THEN 80
```

The second condition literally means "A is less than or greater than zero" or, if you prefer, "A is not equal to zero." The last shows that expressions of various complexities are permitted on both sides of the "relational operator" (the symbol defining the condition). All of the relational operators and their meanings are listed in Part II of this book. Use of arithmetic expressions and arithmetic expressions with relational operators was discussed earlier in this chapter.

Looping by FOR and NEXT

A still more concise method of specifying a loop is by using the FOR and NEXT statements. For example, our program for finding and printing the square of the numbers from 1 through 50 could be further simplified to look like this:

```
10 for i = 1 to 50
20 print i,i**2
30 next i
40 end
```

The FOR statement identifies the beginning of the loop; the NEXT statement identifies the end of it. In between is the statement (or sequence of statements—we only need one for this example) that will be executed repeatedly until the specification in the FOR statement has been satisfied.

In our example, the FOR statement specifies that the statement in the loop (the PRINT statement) will be executed repeatedly for successive values of I from 1 through 50 (an increment of 1 is added to I for each execution of the PRINT statement). When the value of I exceeds 50, execution of the loop is ended, and control is passed to the next logically executable statement following the NEXT statement. In this case, the following statement is an END statement denoting the end of the program. However, other instructions could precede it, or the NEXT

could be the last statement in the program (in which case, the system would supply an END statement).

The specification "I = 1 TO 50" is called a *range specification* because it defines the range of values over which the loop will be executed. As we've seen, the range in our example is 1, 2, 3, ..., 50. The increment is always 1 unless it is explicitly stated to be otherwise; for example:

```
10 FOR I = 1 TO 50 STEP 2
```

This FOR statement explicitly states an increment (or step) of 2. Thus, the statement(s) in the loop will be executed once for every odd value of I from 1 to 50 (i.e., the range is 1, 3, 5, ..., 49). When the value of I exceeds 50 (that is, when it reaches 51), execution of the loop will end. The value of I is then adjusted and will be 49 when the next logically executable statement is executed. If you wanted to execute the loop once for every even value of I from 1 to 50 (i.e., 2, 4, 6, ..., 50), you would say the following:

```
10 FOR I = 2 TO 50 STEP 2
```

Again, when the value of I exceeds 50 (in this case, when it reaches 52), execution of the loop will end. The value of I is then adjusted and will be 50 when the next logically executable statement is executed.

As with expressions appearing in assignment statements and in the body of PRINT statements, the range specifications in FOR statements can be quite complicated. For example, the following FOR statements are permitted:

```
FOR I = A TO B
FOR J = 8*M+Y TO A**3
FOR K = SQR(B) - C TO 550 STEP A/B*2
```

Arrays

An array is a named list or table of data items, all of which are the same type. In BASIC, there are two kinds of arrays—*arithmetic* (which contain only arithmetic values) and *character* (which contain only character values 18 characters in length).

Arithmetic Arrays

An arithmetic array is named by a single alphabetic character. If your array contains ten or fewer data items, no special indication of its size (dimension) is necessary. By referring to an individual data item (member) of your array in any program statement, the computer will recognize that you are working with an array and will automatically allow space for as many as ten members. (This is known as *implicit* declaration.) An individual item (member) is referred to by giving its location in the array. For example, B(1) refers to the first member of the array named B; B(2) refers to the second member, B(3) refers to the third member, and so on. Each number giving the location of a particular member (i.e., 1, 2, 3, etc.) is called a *subscript*. If the following statement were typed:

```
40 LET B(9) = 44
```

only the ninth member of B would be assigned the value 44; all other members would remain unchanged.

If you are working with an array of more than ten members, you must *explicitly* state the array dimension (size) so the computer will allow enough space. For example, to make T an array of 12 arithmetic items, the following statement would have to be given:

```
10 DIM T(12)
```

The number appearing in parentheses is known as the bound of the dimension of the array. It specifies that the array T represents 12 different arithmetic items.

If your array has very few members (e.g., two or three), it is most efficient to use a DIM statement such as,

```
10 DIM A(2), B(3)
```

so that the computer won't automatically provide space for ten members, most of which will be unused.

Note the difference between a subscript and the array bound. A subscript is used to *refer* to a particular member of an array and it can be any valid arithmetic expression (i.e., numeric constant, function reference, etc.). The bound *defines* the number of members of an array; it can only appear in a DIM statement and it must be indicated by positive integers only. An array name cannot appear in a DIM statement if the array dimension has already been stated—either implicitly (through usage) or explicitly (by appearing in another DIM statement).

In BASIC, you can have arrays of one dimension (arithmetic or character), or arrays of two dimensions (arithmetic only). Assume that values have been assigned to T (a one-dimensional array) such that:

T(1) is 31　　T(7) is 79
T(2) is 43　　T(8) is 79
T(3) is 42　　T(9) is 69
T(4) is 57　　T(10) is 58
T(5) is 64　　T(11) is 44
T(6) is 73　　T(12) is 39

Let's say that each of these values represents the average temperature for one month of a particular year; T(1) represents January's average, T(2) represents February's, etc.

For various reasons, another programmer might want to consider the year as divided into four quarters of three months each; he could declare his array (call it M) as follows:

```
10 DIM M(4,3)
```

In this statement, the bounds specify that the array M is a two-dimensional array containing 12 members (the product of 4 and 3), just like the array T. The difference is that the members of M are distributed over two dimensions, whereas in T they are distributed over only one dimension. Conceptually, the two dimensions of M can be thought of as four lists of three items each (i.e., four quarters, three months to each quarter). Assuming that the same temperatures assigned to T are assigned to M, notice the difference in the way each item is referred to:

ARRAY T	ITEM	ARRAY M
T(1)	31	M(1,1)
T(2)	43	M(1,2)
T(3)	42	M(1,3)
T(4)	57	M(2,1)
T(5)	64	M(2,2)
T(6)	73	M(2,3)
T(7)	79	M(3,1)
T(8)	79	M(3,2)
T(9)	69	M(3,3)
T(10)	58	M(4,1)
T(11)	44	M(4,2)
T(12)	39	M(4,3)

Two subscripts are always used to refer to a particular member of M: e.g., M(3,1) refers to the temperature for July, the first month in the third quarter. The number of subscripts in a reference to an array member must always be the same as the number of bounds shown in the DIM statement.

You might also visualize a two-dimensional array such as M as a table of 4 rows and 3 columns, in this way:

M	(m,1)	(m,2)	(m,3)
(1,n)	31	43	42
(2,n)	57	64	73
(3,n)	79	79	69
(4,n)	58	44	39

You can use a two-dimensional array in a program without explicitly stating its bounds in a DIM statement. You would do this by using two subscripts to identify the location of a particular member in the array (for example, A(4,3) would refer to the member in the fourth row and third column of the array A). If the value of either subscript exceeds ten, however, you must use a DIM statement to define how much space your array requires, *before* your first reference to a member of that array. Again, if your two-dimensional array has very few members, space is conserved by giving a DIM statement so that the computer will allow only the amount of space that you require.

Character Arrays

A character array is limited to one dimension, and the array must contain only character data. A character array is named by a single alphabetic character followed by a dollar sign (i.e., A$,...,Z$,#$,@$, and $$). The following is an example of a character array (A$) which contains twelve members:

A$(1)	'MARY ADAMS'
A$(2)	'JOHN BROWN'
A$(3)	'FRED CLAY'
A$(4)	'SARAH DUNN'
A$(5)	'SAMUEL EVANS'
A$(6)	'JACK FROST'
A$(7)	'RUTH GOLD'
A$(8)	'RICHARD HOWE'
A$(9)	'WAYNE IVANS'
A$(10)	'ETTA JACOBS'
A$(11)	'CHARLES KLEIN'
A$(12)	'SUSAN LOWE'

Values are assigned to character arrays through LET statements, and through READ and INPUT statements as explained below. A character array may not be used in a MAT statement (discussed later in this chapter).

Input Values for Arrays

Initially the system sets all arithmetic arrays to zero and all character arrays to blanks (18 blanks for each array member). Arrays can be given other values through READ and INPUT statements just like other variables, and through various MAT statements (arithmetic arrays only) which are discussed later in this chapter. However, when supplying input values for arrays by means of READ and INPUT statements, you must remember that every array member that is to receive a value must be represented in the statement and a value must be typed for each member specified. Consider the following statements:

```
10 dim x(5), y(12)
20 input x(1), x(2), x(3), x(4), x(5), y(4)
```

The DIM statement says that X is an array representing five arithmetic values and Y is an array representing twelve arithmetic values. The INPUT statement says that you will assign values to all five members of X and to the fourth member

of Y. Execution of the INPUT statement causes the computer to print a question mark (?) at the terminal. A valid response would be:

```
? 25, 33, 17, 62, 95, 43
```

The first five values of the input line are assigned to X(1) through X(5), respectively. The last value is assigned to Y(4).

After the input line has been typed, the values of the variables in the data list are as follows:

X(1)	is 25	Y(4)	is 43
X(2)	is 33		
X(3)	is 17		
X(4)	is 62		
X(5)	is 95		

Note that Y(1) through Y(3) and Y(5) through Y(12) are left untouched.

Another way of assigning input values to arrays is through use of a FOR/NEXT loop in conjunction with the READ and DATA statements. For example, if you wanted a list of 15 numbers assigned to an array named A, you could write:

```
10 dim a(15)
20 for i = 1 to 15
30 read a(i)
40 next i
50 data 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47
```

The subscript I is used to step through the numbers in the DATA statement.

Similarly, you could initialize a 3 by 5, two-dimensional array (named B) by using nested FOR/NEXT loops, as follows:

```
10 dim b(3,5)
20 for i = 1 to 3
30 for j = 1 to 5
40 read b(i,j)
50 next j
60 next i
70 data 2,3,4,5,6
80 data 7,8,9,10,11
90 data 12,13,14,15,16
•
•
•
```

In this example, I represents the number of rows (3) in the array B and J represents the number of members in each row (5). For each iteration of I, J is stepped through five times; that is, for each value of I, J is equal to one through five. Thus, when I is one, the values for B(1,1), B(1,2), B(1,3), B(1,4), and B(1,5) are read into the array. When I is two, the values for B(2,1) through B(2,5) are read into the array, and when I is three, the values for B(3,1) through B(3,5) are read into the array. When all fifteen members have been read, both loops are satisfied and processing continues with the next executable statement in the program. Rules for using nested FOR/NEXT loops are given under the heading "Program Statements" in Part II of this publication.

MAT input/output statements (discussed in the next section) can be used to assign values to entire arithmetic arrays without the necessity of using FOR/NEXT loops.

Matrix Operations (MAT Statements)

Arithmetic arrays that have been defined implicitly through usage or explicitly in DIM statements can be used subsequently in MAT statements. Arithmetic arrays can be used in the following MAT input/output statements:

MAT GET	MAT PRINT	MAT PUT
MAT INPUT	MAT PRINT USING	MAT READ

Each of these statements must include the word MAT, as shown. They perform the same functions as their non-MAT counterparts. For example, MAT READ reads values from DATA statements and assigns them to an array variable according to the dimensions and bounds of that array; if there are not enough values to fill the array, execution is terminated.

MAT INPUT, MAT GET, and MAT READ statements allow an array to be redimensioned; that is, new bounds can be specified for the array, provided that the original number of members is not exceeded and the original number of dimensions is not changed. For example, consider the following:

```
10 dim a (10,2)
•
•
•
80 mat input a(4,3)
```

The MAT INPUT statement changes the bounds of A as specified by the parenthesized numbers following A. Since the new size of A (12 members) is less than the defined size (20) and the number of dimensions is the same, the statement is valid. Therefore, twelve values are read from the terminal and assigned to A in a 4 by 3 configuration.

One other MAT statement is available for matrix operations, the MAT assignment statement. This statement provides true mathematical matrix functions (e.g., matrix multiplication and inversion) for two-dimensional arrays and other operations (e.g., addition and subtraction) for one- and two-dimensional arrays. In certain cases, redimensioning is allowed. Table 1 illustrates most of the properties of the MAT assignment statement. Rules governing the use of MAT statements are given in Part II of this publication.

Table 1. Matrix Assignment Examples

Assuming this DIM statement exists
`DIM A(5,5), B(5,5), C(5,5)`
consider each of the following MAT assignment statements:

Example	Effect
`MAT A = B`	The members of B are assigned to the corresponding members of A. A and B must have the same dimensions.
`MAT C = A+B`	The members of B are subtracted from the corresponding members of A and the resulting array is assigned to C. A, B, and C must have the same dimensions.
`MAT C = A-B`	Corresponding members of A and B are added and the resulting array is assigned to C. A, B, and C must have the same dimensions.
`MAT C = (3)*A`	Every member of A is multiplied by 3 and the resulting array is assigned to C. A and C must have the same dimensions and the arithmetic expression must be in parentheses.
`MAT C = A*B`	The result of the mathematical matrix multiplication of B by A is assigned to C. A, B, and C must be two-dimensional and the rows and columns must have the following relationship: $C_{ik} = A_{ij} * B_{jk}$
`MAT C = INV(B)`	C is assigned the matrix inverse of B. Both must be two-dimensional square arrays and have the same dimensions.
`MAT C = TRN(B)`	C is assigned the matrix transpose of B. Both must be two-dimensional and their rows and columns must have the following relationship: $C_{ij} = B_{ji}$
`MAT C = IDN(4,4)`	C is redimensioned to a 4 by 4 configuration and assigned an identity matrix of that size. (Redimensioning is optional.) The array must be square.
`MAT C = CON(4,3)`	C is redimensioned to a 4 by 3 configuration and 1 is assigned to every member of C. (Redimensioning is optional.)
`MAT C = ZER(2,5)`	C is redimensioned to a 2 by 5 configuration and zero is assigned to every member of C. (Redimensioning is optional.)

Large and Small Numbers

In ITF:BASIC, short-form computation results have seven significant digits. Integer format (or I-format) and fixed-decimal format (or F-format) are used to represent numbers whose absolute values are in the range 9,999,999 to 0.1. However, very large and very small numbers can still be represented so long as seven-digit significance is sufficient. (Numbers requiring more than seven-digit significance are described below.) The mass of Earth, for example, about 6.6 sextillion tons, would be written in full as:

6,600,000,000,000,000,000,000

This is normally expressed in scientific notation as 6.6×10^{21} and in BASIC would be expressed in the following comparable format (called *exponential* format, or E-format):

6.6E+21

The E stands for "exponent to the base 10" and the number following the E is the exponent. Very small numbers are treated similarly. The mass of a proton (1.7×10^{-24} grams) would be expressed in E-format as:

1.7E−24

E-format can be used for input or for constants within the program. The computer automatically uses E-format for output when the absolute value is less than 0.1 or greater than 9,999,999, as illustrated by the third and fourth values printed by the program in Figure 4. Note that E-format is not used to print the value zero; zero is printed as 0.

You can specify E-format for output in the Image statement accompanying a PRINT USING statement in this manner:

```
100 :THE MASS IS #.#####!!!! GRAMS
```

The number signs represent the mantissa of the number (the mantissa includes a decimal point and an optional sign); four exclamation points (!) or "or" signs (|) are used for the four-position characteristic, which can range from E-79 to E+75.

```
logon joe proc(itfb)
IKJ56455I JOE LOGON IN PROGRESS AT 11:45:46 ON MAY 4, 1971
READY
edit exp basic new scan
ITF INITIALIZATION PROCEEDING
INPUT
00010 print 2**19
00020 print (2**19)/1000
00030 print 2**100
00040 print 1/(2**100)
00050 (CR)
EDIT
run
 524289.7
 524.2896
 1.267661E+30
 7.888541E-31
EDIT
end
READY
logoff
IKJ56479I JOE LOGGED OFF TSO AT 11:48:10 ON MAY 4, 1971
```

Figure 4. Fixed-decimal Format (F-format) and Exponential Format (E-format)

Seven-digit precision is sufficient for most purposes, but there are situations where greater significance is necessary. When you request "long" precision for a

program run (by typing RUN LPREC in the edit mode; or by typing BASIC LPREC or RUN BASIC LPREC in the command mode), internally computation is carried out to 16 significant digits. Integers are printed with 15 significant digits, while values printed in E-format have eleven significant digits in the mantissa. The Image statement accompanying a PRINT USING statement can specify all 16 digits. Actually, the Image statement can specify more than 16 digits. This will result in padding with zeros for arithmetic data.

The program in Figure 5 approximates the sum:

$$S = \sum_{n=0}^{\infty} \frac{1}{x^n}$$

Statement 10 gives x a value of 1.065, and statement 20 sets N and S (the sum) to zero. Statement 30 computes the term and adds it into the sum. Statement 40 increases the value of N. Statement 50 compares the tentative "next sum" with the sum at that point, and, if they are different, the computer repeats the sequence 30, 40, 50. When the computer's limits of precision are reached, it cannot tell the difference between the "present sum" and the "next sum," and the program ends.

```
logon joe proc(itfb)
IKJ56455I JOE LOGON IN PROGRESS AT 16:27:30 ON MAY 7, 1971
READY
edit sum basic new scan
ITF INITIALIZATION PROCEEDING
INPUT
00010 let x = 1.065
00020 let n,s = 0
00030 let s = s+1/(x**n)
00040 let n = n+1
00050 if s<>s+1/(x**n) then 30
00060 print "number of terms:", (n-1)
00070 print "the sum of terms:", s
00080 print "the last term:", 1/(x**(n-1))
00090 end
00100 (CR)
EDIT
save
SAVED
run
NUMBER OF TERMS:   176
THE SUM OF TERMS:  16.383634
THE LAST TERM:     1.536384E-05
EDIT
run lprec
NUMBER OF TERMS:   528
THE SUM OF TERMS:  16.3846153846145
THE LAST TERM:     3.6258271359E-15
EDIT
end
READY
logoff
IKJ56479I JOE LOGGED OFF TSO AT 16:30:44 ON MAY 7, 1971
```

Figure 5. Approximation of an Infinite Sum

Figure 5 shows the program run twice. The sum in the first run (in normal "short" precision) contains 176 terms. BASIC long precision is requested for the second run by the subcommand:

```
RUN LPREC
```

Over 500 terms are included in the sum this time, and the answer of 16.384615+ is considerably more precise.

Note: Because of the physical limitations of the computer, certain values cannot be precisely represented internally, (e.g., 1/3). Computation involving those values may result in a slight loss of precision and, as a result, printed results may be inaccurate in the rightmost one or two significant digits (i.e., in the least significant positions). To overcome this problem, try printing fewer significant digits (by using the Image and PRINT USING statements), or, if these least significant digits are important, try using long-form arithmetic for your computations. You'll probably find that a combination of the two gives the most satisfactory results.

Creating and Using Files

A file is a group of related data items which are treated as a unit. For example, one line of data collected from an experiment may form an item, and the data collected over a period of time may form a series of related items, i.e., a file. Files are created using the PUT statement.

The next few pages show how to create and use a file. In each example, we will be using the compound interest formula:

$$A = P\left(1 + \frac{R}{100}\right)^T$$

where P is the amount originally invested or deposited, R is the annual interest rate, and T is the number of years involved. A, the amount available at the end of T years, is unknown to us.

The program in Figure 6 calculates 200 values of A (from 1% to 20% for each 10 years), and it puts each one, and the corresponding T and R values, into a file. No output is printed.

Whenever a file is created, it must be named. This is done by including the file name enclosed in single or double quotes in every PUT statement of your program. For example:

```
10 PUT 'TF', A, B
```

would create an output file, name it TF, and place the values of A and B in it.

Naming Files

Normally, file names can be any length, but because ITF and TSO recognize only the first three characters of file names, it is recommended that you choose file names that are three characters or less (you'll see why later). These three characters should adhere to the following TSO file naming conventions:

1. The first character is required and it must be *alphabetic*—any letter of the alphabet (A through Z) or one of the three alphabetic extenders ($, #, and @).
2. The other two characters are optional; if specified, they must be *alphameric*—any alphabetic character (including $, #, and @) or any digit (0-9).

Some examples of valid file names are

```
"int"          '$60'
'ab'           "f#1"
"t"            'r6'
'cos'          "#09"
```

When you create a file, it is automatically saved in an area of permanent storage set aside for your files. The number of files that this area can contain is determined by your installation. Usually, this number is at least twenty. If you have used all the space reserved for your files and you attempt to create a new file, an error message will be displayed at your terminal. In order to create your new file, you must do some "housekeeping[1]" and remove any unwanted files from permanent storage to make room for your new file. This can be done by using the DELETE command. (Just how to use DELETE is shown later in the chapter "Errors and

[1] Detailed information about "housekeeping" and file maintenance is given in Appendix D.

Corrections" under the heading "Modifications in the Command Mode.") The DELETE command (as well as the RENAME command) requires that file names conform to the file naming conventions given earlier. Names which do not conform to these rules cannot be used in the TSO DELETE and RENAME commands.

ITF:BASIC actually places fewer restrictions on file names than TSO does. Because of this, it is entirely possible to create a file (with a valid ITF:BASIC name but an invalid TSO name) and use it, but never be able to delete it or rename it. It will always be taking up space in permanent storage; space that you may need later on. This problem arises in part because when a file is named (and is actually created) in the first PUT statement of your program, the syntax of that statement (including the file name) is checked against what is acceptable to ITF, not against what is acceptable to TSO. Consequently, you do not receive notification that your file name is not acceptable to TSO until you try to use that name in a DELETE or RENAME command.

ITF:BASIC will accept *any* file name that is acceptable to TSO. If you make certain that *all* your file names are acceptable to TSO, file name conflicts will never arise. The responsibility for typing an acceptable TSO file name is yours alone, because when you create a file, ITF will notify you of an error only if your file name violates one of the following ITF:BASIC conditions:

1. The first three characters cannot contain a period, a comma, or a semicolon;
2. A blank cannot precede a non-blank in the first three characters;
3. The first three characters cannot be all blank; and
4. The file name cannot be a null character string (two adjacent quotation marks).

To illustrate the relationship between what TSO accepts and what ITF accepts, let's look at some more examples of file names you might supply:

NAME YOU SPECIFY	ACCEPTABLE TO ITF?	ACCEPTABLE TO TSO?
"q"	yes	yes
'f#1'	yes	yes
"$60"	yes	yes
"3t"	yes	no
'a&b'	yes	no
"%ft"	yes	no
"av."	no	no
' rt'	no	no

File Name Length

As we've already mentioned, only the first three characters of file names are recognized and retained by ITF. Because of this, you can use only these three characters (or fewer, if fewer were specified) in the DELETE and RENAME commands. It is possible, however, to use longer file names in ITF:BASIC statements. If you choose to do so, you must remember that only the first three characters can be used when you delete or rename a file. Look at the following examples:

NAME YOU SPECIFY	ITF RETAINS	ACCEPTABLE FORM IN DELETE AND RENAME COMMANDS
'q file'	Q	Q
'$60 million'	$60	$60
"cost sheet"	COS	COS
'f2 form'	F2	F2

As you can see, any trailing blanks (a blank in the third character position, or blanks in the second and third character positions) are also ignored and should not be specified in the DELETE and RENAME commands.

Because ITF ignores everything beyond the first three characters of a file name, any combination of characters can be used in character positions beyond the third in file references within your program (e.g., in GET, PUT, CLOSE, and RESET statements). As long as the first three characters are always identical, the remaining characters may even vary from statement to statement. For example, a file named

"interest" can be referred to as "interest", "int", "inter", or even "int;pq. fd", and ITF will always recognize it as the same file (INT).

```
logon joe proc(itfb)
IKJ56455I JOE LOGON IN PROGRESS AT 14:24:56 MAY 7, 1971
READY
edit int basic new scan
ITF INITIALIZATION PROCEEDING
INPUT
00010 read p
00020 data 1000.00
00030 for t = 1 to 10
00040 for r = 1 to 20
00050 let a = p*(1+r/100)**t
00060 put 'tf',t,r,a
00070 next r
00080 next t
00090 end
00100 (CR)
EDIT
save
SAVED
run
EDIT
end
READY
logoff
IKJ56479I JOE LOGGED OFF TSO AT 14:30:39 ON MAY 7, 1971
```

Figure 6. The Output Data File

Creating a File

In Figure 6 the PUT statement (statement 60) is similar to a PRINT statement such as:

```
10 PRINT T, R, A
```

except that it includes the name of the file to be created ('TF') enclosed in single or double quotes. As far as the computer is concerned, both PUT and PRINT mean output; the only difference is whether the output goes into a file or is printed at your terminal. By changing PUT to PRINT (removing the file name 'TF'), you can run the program and have all 200 values of A printed at your terminal.

There are essentially two ways of putting values into a file: calculating all the values in a statement or typing them individually. The program in Figure 6 generated a file from the repeated execution of a statement, using nested FOR/NEXT loops (see Part II for rules governing the use of nested FOR/NEXT loops).

End-of-file Indicator

In Figure 7, an INPUT loop is used so that individual values can be entered from the terminal. At statement 10 the computer asks for input; at statement 20 it puts the four values (month, day, year, and price) into the file 'QF'. At statement 30 the computer tests the Y value against 70. If the year entered was less than 1970, the computer goes back to statement 10 and asks for another set of values. When 70 is entered for Y the program ends. Notice that, in this example, the year 1970 is used as an "end-of-file" indicator. If, by means of a subsequent GET, you try to obtain more values from a file than it actually contains, execution of your program is immediately discontinued. It is, therefore, essential that you know how many items each of your files contains or that you actually place some sort of end-of-file indicator (like 70 in the example above) in your file and test for that

```
logon joe proc(itfb)
IKJ56455I JOE LOGON IN PROGRESS AT 09:45:53 ON MAY 8, 1971
READY
edit que basic new scan
ITF INITIALIZATION PROCEEDING
INPUT
00010 input m, d, y, p
00020 put 'qf', m, d, y, p
00030 if y<70 then 10
00040 end
00050 (CR)
EDIT
save
SAVED
run
?     1, 2, 69, 48.75
?     1, 3, 69, 48.375
?     1, 6, 69, 48.25
?     1, 7, 69, 48.75
?     1, 9, 69, 49
?     1, 10, 69, 49.125
?     1, 13, 69, 49

?     12, 29, 69, 45.50
?     12, 30, 69, 45.625
?     1, 2, 70, 45.625
EDIT
end
READY
logoff
IKJ56479I JOE LOGGED OFF TSO AT 09:55:15 ON MAY 8, 1971
```

Figure 7. The Input Loop Used To Put Values into a File

indicator in your program. When using MAT GET (see "Program Statements" in Part II), you must actually know how many items your file contains.

Activating and Deactivating Files

Files must be activated or "opened" before they can be used. This is done by the system at the first appearance of the file name in a PUT statement for output files or in a GET statement for input files.

Normally, a file is deactivated or "closed" by the system after execution of your program. However, if you want to switch an input file to output (or vice versa) and continue to use it in the same program, you must explicitly deactivate it by using the CLOSE statement. (If you did not do so and you attempted to use an output file for input or an input file for output, execution of your program would be terminated.) CLOSE deactivates the file; a subsequent GET or PUT statement reactivates (or opens) the file for its new use and repositions it at the beginning. For example:

```
.
.
.
40 put 'af', a, b, c, d, e
.
.
.
80 close 'af'
90 get 'af', a, b, c, d, e
```

Statement 40 creates an output file named 'AF' and places five values in it. At statement 80, 'AF' is deactivated for output. In statement 90, 'AF' is reactivated as an input file and the same five values are read and made available for use later in the program.

Notice what happens when an input file is closed and reactivated as an output file.

```
•
•
•
40 get 'af', a, b, c, d, e
50 let b = a
60 let a = 36
70 let c = c+b
80 let d = a/b
90 let e = a**3
100 close 'af'
110 put 'af', a, b, c, d, e
•
•
•
```

A previously created file, named 'AF', is activated for input in statement 40 and five values are made available to the program. In statements 50 through 90, new values are acquired for A, B, C, D, and E. Statement 100 deactivates 'AF' as an input file; statement 110 reopens the file for output and places the new values for A through E into the file. Actually, 'AF' is now a new file and any values could be placed in it, not necessarily A, B, C, D, and E.

Repositioning Files

You may have an occasion to use an input file for input or an output file for output more than one time in the same program. To do this, you need to reposition the file so that each time you reuse it, it is set at the beginning. The RESET statement allows you to reposition the file without deactivating it (deactivation is necessary only when the function of the file is changed from input to output or vice versa). For example:

```
•
•
•
50 get 'bf', x,y,z,q,r,s
•
•
•
100 reset 'bf'
110 get 'bf', x,y,z,q,r,s
•
•
•
150 reset 'bf'
160 get 'bf', x,y,z,q,r,s
•
•
•
```

Between statements 50 and 100, the variables X, Y, Z, Q, R, and S might be used in one set of calculations and their values changed. By repositioning the file, the original values of X, Y, Z, Q, R, and S are available for different calculations or uses between statements 110 and 150 and again between 160 and the end of the program. Actually, the RESET statement functions for files in the same way that the RESTORE statement does for READ and DATA statements.

Once a file has been created, it can be used as input to the same program by deactivating and reactivating the file, as we've already seen, or it can be used as input *to some other program* (not to the one in which it was created) just by using the GET statement. In Figure 8 each set of four 'QF' values is read by the GET statement and the program finds the year's high price and prints it after the file has been completely read.

Initially, H, the variable representing the high price, is set to zero (remember that all arithmetic variables are initialized to zero by the system). Statement 30 compares the price just read, P, with H. If P is greater, the computer sets H to the new high price and records the date as M1, D1, and Y1. Statement 80 is an unconditional branch; it returns the computer to statement 10.

Statement 20 tests the year shown in each set of incoming values. It branches to the output statements, 90 and 100, when the set of values for 1970 appears (notice that 1970 is the end-of-file indicator that you set up when you created the file).

```
logon joe proc(itfb)
IKJ56455I JOE LOGON IN PROGRESS AT 11:50:22 ON MAY 9, 1971
READY
edit qtl basic new scan
ITF INITIALIZATION PROCEEDING
INPUT
00010 get 'qf', m,d,y,p
00020 if y > 69 then 90
00030 if p < h then 10
00040 let h = p
00050 let ml = m
00060 let dl = d
00070 let yl = y
00080 go to 10
00090 print 'high price:'
00100 print using 110, ml, dl, yl, h
00110 :##/##/##      $##.##
00120 end
00130 (CR)
EDIT
save
SAVED
run
HIGH PRICE:
 1/26/69      $52.00
EDIT
end
READY
logoff
IKJ56479I JOE LOGGED OFF TSO AT 11:54:44 ON MAY 9, 1971
```

Figure 8. Searching a File for a Single Value

The program in Figure 9 is an example of how the same file might be processed to obtain an average price for each of the twelve months. The variable C is used to keep count of each set of values read, and T represents the total of all prices for the month. Remember that, initially, both C and T have a value of zero. M1 represents the current month and statement 10 sets it initially to 1. The table heading (statement 20) is also included in this initializing procedure, since it is printed only once, and statement 30 prints a blank line under it.

The processing of the file begins at statement 40. The computer gets a set of values, checks the month (statement 50), adds the day's price into the monthly total (statement 110), increases the count (statement 120), and goes back to statement 40 for the next set of values.

When the incoming set of values includes a new month, statements 60 through 80 are executed. Statement 60 causes the month and average price to be edited into a print line, as directed by statement 70. Statement 60 also shows the use of an expression in a PRINT USING statement. The expression T/C tells the computer to divide the monthly total of prices by the number of prices and print the answer (average price). Statement 80 checks the incoming year. If the year is 1970 or greater, statement 140 is executed; if the year is less than 1970, statements 90 through 130 are executed. Statement 100 resets price count and monthly total to 0. Statement 110 adds the first price for the new month into the monthly total, while statement 120 increases the count, and statement 130 directs the computer back to statement 40.

```
logon joe proc(itfb)
IKJ56455I JOE LOGON IN PROGRESS AT 16:03:58 ON MAY 9, 1971
READY
edit qt2 basic new scan
ITF INITIALIZATION PROCEEDING
INPUT
00010 let ml = 1
00020 print 'month   avg.'
00030 print
00040 get 'qf', m, d, y, p
00050 if m = ml then 110
00060 print using 70, ml, t/c
00070 :##     ##.##
00080 if y > 69 then 140
00090 let ml = m
00100 let c, t = 0
00110 let t = t + p
00120 let c = c + 1
00130 goto 40
00140 end
00150 (CR)
EDIT
save
SAVED
run
MONTH   AVG.
  1     49.76
  2     49.35
  3     48.91
  4     45.76
  5     44.69
  6     45.10
  7     43.64
  8     44.62
  9     45.10
 10     47.84
 11     46.91
 12     46.03
EDIT
end
READY
logoff
IKJ56479I JOE LOGGED OFF TSO AT 16:10:32 ON MAY 9, 1971
```

Figure 9. Processing All Values in a File

Defining Your Own Functions and Subroutines

In addition to the 24 intrinsic functions supplied as a part of the BASIC language (a list of these functions is given in Part II of this publication), you can define any other function or write a program segment (subroutine) which you expect to use frequently in your program.

Functions

A user function is named and defined by the DEF statement. The name of the defined function must be a single alphabetic character preceded by the letters FN. Thus, you may define up to 29 functions (i.e., FNA, FNB,...,FNZ, FN@, FN#, FN$). For example:

```
10 DEF FNE(X) = EXP(-X**2)
```

defines the function e^{-X^2} using the intrinsic function EXP. The arithmetic variable X, enclosed in parentheses after the function name (FNE) is called a *dummy variable.* The dummy variable is required and must be a simple arithmetic variable. Your function performs its defined calculation on the arithmetic expression value substituted for this variable. (The expression value substituted for the dummy variable is called an *argument.*) After defining a function, the function name and its accompanying argument can be used anywhere in your program that an arithmetic expression could appear. For example:

```
10 def fne(x) = exp(-x**2)
20 let y = fne(.5)
30 let z = fne(c+2)
40 print fne(3.75)+y/z
```

Defining functions and using them when you need to perform the same calculation on many values of the same variable can be a great timesaving device.

The DEF statement can appear anywhere in your program, and the expression on the right side of the equal sign can be any arithmetic expression that fits on a single line. As we've already seen, it can include any combination of other functions (even those defined by other DEF statements), but it cannot include a reference to itself. A function can involve other variables besides the dummy variable. The following example is valid:

```
70 DEF FNS(X) = SQR(2+LOG(X)-EXP(Y*X)*(X+SIN(2*Z)))
```

User-defined functions are limited to those instances where the value of the function can be expressed within a single BASIC statement. Often, much more complicated functions or even segments of programs, must be calculated at several different points within the program. For these functions, you can set up a subroutine by using the GOSUB and RETURN statements.

Subroutines

Execution of a subroutine begins with the GOSUB statement, where the number specifies the number of the first statement in the subroutine. For example:

```
100 GOSUB 200
```

causes the computer to skip to statement 200 (in this case, the first statement of a five-statement subroutine) before continuing execution. The last statement of the subroutine must be a RETURN statement which directs the computer to return and execute the statement following the GOSUB. It would look something like this:

```
┌─100 GOSUB 200
┌─┼►110 \
│ │ 120  |
│ │ 130  |
│ │ 140  | These are the statements that
│ │ 150  > will be executed after the RETURN.
│ │ 160  |
│ │ 170  |
│ │ 180  |
│ │ 190 /
│ └►200 \
│   210  | These are the statements that
│   220  > will be executed after the GOSUB.
│   230  |
│   240 /
└───250 RETURN
```

Figure 10, a program that determines the greatest common divisor of three integers using the Euclidean algorithm, illustrates the use of a subroutine. The

```
logon ida proc(itfb)
IKJ56455I IDA LOGON IN PROGRESS AT 11:56:30 ON MAY 2, 1971
READY
edit gcd basic new scan
ITF INITIALIZATION PROCEEDING
INPUT
00010 print 'a','b','c','cd'
00020 read a,b,c
00030 let x = a
00040 let y = b
00050 gosub 140
00060 let x = g
00070 let y = c
00080 gosub 140
00090 print a,b,c,g
00100 goto 20
00110 data 60,90,120
00120 data 38456,64872,98765
00130 data 32,384,72
00140 let q = int(x/y)
00150 let r = x-q*y
00160 if r = 0 then 200
00170 let x = y
00180 let y = r
00190 goto 140
00200 let g = y
00210 return
00220 end
00230 (CR)
EDIT
save
SAVED
run
A              B              C              CD
 60             90             120            30
 38456          64872          98765          1
 32             384            72             8
0668    00000020      MSNG DATA+
EDIT
end
READY
logoff
IKJ56479I IDA LOGGED OFF TSO AT 12:00:42 ON MAY 2, 1971
```

Figure 10. Subroutine Example

first two numbers are selected in statements 30 and 40 and their greatest common divisor (CD) is determined in the subroutine, statements 140-210. The CD just found is called X in statement 60, the third number is called Y in statement 70, and the subroutine is entered again from statement 80 to find the greatest common divisor (CD) of these two numbers. The result is, of course, the greatest common divisor of the three given numbers. It is printed out with them in statement 90. Note that execution ends when no more data remains to be read.

The system prints a message at the terminal giving the error message's numeric code, the line at which execution was interrupted, and the text of the message (in this case, indicating that the data has been exhausted), and then it prints "EDIT" on the next line to indicate that you are still in the edit mode. At this point you can do any of the following:

1. Modify your program (adding or deleting statements, or making changes to existing statements)
2. Type "END"—ending the edit mode, and then in response to the system cue "READY":
   a) type "LOGOFF"—ending the terminal session or
   b) or enter another EDIT command and continue your session.

You may use a GOSUB inside a subroutine to "branch" to yet another subroutine. (This is called "subroutine nesting" and is described in Part II of this publication.)

Errors and Corrections

Until you become experienced in using this system, you will probably make some mistakes in typing and logic. This chapter presents information on program modification, the error recognition and messages supplied by the system, and the debugging facility (test mode) provided by ITF—all of which are designed to assist you in correcting your programs.

Program Modification

In the chapter on "Getting Started" we saw how to correct typing errors in an entry before it was actually sent to the computer. This section is concerned with showing you how to modify all or part of a program even though the statements affected have been sent to the computer.

Programs can be manipulated and updated in several ways. You can insert, replace, or delete entire statements in your program; you can change all or part of an existing statement; you can list (have displayed at your terminal) all or part of your program; and you can renumber all or part of the statements in your program. You can also list the names of the programs and data files retained in permanent storage,[1] or delete entirely any number of programs or data files from permanent storage. Most of these modifications are performed through descriptively named subcommands (DELETE, LIST, RENUM, etc.); the others are performed through the command-like properties of statement numbers.

Modifications in the Edit Mode

Most program modifications are performed in the edit mode, which is to say that an EDIT command must be in effect for the program you wish to modify. For example, if you wish to change an existing program called PRG, this EDIT command must be in effect:

```
edit prg basic old
```

After this, you can make various kinds of corrections or alterations to PRG.

Deleting Statements

You can delete one or more statements from your program by using the DELETE subcommand. For example,

```
delete 130
```

deletes statement 130 from the program being edited. Similarly,

```
delete 40 180
```

deletes statements 40 through 180 from the program. Finally,

```
delete
```

deletes *all* statements from the program.

There is one other way in which you can delete single statement lines. Just type the number of the statement that you wish to delete and follow it immediately with a CR. For example, to delete statement 230, you could type

```
230 (CR)
```

[1] Programs must be explicitly placed in permanent storage by use of the SAVE subcommand; files are automatically retained in permanent storage when they are created.

Inserting and Replacing Statements

To insert one or more statements into your program, you have a choice between typing your own statement numbers or using the INPUT subcommand. Generally, to insert one or two statements here and there, it is easier and more efficient to type your own statement numbers. However, if you want to insert a block of statements, it is probably better to use the INPUT subcommand.

To insert a statement by the statement-number technique, simply give the statement a number that fits between those of the two surrounding statements and type it. For example, let's assume that you want to insert a PRINT statement between statements numbered 80 and 90 of your program. Choose a number greater than 80 and less than 90 and then type the statement as shown here:

```
85 print a,x,b
```

Provided no statements exist between 80 and 90, any number from 81 through 89 could be used to cause the insertion. Statement number 85 was chosen to allow for further insertions, should any be needed at that point in your program.

If statement 85 already existed, then the contents of the existing statement would be *replaced* by the above PRINT statement.

To insert statements by the INPUT subcommand, you must specify the INPUT subcommand with the number of the statement you wish to insert. You can also specify a smaller increment for the new statement numbers so that they fit between the numbers of the existing statements. For example, assume that your program begins as follows:

```
00010 dim a(5,5)
00020 mat a = con
00030 mat print a
```

To insert three statements between statements 20 and 30, to number the first insertion 22, and to increment the following insertions by 2, your sequence would look like this:

```
EDIT
input 22 2
00022 dim b(5,5)
00024 mat b = (3)*a
00026 mat a = inv(b)
00028 (CR)
EDIT
```

Notice that the INPUT subcommand causes the automatic statement numbering to start with statement 22 as specified. Subsequent statements are incremented by 2, also as specified. If the increment had not been specified, the standard increment of 10 would have been assumed and only one statement (22) could have been inserted; there would not have been room for the subsequent statements and an error message would be given after the insertion of statement 22.

You can use the INPUT subcommand for replacements too. A full discussion of this use of the INPUT subcommand is given in the *TSO Terminal User's Guide* (see the preface).

Insertions and replacements cannot be done during program creation when you are using the automatic statement numbering supplied by TSO (i.e., the input phase is in effect). If, however, you number your own statements, insertions and replacements (using the statement-number technique) can be made at any time during the creation of a program.

Adding Statements to the End of Your Program

You can add statements to the end of your program in the same way that you insert statements into a program; i.e., (1) you can type the statement numbers yourself, or (2) you can use the INPUT subcommand.

For example, assume that you have an old program named BAL and you want to add three or four statements to the end of it. Let's also assume that the last

statement currently in BAL is numbered 240 and that it is *not* an END statement. Figure 11 shows both ways of doing it.

On the left-hand side of Figure 11, you type your own statement numbers. The numbers used in this case are 250, 260, and 270, but they could have been any three valid numbers greater than 240. These statements are added to the end of the program, which is then saved under the old name BAL. The updated version of BAL thus replaces the previous version of BAL in your permanent storage.

ADDING STATEMENTS VIA OWN NUMBERS	ADDING STATEMENTS VIA INPUT SUBCOMMAND
READY edit bal basic old scan ITF INITIALIZATION PROCEEDING EDIT 250 let x = cos(y) 260 print x,y,z 270 end save SAVED end READY	READY edit bal basic old scan ITF INITIALIZATION PROCEEDING EDIT input INPUT 00250 let x = cos(y) 00260 print x,y,z 00270 end 00280 (CR) EDIT save SAVED end READY

Figure 11. Two Ways of Adding Statements to the End of a Program

On the right-hand side of Figure 11, the INPUT subcommand is used to generate the numbers for the statements to be added. Notice that nothing is specified in the subcommand so numbering starts with the next available statement number in the program (250) and the standard increment of 10 is used for the subsequent numbers. As you can see, the INPUT subcommand method takes up more space on your paper, but the number of characters that *you* actually type is slightly less, because the system is typing the statement numbers for you. If you intend to add many statements to your program, this method is more efficient; for just a couple of statements, however, it would be just as fast for you to type your own numbers.

Changing Parts of Statements

You can change a part of one or more statements without retyping those statements by using the CHANGE subcommand. Essentially, there are two ways to make changes using this subcommand: (1) you can specify the actual sequence of characters to be changed and what they are to be changed to, or (2) you can cause one or more statement lines to be partially displayed and then complete those statements yourself. Rather than present the format of the CHANGE subcommand for these two methods, we'll deal here with examples of each. (You can refer to Part III for the syntax of CHANGE.)

Let's consider some examples of the first method. Assume that you want to change the name of variable A1 to B1 every place that it appears in your program. Further, the statements in your program are numbered from 10 to 380 and the first use of A1 appears in line 40. If you were to retype every statement that contained a reference to A1, you might have a long and tedious job on your hands. Just one CHANGE subcommand, however, would do the job for you:

```
change 40 380 !a1!b1!all
```

Statement numbers 40 and 380 in this subcommand specify the range of statements (i.e., from 40 to 380 inclusive) through which the change is to be made. In this instance, the exclamation character (!) is called a *special delimiter;* the system

requires it to recognize the sequence of characters to be changed and to recognize its replacement. You can use any character as a special delimiter except digits, blanks, commas, semicolons, parentheses, tabs, and asterisks. The system always recognizes the first character after the last statement number as the special delimiter for a particular use of the CHANGE subcommand (provided the character is not one of the previously mentioned exceptions). Thus, the exclamation point is recognized as the special delimiter for *this use* of the CHANGE subcommand. The sequence of characters appearing between the first and second use of the special delimiter (A1) is recognized as the sequence to be changed. The sequence between the second and the third use of the special delimiter (B1) is recognized as the replacement sequence. The word ALL that follows the third special delimiter means that *every* appearance of A1 within statements 40 through 380 is to be change to B1. If ALL were omitted, only the first appearance of A1 would be changed to B1; all subsequent appearances would remain the same. When ALL is omitted, the third special delimiter is not required.

Now let's assume that you want to change the file name 'BF' in the same program to the file name 'QF'. You don't remember exactly where 'BF' appears but you know that it appears only once in the program. This CHANGE subcommand would do it:

```
change  10 380 /'bf'/'qf'
```

The range of statements encompasses the whole program and the special delimiter is /. The system searches the program until it finds the sequence 'BF' and then replaces it with 'QF'.

If you knew that 'BF' appeared in statement 160, however, you could have specified this instead:

```
change 160 /'bf'/'qf'
```

To take one more example of this method, consider this statement in your program:

```
00145 print a(6),a(6),c,d,e,f
```

You want to change the second appearance of A(6) to B(6). If you specify just A as the sequence of characters to be changed, or even A(6), then just the first appearance of that sequence in the statement would be changed, which is not what you want. To ensure that the proper change is made, you must make certain that the sequence of characters to be changed is unique, as in this subcommand:

```
change 145 !,a!,b
```

Once again the special delimiter is the exclamation point. The comma before the A uniquely identifies the sequence to be changed. Thus, the sequence ",a" is replaced by ",b". Note that if you wanted to change both appearances of A(6) to B(6), you could specify:

```
change 145 !a!b!all
```

So much for the first method. The second method allows you to display a specified number of characters or display up to, *but not including*, a specified sequence of characters. In each case, you type the remainder of the statement after the display. For example, assume that statement 230 in your program is 66 characters long (statement number field excluded) and the last four characters are wrong. This CHANGE subcommand

```
change 230 60
```

causes the first 60 characters of the statement to be displayed immediately. The system does not return the print element to the next line. It waits for you to type the remainder of the statement. When you have completed the statement, the system replaces statement 230 with the updated statement.

If you don't want to count characters, you can specify a sequence of characters, up to which the statement is to be displayed. For example, if this erroneous statement exists in your program

```
00165 get 'ab',a,b,c,d?e,f
```

this CHANGE subcommand

```
change 165 !?
```

results in the following display and reply:

```
00165 GET 'AF',A,B,C,D,e,f
```

typed by user (",e,f")

The CHANGE subcommand has caused statement 165 to be displayed up to but not including the "?". On the same line, you have typed the rest of the statement correctly. The number of characters you type doesn't have to be the same as the number of characters you're changing. It can be more or less; it doesn't matter, since the undisplayed portion of the statement is considered deleted.

Note the use of the exclamation point in the subcommand. As in the first method we discussed, a special delimiter is required whenever a sequence of characters is specified.

Renumbering Statements

You can renumber all or part of your program by using the RENUM subcommand. To renumber the entire program, you can simply specify

```
renum
```

and the system will renumber the program using the standard increment of 10. You can specify your own number for the first statement of your program and have the rest of the program renumbered accordingly. For example, if you want the first statement of your program to have number 75, specify

```
renum 75
```

and the statements of your program will be numbered 75, 85, 95, etc. (again the standard increment of 10 is assumed). If you wish to change the increment, you must specify two numbers: a number for the first statement of your program and an increment value (any integer from 1 to 99999). For example,

```
renum 15 5
```

gives the statements of your program the numbers 15, 20, 25, etc.

To renumber part of your program, you must specify three numbers: a new number for the statement at which renumbering is to begin, an increment value, and the actual statement at which renumbering is to start. For example,

```
renum 95 15 80
```

causes statement 80 to be renumbered as 95; all subsequent statements are renumbered accordingly, with an increment of 15. When you use this form of the RENUM subcommand, the first number must never be less than or equal to the number of the statement immediately preceding the statement at which renumbering is to begin. For example, if your program is numbered 10, 20, 30, 40, 50, 60, 70, 80, etc.,

```
renum 75 5 80
```

is valid, but

```
renum 65 5 80
```

is not valid.

Displaying Statements

After you have performed many insertions, replacements, deletions, etc., you will probably want to list your program to see what you have. You can obtain a partial or complete listing of your program by using the LIST subcommand. To obtain a complete listing, specify

```
list
```

For a partial listing, you must include statement numbers in the subcommand. One statement number indicates that one statement is to be listed. For example,

```
list 130
```

displays statement 130. Two statement numbers specify a range of statements, thus, the subcommand

```
list 110 180
```

displays statements 110 through 180.

You can cancel a listing while it is in progress by giving an attention interruption. To restart a listing once it has been stopped, you must specify a new LIST subcommand.

Figure 12 illustrates how you can change, insert, replace, and delete statements as you create a program typing your own statement numbers. We've included two LIST subcommands in this example. The first shows how the program appears after all the changes have been made. The second shows how the program looks after it has been renumbered. Because we saved the program before leaving the edit mode, the second listing of the program also represents the program as it is retained in permanent storage. As you can see, each occurrence of the LIST subcommand causes the system to display the program in statement-number sequence immediately after the subcommand was issued. All lower-case letters are converted to upper case by the system and hence appear as upper case in the listing.

Modifications in the Command Mode

In the command mode you cannot modify the actual contents of a program (as you can in the edit mode), but you can change the name of a permanent program or the name of a data file. You can also remove any programs or data files that are currently being retained in permanent storage. And, you can list the names of the programs and data files which are retained in permanent storage. This last facility is useful if you have forgotten the names of any of your permanent programs or data files.

Renaming Programs and Data Files

Programs and data files can be given new names in the command mode by using the RENAME command. The format of the RENAME command is

RENAME *old-name new-name*

To change the name of an existing program named ABC to XYZ, simply specify

```
rename abc xyz
```

and henceforth, ABC will be known by the system as XYZ.

If 'ABC' were the name of a data file and you wanted to change that name to 'XYZ', you would type the following RENAME command:

```
rename data(abc) data(xyz)
```

Notice that the file name (without the surrounding quotation marks) is enclosed in parentheses and that it is directly preceded by the word DATA. Because data files and programs are kept in different areas of permanent storage and because (except for the surrounding quotation marks) file names and program names are

```
READY
edit abc basic new noscan
ITF INITIALIZATION PROCEEDING
INPUT
00010 (CR)}  ──────────→ termination of automatic statement
EDIT                     numbering (input phase)
10 rem  An example of program modification.
20 rem  Not intended for execution.
30 input a
40 let b = 0
50 let b = a
60 let c = 2*b
70 print b,c
80 let z = sqr(b+c)
90 print 'z is', z
64 let d = 3*c}
67 let e = 4*d}  ──────────→ insertions
70 print b,c,d,e ──────────→ replacement
100 let x = sqr(d+e)}
110 print 'x is', x } ──────→ additions
120 end             }
change ¢c¢q¢all} ──────────→ change every occurrence of "c" to "q"
delete 40}  ──────────────→ deletion
list
00010 REM  AN EXAMPLE OF PROGRAM MODIFICATION.
00020 REM  NOT INTENDED FOR EXECUTION.
00030 INPUT A
00050 LET B = A
00060 LET Q = 2*B
00064 LET D = 3*Q
00067 LET E = 4*D
00070 PRINT B,Q,D,E
00080 LET Z = SQR(B+Q)
00090 PRINT 'Z IS', Z
00100 LET X = SQR(D+E)
00110 PRINT 'X IS', X
00120 END
IKJ52500I END OF DATA
renum
list
00010 REM  AN EXAMPLE OF PROGRAM MODIFICATION.
00020 REM  NOT INTENDED FOR EXECUTION.
00030 INPUT A
00040 LET B = A
00050 LET Q = 2*B
00060 LET D = 3*Q
00070 LET E = 4*D
00080 PRINT B,Q,D,E
00090 LET Z = SQR(B+Q)
00100 PRINT 'Z IS', Z
00110 LET X = SQR(D+E)
00120 PRINT 'X IS', X
00130 END
IKJ52500I END OF DATA
save
SAVED
end
READY
```

Figure 12. Example of Modifying a Program As It Is Being Created

sometimes identical, you must take care to differentiate between the two in the RENAME command. Consequently, when renaming files, both *old-name* and *new-name* must be preceded by the word DATA and the two file names (minus the surrounding quotation marks) must be enclosed in parentheses. To further illustrate this point, let's look at another example. To change the name of a data file known currently as 'INT' to 'TAX', specify the following:

```
rename data(int) data(tax)
```

File Name Warning

A word of caution about file names before we continue. In the chapter "Creating and Using Files" in Part I of this book, you learned that, although ITF recognizes and retains only the first three characters of each file name, you may actually use a longer name in GET and PUT statements (provided the first three characters of each name are unique and that certain characters are not used). In the command mode RENAME and DELETE commands, certain other restrictions also apply. They are:

1. File names must not be enclosed in quotation marks—they must be enclosed in parentheses and they must follow the word DATA.
2. File names must not be more than three characters in length (if you are using a longer name in your GET and PUT statements, make certain that you use only the first three characters when using RENAME and DELETE).
3. The first character of a file name must be *alphabetic*—the letters A through Z or one of the three alphabetic extenders ($, #, and @).
4. The other two characters of a file name (one or two more may be used) must be *alphameric*—an alphabetic character or any digit (0-9).

Some examples of file names that can be used in the RENAME and DELETE commands are:

VALID	INVALID (*x* REPRESENTS A BLANK)
A	A*x*B
AB	*x*AB
ABC	*xx*A
A1	1A
A12	A&B
A#1	%TF
@A	FILE

Even though you cannot foresee the need to rename or delete a file when you create it, the possibility always exists. Therefore, it is recommended that you follow the restrictions given in rules 3 and 4 above when selecting the first three characters of your file names. In this way, you will be able to use RENAME and DELETE should the need ever arise.

Deleting a Program or Data File

In the command mode, the DELETE command can be used to delete one or more programs or data files that are currently retained in permanent storage. In the edit mode, DELETE has a different function which is described earlier in this chapter under the heading "Program Modifications in the Edit Mode."

In the command mode, to delete a program named AVG, simply type the following:

```
delete avg
```

If you wish to delete more than one program, you must follow the command DELETE with a list of the names of the programs to be deleted. The program names must be separated from each other by a comma or by at least one blank, and the entire list must be enclosed in parentheses. For example, to delete the three programs (PRG, INTEREST, and AMT), simply type either of the following:

```
delete (prg,interest,amt)  or  delete (prg interest amt)
```

As in the RENAME command, deleting data files is slightly more complex.[1] Once again, the name of the file to be deleted (minus the surrounding quotation marks) must be enclosed in parentheses, and it must be preceded by the word DATA. For example, to delete a data file named 'QF', you would type

```
delete data(qf)
```

To delete more than one file in the same command, the word DATA must precede each file name (minus the surrounding quotation marks) and each file name must be enclosed in parentheses. Each repetition of DATA(*file-name*) must be separated from the next by a comma or by at least one blank and the entire list (everything following the word DELETE) must be enclosed in another set of parentheses. So, the command to delete two files ('A1' and 'A2') would look like either of the following:

```
delete (data(a1),data(a2)) or delete (data(a1) data (a2))
```

Files and programs can be specified in the same command in the following manner:

```
delete (cost,data(mf),data(pf))
```

This command would cause the system to delete the program named COST as well as the files named 'MF' and 'PF' from permanent storage.

When specifying more than one item in a DELETE command, make certain that your parentheses are matched (i.e., every left parenthesis has a matching right parenthesis). Unbalanced parentheses will cause the system to reject the command. If this occurs, you will receive an error message, and you will have to re-enter the command correctly.

Displaying Names of Permanent Programs and Data Files

You may find that you don't remember the names of all the programs you've saved or the names of the data files you've created. If this happens, you can use the LISTCAT command (in the command mode) to display the names of the programs or data files currently retained in permanent storage under your user identification code.

To obtain a list of the names of all the programs you've saved, simply type

```
listcat
```

and, immediately following the command, the system will print the name of each of the permanent programs saved under your user identification code. The name of each program will be followed by a period and the word BASIC to indicate that the program is an ITF:BASIC program. For example, the following list might result from a LISTCAT command:

```
AVG.BASIC
COST.BASIC
DATA
INT.BASIC
T20.BASIC
```

Notice the appearance of the word DATA in the above list. DATA is the name of the area in permanent storage where your data files are retained; files are considered to be "members" of DATA. If you have not created any data files, the word DATA will not appear in the list typed in response to LISTCAT.

---

[1] See "File Name Warning" under the discussion "Renaming Programs and Data Files" earlier in this chapter.

To display the names of your files, you must include the word MEMBERS in the LISTCAT command, as follows:

```
listcat members
```

This command could result in the following list:

```
listcat members
DATA
—MEMBERS—
  ONE
  QF
  TF
LOOKUP.BASIC
PRC.BASIC
TBL.BASIC
READY
```

As you can see, there are three files in DATA (ONE, QF, and TF). Also, notice that, when the list or display of names is completed, the system types the system cue READY on the next line to indicate that it has finished its list of names and is ready for your next entry.

Messages

There are five types of messages that you can receive from the system:

- System cues[1]
- Prompting messages
- Informational messages
- Broadcast messages
- ITF error messages

System Cues

System cues tell you when the system is ready to accept a new command or subcommand. When the system is in the command mode and it is ready to accept a new command, it prints READY. When the system is ready to accept a subcommand, it prints the name of the command (or mode) that is in effect; that is, EDIT, or TEST.

You've seen all of these already and there's nothing new that we can add here, except this: in the command mode, you can sometimes save a little time by not waiting for the system cue. For example, if you enter the DELETE and RENAME commands and wait for the intervening READY message between the commands, your listing might look like this:

```
READY
delete data(f04)
READY
rename data(in) data(f04)
```

If you enter these commands without waiting for the intervening READY message, your listing would look like this:

```
READY
delete data (f04)
rename data(in) data(f04)
READY
READY
```

[1] In some other TSO publications, the term "system cue" is sometimes known as "mode message."

There is a drawback to entering commands without waiting for the intervening READY system cues. If you make a mistake in one of the commands, the system sends you a message identifying your mistake and then it cancels the remaining commands you have entered. After you correct the error, you have to re-enter the other commands.

Unless you are certain that there are no mistakes in your lines you should wait for the READY system cue before entering a new command.

Prompting Messages

A prompting message tells you that required information is missing or that information you supplied was incorrectly specified. It then asks you to supply or correct that information. For example, if you end the edit mode without having saved anything, the system will prompt you with the following messages:

```
NOTHING SAVED
ENTER SAVE OR END
```

You should respond by entering the requested information—in this case, the SAVE or END subcommand.

Sometimes the prompting message you receive will end with a plus sign (+). The sign means that you can request another message that explains the initial message more fully. If the second message also ends with a plus sign, you can request a further message that will give you more detailed information, and so on. The last available message will not end with a plus sign.

To request the next level of a message:

1. Type a question mark (?) in the first position of the line.
2. Give a CR.

You can enter question marks as long as the last message you have received ends in a plus sign. If you enter a question mark when the message does not end in a plus sign, you receive the following message:

```
NO INFORMATION AVAILABLE
```

If you give an attention interruption while a message is printing, the message will be terminated at the point of the interruption and no further levels of the message will be available.

Informational Messages

An informational message tells you about the status of the system and your terminal session. For example, it can tell you how much time your session took. Informational messages do not require a response.

If an informational message ends with a plus sign (+), you can request an additional message by entering a question mark (?), as described for prompting messages.

Broadcast Messages

Broadcast messages are messages of general interest to users of the system. Both the system operator and any user of the system can send broadcast messages. The system operator can send messages to all users of the system or to individual users. For example, he may send the following message to all users

```
DO NOT USE TERMINALS 03,04,14, AND 15 ON 6/30. THEY
ARE RESERVED FOR ACCOUNTING DEPARTMENT
```

You can send messages to other users or to the system operator, as you saw in the discussion of the SEND command in an earlier chapter. You can suppress certain types of messages. A complete discussion of this topic is contained in the *TSO Terminal User's Guide* (see the preface).

ITF Error Messages

ITF error messages are numbered messages that identify errors in your usage of ITF:BASIC statements, commands, and subcommands. They can appear only in the ITF test mode, the edit mode, or the command mode. In general, the types of errors they identify fall into three categories:

1. *Syntax errors:* errors in the structure of a statement, command, or subcommand (i.e., erroneous punctuation, illegal operands, misspelled keywords, etc.).
2. *Semantic errors:* errors in the structure of your ITF:BASIC program (i.e., invalid nesting, illegal comparison of data types, missing NEXT statements, etc.).
3. *Execution errors:* errors detected during the execution of an ITF:BASIC program (i.e., dividing by zero, improper subscript values, invalid branches, etc.).

Most of the ITF error messages exist in two levels of information. The first level (short form) is generally a compact (about 20 characters or less) description of your error. The second level (long form) gives a more explicit description of your error. The existence of a second level is indicated when the first level ends with a plus sign (+). Like prompting messages, the second level of an ITF error message can be obtained by typing a question mark (?) at your terminal.

Each ITF error message is identified by a four-digit number[1] preceding the actual message text. For a program, the short form of each message will include the number of the statement containing the error; the long form will not include the line number (see the "Note" below for an exception). For example, message 636 might appear in the edit mode as follows:

```
0636      000000230     EXTRA FORS+
```

The long form of this message (obtained by typing a question mark) would look like this:

```
0636    FOR NESTING EXCEEDS IMPLEMENTATION LIMIT (15)
```

Note: The LMSG option of the RUN and BASIC commands suppresses the short forms of error messages so that only the long forms are displayed for a particular program execution. When short forms of ITF error messages are suppressed, the long forms *will* contain the statement numbers of the statements in error.

All of the ITF error messages are fully documented in message number sequence at the back of this book. Each message is shown in both forms and explanations and corrective actions are provided.

Using the Test Mode To Debug Your Programs

The ITF test mode permits you to closely follow and interact with the execution of your ITF:BASIC programs to locate errors in structure and logic (commonly called "debugging"). This test mode is available only with ITF and cannot be used to debug any non-ITF programs that you may own.

Initiating the Test Mode

You initiate the ITF test mode by specifying the word TEST in any one of the following:

1. The BASIC command, e.g., `basic abc test`
2. The RUN command, e.g., `run xyz basic test`
3. The RUN subcommand of EDIT, e.g., `run test`

Note that a program name is specified in each of the first two cases, but not in the third. Program names are not specified in the edit mode RUN subcommand

[1] Message numbers can be suppressed through the PROFILE command (see the *TSO Terminal User's Guide* listed in the preface.

because only the program being edited can be tested, and its name has already been specified in the EDIT command.

The system indicates that the test mode has been initiated by typing the system cue TEST at the beginning of the next line. You type your first test mode subcommand on this line. Subsequent lines may or may not be preceded by the word TEST, depending on what you're doing.

Terminating the Test Mode

You can explicitly terminate the ITF test mode the same way you can explicitly terminate any mode; that is, by an END subcommand or by an attention interruption. If you don't do either, the mode will be automatically terminated when the program being tested has run its course of execution. You can terminate the test mode wherever you can type a subcommand in this mode.

After the test mode is terminated, control returns to the initiating mode; for example, if the test mode was initiated from the edit mode, then control returns to the edit mode. Upon this return, the status of the program is exactly the same as it was when the test mode was initiated.

Test Mode Subcommands

The subcommands of the ITF test mode are AT, OFF, GO, TRACE, NOTRACE, LIST, HELP, and END. All but the last two are used for debugging your program. In addition to these subcommands, BASIC assignment statements can be entered, but no other BASIC statements are allowed.

Briefly, the AT subcommand specifies points in your program at which execution will be automatically interrupted so that you can enter some subcommands and/or assignment statements. The OFF subcommand nullifies one or more of these "breakpoints" in your program.

The GO subcommand must be used to start program execution. When execution has been interrupted (because a "breakpoint" was reached, for example), GO will resume the execution from the point of interruption.

The TRACE subcommand causes selected variables, branch points, file names, and intrinsic functions in your program to be monitored so that you will be notified of references to them during execution. The NOTRACE subcommand nullifies this monitoring for one or more of the items being traced.

The LIST subcommand causes the values of one or more variables to be displayed immediately at your terminal so that you can inspect them and choose your future course of action.

The assignment statement allows you to actually change the values of arithmetic variables in your program and perhaps alter the flow of execution as a result.

The END subcommand, as we mentioned earlier, terminates the mode. HELP allows you to ask the system for information about the use of any of the ITF test mode subcommands (use of HELP was discussed in the chapter "Getting Started").

Starting and Resuming Execution—GO Subcommand

Program execution in the test mode is not started until you give the GO subcommand. RUN TEST and BASIC TEST merely initiate the test mode, they do not result in any execution. The format of the GO subcommand is simply

```
GO
```

Before you type GO, you can enter any of the debugging subcommands described in this section (except LIST). In fact, you will probably always use at least AT or TRACE before you type GO for the first time.

Once execution has been started, it can be interrupted by any of the methods described in "Interrupting Execution—AT and OFF Subcommands." To resume execution after an interruption, just type GO and execution will take up from where it stopped. During an interruption, you can enter any debugging subcommands and/or assignment statements.

Interrupting Execution—AT and OFF Subcommands

When debugging, it is desirable to be able to interrupt execution at a particular point so you can see the results thus far. After inspecting the results, you should be able to resume the execution from the point of interruption.

In the ITF test mode, there are two ways in which you can interrupt execution:

- by giving an attention interruption
- by setting "breakpoints"

Attention Interruptions

If you interrupt execution by an attention interruption, the point at which execution will stop is not always predictable. You really can't guess where execution may be at any particular moment unless your program is generating terminal output, in which case, you can determine which statement is being executed by the information that is being written at your terminal. By giving an attention interruption during this output, you can be fairly certain that execution will be interrupted at the statement immediately following the one that generated the output, but you can't be absolutely certain.

Setting Breakpoints

If you have set breakpoints in your program you will always know exactly where interruptions will occur. Breakpoints are set by specifying statement numbers in the AT subcommand. When execution reaches a statement whose number has been specified as a breakpoint, the system automatically interrupts execution and notifies you of the statement number reached. The statement in that line is not executed until you resume the execution. For example, the subcommand

```
at 70
```

sets a breakpoint at statement 70. When execution reaches statement 70, it will be interrupted and the following will be printed at your terminal:

```
A00070
```

You can then type any test mode subcommands (including other AT subcommands) and assignment statements. To resume execution, just type GO. A typical sequence looks like this:

```
A00070at 230
A00070trace (x)
A00070go
```

Notice that the A00070 message is printed before each line you type until you resume execution.

You can specify more than one statement number in an AT subcommand. For example,

```
at 40,80,90,50,130
```

sets breakpoints at statements 40, 50, 80, 90, and 130.

To nullify a breakpoint, that is, to remove the designation of breakpoint from a statement, you must use the OFF subcommand. For example,

```
off 80,130
```

nullifies the breakpoints previously set at statements 80 and 130. When execution is subsequently resumed, it will not be interrupted at these statements. All other breakpoints, however, remain in effect.

You needn't specify statement numbers in the OFF subcommand. If you specify just

```
off
```

all breakpoints currently in effect will be nullified.

Consider this example:

```
READY
edit sub basic old
ITF INITIALIZATION PROCEEDING
EDIT
run test
TEST  at 20,60,100
TEST  trace (x,y,z)
TEST  go
      .
      .
      .
A00020off 20
A00020go
      .
      .
      .
A00060go
      .
      .
      .
A00100end
EDIT
```

{ test mode execution; any terminal output being generated may appear here

In the above example, the user initiates the test mode for his program SUB through the edit mode. In the test mode, he enters AT and TRACE subcommands and then starts the execution of SUB by giving the GO subcommand. When the breakpoint at statement 20 is reached, execution is interrupted. The user enters an OFF subcommand to discontinue this breakpoint and then types GO to resume the execution. When execution reaches statement 60, which is another breakpoint, execution is interrupted again, and so on. At breakpoint 100, the user decides that he's seen enough to determine the problem and returns to the edit mode to make some changes to SUB.

Monitoring Program Execution—TRACE and NOTRACE Subcommands

Using the TRACE subcommand of the test mode, you can observe the flow of execution in your program. For example, if a variable in your program is assuming unexpected values, you can "trace" that variable and any other variables that contribute to those values. That is, you can have the system display at your terminal every value that each of these variables is assigned during execution, immediately after each assignment.

In addition to variables, you can trace references to files, intrinsic functions, and branch points (statement numbers referred to in a GOTO, GOSUB, IF . . . THEN/GOTO). Each time a reference to a "traced" item is encountered during execution, the system displays the name of that item and, in most cases, the number of the statement in which the reference appears. When the traced item is a branch point, only the number of that statement is displayed, indicating that it is the next statement to be executed.

The following subcommand establishes traces for variables X and Y and file "AB#":

```
trace (x,y,"ab#")
```

As a result of this subcommand, every assignment to X and Y and every reference to the file named "AB#" will be noted at your terminal as it occurs.

If you want to trace *every* reference to a variable, branch point, file, and intrinsic function in your program, you should specify just

```
trace
```

If you want to trace only branch points, you don't have to specify any statement numbers; all you need is

```
trace (*)
```

Note that when you use the asterisk, you cannot specify anything else in that TRACE subcommand. For example, if you want to trace all branch points and the variables I9 and T4, the following two subcommands are needed:

```
trace (*)
trace (i9,t4)
```

The effect of the TRACE subcommand for each item that can be traced is as follows:

- *Variables:* Every time a traced variable is assigned a value, that variable and its value are displayed at your terminal along with the number of the statement that performed the assignment. For example, this display

```
00030      N=-1.00000E+00
```

  means that variable N has just been assigned a value of −1 in statement 30. Notice that the value of N is given in the E-format (exponential format).
- *Branch Points:* Every time the statement bearing the traced statement number is about to be executed, that number is displayed at your terminal. For example, this display

```
00200
```

  means that the statement bearing number 200 is about to be executed.
- *File Names:* Every time that the traced file name is referred to in a GET, PUT, RESET, or CLOSE statement, you receive a message like this:

```
00080      INP FILE BEING REFERENCED
```

  where 80 is the statement containing the reference and INP is the name of the file being referred to.
- *Intrinsic Functions:* Every time a traced intrinsic function is referred to in the program, a display of this type will be generated

```
00140      SQR B.I.F. BEING REFERENCED
```

  which means that the SQR BASIC intrinsic function is now being used in statement 140.

To discontinue a trace for an item, you use the NOTRACE subcommand. Like TRACE, NOTRACE can be specified with a list of items, an asterisk, or nothing, as in each of the following:

```
notrace (x,sqr)
notrace (*)
notrace
```

In the first case, the traces for X and SQR are discontinued; all other traces remain in effect. In the second case, only the traces for branch points are discontinued. In the third case, all traces are discontinued.

Listing Values—LIST Subcommand

The LIST subcommand gives you an immediate display of the values of one or more variables. If you specify LIST with one or more variables following it in parentheses, then only those variables and their values will be displayed at your terminal. If you specify LIST with nothing following it, *all* variables and their values will be displayed. For example, if A, X, and B$ exist in your program,

```
list (a,x,b$)
```

causes their values to be immediately displayed at your terminal. The display could look like this:

```
A= +2.37562E+12
B$="FLOYD SMITH"
X= -8.70000E-02
```

Notice that the variables are not necessarily listed in the order in which they appear in the LIST subcommand and that values for arithmetic variables are printed in the E-format (exponential format) as they were in the TRACE subcommand.

However, if you specify just

```
list
```

then not only A, X, and B$ will be listed, but all other variables in your program as well.

Note that the LIST subcommand has no meaning until execution has actually started. In other words, don't use LIST before using GO for the first time.

Changing Values of Arithmetic Variables—Assignment Statement

The assignment statement is a powerful tool in the test mode, even though it is limited to the following format:

simple-arithmetic-variable = [+ | −] *numeric-constant*

By using the assignment statement, you can effectively change your program during execution. Such changes are temporary and disappear when the test mode is terminated, if not sooner (e.g., by execution of an assignment statement in your program that resets the value of the variable).

Consider this situation: Your program contains a FOR/NEXT loop, the FOR statement of which is defined as follows:

```
40 for i = 1 to 1000
```

You want to trace the variables used in the loop but you don't want to step through all 1000 iterations. By placing a breakpoint at the statement immediately following the FOR statement (assume it's numbered 50), you can change the value of the loop control variable I to whatever you wish when the breakpoint is reached. For example,

```
TEST  at 50
TEST  go
A00050i = 575
A00050go
```

The first time that the breakpoint is reached, you enter the assignment statement to change the value of I from 1 to 575. You resume execution and then observe the results for that iteration. The program then increments the value of I by 1 in accordance with the specification in the FOR statement and then the breakpoint is reached again. If you want to observe the iteration for I equal to 576, just type GO. If you want to observe the iteration for a different value of I, then type another assignment statement and resume execution. This process is repeated until the value of I exceeds 1000, at which point the loop will terminate.

Note that the placement of the breakpoint in the above example is important. Had it been placed at statement 40 instead of statement 50, the assignment statement would have been ineffective because the moment execution was resumed, the FOR statement would have been executed and variable I would have been immediately reset to 1.

Let's take another example. Assume that the program you're testing contains this statement:

```
100 let x = 2.6
```

You've observed that this value of x is causing wrong results and you want to try some other values for x before you make a permanent change to the program. By setting a breakpoint at statement 110 or following, you can change the value of x to whatever you wish when the breakpoint is reached. For example,

```
TEST  at 110
TEST  go
A00110x = 3.5
A00110go
```

At the breakpoint, you've changed the value of x to 3.5 and then resumed execution so that you can observe the effects of this change on your program. If you're satisfied with the results, you can then go back to the edit mode and make the change permanent.

Note: You can use the assignment statement only during an interruption. You cannot use it before you actually start execution.

Part II. The BASIC Language

BASIC Program Structure

Statement Numbers

Every BASIC statement must have a statement number. The number can be up to five digits in length (in the range 00001-99999). There can be no blanks between the digits, but at least one blank must separate the statement number from the BASIC statement it precedes.

BASIC Statements

BASIC statements are organized according to specific rules. An executable statement specifies a program action (for example, LET X = 5), and a nonexecutable statement provides information necessary for program execution (for example, DATA 1,2.5,6E-7). Blanks may be inserted where desired to improve readability; the system disregards them except in character constants, character strings, and in image specifications for the PRINT USING statement.

Statement Lines

A statement line is composed of a BASIC statement prefaced by a statement number.

