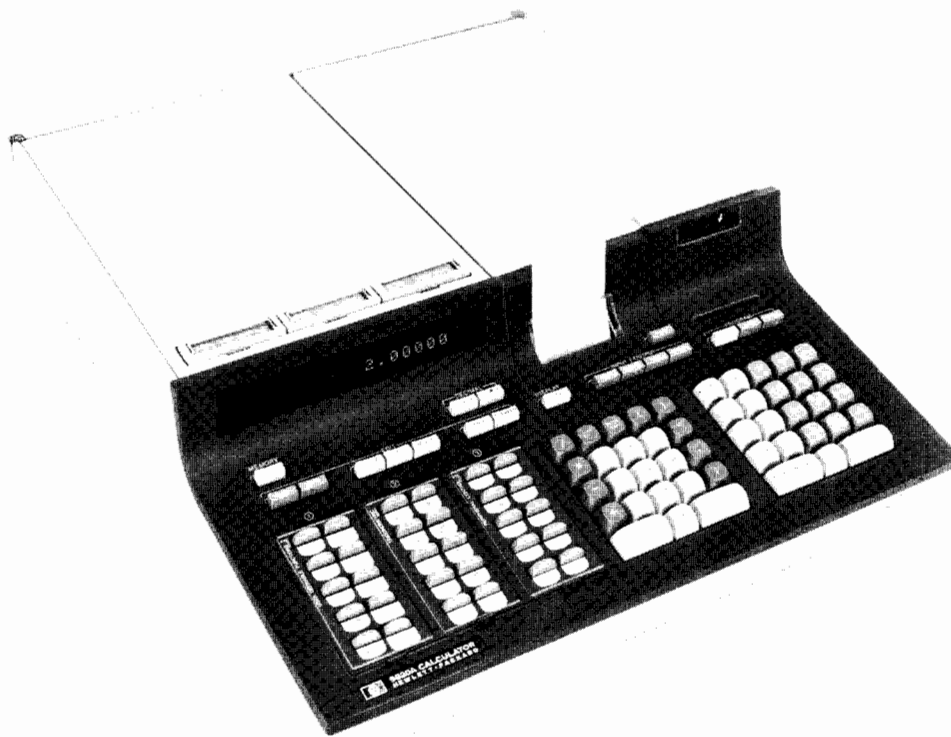




 **HEWLETT-PACKARD 9820A CALCULATOR**
OPERATING and PROGRAMMING

OPERATING AND PROGRAMMING
for the
HEWLETT-PACKARD 9820A CALCULATOR



Shown with optional ROM's installed

Copyright Hewlett-Packard Company 1972



HEWLETT-PACKARD CALCULATOR PRODUCTS DIVISION
P.O. Box 301, Loveland, Colorado 80537, Tel. (303) 667-5000
Rue du Bois-du-Lan 7, CH-1217 Meyrin 2, Geneva, Tel. (022) 41 54 00

TABLE OF CONTENTS

PREFACE:

WHEN TO READ THIS BOOK	ix
HOW TO READ THIS BOOK	ix
TERMINOLOGY	ix
FROM COMPUTERS TO THE MODEL 20	x

CHAPTERS

CHAPTER 1: GENERAL INFORMATION

MODEL 20 DESCRIPTION	1-1
OWNER'S INFORMATION	1-1
Ordering Printer Paper and Magnetic Cards	1-1
Options	1-2
The Keyboard Magazine	1-2
Pre-Written Programs	1-2
Service Contracts	1-2
Inspection Procedure	1-2
Power Requirements	1-3
Grounding Requirements	1-3
Fuses	1-3
Initial Turn-On Procedure	1-4
Loading Printer Paper	1-4
Cleaning the Calculator	1-5

CHAPTER 2: INTRODUCTION TO THE MODEL 20

A SURVEY OF THE CALCULATOR	2-1
The Keyboard	2-1
The Display	2-1
The Printer	2-2
The Magnetic Card Reader	2-2
PLUG-IN ROM'S	2-3
THE CONTROL OF PERIPHERALS	2-4
THE ALGEBRAIC LANGUAGE	2-5
A BRIEF LOOK AT LINES AND STATEMENTS	2-6
THE CONSEQUENCES OF PROGRAMMABILITY	2-7
THE MODEL 20 MEMORY	2-7
The Types of Memory	2-7
The User's RWM	2-7
Data Storage	2-8
Program Storage	2-8
RWM Structure	2-8
KEYS, AND CHARACTERS IN THE DISPLAY	2-9
REVIEW	2-11

◆◆◆◆◆◆◆◆◆◆ CHAPTERS ◆◆◆◆◆◆◆◆◆◆

CHAPTER 3: THE MECHANICS OF MODEL 20 OPERATION

THE FUNDAMENTAL PROCESS	3-1
Initializing the Calculator with Memory Erase	3-1
Writing a Line	3-1
Maximum Line Length	3-2
MODIFYING A LINE	3-3
Segments	3-3
Exact Substitution of a Segment	3-3
Removing a Segment	3-3
Adding a Segment	3-4
Replacing a Segment	3-4
EXECUTING LINES	3-4
SYNTAX ERRORS	3-5
STORING PROGRAMS	3-7
Preliminary Information	3-7
A Special Property of END	3-7
Initializing the Program Memory	3-8
Storing a Single Program	3-8
Stacking Programs	3-9
Listing Programs	3-9
MODIFYING INDIVIDUAL PROGRAM LINES	3-10
INSERTING AND DELETING PROGRAM LINES	3-11
Inserting Lines	3-11
Deleting Lines	3-11
REPLACING A PROGRAM LINE	3-12
ADDITIONAL PROPERTIES OF THE EDITING KEYS	3-12
Additional Properties of RECALL	3-12
Additional Properties of BACK and FORWARD	3-12
RUNNING A PROGRAM	3-13
ERRORS DURING EXECUTION	3-13
TRACE MODE OPERATION	3-14
Keyboard Calculation in the Trace Mode	3-14
Storing Lines While in the Trace Mode	3-14
Running Programs in the Trace Mode	3-14
REVIEW	3-15
EXERCISES	3-16

CHAPTERS

CHAPTER 4: NUMERICAL COMPUTATIONS

NUMERICAL QUANTITIES	4-1
Real Constants	4-1
Real Variables	4-2
Designating a Register	4-2
Significant Digits	4-3
MATHEMATICAL SYMBOLS	4-4
Arithmetic Operators	4-4
The Assignment Instruction	4-5
Multiple Assignment Statements	4-5
Functions	4-6
Special Conventions	4-7
Hierarchy	4-8
Parentheses	4-8
IMPLIED STORE INTO Z	4-9
The Notion of Implied Store	4-9
Ramifications of the Implied Store	4-10
RULES OF MATHEMATICAL COMBINATION	4-10
EXAMPLES	4-12
EXERCISES	4-13

CHAPTER 5: THE MODEL 20 LANGUAGE

DEFINITIONS	5-1
Constants and Variables	5-1
Quantities	5-2
Literals	5-2
Flags	5-2
Operators	5-3
Functions	5-3
Expressions	5-4
Values	5-6
Instructions	5-6
Statements	5-6
Lines	5-7
Labels	5-7
Parameters and Lists	5-8
Programs	5-8
Branching	5-8
Relative Addressing	5-9
Symbolic Addressing	5-9
Records	5-9
Subroutines	5-10

CHAPTERS

CHAPTER 5: THE MODEL 20 LANGUAGE (cont'd)

THE SYNTAX 5-11
Fixed and Float Statements 5-11
Display Statements 5-13
Print and Space Statements 5-14
Assignment Statements 5-16
Enter Statements 5-18
Absolute Go To Statements 5-22
Relative Go To Statements 5-22
Labeled Go To Statements 5-23
Go To Sub and Return Statements 5-24
Location of GTO and GSB in a Line 5-27
High Speed Branching 5-28
Jump Statements 5-29
Restrictions on Branching 5-31
Flag Statements 5-32
If Statements 5-34
Stop Statements 5-35
End Statements 5-36
Record Statements 5-37
"Secure" Record Statements 5-38
Record "Data" Statements 5-40
Keyboard Load Statements 5-41
Stored Load Statements 5-43
Normal and Trace Statements 5-44
Miscellaneous 5-45
REVIEW 5-47
EXERCISES 5-51

APPENDIX I

ANSWERS TO THE EXERCISES A-1
Chapter 3 A-1
Chapter 4 A-1
Chapter 5 A-2
THE DIAGNOSTIC NOTES A-4

TABLE OF CONTENTS

CHAPTERS

APPENDIX I (cont'd)

MODEL 20 INTERNAL STRUCTURE	A-6
The Keyboard	A-6
The Instruction Buffer	A-6
The Display	A-7
The Compiler	A-7
The Arithmetic Unit	A-7
The Note Generator	A-7
The Result Register	A-7
Execute	A-7
Store	A-7
The Uncompiler	A-8
Recall	A-8
Run Program	A-8
Other Operations	A-8
IDIOSYNCRASIES	A-10
THE PROGRAMMING FORM	A-10
RECOMMENDED ROM CONFIGURATIONS	A-12
MODEL 20 KEYBOARD	A-13

APPENDIX II: MODEL 60 CARD READER

GENERAL INFORMATION	A-15
Description	A-15
Warranty	A-15
Installation	A-15
MARKING MODEL 60 CARDS	A-16
USING THE READER	A-18
What to put on the Card	A-18
The Data Card	A-19
How the Reader Works	A-20
CHANGING THE LAMP	A-20

FIGURES

Figure 1-1. Power Cords	1-2
Figure 1-2. Switch Settings for the Various Nominal Powerline Voltages	1-3
Figure 1-3. The Rear Panel	1-3
Figure 1-4. Loading Printer Paper	1-5
Figure 1-5. Removing the Printer Window	1-5
Figure 2-1. A Representation of the Model 20 Calculator	2-0
Figure 2-2. A 6-inch Magnetic Card	2-3
Figure 2-3. Inserting Magnetic Cards	2-3
Figure 2-4. Some ROM's and their Overlays	2-4
Figure 2-5. How Peripherals are Connected	2-5
Figure 2-6. The Model 20 RWM	2-8
Figure 3-1. A Memory Map Illustrating the Assumptions in the Text	3-8
Figure 3-2. Programs Stacked in the Mainline Programming Area	3-9
Figure 3-3. A Sample Program to be Run in the Trace Mode	3-14
Figure 3-4. Results of Running the Program of Figure 3-3	3-15
Figure 4-1. The Relationship of the Special Mathematical Terms	4-7
Figure 5-1. A Program Illustrating the Use of Local Subroutines	5-25
Figure 5-2. Recording Programs	5-37
Figure 5-3. Recording Data	5-41
Figure 5-4. Loading a Single Program	5-41
Figure 5-5. Stacking Programs	5-41
Figure 5-6. Loading a New Program over an Old Program	5-42
Figure 5-7. Loading Data	5-42
Figure 5-8. Automatic Linking of Programs	5-43
Figure I-1. The Internal Structure of the Model 20	A-9
Figure I-2. The Model 20 Programming Form	A-11
Figure I-3. The Recommended Configuration for ROM's	A-12
Figure I-4. The Model 20 Keyboard	A-13
Figure II-1. Setting the Line Voltage Switch	A-15
Figure II-2. Model 9820A Key-Codes	A-16
Figure II-3. The 9820A Program and Data Card for the Model 60	A-17
Figure II-4. Inserting a Card into the Reader	A-18
Figure II-5. The Card Stop in its Extended Position	A-18
Figure II-6. The Model 20 Data Card	A-19
Figure II-7. Exposing the Lamp	A-20
Figure II-8. Removing the Spare Lamp	A-20
Figure II-9. Removing the Old Lamp	A-21

(continued)

 **TABLES** 

Table 1-1. Equipment and Accessories Supplied	1-1
Table 1-2. Powerline Voltages and Fuses	1-3
Table 2-1. Summary of Available ROM's as of early 1972	2-3
Table 2-2. Peripheral Equipment Available as of early 1972	2-5
Table 2-3. The Symbols and Mnemonics for the Keys of the Basic Model 20	2-10
Table 3-1. Condensed Information About the Notes for the Basic Model 20	3-6
Table 4-1. The Arithmetic Operators for the Basic Model 20	4-5
Table 4-2. Functions and Operations Supplied by the Math ROM	4-6
Table I-1. Diagnostic Notes	A-4
Table II-1. Accessories and Equipment Supplied with the Model 60 Card Reader	A-16



PREFACE



WHEN TO READ THIS BOOK

This manual is a comprehensive description of a sophisticated and powerful device — the Model 9820 Programmable Calculator. The functions of training, and of future reference, are of roughly equal importance in this manual. These goals are met by grouping related topics together and thoroughly examining them before moving on to the next group of topics. Thus, it is likely that you will read many pages before enough information has been covered to allow complete and meaningful activity.

For the owner of a new and exciting Calculator, such a deliberate approach is definitely not as satisfying as getting in there and actually *doing something* right away. The Simplified Operating Instructions Booklet was written to provide just such a direct initial approach to the Calculator, and is definitely a good place to start. In fact, we recommend it, even for the experienced programmer or calculator user. The deliberate, textbook approach of this manual will seem more appropriate once you are acquainted with scope and power of the machine.

HOW TO READ THIS BOOK

Chapter 1 contains the owner's information for the Model 20. Be sure to acquaint yourself with the material in that chapter.

Chapters 2 through 5 are meant to be read in order. It is most unlikely, unless you have already read the manual at least once, that you will be able to open the manual to any arbitrary point and start reading. The manual develops its own terminology, and many of the presentations are heavily based upon previously presented material.

Work the exercises at the end of the chapters — many of them were drawn from actual experiences during the learning process.

Appendix II is an operating manual for the Model 9860A Marked Card Reader, when used in a Model 20 System.

TERMINOLOGY

The Model 20 allows a flexible approach to most problems. This is a result of the variety in the operations available to the programmer. In fact, there are far too many variations for all of them to be illustrated by word or by example. Generally, however, the variations can be derived from the fundamental cases, provided certain rules are followed.

PREFACE

TERMINOLOGY (cont'd)

In order for a rule to be precise, it must be written with words that have clear-cut and definite meanings. In an explanation of an algebraic programming language, such as in Chapter 5, certain words having generally familiar (but also *specialized*) meanings are needed. We have chosen such words so that (hopefully) the familiar meaning will intuitively suggest the specialized meaning in most cases.

In Chapter 5, the necessary words are defined. Some of these words first appear in Chapter 4, however. While their use may at first appear somewhat arbitrary, they were chosen to provide an intuitively correct result on first reading, while later providing a technically correct meaning if read with the formal definitions in mind.

People seem to have the most trouble with the formal meanings of the words "value", "quantity", and "expression". Study their definitions carefully. It is unfortunate that the casual impressions created by these words are so close together, while the words have such critical differences in their assigned technical meanings.

FROM COMPUTERS TO THE MODEL 20

Persons who are familiar with other algebraic programming languages, such as FORTRAN, ALGOL, BASIC, et. al., will find many familiar concepts, many different rules, and a few surprises.

Among the notable differences are:

- a. What is called a statement in these languages is equivalent to a *line* in the Model 20's language.
- b. A *line* can be composed of more than one activity (called *statements*). This allows horizontal programming (i.e., along the *line* from left to right by *statements*) as well as the usual vertical programming (by *lines* in their natural order).
- c. Variables are subject to a different naming convention, and the numbering of *lines* is different from statement numbering in, say, FORTRAN.
- d. The Model 20's language permits several types of arithmetic assignment statements. Their use can significantly reduce the amount of programming needed in certain instances.
- e. For the most part, the terminology is similar, although in some places, different. You will still need to read the definitions.

Chapter 1

GENERAL INFORMATION

This chapter presents a general description of the Hewlett-Packard 9820A Calculator, provides owner's information, and presents some general information about calculating systems based on the Model 20 Calculator.

MODEL 20 DESCRIPTION

The Model 20 is an easy to use programmable calculator. The Model 20 language is algebraic and easy to learn, making use of the Calculator quite straightforward.

A magnetic card reader and a built-in printer add to the power and convenience of the Model 20. Both the printer and the large, easy-to-read display have full alphameric capability.

Programs and data are stored in the same general memory, so that memory not used for programming can be used for data storage. The basic Model 20 has 179 registers (with no programs stored). If the machine is equipped with option 001, it has 435 registers.

Much of the versatility of the Model 20 stems from its ability to accept as many as three plug-in ROM's at any one time (plug-in ROM's are described in the next chapter). The ROM's adapt the Model 20 to a wide variety of applications in varied disciplines by defining the half-keys to have particular meanings.

Finally, the Model 20 is the heart of a truly powerful and expandable system. Many peripherals and plug-in ROM's are available for the Model 20. The Model 20's language is especially well suited for the control of peripherals. A Model 20 System with peripherals has a great deal of overall *problem solving power*, and because of the algebraic language, this power is easy to apply.

OWNER'S INFORMATION

Table 1-1 shows the equipment and accessories supplied with the Model 20 Calculator.

ORDERING PRINTER PAPER AND MAGNETIC CARDS

The printer in the Model 20 uses a special type of paper, because of the unique way in which the printer operates. The printer heats small sections of the paper as it passes through the printer, causing those sections to change color. The sections heated correspond to a 5 X 7 matrix of dots (small squares actually), from which individual characters are generated.

Thermal printer paper is available in packs of 6 rolls, and in cases of 60 rolls.

Six-pack of thermal printing paper9281-0401-006

Case of thermal printing paper9281-0401-060

Table 1-1. Equipment and Accessories Supplied.

QTY.	ITEM	PART NO.
2	Simplified Operating Instructions Booklet	09820-90000
2	Operating and Programming Manual	09820-90001
1	Model 20 System Reference	09820-90002
1	Electrical Inspection Booklet	09820-90030
3	Rolls of Thermal Printing Paper	-----*
6	10½" Magnetic Cards	-----*
4	6" Magnetic Cards	-----*
2	8½" X 11" Magnetic Card Holder	9230-0052
1	Pad of Model 20 Programming Forms	09820-90016
1	Model 20 Math Pac, Vol. 1	09820-70000
1	ac Power Cord	-----**
1	Dust Cover	4040-0894
	Spare Fuses — all 250V, Normal Blo, ¼ in. dia. X 1¼ in. lg.	
1	1-amp	2110-0001
1	2-amp	2110-0002
1	6-amp	2110-0056

*See "ORDERING PRINTER PAPER AND MAGNETIC CARDS"

**Different power cords are shipped when requested on the order, see "OPTIONS".

Prices are available, upon request, from any HP Sales and Service Office (addresses are at the back of this manual).

OWNER'S INFORMATION

ORDERING PRINTER PAPER AND MAGNETIC CARDS (cont'd)

Magnetic cards are available in lengths of 6 inches and 10½ inches. Cards can be ordered in quantities of 10 (however, only the 6 inch cards will be in envelopes). Bulk quantities of cards can be ordered (without envelopes) in quantities of 100.

- 10 – 6-inch Cards in envelopes 5060–9152–010
- 10 – 10½-inch Cards (no envelopes) . . . 9162–0045–010
- 100 – 6-inch Cards (no envelopes) 5060–9152–100
- 100 – 10½-inch Cards (no envelopes) . . . 9162–0045–100

A handy 8½" X 11" holder for magnetic cards is also available. It will hold three 6" cards, or three 10½" cards, or both, and is punched to fit a 3-ring binder. Its part number is 9230–0052.

OPTIONS

The basic Model 20 has a maximum of 179 registers. A register is a unit of memory that can contain a data number.

A Model 20 with option 001 installed has an additional 256 registers, bringing the total to 435. If your Model 20 is equipped with option 001, it will have a decal, stating the fact, in the indentation behind the printer paper well, which is beneath the access cover (the right-hand portion of the Calculator's top cover). The decal will mention 429 registers, rather than 435; the decal refers to the maximum number of 'R registers'. Each Model 20 also has 6 'alpha registers'.

Option 001 can be installed in the field by qualified HP service personnel. The part number of the conversion kit is 11228A.

Various power cords are available for your Calculator. Their descriptions and part numbers are shown in Figure 1-1. Contact your local -hp- Sales and Service office for further information.

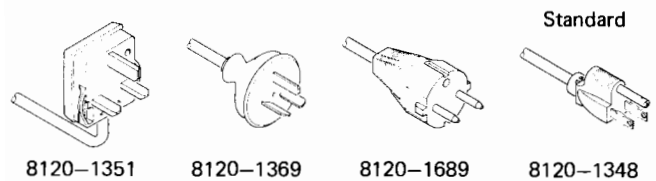


Figure 1-1. Power Cords.

THE KEYBOARD MAGAZINE

The 'KEYBOARD' is a periodic magazine containing general information about Hewlett-Packard calculators. It includes articles and programs written by users; descriptions of new equipment and of new program pacs; programming tips; and other articles of interest to calculator users.

Your purchase of a Hewlett-Packard calculator entitles you to a continuing subscription to the 'KEYBOARD'. In order to place your name on the subscription list for the 'KEYBOARD', please fill out and return the reply card enclosed with the Calculator.

PRE-WRITTEN PROGRAMS

Pre-written (and in some cases, pre-recorded) programs are available for your Model 20. Contact your local Hewlett-Packard sales representative for further details.

SERVICE CONTRACTS

Service contracts are available for the Model 20 Calculator, and for other devices in your Model 20 System. Contact any HP Sales Office for further information.

INSPECTION PROCEDURE

The inspection procedure enables you to confirm that your Model 20 is operating correctly.

Your Model 20 was carefully inspected both mechanically and electrically before it was shipped to you. It should be free of any marks and scratches or electrical malfunctions upon receipt. You should carefully inspect your Calculator for external damage caused in transit, and also check that the items listed in Table 1-1 are present. If there is any damage, file a claim with the carrier and notify HP (addresses of the Sales and Service Offices are given at the back of this manual).

To check the operation of the Model 20, perform the inspection procedure given in the Model 20 System Electrical Inspection Booklet. However, first perform the initial turn-on procedure on page 1-4, and also, load the printer with paper.

OWNER'S INFORMATION

POWER REQUIREMENTS

The Model 9820A Calculator has the following power requirements (refer to Figures 1-2 and 1-3).

LINE VOLTAGE: The Model 20 will operate from nominal powerline voltages of 100, 120, 220, or 240 ac volts. The range of operation is within -10% and $+5\%$ of those values. Two switches on the rear panel of the Calculator allow one of those four voltages to be selected as the operating voltage. (See Figures 1-2 and 1-3.)

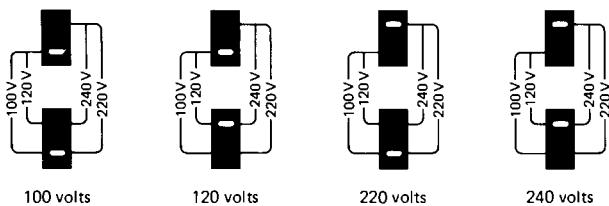


Figure 1-2. Switch Settings for the Various Nominal Powerline Voltages.

NOTE

A different fuse is required at each of the two voltage ranges of 100–120 vac and 220–240 vac. See "FUSES" for further information.

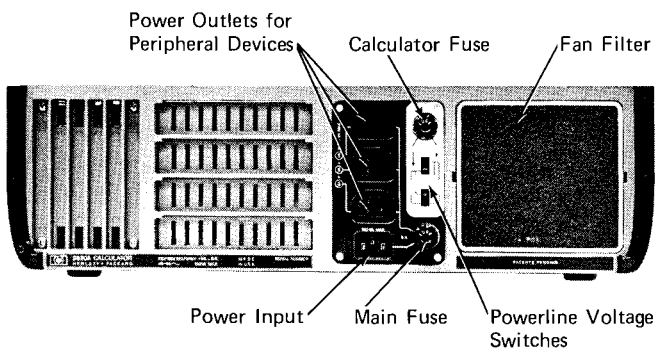


Figure 1-3. The Rear Panel.

LINE FREQUENCY: The Model 20 can be operated on line frequencies of 48 Hz to 66 Hz; the two frequently encountered frequencies are 50 Hz to 60 Hz.

POWER CONSUMPTION: With no peripheral equipment attached, the Model 20 requires a maximum of 150 voltamps.

There are three power outlets on the rear panel of the Calculator. They are used to supply ac power to peripheral equipment, and a maximum combined total of 610 voltamps may be drawn from these outlets. The outlets are not switched on or off by the power switch of the Calculator, but are active when the Calculator is plugged in.

GROUNDING REQUIREMENTS

To protect operating personnel, the National Electrical Manufacturers' Association (NEMA) recommends that the Calculator's keyboard and cabinet be grounded. The Calculator is equipped with a three conductor power cable which, when connected to an appropriate power receptacle, grounds the cabinet and the keyboard of the Calculator.