```
10 LET X = 2*4 +7/Z
```

Statement Number (points to "10"); BASIC Statement (brace under "LET X = 2*4 +7/Z")

BASIC Programs

A BASIC program is a group of statement lines arranged according to the following general rules:

1. No statement line may occupy more than one print line.
2. A print line may contain only one statement.
3. Program statements will be retained in numerical sequence. They may, however, be entered in any order.
4. Executable and nonexecutable statements may be intermixed.

Elements of BASIC Statements

BASIC Character Set

A BASIC program is written using the characters listed below. Any terminal character not listed is a non-BASIC character and may appear only where specifically noted.

1. *Alphabetic characters* (29 characters): A-Z in upper or lower case, and the alphabetic extenders @, #, $.
2. *Digits:* 0,1,2,3,4,5,6,7,8,9.
3. *Special characters* (24 characters): The special characters supported by ITF:BASIC and their representation on some of the terminals that can be used in the TSO environment are given in Table 2.

Table 2. BASIC Special Characters and Their Representation on Some TSO Terminals [1]

<table>
<tr><th>SPECIAL CHARACTER</th><th>2741 (#9812) CORRESPONDENCE</th><th>2741 (#9571) PTTC/EBCD</th><th>TELETYPE MODELS 33/35</th></tr>
<tr><td>−</td><td>−</td><td>−</td><td>−</td></tr>
<tr><td>+</td><td>+</td><td>+</td><td>+</td></tr>
<tr><td>*</td><td>*</td><td>*</td><td>*</td></tr>
<tr><td>/</td><td>/</td><td>/</td><td>/</td></tr>
<tr><td>&</td><td>&</td><td>&</td><td>&</td></tr>
<tr><td>=</td><td>=</td><td>=</td><td>=</td></tr>
<tr><td>(</td><td>(</td><td>(</td><td>(</td></tr>
<tr><td>)</td><td>)</td><td>)</td><td>)</td></tr>
<tr><td>.</td><td>.</td><td>.</td><td>.</td></tr>
<tr><td>,</td><td>,</td><td>,</td><td>,</td></tr>
<tr><td>;</td><td>;</td><td>;</td><td>;</td></tr>
<tr><td>:</td><td>:</td><td>:</td><td>:</td></tr>
<tr><td>"</td><td>"</td><td>"</td><td>"</td></tr>
<tr><td>?</td><td>?</td><td>?</td><td>?</td></tr>
<tr><td>!</td><td>!</td><td>!</td><td>!</td></tr>
<tr><td>|</td><td>±</td><td>|</td><td>!</td></tr>
<tr><td><</td><td>[</td><td><</td><td><</td></tr>
<tr><td>></td><td>]</td><td>></td><td>></td></tr>
<tr><td>↑</td><td>**</td><td>**</td><td>↑ or **</td></tr>
<tr><td>≤</td><td>[=</td><td><=</td><td><=</td></tr>
<tr><td>≥</td><td>]=</td><td>>=</td><td>>=</td></tr>
<tr><td>=</td><td>[]</td><td><></td><td><></td></tr>
<tr><td>blank</td><td>blank</td><td>blank</td><td>blank</td></tr>
</table>

BASIC Short Form (External Representation) [2]

An integer format (I-format) is used to print integer values. Using the PRINT[3] statement (see "Program Statements"), up to seven decimal digits may be printed

[1] If your terminal is not represented in this list, consult the *TSO Terminals* book (see the preface) to determine how to represent any BASIC special characters that do not appear on your keyboard.

[2] Because of the physical limitations of the computer, certain values cannot be precisely represented internally, (e.g., 1/3). Computations involving those values may result in a slight loss of precision and, as a result, printed results may be inaccurate in the rightmost one or two significant digits (i.e., in the least significant positions). To overcome this problem, try printing fewer significant digits (by using the Image and PRINT USING statements), or, if these least significant digits are important, try using long-form arithmetic for your computations. You'll probably find that a combination of the two gives the most satisfactory results.

[3] The PRINT USING and Image statements (see "Program Statements") may be used to specify more than the PRINT statement limit on the number of digits.

for integers whose absolute value is in the range 10^7-1 to 0. Integer values having more than seven decimal digits are printed using the E-format.

Floating-point numbers, written in what is called E-format (exponential format), are used in the PRINT[1] statement to print a value up to seven decimal digits in length, with a sign and a decimal point, followed by the letter E, and a signed characteristic (or exponent) for numbers whose absolute value is less than 10^{-1} or greater than or equal to 10^7-1. Values having more than seven decimal digits are rounded before they are printed. Rounding occurs as follows: if the eighth digit is 5 or greater, 1 is added to the seventh digit and the excess digits are truncated.

Decimal numbers, written in what is called F-format (fixed-decimal format), are used in the PRINT[1] statement to print a value up to seven decimal digits, with a sign, and a decimal point, for numbers whose absolute value is not covered by the ranges of I- and E-formats given above. Decimal numbers having more than seven decimal digits are rounded before they are printed. Rounding occurs as follows: if the eighth digit is 5 or greater, 1 is added to the seventh digit and the excess digits are truncated.

Short-form results are obtained when SPREC has been specified in the RUN command, the BASIC command, or the RUN subcommand, or when no option is specified.

BASIC Long Form (External Representation)

An integer format (I-format) is used in the PRINT[1] statement to print an integer value up to 15 decimal digits whose absolute value is in the range $10^{15}-1$ to 0. Integer values having more than 15 decimal digits are printed using the E-format.

Floating-point numbers, written in what is called E-format (exponential format), are used in the PRINT[1] statement to print a value up to eleven decimal digits in length, with a sign and a decimal point followed by the letter E, and a signed characteristic (or exponent) for numbers whose absolute value is less than 10^{-1} or greater than or equal to $10^{15}-1$. Values having more than 11 decimal digits are rounded before they are printed. Rounding occurs as follows: if the twelfth digit is 5 or greater, 1 is added to the eleventh digit and the excess digits are truncated.

Decimal numbers, written in what is called F-format (fixed-decimal format), are used in the PRINT[1] statement to print a value up to 15 decimal digits, with a sign, and a decimal point for numbers whose absolute value is not covered by the ranges of I- or E-formats given above. Decimal numbers having more than 15 decimal digits are rounded before they are printed. Rounding occurs as follows: if the sixteenth digit is 5 or greater, 1 is added to the fifteenth digit and the excess digits are truncated.

Long-form results are obtained when LPREC is specified in the RUN command, the BASIC command, or the RUN subcommand.

Identifiers

An identifier is a string of characters that represents a decimal number or a character constant. There are five types of identifiers: numeric constants, internal constants, character constants, variables, and function references.

Numeric Constants

A numeric constant is a string of characters whose value is a decimal number. The defined value cannot be changed throughout program execution. The two general forms of a numeric constant are:

1. *Decimal fixed-point:* one or more decimal digits with an optional decimal point. If no decimal point appears, the point is assumed to be immediately to the right of the rightmost digit. A sign may optionally precede a decimal fixed-point constant.

[1] The PRINT USING and Image statements (see "Program Statements") may be used to specify more than the PRINT statement limit on the number of digits.

2. *Decimal floating-point:* a decimal fixed-point constant followed by the letter E, followed by an optionally signed one- or two-digit decimal integer constant. The entire constant may be preceded by a sign.

If the exponential notation is used, the value of the constant is equal to the number to the left of the "E" multiplied by 10 to the power of the number to the right of the "E."

The magnitude of a numeric constant must be less than $7.2 \times 10^{+75}$ and must be greater than 5.4×10^{-79}. If long-form arithmetic is specified, up to 15 significant digits will be retained after conversion. If short-form arithmetic is specified, only seven significant digits will be retained after conversion.

Internal Constants

An internal constant is a string of characters that represents a constant whose value is predefined by ITF:BASIC. The available internal constants and their values are:

MEANING	NAME	VALUE (IN SHORT FORM)	VALUE (IN LONG FORM)
e	&E	2.718282	2.71828182845904
π	&PI	3.141593	3.14159265358979
$\sqrt{2}$	&SQR2	1.414214	1.41421356237309

Character Constants

A character constant is a character string enclosed by a pair of single or double quotation marks. The general form of a character constant is:

'[*c*...]'
"[*c*...]"

where *c* is *any* character.

If a character string bounded by single quotes is to contain a single quote, two consecutive single quotes must be given for the contained quote. Otherwise, the system would recognize it as the end of a character string. The same is true when a character string bounded by double quotes is to contain a double quote; the contained double quote must appear twice.

All characters have significance within a character constant. The system treats all character constants as 18 characters. A shorter character constant will be padded on the right with blanks; a longer character constant will be truncated on the right. A null character string (two adjacent quote marks) will be treated as 18 blank characters. In PRINT and PRINT USING statements, character constants can be longer or shorter than 18 characters (see "Program Statements").

Variables

A variable is a string of characters that represents a data item whose value is assigned and/or changed during program execution. There are two types of variables: simple and array.

Simple Variables

A simple *arithmetic variable* is named by a single alphabetic character or an alphabetic character followed by a digit. A simple arithmetic variable can only be assigned a decimal number. The initial value of all simple arithmetic variables is zero. The absolute range (or magnitude) for arithmetic variables is 5.4×10^{-79} to $7.2 \times 10^{+75}$.

A simple *character variable* is named by an alphabetic character followed by the dollar sign character, "$." A simple character variable can only be assigned a character value. The initial value of all simple character variables is 18 blank characters.

Array Variables

A BASIC array is an ordered set of data. An array variable represents an array.

An *arithmetic array* is named by a single alphabetic character. An arithmetic array may have one or two dimensions and can only contain members whose value is a decimal number. The initial value of all arithmetic array members is zero.

A *character array* is named by an alphabetic character followed by the dollar sign character, "$." A character array must have one dimension and can only contain members whose values are character constants containing 18 characters. The initial value of all character array members is 18 blank characters.

An array member is referred to by a subscripted array name. A subscript is an arithmetic expression of which the truncated integer value must be greater than zero. The general form of a subscripted array name is:

$a(x_1[,x_2])$

where a is an array name and x_1 and x_2 are expressions.

When referring to an array member, the number of subscripts must equal the number of array dimensions. Also, the final reference value must be within the bounds of the array. Arrays are processed row by row (i.e., using the row/column concept of a two-dimensional array); members are processed in horizontal sequence rather than vertical sequence.

Array Declarations

An array declaration states that an array with a specified name and dimensions should be allocated to the user's program. Arrays may be defined explicitly (by the DIM statement) or implicitly through usage.

An array is implicitly declared by the first reference to one of its members, provided that the specified array has not been previously defined by a DIM statement. The array is declared to have one dimension (10) when the member is referred to by an array variable with one subscript, and two dimensions (10,10) when the member is referred to by an array variable with two subscripts.

Functions

In general, a function is a named arithmetic expression which computes a single value from another arithmetic expression called an argument. The argument of a function is a parenthesized arithmetic expression which represents the numerical value on which the operations specified by the definition of the function are to be performed. Together, a function and its argument are called a *function reference*. For example, SIN(x) is a function reference which is evaluated as the sine of the number of radians represented by the value of the variable named X. Function references may be used anywhere in a BASIC expression that an arithmetic variable, constant, or array reference may be used.

The two types of functions in BASIC are (1) intrinsic functions which are supplied by the language (see section on "Intrinsic Functions") and (2) user-written functions, which are defined by use of the DEF statement (see "Program Statements").

Expressions and Operators

An expression is a representation of a value. A single constant, variable, array member, or function reference is an expression. Constants, variables, and function references may be combined with operators to form an expression. Three forms of expressions are defined for BASIC: scalar, array, and relational. The result of the evaluation of a scalar expression is a single value—a scalar. A scalar expression may be either an arithmetic expression or a character expression. The result of the evaluation of an array expression is a collection of values—an array. A relational expression can be used only in the context of an IF statement and results in a true or false value.

Character Expressions

A character expression may be composed of a character variable, character array member, or a character constant.

Arithmetic Expressions and Operators

An arithmetic expression may be an arithmetic variable, arithmetic array member, numeric constant, internal constant, or function reference, or a series of the above separated by binary operators and parentheses.

There are five binary operators, two unary operators, and the right and left parentheses. The evaluation of an arithmetic expression is performed left to right with the priority of various operators defining the order of evaluation. The priority of operators will be defined later.

The five binary arithmetic operators are:

↑ or **	exponentiation
*	multiplication
/	division
+	addition
−	subtraction

The two unary operators are + and −.

Special cases for the arithmetic operators and the resulting action are as follows:

Exponentiation: A**B or A↑B is defined as A raised to the B power.[1]

1. If A=B=0, an error will occur.
2. If A=0 and B<0, an error will occur.
3. If A<0 and B is not an integer, an error of a "negative number to a fractional power" will occur.
4. If A≠0 and B=0, A**B or A↑B is evaluated as 1.
5. If A=0 and B>0, A**B or A↑B is evaluated as 0.

Multiplication and Addition: A*B and A+B, multiplication and addition, respectively, are both commutative (i.e., A*B=B*A and A+B=B+A), but are not always associative due to low-order rounding errors, i.e., A*(B*C) does not necessarily give the same results as (A*B)*C.

Division: A/B is defined as A divided by B. If B=0, a "division by zero" error will occur.

Subtraction: A−B is defined as A minus B. No special conditions exist.

Unary Operators

The + and − signs may also be used as unary operators. Unary operators may be used in only two situations, as follows:

1. Following a left parenthesis and preceding an arithmetic expression, or
2. As the leftmost character in an entire expression which is not preceded by an operator.

For example:

−A+(−(B**(−2))) is valid
A+−B or B**−2 is invalid

Priority of Arithmetic Operators

The evaluation of an arithmetic expression is performed using the priority of operators as follows:

Operators	Priority
exponentiation	highest
unary + or −	↓
multiplication and division	
addition and subtraction	lowest

Operations at the same level of priority are performed from left to right. The normal priority can be modified by enclosing subexpressions within parentheses. Subexpressions so modified will be evaluated beginning with the innermost set of parentheses.

[1] Exponentiation operations and some intrinsic function computations are performed by ITF through proven mathematical approximations. Occasionally, results may be inaccurate in the least significant positions because of the attendant limitations of the approximation methods. To overcome this problem, try printing fewer significant digits (by using the Image and PRINT USING statements), or, if these least significant digits are important, try using long-form arithmetic. You'll probably find that a combination of the two gives the most satisfactory results.

The priority of evaluation of function references is determined by the fact that their arguments are enclosed within parentheses.

Any expression that requires computing a value that is not mathematically defined or is imaginary will not be evaluated and will cause an execution error.

Array Expressions

Array expressions are composed of operations which are performed on the entire collection of members of an arithmetic array. Arithmetic array expressions consist of unary or binary operands.

A unary array expression may have one of the following forms:

An array itself
ZER zero array function
CON unity array function
IDN identity matrix function
INV inverse matrix function
TRN transpose matrix function

A binary array expression may have one of the following forms:

A+B sum of two matrices
A–B difference of two matrices
A*B product of two matrices
(*s*)*A product of the scalar value *s* and the matrix A

Matrix multiplication, the inverse function, the transpose function, and the identity function are restricted to two-dimensional arrays only.

Relational Expressions

Relational expressions are of the general form:

e1 relational-operator e2

A relational expression is either satisfied (true) or not satisfied (false). The relational operators are binary; their representation on some of the terminals supported by TSO are:

	OPERATOR		
DEFINITION	2741 (#9571) PTTC/EBCD	2741 (#9812) CORRESPONDENCE	TELETYPE MODELS 33/35
equal to	=	=	=
not equal to	<>	[ ]	<>
greater than	>	]	>
less than	<	[	<
greater than or equal to	>=	]=	>=
less than or equal to	<=	[=	<=

Two forms of relational expressions are allowed in BASIC, arithmetic and character. The expressions *e1* and then *e2* are evaluated and their values are compared according to the definition of the relational operator used. The evaluation of the entire relational expression results in the expression being either satisfied (i.e., the condition is "true") or not satisfied (i.e., the condition is "false"). The EBCDIC (Extended Binary Coded Decimal Interchange Code) collating sequence is used to determine whether character relational expressions are true or false. The ITF:BASIC character set and the EBCDIC internal representation (in hexadecimal) for each character are given in Appendix B.

Program Statements

This section gives the BASIC statements in alphabetical order, except for the MAT statements, which are given in the chapter "Array Operations (MAT Statements)." Most statements are accompanied by the following information:

1. *Function:* a short description of what the statement does.
2. *General Format:* the syntax of the statement.
3. *Rules:* rules governing the specification and use of the statement in a BASIC program.
4. *Example:* an illustration of how the statement would look in a BASIC program.

CLOSE Statement [1]

Function

The CLOSE statement causes the input or output files to be deactivated.

General Format

CLOSE *file-reference* [,*file-reference*]...
where *file-reference* is a character constant.[2] The character constant cannot be a null character string (two adjacent quotation marks). The first three characters cannot contain a period, a comma, or a semicolon. A blank cannot precede a non-blank in the first three characters, nor can the first three characters be all blank.

Rules

1. If a file is to be used first as an input file and then as an output file (or vice versa) during program execution, it must be explicitly deactivated by the CLOSE statement between input and output references.
2. If a file is not closed explicitly (by the CLOSE statement) at the end of the program, it is automatically closed by the system.
3. If a specified file is not active, its appearance in the CLOSE statement will be ignored.

Example

```
150 CLOSE 'QF', 'ABF'
```

DATA Statement [3]

Function

The DATA statement is used to supply values for variables named in the READ statement.

General Format

DATA *constant* [,*constant*]...
where *constant* is either a numeric, character, or internal constant.

Rule

Prior to program execution, a data table is constructed which contains the values in the DATA statements in order of appearance. Also, a data table pointer is set to refer to the first item in the data table. DATA statements may be placed anywhere in the program; they are placed in the data table in statement number sequence.

---

[1] See also GET, PUT, and RESET statements.

[2] To be acceptable to the DELETE and RENAME commands, the first three characters of a file name must adhere to the following restrictions: the first character is required and must be alphabetic; the other two characters are optional and, if specified, must be alphameric.

[3] See also READ and RESTORE statements.

Examples

```
10 DATA  50, 35, 72
20 DATA  340, 17.32, 34E-51, 5E4
.
.
.
50 DATA  25, 'AB', 4.0
```

DEF Statement

Function

The DEF statement defines a user function that can be referred to anywhere in the program in which it is contained.

General Format

DEF FN*a*(*v*) = *arithmetic-expression*
where *a* is an alphabetic character and *v* is a simple arithmetic variable known as the "dummy variable."

Rules

1. The function will be evaluated during execution by substituting the value of the argument used in the function reference for all appearances of the dummy variable in the arithmetic expression.
2. The argument supplied in the function reference can be any valid arithmetic expression.
3. A function may be defined anywhere in the program (before or after its use), but must be defined only once.
4. A function can include any combination of other functions, even those defined by other DEF statements. A recursive function reference (a function that refers to itself or to a function that refers back to it) will give unpredictable results.
5. The dummy variable has meaning only in the DEF statement. Consequently, it is possible to have a dummy variable with the same name as an arithmetic variable used elsewhere in the program. The system recognizes each as a unique identifier, and no conflict of names or values will result from this duplicate usage.

Examples

```
 70  DEF FNB(X)  =  5*X**2+27*Y
 80  DEF FNA(X)  =  FNB(X)+X**3
.
.
.
120  LET Z = 2
130  LET Y = 24
140  LET R = FNA(Z)+23
```

DIM Statement

Function

The DIM statement is used to explicitly define the dimensions of an array.

General Format

DIM $a_1(d_{11}[,d_{12}])$ $[,a_2(d_{21}[,d_{22}])$...
where a_i is the name of an array and d_{ij} is a positive integer specifying the dimension bound.

Rules

1. An array name cannot appear in a DIM statement if it has already been implicitly or explicitly declared.
2. Array dimensioning and referencing starts at 1. That is, an array having one dimension (*m*) has *m* members and an array having two dimensions (*m,n*) has *m* times *n* members.
3. Each bound must be less than 256.
4. Prior to usage in a MAT statement, an arithmetic array must have been implicitly defined by usage or explicitly defined by a DIM statement. This means that the statement number of the DIM statement, or that of the first appearance of the subscripted array name, must be lower than that of the MAT satement.

5. Arithmetic arrays can be redimensioned in the following MAT statements (see the chapter "Array Operations" in Part II):

   MAT assignment with CON function
   MAT assignment with IDN function
   MAT assignment with ZER function
   MAT GET
   MAT INPUT
   MAT READ

Example

```
10 DIM Z(5),$(2,3),A(20,30)
```

END Statement

Function

The END statement indicates the logical end of a program, i.e., any statements numerically following END are retained but are ignored during execution. END is optional and, if omitted, is assumed to follow the highest-numbered statement in the program.

General Format

END [*character-string*]

Rule

The optional *character-string* is merely a comment which has no effect on program execution.

Example

```
99 END
```

FOR Statement [1]

Function

The FOR statement initiates a FOR/NEXT loop; it causes repeated execution of the statements that numerically follow, up to and including a matching NEXT statement.

General Format

FOR $v = x_1$ TO x_2[STEP x_3]
where v is a simple arithmetic variable and x_i is an arithmetic expression.

Rules

1. FOR and NEXT statements must be paired and are matched when the same simple arithmetic variable is specified for each of the two statements.
2. The simple arithmetic variable is the loop control variable and assumes values within the bounds defined by the values of the expressions separated by the TO keyword. The increment value is defined by the STEP option; if STEP is omitted an increment of 1 is assumed.
3. For positive increments, a FOR/NEXT loop is completed when the NEXT statement is executed and the value of the loop control variable *exceeds* the value defined by the second arithmetic expression. For negative increments, a FOR/NEXT loop is completed when the NEXT statement is executed and the value of the loop control variable is *less than* the value defined by the second arithmetic expression. If the step is assigned a value which is contradictory to the increment direction implied by the bounds (e.g., FOR X=1 TO 5 STEP −6 or FOR X=1 TO −20), the FOR/NEXT loop will not be performed and the control variable will not be changed; it will retain the value it had before the FOR statement was encountered.
4. The values of the expressions on the right side of the equal sign remain constant throughout execution of the loop. Any change to the variables in these expressions during the loop will not alter the original expression value. Thus, the loop bounds and the increment value never change during the execution of a loop.
5. Throughout execution of the FOR/NEXT loop, the control variable is available for computation. Upon exiting or completing the loop, the control variable will

[1] See also NEXT statement.

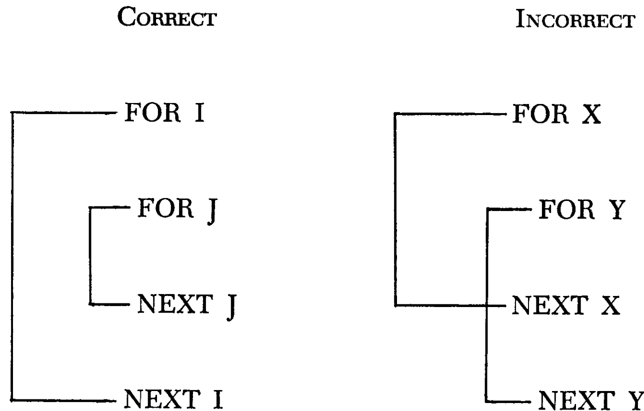
have the value it had at the final iteration, and control is passed to the first logically executable statement following the NEXT statement.

6. If the value of the STEP is zero, the loop is performed an infinite number of times or until the control variable (*v*) is set outside the range.
7. Nesting is permitted only if the internal FOR/NEXT loop is entirely within the external FOR/NEXT loop (as shown in the example below). In ITF:BASIC, FOR/NEXT loops may be nested 15 levels deep (with the outermost FOR/NEXT loop considered to be the first level).
8. Transfer of control into or out of a FOR/NEXT loop is allowable within the constraints that a NEXT statement cannot be executed if its associated FOR statement is inactive. A FOR statement is inactive if it has not been executed, or if the FOR/NEXT loop was previously completed.

Examples

```
20 FOR X = 1 TO 25 STEP 2
•
•
•
70 NEXT X
```

Example of nested loops:



GET Statement [1]

Function

The GET statement causes values to be read from the specified input file, beginning at the current file position, and assigned to the variable references specified in the GET statement.

General Format

GET *file-reference, v*[,*v*]...

where *file-reference* is a character constant.[2] The character constant cannot be a null character string (two adjacent quotation marks). The first three characters cannot contain a period, a comma, or a semicolon. A blank cannot precede a non-blank in the first three characters, nor can the first three characters be all blank. The *v* is either a simple variable or a subscripted reference to an array (it cannot be an unsubscripted reference to an array).

Rules

1. The first appearance of a file reference is an implied declaration and causes the file to be activated for input.
2. Each value read must be of the same type (arithmetic or character) as the corresponding variable reference in the GET statement.

[1] See also CLOSE and RESET statements.

[2] To be acceptable to the DELETE and RENAME commands, the first three characters of a file name must adhere to the following restrictions: the first character is required and must be alphabetic; the other two characters are optional and, if specified, must be alphameric.

3. All arithmetic data is truncated or zero-padded on the right, if necessary, to conform to the kind of arithmetic (long or short) specified for the program in which the values are assigned.
4. Subscripts in GET statements are evaluated as they occur. Thus, an assigned variable in a GET statement may be used subsequently as a subscript in that statement.
5. If a GET statement is executed with insufficient data in the input file, program execution is terminated.
6. A file currently activated as an output file cannot be specified in a GET statement. It must be deactivated by the CLOSE statement.

Examples

```
20 GET 'ITF', X,Y,Z,A(4),A(5)
•
•
•
60 GET 'AF', U,V,W
```

GOSUB Statement[1]

Function

The GOSUB statement causes control to be transferred to the specified statement number.

General Format

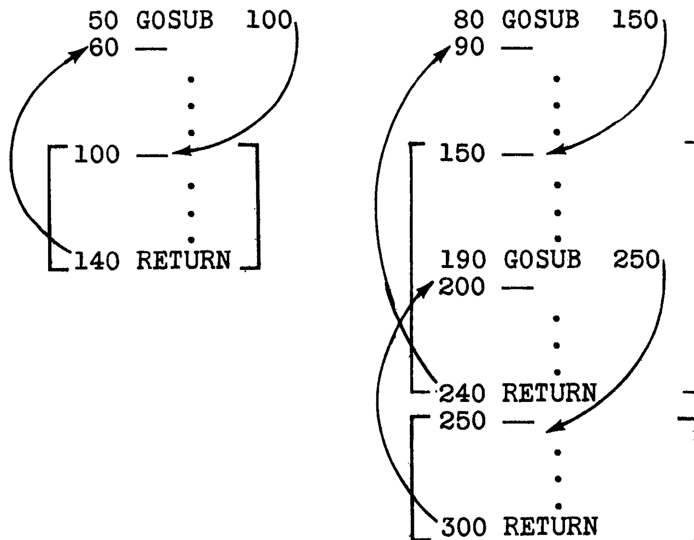
GOSUB *statement-number*

Rules

1. The GOSUB statement sets up a return path such that, when a RETURN statement is executed, control is returned to the next logically executable statement following the last GOSUB statement executed. Execution of a RETURN statement also cancels this return linkage.
2. In ITF:BASIC, there my be no more than 56 active GOSUB statements in a program. If this implementation limit is exceeded, execution of the program is terminated. A GOSUB statement is considered to be active when it has been executed and its associated RETURN statement has not been executed.
3. If the statement branched to is a nonexecutable statement (i.e., DATA, REM, etc.), control is transferred to the first logically executable statement following the specified statement.

Note: GOSUB statements may be used in any manner, but care should be taken to avoid defining recursive GOSUB loops (i.e., a GOSUB into an area of the program that contains a GOSUB leading back to the first GOSUB). This could cause an infinite loop.

Examples



[1] See also RETURN statement.

GOTO Statement

Function

The GOTO statement causes control to be unconditionally transferred to a specific statement (simple GOTO) or to be transferred to one of a set of statement numbers, depending on the value of an expression (computed GOTO).

General Format

1. Computed GOTO
   GOTO s_1[,s_2,...,s_n] ON *arithmetic-expression*
   where s_i is a statement number.
2. Simple GOTO
   GOTO *statement-number*

Rules

1. Computed GOTO:
   This statement causes control to be transferred to the first, second, ..., *n*th statement number if the truncated integer value of the expression is 1,2,...,*n* (respectively) at the time of execution. If the truncated integer value is less than 1 or greater than *n*, control passes to the next logically executable statement. If the statement branched to is a nonexecutable statement (i.e., DATA, REM, etc.), control is passed to the first logically executable statement following the specified statement.
2. Simple GOTO:
   Only one statement number is allowed and control is unconditionally transferred to that statement. If the statement branched to is a nonexecutable statement (i.e., DATA, REM, etc.), control is passed to the first logically executable statement following the specified statement.

Examples

COMPUTED

```
40   GOTO 140,60,34,7,45 ON 3+4/X-Z
```

SIMPLE

```
30   GOTO  645
40   GOTO   29
```

IF Statement

Function

The IF statement tests a relational expression, and if the relation is true, control is transferred to the specified statement number; if the relation is false, the next logically executable statement is executed.

General Format

IF x_1 *op* x_2 {THEN|GOTO} *statement-number*
where x_1 and x_2 are scalar expressions and *op* is a relational operator.

Rules

1. The scalar expressions must both be either arithmetic or character expressions.
2. A character constant containing less than 18 characters will be blank padded to the right prior to the comparison; more than 18 characters will be truncated to the right prior to the comparison.
3. A character constant containing no characters (two adjacent quotes) will be interpreted as 18 blank characters.
4. Comparisons are made following the EBCDIC (Extended Binary Coded Decimal Interchange Code) collating sequence (see Appendix B).
5. If the statement branched to is a nonexecutable statement (i.e., DATA, REM, etc.), control is transferred to the first logically executable statement following the specified statement.

Examples

```
30   IF A(3) > X+2/Z THEN 85
40   IF S1 <> 37.22 GOTO 67
50   IF A+B < C THEN 80
60   IF R$ = A$  GOTO 120
```

Image Statement[1]

Function

The Image statement is used in conjunction with the PRINT USING or MAT PRINT USING statement; it specifies the format that the print line will have.

General Format

:[[*c*]...|*format*]...

where *c* can be any character except "#" and format is a character-, I-, F-, or E-format specification.

Rules

1. An Image statement that consists entirely of a colon (i.e., contains no format specifications) is valid only when the PRINT USING or MAT PRINT USING statement referring to it contains no expressions. Such a reference causes the system to print a blank line. However, if the PRINT USING or MAT PRINT USING statement referring to such an Image statement contains expressions, execution is terminated because there are no format specifications for the values to be printed.
2. The various format specifications are:
   a. *Character-format*—either the I-, F-, or E-formats given below.
   b. *I-format* (integer format)—an optional sign followed by one or more # characters. If the value is negative and no sign is specified, the first # character is assumed to represent the sign.

      Example: [+|−]#[#]...

   c. *F-format* (fixed-decimal format)—an optional sign followed by either:
      (1) No # characters, a decimal point, one or more # characters; or
      (2) One or more # characters, a decimal point, no # characters; or
      (3) One or more # characters, a decimal point, one or more # characters.

      Example: [+|−]{[#]...•#[#]...|#[#]...•[#]...}

      Note that if the value is negative and no sign is specified, the first # character is assumed to represent the sign.
   d. *E-format* (exponential format)—either the I- or F-formats (given above) followed by four ! characters or four | characters. If the value is negative and no sign is specified, the first # character is assumed to represent the sign.

      Example: {*I-format* | *F-format*} {!!!! | ||||}

3. The maximum number of # characters which can be specified for character-, I-, F-, or E-formats is 80.
4. For arithmetic expressions, if more than seven # characters are given in the format specification, and the program is executed using short precision (RUN, RUN SPREC, BASIC, or BASIC SPREC), the printed expression value will fill the entire format specification, but only the first 7 digits can be considered precise. The printed value will be aligned on the decimal point (leading blanks and trailing zeros will be supplied if necessary). Character expression values will be printed left-adjusted. If the expression value is shorter than the specified format specification, blanks will be supplied on the right.
5. The following rules define the start of a format specification:
   a. A # character is encountered and the preceding character is not a # character, decimal point, plus sign, or minus sign.
   b. A plus or minus sign is encountered, which is followed by:
      (1) A # character or
      (2) A decimal point which is followed by a # character.
   c. A decimal point is encountered, which is followed by a # character and:

[1] See also PRINT USING statement.

(1) The preceding character is not a # character, plus sign, or minus sign; or
(2) The preceding character string is an F-format specification.

6. The following rules define the end of a format specification that has been started:
   a. A # character is encountered and:
      (1) The following character is not a # character; or
      (2) The following character is not a decimal point; or
      (3) The following character is a decimal point and a decimal point has already been encountered; or
      (4) The following four consecutive characters are not ! or | characters.
   b. A decimal point is encountered and:
      (1) The following character is not a # character; or
      (2) The following character is another decimal point; or
      (3) The following four consecutive characters are not ! or | characters.
   c. Four consecutive ! or | characters are encountered.

Examples

```
30 :THE ANSWER TO QUESTION # IS +###.####
•
•
•
70 :THE BALANCE FOR ########## IS $#####.##
```

INPUT Statement

Function

The INPUT statement allows the user to assign values to variables from the terminal during execution. When INPUT is encountered, a question mark is printed out, the typing element is moved to the right a few spaces, and execution is interrupted. The values the user types at this time are assigned to the variables given in the INPUT statement.

General Format

INPUT *variable* [,*variable*]...
where *variable* is a simple variable or a subscripted array variable.

Rules

1. When an INPUT statement is encountered in a program, a question mark (?) is printed at the terminal. Data in the form of numeric, character, or internal constants (separated by commas) may then be entered from the terminal.
2. The variables specified assume the values of the data in order of entry; the number of items entered must equal the number of variables in the INPUT statement list. Numeric or internal constants must be entered for arithmetic variables and character constants must be entered for character variables. Subscripted references to arrays are allowed, but unsubscripted array references are not. If the user enters the wrong number of constants or an invalid constant, he will be prompted to re-enter the line (see rule 9, below). However, program execution will be terminated if the constants and variables entered are of different types.
3. All arithmetic data is truncated or zero-padded on the right, if necessary, to conform to the kind of arithmetic (long or short) specified for the program in which the values are assigned.
4. Subscripts in INPUT statements are avaluated as they occur. Thus, if an assigned variable in an INPUT statement is used subsequently as a subscript in that statement the subscript will be evaluated with the new value.
5. Character constants must be bounded by quotation marks. Embedded blanks are significant within a character constant.

6. A character constant containing fewer than 18 characters will be padded on the right with blanks; more than 18 characters will be truncated on the right.
7. A character constant containing no characters (null) will be interpreted as 18 blank characters.
8. When an INPUT statement is executed immediately after a PRINT or MAT PRINT statement in which the final delimiter is a comma or a semicolon, the partially completed print line is printed, the carriage is returned, and the question mark generated by the INPUT statement is printed as the first character on the next print line.
9. The following abbreviated messages are printed at the terminal when the user makes an error entering the values for the variables specified in the INPUT statement. In each case, the user is allowed to correct the error and re-enter the entire data list on the same line that the message is printed.

MESSAGE TEXT	EXPLANATION
NG CON	The magnitude of a numeric constant must be less than $7.2 \times 10^{+75}$ and must be greater than 5.4×10^{-79}. Check to see that your numeric constants are within this range and that you have not forgotten the letter "E" in the exponential format. Also, check the spelling of any internal constant names in your data list.
NG DEL	Constants supplied in response to an INPUT statement must be separated by commas.
NOITEM	You have typed a comma followed by a comma, or you have issued a CR before entering your data.
MSNG '	You have supplied a character constant without enclosing it in single or double quotes.
TOOFEW	You have entered fewer constants than the number of variables specified in the INPUT statement.
EXCESS	You have entered more constants than the number of variables specified in the INPUT statement.

Examples

```
      10 INPUT  X,Y(X),Z(R+3),C1
      .
      .
      .
      90 RUN
?     25,15.5,4,
TOOFEW25,15.5,4,.35

      10  INPUT A$,R
      .
      .
      .
      90 RUN
?     'YES, 20
MSNG ''YES', 20
```

LET Statement

Function

The LET statement evaluates an expression and assigns it to one or more variables. The word LET is optional.

General Format

1. For multiple LET:

   [LET] *variable*[,*variable*]... = *x*

   where *variable* is a simple variable or a subscripted array variable and *x* is an expression.

2. For simple LET:

   [LET] *variable* = *x*

   where *variable* is a simple variable or a subscripted array variable and *x* is an expression.

Rules

1. The variables to the left of the equal sign assume the value of the expression to the right. Subscripts in LET statements are evaluated as they occur. Thus, if an assigned variable in a LET statement is used subsequently as a subscript in that statement, the subscript will be evaluated with the new value.
2. If the expression is arithmetic, all variables to the left of the equal sign must be arithmetic. If it is a character expression or a character constant, all variables to the left of the equal sign must be character variables. Subscripted references to arrays are allowed, but unsubscripted array references are not.
3. A character constant containing fewer than 18 characters will be padded on the right with blanks; more than 18 characters will be truncated on the right.
4. A character constant containing no characters (null) will be interpreted as 18 blank characters.
5. In the test mode, a simple assignment statement can be used once the user has been given control. The form for the simple assignment statement in the test mode is:

   simple-arithmetic-variable = [+|−] *numeric-constant*

Examples

MULTIPLE LET

```
10  LET X(Y+3),Z,Y  =  100.0967
20  X, Y(X),Z,$2  =  -(X+3/E)
30  LET D$,R$,X$  =  J$
```

SIMPLE LET

```
20 LET A1  =  Z(3)/Y(A+4)
30 S1  =  49  +  Z(4)
40 A  =  5
50 LET G$  =  N$
```

NEXT Statement [1]

Function

The NEXT statement marks the physical end of a FOR/NEXT loop.

General Format

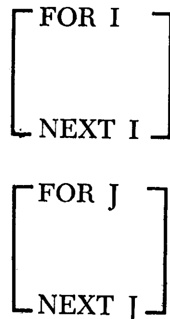
NEXT *simple-arithmetic-variable*

Rules

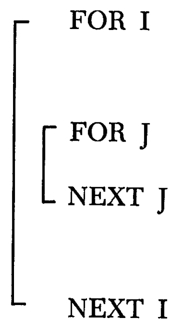
1. This statement terminates a FOR loop. The variable must be the same as the simple arithmetic variable specified in the associated FOR statement.
2. Upon exiting or completing the loop, control is passed to the first logically executable statement following the NEXT statement.

Examples

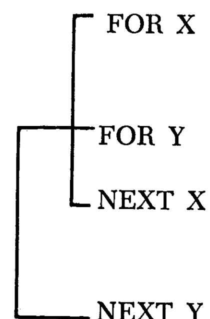
MULTIPLE LOOPS



NESTED LOOPS



INCORRECT NESTING



[1] See also FOR statement.

PAUSE Statement

Function

The PAUSE statement causes program execution to halt and the following message to be printed at the terminal.

PAUSE AT LINE *n*

where *n* is the five digit statement number of the PAUSE statement.

General Format

PAUSE [*character-string*]

Rules

1. The user may resume execution by pressing the carrier return key or by entering any character string followed by a carrier return.
2. The optional *character-string* is a comment which does not affect execution. It appears only when the program is listed.
3. When a PAUSE statement is executed immediately after a PRINT or MAT PRINT statement, the message PAUSE AT LINE *n* is printed on the line below the last line of output from the PRINT or MAT PRINT statement, even if the final delimiter of that statement is a comma or a semicolon.

Example

```
50 PAUSE
```

PRINT Statement

Function

The PRINT statement causes the values of the specified arithmetic and character expressions to be printed at the terminal. The format of a print line is to a large extent controlled by the system; the user can control the density of a line, but the format of the values is standard.

General Format

PRINT [*e*] ['*c*...'[,|;]*e*|['*c*...']{,|;}[*e*]]...['*c*...'|,|;]

where *e* is an arithmetic expression, a simple character variable, or a character array variable; '*c*...' is a character constant; and the comma and semicolon are delimiting characters.

Rules

1. Each data item in the PRINT statement (arithmetic expression or character variable, character constant, or null) is converted to a specified output format and printed at the terminal. The carriage is positioned as specified by the delimiting character or by the data item following the data item being considered. A null delimiter consists of one or more blanks or no characters at all (i.e., one data item directly follows another data item with no intervening space or delimiter). A null delimiter may be used between two data items when one, and only one, of the data items is a character constant.
2. Each line is constructed from two types of print zones: full or packed. Print zones are defined relative to the carriage position at which a data item begins.
   a. If the data item is an arithmetic expression, the size of the packed print zone is determined by the size of the converted field (including the sign, digits, decimal point, and exponent) as follows:

LENGTH OF CONVERTED DATA ITEM	LENGTH OF PACKED PRINT ZONE	EXAMPLE (*x* REPRESENTS A BLANK)
2-4 characters	6 characters	*x*173*xx*
5-7 characters	9 characters	*x*173576*xx*
8-10 characters	12 characters	−45.63927*xxx*
11-13 characters	15 characters	*x*1.735790E−23*xx*
14-17 characters	18 characters	−8922704093115663*x*

   b. If the data item is a character variable or a subscripted character array reference, the size of the packed print zone is 18 characters minus any trailing blanks.
   c. If the data item is a character constant, the size of the packed print zone equals the length of the string enclosed by quotation marks.

3. Each arithmetic data item is converted to output format as follows:
   a. Arithmetic expressions in short-form arithmetic:
      (1) *I-format* (integer format) is used for integers whose absolute value is in the range 10^7-1 to 0. It consists of an optional sign followed by from one to seven digits. Integer values having more than seven digits are printed using the E-format.
      (2) *E-format* (exponential format) is used for floating-point numbers whose absolute value is less than 10^{-1} or greater than or equal to 10^7-1. It consists of an optional sign, seven decimal digits, and a decimal point to represent the mantissa, followed by the letter E, an optional sign, and one or two decimal digits to represent the characteristic. Floating-point numbers having more than seven decimal digits in the mantissa are rounded before they are printed. Rounding occurs as follows: if the eighth digit is 5 or greater, 1 is added to the seventh digit and the excess digits are truncated.
      (3) *F-format* (fixed-decimal format) is used for numbers whose absolute value is not covered by the ranges of the I- and E-formats given above. It consists of an optional sign, seven decimal digits, and a decimal point. Decimal numbers having more than seven decimal digits are rounded as follows: if the eighth digit is 5 or greater, 1 is added to the seventh digit and the excess digits are truncated.
   b. Arithmetic expressions in long-form arithmetic:
      (1) *I-format* (integer format) is used for integers whose absolute value is in the range $10^{15}-1$ to 0. It consists of an optional sign followed by from one to fifteen digits. Integer values having more than fifteen decimal digits are printed using the E-format.
      (2) *E-format* (exponential format) is used for floating-point numbers whose absolute value is less than 10^{-1} or greater than or equal to $10^{15}-1$. It consists of an optional sign, eleven decimal digits, and a decimal point to represent the mantissa, followed by the letter E, an optional sign and one or two decimal digits to represent the characteristic. Floating-point numbers having more than eleven decimal digits in the mantissa are rounded before they are printed. Rounding occurs as follows: if the twelfth digit is 5 or greater, 1 is added to the eleventh digit and the excess digits are truncated.
      (3) *F-format* (fixed-decimal format) is used for numbers whose absolute value is not covered by the ranges of the I- and E-formats given above. It consists of an optional sign, fifteen decimal digits, and a decimal point. Decimal numbers having more than fifteen decimal digits are rounded before they are printed. Rounding occurs as follows: if the sixteenth digit is 5 or greater, 1 is added to the fifteenth digit and the excess digits are truncated.
4. The movements of the carriage at the terminal before, during and after the printing of expression values depend on both the type of expression being printed and the delimiter following it in the PRINT statement. Table 3 shows the variety of carriage actions which are possible.

Examples

Some examples of arithmetic values and the way they are printed are as follows (*x* represents a blank):

GIVEN	PRINTED
123	*x*123
12345678	*x*1.234568E+07
123.4	*x*123.4000
12345.678	*x*12345.68
12345.6745	*x*12345.67

Table 3. Carriage Positions in a PRINT Statement

Data Type	Delimiter	Carriage Position For Printing	Carriage Position After Printing
Arithmetic Expression	Comma	If the line contains sufficient space to accommodate the value, printing will begin at the current carriage position. If not, printing will start at the beginning of the next line.	The carriage will be moved past any remaining spaces in the full print zone. If the end of the line is encountered, the carriage will be moved to the beginning of the next line.
	Semicolon	"	The carriage will be moved past any remaining spaces in the packed print zone. If the end of the line is encountered, the carriage will be moved to the beginning of the next line.
	Null (Not end of statement)	"	The carriage will be left at the print position immediately following the data item.
	Null (End of statement)	"	The carriage will be moved to the beginning of the next line.
Simple Character Variable or Subscripted Character Array Reference	Comma	If at least 18 spaces remain on the line, printing will start at the current carriage position. If less than 18 spaces remain on the line, printing will start at the beginning of the next line.	The carriage will be moved past any remaining spaces in the full print zone. If the end of the line is encountered, the carriage will be moved to the beginning of the next line.
	Semicolon	Printing will start at the current carriage position. If the end of the line is encountered before the data item is exhausted, printing of the remaining characters will begin on the next line.	The carriage will be moved past any remaining spaces in the packed print zone. If the end of the line is encountered, the carriage will be moved to the beginning of the next line.
	Null (Not end of statement)	"	The carriage will be left at the print position immediately following the end of the data item.
	Null (End of statement)	"	The carriage will be moved to the beginning of the next line.
Character Constant	Comma	If at least 18 spaces remain on the line, printing will start at the current carriage position. If less than 18 spaces remain on the line, printing will start at the beginning of the next line. If the end of the line is encountered before the character constant is exhausted, printing of the remaining characters will begin on the next line.	The carriage will be moved past any remaining spaces in the full print zone. If the end of the line is encountered, the carriage will be moved to the beginning of the next line.
	Semicolon or Null (Not end of statement)	Printing will start at the current carriage position. If the end of the line is encountered before the character constant is exhausted, printing of the remaining characters will begin on the next line.	The carriage will be left at the print position immediately following the constant.
	Null (End of statement)	"	The carriage will be moved to the beginning of the next line.
Null	Comma	No printing will occur.	The carriage will be moved to the next full print zone. If the end of the line is encountered, the carriage will be moved to the beginning of the next line.
	Semicolon	"	The carriage will be moved three spaces. If the end of the line is encountered, the carriage will be moved to the beginning of the next line.
	Null	"	If the null data item is the first item on the list, the carriage will be moved to the beginning of the next line. Otherwise, no movement of the carriage will occur.

Following are some examples of character values and the way they are printed:

STATEMENT	PRINTED AS
`10 PRINT 'A','B'`	A ←— 17 blanks —→ B
`20 PRINT 'A';'B'`	AB
`30 LET A$ = 'B'`	
`40 PRINT 'A'A$`	AB
`50 PRINT A$'A',A$;A$`	BA ←— 16 blanks —→ BB
`60 PRINT A$;'A'`	BA
`70 LET A$ = ""`	
`80 PRINT 'A';A$;'A'`	AA

PRINT USING Statement [1]

Function

The PRINT USING statement is used in conjunction with an Image statement to print values. PRINT USING specifies the statement number of the Image statement to be used and the values to be printed; the Image statement specifies the format that the print line will have.

General Format

PRINT USING *statement-number* [,*scalar-reference*]...

where *statement-number* is the statement number of an Image statement and *scalar-reference* is an arithmetic expression or a character expression.

Rules

1. If the statement number does not refer to an Image statement, execution is terminated.

2. Image statements are nonexecutable and may be placed anywhere in a program; they specify the format for single print lines. Character strings appearing in an Image statement are printed exactly as they are entered. Format specifications appearing in an Image statement specify character, integer, fixed-point, or exponential format (see Image statement). Each *scalar-reference* is edited (in order of appearance in the PRINT USING statement) into a corresponding format specification (in order of appearance in the referenced Image statement).

3. If the PRINT USING statement contains at least one *scalar-reference* and no format specification appears in that referenced Image statement, an error occurs. If the number of scalar references in the PRINT USING statement otherwise exceeds the number of format specifications in the Image statement, a carriage return occurs at the end of the Image statement and the Image statement is reused for the remaining scalar references. If the number of scalar references in the PRINT USING statement is less than the number of format specifications in the Image statement, the line is terminated at the first unused format specification.

4. The carriage is repositioned to a new line, if required, before printing the edited line. The carriage is repositioned after printing is completed.

5. Each *scalar-reference* is converted to output format as follows:
   a. The meaning of a scalar reference is extracted from the specified string and edited into the line, replacing all elements in the format specification (including sign, #, decimal point, and |||| or !!!!). If an edited character variable or subscripted character array reference is shorter than the format specification, blank padding occurs on the right. If an edited character variable or subscripted character array reference is longer than the format specification, truncation occurs on the right. A character constant containing no characters results in blank padding of the entire format specification.

[1] See also Image statement.

b. An arithmetic expression is converted in accordance with its format specification as indicated in Table 4.

Table 4. Arithmetic Expression Conversions in a PRINT USING Statement

Sign in Format Specification	Expression Value	Action
+	positive	A plus sign is edited into the print line.
	minus	A minus sign is edited into the print line.
–	positive	A blank is edited into the print line.
	minus	A minus sign is edited into the print line.
none	minus	If the format specification is large enough, the number is printed with a minus sign; otherwise, asterisks are edited into the print line instead of the expression value.

c. An arithmetic expression value is converted according to the type of its format specification, as follows:

I-format: The value of the expression is rounded and converted to an integer as follows: the first fractional digit is examined and if it is 5 or greater, 1 is added to the integer portion. The fractional digits are then truncated.

F-format: The value of the expression is converted to a fixed-point number, rounding the value or extending it with zeros in accordance with the format specification. If the excess digits are fractional, rounding occurs as follows: the first excess fractional digit is examined and if it is 5 or greater, 1 is added to the preceding decimal place. The excess fractional digits are then truncated. If all of the significant digits of the value are excess relative to the format specification, and if the first digit is not the first of the excess digits, then asterisks are printed (see rule 5d below).

E-format: The value of the expression is converted to a floating-point number, rounding the value or extending it with zeros in accordance with the format specification. If the excess digits are fractional, rounding occurs as follows: the first excess fractional digit is examined and if it is 5 or greater, 1 is added to the preceding decimal place. The excess fractional digits are then truncated.

d. If the length of the arithmetic expression value is less than or equal to the length of the format specification, the expression value is edited, right-justified, into the line. If the length of the expression value is greater than the length of the format specification, asterisks are edited into the line instead of the expression value.

Examples

```
30 PRINT USING 40, a,b
40 :RATE OF LOSS ####   EQUALS   ####.## POUNDS

RATE OF LOSS   342   EQUALS     42.04 POUNDS
                ↑                 ↑
              value             value
              of A              of B
```

Format Specification	Arithmetic Value	Printed Form (*x* represents a blank)
###	123	123
###	12	*x*12
###	1.23	*xx*1
##.##	123	*****
##.##	1.23	*x*1.23
##.##	1.23456	*x*1.23
##.##	.123	*x*0.12
##.##	12.345	12.35
###!!!!	123	123E+00
###!!!!	12.3	123E−01
###!!!!	.1234	123E−03
##.##!!!!	123	12.30E+01
##.##!!!!	1.23	*x*1.23E+00
##.##!!!!	.1234	12.34E−02
##.##!!!!	1234	12.34E+02

PUT Statement [1]

Function

The PUT statement causes values to be placed in the specified file.

General Format

PUT *file-reference,x*[,*x*]...

where *file-reference* is a character constant.[2] The character constant cannot be a null character string (two adjacent quotation marks). The first three characters cannot contain a period, a comma, or a semicolon. A blank cannot precede a non-blank in the first three characters, nor can the first three characters be all blank. The *x* can be an arithmetic expression, or a character expression (it cannot be an unsubscripted array reference).

Rules

1. The file created can be used in a GET statement in subsequent execution of another program, or, if deactivated (see CLOSE statement), it can be used in a subsequent GET statement in the same program.
2. The first appearance of a file name is an implied declaration and causes the file to be activated.
3. If the size of the output file is exceeded, program execution is terminated.
4. A file currently activated as an input file cannot be specified in a PUT statement.

Examples

```
30  PUT 'ABF',Z3, 5*X-7, A, C, F3, 9.005
40  PUT 'ABF', A, D$, F4, G$
```

READ Statement [3]

Function

The READ statement specifies variables which are assigned values supplied in DATA statements.

General Format

READ *variable-reference*[,*variable-reference*]...

where *variable-reference* is a simple variable or a subscripted reference to an array (it cannot be an unsubscripted array reference).

---

[1] See also CLOSE and RESET statements.

[2] To be acceptable to the DELETE and RENAME commands, the first three characters of a file name must adhere to the following restrictions: the first character is required and must be alphabetic; the other two characters are optional and, if specified, must be alphameric.

[3] See also DATA and RESTORE statements.

Rules

1. The variable references specified are assigned successive values from the data table beginning at the current position and the data table pointer is updated accordingly. Subscripts in READ statements are evaluated as they occur. Thus, as assigned variable reference in a READ statement may be used subsequently as a subscript in that statement.
2. Arithmetic variables must correspond with arithmetic data and character variables must correspond with character constants.
3. All arithmetic data is truncated or zero-padded on the right, if necessary, to conform to the kind of arithmetic (long or short) specified for the program in which the values are assigned.
4. If a READ statement is executed with insufficient data in the data table, program execution will be terminated.
5. If a READ statement is executed and no DATA statement exists, program execution will be terminated.

Examples

```
10  READ A,B,C
20  READ A(3),Z1,A9
```

REM Statement

Function

The REM statement adds a comment to a program listing; in no way does it affect program execution.

General Format

REM [*character-string*]

Example

```
10 REM THIS PROGRAM DETERMINES THE COST PER UNIT
```

RESET Statement [1]

Function

The RESET statement causes the specified file(s) to be repositioned to the beginning. A subsequent GET or PUT statement will refer to the first item in the file.

General Format

RESET *file-reference*[,*file-reference*]...

where *file-reference* is a character constant.[2] The character constant cannot be a null character string (two adjacent quotation marks). The first three characters cannot contain a period, a comma, or a semicolon. A blank cannot precede a non-blank in the first three characters, nor can the first three characters be all blank.

Rule

If a specified file is not active, its appearance in the RESET statement is ignored.

Example

```
40  RESET 'ABF'
```

RESTORE Statement [3]

Function

The RESTORE statement causes the data table pointer to be repositioned to the first item in the data table, which corresponds to the first item in the first DATA statement in the program. The next READ statement will begin reading at this item.

General Format

RESTORE [*character-string*]

Rules

1. The optional *character-string* is a comment which does not affect the execution of the statement.
2. If no DATA statement(s) exist, the RESTORE statement has no effect on program execution.

Example

```
20 RESTORE
```

---

[1] See also GET and PUT statements.

[2] To be acceptable to the DELETE and RENAME commands, the first three characters of a file name must adhere to the following restrictions: the first character is required and must be alphabetic; the other two characters are optional and, if specified, must be alphameric.

[3] See also DATA and READ statements.

RETURN Statement [1]

Function

The RETURN statement is used in conjunction with the GOSUB statement; it causes control to be transferred back to the next logically executable statement following the last active GOSUB statement.

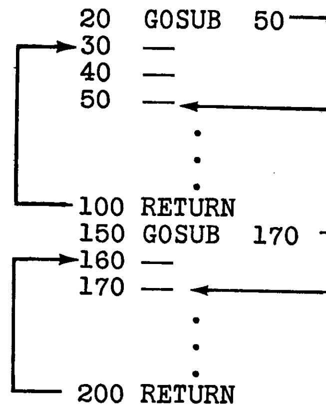
General Format

RETURN [*character-string*]

Rules

1. More than one GOSUB statement may be executed before a RETURN statement is executed, but when a RETURN statement is executed, there must be at least one active GOSUB or program execution will be terminated.
2. The optional *character-string* is a comment which does not affect the execution of the statement. It appears only in the program listing.

Example



STOP Statement

Function

The STOP statement terminates program execution.

General Format

STOP [*character-string*]

Rule

The optional *character-string* is a comment which does not affect execution. It appears only when the program is listed.

Examples

```
30 STOP

80 STOP THIS IS THE END OF THE PROGRAM
```

[1] See also GOSUB statement.

Array Operations (MAT Statements)

This section presents the BASIC MAT statements in alphabetical order (with the exception of the simple MAT assignment, which appears first). Most statements and functions are accompanied by the following information:

1. *Function:* a short description of what the statement does.
2. *General Format:* the syntax of the statement.
3. *Rules:* rules governing the specification and use of the statement in a BASIC program.
4. *Example:* an illustration of how the statement would look in a BASIC program.

Prior to usage in a MAT statement, an array must have been implicitly defined by usage or explicitly defined by a DIM statement (see "Program Statements"). This means that the statement number of the MAT statement must be higher than that of the first usage of the subscripted array name (implicit declaration) or the DIM statement (explicit declaration). Subsequently, an array may be redimensioned by appending one or two subscripts (corresponding to the original number of dimensions), enclosed in parentheses and separated by a comma, to the following MAT statements:

MAT assignment with CON function
MAT assignment with IDN function
MAT assignment with ZER function
MAT GET
MAT INPUT
MAT READ

If redimensioning exceeds the original number of members or changes the original number of dimensions, program execution is terminated. The currently-defined dimensions are observed when executing a MAT statement.

MAT Assignment (Simple)

Function

This statement assigns the members of one array to another array.

General Format

MAT *name-1* = *name-2*
where *name* is the name of an array.

Rule

If arrays specified by *name-1* and *name-2* do not have identical dimensions, program execution is terminated.

Example

```
DIM A(15),B(15)
MAT A = B
```

MAT Assignment (Addition and Subtraction)

Function

This statement assigns the sum or difference of the members of two arrays to the members of a third array.

General Format

MAT *name-1* = *name-2* {+|−} *name-3*
where *name* is the name of an array.

Rule

If the specified arrays do not have identical dimensions, program execution is terminated.

Examples

```
DIM X(2,2),Y(2,2),Z(2,2)
MAT X = Y + Z
```

$$X = \overset{Y}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} + \overset{Z}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}$$

$$X \text{ is } \begin{bmatrix} a+e & b+f \\ c+g & d+h \end{bmatrix}$$

```
MAT X = Y - Z
```

$$X = \overset{Y}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} - \overset{Z}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}$$

$$X \text{ is } \begin{bmatrix} a-e & b-f \\ c-g & d-h \end{bmatrix}$$

MAT Assignment (CON Function)

Function This statement assigns the value one (1) to all members of the specified array.

General Format MAT *name* = CON $[(x_1[x_2])]$
where x is an arithmetic expression and *name* is the name of an array.

Rule The arithmetic expressions (if present) specify redimensioning. They must be valid subscripts. That is, their truncated integer values must be greater than zero.

Examples

```
20  DIM X(4,5)
30  MAT X = CON(3,3)
```

$$X \text{ is } \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

```
60  DIM Y(3,3)
70  MAT Y = CON(4,2)
```

$$Y \text{ is } \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}$$

MAT Assignment (IDN Function)

Function This statement causes the specified array to assume the form of an identity matrix.

General Format

MAT *name* = IDN[(x_1,x_2)]
where x is an arithmetic expression and *name* is the name of a two-dimensional array.

Rules

1. The array must be square (i.e., the values of x_1 and x_2 must be equal) or program execution is terminated.
2. The arithmetic expressions (if present) specify redimensioning. They must be valid subscripts. That is, their truncated integer values must be greater than zero.

Examples

```
DIM X(5,5)
MAT X = IDN(4,4)
```

$$X \text{ is } \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
MAT X = IDN(2,2)
```

$$X \text{ is } \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

MAT Assignment (Inversion)

Function

This statement causes an array to be assigned the mathematical matrix inverse of another array.

General Format

MAT *name-1* = INV (*name-2*)
where each *name* is the name of a two-dimensional array.

Rules

1. The array specified by *name-1* cannot be the same as the array specified by *name-2*.
2. If the arrays are not square (each having two dimensions with identical bounds), program execution is terminated.
3. For the square array A(*m,m*), the inverse of A (if it exists) is B(*m,m*) such that A * B = B * A = I where I is an identity matrix.

   Not every two-dimensional array has an inverse; the inverse of array A exists if DET(A)≠0. If you use INV and the inverse does not exist, execution is terminated. Therefore, it is a good practice to use the DET function (see section on "Intrinsic Functions") to verify that an inverse exists before attempting to use this statement.
4. The accuracy of the matrix inversion function cannot be predicted because it is dependent on the characteristics of the input data and on the size of the problem. The user must be aware of the limitations dictated by numerical analysis considerations. It cannot be assumed that the results are accurate simply because execution of the matrix inversion function is completed.

Example

Consider the following statement where Y is $\begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$:

```
MAT X = INV(Y)
```

The inverse of Y is $\begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix}$ and it is assigned to X. The multiplication of X and Y will thus give a two-by-two identity matrix. That is:

$$\begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix} * \begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

MAT Assignment (Multiplication)

Function

This statement performs the mathematical matrix multiplication of two arrays and assigns the product to a third.

General Format

MAT *name-1* = *name-2* * *name-3*
where each *name* is the name of a two-dimensional array.

Rules

1. The array specified by *name-1* cannot be the same as either array specified by *name-2* or *name-3*.
2. The number of columns of the array specified by *name-2* must equal the number of rows of the array specified by *name-3*. Also, the number of rows of the array specified by *name-1* must equal the number of rows of the array specified by *name-2*. The number of columns in the array specified by *name-1* must equal the number of columns in the array specified by *name-3*. If any of these is not the case, program execution is terminated.

Example

```
MAT Z = X * Y
```

$$Z = \overset{X}{\begin{bmatrix} A & B \\ C & D \end{bmatrix}} * \overset{Y}{\begin{bmatrix} E & F \\ G & H \end{bmatrix}}$$

$$Z \text{ is } \begin{bmatrix} A\times E+B\times G & A\times F+B\times H \\ C\times E+D\times G & C\times F+D\times H \end{bmatrix}$$

MAT Assignment (Scalar Multiplication)

Function

This statement causes one array to be multiplied by an expression and then it assigns the result to the corresponding members of a second array.

General Format

MAT *name-1* = (x) * *name-2*
where x is an arithmetic expression and each *name* is the name of an array.

Rules

1. The expression is evaluated prior to any scalar multiplication.
2. If the arrays do not have identical dimensions, program execution is terminated.

Example

```
MAT Z = (4) * X
```

$$Z = 4\times \begin{bmatrix} A & B \\ C & D \end{bmatrix} \quad Z \text{ is } \begin{bmatrix} 4\times A & 4\times B \\ 4\times C & 4\times D \end{bmatrix}$$

MAT Assignment (Transpose)

Function

This statement causes one array to be replaced by the matrix transpose of another array.

General Format

MAT *name-1* = TRN (*name-2*)
where each *name* is the name of a two-dimensional array.

Rules

1. The array specified by *name-1* cannot be the same as the array specified by *name-2*.
2. Both arrays must have two dimensions and the number of rows in the array specified by *name-1* must equal the number of columns in the array specified by *name-2*. Similarly, the number of rows in *name-2* must equal the number of columns in *name-1*. If any of these is not the case, program execution is terminated.

Example

```
MAT X = TRN (Y)
```

if Y is $\begin{bmatrix} A & B \\ C & D \end{bmatrix}$

then X is $\begin{bmatrix} A & C \\ B & D \end{bmatrix}$

Similarly, if Y is $\begin{bmatrix} A & B & C \\ D & E & F \end{bmatrix}$

then X is $\begin{bmatrix} A & D \\ B & E \\ C & F \end{bmatrix}$

MAT Assignment (ZER Function)

Function

This statement assigns the value zero (0) to all members of the specified array.

General Format

MAT *name* = ZER [(x_1[,x_2])]
where *name* is the name of an array and *x* is an arithmetic expression.

Rule

The arithmetic expressions (if present) specify redimensioning. They must be valid subscripts. That is, their truncated integer values must be greater than zero.

Example

```
MAT X = ZER(3,3)
```

$$X = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

MAT GET Statement [1]

Function

This statement allows arithmetic data to be read into the specific arrays without referring to each member individually.

[1] See also CLOSE and RESET statements.

General Format

MAT GET *file-reference, n-1* $[(x_1[,x_2])]$ $[,n\text{-}2[(x_3[,x_4])]]$...

where *file-reference* is a character constant.[1] The character constant cannot be a null character string (two adjacent quotation marks). The first three characters cannot contain a period, a comma, or a semicolon. A blank cannot precede a non-blank in the first three characters, nor can the first three characters be all blank. The *n* is the name of an array, and x_i is an arithmetic expression.

Rules

1. The members are read in row major order from the input file, beginning at the current file position. Each member read must be arithmetic.
2. All arithmetic data is truncated or zero-padded on the right, if necessary, to conform to the kind of arithmetic (long or short) specified for the program in which the values are assigned.
3. If the input file is exhausted before a specified array is filled, program execution is terminated.
4. The arithmetic expressions (if present) specify redimensioning. They must be valid subscripts. That is, their truncated integer values must be greater than zero.
5. A file is activated for input by the first execution of a MAT GET or GET statement. A file is deactivated by the CLOSE statement, or at the end of execution of a program.
6. A file currently activated as an output file cannot be specified in a MAT GET statement. It must first be deactivated by the CLOSE statement.

Example

```
50  MAT GET  'IF',A(10),Z(2,4)
```

MAT INPUT Statement

Function

The MAT INPUT statement allows the user to assign values from the terminal during execution to members of an arithmetic array without specifying each array member individually. The MAT INPUT statement may also be used to re-dimension arithmetic arrays. When MAT INPUT is encountered, a question mark is printed at the terminal, the typing element is moved to the right a few spaces, and execution is interrupted. The values the user types at this time are assigned, row by row, to the arrays specified in the MAT INPUT statement.

General Format

MAT INPUT *name-1* $[(x_1[,x_2])]$ $[name\text{-}2\ [(x_3[,x_4])]]$...

where *name-1* is the name of a one- or two-dimensional arithmetic array and x_i is an arithmetic expression.

Rules

1. When a MAT INPUT statement is encountered in a program a '?' is printed at the terminal. The user then enters the arithmetic values for the first row of the first array, followed by a carriage return. Data for each subsequent row of that array is requested by the system with the printing of two question marks. Data for the first row of subsequent arrays in that MAT INPUT statement is requested by a single question mark (see example). The final entry for each row of each array must be followed by a carriage return to signify end of row. All data items entered must be separated by commas.
2. If a line is full and input data remains to be entered for the same row, the last value entered on that line must be followed by a comma before the carriage is returned to continue entering values for that row.
3. All data values entered must be arithmetic or the user is requested to re-enter the values for that row.
4. If the number of values entered for a row does not equal the number of members in the corresponding row of the array, the system will indicate (by

[1] To be acceptable to the DELETE and RENAME commands, the first three characters of a file name must adhere to the following restrictions: the first character is required and must be alphabetic; the other two characters are optional and, if specified, must be alphameric.

a message printed at the terminal) that something is wrong. The user can then re-enter the data making the necessary corrections.

5. All arithmetic data is truncated or zero-padded on the right, if necessary, to conform to the form of arithmetic (long or short) specified for the program in which the values are assigned.
6. The arithmetic expressions (if present) specify redimensioning. They must be valid subscripts. That is, their truncated integer values must be greater than zero.
7. The following abbreviated messages are printed at the terminal when the user makes an error entering the values for the arrays specified in the MAT INPUT statement. In each case, the user is allowed to correct the error and re-enter the entire data list on the same line that the message is printed.

MESSAGE TEXT	EXPLANATION
NG CON	The magnitude of a numeric constant must be less than $7.2 \times 10^{+75}$ and must be greater than 5.4×10^{-79}. Check to see that your numeric constants are within this range and that you have not forgotten the letter "E" in the exponential format. Also, check the spelling of any internal constant names in your data list.
NG DEL	Constants supplied in response to the MAT INPUT statement must be separated by commas.
NG TYP	Only numeric and internal constants are permitted in the data list supplied for MAT INPUT statements.
NOITEM	You have typed a comma followed by a comma, or you have issued a CR before entering your data.
TOOFEW	You have entered fewer constants for a row than the number of members contained in a row of the specified array.
EXCESS	You have entered more constants for a row than the number of members contained in a row of the specified array.

8. When a MAT INPUT statement is executed immediately after a PRINT or MAT PRINT statement in which the final delimiter is a comma or a semicolon, the partially completed print line is printed, the carriage is returned, and the question mark generated by the MAT INPUT statement is printed as the first character on the next print line.

Examples

```
        10 DIM A(2,2),B(5),C(6,2)
        20 MAT INPUT A,B,C(3,4)
?       1,2
??      3,4
?       5,6,8,9
TOOFEW5,6,7,8,9
?       10,11,12;13
NG DEL10,11,12,13
??      14,15,16,17
??      18,19,20,21
```

MAT PRINT Statement

Function

This statement causes each member of the specified array(s) to be printed at the terminal.

General Format

MAT PRINT *arithmetic-array* [{,|;}*arithmetic-array*]...[,|;]

where *arithmetic-array* is a one- or two-dimensional arithmetic array and the comma and semicolon are delimiting characters.

Rules

1. Each array in a MAT PRINT statement is printed in row major order (row by row). The first row of each array begins at the start of a new line and is separated from the preceding line by two blank lines. The remaining rows of each array begin at the start of a new line and are separated from the preceding line by a single blank line. After the final or only array has been printed, the carriage will be repositioned. One- or two-dimensional arithmetic arrays may be printed.
2. Each array member is converted to a specified output format and printed. The carriage is repositioned as specified by the delimiting character following the array name. Each line is constructed from two types of print zones, full or packed. Print zones are defined relative to the carriage position at which a data item begins.
   a. The size of the packed print zone is determined by the size of the converted field (including the sign, digits, decimal point, and exponent) as follows:

LENGTH OF CONVERTED DATA ITEM	LENGTH OF PACKED PRINT ZONE	EXAMPLE (*x* REPRESENTS A BLANK)
2-4 characters	6 characters	*x*173*xx*
5-7 characters	9 characters	*x*173576*xx*
8-10 characters	12 characters	−45.63927*xxx*
11-13 characters	15 characters	*x*1.735790E−23*xx*
14-17 characters	18 characters	−8922704093115663*x*

   b. Arithmetic expressions in short-form arithmetic are converted to output format as follows:
      (1) *I-format* (integer format) is used for integers whose absolute value is in the range 10^7-1 to 0. It consists of an optional sign followed by from one to seven digits. Integer values having more than seven digits are printed using the E-format.
      (2) *E-format* (exponential format) is used for floating-point numbers whose absolute value is less than 10^{-1} or greater than or equal to 10^7-1. It consists of an optional sign, seven decimal digits, and a decimal point to represent the mantissa, followed by the letter E, an optional sign, and one or two decimal digits to represent the characteristic. Floating-point numbers having more than seven decimal digits in the mantissa are rounded before they are printed. Rounding occurs as follows: if the eighth digit is 5 or greater, 1 is added to the seventh digit and the excess digits are truncated.
      (3) *F-format* (fixed-decimal format) is used for numbers whose absolute value is not covered by the ranges of the I- and E-formats given above. It consists of an optional sign, seven decimal digits, and a decimal point. Decimal numbers having more than seven decimal digits are rounded as follows: if the eighth digit is 5 or greater, 1 is added to the seventh digit, and the excess digits are truncated.
   c. Arithmetic expressions in long-form arithmetic are converted to output format as follows:
      (1) *I-format* (integer format) is used for integers whose absolute value is in the range $10^{15}-1$ to 0. It consists of an optional sign followed by from one to fifteen digits. Integer values having more than fifteen decimal digits are printed using the E-format.
      (2) *E-format* (exponential format) is used for floating-point numbers whose absolute value is less than 10^{-1} or greater than or equal to $10^{15}-1$. It consists of an optional sign, eleven decimal digits, and a decimal point to represent the mantissa, followed by the letter E, an optional sign, and one or two decimal digits to represent the characteristic. Floating-point

numbers having more than eleven decimal digits in the mantissa are rounded before they are printed. Rounding occurs as follows: if the twelfth digit is 5 or greater, 1 is added to the eleventh digit and the excess digits are truncated.

(3) *F-format* (fixed-decimal format) is used for numbers whose absolute value is not covered by the ranges of the I- and E-formats given above. It consists of an optional sign, fifteen decimal digits, and a decimal point. Decimal numbers having more than fifteen decimal digits are rounded before they are printed. Rounding occurs as follows: if the sixteenth digit is 5 or greater, 1 is added to the fifteenth digit and the excess digits are truncated.

3. The converted array member will be printed at the terminal as follows:
   a. If the line contains sufficient space to accommodate the value, printing will start at the current carriage position.
   b. If the line does not contain sufficient space to accommodate the value, printing will begin at the start of the next line.
4. After the converted member has been printed, the carriage will be positioned as specified by the delimiting character:
   a. If the delimiter is a comma, the carriage will be moved past any remaining spaces in the full print zone; if the end of the line is encountered, the carriage will be moved to the beginning of the next line.
   b. If the delimiter is a semicolon, the carriage will be moved past any remaining spaces in the packed print zone. If the end of the line is encountered, the carriage will be moved to the beginning of the next line.
   c. If the final delimiter is a null, it will be treated as a comma.

Example

In the following example, assume that there are 18 spaces from the beginning of one print zone to the next.

```
10 DIM A(15), X(2,2)
20 MAT READ A
30 DATA 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
40 MAT X = CON
50 MAT PRINT A,X
•
•
•
1       2       3       4       5       6       7

8       9       10      11      12      13      14

15


1       1

1       1
```

MAT PRINT USING Statement [1]

Function

The MAT PRINT USING statement and its associated Image statement allow the BASIC user to have the values of all the members of a specified arithmetic array printed at the terminal in a format of his own choosing, without having to specify each array member individually.

General Format

MAT PRINT USING *s*, *name-1* [[,*name-2*] . . . [*name-n*]]

where *s* is the number of the associated Image statement and *name-i* is the name of an arithmetic array.

[1] See also Image and PRINT USING statements.

Rules

1. Each array referred to in the MAT PRINT USING statement is printed by rows at the terminal according to the format defined by the associated Image statement. (See the PRINT USING and Image statements for information on format specification.) When printed, the first row of each array begins at the start of a new line and is separated from the preceding line by two blank lines. Each succeeding array row begins at the start of a new line and is separated from the preceding row by one blank line. After the last or only array has been printed, the carriage is repositioned to the beginning of the next line.
2. Before being used in a MAT PRINT USING statement, an array must have been previously defined, either implicitly through usage, or explicitly in a DIM statement.
3. One- or two-dimensional arrays may be specified in a MAT PRINT USING statement.
4. If the Image statement specified in the MAT PRINT USING statement does not contain at least one format specification, an error condition results.
5. If the number of members in the array row exceeds the number of format specifications in the associated Image statement, a carriage return occurs at the end of the Image statement and the Image statement is reused to format the remaining members of that row. In this case, if there are additional format specifications in the associated Image statement after all the members of the row have been printed, they are ignored and the first member of the next row will be printed using the first format specification in the Image statement.
6. If the number of members in the array row is less than the number of format specifications in the specified Image statement, the print line is terminated when all the members of that row have been printed. Any additional format specifications will be ignored and the first member of the next row will be printed using the first format specification in the associated Image statement.
7. All values printed must be arithmetic.

Example

```
10 DIM A(4,3)
20 : ### ##.## ##.##!!!!
30 MAT A = CON
40 MAT PRINT USING 20, A
 .
 .
 .
1  1.00 1.00E+00

1  1.00 1.00E+00

1  1.00 1.00E+00

1  1.00 1.00E+00
```

MAT PUT Statement [1]

Function

This statement causes the specified arithmetic arrays to be written on the output file without referring to each member individually.

General Format

MAT PUT *file-reference, name-1*[*,name-2*]...

where *file-reference* is a character constant.[2] The character constant cannot be a null character string (two adjacent quotation marks). The first three characters cannot contain a period, a comma, or a semicolon. A blank cannot precede a

[1] See also CLOSE and RESET statements.

[2] To be acceptable to the DELETE and RENAME commands, the first three characters of a file name must adhere to the following restrictions: the first character is required and must be alphabetic; the other two characters are optional and, if specified, must be alphameric.

non-blank in the first three characters, nor can the first three characters be all blank. Each *name* is the name of an array.

Rules

1. The members are written in row major order into the output file, beginning at the current file position. Each member written must be arithmetic.
2. A file is activated for output by the first execution of a MAT PUT or PUT statement. A file is deactivated by the CLOSE statement, or at the end of execution of a program.
3. A file currently activated as an input file cannot be referenced by a MAT PUT statement. It must first be deactivated by the CLOSE statement.
4. If the size of the output file is exceeded, program execution is terminated.

Example

```
60 MAT PUT 'ITF',A,M,Q
```

MAT READ Statement [1]

Function

This statement is used in conjunction with the DATA statement; it causes arithmetic data to be read into the specified arrays without referring to each member individually.

General Format

MAT READ *name-1*[(x_1[,x_2])][,*name-2*[x_3[,x_4])]]...

where x_i is an arithmetic expression and each *name* is the name of an array.

Rules

1. Beginning at the current position of the data table pointer, the members are read in row major order from the data table, and the pointer is updated accordingly. If the data table is exhausted before a specified array is filled, program execution is terminated. The data pointer may be reset by executing the RESTORE statement.
2. The arithmetic expressions (if present) specify redimensioning. They must be valid subscripts. That is, their truncated integer values must be greater than zero.
3. Each member read must be arithmetic.
4. All arithmetic data is truncated or zero-padded on the right, if necessary, to conform to the kind of arithmetic (long or short) specified for the program in which the values are assigned.
5. If a MAT READ is executed and no DATA statement exists, program execution is terminated.

Example

```
10 DIM $(3,6),B(20),S(10)
70 MAT READ $(2,8),B(12),S
```

[1] See also DATA and RESTORE statements.

Intrinsic Functions

In addition to the five arithmetic operations (i.e., addition, subtraction, multiplication, division, and exponentiation), BASIC supplies twenty-four familiar mathematical functions, such as sine (SIN), cosine (COS), square root (SQR), and natural logarithm (LOG). Table 5 lists these functions in alphabetical order.

The quantity in parentheses immediately following the name of the function is an argument. An *argument* is merely an arithmetic expression representing a value that the function is to act upon. All the ITF:BASIC intrinsic functions require arithmetic expressions as arguments with the exception of DET, which requires as its argument the name of a square arithmetic array.

Table 5. BASIC Intrinsic Functions

FUNCTION	FINDS
ABS(*x*)	Absolute value of *x*
ACS(*x*)	Arccosine (in radians) of *x*
ASN(*x*)	Arcsine (in radians) of *x*
ATN(*x*)	Arctangent (in radians) of *x*
COS(*x*)	Cosine of *x* radians
COT(*x*)	Cotangent of *x* radians
CSC(*x*)	Cosecant of *x* radians
DEG(*x*)	Number of degrees in *x* radians
DET(*x*)	Determinant of an arithmetic array *x* (*x* must be a square array)
EXP(*x*)	Natural exponential of *x*
HCS(*x*)	Hyperbolic cosine of *x* radians
HSN(*x*)	Hyperbolic sine of *x* radians
HTN(*x*)	Hyperbolic tangent of *x* radians
INT(*x*)	Integer part of *x*
LGT (*x*)	Logarithm of *x* to the base 10
LOG(*x*)	Logarithm of *x* to the base *e*
LTW(*x*)	Logarithm of *x* to the base 2
RAD(*x*)	Number of radians in *x* degrees
RND[(*x*)] [1]	Random number between 0 and 1
SEC (*x*)	Secant of *x* radians
SGN(*x*)	Sign of *x* (−1, 0, or 1)
SIN(*x*)	Sine of *x* radians
SQR (*x*)	Square root of *x*
TAN(*x*)	Tangent of *x* radians

[1] Each time RND is called with an argument, the random number generator is initialized to the value of that argument. Subsequent references to RND without using an argument will cause the new number to be generated from the previous one. If RND is called without an argument and there has been no previous initialization, the generator will initialize itself.

Note: Exponentiation operations and some intrinsic function computations are performed by ITF through proven mathematical approximations. Occasionally, results may be inaccurate in the least significant positions because of the attendant limitations of the approximation methods. To overcome this problem, try printing fewer significant digits (by using the Image and PRINT USING statements), or, if these least significant digits are important, try using long-form arithmetic. You'll probably find that a combination of the two gives the most satisfactory results.

Part III. Command Language

Command Language for TSO ITF:BASIC

This section presents the TSO ITF:BASIC commands and subcommands in alphabetical order. The description of each command or subcommand includes:

- *Function:* a short definition of what the command or subcommand does.
- *General Format:* the syntax of the command or subcommand.
- *Rules:* general rules governing the specification and use of the command or subcommand.
- *Abbreviations:* acceptable abbreviations for the command or subcommand and its operands.

Before the descriptions are presented, a summary of the rules for using commands and subcommands is given. These rules should be interpreted with the syntax conventions given in Appendix A in mind, since the command formats follow those conventions.

General Rules of Usage

Syntax of a Command

A command consists of a command name followed, usually, by one or more operands. A command name is typically a familiar English word, often a verb that describes the function of the command. For instance, DELETE deletes a program or a data file. Operands provide the specific information required for the command to perform the requested operation. For example, DELETE requires at least one operand to identify the item to be deleted.

Two types of operands are used with the commands: *positional* and *keyword.* Positional operands follow the command name and precede, or are sometimes associated with, keywords.

Positional Operands

Postional operands are values that follow the command name in a prescribed sequence. The value may be one or more names, symbols, or integers. These operands are shown in lower-case letters.

Keyword Operands

Keywords are specific names or symbols that have a particular meaning to the system. In general, you can include keywords in any order following the positional operands. Keywords are shown in upper-case letters. You must enter them exactly as shown or you can use an acceptable abbreviation. Not all keywords can be abbreviated; those that can are noted in the description of the associated command or subcommand. In general, keywords in commands entered in ITF's test mode cannot be abbreviated.

Delimiters

When you type a command, you should separate the command name from the first operand by one or more blanks. You should separate operands by one or more blanks or commas, as indicated in the command format.

Note: If an operand is parenthesized, you don't have to separate the left parenthesis from the preceding keyword or command name.

Subcommands

The work done by some of the commands is divided into individual operations. Each operation is defined and requested by a subcommand. To make the full range of individual operations available to you, you must enter the command first. You can then enter one of its subcommands to specify the particular in-

dividual operation you want performed. You can continue entering subcommands until you enter the END subcommand. The commands and their subcommands are shown in Tables 6 and 7. Acceptable abbreviations and a brief description of function are also included in these tables.

The syntax of a subcommand is the same as that of a command and the previous discussions of operands and delimiters apply to subcommands as well as commands.

How To Enter a Command or Subcommand

To enter a command, type the command name and any operands required and then give a CR. You enter a subcommand and BASIC statements in the same way. The discussions given in the chapter "Getting Started" (Part I) apply to all the information that you may type.

Certain commands and subcommands can be continued over two or more lines. Excluded are those subcommands that can be entered in the ITF test mode; they must fit on one line.

Continuations

If all of the operands of an eligible command or subcommand do not fit on one line, follow this sequence:

1. Type a hyphen (-) after the last operand on the line.
2. Give a CR.
3. Continue entering operands on the next line. If they do not fit in the second line, repeat from 1.
4. Give a CR.

Table 6. Commands and Their Subcommands, ITF Test Mode Excluded

COMMAND (ABBREVIATION) SUBCOMMAND (ABBREVIATION)	FUNCTION
BASIC	Executes permanent programs; enters ITF test mode.[1]
CONVERT (CONV)	Converts OS ITF:BASIC programs to TSO ITF:BASIC programs.
DELETE (D)	Deletes BASIC programs and data files.
EDIT (E)	Initiates the edit mode.
CHANGE (C)	Corrects parts of statement.
DELETE (D)	Deletes one or more statements.
END	Ends the mode; returns to command mode.
HELP (H)	Requests help about EDIT subcommands.
INPUT (I)	Initiates input phase.
LIST (L)	Displays one or more statements.
RENUM (REN)	Renumbers programs.
RUN (R)	Executes programs; enters ITF test mode.[1]
SAVE (S)	Saves programs (makes a program permanent).
SCAN (SC)	Turns syntax checking on or off.
HELP (H)	Requests help about commands.
LISTCAT (LISTC)	Displays names of your programs and data files.
LOGOFF	Ends your terminal session.
LOGON	Initiates your terminal session.
RENAME (REN)	Renames programs or files.
RUN (R)	Executes programs; enters the ITF test mode.[1]
SEND (SE)	Sends messages to other users or operator.

[1] See Table 7 for the ITF test mode subcommands.

Table 7. ITF Test Mode Subcommands

SUBCOMMAND[1]	FUNCTION
AT	Sets one or more breakpoints.
END	Ends the mode; returns to the mode from which the ITF test mode was initiated.[2]
GO	Starts or resumes execution.
HELP	Requests help about test mode subcommands.
LIST	Displays values of variables.
NOTRACE	Turns off traces.
OFF	Turns off breakpoints.
TRACE	Sets traces for one or more names.

[1] None of the ITF test mode subcommands can be abbreviated.

[2] The ITF test mode is initiated by specifying the TEST option in the BASIC command, the RUN command, or the RUN subcommand of the edit mode.

AT Subcommand [1]

Function

The AT subcommand permits programmer intervention immediately before the execution of a specified statement in the test mode.

General Format

AT *statement-number* [,*statement-number*]...

Rules

1. The AT subcommand is used for debugging programs and can be used only in the test mode.
2. *Statement-number* is the point at which execution is to be suspended and control transferred to the terminal. It is called a *breakpoint.*
3. Every time a breakpoint is reached, execution is suspended and you receive control, until either the breakpoint is nullified by an OFF subcommand or execution of the program is completed.
4. Up to ten different breakpoints can be in effect at any one time for the program being executed.
5. If a statement number specified in an AT subcommand does not exist, the system assumes that it does exist and establishes a breakpoint for it, even though that breakpoint will never be reached. Such breakpoints are included in the total count of breakpoints for your program.
6. Once given control at a breakpoint, you can enter any test mode subcommands and/or simple assignment statements of the form:
   simple-arithmetic-variable = [+|−] *numeric-constant*
7. Execution is resumed from a breakpoint by the GO subcommand.

Abbreviations

None.

BASIC Command

Function

BASIC is used to execute permanent ITF:BASIC programs in the command mode. It can be used to initiate the ITF test mode.

General Format

BASIC *program-name* [TEST|NOTEST] [LMSG|SMSG] [LPREC|SPREC]

Rules

1. *Program-name* is the name of the ITF:BASIC program to be executed or tested. It must be the name of a permanent program (one that was saved in permanent storage).

[1] See also OFF subcommand.

2. TEST specifies that the ITF test mode is to be initiated; NOTEST specifies that no testing is to be performed. If neither is specified, NOTEST is assumed.
3. LMSG specifies that only the long forms of error messages are to be provided for errors detected during execution. SMSG specifies that the short forms of the error messages are to be provided. If you type a question mark after the last short message has been printed, you will receive the expansion. If neither operand is specified, LMSG is assumed.
4. LPREC specifies that calculations are to be performed using long-form arithmetic (fifteen significant digits); SPREC specifies that calculations are to be performed using short-form arithmetic (seven significant digits). If neither is specified, SPREC is assumed.

Abbreviations

None.

CHANGE Subcommand

Function [1]

The CHANGE subcommand is used in the edit mode to modify a sequence of characters in a statement or in a range of statements. You can type the actual change in the CHANGE subcommand itself or you can display part of one or more statements and then type the change at the end of each displayed statement.