FUSES

The Calculator has two fuses located on the rear panel (see Figure 1-3). The main fuse is a 6-amp fuse. The Calculator fuse is either a 2-amp fuse for 100/120 vac operation, or a 1-amp fuse for 220/240 vac operation.

The main fuse protects the Calculator and any devices connected to the three power outlets. The Calculator fuse protects just the Calculator.

Three spare fuses, a 6-amp, a 2-amp, and a 1-amp (listed in Table 1-1) are shipped with each Calculator.

Table 1-2. Power-Line Voltages and Fuses

NOMINAL VOLTAGE	OPERATING RANGE (-10% , $+5\%$ of nominal)	CALC. FUSE
100 volts	90 to 105 volts	2-amp
120 volts	108 to 126 volts	2-amp
220 volts	198 to 231 volts	1-amp
240 volts	216 to 252 volts	1-amp

WARNING

BEFORE CHANGING A FUSE, ENSURE THAT THE CALCULATOR IS DISCONNECTED FROM ANY POWER SOURCE.

◆◆◆◆◆ **OWNER'S INFORMATION** ◆◆◆◆◆

FUSES (cont'd)

To remove a fuse, press inwards on the cap of the fuseholder, while at the same time twisting the cap in the direction indicated by the arrow on the cap. Pull the cap free and remove the fuse. To install a fuse, place either end of the fuse into the pocket in the cap, and reattach the cap to the body of the fuseholder. Do this by pressing inwards on the cap and twisting it in the opposite direction of the arrow.

Always be sure that the proper fuses are correctly installed in the Calculator. Failure to follow this precaution can defeat the purpose of the fuses, and result in needless damage to the Calculator in the event of certain types of malfunctions or unusual line voltage conditions.

INITIAL TURN-ON PROCEDURE

With the Calculator disconnected from any ac power source, verify that the correct Calculator fuse is installed for the line voltage in your area. (Refer to the preceding section for information regarding the fuses.)

The two slide switches located on the rear panel (see Figure 1-2) must be set to the particular line voltage to be used. The switches are set by inserting the tip of a small screwdriver into the white slot on the switch and then sliding the switch so that the slot is opposite the line corresponding to the desired voltage.

Switch the LINE OFF/ON switch, located on the right front of the Calculator, to the OFF position. Connect the power cord to the ac power input connector (see Figure 1-3) at the rear of the Calculator, and plug the other end of the cord into a suitable ac power outlet.

Switch the LINE OFF/ON switch to the ON position; after a few seconds the following display will appear, indicating that the Calculator is ready to operate. (Except for the printer — printer paper must be loaded before it will operate. Loading printer paper is described in the next section.)

0: END F

NOTE

If the Calculator is rapidly switched OFF and then ON, the display may blank and the Calculator appear inoperative. To correct this condition, switch the Calculator OFF for several seconds, and then turn it ON again.

If you are turning on a Calculator that has been transported by any commercial carrier, it is an excellent idea to verify proper operation of the Calculator by performing the electrical inspection procedure given in Model 20 System Electrical Inspection Booklet.

LOADING PRINTER PAPER

Printer paper is loaded by using the following procedure (refer to Figure 1-4). The procedure assumes that the Calculator is turned on.

1. Lift the access cover attached to the Calculator's top cover.
2. Remove and discard the paper core of any previous roll. If the remaining roll is small and a new roll is to be installed, remove the old roll by:
 - a. Unrolling and lifting it upwards until the roll is above the bail, and,
 - b. Grasping the roll firmly while pulling it upward and forward; the paper guide will tear the paper off cleanly.
3. If any paper remains in the printer mechanism, remove it by pressing the PAPER button while gently pulling on the paper as it emerges from the printer window.
4. Remove the first layer of paper from a new roll. (A cleanly torn or cut edge is better than a ragged one, although it usually makes no difference.)
5. Insert the new roll such that the free paper end is positioned as shown in Figure 1-4. Be sure that the bail drops back into place.

OWNER'S INFORMATION

6. Press the PAPER button until the paper advances through the printer mechanism and appears beyond the printer window. Once in a while the advancing paper will be trapped behind the printer window and begin folding into the available space there. If this should happen, release the PAPER button immediately or else the paper may begin to wind around the platen. Now remove the printer window (refer to Figure 1-5) and straighten the paper into its intended position. Then replace the printer window, and press PAPER as before.
7. Lower the access cover.

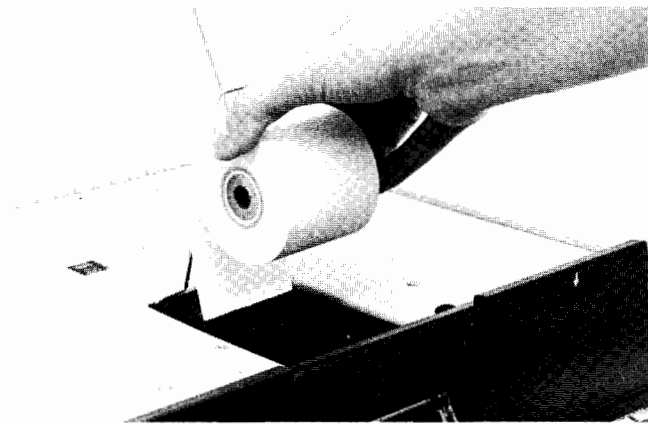


Figure 1-4. Loading Printer Paper.

To remove a partially used roll of printer paper, follow steps 1 through 3 of the preceding procedure.

NOTE

The printer will not operate unless it is loaded with paper, and any attempted use of the printer will result in NOTE 16 appearing in the display.

CLEANING THE CALCULATOR

The Calculator can be cleaned with a soft cloth dampened in clean water, or water containing a soft soap or mild detergent. Do not use an excessively wet cloth or allow water to penetrate inside the Calculator! Also, do not use any abrasive cleaners, especially on the display window.

The fan filter (located at the rear of the Calculator) should normally be cleaned about every three months. Clean the filter by holding it under a running water-faucet or by washing it in warm soapy water followed by rinsing in clean water. Dry the filter thoroughly before installing it.

Turn the Calculator off. The filter is removed by prying it out with a narrow, blunt instrument, such as a screwdriver or nail file. Insert the instrument into one of the slots on either side of the filter, and pry the filter outwards from the rear panel. To replace the filter, first snap one side back into place, and then the other.

The printer window can be removed for cleaning, or to facilitate paper loading, by carefully sliding it in the manner indicated by Figure 1-5.



Figure 1-5. Removing the Printer Window.

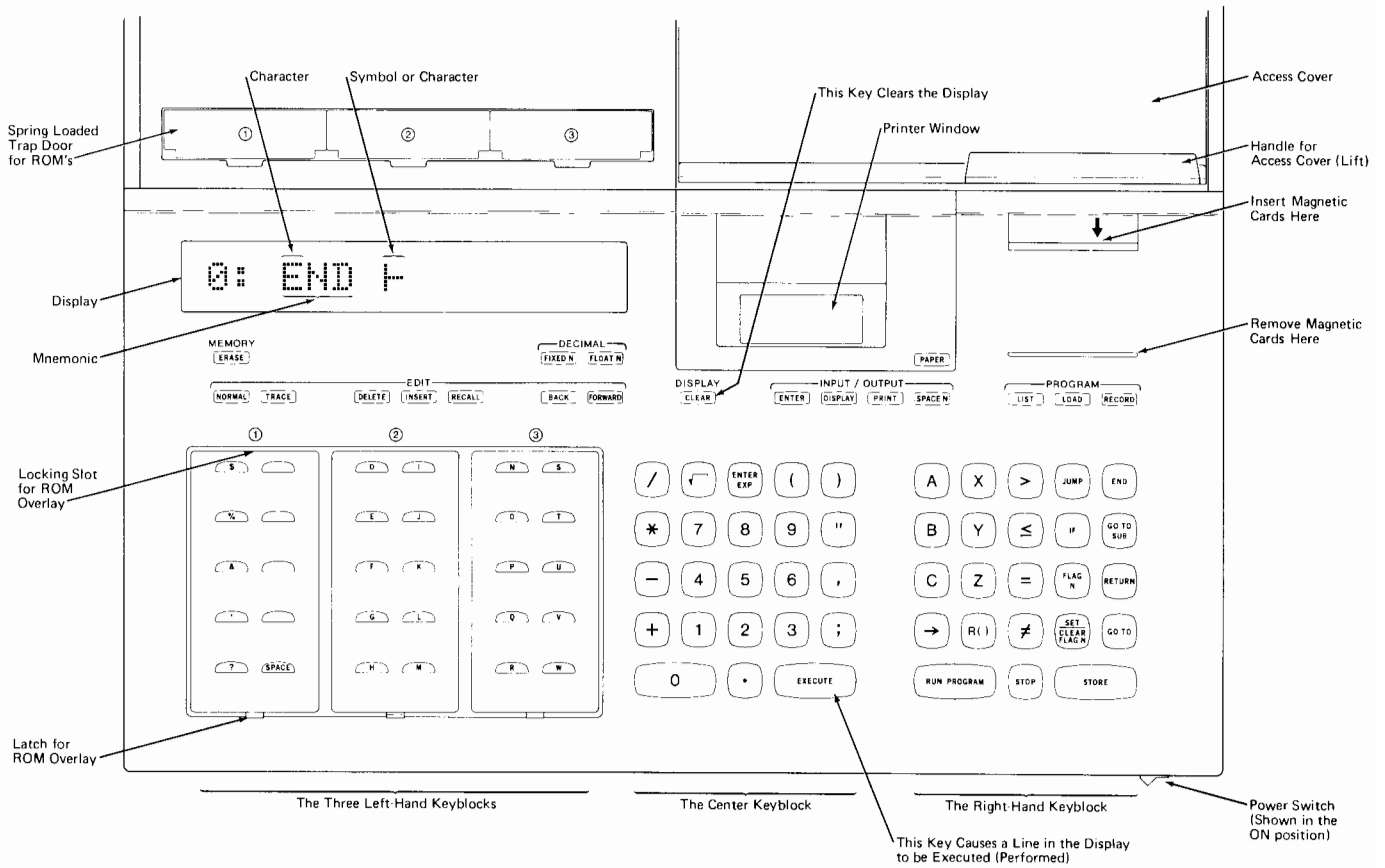


Figure 2-1. A Representation of the Model 20 Calculator.

Chapter 2

INTRODUCTION TO THE MODEL 20

This chapter presents information about many aspects of the Model 20's operation. The information is somewhat general in nature, and intended to lay the foundation for the more specific presentations of the remaining chapters. Those chapters are written in a way that assumes the reader has a fair amount of familiarity with the Model 20; this chapter provides that familiarity.

◆◆◆◆◆ A SURVEY OF THE CALCULATOR ◆◆◆◆◆

Figure 2-1 is a representation of the Model 20 Calculator.

THE KEYBOARD

The center keyblock contains the keys used to enter numbers and to indicate arithmetic operations. It also contains some keys used to provide *punctuation* for things written in the Model 20's language.

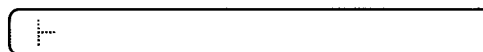
The right-hand keyblock contains keys used to provide access to part of the Model 20's memory, and keys to enhance the programmability of the Model 20.

The three left-hand keyblocks have special significance. These keys are used in conjunction with any plug-in ROM's that are installed. However, pressing a key of a left-hand keyblock which is not defined by a ROM will result in `NOTE 11` in the display, unless the key was used in a *quote field*. A quote field is a series of keys bounded on both sides by the " character; quote fields are used to generate alphameric information.

Each of the three left-hand keyblocks has a number, and is associated with the ROM slot of the same number. The numbers are located at the head of each left-hand key block, and on the spring loaded trap door over each ROM slot. Plug-in ROM's are discussed in "PLUG-IN ROM'S", beginning on page 2-3.

The remaining keys on the keyboard are used for editing the things you have 'said' to the Model 20 in its language, for operating the printer and the magnetic card reader, and for some other miscellaneous things listed below.

- a. MEMORY ERASE clears and initializes the *entire memory* in the same manner as does turning on the Calculator. See "INITIALIZING THE CALCULATOR WITH MEMORY ERASE", page 3-1.
- b. CLEAR removes all information from the display, and leaves it as follows:



CLEAR does not affect data or programs stored in the memory!

CLEAR is frequently used to rid the display of the remnants of a previous operation prior to beginning a new one. In particular, CLEAR is sometimes used to remove a `NOTE` resulting from pressing an improper sequence of keys.

- c. PAPER is used to manually advance the paper in the printer. PAPER is called a button to distinguish it from the keys; PAPER is mechanically connected to the printer mechanism, while the keys are processed electronically. PAPER cannot be programmed; use `SPACE N` for that purpose.
- d. `FIXED N` and `FLOAT N` are used to control the form in which processed and computed numbers are displayed and printed.
- e. ENTER provides the handiest and most important method of entering data into a program.

THE DISPLAY

The display is the primary means of communication with the Model 20. *It is used to indicate, in the Model 20's language, the activity that the*

A SURVEY OF THE CALCULATOR

THE DISPLAY (cont'd)

Calculator is to perform. The display is also used to show the results of that activity, display messages, and to give diagnostic information to the Operator in the form of notes.

For instance, if you wished to perform a calculation, it would have to be indicated in the display before it could be performed. To perform the calculation $(5 + 10)/3$, the display would have to be:



To form this display, press:



To cause the calculation to take place, press:



The answer (five) will then appear in the display.

The display can present as many as 16 characters of information at one time. Each character is formed from a 5 X 7 matrix of light-emitting diodes (LED's).

Although the display is limited to showing only 16 characters *at any one time*, this does not mean that the display cannot be used to convey a larger amount of information*. Nor does it mean that a set of directions written in the Model 20's language is limited to 16 keystrokes. (Also, many keys place more than one character in the display.) If, while specifying a given activity for the Calculator to perform, the number of characters needed exceeds the length of the display, the latest 13 to 16 characters worth of information is shown**. There are also provisions for bringing the earlier (and now non-visible) characters back into view. This subject is given considerable attention in Chapter 3, where the rules are given for generating and manipulating things in the display.

*16 characters will always be enough to indicate any numerical answer, however.

**The reason why it is not always simply 16 is given in Chapter 3.

THE PRINTER

The printer can print data, messages, program listings, and a running record of the Model 20's activities.

In general, the printer has the ability to print the same types of things that can be seen in the display. However, the PRINT key does not cause an automatic printing of what is in the display; in general, the printer will print what it is told to print, according to the list of items following the PRINT.

The printer can print as many as 16 characters in one row (that is, on a line — we won't say 'line' however, as we have a different and very important use for that word).

One data number or message can be printed per row. As before, 16 characters will always be enough to represent any data number. Message segments are also limited to 16 characters per row, but the complete message can consist of more than one row.

The use of the printer is governed by the rules of the Model 20's language. Some of this information is presented in the next chapter, and a complete presentation is given in Chapter 5.

THE MAGNETIC CARD READER

The magnetic card reader makes a reusable record of programs and data by magnetically storing the information on cards coated with a magnetic material.

The dark shiny surface of a card is the magnetic coating, and should be protected from scratches and magnetic objects. The other surface of the card can be written on, so that the card can be identified.

A given *record* (recording) of a program or of some data is not limited to what may fit on one card; as many cards as needed can be used to make the record. However, programs and data cannot be mixed on the same record.

A SURVEY OF THE CALCULATOR

The magnetic coating of a card is divided into two portions, called 'sides'. Each side can hold information, and each side of a card is independent of the other. As a card goes through the magnetic card reader, one side of the card is used (the right-hand side as viewed from the front of the Calculator). To use the other side the card is turned endwise before inserting it into the magnetic card reader.

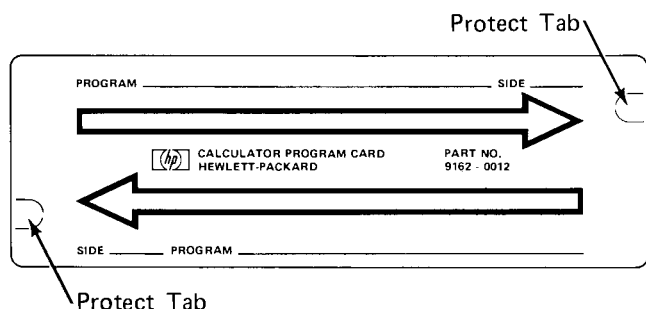


Figure 2-2. A 6-inch Magnetic Card.

It often happens that a program or collection of data requires several sides to be entirely recorded*. The usual convention is to number the sides in the sequence that they were recorded. In this way they can be reentered in the same sequence at a later time. The same sequence must be used or the information will be improperly loaded into the Calculator.

*NOTE 14 will appear if more sides are needed to complete the current card reader operation. Simply press EXECUTE and insert the next side.

The cards themselves are reusable, and new information can be recorded over old information, except as noted below.

Each side of a card can be protected from being rerecorded, by snapping out the tab on that side. See Figure 2-2. This action is permanent, and NOTE 13 will result if an attempt is made to rerecord a protected side of a card.

As with the printer, the operation of the magnetic card reader is subject to the rules of the Model 20's language, and its operation is explained in Chapter 5.

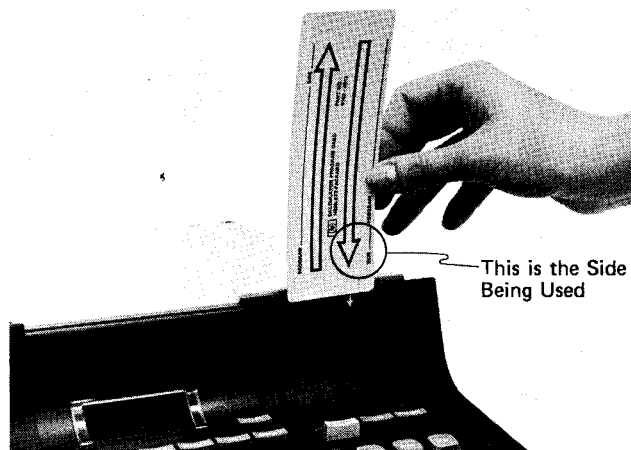


Figure 2-3. Inserting Magnetic Cards.

PLUG-IN ROM'S

A ROM defines one or more left-hand keyblocks. When a ROM is installed, one or more overlays are snapped into place over the appropriate keyblocks; the overlays indicate how the keys have been defined. ROM's and their overlays are user installable, and a given ROM can be installed in any of the three slots. Table 2-1 is a summary of the ROM's that are available as this is written.

Each of the ROM slots 1 through 3 corresponds to the left-hand keyblock of the same number. A ROM installed in a slot defines the associated keyblock. However, it is possible for a ROM to define more than one of the left-hand keyblocks.

Table 2-1. Summary of Available ROM's as of early 1972.

MODEL	DESCRIPTION
11220A	PERIPHERAL CONTROL I — provides general purpose control of most peripherals, and features an especially powerful ability to control the 9862A Plotter.
11221A	MATHEMATICS — provides mathematical functions such as ln, log, sin, cos, and tan, etc.
11222A	USER DEFINABLE FUNCTIONS — allows the user to write his own mathematical functions and use them by name, and allows parameter lists to be included in subroutine calls. Also allows subroutines and functions to be written in terms of dummy variables.

Additional PERIPHERAL CONTROL ROM's will become available.

PLUG-IN ROM'S

(cont'd)

This situation occurs when the User Definable Functions ROM is installed, and one or both of the remaining left-hand keyblocks are not defined by other ROM's.

Remember, that unless the keys of a left-hand keyblock have been defined by a ROM, the use of those keys is limited to generating alphameric messages (or more correctly, *literals*, as defined in Chapter 5 — literals have several uses).

There is a series of ROM's called the Peripheral Control ROM's, which provides the means to control peripherals for the Model 20 System. Each of these ROM's provides the means for controlling several different peripherals, and each ROM is best suited for a certain type of operation.

The keys defined by a ROM are used in accordance with the same general language usage rules which apply to the basic Model 20 itself. However, most ROM's introduce additional usage rules which must also be observed.

When entering programs from magnetic cards (described in Chapter 5), the decision as to which ROM's are to be installed in which slots is usually not arbitrary, but is determined by the configuration that existed at the time the program was recorded.

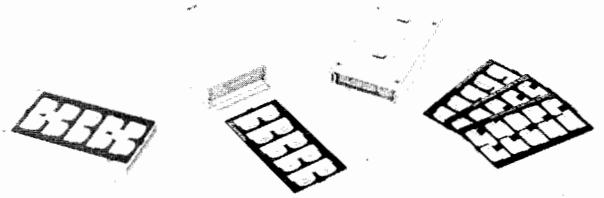


Figure 2-4. Some ROM's and their Overlays.

Whenever a program is recorded on a magnetic card, the configuration of the ROM's in the slots is also recorded. When the card is read to put the program back into the machine, the recorded configuration is checked against the existing configuration. If a ROM is missing, or in a different slot, NOTE 15 will appear in the display. The information on the card will be transferred to the Calculator, however, and it is up to the Operator to decide whether or not to continue with his activities.

The presence of ROM's that were not installed at the time the card was recorded will not cause NOTE 15, as long as they are installed in otherwise unused slots. (However, since the User Definable Functions ROM can define more than one keyblock, some programs may require that unused ROM slots remain empty.)

THE CONTROL OF PERIPHERALS

Table 2-2 lists the peripheral equipment that is (or is soon to be) available for the Model 20 at the time this is written.

With the exception of the Model 60 Marked Card Reader, (whose operation is explained in Appendix II), all peripheral equipment is controlled by keys defined by a Peripheral Control ROM (P.C. ROM). Each P.C. ROM can control most of the peripherals.

From time to time, new P.C. ROM's will become available. These additional P.C. ROM's will add to the flexibility of the Model 20 System, by provid-

ing a broad mix of peripheral control capabilities, and also by providing the ability to control the more sophisticated peripherals that are usually associated with computer systems.

Each P.C. ROM provides additional language elements which allow for the actual control of peripherals. Some of these language elements are dedicated to specific peripherals, while many of them are truly 'general purpose', and work with many peripherals.

Each peripheral connects to the Calculator through a cable and an I/O card. The I/O card is

◆◆◆◆◆ THE CONTROL OF PERIPHERALS ◆◆◆◆◆

inserted into one of the four I/O slots behind the spring loaded trap doors on the rear panel of the Calculator. See Figure 2-5.

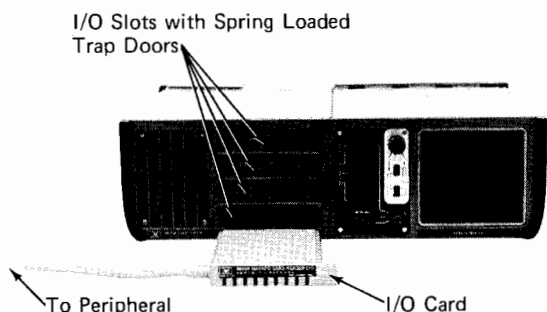


Figure 2-5. How Peripherals are Connected.

The Model 20 can receive up to four I/O cards, and thus can control up to four peripherals at one time. By connecting a Model 9868A I/O Expander, the Calculator can control as many as 13 peripherals.

The control of most peripherals involves a *select code*, which indicates the particular peripheral involved with the external activity. Each type of

peripheral has its own select code. It is sometimes possible to have two or more of the same type of peripheral connected to the Model 20 at the same time, provided that the I/O cards for the 'extra' peripherals have been modified to respond to an unused select code.

Table 2-2. Peripheral Equipment Available as of early 1972.

Model	Description
9860A	Marked Card Reader — reads data and programs from special cards marked in pencil.
9861A	Typewriter Output — provides formatted output.
9862A	Plotter — provides graphical output as large as 10" X 15".
9863A	Paper Tape Reader — allows input of data from paper tape.
9864A	Digitizer — converts graphical data to digital data for direct use by the Calculator.
9865A	Tape Cassette — provides bulk storage on magnetic tape for data and programs.
9868A	I/O Expander — allows as many as 13 peripherals to be connected at one time to the Calculator.

◆◆◆◆◆ THE ALGEBRAIC LANGUAGE ◆◆◆◆◆

By referring to a 'Model 20 language', we mean there are well defined and systematic ways of combining certain characters, and that desired Calculator activity is indicated by 'legitimate' combinations of these characters.

By saying that the language is algebraic, we mean that, as far as computational activities are concerned, the language strongly resembles the conventional notation used in mathematics. It is not meant that the Model 20 does algebra, per se. The Operator must do any algebraic manipulation of the quantities involved; the Model 20 can only evaluate algebraic expressions, or perform some other predetermined activity. It cannot, by ordinary means, reduce or otherwise manipulate an equation. The Model 20 manipulates numbers, not unknown quantities in the algebraic sense.