General Formats

I. Format for specified changes; no display:
   CHANGE *stmt-1* [*stmt-2*] *delim-string1 delim-string2* [*delim*[ALL]]
II. Format for changes through displays:
   CHANGE *stmt-1* [*stmt-2*] {*delim-string*|*count*}

Rules

1. Rules for Format I are:
   a. *Stmt-1* specifies the number of a statement that you want to change. When used with *stmt-2*, it specifies the first statement of a range of statements.
   b. *Stmt-2* specifies the last statement of a range of statements that you want to change.
   c. *Delim-string1* specifies a special delimiter immediately followed by the sequence of characters that you want to change. The special delimiter can be any printable character other than a number, blank, tab, comma, semicolon, parenthesis, or asterisk. Note that a standard delimiter (e.g., a blank) between the special delimiter and the string will be treated as a character in the string.
   d. *Delim-string2* specifies a special delimiter and the sequence of characters that you want to replace *string1* with. The special delimiter must be the same one that precedes *string1*.
   e. ALL specifies that every occurrence of *string1* within the specified statement or range of statements will be replaced by *string2*. If ALL is not specified, only the first occurrence of *string1* will be replaced by *string2*. The special delimiter that precedes ALL must be the same one used with *string1* and *string2*.
2. Rules for Format II are:
   a. *Stmt-1* specifies the number of the statement that you want to change. The system will display part or all of this statement according to the rest of your specification. When used with *stmt-2*, it specifies the first statement of a range of statements.
   b. *Stmt-2* specifies the last statement of a range of statements. The type of display for this range depends on whether the *delim-string* or *count* operand is specified. If *delim-string* is specified, the range is searched for the

[1] Only a subset of the CHANGE subcommand is presented here. This subset has been selected for its applicability to ITF usage. The full subcommand is presented in the *TSO Command Language Reference* publication (see the preface).

first occurrence of *string* and the statement containing it is displayed at your terminal. This display contains all characters in the statement up to but not including *string*. If *count* is specified, every statement in the range is displayed partially or totally, according to the number of characters specified by *count*.

c. *Delim-string* specifies a special character immediately followed by the sequence of characters you want to change. The special delimiter can be any printable character other than a number, blank, tab, comma, semicolon, parenthesis, or asterisk. Note that a standard delimiter (e.g., a blank) between *delim* and *string* will be treated as a character in *string*.

   When this operand is specified for just one statement, that statement is displayed up to but not including the occurrence of *string* in the statement. You can then type the rest of the statement. When this operand is specified for a range of statements, the first statement found to contain *string* in that range is displayed up to but not including *string*. Again, you can then type the rest of the statement.

d. *Count* specifies the number of characters to be displayed at your terminal, starting at the beginning of each statement indicated. Thus, if one statement is specified, *count* specifies how many characters of that statement are to be displayed; if a range of statements is specified, *count* specifies how many characters of *each* statement are to be displayed. You must complete a statement in order for the system to display the next statement in the range. The statement number associated with a statement must not be included in the count.

e. Whenever a statement is partially displayed, that part of the statement that is not displayed no longer exists; it has been deleted. What you type as the rest of the statement replaces the deleted part of the statement.

Abbreviations

The word CHANGE can be abbreviated as C; the word ALL can be abbreviated as A.

CONVERT Command

Function

The function of CONVERT is to make an OS ITF:BASIC program acceptable to TSO ITF. The use of CONVERT should concern only those users who are moving from OS ITF to TSO ITF and who wish to use their OS ITF programs under TSO ITF.

General Format

CONVERT *dsname1* IN(*dsname2*) [LRECL(*rln*)] [BLOCK(*bln*)] BASIC

where:

dsname1 is the data set name of the OS ITF program to be converted.

IN(*dsname2*) specifies the TSO ITF data set name to be given to the converted program.

LRECL specifies the logical record length, *rln*, of the logical records in *dsname2*. Each logical record will contain one line of the converted program and each line will be padded with blanks to the length given by *rln*. If a line exceeds the length given by *rln*, it is an error and the conversion will be discontinued. If LRECL is omitted, LRECL(80) is assumed.

rln specifies the length of each logical record. It must be an integer from 1 through 128. When specifying this value, be sure to allow 8 bytes for the statement number in each statement.

BLOCK specifies the block (or physical record) size. It means that the logical records in *dsname2* are to be blocked into one or more physical records, the size of which is given by *bln*.

bln is the physical record size. It must be an integer multiple of *rln*, which means that if LRECL is omitted, *bln* must be an integral multiple of 80.

BASIC must be specified to indicate that the OS ITF program being converted is written in BASIC.

Rules

1. IN, LRECL, BLOCK, and BASIC can be specified in any order. The only positional requirement is that *dsname1* follow CONVERT.
2. If LRECL is omitted, LRECL(80) is assumed, regardless of whether or not BLOCK is specified.
3. In specifying a logical record length, you should specify a number large enough to ensure that the longest statement in your program (plus the eight-byte line number) will fit in a logical record. If it doesn't fit, the conversion will not be performed. A logical record length of 128 will ensure that all statements fit.
4. *Dsname1* is the name of your program as known to OS ITF. Under OS ITF your programs are members of a partitioned data set (your private library). The name of that partitioned data set is the same as your OS ITF user identification code. Thus, if you are known to OS ITF as BOBA4, the partitioned data set of which your programs are members is also named BOBA4.

   Members of partitioned data sets are referred to by enclosing the name of the member (i.e., the name under which you saved your program) in parentheses after the data set name. Thus, a program that BOBA4 saved under the name SUB actually has the name BOBA4(SUB) as far as OS ITF is concerned.

   To convert BOBA4(SUB) to TSO ITF format, the CONVERT command would begin as follows:

   ```
   convert 'boba4(sub)' in...
   ```

   Notice that single quotation marks surround the data set name. The quotation marks are required because the data set name does not conform to the TSO data set naming conventions. If the quotation marks were not there, TSO would append qualifiers to both ends of the name and the resulting name would be wrong.
5. The partitioned data set referred to by *dsname1* must be cataloged and known to the operating system. Before you use the CONVERT command, consult your installation maintenance personnel to ensure that this has been done.
6. *Dsname2* is the name that you wish to give to the converted program. It should be constructed according to the TSO data set naming conventions. If it is not so constructed, be sure to enclose the name in single quotation marks so that TSO will not append qualifiers to the name.

   In most cases, you can probably specify the same program name that you used under OS ITF. The only time you shouldn't use the same name is when you already own a program of that name under TSO ITF. For example, to complete the CONVERT command begun in rule 4, SUB could be specified with IN as follows:

   ```
   convert 'boba4(sub)' in(sub) lrecl(128)
   ```

   The name of the converted program, as you know it under TSO ITF, is SUB. If this is a BASIC program and your user identification under TSO is BLUNN, the full name, as TSO knows it, is:

   BLUNN.SUB.BASIC
7. The statements of the converted program will have the same numbers as the statements of the OS ITF program.

Abbreviation

The word CONVERT can be abbreviated as CONV.

DELETE Command (Also a Subcommand of EDIT)

Function

The function of DELETE depends on where it is used. In the command mode, DELETE is used to delete one or more data sets (programs or data files that are in permanent storage); in the edit mode, it deletes one or more statements from the program being edited.

General Format [1]

1. In the command mode:
   DELETE {*data-set*|(*data-set-list*)}
2. In the edit mode:
   DELETE *statement-1* [*statement-2*]

Rules

1. DELETE in the command mode:
   a. *Data-set* is the name of the ITF:BASIC program or data file that you want to delete. *Data-set-list* is used when you want to delete more than one program and/or data file. The data sets named in the list must be separated from each other by one or more blanks or a comma and the entire list must be enclosed in parentheses (as shown in the format).
   b. When deleting a data file, you must observe the following:
      1) In the DELETE command, file names cannot exceed three characters (if your file name is three characters or less, there is no problem; if your file name is longer, make certain that you use only the first three characters in the DELETE command). These three characters must conform to the following TSO file naming conventions:
         a) The first character is required and must be *alphabetic*—any letter (A through Z) or one of the three alphabetic extenders ($, #, and @).
         b) The other two characters are optional; if specified, they must be *alphameric*—any alphabetic character or any digit (0-9). If your file name does not conform to these conventions, it cannot be used in the DELETE command.

         Note: Not all file names permitted by ITF are acceptable for use in the DELETE and RENAME commands. For more information see Part I "Creating and Using Files" under the heading "Naming Files."
      2) The first three characters of the file name (minus the surrounding quotation marks) must be enclosed in parentheses immediately following the word DATA. For example,

```
delete data(in)
```

         deletes the file named "IN" from permanent storage. The following example illustrates how to specify more than one file name in *data-set-list*:

```
delete (cost,data(in),data(out))
```

         This command deletes the program named COST and the two files named "IN" and "OUT" from permanent storage.
2. DELETE in the edit mode:
   a. *Statement-1* specifies the number of the statement that you want to delete from the program being edited. When used with *statement-2*, it specifies the first statement in a range of statements to be deleted.
   b. *Statement-2* specifies the last statement of a range of statements that you want to delete. If both *statement-1* and *statement-2* are specified, all statements between and including *statement-1* and *statement-2* are deleted from the program being edited.

[1] Only a subset of DELETE in these modes is presented here. This subset has been selected for its applicability to ITF usage. The full use of DELETE in these modes is given in the *TSO Command Language Reference publication* (see the preface).

Abbreviation

The word DELETE can be abbreviated as D.

EDIT Command

Function

The EDIT command initiates the edit mode, where you can create, update, execute, and save your ITF:BASIC programs.

General Format [1]

EDIT *program-name* BASIC [OLD|NEW] [SCAN|NOSCAN]

Rules

1. *Program-name* is the name of your ITF:BASIC program. This operand must appear before all other operands.
2. BASIC specifies that the program is an ITF:BASIC program.
3. OLD specifies that the program already exists and that a copy of it is to be retrieved for use in this mode. NEW specifies that the program is about to be created. If neither OLD nor NEW is specified, OLD is assumed.

   Automatic statement-numbering occurs when a program is new; it is as if an INPUT subcommand were given immediately after the EDIT command.
4. SCAN specifies that syntax error notifications are to be given to you as those errors are discovered. NOSCAN specifies that these notifications are to be suppressed until you explicitly request them by a SCAN subcommand or until you execute your program. If neither SCAN nor NOSCAN is specified, NOSCAN is assumed.

Abbreviations

The word EDIT can be abbreviated as E; OLD as O; NEW as NE; SCAN as S; NOSCAN as NOS.

END Subcommand

Function

END is a subcommand of the edit and test modes. It terminates the mode it is used in.

General Format

END

Rules

1. END terminates the edit and test modes.
2. When the edit mode is terminated, control returns to the command mode.
3. When the test mode is terminated, control returns to the command mode or to the edit mode, depending on which of these was used to initiate the test mode.

Abbreviations

None.

GO Subcommand

Function

The GO subcommand starts or resumes execution in the test mode.

General Format

GO

Rules

1. GO is a subcommand of the test mode.
2. The first occurrence of GO in the test mode starts the execution of the program being tested. In all other cases, GO resumes execution when that execution has been interrupted by (a) an attention interruption, or (b) a breakpoint established by the AT subcommand.

Abbreviations

None.

HELP Command and Subcommand

Function

HELP gives you information about ITF or about the function, syntax, and operands of commands and subcommands. This information is displayed at your terminal in response to your request for help.

---

[1] Only a subset of the EDIT command is presented here. This subset has been selected for its applicability to ITF usage. A full discussion of the EDIT command is given in the *TSO Command Language Reference* publication (see the preface).

General Format

HELP [*name* [[FUNCTION] [SYNTAX] [OPERANDS[(*keyword-list*)]] | ALL]]

Rules

1. HELP can be given in any mode. In the command mode, HELP can be used to obtain information about commands or about ITF; in other modes, it can be used to obtain information about any of the subcommands that can be used in those modes.
2. If HELP is given in the command mode, *name* must be a command name or ITF; if HELP is given in one of the other modes, *name* must be the name of a subcommand of that mode.
3. If *name* is ITF, no other operands can be specified. In this case, HELP must be given in the command mode and the resulting display will consist of a brief description of the features of ITF.
4. If *name* is a command or subcommand name, it must precede all other operands. If no operands follow *name*, ALL is assumed.
5. If *name* is omitted, no other operands can be specified. In this case, HELP will display a list of all available commands or subcommands (whichever applies) and their functions. From this list, you can select the command or subcommand most applicable to your needs.
6. FUNCTION specifies that you want information about the purpose and operation of the command or subcommand given by *name*.
7. SYNTAX specifies that you want information about the syntax required to use *name* correctly.
8. OPERANDS [(*keyword-list*)] specifies that you want explanations of all or selected operands of *name*. If *keyword-list* is omitted, all operands will be described. If *keyword-list* is specified, only those keyword operands given in the list will be described. The keywords in the list must be separated from each other by commas or blanks.
9. ALL specifies that you want to see all of the information available about the command or subcommand given by *name*. If *name* is a command that has subcommands, the display will begin with a list of those subcommands.
10. FUNCTION and/or SYNTAX and/or OPERANDS can be specified in any order after *name*. They must be omitted if ALL is specified.

Abbreviations

In the edit and command modes, HELP can be abbreviated as H, FUNCTION as F, SYNTAX as S, OPERANDS as O, and ALL as A; no abbreviations are permitted in the ITF test mode.

INPUT Subcommand

Function

The INPUT subcommand initiates or resumes the input phase of the edit mode.

General Format [1]

INPUT [*statement-number* [*increment*]]

Rules

1. *Statement-number* specifies the statement number to be used for the next statement that you will add or insert into your program. Automatic statement numbering will begin with this number.
2. *Increment* specifies the amount by which you want each succeeding statement number to be increased. If this operand is omitted, it is assumed to be 10.
3. If no operands are specified, the next statement number is determined by incrementing the highest existing statement number in your program by 10.

Abbreviation

The word INPUT can be abbreviated as I.

[1] Only a subset of the INPUT subcommand is given here. This subset has been selected for its applicability to ITF usage. The full subcommand is presented in the *TSO Command Language Reference* publication (see the preface).

LIST Subcommand

Function

The function of the LIST subcommand depends on where it is used. In the edit mode, LIST displays one or more statements of your program at your terminal; in the test mode, LIST displays the values of one or more variables.

General Formats

1. In the edit mode:[1]
   LIST [*stmt-1* [*stmt-2*]]
2. In the test mode:
   LIST [(*variable-list*)]

Rules

1. LIST in the edit mode:
   a. *Stmt-1* specifies the number of the statement that you want displayed at your terminal. When used with *stmt-2* it specifies a range of statements to be displayed.
   b. *Stmt-2* specifies the number of the last statement that you want displayed. When you specify this operand, all statements from *stmt-1* through *stmt-2* are displayed.
   c. If no statement number is given, all statements in your program are displayed.
2. LIST in the test mode.
   a. *Variable-list* specifies one or more variables whose values are to be displayed at your terminal. The list must be enclosed in parentheses and the variables must be separated from each other by commas. The list must not contain subscripted variables.
   b. *If variable-list* is omitted, the values of all variables in the program are displayed.

Abbreviations

The word LIST can be abbreviated as L in the edit mode; it cannot be abbreviated in the test mode.

LISTCAT Command

Function

The LISTCAT command lists the names of all the ITF:BASIC programs, and data files that belong to you.

General Format [2]

LISTCAT [MEMBERS]

Rules

1. MEMBERS is the optional operand that must be specified if you want your listing to include the names of the files you have in DATA, the storage area in which your files are retained. If you omit MEMBERS, only the word DATA will be listed for files; its "members" will not be listed.
2. Names of ITF:BASIC programs are listed in the following form:
   name.BASIC
   where *name* is the name under which the program was saved and BASIC identifies it as an ITF:BASIC program.

Abbreviations

The word LISTCAT can be abbreviated as LISTC; MEMBERS as M.

LOGOFF Command

Function

The LOGOFF command terminates your terminal session.

General Format

LOGOFF

---

[1] Only a subset of LIST in the edit mode is given here. This subset has been selected for its applicability to ITF usage. The full discussion of this subcommand is given in the *TSO Command Language Reference* publication (see the preface).

[2] Only a subset of the LISTCAT command is given here. This subset has been selected for its applicability to ITF usage. The full command is discussed in the *TSO Command Language Reference* publication (see the preface).

Rules

None.

Abbreviations

None.

LOGON Command

Function

The LOGON command initiates your terminal session.

General Format [1]

LOGON *user-id* [PROC(*procedure-name*)]

Rules

1. *User-id* is the user identification (and, optionally, password) assigned to you by your installation. You must specify this every time that you log on. If you are required to supply a password and you omit it, TSO will prompt you for it.
2. PROC(*procedure-name*) specifies your log-on procedure. It is the installation-supplied routine[2] that defines what kind of work you do at the terminal and what system resources you need to do that work. Your installation will give you the *procedure-name* to use in this operand. For purposes of this manual, we are assuming that this name is ITFB.

Abbreviation

The word PROC can be abbreviated as P.

NOTRACE Subcommand [3]

Function

NOTRACE is a test mode subcommand; it nullifies traces established by the TRACE subcommand.

General Format

NOTRACE [(*identifier-list*|*)]

Rules

1. *Identifier-list* specifies one or more variables, intrinsic functions, branch point statement numbers, and file names for which traces currently in effect are to be terminated. The list of identifiers must be enclosed in parentheses and the identifiers themselves must be separated from each other by commas. Traces for identifiers not specified in the list remain in effect.
2. The asterisk specifies that all traces currently in effect for branch points are to be terminated; all other traces remain in effect. The asterisk must be enclosed in parentheses.
3. If neither option is specified, all traces in effect are terminated.

Abbreviations

None.

OFF Subcommand [4]

Function

The OFF subcommand turns off the breakpoints established by the AT subcommand. It can be used only in the test mode.

General Format

OFF [*statement-number*[,*statement-number*]]...

Rules

1. The OFF subcommand can be used only in the test mode.
2. *Statement-number* must be the same number that appears in a preceding AT subcommand. Program execution will no longer be suspended when it reaches this statement.
3. An OFF subcommand given when no AT subcommand is in effect is considered an error condition.

---

[1] This is a subset of the LOGON command, chosen for its applicability to ITF:BASIC usage. The full command is given in the *TSO Command Language Reference* publication (see the preface).

[2] It is assumed that installations will supply log-on procedures for their ITF:BASIC users. If this is not so at your installation, consult both the *TSO Terminal User's Guide* (see the preface) and the *TSO ITF Installation Reference Material* publication (Order Number SC28-6841) for information on defining log-on procedures.

[3] See also TRACE subcommand.

[4] See also AT subcommand.

4. An OFF subcommand with no statement numbers can be used to turn off the breakpoints established by all preceding AT subcommands.

Abbreviations

None.

RENAME Command

Function

RENAME is the command used to rename ITF:BASIC programs and data files that are in permanent storage.

General Format [1]

RENAME *old-name new-name*

Rules

1. *Old-name* is the name that you want to change; *new-name* is the name that will replace *old-name*.
2. When renaming a data file, you must observe the following:
   1) In the RENAME command, file names cannot exceed three characters (if your file name is three characters or less, there is no problem; if your file name is longer, make certain that you use only the first three characters in the RENAME command). These three characters must conform to the following TSO file naming conventions:
      a) The first character is required and must be *alphabetic*—any letter (A through Z) or one of the three alphabetic extenders ($, #, and @).
      b) The other two characters are optional; if specified, they must be *alphameric*—any alphabetic character or any digit (0-9). If your file name does not conform to these conventions, it cannot be used in the RENAME command.

      Note: Not all file names permitted by ITF are acceptable for use in the DELETE and RENAME commands. For more information see Part I "Creating and Using Files" under the heading "Naming Files."
   2) The first three characters of the file name (minus the surrounding quotation marks) must be enclosed in parentheses immediately following the word DATA. For example,

      ```
      rename data(inf) data(fla)
      ```

      causes the file named INF to be renamed FLA.

Abbreviation

The word RENAME can be abbreviated as REN.

RENUM Subcommand

Function

RENUM is the edit mode subcommand that renumbers part or all of the program currently being edited.

General Format

RENUM [*new-number* [*increment* [*old-number*]]]

Rules

1. *New-number* specifies the first statement number to be assigned to the section of the program being renumbered.
2. *Increment* specifies the amount by which each succeeding statement number is to be incremented. If omitted, *increment* is assumed to be 10. You cannot use this operand without *new-number*.
3. *Old-number* specifies the location at which numbering is to begin. If *old-number* is omitted, renumbering will start at the beginning of the program. When *old-number* is used, *new-number* must be greater than the number of the statement that actually precedes *old-number* in the program. You cannot use this operand without using the other two operands of RENUM.

---

[1] This is a subset of RENAME, selected for its applicability to ITF:BASIC usage. The full command is described in the *TSO Command Language Reference* publication (see the preface).

4. If no operands are specified, the entire program is renumbered using the standard increment of 10.
5. All references to statement numbers (e.g., GOTO, GOSUB, MAT PRINT USING, PRINT USING, and IF ... THEN/GOTO) are automatically updated when an ITF:BASIC program is renumbered. In this process, any blanks between the digits of the statement number reference will be eliminated and the number will be squeezed together. For example, assuming that the specified increment was twenty, the statement

   `go to 5 1 0` will appear as `go to 530`

   after renumbering and the statement

   `print usin g 8 0, x` will appear as `print usin g 100, x.`

Abbreviation

The word RENUM can be abbreviated as REN.

RUN Command (Also a Subcommand of EDIT)

Function

RUN is used to execute ITF:BASIC programs and to initiate the ITF test mode for program testing.

General Formats

1. In the command mode:
   RUN *prog-name* BASIC [TEST|NOTEST] [LMSG|SMSG] [LPREC|SPREC]
2. In the edit mode:
   RUN [TEST|NOTEST] [LMSG|SMSG] [LPREC|SPREC]

Rules

1. RUN in the command mode:
   a. *Prog-name* is the name of the ITF:BASIC program to be executed or tested. This program must be permanent—that is it must have been previously saved in permanent storage.
   b. BASIC specifies that the program is an ITF:BASIC program.
   c. TEST specifies that the test mode is to be initiated; NOTEST specifies that no testing is to be performed. If neither is specified, NOTEST is assumed.
   d. LMSG specifies that only the long forms of error messages are to be provided for errors detected during this execution. SMSG specifies that the short forms of error messages are to be provided. If you type a question mark after the last short message has been printed, you will receive the expansions. If neither operand is specified, LMSG is assumed.
   e. LPREC specifies that calculations are to be performed using long-form arithmetic (fifteen significant digits); SPREC specifies that calculations are to be performed using short-form arithmetic (seven significant digits). If neither is specified, SPREC is assumed.
2. RUN in the edit mode:
   a. The program to be executed or tested is the one currently being edited (i.e., the one whose name is specified in the EDIT command).
   b. TEST specifies that the test mode is to be initiated; NOTEST specifies that no testing is to be performed. If neither is specified, NOTEST is assumed.
   c. LMSG specifies that only the long forms of error messages are to be provided for errors detected during this execution. SMSG specifies that the short forms of the error messages are to be provided. If you type a question mark after the last short message has been printed, you will receive the expansions of those messages. If neither operand is specified, LMSG is assumed.
   d. LPREC specifies that calculations are to be performed using long-form arithmetic (fifteen significant digits); SPREC specifies that calculations are to be performed using short-form arithmetic (seven significant digits). If neither is specified, SPREC is assumed.

Abbreviation

The word RUN can be abbreviated as R.

SAVE Subcommand

Function

SAVE is a subcommand of the edit mode; it causes the edited program to be permanently retained.

General Format

SAVE [*name*]

Rules

1. *Name* specifies the name under which the program is to be saved; if it is omitted, the program is saved under the name used in the EDIT command.
2. If *name* is the name of an existing program, that program is replaced with the one currently in the edit mode.

Abbreviation

The word SAVE can be abbreviated as S.

SCAN Subcommand

Function

SCAN is an edit mode subcommand; it is used to obtain syntax error notifications.

General Format [1]

```
      [ stmt-1 [stmt-2] ]
SCAN  [ ON              ]
      [ OFF             ]
```

Rules

1. *Stmt-1* specifies that you want notification of syntax errors in the statement having this number. When used with *stmt-2*, it specifies a range of statements.
2. *Stmt-2* specifies that you want notification of all syntax errors between and including *stmt-1* and *stmt-2*.
3. ON specifies you want to be immediately notified of syntax errors in lines that you enter from this point on.
4. OFF specifies that immediate notifications of syntax errors are to be suppressed.
5. If no operands are specified, you will be notified of all syntax errors existing in your program up to this point.

Note: If you have specified the SCAN operand in your EDIT command, you can later specify SCAN OFF to suppress syntax error notifications. If you have specified the NOSCAN operand in your EDIT command, you can later use the SCAN subcommand to obtain notification of syntax errors in your program. In fact, even if you have specified the SCAN operand in the EDIT command, you can later use the SCAN subcommand to obtain notification of syntax errors that still exist in the program. Essentially, with SCAN ON and SCAN OFF, you can turn notifications "on" or "off" as you please.

Abbreviations

The word SCAN can be abbreviated as SC; OFF as OF.

SEND Command

Function

The SEND command is used to send messages to other terminal users or to the system operator.

General Format

```
SEND 'text' [ USER(ident-list) [NOW|LOGON] ]
            [ OPERATOR [(integer)]         ]
```

Rules

1. *Text* is the message to be transmitted. It must be enclosed in single quotation marks as shown. The *text* must not exceed 115 characters, including blanks.

[1] This is a subset of the SCAN subcommand, chosen for its applicability to ITF:BASIC usage. The full subcommand is described in the *TSO Command Language Reference* publication (see the preface).

2. USER(*ident-list*) specifies that the message is to go to the terminal users given by *ident-list,* which contains one or more user identifications separated by commas or blanks.
3. NOW specifies that you want the message to be sent immediately. If the recipient is not logged on, you will be notified and the message will be deleted. LOGON specifies that you want the message to be sent to the recipient now (if he is currently logged on) or when he logs on (if he is not using the system at this time). If neither option is specified, NOW is assumed.
4. OPERATOR(*integer*) specifies that the message is to go to the system operator identified by *integer*. If *integer* is omitted, it is assumed to be 2.
5. If no options are specified after '*text*', the message is sent to the console operator at the central computer location.

Abbreviations

The word SEND can be abbreviated as SE; USER as U; NOW as N; LOGON as L; OPERATOR as O.

TRACE Subcommand [1]

Function

TRACE is a test mode subcommand; it monitors program execution by keeping track of changes to variables and references to branch points, file names, and intrinsic functions.

General Format

TRACE [(*identifier-list* | *)]

Rules

1. *Identifier-list* specifies one or more variables, branch point statement numbers, file names, and intrinsic function names for which "traces" are to be established. The list must be enclosed in parentheses and the items within it must be separated from each other by commas. Subscripted variables are not permitted in the list.
2. The asterisk specifies that *all* branch points are to be traced. No other items can be specified in the subcommand when * is used. A separate TRACE subcommand is required to trace other types of items, in this case. The asterisk must be enclosed by parentheses, as the format shows.
3. If neither option is specified, all variables, branch points, file names, and intrinsic functions are traced.
4. The display generated by a trace depends on the type of item being traced, as follows:
   a. *Branch points:* the statement number is printed immediately before execution of the statement having that number.
   b. *Intrinsic function and file names:* a reference to one of these is noted by a message of this form:
      nnnnn *name* *type* BEING REFERENCED
      where *nnnnn* is the statement number of the reference, *name* is the item being traced, and *type* is B.I.F. (BASIC intrinsic function) or FILE.
   c. *Variables:* when the value of a traced variable changes, ITF displays the variable, its new value (in E- or exponential format), and the number of the statement in which the change took place.
5. Once TRACE is specified, it remains in effect for the items specified until either execution of the program is complete, or until a NOTRACE subcommand negates part or all of its function.
6. More than one TRACE subcommand can be issued while in the testing environment; the effect is cumulative (that is, once execution is restarted, the items specified in all TRACE subcommands will be monitored as explained above).

[1] See also NOTRACE subcommand.

7. If a variable name is the same as that of a file (except for the quotation marks surrounding the file name), a TRACE command for either will give unpredictable results. For example:

```
TRACE ("A",A)
```

will result in either a trace of the file name or a trace of the variable A, but not both. Similarly, TRACE(A), where a file named "A" and a variable named A exist, is also unpredictable.

Abbreviations

None.

Appendixes

Appendix A. Syntax Conventions

The syntax conventions used to illustrate the general forms in Parts II and III of this document are:

a. Upper-case letters, digits, and special characters represent information that must appear exactly as shown.
b. Lower-case letters represent information that must be supplied by the user.
c. Information contained within brackets [ ] represents an option that can be omitted.
d. The appearance of braces { } indicates that a choice must be made between the items contained in the braces.
e. The appearance of the vertical bar | indicates that a choice must be made between the item to the left of the bar and the item to the right of the bar.
f. An ellipsis (a series of three periods) indicates that the preceding syntactical unit may be used one or more times in succession.
g. A list whose length is variable is specified by the format: $x_1,x_2,...,x_n$. This format indicates that a variable number of items may be specified, but that at least one is required (commas must separate the items).
h. The appearance of one or more items in sequence indicates that the items (or their replacements) should also appear in the specified order.

Appendix B. Collating Sequence of the ITF:BASIC Character Set

Note that both upper and lower case letters of the standard English alphabet are represented internally by the EBCDIC bit configuration of the upper case characters only.

Character	Internal Hexadecimal Representation	Name
	40	Blank
.	4B	Period or decimal point
<	4C	Less than sign
(	4D	Left parenthesis
+	4E	Plus sign
|	4F	Logical "or" sign or vertical bar
&	50	Ampersand
!	5A	Exclamation mark
$	5B	Dollar sign
*	5C	Asterisk or multiply symbol
)	5D	Right parenthesis
;	5E	Semicolon
–	60	Hyphen or minus sign
/	61	Slash or division symbol
,	6B	Comma
>	6E	Greater than sign
?	6F	Question mark
:	7A	Colon
#	7B	Pound or number sign
@	7C	Commercial "at" sign
'	7D	Apostrophe or single quotation mark
=	7E	Equal sign
"	7F	Double quotation mark
↑	8A	Up-arrow or exponentiation sign
≦	8C	Less than or equal to sign
≧	AE	Greater than or equal to sign
≠	BE	Not equal sign
A,a	C1	
B,b	C2	
C,c	C3	
D,d	C4	
E,e	C5	
F,f	C6	
G,g	C7	
H,h	C8	
I,i	C9	
J,j	D1	
K,k	D2	
L,l	D3	
M,m	D4	

Character	Internal Hexadecimal Representation
N,n	D5
O,o	D6
P,p	D7
Q,q	D8
R,r	D9
S,s	E2
T,t	E3
U,u	E4
V,v	E5
W,w	E6
X,x	E7
Y,y	E8
Z,z	E9
0	F0
1	F1
2	F2
3	F3
4	F4
5	F5
6	F6
7	F7
8	F8
9	F9

Appendix C. Attention Interruption Summary

Table 8 summarizes the effect of attention interruptions given in the three modes that pertain to ITF:BASIC.

Note: If the attention key at your terminal is being used for line deletions as well as for attention interruptions, please interpret the table with the following in mind:

> If your pressing of the attention key is interpreted as a line deletion operation, you must press the attention key a second time to achieve an attention interruption. A line deletion occurs when you press the attention key after you have begun typing a line (any line in any mode). The system does not respond to a line deletion with a system cue (or a statement number, whichever the case may be); it just waits for you to retype the line.

Table 8. Attention Interruption Summary

Condition When Attention Is Given		Number of Attentions Given: One	Number of Attentions Given: Two
Edit Mode	Input Phase	Phase is terminated. System types EDIT system cue.	Mode is terminated. System reverts to command mode and types READY system cue.
Edit Mode	During Execution of Program or Subcommand	Execution is terminated and system types EDIT system cue.	Mode is terminated. System reverts to command mode and types READY system cue.
Edit Mode	No Execution in Effect	Mode is terminated. System reverts to command mode and types READY system cue.	
Test Mode	During Execution of Program	Execution is interrupted. It can be resumed by a GO subcommand. System types TEST system cue.	Mode is terminated. System reverts to the initiating mode (command or edit) and types appropriate system cue.
Test Mode	No Execution in Effect	Mode is terminated. System reverts to initiating mode (command or edit) and types appropriate system cue.	
Command Mode	During Execution of Program or Subcommand	Execution is terminated. System remains in command mode and types READY system cue.	
Command Mode	No Execution in Effect	No effect. System remains in command mode but does not type READY.	

Appendix D. File Usage Considerations

As a TSO ITF user, you have your own permanent storage area for your ITF files. Each file you create through ITF:BASIC is made a member of *userid*.DATA (where *userid* is your user identification code). The name of a particular member is the same as the first three characters of the file name you specified in the PUT statement that created it. To refer to a member of *userid*.DATA in a TSO command (e.g., DELETE and RENAME), you must always qualify the member name[1] by the word DATA. For example, an ITF file named OUTFILE is known to TSO as DATA(OUT).

PUT statements always refer to members of *userid*.DATA. GET statements, however, can refer to files that are not members of this data set (more about this later).

In general, *userid*.DATA holds at least 20 files. Up to 40 additional files will fit, depending on the size of each file.

File Maintenance

If *userid*.DATA becomes full and there is no room for additional files, use the DELETE command to delete the files that you no longer want. Note, however, that the space used by the deleted files will not be available for re-use until *userid*.DATA has been "compressed." The "compress" operation may be performed on a regular basis by your installation (ask your system administrator about this). If your installation has the TSO COPY utility (which is part of the separately-orderable TSO Data Utilities program product), you can perform the "compress" operation yourself by this sequence of commands:

```
READY
copy data tempdata
READY
delete data
READY
rename tempdata
```

The COPY command copies the contents of *userid*.DATA into a temporary data set (TEMPDATA in our illustration). This copy operation automatically "compresses" the members of the copied data set and frees up the space that had been taken by "deleted" members. The DELETE command deletes *userid*.DATA and the RENAME command renames TEMPDATA as the new, "compressed", *userid*.DATA.

It is a good practice to periodically do some "housekeeping" on *userid*.DATA (i.e., delete unwanted files). To do this, say, just before you log off for the day, issue a LISTCAT MEMBERS command to examine the contents of *userid*.DATA, and then issue a DELETE command to eliminate the unwanted files. After this, you can perform the "compress" operation yourself (if you have the COPY utility at your installation), as just shown.

Using Files Not in userid.DATA

Unlike PUT statements, GET statements in your ITF:BASIC programs can refer to files that are not members of *userid*.DATA. For example, you may have a data file that

[1] To be acceptable in TSO commands, the first three characters of the file name specified in your GET and PUT statements must adhere to the following restrictions: the first character is required and must be alphabetic; the other two characters are optional and, if specified, must be alphameric.

you created using something other than ITF:BASIC and you want to use this data file as input to one of your ITF:BASIC programs. If the file meets the following requirements, it can be used under ITF:BASIC:

1. It must be sequentially organized with fixed-length unblocked 120-character records.
2. It must be cataloged and known to TSO.
3. It must be allocated in the session in which it is to be used.

To allocate such a file, use this form of the ALLOCATE command.[1]

ALLOCATE DATASET(*dsname*) FILE(*file-name*) OLD

For example, if the name of the data file that you want to use is MYDATA04 and the new name that you want to associate with this data file is DT4, you would give this ALLOCATE command in your terminal session:

```
allocate dataset ('mydata04') file(dt4) old
```

With this ALLOCATE command in effect, DT4 could be used as an input file in ITF:BASIC statements throughout the session.

When choosing *file-name* in the ALLOCATE command, be sure that the name you select doesn't match any of the names that you already have in *userid*.DATA. If you inadvertently specify a matching name, all references to that name in ITF:BASIC statements will be interpreted as references to the corresponding member of *userid*.DATA and not to the intended data file.

---

[1] The ALLOCATE command is described fully in the *TSO Command Language Reference* publication (see the preface).

Appendix E. Differences Between OS ITF and TSO ITF

Users migrating from OS ITF to TSO ITF may find the transition smoother if they understand the differences described below. This appendix *does not* compare TSO's extensive facilities with those of OS ITF; a quick look at their command languages will show the relative strengths of each system. The differences described here fall into three categories:

- terminological (e.g., TSO "command mode" versus OS ITF "control mode")
- visual (i.e., how they differ in their appearance at your terminal)
- functional (i.e., differences in equivalent facilities)

Terminological Differences

Listed below are those terms that differ between OS ITF and TSO ITF. Those OS ITF terms that have no TSO ITF equivalent are so noted.

OS ITF TERM	EQUIVALENT TSO ITF TERM
"command"	"command" or "subcommand" (depending on whether it is entered in the command mode or in one of the other modes)
"common library"	There is no equivalent facility in TSO and, therefore, no equivalent term.
"control mode"	"command mode"
"private library"	"permanent storage" can be considered the equivalent, although there is no private library facility in TSO, per se.
"test submode"	"test mode"
"text collection"	"text" (ITF does not provide a text facility under TSO—TSO provides it)

Visual Differences

OS ITF and TSO ITF look very much the same at your terminal. The only real difference is in the appearance of system cues in the command (control) and edit modes. Under OS ITF, every line that you type in these modes is preceded by the ITF-typed READY or EDIT system cue, whichever applies. Under TSO ITF, READY and EDIT never appear on the line you are typing. When one appears, it is on a line by itself. This difference is particularly evident when you are typing your own statement numbers in the edit mode. Consider the following:

OS ITF	TSO ITF
READY edit xyz basic	READY
EDIT 40 let x = 1	edit xyz basic old scan
EDIT 53 let y = 2	ITF INITIALIZATION PROCEEDING
EDIT 58 goto 80	EDIT
EDIT save	40 let x = 1
EDIT end	53 let y = 2
READY logoff	58 goto 80
	save
	SAVED
	end
	READY
	logoff

As you can see, OS ITF always types EDIT at the beginning of each line when you're editing an old program, but TSO doesn't.

A minor visual difference concerns error messages. Under OS ITF, messages that have no long form are preceded by three asterisks. Under TSO ITF, the absence of a plus sign at the end of an error message indicates that the message has no further levels of information; the appearance of a plus sign means that the message has at least one more level of information, which can be obtained (as in OS ITF) by typing a question mark.

Functional Differences

Equivalent facilities under OS ITF and TSO ITF differ as follows:

- *Program Storage:* OS ITF provides private libraries and a common library. TSO does not assign libraries to its users.
- *ITF Test Mode:* Under TSO, ITF's test mode can be entered through the command mode as well as the edit mode. Under OS ITF, the test mode can be entered through the edit mode only.
- *MERGE Command:* OS ITF provides a MERGE command in the edit mode. Under TSO, MERGE is part of a separately-orderable IBM Program Product called TSO Data Utilities, which may or may not be available at your installation.
- *Text Facility:* OS ITF provides a text facility in the edit mode. Under TSO, ITF has no text facility; it is a TSO feature.
- *Execution of ITF:BASIC Programs:* In addition to the program execution facilities of the edit mode and the test mode (both of which OS ITF has), TSO allows ITF:BASIC programs to be executed in the command mode (via RUN and BASIC commands).
- *Error Messages:* TSO ITF allows you to suppress the short forms of ITF error messages; OS ITF does not.
- *Syntax Checking in Edit Mode:* Under OS ITF, each statement is automatically checked for syntactical correctness as it is entered. If an error is detected, the statement is immediately discarded. Under TSO ITF, syntax checking is performed as statements are being entered, but erroneous statements are not discarded; they remain in the program until they are replaced or deleted. Also, TSO ITF allows you to postpone syntax error messages until you ask for them.
- *EDIT Command Options:* Because TSO has a wider scope than OS ITF, TSO's EDIT command requires more information than OS ITF's. Namely, under TSO, you must specify NEW for new programs and you must specify SCAN if you want syntax error messages to appear as syntax errors are detected.

- *SEND Command:* The OS ITF SEND command is used to test the accuracy of terminal transmission. The TSO SEND command is used to send messages to other users; it is not used to test terminal transmission.
- *RENUM Command:* When a BASIC program is renumbered, statements containing references to other statements (e.g., GOTO, GOSUB, PRINT USING, MAT PRINT USING, and IF . . . THEN/GOTO) are automatically updated. Under OS ITF, *all* the blanks contained in the statement as originally typed are eliminated and the entire statement is squeezed together. Under TSO ITF, only the blanks between the digits of the statement number reference are eliminated; the rest of the statement remains as it was originally typed.
- *Automatic Line-numbering:* In OS ITF, users must type their own statement numbers for BASIC programs. TSO ITF gives the BASIC user the option of typing his own statement numbers or of using the system-typed statement numbers provided in the input phase of the edit mode.
- *File Names in DELETE and RENAME Commands:* The OS ITF DELETE and RENAME commands accept any file name that is acceptable in ITF:BASIC statements. The TSO DELETE and RENAME commands will accept only the first three characters of a file name used in ITF:BASIC statements. These three characters must also adhere to the following TSO restrictions: the first character is required and must be alphabetic; the other two characters are optional and, if specified, must be alphameric.

Glossary

ALPHABETIC CHARACTER Any of the 26 letters (A through Z) of the English alphabet and any of the following special characters (called *alphabetic extenders*): #, @, and $.

ALPHABETIC EXTENDER Any one of the following three special characters: #, @, and $.

ALPHAMERIC CHARACTER Either a digit or an alphabetic character.

ARGUMENT An arithmetic expression appearing in parentheses following a function name (either a user-written function or an intrinsic function). The expression represents the value that the function is to act upon.

ARITHMETIC ARRAY A named table of arithmetic data items. An array may be implicitly declared through usage or explicitly declared in a DIM statement. ITF:BASIC allows one- and two-dimensional arrays.

ARITHMETIC DATA Data with a decimal numeric value.