Just as the unit of communication in the English language is the sentence, the unit of communication in the Model 20's language is the *line*.

The rules for combining the elements of the Model 20's language into lines are called the *syntax* of the language, although we shall often speak of the syntax of individual statements and instructions.

There are two important operational consequences of the Model 20's algebraic language:

- a. Due to the use of conventional mathematical notation, a computational activity must be entirely specified before it can be performed. An expression like $18/3(1+2)$ is keyed in its entirety (as a line) before it is evaluated. This is in contrast with most desk top calculators, where pressing each key results in immediate computational activity. The advantage here is obvious: the user does not need to concern himself with an analysis of the expression to determine which keys to press first, in order to compute the correct result.

◆◆◆◆◆ THE ALGEBRAIC LANGUAGE ◆◆◆◆◆

(cont'd)

- b. For the most part, the mechanics of doing things with lines (entering them, altering them, and inserting or deleting lines of programs as discussed later) are quite separate from understanding what a particular line will accomplish when it is performed, and from the syntax rules used in planning a particular line.

This can be likened to learning to drive a car: the tasks of learning to start the engine and operate the transmission can be learned without regard to traffic rules. But before driving, the traffic rules must also be learned. It is interesting to note, however, that either body

of knowledge can be learned first. So it is with the Model 20. But in this manual, we choose to present the mechanics of operation first, leaving the reader to learn as much of the syntax, afterwards, as he chooses.

The rest of this chapter lays the foundation needed for understanding the mechanics of Model 20 operation, presented in Chapter 3. The presentation of the syntax proper is in the second half of Chapter 5. However, as with the mechanics, some preparation for that presentation is needed, and is contained in Chapter 4, and the first half of Chapter 5.

◆◆◆◆◆ A BRIEF LOOK AT LINES AND STATEMENTS ◆◆◆◆◆

A *statement* is the smallest complete syntactical unit which is capable of completely specifying an activity (but not considering those activities which fall into the 'mechanics' classification – those are not considered to have any syntax associated with them).

A line is made up from one or more statements, separated by semicolons. The number of statements in a line is usually arbitrary, and is determined by the user according to his feelings at the time the line is written.

Our purpose here is served by gaining an intuitive understanding of lines and statements; the terms are rigorously defined in Chapter 5. For now, an example will suffice.

The three statements in the line shown in the example below instruct the Model 20 to:

- a. Display or print any numbers as fixed point numbers with two places to the right of the decimal point (i.e., the number 18 would be printed as 18.00).
- b. Print the text shown within the quotation marks exactly as it is given within them, even

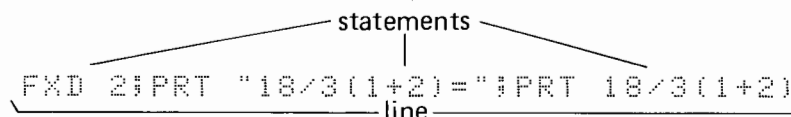
though there are numbers in the text that do not agree with the previous specification as to the form of numbers.

- c. Print the result of the computation $18/3(1+2)$, which is 2.00.

The same activity could also be specified with three separate lines, each containing one of the statements in the original line.

The execution of a line proceeds by statements from left to right. Execution within a statement proceeds according to rules for each individual type of statement. In the expression $18/3(1+2)$, the 1 and 2 are added, and then multiplied by three before the division into 18; obviously not left to right. There are some exceptions to the rule of line execution by statements, in the left to right order. These are covered as they occur and need not concern us here, except to note that the rule is usually true, though not always.

The mechanics of Model 20 operation provide means for the alteration of a line prior to its execution, with or without having to re-write the entire line.



◆◆◆ THE CONSEQUENCES OF PROGRAMMABILITY ◆◆◆

Some of the mechanics of Model 20 operation are a direct result of the fact that the Model 20 is programmable. To adequately prepare for the discussion in the next chapter, we need to define a program (again, mostly on an intuitive basis), and present some concepts that relate to the programmability of the Model 20.

A program is a sequence of lines which, together, perform some useful activity. Each line in a program is associated with a *line number*. (Usually) the first line in a program is numbered 0; the rest are numbered sequentially with the unsigned integers 1, 2, etc. The Model 20 itself assigns the line numbers as the lines are entered as part of the program.

The lines of a program are executed one at a time, generally (but not always) beginning with line 0. After each line is executed, the one following it (the one with the next higher line

number) is executed next, unless something in the previous line alters this.

The *program line counter* is an entity inside the Model 20 which keeps track of which line is currently being executed, or if no line is currently being executed, the line that will be executed next. Unless something in the current line alters the program line counter, the counter is automatically set to the next line in the program when the current line is finished.

Aided by a syntax for changing the program line counter, the mechanics of Model 20 operation provide means to inspect, change, insert and delete any lines in a program, even after the entire program has been loaded into the machine, and, to do so without having to reenter the program each time such a modification is made. The procedures for these activities are given in the next chapter.

◆◆◆ THE MODEL 20 MEMORY ◆◆◆

THE TYPES OF MEMORY

The Model 20 has four kinds of memory:

- a. Internal Read-Only-Memory (ROM) — This memory defines how the Model 20 operates, and gives each key its individual properties. ROM is fixed and its contents cannot be altered.
- b. Internal Read-Write-Memory (RWM) — RWM is memory whose contents can be changed by the activities of the Calculator. The Model 20 uses internal RWM for its own 'bookkeeping' while it accomplishes its calculator activities.
- c. Plug-in ROM — Plug-in ROM's are used to supplement the internal ROM by defining the keys of the left-hand keyblocks.
- d. User RWM — This is the main memory area of the Calculator, and is the one available for storing data and programs.

This discussion is primarily about the User's RWM, although the other types of memory do play a part in the explanations to follow.

THE USER'S RWM

The size of the User's RWM (usually measured in registers) depends upon the presence or absence of option 001, *as well as upon which plug-in ROM's are installed*. Some plug-in ROM's transform a small amount of User RWM into Internal RWM, because of additional internal bookkeeping requirements.

All RWM is *volatile*. By this we mean that the contents of the memory are lost when power is removed. When power is restored, the memory is initialized to contain the *null program* and all zeros in the data registers. The null program is the single line program:

```
0: END F
```


THE MODEL 20 MEMORY

DATA STORAGE

The unit of memory usage for data storage is the *register*. A register is a unit of memory that can hold up to 12 significant digits of a number within the range $-9.9999999999 \times 10^{99}$ to -10^{-99} , 0, and 10^{-99} to $9.9999999999 \times 10^{99}$.

Registers have names. There are six registers, called the 'alpha registers', whose individual names are A, B, C, X, Y and Z. There are six keys on the keyboard, each one of which corresponds to one of the alpha registers. The six alpha registers are *always* available for use.

There are registers, called the 'R registers', named R0, R1, R2, etc. In the basic Model 20 the highest numbered R register is R172; with option 001 installed, the highest number is 428.

The R registers are designated by the 'R ()' key, followed by other keys to indicate *which* R register.

The number of R registers available for the solution of any given problem varies according to which plug-in ROM's are installed and how much memory is required to store the program(s) used to solve the problem. This is fully discussed below.

PROGRAM STORAGE

The unit of memory usage for program storage is the line. A line need not have any particular length, and individual long lines will require more memory to store than will individual short lines.

RWM STRUCTURE

Figure 2-6 illustrates the structure of the RWM in the Model 20.

Both programs and data are stored in User RWM; the amount of memory available for data storage is whatever is left over from program storage.

Notice that the area for program storage is titled 'mainline programming'.

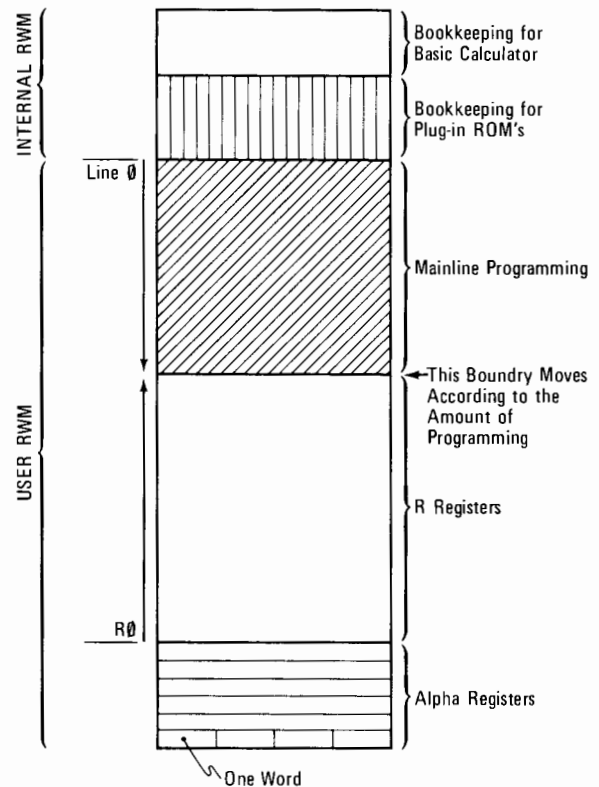


Figure 2-6. The Model 20 RWM.

When the User Definable Functions ROM is installed, the programming area of memory can be divided into the definable function area and mainline programming area.*

Programs for both programming areas are written according to the same general procedures, and most things said in this manual apply to programs for either area.

The alpha registers are always available, regardless of how much memory is required to store the program(s). The highest numbered R registers are sacrificed first, as the program(s) are stored. R registers that have been sacrificed, to make room for programming, or because of the installation of certain ROM's, are simply not in existence anymore, and any attempt to access them is an error.

*The User Definable Functions ROM provides quite a bit of additional capability for handling programs. The *subprograms* that can be written using that ROM are not stored or accessed in quite the same way as are ordinary programs, hence the term 'mainline programming' for the ordinary type of programs.

◆◆◆◆◆ THE MODEL 20 MEMORY ◆◆◆◆◆

A listing of a program on the printer after the program has been stored will indicate how many R registers are available. The last line of the listing will look something like this: R80. In this instance R0 through R79 (a total of 80) are available.

You should not think of the RWM as being fundamentally composed of registers. The actual unit of memory construction is something called a *word*. For instance, four words make a register, and each line is made up of a variable number of words.

The number of words used in bookkeeping for the basic Calculator is fixed (256). The number of words used in forming the 6 alpha registers is also fixed (24).

The number of words used for internal bookkeeping for each plug-in ROM depends upon the ROM. For instance, the Mathematics ROM does not use any, while Peripheral Control I (P.C. I) uses 22 words (the equivalent of $5\frac{1}{2}$ registers — but it

appears as a loss of 6 R registers, as half a register is of no use as a register).

Suppose P.C. I and another ROM, that also requires 22 words of bookkeeping, were both installed in the Calculator. Either ROM by itself would consume the memory required for 6 registers, but together, they only use 44 words, or 11 registers.

As each line of programming is stored in memory, only as many words as necessary are used to represent the line.

Those words that are left over from all of the preceding allocations are grouped into groups of four, and treated as R registers.

There is a rough rule of thumb which describes the relationship between the amount of memory used by a register and by a line. *On the average*, every 8 keystrokes of a (stored) line use the memory required for one register.

◆◆◆◆◆ KEYS, AND CHARACTERS IN THE DISPLAY ◆◆◆◆◆

The keys on the keyboard can be classified into three groups:

- a. The syntax related group, in which each key either alters a character or places an additional character in the display. Most keys fit into this group, and these are the keys used to write lines.
- b. The mechanics related group, whose keys manipulate individual characters already in the display, or, do not place an associated character(s) in the display when they are pressed.

- c. The ambivalent group, whose keys fit into either of the other two groups, according to the way they are used. These keys are: STOP, FIXED N, FLOAT N, NORMAL, TRACE, and SET/CLEAR FLAG N.

Characters can be classified as symbols, or grouped together and called mnemonics.

Symbols: Γ , +, $\frac{1}{2}$, \div , etc.

Mnemonics: GTO, STP, SFG, etc.

◆◆◆ KEYS, AND CHARACTERS IN THE DISPLAY ◆◆◆

Table 2-3. The Symbols and Mnemonics for the Keys of the Basic Model 20.

KEY	IN QUOTE FIELD?		KEY	IN QUOTE FIELD?		KEY	IN QUOTE FIELD?		KEY	IN QUOTE FIELD?	
	NO	YES		NO	YES		NO	YES		NO	YES
(1)	1	1	R()	R	:	LOAD	LODb	NOTE 2	S	NOTE 3	S
(2)	2	2	=	=	=	RECORD	RECb	NOTE 2	T	NOTE 3	T
(3)	3	3	≠	≠	≠	A	A	A	U	NOTE 3	U
(4)	4	4	>	>	>	B	B	B	V	NOTE 3	V
(5)	5	5	≤	≤	≤	C	C	C	W	NOTE 3	W
(6)	6	6	GO TO	GTO b	⊘	X	X	X	\$	NOTE 3	\$
(7)	7	7	GO TO SUB	GSS b	NOTE 2	Y	Y	Y	%	NOTE 3	%
(8)	8	8	RETURN	RET b	NOTE 2	Z	Z	Z	&	NOTE 3	&
(9)	9	9	STOP	STP b	!	D	NOTE 3	D	'	NOTE 3	'
0	0	0	END	END b	NOTE 2	E	NOTE 3	E	?	NOTE 3	?
.	.	.	JUMP	JMP b	NOTE 2	F	NOTE 3	F	SPACE	NOTE 3	<i>b</i>
ENTER EXP	E	↑	IF	IF b	NOTE 2	G	NOTE 3	G	These keys are the Mechanics Group and have no Mnemonics or Symbols.		
+	+	+	FLAG N	FLG b	NOTE 2	H	NOTE 3	H	EXECUTE	—	—
-	-	-	SET CLEAR FLAG N	SFG b	NOTE 2	I	NOTE 3	I	STORE	—	—
*	*	*	SET CLEAR FLAG N SET CLEAR FLAG N	CFG b	NOTE 2	J	NOTE 3	J	RUN PROGRAM	—	—
/	/	/	FIXED N	FXD b	NOTE 2	K	NOTE 3	K	CLEAR	—	—
√	√	√	FLOAT N	FLT b	NOTE 2	L	NOTE 3	L	ERASE	—	—
(((ENTER	ENT b	NOTE 2	M	NOTE 3	M	BACK	—	—
)))	DISPLAY	DSP b	⚡	N	NOTE 3	N	FORWARD	—	—
;	;	;	PRINT	PRT b	NOTE 2	O	NOTE 3	O	DELETE	—	—
,	,	,	SPACE N	SPC b	NOTE 2	P	NOTE 3	P	INSERT	—	—
"	"	NOTE 1	NORMAL	NOR b	NOTE 2	Q	NOTE 3	Q	RECALL	—	—
→	→	→	TRACE	TRC b	NOTE 2	R	NOTE 3	R	LIST	—	—

NOTES

1. The " character never occurs inside a quote field; it is used exclusively to begin or terminate a quote field.
2. This key produces one character with an arbitrary pattern. Sometimes the pattern will vary according to which plug-in ROM's are installed.

3. If this key is used outside of a quote field, NOTE 11 will result unless the key is defined by a plug-in ROM, in which case the mnemonic or symbol is determined by the ROM.
4. The character *b* denotes a blank space.

◆◆◆◆◆ KEYS, AND CHARACTERS IN THE DISPLAY ◆◆◆◆◆

(cont'd)

Generally, each key in the syntax related group corresponds to a particular symbol or mnemonic, which is placed in the display each time the key is pressed. A list of keys and their mnemonics is given in Table 2-3, inside the rear cover, and also in the System Reference. The table shows the symbol or mnemonic for each key as it is used both inside and outside of a quote field (some

keys have two different symbols or mnemonics under the two different conditions).

The mechanics related group of keys comprise the bulk of the mechanics of Model 20 operation, and are explained in detail in the next chapter. The ambivalent keys STOP, NORMAL, and TRACE are also explained in the next chapter. The others of this group are explained in Chapter 5.

◆◆◆◆◆ REVIEW ◆◆◆◆◆

The following important concepts were presented in this chapter.

1. MEMORY ERASE is used to initialize the entire memory of the Calculator.
2. CLEAR removes all information from the display, but leaves the memory unchanged.
3. The Model 20 has an algebraic language, that is, a formal way of indicating calculations and other activity, and the notation of the language resembles that of conventional mathematics. The rules of the language are called its syntax.
4. The line is the fundamental unit of communication with the Model 20.
5. A line is made up of one or more statements, separated by semicolons.
6. A statement is the smallest syntactically complete unit of the Model 20's language which can completely specify an activity.
7. The display is used to indicate the activity the Calculator is to perform (that is, it shows a line), and, to indicate results of calculations.
8. A program is a sequence of lines which, together, perform some useful activity; it is a recipe for doing something.
9. Each line of a program has a line number. Line numbers are consecutive and start at zero.
10. The program line counter keeps track of which line of a program is currently being executed, or is about to be executed or stored next.
11. The amount of Read-Write-Memory available to the User (User's RWM) is dependent upon:
 - a. Option 001
 - b. Plug-in ROM's
12. RWM is volatile.
13. The unit of memory usage for data storage is the register.
14. The unit of memory usage for program storage is the line.
15. On the average, every 8 keystrokes of a (stored) line use the memory required for one register.
16. The number of R registers available depends upon the total amount of User's RWM available, and the amount of programming that has been stored; whatever RWM left over is used as R registers.
17. The printer can print as many as 16 characters in one row.

REVIEW

(cont'd)

18. The PRINT key does not cause an automatic printout of what is in the display; the printer will print what it is told to print according to the list of things following the PRINT.
19. Each magnetic card has two sides, and each side can hold information independently of the other.
20. Data and programs cannot be mixed on the same side of a card.
21. New information can be recorded over old information, unless that side has been protected by snapping out the tab.
22. Plug-in ROM's are used to define the three left-hand keyblocks.
23. Any ROM can be installed in any slot, but the configuration used will usually make a difference when the magnetic card reader is involved.
24. Peripheral Control ROM's are used to enable the Model 20 to control peripherals. The only peripheral that can be used without a P.C. ROM is the Model 9860A Marked Card Reader.
25. The basic Model 20 can control up to 4 peripherals at one time. With the Model 9868A I/O Expander, the number is increased to 13.
26. Most of the keys on the keyboard (except the mechanics related group) have an associated symbol or mnemonic.

Chapter 3

THE MECHANICS OF MODEL 20 OPERATION

This chapter explains the procedures involved in the writing, modification, and subsequent implementation of a line. Collectively, this body of knowledge is called the mechanics of Model 20 operation, as it provides the means for putting the Model 20's language to work.

◆◆◆◆◆ THE FUNDAMENTAL PROCESS ◆◆◆◆◆

The user writes lines that convey his intentions. To do this he presses the proper keys in sequence. As the keys are pressed, the line appears in the display. After each line is written, it is either *executed* (the activity specified by the line carried out immediately) or *stored* as a line in a program.

After a program has been built, it can be *run* (each line in the program *automatically executed*).

INITIALIZING THE CALCULATOR WITH MEMORY ERASE

Before beginning the solution of a problem with the Calculator, it is a good practice to initialize the Calculator by pressing MEMORY ERASE. However, since this is equivalent to turning the Calculator off, and then on again (*completely erasing the memory*), exercise some care in deciding when to do this; you may inadvertently destroy someone else's program.

Pressing MEMORY ERASE does the following things:

1. Clears all User RWM by:
 - a. Loading the null program.
 - b. Zeroing all alpha and R registers.
2. Clears all flags (see FLAGS, page 5-2).
3. Establishes FLOAT 9 with previous FIXED 0. (The significance of the previous FIXED 0 is explained in "FIXED AND FLOAT STATEMENTS", on page 5-11.)
4. Specifies the Normal Mode (see TRACE MODE OPERATION, page 3-14).

5. Scans the ROM slots to determine what ROM's are installed and the amount of RWM to use for internal bookkeeping.
6. Performs any necessary initialization needed by ROM's which are plugged in, such as specifying degrees, etc.

WRITING A LINE

If, when beginning to write a new line, you are in doubt about whether pressing a series of keys will start a new line, or merely add on to a previous line, press CLEAR. After that, pressing keys will start a new line.

Press the keys which produce the desired line in the display. If you get a NOTE in the display, you have made an error. For now, press CLEAR and start over. Refer to Table 3-1 for a summary of the NOTE's.

As each key is pressed, its symbol or mnemonic is placed in the display, to the right of any previous characters in the display. If enough keys are pressed to fill the display, and then more keys are pressed, the characters shift to the left to make room on the right for each additional symbol or mnemonic. Under these circumstances only the last 13 to 16 characters placed in the display will be visible*; however, the non-displayed characters in the beginning of the line are still recognized as being part of the line.

*In general, this does not mean that the results of the last 13 to 16 keystrokes are displayed; most mnemonics have more than one character. See the next example in the text for why the number of visible characters is sometimes less than 16.

◆◆◆◆◆ THE FUNDAMENTAL PROCESS ◆◆◆◆◆

WRITING A LINE (cont'd)

—— EXAMPLE ——

Suppose that you wished to write the line:

PRT 3,4,r(3*3+4*4)

PRESS: CLEAR

PRESS: PRINT 3 , 4 , ⌊ (3
* 3 + 4 * , but don't press 4
) yet.

At this point the display will be:

PRT 3,4,r(3*3+4*

Notice that the display is full, (that is, it is occupied by 16 characters, one of which is a *blank*) even though only 13 keys have been pressed.

When the next key of the line is pressed, the items in the display will shift to the left to make room for the 4. In this case the shift will be four characters because the left-most item in the display happens to be `PRTb`; a mnemonic of four characters. Mnemonics are never split; either all of a mnemonic is visible, or none of it is visible.

PRESS: 4)

After the display has shifted four characters to the left, and the right-most 4 and) are added, there are still two blank spaces in the right-most portion of the display. The final display is one of only 14 characters.

3,4,r(3*3+4*4)

If the line is composed of more than one statement, each statement must be separated from the others by a semicolon. No special punctuation is required at the end of the last statement in the line.

Suppose you wished to store the number 10 in register A, the number 20 in register B, and the

number 30 in register C. This activity can be specified in one line by combining three statements:

10+A!20+B!30+C

After a line has been written it can be executed (by pressing EXECUTE). After execution, the display may contain an answer associated with the line. At this time a new line can be written simply by pressing the keys that describe the line. As the first key is pressed, the display is cleared, and the mnemonic or symbol for that key is placed in the display.

MAXIMUM LINE LENGTH

The maximum number of keystrokes permitted in a line is not fixed, but depends upon which keys are pressed, and upon the order in which they are pressed. The number of characters needed to represent the line in terms of its symbols and mnemonics is *not* what determines the maximum length of a given line.

There is no readily understood rule for determining what the maximum number of keystrokes is for a given situation. However, its lowest value is about 35, and its maximum value is about 68; typically it is around 50. This is high enough to avoid any practical limitations in the majority of cases.

If you write a line that is too long, `NOTE 09` will appear in the display, and no further entries will be accepted. The line must be shortened before it can be used. *Press CLEAR and write the shortened version of the line.* Don't try to salvage any portion of a line that is too long by use of the editing techniques involving BACK or DELETE! Many times the easiest (or perhaps the only) thing to do is split the original line into two lines. Other times, it may be possible to shorten the line by making more efficient use of the language.

Some lines can be successfully keyed in, but will result in `NOTE 09` after an attempt to execute or store the line. Do not attempt to salvage such a line; press CLEAR and write a shortened version of the line.

◆◆◆◆◆ THE FUNDAMENTAL PROCESS ◆◆◆◆◆

The reason for the ambiguity concerning maximum line length has to do with a process called *compilation*, and with the internal structure of the Model 20's display. The section in Appendix I

titled "MODEL 20 INTERNAL STRUCTURE" contains some explanation of the compilation process and of how it relates to the display.

◆◆◆◆◆ MODIFYING A LINE ◆◆◆◆◆

If a line that has just been written (and not stored as a line in a program) requires extensive modification, or is short enough to rewrite easily, press CLEAR and rewrite the line. Pressing CLEAR completely removes a line (or anything else) from the display.

SEGMENTS

A part of a line which is a sequence of consecutive symbols and mnemonics is called a *segment* of that line. A segment of a line can be as short as a single symbol or mnemonic, or longer. For example, A;20+ is a segment of the line:

```
10+A;20+B;30+C
```

Modifying a line without pressing CLEAR and simply rewriting it, involves one or more of the following situations: adding, removing, or altering one or more segments of a line.

An alteration of an existing segment may make it longer (*in number of keystrokes — the number of characters representing the segment do not count*), or shorter, or it may remain the same length (again as determined by the number of keystrokes — but the number of characters might change).

EXACT SUBSTITUTION OF A SEGMENT

To substitute a segment of a line with another segment, when the number of keystrokes in each segment is equal, press BACK until the left-hand item of the original segment is the right-most symbol or mnemonic visible in the display. Then press BACK one more time.

Now press the keys of the new segment.

Then, if the right-most item of the line is not visible in the display, press FORWARD until it is. Now you can continue writing the line, or execute it, or store it, as is appropriate.

EXAMPLES

Suppose you had written (4+2) and then pressed * when you meant to press /. Simply press BACK, and then /, and continue writing the line.

```
A(B+C)/X+ZIZ+
```

Suppose you had written:

and then discovered that it should be:

```
A(B+C/X)+Z.....
```

The segment)/X needs to be replaced with the segment /X). Press BACK until) is the right-most character visible in the display. Now press BACK one more time, followed by the proper keys: /X). Now press FORWARD until the + is visible, and continue writing the line.

REMOVING A SEGMENT

To remove a segment from a line, press BACK until the right-most item of the segment is the right-most item visible in the display. Now press DELETE once for each symbol or mnemonic in the segment to be removed.

Then, if the right-most item of the line is not visible in the display, press FORWARD until it is. Now you can continue writing the line, or execute it, or store it, as is appropriate.

EXAMPLE

Suppose you wished to delete the segment

```
X+Y;PRT (A+B)/A;
```

from the line:

```
FXD 2;X+Y;PRT (A+B)/A;GTO 4
```


MODIFYING A LINE

REMOVING A SEGMENT (cont'd)

Press BACK until the display looks like:

```
X+Y:PRT (A+B)/A:
```

Now press DELETE 13 times. Notice how at first the display shifts to the right, bringing the first part of the line into view, which in this case is FXD δ . But the FXD won't appear until there is room in the display for all four characters. After the first part of the line comes into view, notice that the line appears to shorten by losing the items on the right-hand edge of the display while the rest of the line does not move. After the segment has been deleted, press FORWARD until the end of the now modified line comes into view.

```
FXD 2:GTO 4
```

Now you could continue writing the line, execute it, or store it, as is appropriate.

NOTE

Never attempt to remove the symbol \vdash from the end of a line by pressing DELETE; use BACK instead. Otherwise, the *entire line* may be removed.

ADDING A SEGMENT

To add a segment to the interior of a line, press BACK until the right-most item visible in the display is the symbol or mnemonic which is to immediately precede the segment you are planning to add. Now press INSERT, followed by the keys

which describe the desired segment. Then press FORWARD until the end of the line is in view.

As the keys following INSERT, but preceding FORWARD, are pressed, they are inserted into the line with no loss of any other characters in the line. The right-hand portion of the line is shifted to the right to make room for the additional items being inserted into the line. This action continues until either BACK, FORWARD, DELETE, CLEAR, EXECUTE, or STORE is pressed. Usually the insertion of a segment is terminated with FORWARD when it is used to return to the end of a line.

EXAMPLE

Suppose that you wished to insert the segment 20 \rightarrow B \vdash into the line:

```
10 $\rightarrow$ A $\vdash$  30 $\rightarrow$ C
```

Press BACK until \vdash is the right-most item in the display. Then press INSERT, and write the segment 20 \rightarrow B \vdash . Now press FORWARD until the end of the line is visible.

```
10 $\rightarrow$ A $\vdash$  20 $\rightarrow$ B $\vdash$  30 $\rightarrow$ C
```

If you make an error during the entry of a segment that is being inserted, press DELETE to remove the unwanted items. Then press INSERT again, and continue to write the desired segment.

REPLACING A SEGMENT

To replace a segment of a line with another segment of different length, delete the original segment and insert the new one in its place.

EXECUTING LINES

Once a line has been written, it is executed (the activity specified by the line accomplished immediately) by pressing EXECUTE. The right-most item in the line must be visible in the display before the line can be executed (except as noted below).

After a line has been executed the display will depend upon the nature of the line itself. Usually, the display will contain some type of quantity that can be considered to be a result of the last bit of activity accomplished by the line. However, sometimes the display will contain only the

EXECUTING LINES

symbol \vdash . This symbol is called the 'lazy T', or the 'end of line' symbol.

After a line has been executed, it can be brought back into the display by pressing BACK or FORWARD, or executed again simply by again pressing EXECUTE. The section of Appendix I titled "MODEL 20 INTERNAL STRUCTURE" offers some insight as to why this is so.

EXAMPLE

Suppose that you have executed the line:

$0 \rightarrow A$

That placed the number zero into register A.

Now suppose you wrote:

$A+1 \rightarrow A$

This line adds one to the value already in register A, and places that new result back into register A. Thus, the line increments the value of A each time that line is executed. Also, each time the line is executed, the result of the operation is placed in the display (although the line itself is also still present in the Model 20's memory). Thus, each time EXECUTE is pressed, the number in the display will increase by one: 1, 2, 3, . . .

SYNTAX ERRORS

The syntax of the Model 20's language consists largely of rules about what combinations of the various types of keys (arithmetic operations, the digits, punctuation, and several other classifications) are permitted.

Pressing a sequence of keys which does not fit into any of the combinations allowed is called a syntax error. Most syntax errors are detected when they are written.

A syntax error has to do with some breach of form in a statement, rather than by impossibilities suggested by the specific (numerical) content of the statement. Hence, $*/$ is a syntax error (two arithmetic operators can appear side by side only under special circumstances), while an attempt to store a number into a non-existent R register is not a syntax error. It is an error during execution, which is a different class of errors, discussed on page 3-13.

If while writing a line, you make a syntax error, the display will change to one which contains the word NOTE, followed by some number. The

number indicates the general nature of the error. The line cannot be continued until the error is corrected. Table 3-1 is a condensed list of the NOTE's and their meanings. Appendix I contains an expanded table of the Notes for the basic Model 20.

If the line is short, and easily rewritten, you may wish to press CLEAR, which simply removes the entire line, and thus the error.

If you wish to salvage the line, press FORWARD; this will bring the line into view*. Now correct the line using any of the line modification techniques described earlier in this chapter. Generally, this will consist of pressing BACK, followed by the correct key. Other times you might find either a missing or an extra parenthesis (or some other thing), and will have to go back into the interior of the line to change, insert, or delete some item in order to make the correction.

*If you already know that the error consists of the latest keystroke being incorrect, simply press BACK, and the display will appear as it did before the incorrect key was pressed. Now press the correct key and continue writing the line.

◆◆◆◆◆◆◆◆◆◆ **SYNTAX ERRORS** ◆◆◆◆◆◆◆◆◆◆

Table 3-1. Condensed Information About the Notes for the Basic Model 20.

INDICATION	MEANING
NOTE 01	Most syntax errors. Improper key while in Enter Mode.
NOTE 02	An instruction is followed by a parameter of the wrong type, illogical value, or, a parameter is missing. If flag 14 is not set: Square root of a quantity whose value is negative.
NOTE 03	Extra (or missing).
NOTE 04	Extra) or missing (.
NOTE 05	Parameter out of range when specifying an R register, i.e., too large, or negative.
NOTE 06	Attempt to store a number into something other than a register.
NOTE 07	A RET has not been preceded by a matching GSB.
NOTE 08	Improper branching statement.
NOTE 09	Attempt to use a line that is too long. Subroutines nested too deeply.
NOTE 10	A computation has resulted in an intermediate or final numerical result that is outside the range of the Calculator. NOTE 10 will occur only if flag 14 is not set.
NOTE 11	Pressing a key of a left-hand keyblock while an associated plug-in ROM is not installed, unless the key is used within a quote field. Attempt to execute an ENTER statement from the keyboard.
NOTE 12	Attempt to store a line when there is insufficient memory to accommodate it. No GTO or GSB preceding LOD when loading information under control of an existing program.
NOTE 13	Attempt to record on a protected side of a magnetic card.
NOTE 14	Magnetic card reader operation is not completed; press EXECUTE and insert the next side.
NOTE 15	A program is being loaded from a magnetic card while the three left-hand keyblocks are not defined in the same way as when the card was recorded.
NOTE 16	Attempted printer operation while the printer is out of paper.

STORING PROGRAMS

PRELIMINARY INFORMATION

A program is formed by writing a series of lines and storing them. After a line is written, it is stored in the (mainline) program memory by pressing STORE. (Unless otherwise specified by operations available only with an appropriate plug-in ROM, storing a line always means storing it into the mainline programming memory.) See Chapter 5 for information about using the magnetic card reader to program the Calculator.

Precisely where in the program memory the line is stored depends upon the current setting of the program line counter. For instance, if a line were stored while the program line counter was set at 2, that line would be physically stored in the memory immediately following lines 0 and 1, and would be designated line 2.

Immediately after a line has been stored, the display will contain, starting from the left, the line number assigned to the line, followed by a colon, followed by as much of the right-hand portion of the line as can be displayed, followed by \bar{L} .

EXAMPLE

Suppose the line shown below was written and then stored while the program line counter was set to 15.

(A+1)÷A

Then, after pressing STORE the display would be:

15: A+1÷A \bar{L}

What appears in the display is a *replica* of the line. The replica may appear in slightly altered form, as compared to the way the line was originally written. Unnecessary parenthesis are removed (and in some cases parenthesis are inserted) by the compiler as the line is stored. The compiler is briefly discussed in the section of Appendix II titled "MODEL 20 INTERNAL STRUCTURE".

After a line is stored, the program line counter is automatically set to its next higher value. Thus,

unless the program line counter is somehow altered prior to the next use of STORE, the next line stored will have a line number one greater than the previous line stored. The program line counter can be controlled from the keyboard with the END statement and the (absolute) GO TO statement. (There are other ways to control the program line counter, either from the keyboard or from within a program being run. With regard to the current topic, see the discussion, on page 3-12, of the additional properties of RECALL, BACK, and FORWARD.)

When executed from the keyboard, both of the following statements set the program line counter to line 0:

GTO 0
END

GTO is the mnemonic for the GO TO key. To set the program line counter to any other line, say n, use:

GO TO (n)

With one exception, the program line counter cannot be set to the number of a line that has *not already been stored*.

The exception is this: The program line counter can be set to the line number to follow the highest numbered line that has already been stored.

Suppose that six lines (0 through 5) have already been stored. Then GTO 6 is permitted and can be executed. However, GTO 7, or higher, will result in an error during execution.

A major consequence of the preceding rules is that lines must be stored in the sequence of their intended line numbers. That is, the line which is to be line 5 cannot be stored until line 4 has been stored, and so on, all the way back to line 0. Hence, line 0 must be the first line stored in the mainline (or any other) program area of RWM.

A SPECIAL PROPERTY OF END

The end statement has a special property: When a line containing an END is stored, any lines with higher line numbers are removed, and the memory used by those lines is changed back into R registers.

STORING PROGRAMS

A SPECIAL PROPERTY OF END (cont'd)

END also has other properties that are described in "END STATEMENTS", on page 5-36.

In general, the last line of every program should contain an end statement.

INITIALIZING THE PROGRAM MEMORY

It is wise to initialize the program area of RWM before storing a program. The preferred method of doing this is by pressing MEMORY ERASE. If you don't wish to do that, follow the procedure given below.

1. Set the program line counter to the line number at which the program is to begin (usually that will be line 0, but not always — it is possible to 'stack' programs).
2. Press END, STORE.
3. Set the program line counter back to the line number at which the program is to begin.

At this time the memory has been initialized; write the first line of the program and press STORE.

It is not at all obvious why such an initialization procedure is necessary. The reason is contained in the following explanation.

Suppose there were a 45 line program stored in the Calculator, and that you wished to load a five line program in its place. If you were to simply set the program line counter to 0, and then begin storing the five lines, those five lines would replace the original first five lines, *but you might still have a 45 line 'program'*. That would be the case if the fifth line stored did not contain an end statement.

Although it is generally a good idea for the last line of a program to contain an end statement, there are times (when stacking programs) that it is desirable that there not be an end in the last line of a program.

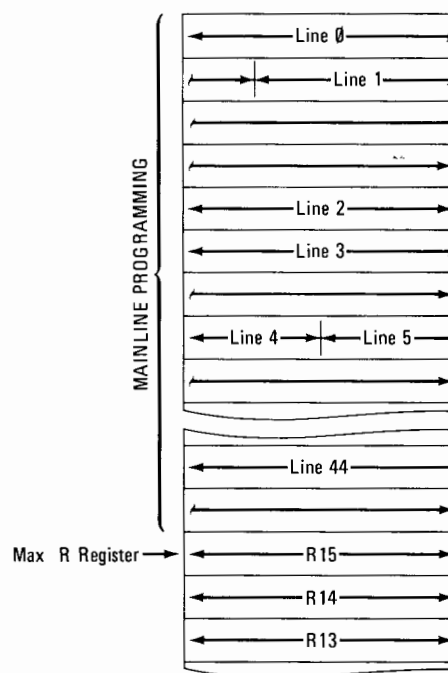


Figure 3-1. A Memory Map Illustrating the Assumption in the Text.

In our example, the presence of those extra 40 lines could be detrimental for at least two reasons: First, those extra lines reduce the number of R registers available, and those registers lost may be needed. Secondly, the amount of memory available might not permit all five lines to be stored. This could happen if the new five lines occupied more memory than the first five lines of the original program. As each of the new lines is stored, the lines with higher line numbers are moved 'up' or 'down' in the memory to compensate for any difference in line length between the old and new line. If, in this example, part of line 44 needed to move below R0, NOTE 12 (insufficient memory) would result.

By first storing an END, at the line number where the five lines are to begin, all lines of programming stored below that (lines 1 through 44) are removed. Then the line consisting of the END is removed, and the desired lines stored.

STORING A SINGLE PROGRAM

If the mainline program memory is to contain only a single program, that program is stored by

STORING PROGRAMS

first initializing the program memory, and then keying in each line and storing it, beginning at line 0.

EXAMPLE

Suppose you wished to enter the following program:

```
0: 0→A
1: A+1→A
2: PRT A
3: GTO 1
4: END
```

To do so, press: MEMORY ERASE, or END EXECUTE STORE EXECUTE*.

That initializes the program area of RWM, and sets the program line counter to 0. Now write each of the lines, one at a time, beginning with line 0. After each individual line is written, press STORE.

(If you wish, you can run this program after entering it by pressing END RUN PROGRAM; to halt the program, press STOP once.)

STACKING PROGRAMS

It is possible to put more than one program in the mainline programming area. This is called 'stacking' programs, and is illustrated in Figure 3-2.

The ability to stack programs is enhanced by the concepts of *relative addressing* and *symbolic addressing*, which are discussed in Chapter 5. These techniques make it possible to *branch*** without referring to individual line numbers. This is important, because when programs are stacked the line number of each line will depend upon where each program is placed in the stack. If the programs are written using relative or symbolic addressing, they can be stacked in any desired fashion.

*That sequence of keys is exactly equivalent to the sequence: END EXECUTE END STORE END EXECUTE. This is because the Calculator 'remembers' the last line it executed or stored.

**Branching is the name describing a program's ability to automatically go from one part of itself to a different part.

To stack a series of programs, first initialize the program area from line 0 onwards. Then store the first program. Now store the next program. It is not necessary to do any further initialization of the program area memory. Continue storing the programs until they are all stacked.

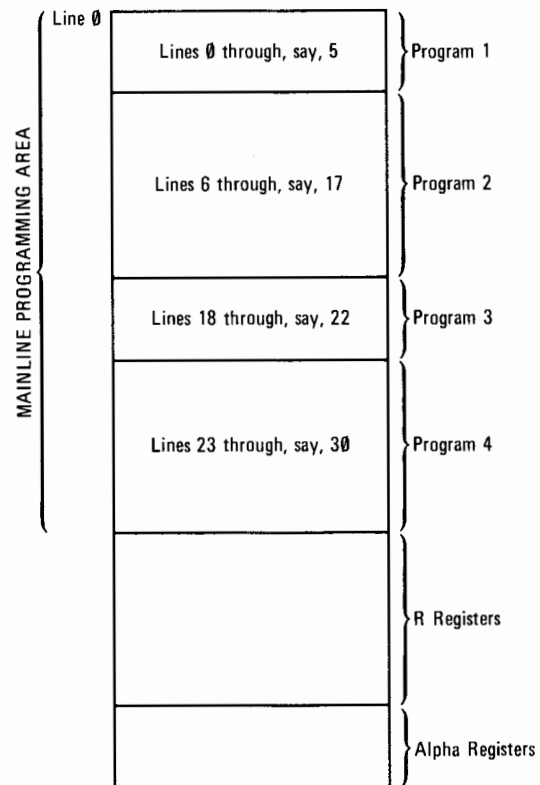


Figure 3-2. Programs Stacked in the Mainline Programming Area.

LISTING PROGRAMS

After a program has been stored it is often desirable to obtain a printed listing of the program. This is done by setting the program line counter to the number of the line at which the listing is to begin (usually a listing of the entire mainline program area is desired — starting at line 0) and then pressing LIST.

Any of the following key sequences will list *all* programming stored in the mainline programming area.

```
END EXECUTE LIST
END LIST
GO TO 0 EXECUTE LIST
GO TO 0 LIST
```

3-10 THE MECHANICS OF MODEL 20 OPERATION

STORING PROGRAMS

LISTING PROGRAMS (cont'd)

The listing is terminated when STOP is pressed, or when the last line of programming has been listed; END and STOP statements within the programming will not halt a listing.

A listing of the program in the previous example looks like:

```
0:
0→A┌
1:
A+1→A┌
2:
PRT A┌
3:
GTO 1┌
4:
END ┌
R418
```

The last item on the listing is the number of R registers available. In this particular instance the listing indicates that R0 through R417 were available for the particular Model 20 configuration used.

After the listing is completed, the program line counter is automatically set to line 0 of the mainline programming area, so that the program is ready to run; simply press RUN PROGRAM.

A useful method to determine the number of R registers available without listing the entire program memory is to press LIST STOP. A partial listing will occur, followed by the number of R registers available.

MODIFYING INDIVIDUAL PROGRAM LINES

To modify a stored line of a program, it is necessary to have a replica of the line in the display. If a replica of the line is not in the display, one can be put there by pressing GO TO n EXECUTE RECALL, or just GO TO n RECALL, where n is the number of the line to be modified*. A replica of the line will appear in the display, in the same form that it appeared just after the line was first stored. The line itself is still stored in the program memory; the display will contain only a replica of the line, and not the actual line itself.

NOTE

The END statement has some very special properties. If a line containing an END is stored, or, if a stored line containing an END is modified, all lines of programming following that line are removed. For this reason the END statement is not usually combined with other statements in a line, but is used on a line by itself, and then only as the last line in the program.

*All that is really required is that the program line counter be at the line to be recalled before pressing RECALL.

Ignore the line number and the colon following it, and treat the ┌ at the end of the line as any other symbol or mnemonic. Modify the line using any of the methods described earlier in this chapter.

If your modification left the ┌ at the end of the line intact, press FORWARD until the ┌ reappears. *When it does, the line (as seen in the display) will automatically be stored in place of the original line in the program memory.*

If your modification of the line removed the ┌ from the end of the line, press FORWARD if necessary, to bring the end of the line into view. Then press STORE. At this time the original line is replaced with its modified replica.

It is quite possible that the modified replica and the original line are not the same length. This causes no problem at all; the stored lines following the altered line are automatically shifted to either make room for a longer line, or to take up memory left over from shortening a line. Also, the number of available R registers is adjusted, if necessary.

◆◆◆ INSERTING AND DELETING PROGRAM LINES ◆◆◆

In addition to modifying individual lines of a program (described in the previous section) it is also possible to insert entire lines into, or delete entire lines from, the interior of a program.

Say, for instance, we wished to add a line between line 4 and line 5. The added line would become the new line 5, while the old line 5 would automatically become the new line 6, etc.

Or, suppose we wished to remove line 3 from a program. The old line 4 would become the new line 3, and the old line 5 would automatically become the new line 4, etc.

In both cases, the number of available R registers is automatically adjusted after the change.

INSERTING LINES

To insert a line into a program, first set the program line counter to the line number that the new line is to have. Then write the line and press INSERT STORE. The new line will be stored, all succeeding lines of the program shifted to make room, and their line numbers increased by one. A replica of the line just inserted will appear in the display, just as for any line that has just been stored.

—— EXAMPLE ——

Suppose that you wished to add the line:

```
IF A>25;STP F
```

between lines 0 and 1 of the following program:

```
0: 0→A
1: A+1→A
2: PRT A
3: GTO 1
4: END
```

First, set the program line counter to 1, as that is to be the line number of the line after it is inserted.

PRESS: GO TO 1 EXECUTE

Now write the line:

```
IF A>25;STP F
```

Then PRESS: INSERT STORE

1: IF A>25;STP F

Pressing END LIST would list the program as follows:

```
0:
0→A←
1:
IF A>25;STP F
2:
A+1→A←
3:
PRT A←
4:
GTO 1←
5:
END F←
R417
```

DELETING LINES

To delete a line from a program, it is necessary to first place a replica of the line in the display. *The F at the end of the line must be visible in the display.* To get the replica in the display, set the program line counter to the line number of the line to be removed, and then press RECALL. Then press DELETE. The line will be removed, the lines following the deleted line shifted to take up the gap in the memory, and their line numbers decreased by one. After a line, say line number n, has been deleted, the display will indicate what the new line number n is; its replica will be in the display.

—— EXAMPLE ——

To delete the line that was inserted in the previous example (line 1),

PRESS: GO TO 1 RECALL DELETE

1: A+1→A←

3-12 THE MECHANICS OF MODEL 20 OPERATION

◆◆◆ INSERTING AND DELETING PROGRAM LINES ◆◆◆

DELETING LINES (cont'd)

A listing would reflect the change:

```
0:
0→AF
1:
A+1→AF
2:
PRT AF
3:
GTO 1F
4:
END F
R418
```

One thing to keep in mind when inserting or deleting lines is this: References to line numbers within the program itself may become incorrect, since it becomes quite possible that not all the lines comprising the program will have the same line numbers as they had originally.

◆◆◆ REPLACING A PROGRAM LINE ◆◆◆

There are three ways to replace an existing program line with another line.

The first way is to set the program line counter to the number of the line to be changed, write the line, and press STORE.

The second way is useful if you wish to verify that the line being replaced is actually the one you have in mind.

Set the program line counter to the number of the line, and then press RECALL. This brings a replica of the line into view. (If at this time you wish to use BACK to inspect the interior of the line for further verification, you may do so.) Now press CLEAR, write the new line, and press STORE.

The third method is to simply delete the line to be changed, and insert a new one in its place.

◆◆◆ ADDITIONAL PROPERTIES OF THE EDITING KEYS ◆◆◆

ADDITIONAL PROPERTIES OF RECALL

Pressing RECALL places into the display a replica of the stored line specified by the current setting of the program line counter, and then increments the value of the program line counter by one. Thus, pressing RECALL again brings into the display a replica of the next line in the program, etc. While the replica of a line is visible in the display, the line itself is still stored in the program memory.

If, after recalling every line of a program with RECALL, RECALL is pressed again, or (which is the same thing) RECALL is pressed while the program line counter is set to the first non-existent line number, the number of available R registers is printed, and the program line counter is set back to 0. Further use of RECALL will begin recalling lines of the program from line 0.

ADDITIONAL PROPERTIES OF BACK AND FORWARD

Once the replica of a stored line is visible in the display, pressing BACK will remove the right-most item of the line from view. If BACK is pressed enough times to remove the entire replica from view, pressing BACK one more time will bring a replica of the preceding line of the program into the display. It is then possible to step back through that line, and bring a replica of its predecessor into the display, etc. This process deals only with the replicas of the lines; the stored lines themselves are not altered by this process.

An analogous thing occurs when using FORWARD to step through the replica of a stored line. The difference here is that a replica of the successor of the current line is brought into the display, rather than a replica of its predecessor.

◆◆◆◆◆ RUNNING A PROGRAM ◆◆◆◆◆

If a program begins at line 0, the preferred method of starting it is to press:

```
END RUN PROGRAM
```

GO TO 0 RUN PROGRAM is not recommended, for reasons described in "END STATEMENTS", on page 5-36.

If the program begins at some other line number, n, the preferred method is to press:

```
END EXECUTE GO TO n
RUN PROGRAM
```

RUN PROGRAM executes the line in the display before beginning to run the program at the line number indicated by the program line counter. This is why it is not necessary to press END EXECUTE RUN PROGRAM; END RUN PROGRAM does the same thing.