ARITHMETIC EXPRESSION An arithmetic variable, arithmetic array member, internal constant, numeric constant, function reference, or a series of the above separated by binary operators and parentheses.

ARITHMETIC OPERATORS The symbols representing the operations which can be performed upon arithmetic data. They are:

+	addition or unary plus sign
−	subtraction or unary minus sign
*	multiplication
/	division
↑ or **	exponentiation

ARITHMETIC VARIABLE A single alphabetic character or an alphabetic character followed by a digit used to represent an arithmetic data item whose value is assigned and/or changed during program execution.

ARRAY A named list or table of data items, all of which are the same type—arithmetic or character.

ARRAY DECLARATION The specification of the name and dimensions of an array to be allocated to the user's program. Arrays may be declared explicitly (by the DIM statement) or implicitly through usage.

ARRAY EXPRESSION
1. An expression representing an array value; i.e., an expression in which at least one operand is an array.
2. An operation which is performed on the entire collection of members of an arithmetic array.

ARRAY MEMBER A single data item in an array (as opposed to the entire array).

ARRAY VARIABLE An alphabetic character (for arithmetic arrays) or an alphabetic character followed by the dollar sign character ,$, (for character arrays) whose designation represents an entire array.

ASSIGNMENT The process of giving values to variables.

ATTENTION INTERRUPTION The interruption of execution caused by the pressing of the attention key.

ATTENTION KEY — The key on the terminal keyboard which, once depressed, causes interruption or cancellation of the action in progress (e.g., program execution, program listing) and returns control to you at the terminal.

BINARY OPERATORS — The symbols which represent the operations which can be performed on two items of arithmetic data. They are:

+	addition
−	subtraction
*	multiplication
/	division
↑ or **	exponentiation

BOUND — The upper limit of an array dimension. The lower limit is always assumed to be 1.

BRANCH POINT — A statement number referred to in a GOTO, GOSUB, or IF . . . THEN/GOTO statement.

BREAKPOINT — The point at which program execution in the test mode is to be interrupted and control returned to the terminal; established by the AT subcommand.

BUILT-IN FUNCTION — *See* intrinsic function.

CARRIAGE RETURN — *See* carrier return.

CARRIER RETURN (CR) — Ending a line by pressing the appropriate key(s) on your terminal.

CHARACTER ARRAY — A named table of character data items. An array may be implicitly declared through usage or explicitly declared in a DIM statement. ITF:BASIC allows character arrays of one dimension.

CHARACTER CONSTANT — One or more characters from the BASIC character set enclosed by a pair of single or double quotation marks.

CHARACTER-DELETION CHARACTER — A character within a line specifying that the immediately preceding character is to be deleted from that line.

CHARACTER EXPRESSION — A character variable, character array member, or a character constant.

CHARACTER FORMAT — A format specification used in the Image statement when the item to be printed is character data.

CHARACTER STRING — One or more characters that you can type at your terminal. Unlike the character constant, the character string is not enclosed in quotation marks and can be used only on those BASIC statements which permit a comment after the keyword (e.g., END, STOP, REM, RESET, etc.).

CHARACTER VARIABLE — An alphabetic character followed by the dollar sign character ($) used to represent a character data item whose value is assigned and/or changed during program execution.

CHARACTERISTIC — In the notation of a floating-point number, the four positions which include the letter E, the sign, and the two integers forming the exponent (i.e., everything but the mantissa).

COMMAND — Under TSO, a request from a terminal for the execution of a particular program called a command processor. Any subsequent commands processed directly by that command processor are called subcommands.

COMMAND MODE — One of three modes of operation that apply to ITF:BASIC under TSO. In the command mode, program maintenance and system control functions are performed; also, permanent programs can be executed or tested. This mode is identified by the READY system cue.

COMMENT — A remark or note included in the body of a program by the programmer. It has *no* effect on the execution of the program; it merely documents it. Comments are written as a character string and may appear as a part of any program statement that has no operands (e.g., REM, STOP, END, RESET, etc.).

COMPARISON OPERATORS — *See* relational operators.

CONSTANT — A data item whose value never changes. ITF:BASIC has three types of constants. They are:
1. *numeric*—one or more digits whose value is a decimal number.
2. *character*—one or more characters enclosed in single or double quotation marks.
3. *internal*—the value of π, $\sqrt{2}$, and e.

DATA — A representation of a value. The kinds of data permitted in ITF:BASIC are arithmetic and character.

DATA FILE — *See* file.

DATA TABLE — A list of the values contained in the DATA statements of your program. DATA statements are processed in statement number sequence (lowest to highest). The values of each DATA statement are collected and placed in a single table in order of their appearance (left to right).

DATA TABLE POINTER — An indicator that moves sequentially through the data table, pointing to each value as it is assigned to a corresponding variable in a READ statement. Initially, the indicator refers to the first item in the table. It can be repositioned to the beginning of the table at any time by the RESTORE statement.

DEBUGGING — The process a programmer uses to detect and remove errors from his program. It involves a study of the listing, analysis of the logic used, a check to see that the input data is correct, etc.

DEBUGGING SUBCOMMANDS — The subcommands used in the test mode, which help you to track down logical and semantic errors contained in your program. These subcommands are: TRACE, NOTRACE, LIST, GO, AT, and OFF.

DECLARATION — *See* explicit declaration and implicit declaration.

DELIMITER — Any valid special character or combination of special characters used to define the limits of identifiers, statement lines, or commands.

DIAGNOSTIC MESSAGE — *See* error message.

DIAL-UP TERMINAL — A terminal connected to the computer by a telephone.

DIGITS — 0,1,2,3,4,5,6,7,8,9.

DIMENSION — The parenthesized number or numbers following the array name in a DIM statement (for explicit declarations) or in its first use in the program (for implicit declaration). It specifies how many members the array contains and how those members are arranged.

DUMMY VARIABLE — The simple arithmetic variable enclosed in parentheses after the name of a user-written function in a DEF statement. The function performs its defined calculation on the arithmetic expression value substituted for the dummy variable when the program is executed.

EDIT MODE — One of the three modes of operation that apply to ITF:BASIC under TSO. In the edit mode, programs are created, updated, executed, and tested. The edit mode is identified by the EDIT system cue, or by automatic statement numbering (if the input phase is used).

END-OF-FILE INDICATOR — A user-supplied unique value designating the last entry in a data file.

ERROR MESSAGE — An indicator from the computer that an error has occurred.

EXECUTION — The performance of instructions given to a computer.

EXECUTION ERROR — An error discovered during execution of a BASIC program (i.e., dividing by zero, assigning a variable a value which is outside the permitted range, etc.).

EXPLICIT DECLARATION — The use of a DIM statement to specify the dimensions of an array.

EXPONENT — An integer constant specifying the power of ten by which the base of the decimal floating-point number (mantissa) is to be multiplied.

EXPONENTIAL FORMAT (E-FORMAT) —
1. A format specification used in the Image statement when the item to be printed is a floating-point number.
2. A number written in the form of a mantissa and exponent.

EXPONENTIATION — Raising a value (m) to a power (n); i.e., multiplying m by itself n-1 times.

EXPRESSION — A representation of a value, e.g., variables and constants appearing alone or in combination with operators. Three forms of expressions are defined in ITF:BASIC: scalar, array, and relational.

FILE — A named group of related data items which are retained in permanent storage.

FILE NAME — The name associated with a file.

FIXED-DECIMAL FORMAT (F-FORMAT) — A format specification used in the Image statement when the item to be printed is a fixed-point number.

FIXED-POINT CONSTANT — One or more decimal digits with an optional decimal point and optionally preceded by a sign.

FLOATING-POINT CONSTANT — A decimal fixed-point constant followed by the letter E, followed by an optionally signed one- or two-digit decimal integer constant. The entire constant may be preceded by a sign.

FLOATING-POINT DATA — Numbers written in the form of a mantissa and an exponent.

FORMAT SPECIFICATION — Character-, I-, F-, or E-formats used in an Image statement to specify the printed appearance of the values of character and arithmetic expressions.

FULL PRINT ZONE — Eighteen horizontal print positions. In a PRINT or MAT PRINT statement, a comma is used to indicate that a full print zone should be used.

FUNCTION — A named arithmetic expression that computes a single value from another arithmetic expression. *See also* intrinsic function and user-written function.

FUNCTION REFERENCE — The appearance of an intrinsic function name or a user-written function name in an arithmetic expression.

IDENTIFICATION CODE — A unique combination of one to seven alphameric characters which, when typed in the LOGON command, allow you to use the system. The first character must be an alphabetic character.

IDENTIFIER — A string of characters that represents a decimal number or a character constant. There are five types of identifiers in ITF:BASIC: numeric constants, internal constants, character constants, variables, and function references.

IMPLICIT DECLARATION — The specification of the dimensions of an array by the first appearance of the subscripted array name in the program (i.e., not explicitly specified in a DIM statement).

INPUT/OUTPUT — The transfer of data between an external medium (i.e., the terminal typewriter or a data file) and internal storage.

INPUT PHASE	A phase of edit mode operation in which the system supplies the statement numbers for your statements. Also called input mode in other TSO publications.
INTEGER CONSTANT	One or more decimal digits optionally preceded by a sign.
INTEGER FORMAT (I-FORMAT)	A format specification used in the Image statement when the item to be printed is an integer.
INTERNAL CONSTANT	System-supplied values for π, $\sqrt{2}$, and e which are invoked by typing the identifiers &PI, &SQR, and &E.
INTERRUPTION	The suspension of an activity by the system because of an error or a user request (pressing the attention key, for example).
INTRINSIC FUNCTION	Any of the 24 functions supplied by ITF:BASIC (i.e., SIN, COS, SQR, etc.).
ITERATIVE LOOP	A FOR/NEXT loop, the statements of which are executed repeatedly until the value of the control variable exceeds a predefined limit or until control is transferred out of the loop.
LINE	A single line of one or more characters typed at the terminal and entered into the system.
LINE-DELETION CHARACTER	A character that causes deletion of itself and all preceding characters in a line of terminal input.
LINE NUMBER	*See* statement number.
LOG OFF	The process of ending a terminal session.
LOG ON	The process of establishing a connection with the system; starting a terminal session.
LONG-FORM ARITHMETIC	Precision whereby, externally, values printed with I- and F-format have a maximum of 15 significant digits while values printed in the E-format have a maximum of eleven significant digits in the mantissa.
LONG PRECISION	*See* long-form arithmetic.
LOOP	A sequence of instructions that are executed repeatedly until a terminating condition prevails.
LOWER CASE	this is lower case—no capital letters.
MANTISSA	In floating-point notation (exponential or E-format), the number that precedes the E. The value represented is the product of the mantissa and the power of ten specified by the exponent.
MATRIX	A two-dimensional arithmetic array that can be used in a MAT statement.
MEMBER	*See* array member.
MODE	A method of operation; there are three possible modes in which ITF:BASIC under TSO can operate: command, edit, and test.
NESTING	1. The occurrence of a FOR/NEXT loop within another FOR/NEXT loop. 2. The occurrence of a GOSUB statement when one or more GOSUB statements is already active. 3. The use of more than one set of parentheses to indicate the order of evaluation in a complex arithmetic expression.
NULL CHARACTER STRING	Two adjacent single or double quotation marks that specify a character constant of 18 blank characters.

NULL DELIMITER — One or more blanks or no characters at all (i.e., one data item directly follows another data item with no intervening space or delimiter). A null delimiter may be used between two data items in a PRINT or MAT PRINT statement to specify a packed print zone when one, and only one, of the data items is a character constant.

NUMERIC CONSTANT — A string of characters whose value is a decimal number. The defined value cannot be changed during program execution. The two general forms of a numeric constant are: decimal fixed-point and decimal floating-point.

OPERATOR — A symbol specifying an operation to be performed. *See also* arithmetic operators, binary operators, relational operators, and unary operators.

OUTPUT — *See* input/output.

PACKED PRINT ZONE — Eighteen horizontal print positions or less, depending on the type of data (arithmetic or character) being printed. The semicolon or null delimiter specify that a packed print zone is to be used.

PERMANENT PROGRAM — A program that has been placed in permanent storage through use of the SAVE command.

PERMANENT STORAGE — The internal area in which your permanent programs and data files are saved.

PRECISION — The number of digits over which significance can be expressed and maintained.

PRINT ZONES — *See* full print zone and packed print zone.

PROGRAM — A logically self-contained sequence of BASIC statements that can be executed by the system to attain a specific result.

PROGRAM NAME — A string of one to eight characters by which you identify your program.

PROGRAMMER-DEFINED FUNCTION — *See* user-written function.

REDIMENSIONING — The changing of the dimensions of an array. Redimensioning can occur in any of the following MAT statements: MAT assignment with the CON, IDN, or ZER function, MAT GET, MAT INPUT, and MAT READ. Redimensioning must not exceed the original number of array members nor change the original number of dimensions.

RELATIONAL EXPRESSIONS — A test of the relationship between two arithmetic expressions or two character expressions which will have true or false results.

RELATIONAL OPERATORS — The operators used in relational expressions. They are:

equal to	$=$
not equal to	$\neq$
greater than	$>$
less than	$<$
greater than or equal to	$\geq$
less than or equal to	$\leq$

REMARK — *See* comment.

RUN — The execution of a program.

SCALAR — A single data item (as opposed to an array of items).

SCALAR EXPRESSION — An arithmetic expression or a character expression.

SCALAR REFERENCE — A reference to a single data item.

SEMANTIC ERROR — An error in the structure of your program (i.e., invalid loops, invalid GOTO, etc.) discovered after a RUN subcommand, a RUN command, a BASIC command, or a GO subcommand is given, but before execution actually begins.

SESSION — The time spent at the terminal between logging on and logging off.

SHORT-FORM ARITHMETIC — Precision whereby, externally, values printed with I- and F-format have a maximum of seven significant digits while values printed with E-format have a maximum of seven significant digits in the mantissa.

SHORT PRECISION — *See* short-form arithmetic.

SIGNIFICANT DIGIT — The left-most non-zero digit.

SIMPLE ARITHMETIC VARIABLE — A variable that can only be assigned a decimal number. It is named by a single alphabetic character or an alphabetic character followed by a digit.

SIMPLE CHARACTER VARIABLE — A variable that can only be assigned a character value. It is named by an alphabetic character followed by the dollar sign character, $.

SIMPLE VARIABLES — *See* simple arithmetic variable and simple character variable.

SPECIAL CHARACTERS — Any characters on the keyboard which are not alphameric characters.

STATEMENT LINE — A BASIC statement prefaced by a statement number.

STATEMENT NUMBER — The number which prefaces a BASIC statement. It can be up to five digits in length (in the range 00001 to 99999).

SUBCOMMAND — A request for a particular operation to be performed, the particular operation falling within the scope of work requested by the command to which the subcommand applies.

SUBROUTINE — A program segment (sequence of statements) branched to by a GOSUB statement. The last statement of a subroutine must be a RETURN statement which directs the computer to return and execute the statement following the GOSUB.

SUBSCRIPT — Any valid arithmetic expression (whose truncated integer value is greater than zero) used to refer to a particular member of an array.

SYNTAX CHECKING — A method the computer uses to automatically check each statement or command you type to ensure that it contains correct spelling and punctuation. If the entry is incorrect, the computer will automatically discard it and type out an error message.

SYNTAX ERROR — An error in the format of a BASIC statement (i.e., invalid operator, etc.), a system command, or a system subcommand.

SYSTEM CUE — A computer-printed reminder of the current mode of operation. It may be READY, EDIT, or TEST.

SYSTEM-SUPPLIED CONSTANTS — *See* internal constants.

TERMINAL — A device resembling a typewriter that is used to communicate with the system.

TEST MODE — One of the three modes of operation that apply to ITF:BASIC under TSO. The test mode, in which programs are debugged, can be entered through the command mode or edit mode. It is identified by the TEST system cue.

UNARY OPERATORS — An operator that precedes, and thus is associated with, an arithmetic expression. The unary operators are + (plus) and − (minus).

UPPER CASE — THIS IS UPPER CASE—ALL CAPITAL LETTERS.

USER — Anyone utilizing the services of a computing system; within the ITF:BASIC context, anyone who uses a computing system from a terminal.

USER-IDENTIFICATION CODE — *See* identification code.

USER-WRITTEN FUNCTION — A function defined by the user in a DEF statement.

VARIABLE — An identifier having a value that may change during execution of a program.

This section lists the diagnostic messages generated by ITF:BASIC. The following information is included in the description of each message:

- Short -- the first level of the message; if it ends in a plus sign (+), a second level exists and can be obtained by typing a "?" at your terminal. Messages that do not end in a plus sign have no second level.
- Long -- the message text generated in response to your "?".
- Explanation -- a detailed description of what caused the message. In most instances, this will include the specific action you should take to correct the error.
- Action -- your response to the message.

<u>Note</u>: For all messages that say "SYSTEM ERROR", you should note the message number and contact your installation maintenance personnel.

Execution error messages (noted in this section by an asterisk) will have the following format when displayed at the terminal:

message-number   statement-number   message-text

In addition to the diagnostic messages contained in this section, ITF:BASIC provides error recovery messages for incorrect data typed by the user in response to the INPUT and MAT INPUT statements. These messages, which are not preceded by a message number, are displayed at the terminal in an abbreviated form and indicate the cause of the error. In each case, the user is allowed to correct the error and re-enter the entire data list on the same line that the message is printed. An alphabetically arranged list of these messages and a brief explanation of each message is given below:

<u>Message Text</u>:	<u>Explanation</u>:
NG CON	The magnitude of a numeric constant must be less than $7.2\text{x}10^{+75}$ and must be greater than $5.4\text{x}10^{-79}$ Check to see that your numeric constants are within this range and that you have not forgotten the letter "E" in the exponential format. Also, check the spelling of any internal constant names in your data list.
NG DEL	Constants supplied in response to an INPUT or MAT INPUT statement must be separated by commas.
NG TYP	<u>For MAT INPUT only</u>: Only numeric and internal constants are permitted in the data list supplied for MAT INPUT statements.
NOITEM	You have typed a comma followed by a comma, or you have issued a CR before entering your data.
MSNG	<u>For INPUT only</u>: You have supplied a character constant without enclosing it in single or double quotation marks.

TOOFEW

For INPUT: You have entered fewer constants than the number of variables specified in the INPUT statement.

For MAT INPUT: You have entered fewer constants for a row than the number of members contained in a row of the specified array.

EXCESS

For INPUT: You have entered more constants than the number of variables specified in the INPUT statement.

For MAT INPUT: You have entered more constants for a row than the number of members contained in a row of the specified array.

NUMBERED MESSAGES

1 Short: COMMAND SYSTEM ERROR+
Long: service-routine ERROR CODE nnnn
Explanation: An error has occurred in the TSO routine named service-routine.
Action: Note the error code and contact your installation maintenance personnel.

2 Short: {UTILITY DATA SET|DATA SET dsname} NOT ALLOCATED, TOO MANY DATA SETS+
Long: USE FREE COMMAND TO FREE UNUSED DATA SETS
Explanation: All of the data set allocations in your log-on procedure have been used up.
Action: Use the LISTALC command to display the names of the data sets that have been allocated. Free one or more of these data sets with the FREE command. (FREE and LISTALC are described in the TSO Command Language Reference publication.) If the problem recurs often, contact your installation maintenance personnel for additional allocations in your log-on procedure.

3 Short: DATA SET dsname NOT ALLOCATED, REQUIRED VOLUME NOT MOUNTED+
Long: VOLUME OR CVOL NOT ON SYSTEM AND CANNOT BE ACCESSED
Explanation: A machine error may have occurred.
Action: Contact your installation maintenance personnel.

4 Short: DATA SET dsname NOT IN CATALOG
Long: None.
Explanation: The data set name, as you have entered it, cannot be located in the catalog. You may have misspelled the name or perhaps forgotten to save the data set after creating it.
Action: Check your spelling. Use the LISTCAT command to check the contents of the catalog. If the data set is not listed in the catalog but it exists, contact your installation maintenance personnel.

5 Short: DATA SET dsname WILL CREATE INVALID CATALOG STRUCTURE+
Long: A QUALIFIER CANNOT BE BOTH AN INDEX AND THE LAST QUALIFIER OF A DATA SET NAME
Explanation: You have used a descriptive qualifier that matches

Action: Use another name for your data set.

one of your user-supplied names.

6 Short: DATA SET dsname NOT ALLOCATED, SYSTEM OR INSTALLATION ERROR+
Long: CATALOG ERROR CODE 20
Explanation: A catalog error has occurred.
Action: Contact your installation maintenance personnel.

7 Short: DATA SET dsname NOT ALLOCATED, SYSTEM OR INSTALLATION ERROR+
Long: CATALOG I/O ERROR
Explanation: An error has occurred while attempting to catalog the specified data set.
Action: Contact your installation maintenance personnel.

8 Short: DATA SET dsname NOT ALLOCATED, SYSTEM OR INSTALLATION ERROR+
Long: DYNAMIC ALLOCATION ERROR CODE nnnn
Explanation: A dynamic allocation error has occurred.
Action: Make sure that you cause the second level (long form) of this message to be produced so that the error code is known. Then contact your installation maintenance personnel.

9 Short: DATA SET dsname NOT ALLOCATED, SHARED+
Long: USE FREE COMMAND TO FREE THE DATA SET
Explanation: None.
Action: Free dsname with the FREE command. The FREE command is described in the TSO Command Language Reference publication.

10 Short: DATA SET dsname ALREADY IN USE, TRY LATER+
Long: DATA SET IS ALLOCATED TO ANOTHER JOB OR USER
Explanation: The requested data set is being used by someone else.
Action: Wait a few moments and try again. If this occurs often, you might consider asking your installation personnel to make a copy for your own exclusive use.

11 Short: { UTILITY DATA SET | DATA SET dsname | FILE ddname } NOT ALLOCATED+
Long: NO UNIT AVAILABLE
Explanation: The system is unable to allocate the specified file or data set.
Action: Contact your installation maintenance personnel.

12 Short: DATA SET dsname NOT ALLOCATED, REQUIRED VOLUME NOT MOUNTED+
Long: VOLUME NOT ON SYSTEM AND CANNOT BE ACCESSED
Explanation: The volume specified for dsname has not been mounted.
Action: The correct volume must be mounted before TSO is started for the day. Contact your installation maintenance personnel. You will not be able to use dsname in this session.

13 Short: { UTILITY DATA SET | DATA SET dsname | FILE ddname } NOT ALLOCATED+
Long: INVALID UNIT IN USER ATTRIBUTE DATA SET
Explanation: The device type specified for your data set or file is not supported by TSO. Your user attribute data

		set needs changing.
	Action:	Contact your installation maintenance personnel.
14	Short:	DATA SET dsname NOT ALLOCATED, NOT ENOUGH SPACE ON VOLUMES+
	Long:	USE DELETE COMMAND TO DELETE UNUSED DATA SETS
	Explanation:	All of the storage space that the installation has allocated to TSO has been used.
	Action:	Delete any data sets that you no longer need. If storage is still not available after this, contact your installation maintenance personnel for more storage.
15	Short:	INVALID SYSOUT CLASS
	Long:	None.
	Explanation:	None.
	Action:	Contact your installation maintenance personnel.
16	Short:	INVALID DATA SET NAME, dsname EXCEEDS 44 CHARACTERS
	Long:	None.
	Explanation:	A data set name cannot exceed 44 characters in length (periods included). This excessive length may have resulted from the system's appending of your user identification and a descriptive qualifier to the name you specified.
	Action:	Use a shorter data set name.
17	Short:	FILE {JOBLIB|STEPLIB} INVALID
	Long:	None.
	Explanation:	None.
	Action:	Contact your installation maintenance personnel.
18	Short:	DATA SET dsname NOT ON A DIRECT ACCESS DEVICE, NOT SUPPORTED
	Long:	None.
	Explanation:	The data set specified is not on a direct access device. TSO supports only direct access devices.
	Action:	Have your installation transfer the data set to a direct access device. You may wish to do so yourself if you're familiar with the procedure.
19	Short:	DATA SET dsname RESIDES ON MULTIPLE VOLUMES, NOT SUPPORTED
	Long:	None.
	Explanation:	A data set must reside on a single direct access device.
	Action:	Have your installation transfer the data set to a single direct access. You may wish to do so yourself if you're familiar with the procedure.
20	Short:	{DATA SET dsname|FILE ddname} NOT FREED+
	Long:	SUBALLOCATED DATA SET
	Explanation:	None.
	Action:	Contact your installation maintenance personnel.
21	Short:	{DATA SET dsname|FILE ddname} NOT FREED+
	Long:	GENERATION DATA GROUP
	Explanation:	None.
	Action:	Contact your installation maintenance personnel.
22	Short:	{DATA SET dsname|FILE ddname} NOT FREED+
	Long:	PASSED DATA SET
	Explanation:	None.
	Action:	Contact your installation maintenance personnel.

23 Short: FILE ddname NOT ALLOCATED, IN USE
Long: None.
Explanation: The file given by ddname is already being used by someone else.
Action: Continue with other work, if possible, and try to use ddname again later.

50 Short: DATA SET dsname NOT USABLE+
Long: I/O SYNAD ERROR system-error-information
Explanation: The system-error-information identifies the type of I/O error that occurred.
Action: Make sure that you cause the second level (long form) of the message to be displayed so that you can determine the cause of the error. Contact your installation maintenance personnel.

51 Short: DATA SET dsname NOT USABLE+
Long: CANNOT OPEN DATA SET
Explanation: The system erred while trying to open dsname.
Action: Retry. If this fails, contact your installation maintenance personnel.

52 Short: NOT ENOUGH MAIN STORAGE TO EXECUTE
Long: None.
Explanation: There is not enough main storage to execute your program at this time.
Action: Try to re-execute later. If this problem occurs often, ask your installation maintenance personnel for more space.

53 Short: COMMAND SYSTEM ERROR+
Long: PARSE ERROR CODE nnnn
Explanation: An error has occurred within the TSO PARSE routine.
Action: Make sure that you cause the second level of this message (long form) to be produced and then contact your installation maintenance personnel.

54 Short: USER INPUT DATA SET dsname MUST BE PHYSICAL SEQUENTIAL
Long: None.
Explanation: The data set associated with the file specified in your GET statement does not have the physical sequential organization. You may have allocated the wrong data set or you may have specified the wrong file name in your GET statement.
Action: If you specified the wrong data set in your ALLOCATE command, use the FREE command to free the data set and then re-enter ALLOCATE for the proper data set. If you specified the wrong file name in your GET statement, change the GET statement accordingly. (The FREE command is described in the TSO Command Language Reference publication.)

55 Short: DATA SET NOT A PARTITIONED DATA SET
Long: None.
Explanation: TSO ITF is trying to allocate the partitioned data set userid.DATA to hold your ITF files. You have given a non-partitioned data set this name and, as a result, ITF cannot allocate it as a partitioned data set.
Action: Either rename or delete the data set that you've named userid.DATA and then re-execute.

56 Short: DATA SET dsname NOT USABLE+
Long: DATA SET DOES NOT CONTAIN FIXED LENGTH RECORDS

Explanation: The data set associated with the file you have specified in your GET statement does not contain fixed-length records. You may have allocated the wrong data set or you may have specified the wrong file name in your GET statement.

Action: If you specified the wrong data set in your ALLOCATE command, use the FREE command to free the data set and then re-enter ALLOCATE for the correct data set. If you specified the wrong file name in your GET statement, change the GET statement accordingly. (The FREE command is described in the _TSO Command Language Reference_ publication.)

57 **Short:** DATA SET dsname NOT USABLE+

Long: DATA SET MUST HAVE RECORDS OF 120 CHARACTERS

Explanation: The records in the data set associated with the file specified in your GET statement are not 120 characters long. You may have allocated the wrong data set or you may have specified the wrong file name.

Action: If you allocated the wrong data set, use the FREE command to free the data set and then allocate the correct data set. If you specified the wrong file name in your GET statement, change the GET statement accordingly. (The FREE command is described in the _TSO Command Language Reference_ publication.)

58 **Short:** DATA SET dsname EMPTY

Long: None.

Explanation: The data set associated with the file named in your GET statement is empty. You may have allocated the wrong data set or you may have specified the wrong file name.

Action: If you allocated the wrong data set, use the FREE command to free the data set and then allocate the correct data set. If you used the wrong file name in your GET statement, change the GET statement accordingly. (The FREE command is described in the _TSO Command Language Reference_ publication.)

59 **Short:** DATA SET dsname NOT USABLE+

Long: DATA SET MAY BE USED ONLY FOR INPUT

Explanation: The data set associated with the file named in your PUT statement can be used only for input. Only members of _userid_.DATA can be used in PUT statements.

Action: Change the file name in your PUT statement and then re-execute.

61 **Short:** RECORD EXCEEDS MAXIMUM OF 128 CHARACTERS

Long: None.

Explanation: Your ITF:PL/I or ITF:BASIC statement exceeds the ITF maximum of 128 characters (including the 8 characters for the line number).

Action: List the data set and see which statements are too long. Then shorten the statements in error (e.g., if the problem is with an ITF:BASIC DATA statement, spread the data over two or more DATA statements)

62 **Short:** COMMAND SYSTEM ERROR+

Long: UNABLE TO ISSUE STAE

Explanation: An internal error has occurred.

Action: Contact your installation maintenance personnel.

63 Short: COMMAND SYSTEM ERROR+
Long: NESTING LEVEL EXCEEDED
Explanation: An internal error has occurred in the ITF dispatcher.
Action: Contact your installation maintenance personnel.

64 Short: COMMAND SYSTEM ERROR+
Long: NO MORE SAVE AREAS
Explanation: An internal error has occurred in the ITF dispatcher.
Action: Contact your installation maintenance personnel.

65 Short: COMMAND SYSTEM ERROR+
Long: NESTING LEVEL IS 0
Explanation: An internal error has occurred in the ITF dispatcher.
Action: Contact your installation maintenance personnel.

67 Short: DATA SET dsname NOT USABLE+
Long: ERROR OCCURRED WHILE ATTEMPTING TO UPDATE DIRECTORY
Explanation: An internal error has occurred while the system was trying to update the directory associated with userid.DATA.
Action: Retry. If the problem recurs, contact your installation maintenance personnel.

69 Short: INVALID LINE NUMBER ENCOUNTERED+
Long: ITF DATA SET MUST CONTAIN VALID LINE NUMBERS
Explanation: The data set that you are trying to use as an ITF:BASIC or ITF:PL/I program contains either an invalid line number or no line numbers at all. ITF programs must have valid line numbers.
Action: List the contents of the data set. Perhaps you specified the wrong data set name.

101 Short: INV CMD+
Long: INVALID COMMAND
Explanation: Check your spelling. Information on the use of commands and subcommands is given in Part III. This message may also arise because of a transmission error; if you are certain that the line you typed is correct, re-enter the line in the same form.
Action: Re-enter the corrected command or subcommand.

102 Short: INVALID TEST SUBCOMMAND
Long: None.
Explanation: You have made an error in your test mode subcommand. If the error was (1) missing parentheses on the identifier list for the LIST, TRACE, or NOTRACE subcommands, (2) a space between NO and TRACE in the NOTRACE subcommand, or (3) any other syntax error in an END, GO, or LIST test mode subcommand, you must re-enter the entire subcommand. If, however, you have made a syntax error in an AT, OFF, TRACE, or NOTRACE subcommand, that part of the subcommand preceding the occurrence of the error is accepted; the rest of the subcommand is rejected and this message is printed at the terminal. For example:

```
A40,50;60
TRACE (A,X,ZZZZ,Y)
```

In the AT subcommand given above, 40 and 50 are accepted and 60 is rejected because it follows an invalid delimiter. In the TRACE subcommand given

above, A and X are accepted but ZZZZ and Y are rejected. In this case, ZZZZ is an invalid identifier and Y follows the occurrence of the error. You could enter another subcommand including just that portion of the subcommand which was rejected; however, it is recommended that you re-enter the entire subcommand.

Action: You may either (1) re-enter the entire subcommand, or (2) enter another subcommand including just that portion of the subcommand which was rejected (see the explanation above).

103
Short: INV CMD+
Long: COMMAND INVALID IN xxxx MODE
Explanation: Check to see that you have typed a statement number before a statement entered in the edit mode. Also make sure that you have typed the numeral "1" rather than the lower-case letter "l" in statement numbers. Rules governing the use of system commands and subcommands are given in Part III.
Action: Enter the correct command or subcommand.

116
Short: PARENS EMPTY+
Long: PARENTHESES MUST ENCLOSE AT LEAST ONE CHARACTER
Explanation: A left parenthesis and its corresponding right parenthesis must be separated by at least one character.
Action: Re-enter the corrected command or subcommand.

117
Short: TOO MNY (+
Long: TWO LEFT PARENTHESES FOUND WITHOUT INTERVENING RIGHT PARENTHESIS
Explanation: A right parenthesis has been omitted or a left parenthesis appears where a right parenthesis was intended. Every left parenthesis must have a corresponding right parenthesis and vice versa.
Action: Re-enter the command or subcommand inserting a right parenthesis where needed.

118
Short:) MSNG+
Long: LEFT PARENTHESIS FOUND WITHOUT MATCHING RIGHT PARENTHESIS
Explanation: Every left parenthesis must have a corresponding right parenthesis and vice versa.
Action: Re-enter the command or subcommand inserting a right parenthesis where needed.

119
Short: WORD TOO LONG+
Long: TOO MANY CONSECUTIVE CHARACTERS WITHOUT DELIMITER
Explanation: The following commands and subcommands require blanks as delimiters: CONVERT, DELETE (in the edit mode), EDIT, LIST (in the edit mode), RENAME, RENUM, and RUN. AT, LIST (in the test mode), NOTRACE, OFF, and TRACE require commas as delimiters. The syntax of each system command and subcommand is given in Part III.
Action: Re-enter the command or subcommand inserting a comma or a blank where needed.

125
Short: INV COMNT+
Long: NO COMMENTS ARE ALLOWED ON ANY COMMANDS
Explanation: None.
Action: Delete the comment and re-enter the command or subcommand.

127 Short: SYSTEM ERROR+
Long: SYSTEM ABEND CODE nnn
Explanation: An error has occurred within an ITF routine.
Action: Be sure to obtain the second level (long form) of the message. If the error occurred in the command mode, re-enter the command. If the error occurred in the edit mode, save the current program, end the mode, and then re-enter the edit mode for that program. This should re-establish the conditions that existed prior to the occurrence of the error. In any event, contact your installation maintenance personnel.

139 Short: INV KYWD+
Long: CHAR60/CHAR48 NOT VALID FOR BASIC
Explanation: CHAR60/CHAR48 are valid only for PL/I.
Action: Re-enter the command omitting the invalid keyword.

142 Short: PGM EMPTY+
Long: PROGRAM IS EMPTY
Explanation: You have deleted all the statements from your program or you have saved a program which contains no statements, and then issued a LIST or RENUM subcommand. This messages is to inform you that the program contains no statements; therefore, the service you requested cannot be performed.
Action: Add statements to your program and re-enter the subcommand, or delete the program from permanent storage by the DELETE command.

146 Short: SYSTEM ERROR
Long: None.
Explanation: None.
Action: Contact your installation maintenance personnel.

147 Short SYSTEM ERROR
Long: None.
Explanation: None.
Action: Contact your installation maintenance personnel.

148 Short: SYSTEM ERROR
Long: None.
Explanation: None.
Action: Contact your installation maintenance personnel.

149 Short: SYSTEM ERROR
Long: None.
Explanation: None.
Action: Contact your installation maintenance personnel. space.

170 Short: INV SYNTAX+
Long: ELEMENTS OF COMMAND INVALID OR IN IMPROPER SEQUENCE
Explanation: Syntax for system commands and subcommands and rules for their usage are given in Part III.
Action: Re-enter the corrected command or subcommand.

171 Short: NO EXEC STMTS+
Long: THIS PROGRAM CONTAINS NO EXECUTABLE STATEMENTS
Explanation: You are typing to execute a program that contains only comments or non-executable statements.
Action: Insert the statements you need and re-execute the program.

190 Short: PGM NOT EX+
Long: A NULL PROGRAM CANNOT BE RUN
Explanation: A null program (one that contains no statements) can never be executed.
Action: Re-enter the command or subcommand specifying the name of a program that contains executable statements.

300 Short: CMD SYSTEM ERROR+
Long: {PUTGET|PARSE} ERROR CODE nnnn
Explanation: The system has erred in translating or executing your command.
Action: Re-enter the CONVERT command. If the problem recurs, make note of the message number and error code and contact your installation maintenance personnel.

303 Short: NO MEMBER SPECIFIED FOR PARTITIONED DATA SET dsname
Long: None.
Explanation: The dsname in your CONVERT command is a partitioned data set. The required member name is missing. You must always indicate which member of the partitioned data set is involved. Perhaps you intended to specify the name of some other data set and misspelled it.
Action: Re-enter the CONVERT command with either an appropriate member name in parentheses after dsname or a new dsname.

304 Short: MEMBER member-name NOT IN DATA SET dsname
Long: None.
Explanation: No member in dsname has the name you have specified. You may have misspelled the member name or the data set name.
Action: Re-enter the CONVERT command with the correct name(s).

305 Short: ORGANIZATION OF DATA SET dsname NOT ACCEPTABLE+
Long: ORGANIZATION MUST BE PARTITIONED OR SEQUENTIAL
Explanation: Only the names of partitioned and sequential data sets can be specified in the CONVERT command. Perhaps you have specified the wrong dsname.
Action: If dsname is correct, the associated data set must be recreated with an acceptable data set organization before the CONVERT command can be used for it. If dsname is wrong, re-enter the CONVERT command with the correct dsname.

306 Short: MEMBER member-name SPECIFIED BUT dsname NOT PARTITIONED
Long: None.
Explanation: The data set given by dsname is not a partitioned data set and therefore has no members. You probably are using the wrong dsname. (If you are trying to convert one of your OS ITF collections to TSO ITF form, you should be using your OS ITF user identification code as the dsname.)
Action: Re-enter the CONVERT command with the correct dsname.

307 Short: MEMBER member-name OF dsname ALREADY EXISTS+
Long: DATA SET OR MEMBER MUST BE NEW
Explanation: The name that your have specified for the data set that will hold your converted program matches that of an existing sequential data set or member of a

partitioned data set. The sequential data set or member must not already exist.

Action: Re-enter the CONVERT command with another name for the data set or member.

308 Short: NO CONVERT OUT FOR BASIC, ONLY CONVERT IN
Long: None.
Explanation: CONVERT OUT can be used only for ITF:PL/I programs. Perhaps you mistakenly specified the name of an ITF:BASIC program where you intended to specify that of an ITF:PL/I program.
Action: Re-enter the CONVERT command with the name of an ITF:PL/I program, if this was your original intention.

310 Short: INVALID LRECL, nnn+
Long: 128 MAX LRECL FOR BASIC AND IPLI
Explanation: The maximum logical record length allowed by ITF is 128. You have specified a value larger than this in your LRECL option.
Action: Re-enter the CONVERT command and specify a logical record length of 128 or less.

311 Short: DATA SET dsname NOT USABLE+
Long: CANNOT OPEN DATA SET
Explanation: An error occurred while the system was trying to open dsname.
Action: Re-enter the CONVERT command. If the problem recurs, make note of the pertinent information (message number and dsname) and then contact your installation maintenance personnel.

312 Short: INVALID LRECL, 0
Long: None.
Explanation: You have erroneously specified a logical record length of 0 in the LRECL option. The value specified in LRECL must be an integer from 1 through 128 (for CONVERT IN) or from 1 through 100 (for CONVERT OUT).
Action: Re-enter the CONVERT command and specify a valid logical record length in LRECL.

313 Short: INVALID DATA SET NAME, dsname+
Long: EXCEEDS 44 CHARACTERS
Explanation: The data set name specified in your CONVERT command is longer than 44 characters when fully qualified. (Note that even though the name that you actually specified may be less than 44 characters, it is long enough so that the qualifiers appended by the system cause the excessive length.) Qualified names are discussed in the TSO Terminal User's Guide, GC28-6763.
Action: Re-enter the CONVERT command with a shorter data set name.

314 Short: INVALID BLOCK SIZE, nnn+
Long: BLOCK NOT MULTIPLE OF LRECL
Explanation: The block size specified in the CONVERT command must be an integral multiple of the logical record length specified in LRECL. If LRECL is omitted, the block size must be an integral multiple of 80 (for CONVERT IN) or an integer from 1 through 100 (for CONVERT OUT).
Action: Re-enter the CONVERT command with a valid block size.

315 Short: IN OR OUT AND DATA SET2 MUST BE SPECIFIED[, REENTER-]
Long: None.
Explanation: The CONVERT command must always include the IN or OUT option. In turn, IN or OUT must always include the name to be given to the converted program.
Action: If "REENTER-" appears at the end of the message, enter only the IN or OUT option with the required data set name in parentheses. If "REENTER-" does not appear, then you must re-enter the entire CONVERT command to include the IN or OUT option.

316 Short: GOFORT, IPLI, OR BASIC MUST BE SPECIFIED[, REENTER-]
Long: None.
Explanation: You must always indicate the type of program being converted. GOFORT, IPLI, and BASIC indicate Code and Go FORTRAN, ITF:PL/I, and ITF:BASIC, respectively. (ITF users should ignore the GOFORT reference.) BASIC is not allowed for CONVERT OUT usage.
Action: If "REENTER-" appears at the end of the message, then enter only IPLI or BASIC, whichever applies. If "REENTER-" does not appear, you must re-enter the entire CONVERT command to include IPLI or BASIC.

319 Short: CHAR60/CHAR48 VALID ONLY FOR IPLI OUT
Long: None.
Explanation: CHAR60 and CHAR48 are ITF:PL/I options and they are valid only for CONVERT OUT.
Action: Re-enter the CONVERT command correctly.

320 Short: INVALID BLOCK SIZE, 0
Long: None.
Explanation: You have erroneously specified a block size of 0 in your CONVERT command. The block size must be an integral multiple of the logical record length. If no logical record length is specified, the block size must be a multiple of 80 (for CONVERT IN) or an integer from 1 through 100 (for CONVERT OUT).
Action: Re-enter the CONVERT command and specify a valid block size.

321 Short: INVALID BLOCK SIZE, nnn+
Long: BLOCK EXCEEDS MAX OF 100 FOR PACKED RECORDS
Explanation: When LRECL is omitted for CONVERT OUT, the block size specification (if given) must not exceed 100.
Action: Re-enter the CONVERT command and specify a valid block size.

350 Short: INVALID BLOCK SIZE, nnn+
Long: BLOCK SIZE EXCEEDS MAXIMUM
Explanation: The block size that you have specified exceeds the maximum permitted for the type of direct access storage being used at your installation (e.g., for 2311 disk storage, the maximum block size is 3625). Contact your installation maintenance personnel for details about the type of direct access storage being used.
Action: Re-enter the CONVERT command with a valid block size.

351 Short: DATA SET dsname NOT USABLE+
Long: CANNOT OPEN DATA SET
Explanation: An error occurred while the system was trying to

open dsname.
Action: Re-enter the CONVERT command. If the problem recurs, note the message number and dsname and contact your installation maintenance personnel.

353 Short: NOT ENOUGH MAIN STORAGE TO EXECUTE COMMAND
Long: None.
Explanation: There is currently not enough room in main storage to perform the conversion. Specifically, room is needed to hold the OS ITF collection being converted. This may be a temporary problem caused by the concurrent demands of other terminal users.
Action: Wait a few moments and then re-enter the CONVERT command. You may keep trying, but if the problem keeps recurring you should contact your installation maintenance personnel.

354 Short: DATA SET dsname EMPTY
Long: None.
Explanation: The data set named dsname contains no information. You may have specified the wrong data set name in your CONVERT command.
Action: If the data set name was wrong, re-enter the CONVERT command with the correct name.

355 Short: INPUT SOURCE LRECL EXCEEDS SPECIFIED LRECL
Long: None.
Explanation: The logical record length of the program being converted exceeds the logical record length specified in the CONVERT IN command.
Action: Re-enter the CONVERT command with a logical record length that is at least as large as that of the program being converted.

407 Short: PROG TOO BIG+
Long: TOO MANY STATEMENTS IN THIS COMPILATION FOR USER AREA
Explanation: There is insufficient space in your current user area to entirely contain this program.
Action: Either shorten your program or contact your installation maintenance personnel to obtain more space.

447 Short: SYSTEM ERROR
Long: None.
Explanation: None.
Action: Contact your installation maintenance personnel.

*464 Short: INV NUM ARGS FOR B-IN FUNC+
Long: ONLY ONE ARGUMENT IS ALLOWED FOR THIS BUILT-IN FUNCTION
Explanation: In ITF:BASIC all intrinsic functions require one argument with the exception of RND for which the argument is optional (see Part II under the heading "Intrinsic Functions").
Action: Correct the statement and re-execute the program.

487 Short: SYSTEM ERROR
Long: None.
Explanation: None.
Action: Contact your installation maintenance personnel.

*494 Short: HYP-SINE, -COSINE |x| > 174.673+
Long: ABSOLUTE VALUE OF HYPERBOLIC SINE OR COSINE ARGUMENT MUST BE LESS THAN OR EQUAL TO 174.673

Explanation: The absolute value of the argument you supplied to the HSN or HCS intrinsic function exceeds 174.673.
Action: Check the logic of your program, correct the statement(s) in error, and re-execute the program.

*497 Short INV ARG SQUARE ROOT+
Long: SQUARE ROOT BUILT-IN FUNCTION DOES NOT ACCEPT NEGATIVE ARGUMENTS
Explanation: The value of the argument passed to the SQR intrinsic function must not be less than zero.
Action: Check the logic of your program, correct the statement(s) in error, and re-execute the program.

531 Short: SYSTEM ERROR
Long: None.
Explanation: None.
Action: Contact your installation maintenance personnel.

*532 Short: EXP ARG TOO BIG+
Long: BUILT-IN FUNCTION EXP RECEIVED ARGUMENT GREATER THAN 174.6
Explanation: The value of the argument passed to the EXP intrinsic function must be within the range -180.2 to 174.6. If the argument's value is less than -180.2, a zero result is returned.
Action: Check the logic of your program, correct the statement(s) in error, and re-execute the program.

*533 Short: LOG ARG <= 0+
Long: ARGUMENT TO LOGARITHM (BASE E, 2, OR 10) BUILT-IN FUNCTION CANNOT BE LESS THAN OR EQUAL TO ZERO
Explanation: The value of the argument passed to the LOG, LTW, or LGT intrinsic functions must not be less than or equal to zero.
Action: Check the logic of your program, correct the statement(s) in error, and re-execute the program.

*536 Short: INV SIN/COS ARG+
Long: ABSOLUTE VALUE OF ARGUMENT OF SIN OR COS BUILT-IN FUNCTION MUST BE LESS THAN PI * 2 ** 50
Explanation: The value of the argument you have passed to the SIN or COS intrinsic function exceeds or equals π * 2 ** 50.
Action: Check the logic of your program, correct the statement(s) in error, and re-execute the program.

*547 Short: DIV BY ZERO+
Long: DIVISION BY A FLOATING POINT NUMBER WITH ZERO FRACTION WAS ATTEMPTED
Explanation: Division by zero is not allowed in ITF. Check your program logic, correct the necessary statement(s), and re-execute the program.
Action: Correct the error and re-execute the program.

601 Short: IDENT AREA FULL+
Long: INTERNAL AREA IN WHICH IDENTIFIERS ARE KEPT IS FULL
Explanation: Your program contains too many identifiers for the current size of the internal identifier area.
Action: Either eliminate some of the identifiers in your program and re-execute it, or contact your installation maintenance personnel to obtain a larger user area.

602 Short: INV NUM CON+
Long: INVALID NUMERIC CONSTANT

Explanation: The magnitude of a numeric constant must be less than $7.2x10^{+75}$ and must be greater than $5.4x10^{-79}$. Either your numeric constant is outside this range, or you have left out the letter "E" in the exponential format.

Action: If the error message appeared as you were creating the program, correct the statement and continue. If, however, the error message appeared after you issued a RUN or BASIC command, or a RUN subcommand, you must correct the statement and re-execute the program.

603 Short: SYNTAX ERR EXPR+

Long: SYNTAX ERROR IN AN EXPRESSION

Explanation: Either your expression is incomplete (you have issued a carriage return before completing the expression) or you have accidentally struck the shift-key or the wrong character on the keyboard. Also, check to see that parentheses (if present) are matched and that you have used prefix operators correctly.

Action: Re-enter the corrected statement.

*604 Short: OPND ** INV+

Long: IN X**Y, Y MUST BE AN INTEGER WHEN X IS LESS THAN ZERO

Explanation: ITF:BASIC does not permit you to raise a negative number to a fractional power (which would be the result of the exponentiation operation when X is less than zero and Y is not an integer).

Action: Correct the program and re-execute it.

605 Short NO SPACE+

Long: WORKSPACE FULL

Explanation: Your program is too large for the current workspace.

Action: Either shorten your program or contact your installation maintenance personnel to obtain more space.

*606 Short: PREV REF ARRY+

Long: ARRAY IN DIM STATEMENT ALREADY REFERENCED

Explanation: An array name cannot appear in a DIM statement if it has already been used in another statement (implicit declaration) or if it has already been used in another DIM statement (explicit declaration).

Action: Correct the statement(s) in error and re-execute the program.

*607 Short: NUM DIM INV+

Long: ARRAY DID NOT HAVE THE SAME NUMBER OF DIMENSIONS THROUGHOUT THE PROGRAM

Explanation: Early in your program you implicitly or explicitly declared an array to have a certain number of dimensions, and later in your program you have referred to that array using a different number of dimensions than you originally stated.

Action: Check the logic of your program, correct the statement(s) in error, and re-execute the program.

608 Short: MSNG , OR ON+

Long: IN COMPUTED GOTO, STATEMENT NUMBERS MUST BE FOLLOWED BY A COMMA OR THE KEYWORD ON

Explanation: The correct syntax of the computed GOTO is:

GOTO s_1[,s_2,...,s_n] ON arithmetic-expression

where s_i is a statement number. Rules governing the use of the computed GOTO are given in Part II under the heading "Program Statements."

Action: Re-enter the corrected statement.

609 Short: MSNG STMT NUM+
Long: IN COMPUTED GOTO, EACH COMMA MUST BE FOLLOWED BY A STATEMENT NUMBER
Explanation: The correct syntax of the computed GOTO is:

GOTO s_1[,s_2,...s_n] ON arithmetic-expression

where s_i is a statement number. Rules governing the use of the computed GOTO are given in Part II under the heading "Program Statements."
Action: Re-enter the corrected statement.

610 Short: TOO MNY DIM+
Long: ARRAYS MAY NOT HAVE MORE THAN TWO DIMENSIONS
Explanation: In ITF:BASIC, arithmetic arrays must have one or two dimensions.
Action: Re-enter the statement correcting the number of dimensions.

*611 Short: MSNG ,+
Long: THE ONLY VALID DELIMITER IN THIS STATEMENT IS A COMMA
Explanation: Perhaps you have made a typing error or you have attempted to separate items by blanks where a comma must be used. See Part II under the heading "Program Statements" to obtain the correct syntax of the statement you are using.
Action: Correct the statement(s) in error and re-execute the program.

*612 Short: PREV DEF FUNC+
Long: FUNCTION DEFINED IN DEF STATEMENT MAY NOT BE DEFINED MORE THAN ONCE
Explanation: A function may be defined anywhere in the program (before or after its use), but must be defined only once.
Action: Correct the statement(s) in error and re-execute the program.

613 Short: INV FUNC NM+
Long: A SINGLE ALPHABETIC CHARACTER MUST FOLLOW THE USER FUNCTION FN
Explanation: The name of the defined function must be a single alphabetic character, preceded by the letters FN (e.g., FNA, FNB,...,FN@, FN#, FN$).
Action: Re-enter the statement correcting the function name.

614 Short: INV FUNC VAR+
Long: THE DUMMY VARIABLE USED WITH THE USER FUNCTION FN MUST BE A SIMPLE ARITHMETIC VARIABLE
Explanation: In ITF:BASIC, simple arithmetic variables are indicated by a single alphabetic character or by an alphabetic character followed by a digit. The dummy variable you have used is not a valid simple arithmetic variable name.
Action: Re-enter the statement using a valid dummy variable.

615 Short: MSNG = OR UNID STMT+
Long: MISSING EQUAL SIGN IN ASSIGNMENT STATEMENT OR UNIDENTIFIABLE STATEMENT TYPE
Explanation: Check the spelling of statement keywords and check for possible typing errors. The correct syntax of ITF:BASIC statements is given in Part II under the heading "Program Statements." This message is also given when a carriage return is given before completion of the statement.
Action: Re-enter the corrected statement.

616 Short: FUNC MSNG (+
Long: FUNCTION NAME MUST BE FOLLOWED BY LEFT PARENTHESIS
Explanation: In ITF:BASIC, all function names except RND must be followed by a left parenthesis, an argument (an arithmetic expression representing the value that the function is to act upon), and a right parenthesis.
Action: Re-enter the corrected statement.

617 Short: MSNG (+
Long: LEFT PARENTHESIS MISSING FROM STATEMENT
Explanation: Every right parenthesis must have a corresponding left parenthesis and vice versa. Check to see that the right parenthesis in your statement was intended and is not just a typing error.
Action: Re-enter the statement inserting a left parenthesis where needed.

618 Short: MSNG)+
Long: RIGHT PARENTHESIS MISSING FROM STATEMENT
Explanation: Every left parenthesis must have a corresponding right parenthesis and vice versa. Check to see that the left parenthesis in your statement was intended and is not just a typing error.
Action: Re-enter the statement inserting a right parenthesis where needed.

619 Short: NOT POS INT+
Long: BOUNDS IN A DIM STATEMENT MUST BE POSITIVE INTEGERS
Explanation; In ITF:BASIC, array bounds in a DIM statement must be specified by positive integers that are within the range 1-255. Zero, negative or fractional numbers are not valid.
Action: Re-enter the corrected statement.

*620 Short: NM NOT DCL+
Long: NAMES USED IN MAT STATEMENTS MUST FIRST BE IMPLICITLY OR EXPLICITLY DECLARED
Explanation: Prior to usage in a MAT statement, an array must have been implicitly defined by usage in a non-MAT statement or explicitly defined by a DIM statement. This means that the statement number of the MAT statement must be higher than that of the first usage of the subscripted array name (implicit declaration) or the DIM statement (explicit declaration).
Action: Correct the statement(s) in error and re-execute the program.

*621 Short: MAT NOT CNF+
Long: MATRICES MUST BE CONFORMABLE IN THIS STATEMENT
Explanation: "Conformable" has different meanings according to the matrix function or MAT statement being used. It may mean that the two matrices must have identical

dimensions, or that the rows and columns of the matrices must have a certain relationship. For specific rules governing the use of MAT statements, see Part II under the heading "Array Operations."

Action: Correct the statement(s) in error and re-execute the program.

622 Short: SAME MAT+

Long: NAMES ON THE LEFT AND RIGHT SIDES OF THE EQUAL SIGN MUST REFER TO DIFFERENT MATRICES

Explanation: This error could occur in any one of the following forms of the MAT assignment statement: inversion (INV), multiplication, and transpose (TRN). The correct syntax of all MAT statements and rules governing their use are given in Part II under the heading "Array Operations."

Action: Re-enter the statement using different array names on the left and right sides of the equal sign.

623 Short: INV STMT NUM+

Long: STATEMENT NUMBERS MUST CONSIST OF ONE TO FIVE DIGITS

Explanation: Statement numbers must be from one to five digits in length (within the range 00001 to 99999). Make certain that you are typing the numeral "1" rather than the lower-case letter "l" and that you have not typed an extra comma (or other character) after a statement number following THEN or GOTO in an IF statement.

Action: Re-enter the corrected statement.

*624 Short: INV FOR/NEXT BRANCH+

Long: INVALID BRANCH INTO FOR/NEXT LOOP

Explanation: Transfer of control into or out of a FOR/NEXT loop is allowable within the constraints that a NEXT statement cannot be executed if its associated FOR statement is inactive. A FOR statement is inactive if it has not been executed, or if the FOR/NEXT loop was previously completed.

Action: Check the logic of your program, correct the statement(s) in error, and re-execute the program.

626 Short: THEN/GOTO MSNG+

Long: IF STATEMENT MUST CONTAIN "THEN" OR "GOTO"

Explanation: The correct syntax of the IF statement is:

IF x_1 op x_2 {THEN|GOTO} statement-number

where x_1 and x_2 are scalar expressions and op is a relational operator. Rules governing the use of the IF statement are given in Part II under the heading "Program Statements."

Action: Re-enter the corrected statement.

627 Short: TO MSNG+

Long: FOR STATEMENT MUST CONTAIN KEYWORD "TO"

Explanation: The correct syntax of the FOR statement is:

FOR v = x_1 TO x_2[STEP x_3]

where v is a simple arithmetic variable and x_i is an arithmetic expression. Rules governing the use of the FOR statement are given in Part II under the heading "Program Statements."

Action: Re-enter the corrected statement.

628 **Short:** MSNG STMT NUM+
Long: STATEMENT NUMBER MISSING
Explanation: You have omitted the statement number from one of the following statements: MAT PRINT USING, PRINT USING, GOTO, GOSUB, or IF...THEN/GOTO. If you are not certain of the statement numbers, use LIST in the edit mode to display your program.
Action: Re-enter the statement inserting the desired statement number.

629 **Short:** NON ARITH EXPR+
Long: EXPRESSIONS IN THIS STATEMENT MUST BE ARITHMETIC
Explanation: Only arithmetic expressions are allowed to follow the word ON in the computed GOTO statement. Correct syntax for the GOTO statement and rules governing its use are given in Part II under the heading "Program Statements."
Action: Re-enter the corrected statement.

630 **Short:** EXPR DIFF TYPE+
Long: BOTH EXPRESSIONS IN THIS STATEMENT MUST BE OF THE SAME TYPE - BOTH CHARACTER OR BOTH ARITHMETIC
Explanation: You have attempted to assign an arithmetic value to a character variable (or vice versa) in a LET statement, or you have tried to compare a character expression to an arithmetic expression in an IF statement. Rules governing the use of the IF and LET statements are given in Part II under the heading "Program Statements."
Action: Re-enter the corrected statement.

631 **Short:** EXTRA GOSUB+
Long: TOO MANY ACTIVE GOSUBS
Explanation: A GOSUB statement is considered to be active when it has been executed and its associated RETURN statement has not been executed. In ITF:BASIC, there may be no more than 56 active GOSUB statements in a program. Your program exceeds this implementation limit.
Action: Alter your program logic, eliminating excess GOSUB statements, and then re-execute the program.

632 **Short:** MSNG DEL+
Long: MISSING DELIMITER
Explanation: Perhaps you have made a typing error or you have attempted to separate items by blanks where a comma must be used. See Part II under the heading "Program Statements" to obtain the correct syntax of the statement you are using.
Action: If this message appeared as you were creating the program, correct the statement and continue. If, however, the message appeared after you issued a RUN or BASIC command, or a RUN subcommand, you must correct the statement and re-execute the program.

633 **Short:** NUM FOR/NEXT NOT =+
Long: THE PROGRAM MUST CONTAIN THE SAME NUMBER OF FORS AND NEXTS
Explanation: FOR and NEXT statements must be paired. Your program contains an extra FOR or an extra NEXT statement.
Action: Check the logic of your program. Correct the statement(s) in error and re-execute the program.

*634 **Short:** OPND RANGE ** INV+
Long: IN X**Y, ERROR IF X < 0 AND Y < -2**31 OR Y > 2**31-1
Explanation: ITF:BASIC does not permit you to raise a negative number to a power which is outside the range -2**31 to 2**31-1.
Action: Check the logic of your program. Correct the statement(s) in error and re-execute the program.

635 **Short:** NOT ARITH VAR+
Long: FOR AND NEXT STATEMENTS MUST BE FOLLOWED BY A SIMPLE ARITHMETIC VARIABLE
Explanation: In ITF:BASIC, simple arithmetic variables are indicated by a single alphabetic character or by an alphabetic character followed by a digit. The variable you have used in the FOR and/or NEXT statement is not a valid simple arithmetic variable name.
Action: Re-enter the statement using a valid simple arithmetic variable.

*636 **Short:** EXTRA FORS+
Long FOR NESTING EXCEEDS IMPLEMENTATION LIMIT (15)
Explanation: In ITF:BASIC, FOR/NEXT loops may be nested 15 levels deep (with the outermost FOR/NEXT loop considered to be the first level).
Action: Alter your program logic, eliminating the excess nested FOR/NEXT loops, and re-execute the program.

*637 **Short:** NO FMT SPEC+
Long: PRINT USING OPERAND REQUIRES FORMAT SPECIFICATION IN IMAGE
Explanation: Arithmetic and character expressions given in the PRINT USING statement are printed according to corresponding character-, I-, F- or E-formats given in the specified Image statement. Your Image statement does not provide a format specification for the expression to be printed. The correct syntax of the PRINT USING and Image statements and rules governing their use are given in Part II under the heading "Program Statements."
Action: Correct the statement(s) in error and re-execute the program.

638 **Short:** TOO MNY DIMS+
Long: CHARACTER ARRAYS MAY NOT HAVE MORE THAN ONE DIMENSION
Explanation: In ITF:BASIC, a character array is limited to one dimension (arithmetic arrays may have one or two dimensions), and must contain only character data.
Action: Re-enter the statement using only one dimension.

639 **Short** MSNG OR INV IDENT+
Long: MISSING OR INVALID IDENTIFIER
Explanation: This message occurs whenever you have done one of the following:

1. forgotten to type an identifier -- e.g., LET = 72
2. made a typing error -- e.g., LET X = &PT rather than &PI
3. incorrectly named a variable -- e.g., LET AB = 72 rather than A1, A2,...,A9

Rules for naming variables and the correct names for internal constants are given in Part I under the heading "Writing a Program" and in Part II under the

heading "Elements of BASIC Statements."

Action: Re-enter the corrected statement.

640 Short: EXTRA DEC PT+
Long: TOO MANY DECIMAL POINTS IN NUMBER
Explanation: In ITF:BASIC, only decimal fixed-point and decimal floating-point numbers may contain a decimal point. In both cases (F-format and E-format), the decimal point is optional, but no more than one decimal point can be specified.
Action: If this message appeared when you were creating the program, correct the statement and continue. If, however, the message appeared after you issued a RUN or BASIC command, or a RUN subcommand, you must correct the statement and re-execute the program.

*641 Short: BIG TAN-COT+
Long: TANGENT OR COTANGENT ARGUMENT EXCEEDS MAXIMUM ALLOWED
Explanation: You have supplied an argument to the TAN or COT intrinsic function which has a value that exceeds $\pi * 2^{18}$ for short-form arithmetic or $\pi * 2^{50}$ for long-form arithmetic.
Action: Check the logic of your program, make the necessary corrections, and re-execute the program.

642 Short: INV CHAR CON+
Long: INVALID CHARACTER CONSTANT
Explanation: A character constant is one or more characters enclosed by a pair of single or double quotation marks. Check to see that your character constant is enclosed in quotation marks. If your character constant is to contain a quotation mark, make certain you are following the rules given in Part II under the heading "Elements of BASIC Statements."
Action: If the message appeared as you were creating the program, correct the statement and continue. If, however, the message appeared after you issued a RUN or BASIC command, or a RUN subcommand, you must correct the statement and re-execute the program.

*643 Short: UNDEF TAN/COT+
Long: TANGENT OR COTANGENT APPROACHES INFINITY
Explanation: This message is given when you ask for one of the following:

$TAN(90^\circ \pm n * 180^\circ)$
$COT(\pm n * 180^\circ)$

where _n_ is an integer.
Action: Check your program logic, correct the necessary statement(s), and re-execute the program.

644 Short: BLANK STMT+
Long: BLANK STATEMENT OR NO OPERANDS FOLLOWING KEYWORD THAT REQUIRES OPERANDS IS INVALID
Explanation: In an ITF:BASIC program, a statement number with only blanks following it is invalid. This message is also given when you supply a statement keyword without its required operands.
Action: Re-enter the entire statement making the necessary corrections.

*645 Short: EXTRA ,+
Long: EXTRA COMMA IN STATEMENT

Explanation: You have forgotten to type an identifier, or you have typed two consecutive commas.

Action: If this message appeared when you were creating the program, correct the statement and continue. If, however, the message appeared after you issued a RUN or BASIC command, or a RUN subcommand, you must correct the statement, and then re-execute the program.

646 Short: MSNG ,+

Long: COMMA MISSING FROM STATEMENT

Explanation: Perhaps you have made a typing error or you have attempted to separate items by blanks where a comma must be used. See Part II under the heading "Program Statements" to obtain the correct syntax of the statement you are using.

Action: Re-enter the corrected statement.

647 Short: INV ARRY NM+

Long: INVALID ARRAY NAME

Explanation: In ITF:BASIC, arithmetic arrays are named by a single alphabetic character (A,B,...,Z,@,#,$); character arrays are named by a single alphabetic character followed by a dollar sign (A$,B$,...,Z$, @$,#$,$$).

Action: Re-enter the statement using the correct array name.

648 Short: SUBSC > LIM+

Long: MAXIMUM ARRAY BOUND IS 255

Explanation: Neither subscript in an array reference or in an array declaration can exceed 255.

Action: If the error message appeared as you were creating the program, correct the statement and continue. If, however, the error message appeared after you issued a RUN or BASIC command, or a RUN subcommand, you must check the logic of your program, correct the statement(s) in error, and re-execute the program.

*649 Short: ONLY DELIM AND CON+

Long: THE ONLY ITEMS ACCEPTED IN THIS STATEMENT, OTHER THAN DELIMITERS, ARE VALID CONSTANTS

Explanation: Data supplied in DATA statements and data retrieved from a file by means of the GET statement must be numeric, internal, or character constants, separated by commas. (Remember that character constants must be enclosed in quotation marks.)

Action: If this message appeared when you were creating the program, correct the statement and continue. If, however, the message appeared after you issued a RUN or BASIC command, or a RUN subcommand, you must correct the statement, and then re-execute the program.

650 Short: ONLY DELIM AND VAR+

Long: THE ONLY ITEMS ACCEPTED IN THIS STATEMENT, OTHER THAN DELIMITERS, ARE VALID VARIABLES

Explanation: In ITF:BASIC, input/output statements (READ, INPUT, GET, PUT, and their MAT statement counterparts) must contain a list of variables separated by commas. Check for typing and spelling errors.

Action: Re-enter the corrected statement.

651 Short: MSNG FN+

Long: "FN" MUST FOLLOW "DEF" IN THE DEFINE FUNCTION

STATEMENT

Explanation: The correct syntax of the DEF statement is:

```
DEF FNa(v) = arithmetic-expression
```

where a is an alphabetic character and v is a simple arithmetic variable. Rules governing the use of the DEF statement are given in Part II under the heading "Program Statements."

Action: Re-enter the corrected statement.

652 **Short:** INV STEP+
Long: SECOND EXPRESSION IN FOR STATEMENT MUST END THE STATEMENT OR BE FOLLOWED BY "STEP"
Explanation: The correct syntax of the FOR statement is:

FOR v = x_1 TO x_2 [STEP x_3]

where v is a simple arithmetic variable and x_i is an arithmetic expression. Rules governing the use of the FOR statement are given in Part II under the heading "Program Statements."

Action: Re-enter the corrected statement.

653 **Short:** EXTRA CHAR+
Long: EXTRA CHARACTERS AFTER LOGICAL END OF STATEMENT
Explanation: This message appears if you have typed a comment as part of a statement other than one of those statements which allows a comment (i.e., END, PAUSE, REM, RESTORE, RETURN, and STOP), or if you have typed an extra comma at the end of a list.
Action: Re-enter the corrected statement.

654 **Short:** INV FUNC+
Long: CON, ZER, IDN, TRN, AND INV FUNCTIONS MAY ONLY BE USED WITH MAT ASSIGNMENT STATEMENT
Explanation: CON, ZER, IDN, TRN, and INV are matrix functions. Consequently, they can be used only in MAT assignment statements. The syntax of the MAT statements and rules governing their use are given in Part II under the heading "Array Operations."
Action: Re-enter the corrected statement.

655 **Short:** INV ARRY NM+
Long: THE ONLY VALID OPERANDS IN THIS MAT STATEMENT ARE ARRAY NAMES
Explanation: In ITF:BASIC, arithmetic arrays are named by a single alphabetic character (A,B,...,Z,@,#,$). Only arithmetic arrays can be used in MAT statements; character arrays cannot be used in MAT statements. Check your statement for typing and spelling errors. Correct syntax for MAT statements and rules governing their use are given in PART II under the heading "Array Operations."
Action: Re-enter the corrected statement.

656 **Short:** INV MAT SYNTAX+
Long: SYNTAX OF RIGHT SIDE OF MAT SCALAR MULTIPLICATION IS: PARENTHESIZED EXPRESSION, MULTIPLICATION SIGN, AND MATRIX NAME
Explanation: The correct syntax of the MAT assignment (scalar multiplication) is:

```
MAT name-1 = (x) * name-2
```

where x is an arithmetic expression and each name is the name of an array. Make certain that your arithmetic expression precedes the multiplication sign and that it is contained in parentheses. Rules governing the use of this statement are given in Part II under the heading "Array Operations."

Action: Re-enter the corrected statement.

657 Short: INV ;+
Long: SEMI-COLON NOT VALID IN THIS CONTEXT
Explanation: The semicolon can be used as a delimiter only in the PRINT and MAT PRINT statements.
Action: Re-enter the corrected statement.

*658 Short: NEXT VAR NOT = FOR VAR+
Long: NEXT STATEMENT VARIABLE MUST MATCH THE PREVIOUS FOR STATEMENT VARIABLE
Explanation: FOR and NEXT statements must be paired and are matched when the same simple arithmetic variable is specified for each of the two statements. If you are using nested FOR/NEXT loops, make certain you are doing it correctly. Examples of correctly and incorrectly nested FOR/NEXT loops are given in Part II under the heading "Program Statements."
Action Correct the statement(s) in error and re-execute the program.

659 Short: NOT IMAGE+
Long: STATEMENT (xxxxx) REFERENCED IN A PRINT USING STATEMENT MUST BE AN IMAGE STATEMENT
Explanation: The statement number you used in the PRINT USING statement does not refer to an Image statement. If you are not certain of the statement numbers, use LIST in the edit mode to display your program. Then use the correct statement number in the PRINT USING statement. Or, if this is not the case, check for typing errors. The correct syntax of the PRINT USING and Image statements and rules governing their use are given in Part II under the heading "Program Statements."
Action: Correct the statement(s) in error and re-execute the program.

*660 Short: INV TYPE DATA+
Long: THE DATA BEING ASSIGNED MUST BE THE SAME TYPE AS THE VARIABLE RECEIVING THE DATA
Explanation: In READ and DATA statements, and in INPUT statements and their typed response, arithmetic variables must correspond to arithmetic data and character variables must correspond to character constants. The same is true of variables and values assigned in LET statements.
Action: Correct the statement(s) in error and re-execute the program, or, for response to INPUT statements, you must re-execute in order to make your corrections.

661 Short: UNDEF STMT NUM+
Long: STATEMENT NUMBER (xxxxx) REFERENCED IN A STATEMENT NOT DEFINED IN PROGRAM
Explanation: This error is caused by an erroneous statement number in one of the following statements: MAT PRINT USING, PRINT USING, GOTO, GOSUB, or IF...THEN/GOTO. If you are not certain of the statement numbers, use LIST in the edit mode to display your program. Also, check for possible

typing errors. It is also possible that you have forgotten to include the statement which the number refers to.

Action: Check the logic of your program, make the necessary corrections, and re-execute the program.

662 **Short:** UNDEF FN+
Long: FUNCTION (FNx) USED IN A STATEMENT NEVER DEFINED
Explanation: A user function must be defined in a DEF statement (see Part II under the heading "Program Statements"). A function may be defined anywhere in the program (before or after its use), but it must be defined.
Action: Check the logic of your program, make the necessary corrections, and re-execute the program.

*663 **Short:** INV FILE USE+
Long: FILES MAY NOT BE USED FOR BOTH INPUT AND OUTPUT
Explanation: If a file is to be used first as an input file and then as an output file (or vice versa) during program execution, it must be explicitly deactivated by the CLOSE statement between input and output references. The correct syntax of the CLOSE statement and rules governing its use are given in Part II under the heading "Program Statements."
Action: Check the logic of your program, make the necessary corrections, and re-execute the program.

*667 **Short:** INV ASN/ACS ARG+
Long: ARGUMENTS TO ARCSINE AND ARCOSINE MUST LIE BETWEEN +1 AND -1
Explanation: You have supplied an argument to the ASN or ACS intrinsic function which has a value outside the range +1 to -1.
Action: Check the logic of your program, correct the statement(s) in error, and re-execute the program.

*668 **Short:** MSNG DATA+
Long: NOT ENOUGH DATA ITEMS FOR READ OR GET
Explanation: This message is issued when (1) there is insufficient data in the specified file to satisfy the number of variables in the associated GET statement, or (2) when you have supplied fewer values in DATA statements than the number of variables in the associated READ statement. You may have intended to do this to end program execution when the input data was exhausted. If this is the case, continue in the edit mode. If, however, the error was unintentional, you must supply more values, or remove the excess variables in order to execute successfully.
Action: If the error was intentional, continue in the edit mode. If the error was unintentional, correct the statement(s) in error and re-execute the program.

669 **Short:** INV FILE REF+
Long: INVALID FILE NAME REFERENCE
Explanation: In ITF:BASIC, a file name is a character constant of any length, but it cannot be a null character string (two adjacent quotation marks). The first three characters of the file name cannot contain a period, a comma, or a semicolon. A blank cannot precede a nonblank in the first three characters nor can the first three characters be all blank.
Action: Re-enter the statement using a valid file name.

*670 Short: NO ACT GOSUB+
Long: RETURN STATEMENT FOUND WITH NO ACTIVE GOSUB
Explanation: Execution of a GOSUB statement must precede that of its corresponding RETURN statement (that is, the GOSUB statement must be made active before its RETURN statement is executed). For rules governing the use of GOSUB and RETURN statements, see Part II under the heading "Program Statements."
Action: Check the logic of your program, correct the statement(s) in error, and re-execute the program.

671 Short: INV PRNT FLD+
Long: INVALID PRINT FIELD IN PRINT STATEMENT
Explanation: A print field may not begin with a right parenthesis, an asterisk, a slash, a double asterisk, or any relational operator.
Action: Re-enter the corrected statement.

672 Short: NON ARITH SUBSC+
Long: SUBSCRIPT EXPRESSIONS MUST BE ARITHMETIC
Explanation: Subscripts for character and arithmetic arrays must be arithmetic expressions whose values are positive and whose truncated integer portions are within the range 1-255. The arithmetic expression may be an arithmetic variable, a subscripted arithmetic array reference, a numeric constant, a function reference, or a combination of the above separated by binary operators and parentheses.
Action: Re-enter the statement using arithmetic expressions as subscripts.

*673 Short: NON SQR MAT+
Long: ARRAY USED WITH THE IDN FUNCTION, MUST BE A SQUARE MATRIX (2 DIMENSIONS)
Explanation: The array which is to assume the form of the identity matrix must be two-dimensional and the values of the two bounds must be equal.
Action: Check the logic of your program, correct the necessary statement(s), and re-execute the program.

*674 Short: REDIM < 1+
Long: THE REDIMENSIONING SPECIFIED A BOUND < 1
Explanation: Zero and negative values are not valid subscripts in ITF:BASIC. Subscripts must have positive integer values.
Action: Check the logic of your program, correct the necessary statement(s), and re-execute the program.

*675 Short: REDIM > DIM+
Long: THE REDIMENSIONING SPECIFIED MORE ELEMENTS THAN THE ORIGINAL ARRAY HAD
Explanation: An array may be redimensioned as long as the original number of dimensions is not changed and the total number of members is not exceeded.
Action: Check the logic of your program, correct the necessary statement(s), and re-execute the program.

676 Short: CHAR ARRY INV+
Long: A CHARACTER ARRAY MAY NOT BE USED IN THIS STATEMENT
Explanation: In ITF:BASIC, character arrays may not be used in any MAT statement.
Action: Re-enter the corrected statement.

677 Short: SIGNED OPND+
Long: THE OPERANDS IN THIS STATEMENT MUST BE UNSIGNED

Explanation: You have placed a plus or minus sign in front of an array name used as an operand in a MAT statement. This is not permitted. The correct syntax of the MAT statements and rules governing their use are given in Part II under the heading "Array Operations."

Action: Re-enter the corrected statement.

*678 Short: UNDEF SEC/CSC+

Long: SECANT OR COSECANT APPROACHES INFINITY

Explanation: This message is given when you ask for one of the following:

```
SEC(90° ± n *180°)
CSC(± n *180°)
```

where n is an integer.

Action: Check your program logic, correct the necessary statement(s), and re-execute your program.

*679 Short: DIM < 2+

Long: ARRAY IN MATRIX MULTIPLICATION MUST HAVE 2 DIMENSIONS

Explanation: The correct syntax of the MAT assignment statement performing matrix multiplication is:

```
MAT name-1 = name-2 * name-3
```

where each name is the name of a two-dimensional array. Rules governing the use of this statement are given in Part II under the heading "Array Operations."

Action: Correct the statement(s) in error and re-execute the program.

*680 Short: MAT NOT CNF+

Long: COLUMNS AND ROWS OF MATRICES MUST CONFORM TO RULES FOR MATRIX MULTIPLICATION

Explanation: The correct syntax of the MAT assignment statement performing matrix multiplication and the rules governing its use are given in Part II under the heading "Array Operations."

Action: Check the logic of your program, correct the statement(s) in error, and re-execute the program.

*686 Short: NO INVERSE+

Long: THE INVERSE DOES NOT EXIST FOR THIS MATRIX (DETERMINANT = 0)

Explanation: Not every two-dimensional array has an inverse; the inverse of array A exists if DET(A) ≠ 0. It is a good practice to use the DET function (see Part II under the heading "Intrinsic Functions") to verify that an inverse exists before attempting to use the MAT assignment statement performing matrix inversion.

Action: Check the logic of your program, correct the statement(s) in error, and re-enter the statement.

687 Short: INV DET SYNTAX+

Long: SYNTAX OF DET FUNCTION IS DET (LETTER)

Explanation: The correct syntax of the DET function is:

```
DET (arithmetic-array-name)
```

The quantity in parentheses immediately following the word DET is an argument. In this case, the argument must be the name of a square arithmetic array.

Action: Re-enter the corrected statement.