To halt a program that is being run, press STOP one time for about a half second. The program

will halt after completing the line in progress when STOP was pressed. Depending upon the nature of that line, the display may contain a resulting quantity for the line.

If, after the program has been halted, STOP is pressed again, the mnemonic STP will appear in the display. If STOP is pressed again NOTE 01 will appear in the display, as STP STP is not a valid syntax.

After a program has been halted by pressing STOP (or by a STP within the program, or by an error during execution), do not press RUN PROGRAM to cause the program to continue, unless there are no subroutines or subprograms used in the overall program. If there are, start the program over from the beginning, instead. This is because halting the program in any way except with an ENTER statement (see "ENTER STATEMENTS", page 5-18) alters some of the internal conditions necessary to guarantee properly continued execution of subroutines and subprograms.

◆◆◆◆◆ ERRORS DURING EXECUTION ◆◆◆◆◆

Not all errors are detected when a line is written. Most syntax errors are detected when a line is written, but there is another type of error, called an error during execution, which is not.

For example, specifying an R register that does not exist, and writing an arithmetic expression that is an arithmetic impossibility due to the specific numbers involved (logarithms of non-positive numbers, division by zero, etc.) are things that result in errors during execution.

If the user wishes, he can instruct the Model 20 to allow most of the arithmetic faux pas that would otherwise result in errors during execution. A discussion about this, and a list of the errors involved is given in rule 11 in "RULES OF MATHEMATICAL COMBINATION", on page 4-10.

When a line executed from the keyboard (with EXECUTE) results in an error during execution,

the display will contain a NOTE. See Table 3-1 and also Appendix II for information about each individual NOTE. Pressing BACK will bring the line into view again. However, that is not required; no special action is required before continuing with your activities.

When a line of a program results in an error during execution, the program halts, and the words NOTE n IN m appear in the display. The number n indicates the nature of the error; m specifies the line number that resulted in the error. A replica of that line can be placed in the display by pressing RECALL. However, do not attempt to restart the program except from its beginning, unless you are certain that the error did not occur within a subroutine or subprogram.

The explanations in Chapter 5 of the individual syntaxes point out what these errors are, so that you may avoid them.

TRACE MODE OPERATION

The Model 20 has a feature, called the *Trace Mode*, which enables the Model 20 to make a printed record of its activities. The form this printed record takes depends upon what kind of activity is in progress.

The TRACE key is used to put the Model 20 into the Trace Mode. When the Model 20 is not in the Trace Mode it is in the *Normal Mode*. The mnemonic for the TRACE key is TRC; for NORMAL the mnemonic is NOR. The Model 20 will always be in the Normal Mode immediately after it is turned on.

TRC and NOR are each complete statements; no other keys are needed to complete the syntax of their respective statements.

The Calculator can be placed in the Trace Mode directly from the keyboard, or by command of a program. In either instance, the Trace Mode is established when a line containing a TRC statement is executed. The same thing applies to the Normal Mode and the NOR statement.

KEYBOARD CALCULATION IN THE TRACE MODE

While in the Trace Mode, the Model 20 prints an exact replica of each line executed from the keyboard, and the results of any activities in the line (for those activities which produce a quantity that can be considered a result; some keys, such as END and GO TO do not produce such a quantity).

The following example should speak for itself:

```

FXD 2┌
0→A;0→B┌
           0.00
           0.00
A+1→A;B+10→B┌
           1.00
           10.00
A+1→A;B+10→B┌
           2.00
           20.00
PRT "A=",A,"B=",
B┌
A=
           2.00
B=
           20.00

```

Those items followed by ┌ in the printout are the lines that were executed. Immediately below each line replica are the results of that line.

STORING LINES WHILE IN THE TRACE MODE

While in the Trace Mode, the Model 20 prints a complete replica of each line as it is stored. This feature is quite valuable when storing a program that you are making up as you go along; the Trace Mode provides you with a printed record to refer to, so you don't have to trust your memory to tell you what you have written so far.

The resulting printed record looks exactly like a listing, except that the number of R registers available does not appear on the printout. That too is easily obtained by pressing RECALL after the last line has been stored. However, first be sure that you have stored the last line of the program, as the program line counter is set to zero after the number of available R registers is printed. If you decide to add more lines onto the end of the program, you will first need to set the program line counter back to the next line number to occur in the program, before writing and storing any lines.

RUNNING PROGRAMS IN THE TRACE MODE

While running a program in the Trace Mode the Model 20 prints the line number of each line as it is executed, and below that, any quantities that were stored into registers by that line. Figure 3-4 shows the resulting printout when the program in Figure 3-3 is run.

```

0:
FXD 2;TRC ┌
1:
10→A┌
2:
20→B;30→C┌
3:
NOR ;STP ┌
4:
END ┌
R425

```

Figure 3-3. A Sample Program to be Run in the Trace Mode.

TRACE MODE OPERATION

```

1:
2:      10.00
3:      20.00
4:      30.00
5:
6:
7:
8:
9:
10:
11:
12:
13:
14:
15:
16:
17:
18:
19:
20:

```

Figure 3-4. Results of Running the Program of Figure 3-3.

Running a program in the Trace Mode is a frequently used method to help debug a program that is not operating as intended. It allows you to analyze the numbers stored during the execution of the program. However, it is not intended to be a means of permanently recording data associated with a given problem which the Model 20 is to solve; PRINT is far better suited for that purpose.

Both TRACE and NORMAL are programmable, and a program can establish the Trace Mode for all or part of a program, by having TRC and NOR statements properly located in it. A program can (without alteration) be run in the Trace Mode simply by pressing TRACE EXECUTE before starting to run the program. This is fine for short programs which are not especially complex, or do not have especially long execution times. However, the usual way of using TRACE with a

complex program, or one with a long execution time, is to insert TRC and NOR statements into selected parts of the program. After the problem has been found and corrected, the statements are then removed. There are several advantages to this procedure.

First, since only the part of the program suspected of containing the error is run in the Trace Mode, the amount of time spent printing is reduced to a minimum. Secondly, the amount of information to be evaluated is held to just the necessary amount, and no more. Last, but not least, it saves printer paper.

For instance, if you wished to modify a program such that it establishes the Trace Mode for lines 13 through 15, add a TRC statement to the end of line 12 and a NOR statement to the end of line 15. This procedure assumes that there are no GO TO's or other branching that can lead to line 13, without going through line 12 first.

The Model 20 can be placed in the Trace Mode *while running any program* (which does not execute its own NOR statement) simply by pressing TRACE. *It is not necessary to halt the program first.* Similarly, the Normal Mode can be reestablished, while the program is running, by pressing NORMAL.

REVIEW

The following important concepts were presented in this chapter:

1. The fundamental process of Model 20 operation is to write a line by pressing the keys that generate a display that conveys your intentions. Then the line is either executed or stored.
2. The maximum number of keystrokes permitted in a line varies with the circumstances. The maximum number is about 68, and the probable minimum is 35, with 50 being typical.
3. If you wish to rewrite a line from its beginning, press CLEAR, and then begin writing the line again.
4. By using BACK, FORWARD, INSERT, and DELETE, it is possible to add, remove, or alter one or more segments of a line without rewriting the entire line.
5. The mainline programming area begins at line 0. The program line counter determines which line of a program is under consideration. Lines of programming must be stored in the order of their intended line numbers.

3-16 THE MECHANICS OF MODEL 20 OPERATION

REVIEW

(cont'd)

- By using BACK, FORWARD, INSERT, and DELETE, it is possible to add, remove, or alter one or more lines of a program without rewriting the entire program.
- Syntax errors must be corrected before the line can be completed or used in any way.
- Errors during execution halt a program.
- STOP also halts a program.
- Once a program has been halted, it should not be started over except from its beginning; unless it is known that the program was not halted inside a subroutine or subprogram.
- The Trace Mode provides a printed record of Model 20 activities. When the Model 20 is not in the Trace Mode it is in the Normal Mode. Both TRC and NOR are programmable.

EXERCISES

Answers are on page A-1.

- Suppose you were writing a long line and forgot what the front part of it looked like. How could you find out?
- Suppose you were writing a long line composed of many statements, and the words NOTE 09 appeared in the display. What is wrong and how do you correct it?
- You are trying to key in the following line, but get NOTE 01 in the display before it is completely entered. Why?
 $10 \rightarrow A, 20 \rightarrow B$
- You have just written this much of a line:
 $12.5(X-2.1)/18.6+Y!Y+$
Now you discover that the + should have been a -. What is the easiest way to correct the mistake?
- What sequence of keys would you press to change the line:
 $r(A+B) \rightarrow X; X/AB \rightarrow Y$
into the line:
 $r(A+B) \rightarrow X; r(AB) \rightarrow C; X/AB \rightarrow Y$
without pressing CLEAR and rewriting the line?
- You have just stored the first four lines of the only program that is currently stored in the Calculator. Upon executing the line:
 $GTO 5$
NOTE 08 appears in the display. What is wrong and why?
- You have just written and stored a program while in the Trace Mode. Without listing the program with LIST, how can you find out how many R registers are left?
- You wish to replace line 3 of a 10 line program with an altogether different line of different length. What sequence of keys would you press if the new line is:
 $A+1 \rightarrow A$
- Both STORE and EXECUTE operate only on what is 'visible' in, and 'to the left of', the display.
Suppose you keyed in the line:
 $PRT AB,C$
and discovered that a comma was missing between A and B.
Now suppose you used BACK and INSERT to put the comma after A, but did not use

EXERCISES

FORWARD to bring the rest of the line in view, leaving the display like:

```
PRT A;
```

What would happen if you now pressed EXECUTE or STORE?

10. Suppose you wanted to modify a 60 line program by replacing line 31 with an altogether different line. The original line 31 was 10 keystrokes long, and its replacement is 60 keystrokes long. While trying to store the new line, NOTE 12 appears in the display. What is wrong?

11. You have properly loaded a 40 line program into the Calculator. Suppose that some of its lines are:

```
15: PRT "ROOT 1=";A+B;END
21: PRT "ROOT 2=";A-B;END
29: PRT "NO SOLUTION";END
39: END
```

(Among other things, an END in a program acts like a 'go to line zero and stop'.)

Beginning with line zero, you use a series of consecutive RECALL's to step through the program on a line-by-line basis. After recalling line 23, you switch to a series of consecutive FORWARD's to step through each remaining line of the program. Eventually, the display gets 'stuck' as shown, and does not change as FORWARD is pressed.

```
29: LUTION";END F
```

Pressing RECALL results in the number of R registers being printed, and a listing of the program reveals that lines 30 through 39 are missing. What happened?

4-0

NOTES

Chapter 4

NUMERICAL COMPUTATIONS

The purpose of this chapter is to explain how the Model 20 language is used to formulate mathematical expressions. Mathematical expressions consist of numerical quantities and mathematical operations. Numerical quantities can be classified as *constants* and *variables*, and mathematical operations as *arithmetic operations* and *functions*.

The first part of this chapter deals with the numerical quantities that can be handled directly with the basic Model 20, and with the nature of the mathematical operations that can be performed with them. Some mention of the Mathematics ROM (Math ROM) is included so that a complete description of the nature of mathematical operations on real numerical quantities can be developed.

The latter portion of this chapter is a complete presentation of the general rules of formulating mathematical expressions.

NUMERICAL QUANTITIES

Numerical quantities in the basic Model 20 can be represented as real constants or as real variables*. The next four sections explain how this is done.

REAL CONSTANTS

A real constant is a number that is represented as itself, as a series of digits; e.g., 3.1416, 186000, 1.86E5. (There is another way to represent a number, which does not directly involve the digits of the number.) A precise definition of a constant is given in Chapter 5.

Commas cannot be inserted into numbers for the sake of their readability; an attempt to do so will generate a syntax error (NOTE 01).

There are two forms in which real constants can be written: *fixed point* and *floating point*.

A fixed point number is one which is written with the decimal point in its true location. For example, the numbers below are all written in fixed point.

```
3.1416   -10   -10.   .0005
```

A floating point number is written in a manner

*The term 'real' denotes that the quantity is a *representation* of a number that is real in the mathematical sense, as opposed to a complex number of the form $a + bi$. However, a 'real constant' or 'real variable' is, in the strictest mathematical sense, only a rational number. The square root of two, for instance, can only be represented as a finite number of digits.

which corresponds to scientific notation: as a fixed point number multiplied by some integral power of ten. The integral power used is called the *exponent* of the number. However, rather than write 1.86×10^5 , we let the character E represent the symbols $\times 10$, and write 1.86E5. E is the mnemonic for the ENTER EXP key (EXP is an abbreviation for EXPONENT).

The exponent of a floating point number may be signed or unsigned, but must contain one or two digits** and no decimal point or register names. The largest exponent is E99, and the smallest is E-99. The exponent may also be zero. Some valid and non-valid exponents are shown below.

E03, E3, E+03, and E+3 are all valid and equivalent.

E-05 and E-5 are valid and equivalent.

E100 and E-100 are not valid exponents. Each will cause an error during execution; no more than two significant digits can follow E.

E(1+1) and E2.2 are not valid exponents. Each will cause a syntax error (NOTE 01). The digits 0 through 9, +, and - are only items which can immediately follow E.

**The only exception to the one or two digit rule is this: leading zeros are permitted in any exponent, and may make it longer than two digits. Thus, 2.2E11 and 2.2E0011 are both valid ways of writing the same number.

4-2 NUMERICAL COMPUTATIONS

NUMERICAL QUANTITIES

REAL CONSTANTS (cont'd)

EY, E-Y, and E(A + B) are not valid exponents. Each will cause a syntax error. Register names cannot be used in the specification of an exponent.

E, E+, and E- are not in themselves valid exponents; E must be followed by at least one digit.

The fixed point part of a floating point number can be written in any permissible fixed point form. All of the following are permissible floating point numbers:

-1.23E5	68.2E-2
.0001E-5	1512E6

There is an easy way to write numbers of the form 1×10^n , where $-99 \leq n \leq 99$. Instead of writing forms like .001, or $1E-3$ (they are the same number), simply write E-3. Similarly, instead of writing -1000, or $-1E4$, simply write -E4. When E is not immediately preceded by a numerical quantity, the Model 20 assumes 1E is intended.

When a line containing a floating point constant is stored, the following conventions apply:

- A leading plus sign on the exponent following E is dropped. Thus, $2.4E+6$ is stored as $2.4E6$.
- Leading zeros in an exponent are retained, but not counted as part of the two digits allowed following E. Thus, $5.2E0011$ is stored as $5.2E0011$.
- Numbers of the form $E \pm n$ are stored as $1En$ or as $1E-n$. Thus, $E10$ is stored as $1E10$, while $E-10$ is stored as $1E-10$.

Any listing or display of stored lines containing floating point constants will conform to these conventions. The printed or displayed result of any manipulation of numbers, when in floating point, also follows those conventions.

To write a real constant, press the keys which correspond to the symbols and digits (from left to right) that make up the number.

To write the number -1.34,

PRESS:

To write the number $1.26E-5$,

PRESS:

REAL VARIABLES

A real variable is a register containing a number that could be written as a real constant. The name of the register can be used in place of the number itself when writing mathematical expressions. The Model 20 language also permits other types of variables. A precise definition of a variable is given in Chapter 5.

The real variables available in the basic Model 20 are the registers A, B, C, X, Y, Z, and the available R registers.

DESIGNATING A REGISTER

Each of the registers A, B, C, X, Y, and Z, has a key which corresponds to that particular register, and no other. The mnemonics of each of these keys are simply the names of their respective registers: A, B, C, X, Y, and Z. To designate one of these registers, simply have its mnemonic appear in the appropriate manner in the line being written.

An R register is designated by writing R (with the R () key) followed by a (non-negative) value which specifies the intended register. The value can be a real constant, or a variable name, or a mathematical expression whose value is to be computed.

The mnemonic for the R () key is simply R*. The value following R may or may not need to be

*One of the keys in the right-most block of half-keys is the R key. Under the proper circumstances, it also has the mnemonic R. However, this is never cause for confusion about which key to press, or which keys were pressed to produce a particular display. The R half-key has nothing whatever to do with R register designation; it is used to specify some activity defined by a plug-in ROM, or to generate the character R when it is to appear in a quote field. The R () key produces a colon when used in a quote field.

NUMERICAL QUANTITIES

enclosed within parentheses, although that is always permitted. The following line segments all designate R35:

R35 R0035 R(30+5)

R(41-6) R(70/2) R(R5+C)

provided $35 \leq R5+C < 36$.

No parentheses are required when an R register is designated by a series of digits immediately following R, such as R100; however, R (100) is acceptable. Parentheses are required when the quantities following R are to be grouped together and treated as a unit. Thus, R(70/2) denotes R35, while R70/2 denotes one half of the number contained in R70. However, forms like $R\sqrt{4}$ (meaning R2) do not require parentheses, as there is only one possible interpretation.

Variable names can appear as part of the value following R. Thus, if register A contains the number 15, RA denotes register 15. If R5 contains the number 10, and C contains the number 25, R(R5+C) denotes R35.

The register denoted by the form RRR ... R <value>* is determined by the value and by the numbers contained in the various registers. For example, RR2 denotes R8 if R2 contains the number 8; in general, it denotes Rn where R2 contains the number n. The length of the form RRR ... R <value> is limited only by the maximum length of the line containing it.

When the value following R is not strictly an integer, the fractional part of the value is ignored. Thus, R35.6 denotes R35.

A plus sign immediately following R is dropped when the line containing it is stored. R+35 is stored as R35. The form R- is not permitted, and causes a syntax error (NOTE 01). R(-....) is permitted, except as noted below.

If R is followed by a value whose integer part is less than zero, or greater than the number of available R registers, an error during execution

*The symbols <.....> denote that whatever is inside them is only the name of the thing which we wish to indicate as being in that location, rather than the thing itself. In this case, R <value > means R followed by some quantity or expression.

results. (The indication will be either NOTE 05 or NOTE 06, depending upon the exact circumstances.)

SIGNIFICANT DIGITS

When a number is keyed in as a real constant, it can contain as many digits as desired, and the length of the line permits. However, to key in more than 12 *significant* digits is wasteful, as when the line containing the number is executed, the additional digits are treated as if they were zeros. In every case, however, the number of digits to the left of the decimal point, or the number of consecutive leading zeros to the right of the decimal point, are counted, so that the true location of the decimal point will be properly understood.

As a line containing real constants is executed, the real constants are converted into an internal form of 12 significant digits, which is digit-point-digit-digit-... exponent. All real constants are converted to this form before they are stored into any registers, or used in any mathematical computation.

For example:

- (1) 1234567890123456 is stored in a register as:
- (2) 1.23456789012E15

The last four digits of the original number (1) are lost.

Note, however, that the number:

- (3) .0000000000000001234567890123456 if keyed in as shown, is stored in a register as:
- (4) 1.23456789012E-16

The consecutive leading zeros after the decimal point in (3) are not significant.

Either of the numbers (1) and (3) in the example above could be the fixed point portion of some floating point number which has its own exponent. When a line containing such a number is executed, the number is converted to the digit-point-digit-digit-... exponent form. During this process, the proper exponent for the internal form is computed. The computed exponent may well have a higher absolute value than the original

NUMERICAL QUANTITIES

SIGNIFICANT DIGITS (cont'd)

exponent assigned to the number. That is perfectly permissible as long as the computed exponent does not have an absolute value greater than 99. This is just another way of saying that numbers whose absolute values are greater than $9.9999999999999999E99$, or less than $1E-99$ (except for zero), cannot be entered.

Although all data is understood only to 12 significant digits, no more than 10 can be displayed or printed; the last two digits (called the *guard digits*) are used to maintain 10 place accuracy.

The user determines the form in which numbers are printed or displayed (other than during an actual keyboard entry or during a listing) by means of the FIXED N and FLOAT N keys. N represents the desired number of digits to be shown to the right of the decimal point, and may range from 0 to 9, inclusive.

Under a floating point specification, numbers will always be displayed or printed in the form digit-point-digit-digit-... exponent.

A number shown under a FIXED N or FLOAT N specification will always be correctly rounded to

reflect the value of the digits that are not visible.

Not every number can be represented with all (or sometimes any) fixed point specifications, as some numbers may have absolute values that are too large for the current (or maximum) fixed point specification*. When the current fixed point specification cannot represent a number, the form is momentarily changed, for that number only, to the previous floating point specification. After that, the form reverts back to the current fixed point specification.

It sometimes happens that a number, whose absolute value is less than 1, cannot be represented in fixed point except as .00000 There is no automatic conversion to floating point under these circumstances.

These and further properties of the FIXED N and FLOAT N keys are fully explained in "FIXED AND FLOAT STATEMENTS", on page 5-11.

*Physical limitations are part of the reason for this. To display a fixed point number in the vicinity of $\pm 10^{45}$ would require a display of at least 48 characters (one for a sign, one for the decimal point, and 46 for the number). Even if this were done, only the first 12 digits of the number are significant, anyway.

MATHEMATICAL SYMBOLS

The mathematical symbols consist of the *arithmetic operators*, and the symbols and mnemonics for functions.

ARITHMETIC OPERATORS

The arithmetic operators are used to indicate arithmetic operations among variables and constants. The arithmetic operators available with the basic Model 20 are shown in Table 4-1.

The plus sign is used in exactly the same way as in ordinary mathematical notation. A unary plus sign* is usually used when some sort of visual emphasis is to be placed on the fact that it is the actual sign of the quantity (whatever it is) and

not its opposite sign that is intended. Since no sign in front of a quantity means the same thing, the unary plus sign is not frequently used in the Model 20 language; in almost every case the Model 20 drops a unary plus sign when the line containing it is stored.

The minus sign is also used in exactly the same way as in ordinary mathematical notation. The special usage of placing a minus sign in front of a quantity to indicate that the negative of that quantity is intended, is called unary minus.

NOTE

A plus or minus sign following E in an exponent of a floating point number is not considered an arithmetic operator; it is simply part of the number.

*The term unary comes from the fact that in this usage there is only one number or quantity associated with the operation; usually there are two, one on either side, and the operation is called a binary operation.

MATHEMATICAL SYMBOLS

Table 4-1. The Arithmetic Operators of the Basic Model 20.

NAME	SYMBOL	USE
plus	+	addition: $2+3$, $A+B$, $5+R1$, etc.
unary plus	+	exactly in the same way as in ordinary mathematical notation: $+A$ is identical to A .
minus	-	subtraction: $2-3$, $A-B$, $5-R1$, etc.
unary minus	-	indicates the negative of a quantity: -10 , $-(A+B)$, etc.
times	*	multiplication: $2*5$ (2 times 5), $A*B$, etc.
implied multiply	none	multiplication: AB instead of $A*B$, $5(B+C)$ instead of $5*(B+C)$, A^2 instead of $A*2$, etc.
slash	/	division: $4/2$ (4 divided by 2), A/B , $A/(X+Y)$, etc.

The times sign is * rather than X or ·, in order not to produce confusion between the times sign and the variable X, or the decimal point, respectively.

Notice that multiplication can be indicated simply by placing the quantities, to be multiplied, side by side. In many cases it is necessary to place one or more of the quantities within parentheses for grouping, or because of the presence of a unary plus or minus sign attached to any but the left-most factor.

THE ASSIGNMENT INSTRUCTION

A much used instruction in mathematical statements is the assignment instruction: \rightarrow . It stores the numerical value specified on the left into the register named on the right. The various uses of the assignment instruction are fully explained in 'ASSIGNMENT STATEMENTS', on page 5-16.

We have already seen several examples of one use of the assignment instruction. For example, recall the statements $5\rightarrow A$, which stores the number 5 into register A ('assigns A the value 5'), and $B+1\rightarrow B$, which adds one to the value stored in B.

MULTIPLE ASSIGNMENT STATEMENTS

There are other useful forms of assignment statements, one of which is:

⟨value⟩ \rightarrow ⟨register name⟩ \rightarrow . . . \rightarrow ⟨register name⟩

For instance, $0\rightarrow A\rightarrow B\rightarrow C$ stores the number zero into the three registers A, B, and C.

Henceforth, increasing use will be made of notations like:

⟨mathematical expression⟩ \rightarrow ⟨real variable⟩

The major reason for this is that in the majority of cases, such notations and conventions will shorten and simplify the text, while at the same time providing very precise meanings. In the specific example above, we shall say that the assignment instruction assigns the variable the value of the expression.

A sometimes useful form is illustrated by the statement:

$$A+B\rightarrow X-Y\rightarrow C$$

This equivalent to the separate statements:

$$A+B\rightarrow X$$

$$X-Y\rightarrow C$$

MATHEMATICAL SYMBOLS

MULTIPLE ASSIGNMENT STATEMENTS (cont'd)

The first item immediately to the right of each \rightarrow must be a register name.

This form is often used to save an intermediate result of a computation.