Index

- (blank)
  - as a special character 83
- .(period) 83
- <(less than)
  - as a relational operator 36, 88
  - as a special character 83
  - Correspondence equivalent 83
- <>(less than or greater than)
  - as a relational operator 36, 88
  - as special characters 83
  - Correspondence equivalent 83
- <=(less than or equal to)
  - as a relational operator 36, 88
  - as special characters 83
  - Correspondence equivalent 83
- +(plus sign)
  - as a binary arithmetic operator 87, 34
  - as a special character 83
  - as a unary arithmetic operator 87, 34
  - at end of message 71, 72
  - in array operations 107-108, 44
  - in exponential format
    - as input 45
    - as output 84
- |(vertical bar)
  - Correspondence equivalent 83
  - Teletype equivalent 83
  - (*see also* vertical bar)
- & (ampersand) 83
- &E internal constant 85, 29
- &PI internal constant 85, 29
- &SQR2 internal constant 85, 29
- *(asterisk or multiply sign) (*see* asterisk)
- **(exponentiation sign)
  - as a binary arithmetic operator 87, 34
  - as special characters 83
  - Teletype equivalent 83
- ;(semicolon)
  - as a delimiter (*see* PRINT statement; MAT PRINT statement)
  - as a special character 83
- –(minus sign or hyphen)
  - as a binary arithmetic operator 87, 34
  - as a special character 83
  - as a unary arithmetic operator 87, 34
  - in array operations 107-108, 44
  - in exponential format
    - as input 45
    - as output 84
- /(slash or division sign)
  - as a binary arithmetic operator 87, 34
  - as a special character 83
- mma)
  - a delimiter (*see* PRINT statement; MAT PRINT statement)
  - ial character 83
- in)
  - tional operator 36, 88
  - ial character 83
  - ndence equivalent 83
- >=(greater than or equal to)
  - as a relational operator 36, 88
  - as special characters 83
  - Correspondence equivalent 83
- ?(question mark)
  - as a system request (*see* INPUT statement; MAT INPUT statement)
  - as a user request 71, 72
  - as a special character 83
- !(exclamation mark)
  - as a special character 83
  - as an equivalent for | 83
  - in Image statement 95, 102
- :(colon)
  - as a special character 83
  - in an Image statement 38, 95
- # character (*see* Image statement)
- ((left parenthesis) 83
-)(right parenthesis) 83
- '(single quotation mark or apostrophe)
  - as a special character 83
  - used with character constants 29, 85
  - used with file names 49-50
- =(equal sign)
  - as an assignment symbol (*see* LET statement)
  - as a relational operator 36, 88
  - as a special character 83
- "(double quotation mark)
  - as a special character 83
  - used with character constants 29, 85
  - used with file names 49-50
- [(left bracket)
  - as an equivalent for < 83, 88
-](right bracket)
  - as an equivalent for > 83, 88
- ≤(less than or equal to)
  - as a relational operator 36, 88
  - as a special character 83
- ≥(greater than or equal to)
  - as a relational operator 36, 88
  - as a special character 83
- ±(plus or minus sign)
  - as an equivalent for | 83
- ↑(up-arrow)
  - as a special character 83
  - as an equivalent for ** 87, 34

A

- abbreviated error messages 72, 161
- abbreviations, of commands and subcommands 124
- ABS intrinsic function 119
- absolute value intrinsic function 119
- absolute value of numbers 119
- account number 16
- accuracy of the matrix inversion function 109
- ACS intrinsic function 119
- activating files 52-53
- active GOSUB statements 93, 106
- adding statements to end of program 62-63, 67

addition
  as a binary arithmetic operation 34, 87
  in arrays 107-108, 44
  special cases 87
ALL operand
  in CHANGE subcommand 63-64, 126
    abbreviation 127
  in HELP command 131
    abbreviation 131
ALLOCATE command 148
alphabet, extended 18, 83
alphabetic characters 18, 83
  definition of 153
alphabetic extenders 18, 83
  definition of 153
alphameric character 18, 153
apostrophe (*see* quotation mark)
arccosine intrinsic function 119
arcsine intrinsic function 119
arctangent intrinsic function 119
argument
  definition of 153
  description of 35, 119
  in intrinsic functions 119
  in user-written functions 57
arithmetic
  expressions and calculations 33-36
  long-form 84, 45-47
  short-form 83-84, 45-47
arithmetic arrays
  assigning values to 42-43
  declaration of 86, 40-42
    explicit 40, 86
      (*see also* DIM statement)
    implicit 40, 86
  definition of 153
  initial value of 42, 85
  naming of 85, 40
arithmetic expressions
  conversion for printing 103
  definition of 153
  description of 33, 86-88
  evaluation of 34-35, 86
  precision
    in format specifications 95
    loss of 47, 119
  printing of
    Image statement 95-96
    MAT PRINT 113-115
    MAT PRINT USING 115-116
    PRINT 99-102
    PRINT USING 102-104
arithmetic operators
  (*see also* binary and unary operators)
  definition of 153
  priority of 87, 34-35
  special cases of 87
arithmetic values
  examples of printed format 100, 104
arithmetic variables 30, 85
  definition of 153
array
  arithmetic 85-86, 40-42
  assigning values to an 42-43
  bounds 41, 86
  character 86, 42
  declarations of
    explicit 86, 40-41
      (*see also* DIM statement)
    implicit 86, 40-41
  definition of 153
  dimension 40-42, 85-86
  expression 153, 88
  initial value of
    arithmetic 85, 42
    character 86, 42
  member 40, 85-86
    definition of 153
  multiplication 110, 44
  one-dimensional 41
  operations (*see* matrix operations)
  redimensioning 44, 107
  scalar multiplication 110, 44
  square 109
  two-dimensional 41-42
  variables 85-86, 153
ASN intrinsic function 119
assigning values
  to arrays
    with a FOR/NEXT loop 43
    with an INPUT statement 42-43
    with MAT input/output statements 43-44
  to variables
    arithmetic 30-33
      in test mode 77, 98
    character 31
assignment 153
assignment statement (*see* LET statement; MAT assignment statement)
asterisk (*)
  as a binary arithmetic operator 34, 87
  as a unary arithmetic operator 34, 87
  in array operations 44, 110
  in the NOTRACE subcommand 133, 76
  in the TRACE subcommand 137, 75-76
asterisks edited into a print field 103
AT subcommand
  description 74-75
  reference information 125
"at" sign (*see* @)
ATN intrinsic function 119
attention interruption 16-17
  definition of 153
  summary table 145
  to cancel execution
    in command mode 27, 145
    in edit mode 24, 145
  to cancel listing 66
  to end input phase 20, 145
  to end mode
    edit 24, 145
    test 73
  to interrupt test mode execution 74
  vs. line deletion 18, 145
attention key 154
  (*see also* attention interruption)
ATTN key
  for attention interruption 17
  for line deletions 18
automatic statement numbering 20

B

backspace 18
BACKSPACE key 18
base 2 logarithm, function for 119
base 10 logarithm, function for 119
BASIC command
  description 27, 46-47
  reference information 124, 125-126
BASIC language 9
B.I.F. 76, 137
  (*see also* intrinsic functions)
binary array expressions 88
binary operators (*see* arithmetic operators)
blank

as a delimiter in commands 123
as a special character 83
counted as a character 18
eliminated after renumbering 135, 151
blank padding (*see* padding)
BLOCK option 127
bound
definition of 154
of a dimension 40, 86
of a loop 91
braces 141
brackets
as a syntax convention 141
as an equivalent for < and > 34, 87
branch (transfer of control)
(*see also* IF statement; GOSUB statement; GOTO statement)
into or out of a FOR/NEXT loop 92
branchpoint 154
(*see also* NOTRACE subcommand; TRACE subcommand)
BREAK key 17
(*see also* attention interruption)
breakpoint
definition of 154
placement of 77-78
set by (*see* AT subcommand)
turned off by (*see* OFF subcommand)
broadcast messages 71
built-in functions (*see* intrinsic functions)

C

cancelling execution
in command mode 27, 145
in edit mode 24, 145
carriage position (*see* MAT PRINT statement; PRINT statement; PRINT USING statement)
carrier return (CR)
definition of 154
to end input phase 20, 67
to resume execution after a PAUSE statement 99
CHANGE subcommand
abbreviations in 127
description 63-65
reference information 124, 126-127
character arrays
assigning values to 42
declaration of
explicit 86
(*see also* DIM statement)
implicit 86
definition of 154
initial value of 86, 42
naming of 86, 42
restrictions 42
character comparison 88, 94
collating sequence used for 143-144
character constant
as a file name or reference 92
as an expression 35-36, 86
definition of 154
description of 85, 29
length of 85, 36
null 36
use of 30, 37
character-deletion character
definition of 154
description of 18
character expressions
definition of 154
description of 35-36, 86
length of 36
in relational expressions 94, 88
printing of 99, 101-102
use of 35-36
character-format 154, 95
character position 17
character set 18-19
character string
(*see also* comment)
definition of 154
in MAT PRINT USING statements (*see* Image statement)
in PRINT USING statements 102
character values, example of printed format 102
character variables
definition of 154
description of 85, 30
characters
alphabetic 153, 83
alphameric 153, 18
digits 155, 83
for character deletions 18
for line deletions 18
maximum per entry 17
recognized by BASIC 83
special 159, 83
characteristic of a floating-point number) 154, 45
close, explicit (*see* CLOSE statement)
CLOSE statement
description 52-53
reference information 89
collating sequence 143-144
colon
as a special character 83
in Image statements 38, 95
comma
as a delimiter
in BASIC statements (*see* MAT INPUT statement; MAT PRINT statement; INPUT statement; PRINT statement)
in commands 123
as a special character 83
command
as an entry 17
definition of 154
how to enter 124
name of 123
syntax of 123
command language 9, 123-138
command mode 25-27
commands in
BASIC 125-126, 27
CONVERT 127-128, 27
DELETE 129-130, 68-69
EDIT 130, 19-22
HELP 130-131, 25-26
LISTCAT 132, 69-70
LOGOFF 132-133, 16
LOGON 133, 15-16
RENAME 134, 66-68
RUN 135-136, 27
SEND 136-137, 26
definition of 154
execution in 27
program modification in 66-70
system cue for 16
commands
reference information 123-138
requesting help about 25-26, 130-131
summary tables 124-125
comments or remarks
as program documentation 155
definition of 155
in program statements
END 91
PAUSE 99
REM 105
RESTORE 105

RETURN 106
STOP 106
common library 149
common logarithm (base 10), function for 119
comparison operators (*see* relational operators)
compressing *userid*.DATA 147
computed GOTO 94
CON matrix function 108, 44
conditional transfer 94
(*see also* computed GOTO)
constant 29, 84-85
definition of 155
types of
character 154, 85
internal 157, 85
numeric 158, 84
uses of 29, 37
contained quotation mark 85
continuation of lines 17, 124
control
conditional transfer of 94
return of from test mode 73
tracing transfer of 75-76
unconditional transfer of 94
control mode 149
control variable (*see* loop control variable)
conventions
syntax 141
typing (used in this book) 15
CONVERT command 124, 127-128
abbreviations in 128
use of 27
converted array member, printing of 115
converted data item, printing of 99-100
converted field 99
COPY utility 147
corrections (*see* program modification; typing errors)
Correspondence Keyboards 18
equivalence for BASIC characters 83, 88
COS intrinsic function 119
cosecant intrinsic function 119
cosine intrinsic function 119
COT intrinsic function 119
cotangent intrinsic function 119
CR (*see* carrier return)
creating a file 51-52
creating a program 20-22
CSC intrinsic function 119
current file position 92

D

data 155
DATA
in LISTCAT displays 69-70
requirement for file names
(*see also* *userid*.DATA)
in DELETE command 129, 68
in RENAME command 134, 68
data file (*see* file)
data item
in an array (*see* member)
in a PRINT statement 99-101
data list
in a DATA statement 89-90
in a GET statement 89-90
in a MAT GET statement 111-112
in a MAT INPUT statement 112-113
in a MAT PUT statement 116-117
in an INPUT statement 96-97
in a PUT statement 104
DATA statement
description 31-32
reference information 89-90
data table
definition of 155
description of 89, 105
data table pointer
definition of 155
description of 89, 105
deactivating files 52-53
(*see also* CLOSE statement)
debugging 155, 72
debugging aids
subcommands 155, 73-78
diagnostic messages 161, 72
test mode 72-78
declarations, array 153
explicit 156, 40
(*see also* DIM statement)
implicit 156, 40
declaring array bounds
explicitly 86, 40
(*see also* DIM statement)
implicitly 86, 40
DEF statement
description 57
reference information 90
defining user-written functions 57
(*see also* DEF statement)
DEG intrinsic function 119
degrees, number of, intrinsic function 119
DELETE command/subcommand
abbreviation 130
description
in command mode 68-69
in edit mode 61
file name restriction in 129
recommendation for "housekeeping" 147
reference information 124, 129-130
deleting
characters 18
files 129-130, 68-69
lines or statements
via DELETE subcommand 129-130, 61
via line-delete character 18
programs 129-130, 68-69
delimiter 155
delimiter, special 63-64, 126-127
(*see also* CHANGE subcommand)
descriptive statements
DEF 90, 57
DIM 90-91, 40-42
DET intrinsic function 119, 109
determinant intrinsic function 119
diagnostic aids (*see* debugging aids)
diagnostic messages 161
dial-up terminal or mechanism 155
differences between OS ITF and TSO ITF 149-151
digits 155
significance in, precision 46-47, 119
DIM statement
description 40-42
reference information 90-91
dimension 155
dimension bound 90, 40
dimensions, array 85-86, 40-42
displaying
names in storage 69-70, 132
statements in programs 66, 132
values of variables in test mode 76-77, 132
via CHANGE subcommand 63-64, 126-127
via LIST command/subcommand 132
via LISTCAT command 69-70, 132
division
as a binary arithmetic operation 34, 87

by zero 87
special cases of 87
documentation of programs (*see* comments or remarks; REM statement)
dollar sign (*see* $)
dummy variable
definition of 155
description of 57, 90

E

E 45
(*see also* E-format)
E-format (exponential format)
external long-form representation 84
external short-form representation 83-84
in the MAT PRINT statement 114-115
in the MAT PRINT USING statement (*see* Image statement)
in the PRINT statement 100
in the PRINT USING statement (*see* Image statement)
in the test mode 76, 137
e(natural exponential)
as an internal constant 29, 85
as an intrinsic function 119
EBCDIC collating sequence 143-144
EDIT command
abbreviations 130
description 19-22
reference information 124, 130
edit mode
creating a program in 20-22
definition of 155
execution in 23-24
initiation of 19
input phase of 20-21
program modification in 61-66
subcommands in
CHANGE 63-65, 126-127
DELETE 61, 129-130
END 24-25, 130
HELP 26, 130-131
INPUT 62-63, 131
LIST 66, 132
RENUM 65, 134-135
RUN 23, 135-136
SAVE 24, 136
SCAN 22-23, 136
syntax checking in 22-23
system cue for 21
termination of 24-25
EDIT system cue 21
elements
of an array (*see* member)
of BASIC statements 83-88
ellipsis 141
end of a format specification 96
end-of-file error, avoidance of 51
end-of-file indicator 51-52, 155
END statement 14, 91
END subcommand
for ending edit mode 24-25
for ending test mode 73
reference information 124-125, 130
ending a loop 38-40
(*see also* FOR statement; NEXT statement)
ending a mode 130
ending a program 14, 91
ending a session 16
ending a subroutine 57-59
(*see also* GOSUB statement; RETURN statement)
ending lines 17-18
entry
character position of 17
continuation of 17
end of 17-18
maximum length of 17
of input values (*see* INPUT statement; MAT INPUT statement)
environment, testing (*see* test mode)
equal sign (*see* =)
error correction (*see* program modification; typing errors)
error messages
abbreviated (short form) 17, 161
definition of 156
detailed (long form) 17, 161
discussion of 17
examples of 17
list of 161
SCAN subcommand for 22-23, 136
error notification (*see* error messages)
error recognition (*see* error messages)
error recovery messages (INPUT, MAT INPUT) 161-162
errors
end-of-file, avoidance of 51-52, 155
execution 72
notification of (*see* error messages)
recognition of (*see* error messages)
semantic 72
syntax 72, 22-23
correction of in edit mode 21-23, 61-66
SCAN subcommand for determining 22-23, 136
system 161
types of 72
typing 17-18
executable statements 81
execution
cancellation of
in command mode 27
in edit mode 24
changes of sequence in (*see* IF statement; GOSUB statement; GOTO statement)
definition of 156
in command mode 27
in edit mode 23-24
in test mode 73
errors 72
interruption of in test mode 74
of programs
in command mode 27
in edit mode 23-24
in test mode 73
tracing of 75-76
EXP intrinsic function 119
explicit declaration 156, 40-41
(*see also* DIM statement)
exponent 156, 45-46
exponential format 156
(*see also* E-format)
exponential intrinsic function, natural 119
exponentiation
as a binary arithmetic operation 87, 34
definition of 156
special cases of 87
expressions
arithmetic 33-35, 86-88
conversion for printing 103
evaluation of 34-35, 86
precision
in format specifications 95
loss of 47, 119
printing of (*see* Image statement; MAT PRINT statement; MAT PRINT USING statement; PRINT statement; PRINT USING statement)
array 88, 153
character 35-36, 154
in relational expressions 94, 88
length of 36

    - printing of 99, 101-102
    - use of 35-36
  - definition of 156
  - description of 33-36
  - order of evaluation of 34-35, 86
    - changed by parenthesization 34-35
    - priority of operators in 34-35, 87
  - relational 36
    - evaluation of 88, 143-144
- Extended Binary-Coded-Decimal Interchange Code 143-144
- extenders, alphabetic 83, 153
- extents, array (*see* bounds)
- external representation of numbers
  - long form 84, 45-47
  - short form 83-84, 45-47

F

- F-format (fixed-decimal format)
  - external representation 83-84
  - in MAT PRINT 114-115
  - in MAT PRINT USING (*see* Image)
  - in PRINT 100
  - in PRINT USING (*see* Image)
- false relation 88, 36
- file
  - activation of 52-53
  - creation of 51-52
    - (*see also* PUT statement; MAT PUT statement)
  - DATA qualifier name for 68-69
    - in LISTCAT displays 69-70
  - deactivation of 52-53
    - (*see also* CLOSE statement)
  - definition of 156
  - deletion of 68-69
  - input from (*see* GET statement; MAT GET statement)
  - listing names of 69-70
  - maintenance of 147-148
  - name of 49-51
    - DATA qualifier for 68-69
    - tracing of 75-76
    - TSO-ITF compatibility 49-51
  - putting values into (*see* PUT statement; MAT PUT statement)
  - renaming of 66-68
  - repositioning of 53
  - storage of 49-50
  - tracing references to 75-76
  - usage considerations 147-148
- file name
  - (*see also* file)
  - definition of 156
  - length of 50-51
  - TSO-ITF compatibility of 49-51
- fixed-decimal format 156
  - (*see also* F-format)
- fixed-point constant 156, 84
- floating-point constant 156, 84
- floating-point data 156, 45
- FOR/NEXT loops
  - branching into or out of 92
  - initiation of 39-40, 91-92
  - multiple 98
  - nesting, examples of 92, 98
  - physical end of 39-40, 98
  - use of 39-40, 43
- FOR statement 91-92
  - increment specification 40
  - range specification 40
  - use of 39-40, 43
- format of a print line (*see* Image statement; MAT PRINT statement; MAT PRINT USING statement; PRINT statement; PRINT USING statement)
- format specification 156, 95-96
- forms of error messages 72, 161
- full print zone 156, 37
  - in MAT PRINT 114
  - in PRINT 99
- function
  - definition of 156
  - intrinsic 119, 157
  - nested 35
  - references 86
  - tracing references to 75-76
  - user-written 160, 57
- function of commands, "help" about 25-26, 130-131
- FUNCTION operand of HELP 25-26, 130-131
  - abbreviation 131
- function reference 156, 86
  - (*see also* intrinsic functions; user-written functions)
- functional differences, OS ITF VS. TSO ITF 150-151

G

- GET statement
  - description 50-55
  - reference information 92-93
- glossary 153
- GO subcommand
  - description 73, 75
  - reference information 125, 130
- GOSUB statement
  - active 93, 106
  - description 57-59
  - reference information 93
- GOTO keyword of IF statement 94, 39
- GOTO statement
  - description 38-39
  - reference information 94
- "greater than" operator (*see* >)
- "greater than or equal to" operator (*see* >=)

H

- HCS intrinsic function 119
- HELP command/subcommand
  - abbreviations in 131
  - description 25-26
  - reference information 130-131, 124-125
- "housekeeping" 147
- HSN intrinsic function 119
- HTN intrinsic function 119
- hyperbolic cosine intrinsic function 119
- hyperbolic sine intrinsic function 119
- hyperbolic tangent intrinsic function 119

I

- I-format (integer format)
  - in MAT PRINT 114
  - in MAT PRINT USING (*see* Image statement)
  - in PRINT 100
  - in PRINT USING (*see* Image statement)
  - long-form external representation 84
  - short-form external representation 83-84
- identification code
  - definition of 156
  - use in CONVERT command 128
  - use in file usage and maintenance 147-148
  - use in logging on 15-16
- identifier list
  - in NOTRACE subcommand 113
  - in TRACE subcommand 137-138
- identifiers 156, 84-86
  - constants 155, 84-85
  - function references 156, 86
    - intrinsic 157, 119
    - user-written 160, 57
  - variables 160, 85-86

array 40-44, 85-86
simple 30, 85
identity matrix 108-109, 44
identity matrix function 108-109, 44
IDN matrix function 108-109, 44
IF statement
description 39
reference information 94
image format specifications 95
Image statement
description 37-38
reference information 95-96
implicit declaration
definition of 156
description of 40
incompatibilities between TSO and ITF file names 49-51
incorrect nesting, example of 92, 98
increment
of INPUT subcommand 131-132, 62-63
of RENUM subcommand 134-135, 65
value in loops 91-92, 39-40
infinite loop 38-39, 93
informational messages 71
initial value of
arrays 85-86, 42
variables 85, 30
IN operand of CONVERT command 127-128
input
definition of 156
file 52-55
for arrays 42
from terminal (*see* INPUT statement; MAT INPUT statement)
input mode (*see* input phase)
input phase
automatic initiation of 20-21
definition of 157
line deletion in 145, 20
subcommand for 131
syntax error during 21-22
INPUT statement
description 32-33
reference information 96-97
terminal response to 96-97, 32-33
use in a loop 51-52
INPUT subcommand
for adding statements to end of program 62-63, 131
for inserting statements 62, 131
for replacing statements 62, 131
for resuming input phase 20, 131
increment in 62-63, 131
reference information 131, 124
inserting statements 62
INT intrinsic function 119
integer format (*see* I-format)
integer intrinsic function 119
Interactive Terminal Facility (*see* ITF)
internal constants
definition of 157
description of 29, 85
interruption
attention 16-17, 153
by AT subcommand 74-75, 125
by breakpoint 74-75, 125
definition of 157
in edit mode 66, 145
in test mode 74, 145
interruption, attention 153
(*see also* interruption)
summary table 145
to end input phase 20, 145
intrinsic function
definition of 157
list of 119
tracing references to 75-76, 137-138
INV matrix function 109-110, 44
inversion of matrices 109-110, 44
interative loop 157
(*see also* loop)
ITF (Interactive Terminal Facility) 9
conversion information 127-128
error messages 72, 161
"help" information about 130-131
OS vs. TSO 149-151
ITF:BASIC 9
(*see also* ITF)
ITF operand of HELP 130-131
ITF test mode (*see* test mode)

K

keyboard
BASIC special characters on 83
entries from 17-19
features 15
keyword operand 123
"help" information about 130-131
keys, special
ATTN 17-18
BACKSPACE 18
BREAK 17
LINE RESET 17
RETURN 16

L

language 9
large and small numbers 45-47
left parenthesis (*see* parentheses)
left bracket (*see* brackets)
length
of character constants 36
of file names 50
of format specifications 95
of keyboard entries 17
of messages
indicating errors 72, 161
you send 26, 136-137
of records in CONVERT command 127-128
"less than" operator (*see* <)
"less than or equal to" operator (*see* <=)
LET statement
description 31-32
reference information 97-98
letters, use of upper- and lower-case 15
LGT intrinsic function 119
library, OS ITF 149
limits, array (*see* bounds; dimensions)
line 157
(*see also* statement; entry)
line-deletion character
compared to attention interruption key 145, 18
definition of 157
use of 18
line numbers (*see* statement numbers)
LINE RESET key 17
lines, statement 81
LIST subcommand
abbreviation 132
in edit mode 66
in test mode 76-77
reference information 132, 124-125
LISTCAT command
abbreviation 132
description 69-70
reference information 132, 124

listing or display
  cancellation of 66
  of permanent storage contents 69-70, 132
  of program contents 66, 132
  of values of variables 76-77, 132
LMSG operand 72
  of BASIC command 125-126
  of RUN command/subcommand 135-136
LOG intrinsic function 119
logarithmic intrinsic functions
  to the base *e* (LOG) 119
  to the base 2 (LTW) 119
  to the base 10 (LGT) 119
log off 157
  (*see also* LOGOFF command)
log on 157
  (*see also* LOGON command)
logical end of a program 14, 91
logical records in CONVERT command 127-128
LOGOFF command
  description 16
  reference information 132-133, 124
LOGON command
  description 15-16
  reference information 133, 124
LOGON operand of SEND command 136-137, 26
log-on procedure 16, 133
long form of error message 72, 161
long-form arithmetic
  definition of 157
  external representation of 84
  in MAT PRINT 114-115
  in PRINT 100
  specification for (LPREC) 45-47
    in BASIC command 125-126
    in RUN command/subcommand 135-136
  use of 45-47
long precision (*see* long-form arithmetic)
loop
  bounds of 91
  debugging of 77
  definition of 157
  FOR and NEXT 39-40, 43
  infinite 38, 93
  multiple 98
  nested 92, 98
  with GET and LET 54-55
  with INPUT and ? 51-52
  with READ and DATA 43, 58
loop control variable 91-92
loop range specification 40, 91-92
lower-case letters
  as a convention in this publication 15
  as a syntax convention 141
  definition of 157
  in listing or displaying programs 66
LPREC operand
  of BASIC command 125-126
  of RUN command/subcommand 135-136
  use of 45-47
LRECL operand of CONVERT command 127-128
LTW intrinsic function 119

M

magnitude
  of a numeric constant 85
  of a variable 85
maintenance, file 147
mantissa
  definition of 157
  size allowed in MAT PRINT 114-115
  size allowed in PRINT 100
margin setting 17
MAT assignment statements 107-111, 44
  addition and subtraction 107, 44
  CON (unity) function 108, 44
  IDN (identity) function 108-109, 44
  INV (inversion) function 109-110, 44
  multiplication 110, 44
  scalar multiplication 110, 44
  simple 107, 44
  TRN (transpose) function 111, 44
  ZER (zero) function 111, 44
MAT GET statement 111-112
MAT INPUT statement 112-113
MAT PRINT statement 113-114
MAT PRINT USING statement 115-116
MAT PUT statement 116-117
MAT READ statement 117
MAT statements (array operations)
  description 43-44
  reference information 107-117
matrix 157
matrix addition and subtraction 107, 44
matrix assignment, simple 107, 44
matrix functions
  identity (IDN) 108-109, 44
  inversion (INV) 109-110, 44
  transpose (TRN) 111, 44
  unity (CON) 108, 44
  zero (ZER) 111, 44
matrix multiplication 110, 44
matrix scalar multiplication 110, 44
member (of an array) 157
member (of DATA) 69-70, 147
MEMBERS operand of LISTCAT 132, 69-70
MERGE command, OS ITF 150
messages
  broadcast 71
  command for sending 136-137, 26
  informational 71
  ITF error 72, 161
  levels of 72, 161
  list of 161
  mode (*see* system cue)
  numbers of 72
  prompting 71
  recovery (*see* INPUT statement; MAT INPUT statement)
  types of 70
minus sign (*see* —)
mode
  definition of 157
  message (*see* system cue)
  types of
    command 25-27, 66-70
    edit 19-25, 61-66
    test 72-78, 27
  usage 19-27
Model 33 Teletype BASIC character equivalents 83
Model 35 Teletype BASIC character equivalents 83
modification of programs
  in the command mode 66-70
  in the edit mode 61-66
monitoring program execution 75-76
multiple LET 30-31, 97-98
multiple loops 98
multiplication
  as an arithmetic operation 34, 87
  in arrays 44, 88
    matrix 110, 88
    scalar, matrix 110, 88
  rounding errors 87
  special cases 87
multiply symbol (*see* asterisk)

N

names
  of arrays 85-86
  of commands and subcommands 124-125
  of files 49-51, 68-69
    TSO-ITF compatibility considerations 49-51
  of functions
    intrinsic 119
    user-written 57
  of internal constants 21, 85
  of programs 158
    in EDIT command 19-20
    in SAVE subcommand 24
  of variables 30, 85
named tables (*see* array)
natural exponential
  internal constant 29, 85
  intrinsic function 119
natural logarithm, intrinsic function for 119
nested loops 92
nesting
  definition of 157
  FOR and NEXT statements 92, 98
  function references 35
  GOSUB and RETURN statements 93
  parenthesized expressions 35
  subroutines 93
NEXT statement
  description 39-40
  reference information 98
NEW operand of EDIT command 130
new programs
  creating 20-22
  saving 24
nonexecutable statements 81
NOSCAN operand of EDIT command 130
"not equal to" operator (*see* <>)
NOTEST operand
  of BASIC command 125-126
  of RUN command/subcommand 135-136
notification, syntax error (*see* error messages; SCAN subcommand)
NOW operand of SEND command 136-137
null character constant (*see* null character string)
null character string 157
null delimiter 158
  in MAT PRINT 113
  in PRINT 99-102
null line 20
nullifying breakpoints (*see* OFF subcommand)
nullifying traces (*see* TRACE subcommand)
number of degrees intrinsic function 119
number of radians intrinsic function 119
number sign (*see* Image statement; special characters)
numbering statements (*see* statement; statement numbers)
number(s)
  finding absolute value of 119
  large 45-47
  random 119
  small 45-47
  statement 81, 9
numeric constant
  definition 158
  description 84-85, 29
numerical analysis considerations 109

O

OFF operand of SCAN subcommand 136
OFF subcommand
  description 74
  reference information 133-134
OLD operand of EDIT command 130
old program 19
ON operand of SCAN subcommand 136
one-dimensional array 41, 85-86
operands of commands
  keyword 123
  positional 123
OPERANDS operand of HELP command/subcommand 130-131
operation
  matrix (array) 107-117
  terminal 15-17
operator 158
  arithmetic 34, 86-87
  relational 36, 88
"or" sign
  as a special character 83
  as a syntax convention 141
  in an Image format specification 95
OS ITF
  compared to TSO ITF 149-151
  conversion information 127-128
output (*see* input)
output file, use of 51-55

P

packed print zone 158, 37
  in MAT PRINT 114
  in PRINT 99
padding
  with blanks 95, 102
  with zeros 95, 103
parentheses
  as special characters 83
  in expressions 35
  in functions 119, 57
  in lists (*see* CONVERT command; LIST subcommand; NOTRACE subcommand; TRACE subcommand)
  in subscripts 86
partitioned data set in CONVERT command 128
password 15
PAUSE statement 99
percent symbol (*see* %)
period 83
permanent storage
  for files 49, 147
  displaying names of items in 69-70, 132
physical organization of a file 148, 127-128
physical records in CONVERT command 127-128
pi internal constant 85, 29
"plus or minus" character (*see* ±)
plus sign
  as a special character 83
  as an operator 87, 34
  at end of message 71, 72
  in exponential format 45, 84
position, character 17
pound sign (*see* Image statement; special characters)
precision
  (*see also* long-form arithmetic; short-form arithmetic)
  definition of 158
  loss of 47, 119
prefix operators (*see* unary operators)
print line 37
print position 37, 101
  (*see also* Image statement; MAT PRINT statement; MAT PRINT USING statement; PRINT statement; PRINT USING statement)
PRINT statement
  description of 36-88
  reference information 99-102
PRINT USING statement
  description of 37-38
  reference information 102-104
print zones 158

full 37, 156
packed 37, 158
printing of arithmetic expression values
carriage position table for PRINT 101
conversion table for PRINT USING 103
examples 100, 102-104
in MAT PRINT 113-115
in MAT PRINT USING 115-116
(*see also* Image statement)
in PRINT 99-102
in PRINT USING 102-104
(*see also* Image statement)
loss of precision 47, 119
printing of character expression values
examples 102
in PRINT 99-102
in PRINT USING 102-104
(*see also* Image statement)
printing results 36-38
priority of operators 87-88, 34
private library, OS ITF 149
problem solving 9
PROC operand in LOGON command 133
procedure
log-on 16, 133
terminal operating 15-17
PROFILE command 18
program
BASIC 81
conversion of 127-128
creation of 20-22
debugging of 72-78
definition of 158
deletion of 68-69, 129-130
documentation of (*see* comments or remarks)
editing (*see* program modification)
execution of
in command mode 27
in edit mode 23-24
in test mode 73
name of 158, 19-20
modification of 61-70
renaming of 66-68, 134
requirements of 81
saving of 24, 136
structure of 81
termination of 14, 91
testing of 72-78
updating contents of (*see* program modification)
program input 31
(*see also* DATA statement; READ statement)
program modification
in the command mode 66-70
in the edit mode 61-66
program name 158, 19-20
program product 9
program statements 89-106
(*see also* MAT statements)
program testing 72-78
programmer-defined function (*see* user-written function)
programmer-written function (*see* user-written function)
programming language 9
prompting messages 71
PTTC/EBCD keyboard 15, 83
PUT statement
description 49-55
reference information 104

Q

qualified names
of files 147
of programs 19, 128
question mark
as a special character 83
as a system request (*see* INPUT statement; MAT INPUT statement)
as a user request 71-72, 161
quotation marks
contained 85
double 83
restriction 85
single 83
use with character constants 29, 85
use with file names
in ITF 49-51
in TSO 66-68, 49-51
quoted string of characters (*see* character constant; character string; file name)

R

radians, number of, intrinsic function 119
RAD intrinsic function 119
random number intrinsic function 119
range specification 40, 91-92
READ statement
description 31-33
reference information 104-105
READY system cue 16
recognition of errors (*see* error messages)
record length
for CONVERT command 127-128
for files 148
recovery messages (*see* INPUT statement; MAT INPUT statement)
recursive GOSUB loops 93
redimensioning 158, 107
references
array (*see* array)
function 86
file (*see* file name)
relational expressions 36, 88
relational operators 158
various character representations of 36, 88
REM statement 105
remarks (*see* comments)
remote terminal (*see* terminal)
RENAME command
abbreviation 134
description 66-68
file name restriction in 68
reference information 134
renaming 66-68, 134
RENUM subcommand
abbreviation 135
description 65
OS ITF vs. TSO ITF 151
reference information 134-135
renumbering statements 65
repeated execution (*see* loops)
replacing statements 62
repositioning files 53
RESET statement
description 53
reference information 105
resuming execution, test mode 73
RESTORE statement
description 32
reference information 105
RETURN key 16
return linkage 93
(*see also* GOSUB statement)
return of control, test mode 73
RETURN statement
description 57-58
reference information 106

right parenthesis (*see* parentheses)
RND intrinsic function 119
rounding
  errors 87
  of long-form values 84
  of short-form values 83-84
run 158
  (*see also* execution)
RUN command/subcommand
  description
    in command mode 27
    in edit mode 23-24
  reference information 135-136
  test mode, initiated by 72-73

S

SAVE subcommand
  description 24
  reference information 136
saving
  files 49, 147
  programs 24, 136
scalar 158
  expression 158, 86
  reference 158, 86
SCAN operand of EDIT command 130, 22-23
  compared to SCAN subcommand 136
SCAN subcommand
  description 23
  reference information 136
  vs. SCAN/NOSCAN operands of EDIT command 136
scientific notation 45
  (*see also* E-format)
SEC intrinsic function 119
secant intrinsic function 119
semantic error 159, 72
semicolon
  as a delimiter (*see* MAT PRINT statement; PRINT statement)
  as a special character 83
SEND command
  abbreviations in 137
  description 26
  OS ITF VS. TSO ITF 151
  reference information 136-137
sending messages 26, 136-137
sequence, change in execution (*see* branch)
sequential organization of files 148
session
  definition 159
  ending of 16, 132-133
  example of mode use during a 20-21
  starting of 15-16, 133
set, character 18-19
SGN intrinsic function 119
short-form arithmetic
  definition 159
  external representation 83-84
  in Image format specification 95
  in MAT PRINT statement 114-115
  in PRINT statement 100
  loss of precision, using 47
  specification for (SPREC) 45-47
    in BASIC command 125-126
    in RUN command/subcommand 135-136
  use of 45-47
short form of error message 72, 161
short precision (*see* short-form arithmetic)
sign in format specification 103
sign intrinsic function 119
significant digits
  definition 159
  in E-format 84
  loss of 47, 119
simple arithmetic variable 159, 85
simple assignment statement 77-78, 97-98
simple character variable 159, 85
simple GOTO statement 94, 38-39
simple LET statement 97-98, 31-32
simple variables 159, 85
simulated attention key 17
SIN intrinsic function 119
sine intrinsic function 119
size
  of files 51-52
  of print field for converted data item 99
slash
  as a binary arithmetic operator 87, 34
  as a special character 83
small numbers 45-47
SMSG operand 72
  of BASIC command 125-126
  of RUN command/subcommand 135-136
special characters 159, 83
special delimiter 63-64, 126-127
splitting lines on entries 17, 124
SPREC operand 45-47
  of BASIC command 125-126
  of RUN command/subcommand 135-136
SQR intrinsic function 119
square arrays 109
square-root intrinsic function 119
square root of 2 internal constant 29, 85
standard increment
  for INPUT subcommand 131
  for input phase 20
  for RENUM subcommand 134-135
start of a format specification 95-96
start of terminal operation 15-16
statement lines 159, 81
statement number sequence 9, 81
statement numbers
  definition 159
  OS ITF VS. TSO ITF 151
  system-supplied 20-21, 124
  user-supplied 20-21, 67
statements (*see* program statements; MAT statements)
statements, BASIC 81
status word (*see* system cue)
STEP keyword of FOR statement 40, 91-92
STOP statement 106
storage, permanent 158
subcommands
  definition 159
  entry information 124
  "help" information about 25-26, 130-131
  of edit mode 124
    discussion 22-26, 61-66
  of test mode 125, 73-78
  summary tables 124-125
subroutine 159, 57-59
  (*see also* GOSUB statement; RETURN statement)
subroutine nesting, example of 93
subscript 159, 40-41
subscripted array name 86
subtraction
  as a binary arithmetic operation 34, 87
  in arrays 107-108
  special cases 87
suppression of short messages 72
  (*see also* LMSG operand)
syntactical unit 141
syntax checking
  definition 159
  in edit mode 22-23
  OS ITF VS. TSO ITF 150

syntax conventions 141
syntax errors 159, 72
  controlling messages for 22-23
  effect on input phase 21-22
syntax notation (*see* syntax conventions)
syntax of commands 123-124
  "help" information about 26, 130-131
SYNTAX operand of HELP command/subcommand 130-131, 26
System/360 Operating System 9
system command (*see* command)
system cue 159
  EDIT (edit mode) 21
  READY (command mode) 16
  TEST (test mode) 73
system errors 161
system operator, messages for 26, 136-137
system-supplied constants (*see* internal constants)

T

TAN intrinsic function 119
tangent intrinsic function 119
Teletype terminals
  attention interruption key 17
  equivalents for BASIC characters on 83
terminal
  connection of 15-16
  definition of 159
  dial-up 155
  entries from 17-19
  equivalents for BASIC characters on 83
  operation of 15-17
  TSO supported 15
terminal-oriented input 33
  (*see also* INPUT statement; MAT INPUT statement)
terminal session 16
terminal user (*see* user)
termination
  due to end-of-file 51
  of edit mode 24-25
  of test mode 73
  statement for (STOP) 106
  subcommand for (END) 130
terminology, OS ITF vs. TSO ITF 149
test mode
  assignment statement in 77-78
  definition of 159
  execution in 73-74
  initiation of 72
  interrupting execution in 74
  OS ITF vs. TSO ITF 149-150
  subcommands in 73, 125
    AT 74-75, 125
    END 73, 130
    GO 73, 130
    HELP 73, 130-131
    LIST 76-77, 132
    NOTRACE 75-76, 133
    OFF 74-75, 133-134
    TRACE 75-76, 137-138
  system cue for 73
  termination of 73
  tracing execution in 75-76
  use of attention key in 73
TEST operand 72
  of BASIC command 125-126
  of RUN command/subcommand 135-136
test submode, OS 149-150
TEST system cue 73
testing environment (*see* test mode)
text collection, OS 149
text handling 15
  OS ITF vs. TSO ITF 150
THEN keyword of IF statement 39, 94
time-sharing 9
Time Sharing Option 9
TO keyword of FOR statement 39-40, 91-92
trace (*see* TRACE subcommand)
TRACE subcommand
  description 75-76
  reference information 137-138
transfer of control
  (*see also* GOSUB statement; GOTO statement; IF statement)
  into or out of a FOR/NEXT loop 92
transpose matrix function 111, 44
true relation 88, 36
truncation
  of character constants 36
  of excess digits 84
TSO 9
TSO ITF:BASIC 9
  compared to OS ITF:BASIC 149-151
  conversion information 127-128
TSO terminals 15
two-dimensional array 41-42
typing conventions 17-19
typing errors, correction of 17-18

U

unary array expressions 88
unary operators
  definition of 159
  description of 87
unconditional transfer (*see* simple GOTO statement)
unity matrix function 88, 44
updating program contents (*see* program modification)
upper-case letters
  as a convention in this publication 15
  as a syntax convention 141
  definition of 159
  in program listings or displays 69-70
user 160
user-defined function 57, 90
user identification code
  definition of (identification code) 156
  use of 15-16, 133
user library (*see* private library, OS ITF)
USER operand of SEND command 136-137, 26
user-written function 160
  defining a 90
  using a 57
userid (*see* user identification code)
userid.DATA 147-148
using files 54-55
using the terminal 15-17

V

values
  as input for arrays 42-43
  as input for files 51-52
  assignment of
    using DATA and READ statements 31-32
    using INPUT statement 33
    using LET statement 30-31
    using MAT INPUT statement 112-113
  assignment to variables 30-33
values of variables displayed in the test mode 76-77
variable 160
  array
    arithmetic 85-86, 40-42
    character 86, 42
  simple 85, 30-31

vertical bar
  as a special character 83
  as a syntax convention 141
  in Image format specifications 95, 102
visual differences, OS ITF vs. TSO ITF 149-150

Z

ZER matrix function 111, 44
zero-divide 87
zero-filled (*see* padding)
zero matrix function 111, 44

1052 Printer-Keyboard 17
2741 terminals
  attention interruption key 17
  entries from 17-19
  equivalents for BASIC characters on 83
9571 feature (*see* PTTC/EBCD keyboard)
9812 feature (*see* Correspondence keyboards)

READER'S COMMENTS

TITLE: IBM System/360 OS (TSO)
ITF: BASIC
Terminal User's Guide

ORDER NO. SC28-6840-0

Your comments assist us in improving the usefulness of our publications; they are an important part of the input used in preparing updates to the publications. All comments and suggestions become the property of IBM.

Please do not use this form for technical questions about the system or for requests for additional publications; this only delays the response. Instead, direct your inquiries or requests to your IBM representative or to the IBM Branch Office serving your locality.

Corrections or clarifications needed:

Page *Comment*

Please include your name and address in the space below if you wish a reply.

Thank you for your cooperation. No postage necessary if mailed in the U.S.A.

SC28-6840-0

cut along this line

fold

fold

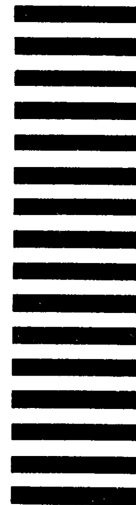
FIRST CLASS
PERMIT NO. 33504
NEW YORK, N.Y.

BUSINESS REPLY MAIL

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .

IBM CORPORATION
1271 Avenue of the Americas
New York, New York 10020



Attention: PUBLICATIONS

fold

fold

IBM S/3[illegible]DS(TSO) ITF: BASIC TUG Printed in U.S.A. SC28-6840-0

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)