When a form like this is stored, parentheses are added around various segments of the statement, involving all \rightarrow 's except the right-most one. For instance,

$$A \rightarrow B \rightarrow X - Y \rightarrow C$$

is stored as:

$$(A + B \rightarrow X) - Y \rightarrow C$$

and

$$R1 / R2 \rightarrow X R5 \rightarrow A + 2B \rightarrow Y$$

is stored as:

$$((R1 / R2 \rightarrow X) R5 \rightarrow A) + 2B \rightarrow Y$$

If the parentheses were used when the line was originally written, no additional ones will be added when the line is stored.

Statements of the form shown below *do not* receive additional parentheses when they are stored.

$$0 \rightarrow A \rightarrow B \rightarrow C$$

FUNCTIONS

A function is a computational rule or process which operates upon a value or values supplied to it (called the argument of the function), from which a result is computed, called the value of the function.

The Model 20 language provides for the use of the more common mathematical functions. The basic Model 20 has the square root function (whose

Table 4-2. Functions and Operations Supplied by the Math ROM.

FUNCTION OR OPERATION	MNEMONIC OR SYMBOL	NAME
$\ln x$	LN	natural logarithm
e^x	EXP	exponential function
$\log x$	LOG	common logarithm
$\log^{-1} x$	TN†	antilog (base 10)
a^b	† †	exponentiation
$\sin \theta$	SIN	sine
$\cos \theta$	COS	cosine
$\tan \theta$	TAN	tangent
$\sin^{-1} x$	ASN	inverse sine, or arcsine
$\cos^{-1} x$	ACS	inverse cosine, or arccosine
$\tan^{-1} x$	ATN	inverse tangent, or arctangent
$ x $	ABS	absolute value
integer of x	INT	integer
π	π	pi
selection of degrees, radians, or grads	TBL	Table

† † by itself is not permitted. The simplest form is $\langle \text{quantity} \rangle \uparrow \langle \text{quantity} \rangle$; a^b is represented by $A \uparrow B$.

MATHEMATICAL SYMBOLS

symbol is Γ), while the Math ROM provides the additional functions and operations shown in Table 4-2.

The Math ROM is fully described by its operating manual. It is mentioned here only for the sake of completeness in the presentation of the rules for forming mathematical expressions.

In the Model 20 language, functions can be classified as *supplied functions* or as *definable functions*. A supplied function is one whose computational rule or procedure is determined by the design of the basic Model 20 or by the design of any of the plug-in ROM's. The only supplied function available with the basic Model 20 is Γ .

Definable functions are functions which have all of the operational properties of the supplied functions, as described below, but whose computational rules or procedures are specified by the user himself. These very powerful functions are available with the Definable Functions ROM.

Each of the supplied functions mentioned to this point has one argument*. (Exponentiation, a^b is represented by $A \uparrow B$, is considered an arithmetic operation, and not a function.) The argument of each of these functions can be a real constant, a variable, an entire mathematical expression, or even another function. Once the argument of a function is specified, the mnemonics and symbols representing the function and its argument are just another name for a number (although you might not know what that number is). As such, the function can be treated in the same way as any number. For example, all of the following statements are valid:

$\Gamma 4 \rightarrow A$	$\Gamma \Gamma 16 \rightarrow A$
$10 \Gamma 4 \rightarrow A$	$\Gamma (A+B) \rightarrow X$
$(\text{SIN } 30) (\text{SIN } 30) \rightarrow X$	$X \Gamma (A+B) \rightarrow Y$
$\text{SIN SIN } 30 \rightarrow X$	$\uparrow \text{SIN } 30 \rightarrow Z$

*It is possible, with the User's Definable Functions ROM, to create definable functions that have multi-parameter arguments. In mathematical terms, these are functions of more than one variable.

SPECIAL CONVENTIONS

Throughout the manuals for the Model 20 System, special meanings are assigned to various words. Among those are the words *variable*, *constant*, *quantity*, *expression*, and *value*.

Variables and constants have already been mentioned in the beginning of this chapter. All of these terms are rigorously defined in Chapter 5. For now, an intuitive understanding of the remaining terms is sufficient. Refer to Figure 4-1.

A quantity is any *single* variable or constant.

An expression is a *combination* of variables, constants, functions, and operators.

A value is a quantity or expression.

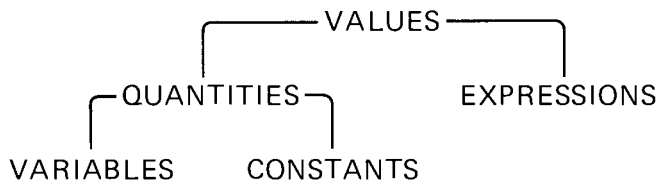


Figure 4-1. The Relationship of the Special Mathematical Terms.

EXAMPLES

A is a variable, a quantity, and a value.

-3.14 is a constant, a quantity, and a value.

-3.14 could also be considered an expression.

$-A$ is *not* a variable — it is an expression, and a value. (It can't be a variable — there is no register named 'minus A'.)

$A+B$ is an expression, and a value.

RA
 $R(A+B)$
 RRA } are variables, as they are simply names of registers. (The $(A+B)$, if separate, would be an expression, but the language unit $R(A+B)$ is simply a variable.)

MATHEMATICAL SYMBOLS

HIERARCHY

The mathematical operations have a *hierarchy* which, in conjunction with parentheses, specifies the order in which the various operations will be performed when evaluating a mathematical expression. The hierarchy is:

- First: Functions
- Exponentiation
- Unary Minus
- Implied Multiply
- Explicit Multiplication, and Division
- Addition and Subtraction, and Unary Plus
- Last: Relational Operators*

Under the hierarchy, *and assuming the absence of parentheses*, any functions in an expression are performed first, then any exponentiations (available with the Math ROM), then any implied multiplications, and so on. If there is more than one function, or more than one exponentiation, etc., they are performed from left to right. As each operation is performed it is replaced by its numerical result and the process continues.

EXAMPLES

$$A + B - C \div X$$

Diagram showing evaluation order for $A + B - C \div X$. Brackets and arrows indicate: 1. $A + B$, 2. $C \div X$, 3. $(A + B) - (C \div X)$.

$$AB \div CX * Z \div Y$$

Diagram showing evaluation order for $AB \div CX * Z \div Y$. Brackets and arrows indicate: 1. AB , 2. CX , 3. $AB \div CX$, 4. $Z \div Y$, 5. $(AB \div CX) * (Z \div Y)$.

$$2A \uparrow 3 + r7B \div X \div Y$$

Diagram showing evaluation order for $2A \uparrow 3 + r7B \div X \div Y$. Brackets and arrows indicate: 1. $r7B$, 2. $2A \uparrow 3$, 3. $2A \uparrow 3 + r7B$, 4. $X \div Y$, 5. $(2A \uparrow 3 + r7B) \div (X \div Y)$, 6. $2A \uparrow 3$, 7. $(2A \uparrow 3) \div ((2A \uparrow 3 + r7B) \div (X \div Y))$.

PARENTHESES

In general, parentheses are used to group quantities and to indicate multiplication in the same way as in conventional mathematical notation. The Model 20 language also has special uses for parentheses that are mentioned as they occur.

For instance, to multiply $9A+7B$ by $4X+5$, you could write either of:

$$(4X+5)(9A+7B) \text{ or } (9A+7B) * (4X+5)$$

Parentheses can be 'nested', that is, a portion of an expression enclosed in parentheses can enclose a portion of itself in parentheses:

$$6((R1+R(R7+R8))/(R2+(R3/A))) \div X$$

The 'depth' to which parentheses can be nested (the number of times parentheses can contain parentheses which contain parentheses . . .), is limited only by the maximum length of the line containing the nest. When a line containing parentheses is stored, the parentheses are removed by the compiler. Abundant use of parentheses in a program does not increase the amount of memory needed to store it.

The evaluation of an expression within a level of parentheses (that is, between a '(' and its associated ')') proceeds according to the same rules as before, i.e., evaluation from left to right by levels of the hierarchy. If this process encounters yet another level of parentheses, evaluation of the current level ceases until the new level is evaluated. *That* level is also evaluated by a left to right application of the hierarchy, and so on. As evaluation of each level of parentheses is completed, the evaluation of the previous level is resumed.

Thus in the expression:

$$10(A \div LN (B+C) + X) \div Y$$

The value $(B+C)$ is found before any other computations are performed. Note that $(B+C)$ is also the deepest nest.

However, the deepest nest is not always evaluated first. Consider:

$$rA * (B \div (X+Y)) \div Z$$

Here the square root of A is found first, and then $(X+Y)$ is found, so that it can be divided into

*The Relational Operators ($=$, \neq , $>$, and \leq) have not been discussed yet, but since they are subject to the hierarchy, they have been shown in the list. See 'OPERATORS', on page 5-3.

MATHEMATICAL SYMBOLS

B. Finally the multiplication is performed and the result stored into Z.

Thus, by grouping things together, parentheses can be used to force operations to be performed in a desired order. Consider the following additional examples, and the effect of parentheses for grouping upon the order of operations.

———— EXAMPLES ————

Suppose that it was your intention to:

divide AB by $-2R3 * 4R6$.

Realizing that $AB / -2R3 * 4R6$

is incorrect because $-2R3$ will not be multiplied by $4R6$ before the division, parentheses are used to produce the desired result. It can be done in several ways.

One way is simply to group the entire denominator together:

$$AB / (-2R3 * 4R6)$$

Another way would be to make all of the multiplications in the denominator be implied multiplications, so that they would be accomplished prior to the division:

$$AB / (-2R3) (4R6)$$

Perhaps the best way would be to write:

$$AB / -8R3R6$$

—————
The representation of compound fractions is another area where parentheses play an important role in determining the order of operations. To

represent the compound fraction:

$$\frac{\frac{a}{b}}{\frac{x}{y}}$$

you would write:

$$(A/B) / (X/Y)$$

Simply writing:

$$A/B/X/Y$$

represents the fraction $\frac{a}{bxy}$, which is definitely not what was intended*.

—————
Consider the following pair of expressions:

$$\Gamma A+B \text{ meaning } (\sqrt{a}) + b$$

$$\Gamma 4A \text{ meaning } (\sqrt{4}) a$$

While it is not surprising that the top expression means $(\sqrt{a}) + b$ and not $\sqrt{a+b}$, many people at first make the error of assuming the second statement means $\sqrt{4a}$ rather than $(\sqrt{4})a$. The correct way of writing $\sqrt{a+b}$ and $\sqrt{4a}$ are shown below.

$$\Gamma (A+B) \text{ meaning } \sqrt{a+b}$$

$$\Gamma (4A) \text{ meaning } \sqrt{4a}$$

—————
*Of course, the compound fraction $\frac{\frac{a}{b}}{\frac{x}{y}}$ could always be written in the form $\frac{ay}{bx}$, which is simply AY/BX .

IMPLIED STORE INTO Z

THE NOTION OF IMPLIED STORE

A statement involving numerical activity usually contains an instruction, such as PRT, DSP, or \rightarrow . If there is no such instruction, the form \langle quantity $\rangle \rightarrow Z$, or \langle mathematical expression $\rangle \rightarrow Z$, is usually

automatically assumed when the line is executed or stored.

The automatic addition of $\rightarrow Z$ onto the end of a statement is called the 'implied store in Z'.

4-10 NUMERICAL COMPUTATIONS

IMPLIED STORE INTO Z

THE NOTION OF IMPLIED STORE (cont'd)

For instance, if you press A EXECUTE to view the contents of A, the line $A \rightarrow Z$ is what is actually executed. The contents of A are seen because that is the numerical result associated with the last assignment instruction executed in the line. Meanwhile, the contents of Z have been replaced by those of A, and are lost. The recommended procedure for viewing the contents of a register is to use the PRINT or DISPLAY statements, as they do not disturb the contents of any registers.

Because of the implied store into Z, the Z register is not recommended for storing data during calculations performed from the keyboard, except in certain situations. For instance, suppose you wished to add a series of numbers: n_1, n_2, n_3, \dots

First, set Z to zero by executing the line $0 \rightarrow Z$. Then, add the numbers by executing the following lines:

$$\begin{aligned} \langle n_1 \rangle + Z \\ \langle n_2 \rangle + Z \\ \langle n_3 \rangle + Z \end{aligned}$$

Because of the implied store into Z, this is what is

actually happening:

$$\begin{array}{ll} 0 \rightarrow Z & \\ n_1 + Z \rightarrow Z & n_1 + 0 \rightarrow Z \\ n_2 + Z \rightarrow Z & n_2 + n_1 \rightarrow Z \\ n_3 + Z \rightarrow Z & n_3 + (n_1 + n_2) \rightarrow Z \\ \vdots & \vdots \end{array}$$

RAMIFICATIONS OF THE IMPLIED STORE

Implied store can also apply to multiple assignment statements. Consider the statement:

$$(A+B \rightarrow X) - Y$$

It will be executed or stored as the statement:

$$(A+B \rightarrow X) - Y \rightarrow Z$$

However, if the statement were written as:

$$A+B \rightarrow X - Y$$

it would be executed or stored as the statement:

$$(A+B \rightarrow X) - Y$$

The extra parentheses are added, but since the original statement already had a \rightarrow that was not enclosed in parentheses, the additional $\rightarrow Z$ was not added.

RULES OF MATHEMATICAL COMBINATION

This section sets forth the rules to be followed when formulating mathematical expressions.

1. With the exception of rule 2 below, two operators must not appear side by side.
2. One or more unary plus or minus signs may appear together or follow a $*$ or $/$ or a relational operator.

$---B$ is equivalent to $-B$

$--B$ is equivalent to B

$+++B$ is equivalent to B

$A+-B$ is equivalent to $A-B$

$A/-B$ is equivalent to $A/(-B)$

X/Y is also permitted.

$A*-B$ is equivalent to $-AB$ or $-(AB)$

3. When two quantities or expressions are written side by side it is understood that they are to be multiplied together (implied multiplication).
4. Parentheses must be used to indicate groupings. Parentheses may be nested. Whenever parentheses are used, they must contain at least one real constant, or one variable; $()$ is not permitted. Parentheses must be balanced; there must be the same number of $($ as there are $)$ at the time the line is executed or stored.

◆◆◆ RULES OF MATHEMATICAL COMBINATION ◆◆◆

5. The order of operations within an expression is from the left to right by levels of the hierarchy:

- First: Functions
 - Exponentiation
 - Unary Minus
 - Implied Multiply
 - Explicit Multiplication, and Division
 - Addition, Subtraction, and Unary Plus
- Last: Relational Operators

6. When an expression contains other expressions enclosed within parentheses, the order in which the expressions are evaluated depends upon their location with respect to the other elements of the overall expression and the hierarchy. When such a nested expression is reached, the evaluation of the outer expression is temporarily suspended until the nested one is evaluated. Then the evaluation of the outer expression is resumed.

7. When in doubt about the order in which the operations of an expression will be performed, parenthesize!!

8. A mathematical expression must not start with an operator, unless that operator is unary plus or unary minus.

9. Unless the argument of a supplied function is a single real variable, or single real constant, or another function, the argument must be enclosed in parentheses. Also, if the argument is to be preceded with a unary minus, the entire argument (including the minus sign) must be enclosed in parentheses*. Thus:

$$r(A+B) \text{ represents } \sqrt{a+b}$$

$$r(AB) \text{ represents } \sqrt{ab}$$

$$r(-A) \text{ represents } \sqrt{-a}$$

$$\text{SIN}(-30) \text{ represents } \sin -30$$

10. The expression $A \uparrow B \uparrow C$ is treated as $(A \uparrow B) \uparrow C$; representing $(ab)^c$. If $a(b^c)$ is desired, write $A \uparrow (B \uparrow C)$. (In general, $(ab)^c \neq a(b^c)$; for instance, $(2^2)^3 = 4^3 = 64$, whereas $2(2^3) = 2^8 = 256$.)

11. Mathematical indiscretions, such as division by zero, and taking the square root of a negative value, result in errors during execution. If the user sets flag 14 prior to the error (see "FLAGS", page 5-2, for information about flags) such mathematical errors will be ignored. Otherwise, they will place a NOTE in the display, and halt any program being run.

Regardless of whether flag 14 is set or not, such a mathematical indiscretion will cause flag 15 to be set. Thus, by checking flag 15 after a problem has been solved, it is possible to determine whether or not any such errors occurred, even though they may have been ignored.

The mathematical errors which can cause an error during execution are listed below.

- i. Division by zero.
- ii. Square root of a negative value. (This does not mean that $r(-4A)$ is not permitted. A itself may well be negative, so that $-A$ is positive.)
- iii. Logarithms of zero or of negative values.
- iv. Other functions of arguments for which the function is not defined; for instance, $\tan 90^\circ$. Also, certain functions have principal ranges of arguments so that they may be mathematically classified as functions. For instance, the range of the argument for $\sin^{-1} x$ is $-1 \leq x \leq 1$; $\text{ASN } 2$ results in an error during execution.
- v. A legitimate computation which produces a result (either intermediate or final) whose absolute value is either greater than 9.9999999999E99, or less than E-99.

In each of these five cases, some number is associated with the attempted computation, and is treated as the result, even though this is mathematically incorrect. See the error summary in the System Reference for what that number is in each specific case.

*Unary plus can precede the argument of a function without using parentheses. Thus, $r+A$ is permitted, while $r-A$ or $r+-A$ are not.

4-12 NUMERICAL COMPUTATIONS

RULES OF MATHEMATICAL COMBINATION

12. A statement can consist of several expressions connected by assignment instructions. A statement of the form:

$\langle \text{value} \rangle \rightarrow \langle \text{math expr} \rangle \rightarrow \langle \text{math expr} \rangle \rightarrow \dots$

is permitted as long as the first item after each \rightarrow is a register name.

For example:

```
A+B→C+X→Y5→R1
```

is treated internally by the Model 20 as the series of statements:

```
A+B→C
C+X→Y
Y5→R1
```

After the statement has been completely executed, C will contain $A+B$, Y will contain $A+B+X$, and R1 will contain $5(A+B+X)$.

EXAMPLES

This section contains some sample expressions and statements that illustrate how the Model 20 language is used to formulate mathematical computations.

EXPRESSION OR STATEMENT	MATHEMATICAL MEANING
3.14159	The value of the number 3.14159.
1.86E5	The value of the number 186000.
6.23E-27	The value of the number 6.23×10^{-27} .
A	The value of the number which is currently stored in register A (a).
-B	The negative of the number which is stored in register B ($-b$).
3.14→A	Store the number 3.14 into the register A. Whatever may have been stored in there before is lost (make a equal 3.14).
A→B	Store the value of A into B. A remains unchanged (make b equal a).
B-10.6	The difference in the values of b and 10.6.
A+X	The sum of a and x .
X*Y	The product of x and y .
X/A	The quotient of x divided by a .
Y/-B	The quotient of y divided by the negative of b .
√A	The value of the square root of a .
(X-Y)/(A+B)	The quotient of $x - y$ divided by $a + b$.
X↑Y	x raised to the power of y , (x^y).
1/(ALN X)	The reciprocal of $a \ln x$.

EXAMPLES

Mathematical Notation	Correct Expression or Statement	Incorrect Expression or Statement
$-(a+b)$	$-(A+B)$ or $-A-B$	$-A+B$ or $-+A+B$
a^{x+2}	$A↑(X+2)$	$A↑X+2 = a^x + 2$
$(a^{x+2})c$	$A↑(X+2)*C$ or $A↑(X+2)C$ or $CA↑(X+2)$	$A↑X+2*C = a^x + 2c$
$(ca)^x$	$(CA)↑X$	$CA↑X = c(a^x)$
$\frac{ab}{xy}$	AB/XY or $(A*B)/(X*Y)$	$A*B/X*Y = \frac{aby}{x}$
$\frac{a}{x} \cdot \frac{b}{y}$	$A/X*B/Y$ or $(A/X)(B/Y)$ or AB/XY	$A/XB/Y = \frac{a}{xby}$
$a + \frac{x}{3+z}$	$X/(A+Y/(3+Z))$	$X/(A+Y/3+Z)$
$\sqrt{-a}$	$r(-A)$	$r-A$
$\sqrt{a^2 + b^2}$	$r(AA+BB)$ or $r(A↑2+B↑2)$	$rAA+BB = a\sqrt{a+b^2}$
e^{x+1}	$EXP(X+1)$	$EXP X+1 = e^x + 1$
Evaluate $a = b^2 + c^2$	$BB+CC→A$ or $B↑2+C↑2→A$	$A=BB+CC$ '=' is not used to establish equality. It is a relational operator used to <i>enquire</i> about equality.

EXERCISES

Answers are on page A-1.

1. Write the equivalent expressions or statements in the Model 20's language:

- a. $a(b + c)$
- b. $-\left(\frac{a}{2}\right)$
- c. $\frac{xy}{ab}$
- d. $\frac{\frac{x}{y}}{\frac{a}{b}}$
- e. $\sqrt{-(a - b)}$

- f. $\sqrt{(x - y)^2 + (a - b)^2}$
- g. $y = ax^2 + bx + c$
- h. let $a = \sqrt{256}$
(The remaining expressions may require the Math ROM.)
- i. $-a^b$
- j. $(a + b)^{\frac{1}{4}}$
- k. $e(a + b)$
- l. $\sin -30$
- m. $\sin x^2$
- n. $\sin^2 x$
- o. $\sin \sin x$

4-14 NUMERICAL COMPUTATIONS

EXERCISES

2. Locate the errors in each of the following expressions and statements.

- $r - A \rightarrow B$
- $(A / -B) / (X - / Y \rightarrow Z$
- $6((R7 + R(R2 + A) / (R3 / (R5 + B)))$
- $AB / CX \rightarrow R1 R1 + R2 \rightarrow 5Y \rightarrow Z$
- $1.342E$
- $(A + B)E3 + E9.9 \rightarrow X$
- $1,457.31 \rightarrow X$
- $2EA / B \rightarrow X$
- $10((A + B())X + Y) / 3.1 + C)$

3. In an attempt to produce this display:

3.45E5+X+A

you press:

3 . 4 5 $\frac{E}{\square}$ 5 + X \rightarrow A

A NOTE appears in the display. What is wrong?

4. R1 contains 1.0 and R5 contains 2.0. What is the numerical value of the expression:

$$R1 + R5 / 5$$

5. Let's say you wrote the line:

```
153896.003150782248-
153896.003150782951
```

What is the actual difference between the numbers? Is this the same number obtained upon executing the line? Is the actual difference within the range of the machine? Explain what happens when the line is executed.

6. You are debugging a program. You have performed a calculation and stored the result in register A. Upon executing the line:

```
FLT 9;DSP A
```

The display appears as

2.5000000000E 01

There is a line in your program involving the segment:

(expression) \rightarrow RA

After a while you discover that the answer is being placed in R24 instead of R25.

Now that your suspicions are aroused, you execute the line:

```
DSP A-25
```

The display appears as:

-3.0000000000E-10

Why is the value of the expression being stored in R24, and why is A displayed as 'exactly' 25?

7. Not all calculations done on computers or calculators turn out as expected, especially when iterative (repetitive) methods are involved. This example illustrates one type of thing that can go wrong.

We are going to calculate and print successive values of x_i in the sequence:

$$\begin{aligned} x_0 &= 1/9 \\ x_1 &= 10 \cdot 1/9 - 1 (=1/9) \\ &\vdots \\ x_i &= 10x_{i-1} - 1 (=1/9) \\ &\vdots \end{aligned}$$

Notice that each x_i in the sequence has a value of $1/9$ (1.1111111111E-01).

EXERCISES

We can do this easily by first executing the line:

```
FLT 9:1/9+X
```

Now write the line:

```
PRT 10X-1+X
```

Each time EXECUTE is pressed the next x_i is computed and printed.

Press execute at least 25 times.

Explain why the x_i aren't all equal to $1/9$. Are x_1 and x_2 really equal to $1.1111111111E-01$? (Remember, the printer prints to 10 digits, while calculations are performed to 12.)

You should see a printout like this:

```

x1 → 1.1111111111E-01
      1.1111111111E-01
      1.1111111110E-01
      1.1111111100E-01
      1.1111111000E-01
      1.1111110000E-01
      1.1111000000E-01
      1.1110000000E-01
      1.1100000000E-01
      1.1000000000E-01
      1.0000000000E-01
x12 → 0.0000000000E 00
      -1.0000000000E 00
      -1.1000000000E 01
      -1.1100000000E 02
      -1.1110000000E 03
      -1.1111000000E 04
      -1.1111100000E 05
      -1.1111110000E 06
      -1.1111111000E 07
      -1.1111111100E 08
      -1.1111111110E 09
      -1.1111111111E 10
      -1.1111111111E 11
x25 → -1.1111111111E 12
      -1.1111111111E 13
      -1.1111111111E 14
      -1.1111111111E 15
      -1.1111111111E 16
      -1.1111111111E 17
      -1.1111111111E 18
      -1.1111111111E 19
      -1.1111111111E 20
      -1.1111111111E 21
      -1.1111111111E 22
      -1.1111111111E 23
      -1.1111111111E 24
    
```

5-0

NOTES

Chapter 5

THE MODEL 20 LANGUAGE

This chapter is a formal presentation of the concepts and syntactical rules of the Model 20's algebraic language.

The first part of this chapter defines the terms used in explaining the syntax, which is presented in the second part of the chapter. The definitions are used in precise explanations of the various syntaxes.

A formal presentation, based on carefully chosen definitions, systematizes and generalizes the information — making it possible to learn only the definitions and some rules, rather than literally hundreds of individual possibilities.

The material in this chapter is presented in a special format. Each topic is presented in two parts; one part in the left-hand column, and the other part directly opposite in the right-hand column. The remarks in the right-hand column lead to an understanding of the material in the left-hand column, or clarify it, or refer the reader to another section of the manual for additional information. Each topic is terminated by a horizontal line.

Persons without programming experience in an algebraic programming language should read the left-hand column, and then the right-hand section, of each topic. The experienced programmer can probably read just the left-hand columns; they constitute the formal presentation.

DEFINITIONS

CONSTANTS AND VARIABLES

*A **constant** is a number that is expressed in an explicit manner, as part of a line.*

A **real constant** is a number that, in the mathematical sense, is both rational and non-complex.

Numeric constants are often referred to as fixed point constants, or as floating point constants. Both types, however, are internally represented in floating point fashion.

From time to time, different types of constants may come into existence as new plug-in ROM's add to the Model 20's language.

*A **variable** is a numeric or logical entity that is referred to by the name of the register, or other unit of memory, containing the entity, rather than by an explicit appearance of the entity itself.*

A **real variable** is one whose value could be that of any real constant.

In the non-programming sense, a constant is generally considered to be something whose value does not change for the duration of the problem. We use a more restrictive definition to distinguish between things that are stored in registers (variables) and things whose values are explicitly indicated by part of a line.

While it is true that, in the problem solving sense, a number in a register could be used as a 'constant', machine and programming considerations make the restrictive definition of a constant desirable.

In terms of programming, the significant difference between variables and constants is this: The value of a variable can be changed by the machine itself, by following the directions given in a line; the value of a constant cannot be changed except by modifying the line containing it.

5-2 THE MODEL 20 LANGUAGE

DEFINITIONS

QUANTITIES

'Quantity' is a general term used to denote any single constant or variable.



For further information, see "SPECIAL CONVENTIONS", page 4-7.

LITERALS

A **literal** is a sequence of characters beginning and ending with quotation marks.

```
"THIS IS A LITERAL!"
```

By themselves, literals cannot cause any useful activity. They are usually used in conjunction with other instructions, and as *labels*.

Literals are used in displaying and printing messages, and in *symbolic addressing*. The notion of symbolic addressing is discussed in "SYMBOLIC ADDRESSING", on page 5-9.

FLAGS

A **flag** is a variable whose value is either 'set', or 'clear', corresponding to the numbers one and zero, respectively.

A flag is not a *real* variable.

There are 16 flags, named flag 0 through flag 15.

Flags are designated by the mnemonic `FLG` followed by a value which determines the particular flag intended:

```
FLG 15    FLG A
```

There are instructions to set and clear the flags:

```
SFG 14    sets flag 14
CFG 6      clears flag 6
```

The mnemonic `CFG` is obtained by pressing the SET/CLEAR FLAG N key twice in succession.

In general, flags are not used to represent numbers; they are used to represent *conditions*.

These conditions can be determined by the user. When the condition occurs (perhaps it is detected by some test in the program) the desired flag is set (or possibly cleared). Then, at a later time, the flag can be interrogated (its value checked) and subsequent activity based on the result. In this way an event that occurs in one part of a program can affect events that occur later in the program.

Three of the flags are associated with definite meanings. If flag 14 is set, certain arithmetic faux pas, such as division by zero, will be ignored. Flag 15 is automatically set whenever such faux pas occur. Flag 13 is used in conjunction with the ENTER statement.

 DEFINITIONS

 OPERATORS

An **operator** is a symbol that indicates an activity to take place upon a single value (a **unary operator**), or, between two values (a **binary operator**).

The values involved in the activity are called **operands**.

The **arithmetic operators** are:

- ↑ Exponentiation (requires Math ROM)
- ⊞ Implied Multiply (no symbol)
- * Explicit Multiplication
- ÷ Division
- + Addition (also unary plus)
- Subtraction and Unary Minus (negation)

The **relational operators** are:

- = Equals
- ≠ Does not Equal
- > Greater Than
- ≤ Less than or Equal

The arithmetic operators were discussed in Chapter 4, beginning on page 4-4, and are used in the construction of *mathematical expressions*.

The relational operators are used in forming *relational expressions*, which are used in constructing tests of numerical relationships.

The arithmetic and relational operators are the familiar ones that are used in mathematics. As far as these operators are concerned, the essential differences between mathematical notation and the Model 20's language are these:

1. Raising a number to a power (available with the Math ROM):

$A \uparrow B$ corresponds to a^b .

2. The times sign for multiplication:

$A * B$ corresponds to $a \times b$ or $a \cdot b$.

3. The equal sign:

$A = B$ is used in inquiring about possible equality between A and B, and corresponds to the 'a = b' part of the question 'does a = b?'

To actually make A and B equal ('let a = b'), use $A \rightarrow B$, or $B \rightarrow A$, whichever is appropriate.

4. The relational operator < is not available. To ask is $a < b$, use $B > A$.

 FUNCTIONS

A **function** is a computational rule or process which operates upon a value or values to produce a single result.

The value operated upon is called the **argument** of the function.

Each individual function has a name.

Supplied functions are those functions whose names and computational rules are determined by the design of the Calculator, and cannot be altered.

Definable Functions (available with the User Definable Functions ROM) have names and computational rules that are determined by the programmer.

Functions are used in very much the same way as in conventional mathematical notation. Functions are discussed in Chapter 4, beginning on page 4-6.

DEFINITIONS

EXPRESSIONS

An **expression** is a valid sequence of variables, constants, functions, parentheses, and operators.

A **mathematical expression** is an expression containing no operators other than arithmetic operators:

ΓA $-5+4$ $R1+R2$
 $X\uparrow X$ AB $(A+B)/C$

NOTE

$R(A+B)$ is not considered an expression; it is simply a variable, even though $(A+B)$, if considered by itself, is an expression.

A **relational expression** consists of two (or more) values, separated by one (or more) relational operators:

$FLG\ 5\neq 0$ $X+5>RRA$
 $(X+Y=Z)+(A+B\leq C)>R0$

The word 'expression' will be used to denote any of the types of expressions that can be written in the Model 20's language. Particular types of expressions will be denoted as 'mathematical expression', 'relational expression', etc.

Every mathematical or relational expression has a numeric value.

The numeric value of a relational expression is: one if the relationship is true, and zero if it is false.

The general form of a simple relational expression is:

$\langle \text{value} \rangle \langle \text{relational operator} \rangle \langle \text{value} \rangle$

The numeric value of each $\langle \text{value} \rangle$ is found, internally held, and rounded to 10 places. These rounded values are used in the evaluation of the expression. The original $\langle \text{value} \rangle$'s remain unchanged.

In casual terms, any collection of quantities that have been combined with one another is an expression.

An isolated unsigned constant or variable is *not* considered an expression.

The condition of validity, referred to in the formal definition, is met if the expression conforms to some syntax or rule involving the component elements of the expression.

The rules of formulating mathematical expressions are the subject of Chapter 4. The latter part of this topic is a discussion of relational expressions.

The numeric value of the following mathematical expression is five:

$(\Gamma 25+15)/4$

A relational expression is either true or false, and has a numeric value of one or zero, respectively.

$A=A$ is certainly true, and has a numeric value of one.

$A=A+1$ is certainly false, and has a numeric value of zero.

$A=B$ can be true or false, depending upon the numeric values of A and B.

DEFINITIONS

Relational expressions can be combined to form *complex relational expressions*.

In the absence of parentheses, relational operators are taken from left to right, and the component relationships replaced with their equivalent numeric values.

Thus, the expressions:

$$A=B=C \text{ and } (A=B)=C$$

are each equivalent to:

$$1=C \text{ or } 0=C$$

depending solely upon equality between A and B. The relationships between A and C, and between B and C, are not involved.

Another form of complex relational expression is illustrated by:

$$(A=B) \neq (X>Y)$$

To evaluate this form, replace each component relational expression with its value, and evaluate the simple relational expression.

Relational expressions can be combined with mathematical operations, or, used as 'quantities' in mathematical expressions. Such expressions are called **mixed expressions**.

These types of expressions simply use the values of the relational expressions as ordinary numeric quantities. Consider:

$$(A=B) (X=Y) \rightarrow A$$

Here, the product of the values of the component relational expressions represents the 'logical and' of the conditions 'A equals B' and 'X equals Y'.

The only limitation on the size or degree of sophistication of a mixed expression is the length of the line containing it.

The expression:

$$A=B=C$$

has an overall value of one (true) if:

- a. A equals B, and C equals one.
- b. A does not equal B, and C equals zero.

On the other hand, it has an overall value of zero (false) if:

- c. A equals B, but C does not equal one.
- d. A does not equal B, but C does not equal zero.

A similar analysis of the expression:

$$(A=B) \neq (X>Y)$$

shows that its value is one if:

- a. A does not equal B and X is greater than Y.
- b. A equals B and X is not greater than Y.

The overall value is zero if:

- c. A equals B and X is greater than Y.
- d. A does not equal B and X is not greater than Y.

Mixed expressions find occasional use in just about every type of computational activity. It takes a little while to become proficient in their use, but they can be excellent space savers in a program. See the exercises at the end of the chapter for examples.

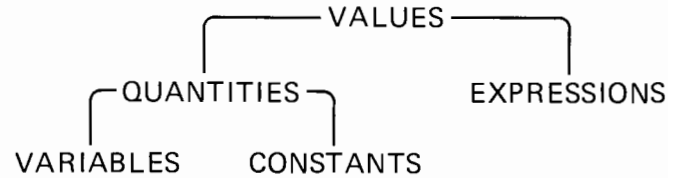
5-6 THE MODEL 20 LANGUAGE

DEFINITIONS

VALUES

'Value' is a general term used to denote any single quantity or expression.

The word 'value' is used two ways: with the exact meaning given by the definition, and with the meaning 'numerical amount' in the arithmetic sense. In places where 'value' seems ambiguous, either meaning can generally be used with success. Most times the context makes the intended meaning clear.



INSTRUCTIONS

'Instruction' is a general term used to refer to symbols and mnemonics that specify some particular activity, but are neither functions nor operators. An instruction is simply a command to do something.

Some examples of instructions are:

```
STOP STP
PRINT PRT
DISPLAY DSP
GAZINTA* →
```

*'Gazinta' is a corruption of the words "goes into".

STATEMENTS

A statement is a valid instruction, or a valid combination of instructions and values.

```
STP TRC END
PRT "A=",A,"B=",B
10r(X/1.3+Y↑1.6)→R5
```

A literal is *not* a statement.

The statement is the smallest complete syntactical unit which can completely specify an activity*.

Each statement contains at least one instruction; 2+5 is a statement because of the →Z added by the implied store into Z.

The condition of validity is met if the statement under consideration conforms to an established syntax.

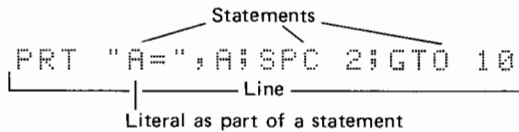
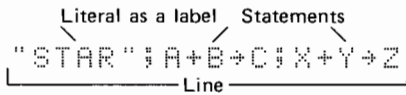
*Literals can also be 'complete syntactical units', but do not, by themselves, specify an activity.

DEFINITIONS

LINES

A **line** is a sequence of one or more statements and literals, separated by semicolons.

There is a great deal of pertinent information about lines. See "A BRIEF LOOK AT LINES AND STATEMENTS", page 2-6, and most sections of Chapter 3.



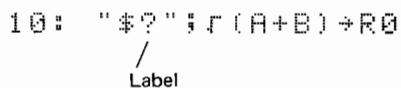
Note that the definition of a line has been broadened, from the definition in Chapter 2, to allow literals to be used in the line in the same way as statements are used. This is done to account for labels (the subject of the next topic).

LABELS

A **label** is a literal used as the very first segment of a line, and separated by a semicolon from the statements in the line.

The importance of labels and symbolic addressing is that, while the line number of a line might change — due to editing — a label remains unchanged.

Labels provide a method of identifying stored lines without using their line numbers. This is the subject of "SYMBOLIC ADDRESSING", page 5-9.



If the above line were stored in the Calculator, then the following two statements would be equivalent:

```

GTO 10
GTO "&#36?"
  
```

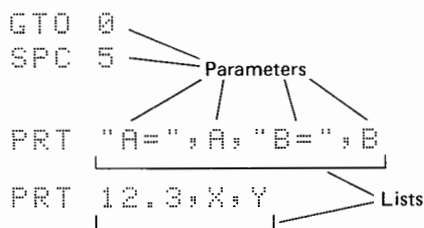
DEFINITIONS

PARAMETERS AND LISTS

A **parameter** is a value or literal associated with an instruction or suitable function.

Certain instructions use parameters to determine the specific activity to be performed.

A **list** is a sequence of one or more parameters separated by commas.



Some instructions never have parameters or lists associated with them. Others always have at least one parameter, while some may or may not require parameters, depending upon the usage.

Some instructions require that their parameters be of particular types, that is, their parameters could be variables, but not constants, or, variables or constants, but not expressions.

PROGRAMS

A **program** is a sequence of consecutive lines that, together, will perform some useful activity once they are stored in the Calculator.

As each line is stored it is automatically assigned a **line number**. Line numbers are unsigned, consecutive integers, beginning with 0.

A program is a recipe for doing something.

Chapter 3 contains a great deal of pertinent information about the mechanics of programming the Model 20.

The "SIMPLIFIED OPERATING INSTRUCTIONS" booklet provides some introductory remarks about programming, intended for the non- or novice programmer.

BRANCHING

Branching is the general term used to denote a program's ability to alter the natural order in which the lines of the program are executed.

The Model 20's language permits both *conditional* and *unconditional* branching.

Although a command to branch can be located anywhere in a line, the branch itself is always made to the beginning of a line; there is no way to branch into the middle of a line.

Branching instructions can be thought of as a means to control the program line counter.

Addressing is a term that is frequently used in indicating some specific form of branching.

The natural order in which the lines of a program are executed is simply one after another, in the sequence in which they are written; that is, in the order of their line numbers.

The natural order is altered by a line specifying some line, other than its natural successor, to be the next line executed.

An unconditional branch is a command which specifies a branch which is to be accomplished without any further consideration.

A conditional branch is a command to branch provided that some condition is met, otherwise the command to branch is ignored.

DEFINITIONS

RELATIVE ADDRESSING

Relative addressing is a means of branching to a line indicated as being so many lines before or after the line containing the branching instruction.

Relative addressing is accomplished with RELATIVE GO TO and JUMP statements. JUMP statements allow the parameter specifying the direction, and number of lines of the branch, to be computed at the time the branch is made.

The significance of relative addressing is that it is independent of the actual line numbers involved. All that is required is that the lines of the program be in their original order; the program itself might be part of a stack of programs.

SYMBOLIC ADDRESSING

Symbolic addressing is a means of branching to a line that is identified with a label of the programmer's choice.

Symbolic addressing is valuable because it is totally independent of the line numbers, and of the sequence, of the lines which constitute the program.

A time saving technique which is often used in writing and debugging programs is to use symbolic addressing in as many of the branching statements as possible. Then, as lines are inserted and deleted to bring the program to its proper form, the addressing stays valid even though the line numbers and original sequence are altered. Finally, once the program is running, the symbolic addressing can be replaced, if desired, with absolute addressing by line numbers.

RECORDS

A **record** is a collection of one or more sides of magnetic card(s) upon which programming or data has been recorded.

A record cannot contain both programming and data at the same time.

DEFINITIONS

SUBROUTINES

A subroutine is a program, or program segment, which performs some specific task, and is used as a component in another program.

The subroutines that can be written with the basic Calculator are called **local subroutines** to distinguish them from another type of subroutine available with the User Definable Functions ROM.

Local subroutines are stored in the mainline programming area.

Local subroutines are initiated with GO TO SUB statements, and are terminated with RETURN statements.

Subroutines are useful as 'building blocks' when writing a program, and are used in two general ways:

- a. Certain often needed computational procedures can be programmed ahead of time as subroutines, allowing the programmer to combine them as needed in each specific problem.
- b. A certain problem may require the same procedure many times in different parts of the program. Considerable memory can be saved by writing a subroutine, and branching to it as needed.

So far, nothing has been described that would be impossible using only ordinary branching. With ordinary branching, however, there is no easy way for the 'subroutine' to determine where to branch back to.

The GO TO SUB and RETURN statements allow the Calculator itself to know where to branch back to, and to do so automatically.

The usual method of communication between a program and a local subroutine is to have the main program place certain values in registers before branching to the subroutine. The subroutine 'finds' its input data in those registers, and before branching back to the main program, places the result(s) in some register(s). In the same way, the main program 'finds' the result(s), and proceeds with its task.

THE SYNTAX

This part of Chapter 5 presents the specific syntaxes permitted by the Model 20 without plug-in ROM's. The main vehicles for conveying this information are what are called *literal* and *figurative* syntax forms. These are supplemented with written explanation and examples.

A literal syntax is an exact and technical description of which types of language elements may occupy various positions in a given statement. A literal syntax allows you to decide if a statement you have written is syntactically permissible. Sometimes several literal syntaxes will be required to describe all the variations in form that a particular type of statement may have.

A figurative syntax is less technical, and is a description of what the various language elements in a statement represent, in terms of the activity caused by the statement. A figurative syntax helps you to decide if a permissible statement you have written is actually going to do what you had in mind when you wrote it, or do something else. A calculator or computer does what it was actually instructed to do, not what someone *thought* they told it to do.

In showing the syntactical forms that a statement may have, the names of the various language elements are enclosed within the symbols (< . . . >), which are used to indicate that their content is only the name of the thing that is to be in that position, rather than the thing itself.

It often happens that optional language elements are associated with a particular syntax. Such cases will be indicated by placing the optional item within the symbols [. . .].

The examples given for a particular syntax will not necessarily illustrate every possible variation.

A special convention is used in the sample exercises. Except when shown as part of a listing of a program, the end of line symbol † indicates that EXECUTE should be the next key pressed. For example:

4+5† means press 4 + 5 EXECUTE

Program listings are easily identified by their shaded backgrounds.

FIXED AND FLOAT STATEMENTS

Literal Syntax:

FXD [<value>]
FLT [<value>]

Figurative Syntax:

FXD n
FLT n

Sample Exercise:

FXD 0;10.9090909090+X†

11

Note that a specification of fixed 0 does not show a decimal point.

FXD 1†

10.9

Where $0 \leq n \leq 9$ is the number of desired places to the right of the decimal point.

(continued)

THE SYNTAX

FIXED AND FLOAT STATEMENTS

(continued)

FIXED statements specify fixed point displays and printouts, while FLOAT statements specify floating point.

If (value) is not strictly an integer, its digits to the right of its decimal point are ignored.

NOTE 02 will result if the integer part of (value) is less than zero, or greater than nine.

Numbers displayed or printed under a fixed or float specification are always correctly rounded to reflect the value of the unseen digits. However, FIXED and FLOAT statements never affect the appearance of a displayed or printed literal.

2→A;FXD A←

10.91

FXD 3.3←

10.909

FLT A+5;DSP X←

1.0909091E 01

Note that the right-most digit of the number is 1 rather than 0; this is due to rounding.

FLT -1← or FXD 11←

NOTE 02

FXD 0;DSP "3.0001"←

3.0001

While the machine is operating under a fixed point specification, it 'remembers' the previous floating point specification, and vice versa.

The instructions FXD and FLT, when used without a parameter, specify their respective forms; the previous parameter for that form is automatically assumed.

While the machine is running a program, it can be placed under a fixed point or floating point specification, without stopping the program, simply by pressing FIXED N or FLOAT N, respectively. In each case the previous parameter for that particular form is automatically assumed. Of course, this action is subject to FIXED and FLOAT statements within the program itself.

When the machine is turned on, or initialized with MEMORY ERASE, the specification FLT 9, with a previous specification of FXD 0, is established.

Sample Exercise:

FXD 0;FLT 3;1234567←X←

1.235E 06

FXD ←

1234567

FLT ←

1.235E 06

◆◆◆◆◆ THE SYNTAX ◆◆◆◆◆

A number that is too large to be displayed or printed under the current fixed point specification will appear in a form determined by the previous floating point specification. (The current specification remains as fixed point, however.)

There is no reversion to floating point when a number is small enough to appear as ±.0000...

Executing a FIXED or FLOAT statement will not interfere with the display of an answer already visible in the display.

Sample Exercise:

```
FLT 5:FXD 3:12345678+XF
```

1.23457E 07

```
FXD 2F
```

12345678.00

In fixed point, the sum of the number of digits to the left and to the right of the decimal point must not exceed 10, or the number will be shown in floating point.

————— DISPLAY STATEMENTS —————

Literal Syntax:

```
DSP [(list)]
```

Figurative Syntax:

```
DSP parameter1, parameter2, ...
```

The DISPLAY statement causes each item in the parameter list to be shown in the display. The parameters are displayed individually, and in the left to right order of the parameter list. Each parameter is visible for approximately .175 seconds.

If a DISPLAY statement is executed from the keyboard, the last parameter in the list will remain visible until some other activity is performed.

Parameters other than literals are displayed: on the right side of the display; as their respective numeric values; and according to the current fixed or floating point specification.

Literals are displayed as the text of the literal, and on the left side of the display. If the literal is longer than 16 characters, only the left-most 16 will be displayed.

Sample Exercise:

```
FXD 2:1+A:2+B+  
DSP A,B+
```

1.00



2.00

```
DSP "A=",A+
```

A=



1.00

```
FLT 9:FXD 8:DSP 99+
```

99.00000000

```
DSP 100+
```

1.0000000000E 02

```
DSP "100"+
```

100

```
DSP "ABCDEFGHIJK  
LMNOPQRSTUVWXYZ"+
```

ABCDEFGHIJKLMN

5-14 THE MODEL 20 LANGUAGE

THE SYNTAX

DISPLAY STATEMENTS

(continued)

The syntax:

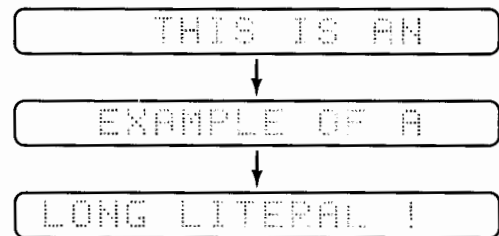
```
DSP (list) ; DSP ; DSP ; ...
```

displays all the parameters in the list, and, causes the last parameter in the list to be displayed one additional time for each extra DSP.

The first DISPLAY statement in a line should be followed by a list; using DSP without a list is not recommended except as shown in the above syntax. It is also recommended that the sequence of DISPLAY statements be a continuous one, without any other type of statements in the middle of the sequence.

Sample Exercise:

```
DSP "bbbbTHISbISbAN";  
DSP ; DSP ; DSP ; DSP "bb  
EXAMPLEbOFbA";  
DSP ; DSP ; DSP ; DSP "LONG  
bLITERALb!"
```



PRINT AND SPACE STATEMENTS

Literal Syntax:

```
PRT (list)
```

Figurative Syntax:

```
PRT parameter1, parameter2, ...
```

The PRINT statement causes each item in the parameter list to be printed by the printer. Each parameter is printed on a row by itself; the order in which the parameters are printed is the left to right order of the parameter list.

If a PRINT statement is executed from the keyboard, the last parameter in the list will remain visible in the display until some other activity is performed.

Parameters other than literals are printed: on the right side of the paper; as their respective numerical values; and according to the current fixed or floating point specification.

Literals are printed as the text of the literal, and on the left side of the paper. If the literal is longer than 16 characters, only the left-most 16 will be printed.

Sample Exercise:

```
FLT 5; FXD 2+  
10+A; 20+B; 2.6E10+C+  
PRT "A="; A; "B="; B; "C="; C+
```

```
A= 10.00  
B= 20.00  
C= 2.60000E 10
```

```
PRT "A+B="; A+B+
```

```
A+B= 30.00
```

◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆ THE SYNTAX ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆

The syntax:

```
PRT <list>;PRT ;PRT ;...
```

behaves in a manner analogous to the DISPLAY statement of the same form. See the previous section.

Literal Syntax:

```
SPC [<value>]
```

Figurative Syntax:

```
SPC n
```

Where $0 \leq n \leq 15$ is the number of rows the printer paper is to be advanced.

SPACE statements are used to advance the printer paper, under the control of the Calculator.

SPACE statements are written with the SPACE N key, and should not be confused with statements involving SPACE (a key in the left-most keyblock) or with PAPER, (a mechanical method of advancing the paper).

If <value> is not strictly an integer, the digits to the right of the decimal point are ignored.

NOTE 02 will result if the integer part of <value> is less than 0, or greater than 15.

SPC by itself is equivalent to SPC 1, but is a keystroke shorter.

A SPACE statement generally clears the display when it is executed.

Sample Exercise:

```
PRT "*" ; SPC 10 ; PRT "*" F
```

*

*

```
2 → X ; 4 → Y ; PRT "*" ;
```

```
SPC X + Y ; PRT "*" F
```

*

*

```
SPC X - Y F or SPC Y Y F
```

NOTE 02

```
PRT "*" ; SPC ; PRT "*" F
```

*

*

SPC 0 does not advance the paper.

THE SYNTAX

ASSIGNMENT STATEMENTS

Literal Syntax:

$\langle \text{value} \rangle \div \langle \text{real variable} \rangle$

Figurative Syntax:

Assign the numeric value to the variable.

This is the basic statement used to put numbers into registers.

NOTE 06 results if $\langle \text{real variable} \rangle$ is a 'non-existent' R register, or if the item immediately following \div is anything other than a register name.

NOTE 05 results if $\langle \text{value} \rangle$ contains a 'non-existent' R register.

Sample Exercise:

FXD 2:100+AF

100.00

Press CLEAR.

DSP AF

100.00

100+R10000F

NOTE 06

R4000+AF

NOTE 05

Literal Syntax:

$\langle \text{value} \rangle \div \langle \text{real variable} \rangle \div \langle \text{real variable} \rangle \div \dots$

Figurative Syntax:

Assign the numeric value to each variable.

Whenever an assignment statement is executed from the keyboard, the last numeric value assigned will be displayed until some other activity is performed.

Sample Exercise:

FXD 2:25+A+B+CF

25.00

PRT A,B,CF

25.00

25.00

25.00

Literal Syntax:

$\langle \text{value} \rangle \div \langle \langle \text{real variable} \rangle \dots \rangle \div \langle \langle \text{real variable} \rangle \dots \rangle$

Figurative Syntax:

value \div real variable₁
real variable₁ ... \div real variable₂ ...

The statement:

R1R2+R3R4+X

is equivalent to the series of statements:

R1R2+R3

R3R4+X

If the statement were stored while the program line counter was at two, the display would be:

2: (R1R2+R3)R4+XF

This form is called the multiple assignment statement, and is usually used to save some intermediate results of an involved calculation.

◆◆◆◆◆ THE SYNTAX ◆◆◆◆◆

When a statement of this form is stored, parentheses are added around each of the 'interior assignments'.

A parameter in a statement can usually be replaced with an assignment statement involving the parameter. When the overall statement is executed, the assignment statement is performed at the time the parameter it represents is needed; the value that was assigned is taken as the parameter.

Sample Exercise:

```
FXD 2:0+A+B
PRT A+1+A,B+10+B:SPC 1+
Press EXECUTE several times.
```

```

1.00
10.00
2.00
20.00
3.00
30.00
4.00
40.00
5.00
50.00
```

A line of the form:

```
[... ] <value> [ # ... ]
```

is stored as:

```
[... ] <value> +Z [ # ... ]
```

The addition of +Z is called the 'implied store into Z'.

A statement of the form:

```
<value> + <<real variable> ... > + <<real variable> ... >
```

may or may not have +Z added when the statement is stored. The +Z will be added if the statement was originally written with at least the outer-most level of the enclosing parentheses for the 'interior assignments'.

The implied store into Z never affects the elements of a parameter list.

Sample Exercise:

```
FXD 2:ENDF 0+ PRT Z+
1.00
25+ PRT Z+
25.00
100+A+ A+ PRT Z+
100.00
```

write the line:

```
10+B:A:25+C
```

then press STORE.

```
0:10+B:A+Z:25+C
```


THE SYNTAX

ENTER STATEMENTS

Literal Syntax:

ENT [(literal) ;] (real variable)
 [[; (literal) ; (real variable) ····]

Figurative Syntax:

$n_1 \rightarrow$ real variable₁ ; $n_2 \rightarrow$ real variable₂ ; ····

Where the n_i are supplied by the Operator at the time the ENTER statement is executed.

ENTER statements are a means to suspend the execution of a program for the purpose of assigning values to a list of real variables. When the ENTER statement is encountered, the Calculator will 'ask' for the value to be assigned to the first real variable in the list. The value is supplied to the Calculator by keying in a constant, or an expression, representing the value. Then RUN PROGRAM is pressed to cause the assignment to take place. If there are more variables in the list, the Calculator will ask for the next value, otherwise the execution of the program is resumed.

NOTE

ENTER statements cannot be executed from the keyboard. They can be used only in the context of a program.

NOTE 11 will result if a line containing an ENTER statement is executed from the keyboard.

NOTE

If a line contains both an ENTER statement and a GO TO or GO TO SUB statement (see pages 5-22 through 5-27), the ENTER statement **must** precede those statements. If this precaution is not observed, the GO TO or GO TO SUB statement will be ignored.

Sample Exercise:

Press MEMORY ERASE, or, END EXECUTE STORE EXECUTE. Then store the following program:

```
0:
FXD 2F
1:
ENT A,B,C
2:
PRT "A=",A;"B=",
B;"C=";C;SPC 8F
3:
GTO 1F
4:
END F
```

PRESS:

PRESS:

PRESS:

PRESS:

```
A=
10.00
B=
20.00
C=
30.00
```

⋮

THE SYNTAX

When an ENTER statement is encountered, the display will show (on the left) the name of the first or next variable in the list, or, the text of the literal that is immediately to the left of that variable. Only the left-most 16 characters of a long literal can be displayed.

While the Calculator is waiting for RUN PROGRAM to be pressed, you may make such calculations as may be necessary to find the value to be entered. Also, the value that is to be assigned may take any form that is permitted as a value on the left side of an ASSIGNMENT statement.

Please note however, that if an expression is used to specify the value that a variable in the list is to assume, the expression itself must be visible in the display when RUN PROGRAM is pressed. If it has been executed to learn what its value is, the value seen on the right of the display will be ignored, and treated as 'no entry' (described shortly). To cause the Calculator to accept the calculated value, restore the expression to the display by pressing BACK. Then press RUN PROGRAM.

During the execution of an ENTER statement, but before it is satisfied, the Calculator is in the *Enter Mode*. During this mode, keys that would alter the program line counter are treated as syntax errors, or are ignored.

To abort an ENTER statement, and remove the Calculator from the Enter Mode, use the following procedure:

1. If necessary, press CLEAR to rid the display of any NOTE's.
2. Press STOP until STP appears in the display.
3. Then press CLEAR.

ENTER statements cannot be nested; that is, you may not do anything that requires another ENTER statement when the Calculator is already in the Enter Mode.

ENTER statements are not completely syntax checked. For that reason you must be careful to write meaningful ENTER statements, as illogical statements can be stored. However, they will result in errors during execution, and halt the program.

Sample Exercise:

Initialize the program memory. Then store and run the following program:

```
0:
FXD 2+
1:
ENT "WHICH R REG
ISTER";R1;"WHAT
VALUE";RR1+
2:
PRT RR1;"IS STOR
ED IN REG";R1+
SPC 8+
3:
GTO 0+
4:
END +
```

Sample Exercise:

Initialize the program memory. Then store and start the following program.

```
0:
ENT A:PRT A+
1:
SPC 1;END +
```

A

GTO ... + or END +

NOTE 01

The Calculator is in the Enter Mode. To abort the Enter Mode, press CLEAR STOP STOP CLEAR.

THE SYNTAX

ENTER STATEMENTS

(continued)

You may not use an ENTER statement:

- a. As part of a line that is to be executed *from the keyboard*.
- b. As part of a local subroutine that is to be initiated with a *GSE from the keyboard*.

And if the User Definable Functions ROM is installed:

- c. As part of a subprogram that is to be initiated with **F ★ EXECUTE**.
- d. As part of an Immediate Execute subprogram that is to be used while the Calculator is already in the Enter Mode.

Each of these situations will result in **NOTE 11**. If it should happen, however, abort the ENTER statement using the procedure given for that purpose.

The parameter that indicates a real variable, to be assigned a value, can be an expression. This is permitted as long as the left-most item of the expression is that variable. The expression can also contain its own ASSIGNMENT statements.

An ENTER statement can be thought of as a series of ASSIGNMENT statements. For this reason, the variables in the list must be of the type that could immediately follow a \rightarrow .

NOTE

If **NOTE 11** occurs as case c or case d, a special procedure is needed to abort the Enter Mode. See the Operating Manual for the User Definable Functions ROM.

Sample Exercise:

Enter and run the following program:

```

0:
FXD 0F
1:
ENT AA+B,XX+YF
2:
PRT A,B,X,YF
3:
GTO 0F
4:
END F

```

PRESS:

AA→B

PRESS:

XX→Y

PRESS:

THE SYNTAX

```

10
100
20
400
    
```

In this example, A is assigned the value 10 by the action of the ENTER statement. When the value of A is specified, AA→B is accomplished prior to assigning a value to X.

While in the Enter Mode, no value is assigned to a variable if:

- a. RUN PROGRAM is pressed prior to specifying a value, or,
- b. A value is specified, but EXECUTE or CLEAR is pressed, followed by RUN PROGRAM.

In each case, the variable in question retains its original value, and Flag 13 is set.

Flag 13 is a useful indicator that the last datum has been entered. Consider the following program for summing a series of numbers:

```

01
FXD 210+BF
1:
ENT "NEXT NUMBER
",BF
2:
IF FLG 13=01PRT
ATA+B→B;GTO 1F
3:
PRT "TOTAL",B;
SPC 8;END F
    
```

Line 2 is a conditional branch based on flag 13. Flag 13 is normally zero, allowing the new value of A to be printed and added to the previous total. After the last number has been entered, however, RUN PROGRAM is pressed without making an entry. This sets flag 13, which allows the IF statement of line 2 to branch to line 3. Then the result is printed and the program terminated.

5-22 THE MODEL 20 LANGUAGE

THE SYNTAX

ABSOLUTE GO TO STATEMENTS

Literal Syntax:

GTO (unsigned integer constant)

Figurative Syntax:

GTO n

Where n is a line number.

The ABSOLUTE GO TO statement is used to branch from a line to the beginning of the line whose line number is n.

NOTE 01 will result immediately from an attempt to make n a variable, or, if n has a decimal point.

Sample Exercise:

Store and run the following program:

```
0:
PRT "0";GTO 4+
1:
PRT "1";GTO 3+
2:
PRT "2";GTO 1+
3:
PRT "3";GTO 0+
4:
PRT "4";GTO 2+
5:
END +
```

Result:

```
0
4
2
1
3
0
4
2
1
3
```

RELATIVE GO TO STATEMENTS

Literal Syntax:

GTO + (integer constant)

GTO - (integer constant)

Figurative Syntax:

GTO nth higher line number

GTO nth lower line number

The RELATIVE GO TO statement is used to branch from a line, whose line number is, say, m, to the beginning of the line whose line number is m+n, or m-n, depending upon the sign preceding (integer constant).

Sample Exercise:

The following program is equivalent to the one shown in ABSOLUTE GO TO statements.

```
0:
PRT "0";GTO +4+
1:
PRT "1";GTO +2+
2:
PRT "2";GTO -1+
3:
PRT "3";GTO -3+
4:
PRT "4";GTO -2+
5:
END +
```

◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆ THE SYNTAX ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆

NOTE 01 will result immediately from an attempt to make \langle integer constant \rangle have a decimal point, or, be a variable instead.

GTO +0 and GTO -0 each cause a branch back to the beginning of the line which contains them.

Result:

0
4
2
1
3
0
4
2
1
3
0

————— LABELED GO TO STATEMENTS —————

Literal Syntax:

GTO \langle literal \rangle

Figurative Syntax:

GTO the line labeled with the same literal.

The LABELED GO TO statement is used to branch from a line to the beginning of the line having a label matching \langle literal \rangle .

The branch occurs in the form of a search beginning at line 0. The first line encountered having a label matching \langle literal \rangle is taken as the destination of the branch. Any other lines having the same label would never be reached by a LABELED GO TO statement.

Neither \langle literal \rangle nor the associated label are subject to any special restrictions on their length. However, if either is longer than four characters, only the right-most four characters are considered in the search.

Sample Exercise:

Load and run the following program:

```
0:
PRT "0";GTO "+";F
1:
"YYCC";PRT "1";
GTO 0F
2:
"XXBB";PRT "2";
GTO "YYYCC" F
3:
"XXAA";PRT "3";
GTO "XXXBB" F
4:
"AA";PRT "4";
GTO "XXAA" F
5:
"A";PRT "5";GTO
"AA" F
6:
"+";PRT "6";GTO
"A" F
```

Result:

0
6
5
4
3
2
1
0
6
5
4

THE SYNTAX

GO TO SUB AND RETURN STATEMENTS

Literal Syntax:

- GSSB <unsigned integer constant>
- GSSB + <integer constant>
- GSSB - <integer constant>
- GSSB <literal>

Figurative Syntax:

- GSSB at the line numbered n
- GSSB at the nth higher line number
- GSSB at the nth lower line number
- GSSB at the line labeled with the same literal

GO TO SUB statements are used to branch to local subroutines. The line at which the subroutine starts is identified by any of the methods used with GO TO statements. Thus, there is an ABSOLUTE GO TO SUB statement, a RELATIVE GO TO SUB statement, and a LABELED GO TO SUB statement. Insofar as each of these statements causes a branch from one line to another, each behaves in the same way as its corresponding GO TO statement.

However, one additional thing happens when a branch is made with GSSB from, say, line number n. The Calculator automatically assumes that, when the subroutine is completed, the execution of the main program is to resume at the beginning of the line numbered n+1. There is no way to alter this automatic assumption.

Literal Syntax:

```
<n:> [ . . . . . ] RET F
```

↙ end of line

Figurative Syntax:

Return from subroutine to previous program.

RETURN statements are used to terminate subroutines and branch back to the previous program. The destination of the branch will always be the line following the one which branched to the subroutine.

RET must never occur except as the last statement in a line.

Several lines in the subroutine may contain RETURN statements.

Figure 5-1 is a program that illustrates some typical uses of local subroutines. In all, there are 23 lines of programming.

Lines 0 through 13 are the main program. It branches to local subroutines labeled "ORIG", "PRT", "SORT", and "NEW". Although the program uses mostly symbolic addressing, any of the other types could also have been used.

The main program does the following things:

- a. Enters 3 numbers into R1-3.
- b. Prints those numbers in their original order.
- c. Re-arranges the numbers into numerical order.
- d. Prints the new order.
- e. Enters 3 numbers into R4-6.
- f. Prints those numbers in their original order.
- g. Re-arranges those numbers into numerical order.
- h. Prints the new order.
- i. Starts over.

The subroutines do the following things:

- a. "ORIG" prints a heading.
- b. "PRT" prints R1-3, or R4-6, depending upon the value of X, which is controlled by the main program.
- c. "SORT" re-arranges the contents of R1-3, or R4-6, depending upon the value of X. "SORT" uses A, B, and C as temporary storage.

"SORT" uses an unusual series of mixed expressions to re-arrange the numbers, as we have not yet covered the IF statement. Consider line 16. Suppose A is the smallest number. Then line 16 is equivalent to:

$$A * 1 + B * 0 + C * 0 + R X$$

"SORT" has one failing, in addition to being cumbersome; what happens if some or all of the input data are equal?

- d. "NEW" prints a heading.

THE SYNTAX

Main program	Subroutine "SORT"	Results
0: FLT 5;FXD 2F	15: "SORT";RX→A;R(X+1)→B;R(X+2)→C	ORIGINAL ORDER: 45.00 96.00 -10.10
1: ENT "N1";R1;"N2";R2;"N3";R3F	16: A(B>A)(C>A)+B(A>B)(C>B)+C(A>C)(B>C)→RXF	
2: GSB "ORIG" F	17: A(A>B)(A>C)+B(B>A)(B>C)+C(C>A)(C>B)→R(X+2)F	NEW ORDER: -10.10 45.00 96.00
3: 1+X;GSB "PRT" F	18: A(A>RX)(R(X+2)>A)+B(B>RX)(R(X+2)>B)+C(C>RX)(R(X+2)>C)→R(X+1)F	
4: GSB "SORT" F	19: RET F	ORIGINAL ORDER: 22.00 56.00 36.00
5: GSB "NEW" F		
6: GSB "PRT" F		
7: ENT "N4";R4;"N5";R5;"N6";R6F		
8: GSB "ORIG" F		
9: 4+X;GSB "PRT" F		
10: GSB "SORT" F		
11: GSB "NEW" F		
12: GSB "PRT" F		
13: SPC 4;GTO 1F		
	Subroutine "NEW"	
	20: "NEW";PRT "NEW ORDER:";RET F	NEW ORDER: 22.00 36.00 56.00
	Subroutine "PRT"	
	21: "PRT";PRT RX,R(X+1),R(X+2);SPC 4;RET F	
	Subroutine "ORIG"	
14: "ORIG";PRT "ORIGINAL ORDER:"; RET F		
	Last line of programming	
	22: END F	

Figure 5-1. A Program Illustrating the use of Local Subroutines.

THE SYNTAX

GO TO SUB AND RETURN STATEMENTS

(continued)

In the basic Calculator, local subroutines can be nested to a depth not to exceed 31 levels. In each specific problem solving situation, the actual depth available is determined by internal conditions at the time the nest is generated; as such, it is not easily predicted. NOTE 09 will occur as an error during execution if there is an attempt to nest too deeply.

If the User Definable Functions ROM is installed, local subroutines may be nested approximately twice as deep as would otherwise be possible.

NOTE 07 will occur if a RETURN statement is encountered, and the machine is not executing a local subroutine or a subprogram.

Subroutines are said to be nested when a subroutine branches to (and is returned to from) yet another subroutine before returning to the main program.

The following program and printed result illustrates the notion of nesting:

```

0:
PRT "IN MAIN PRO
GRAM"␣
1:
GSB "SUB1"␣
2:
PRT "IN MAIN PRO
GRAM"; "AGAIN"␣
3:
STP ␣
4:
"SUB1";PRT "ONE
LEVEL DEEP"␣
5:
GSB "SUB2"␣
6:
RET ␣
7:
"SUB2";PRT "TWO
LEVELS DEEP"␣
8:
RET ␣
9:
END ␣

```

Result:

```

IN MAIN PROGRAM
ONE LEVEL DEEP
TWO LEVELS DEEP
IN MAIN PROGRAM
AGAIN

```

◆◆◆◆◆ THE SYNTAX ◆◆◆◆◆

A local subroutine can be executed as a program in its own right. When a GO TO SUB statement is written and followed by EXECUTE (*don't use RUN PROGRAM!*) the designated subroutine is treated as if it were a separate program to be run by itself. Any RETURN's, *located within the subroutine itself*, assume the meaning 'go to line 0 and then stop'. Should that subroutine branch to other subroutines, *their* RETURN's are treated as ordinary RETURN's.

Sample Exercise:

If you were to load the program of the previous example, you could execute its subroutines from the keyboard:



ONE LEVEL DEEP
TWO LEVELS DEEP



TWO LEVELS DEEP

— LOCATION OF GTO AND GSB IN A LINE —

A GO TO or GO TO SUB statement can usually be located anywhere among other statements of a line. However, a GTO or GSB is not actually executed until the end of the line has been reached.

If a line contains a combination of GTO's and GSB's, the last one encountered is the one which will be executed when the line is completed.

STOP is also an instruction which is not executed until the end of the line, even though it may be located in the interior of the line. If a line contains both a STOP and either a GO TO or GO TO SUB statement, the program line counter will be set to the new line number, but that line will not be executed.

A GO TO or GO TO SUB statement must follow an ENTER statement if both statements are on the same line. Otherwise, the GO TO or GO TO SUB statement will be ignored.

The following pairs of lines are equivalent:

A+1→A;GTO 5
GTO 5;A+1→A

A+1→A;GSB "→"
GSB "→";A+1→A

GTO 1;GTO 5;GTO 15
GTO 15

A+1→A;STP ;GTO 0
A+1→A;GTO 0;STP

THE SYNTAX

HIGH SPEED BRANCHING

GO TO and GO TO SUB statements are executed in either of two modes: high speed or slow speed. During a slow speed branch the Calculator finds the destination of the branch by counting the number of lines, either from line 0, or from the current line, for absolute and relative addressing, respectively. Symbolic addressing requires a search beginning at line 0.

Once the destination of an individual branching statement is located, the 'internal address' of the destination is actually added into the line (as part of the GO TO or GO TO SUB statement) causing that particular branch. (This is an internal matter, and cannot be observed by recalling lines or listing the program.) The next time that GO TO or GO TO SUB statement is executed, no counting or searching will be required. The machine will 'know', in advance, where the destination is, and simply 'go there' — *provided one other condition is met.*

That condition is that an END statement has been executed prior to starting to run the program, and that no subsequent editing instructions (BACK FORWARD DELETE INSERT) have been used.

The END statement forces the first use of each individual branching statement to occur in the slow speed mode. This is because of possible uncertainty about the validity of the internal address of each destination; by first branching in the slow speed mode, known correct internal addresses are found. These addresses are then used in subsequent branching by each statement.

If, after storing or editing a program, it is started without first executing an END statement, the entire program will be run in the slow speed mode.

Sample Exercise:

Enter the following program:

```

0:
DSP "ABC";IGTO 99
1:
A+Z†
2:
A+Z†
3:
A+Z†
4:
A+Z†
5:
A+Z†
6:
A+Z†
7:

```

These lines are simply to take up space.

```

94:
A+Z†
95:
A+Z†
96:
A+Z†
97:
A+Z†
98:
A+Z†
99:
DSP "ABC";IGTO -9
100:
END †

```

The A+Z's are easily obtained by simply pressing:



The implied store into Z does the rest.

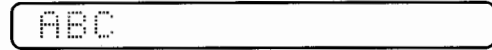
After the program has been loaded, press:



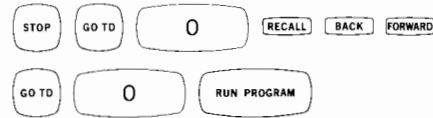
This starts the program and enables the high speed mode to take effect.

THE SYNTAX

You should see a relatively constant display of:



Now press:



This time the display will appear as:



(blinking)

The blinking is caused by the extra time needed to search for destinations while in the slow speed mode.

JUMP STATEMENTS

Literal Syntax:

`(n:) [. . . ;] JMP <value> ↑`
 ↑
 end of line

Figurative Syntax:

`JMP ±n lines`

The JUMP statement is used to branch from a line, whose line number is, say, m, to the beginning of the line whose line number is m+n, or m-n, depending upon whether <value> is positive or negative, respectively.

The JUMP statement is similar to the RELATIVE GO TO statement, except that:

- a. Since JMP can be followed by a value, the destination of the branch can be computed at the time the branch is made, and,
- b. JUMP statements are slower than RELATIVE GO TO's because <value> must be evaluated each time the statement is executed. There is no high speed branching possible with JUMP statements.

An attempt to execute a JUMP statement from the keyboard will either be ignored, or result in NOTE 08, depending upon the circumstances.

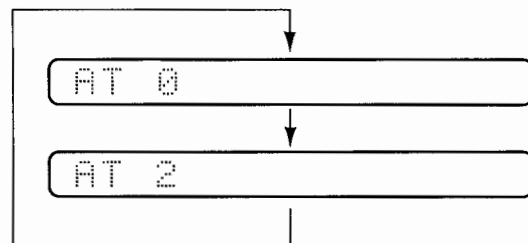
(continued)

Sample Exercise:

Load and run the following program:

```
0:
  DSP "AT 0";JMP 1
+1↑
1:
  DSP "XXX"↑
2:
  DSP "AT 2";JMP -
(1+1)↑
3:
  END ↑
```

Result:



Notice that the value following the JMP in line 0 does not need to be enclosed in parenthesis.

THE SYNTAX

JUMP STATEMENTS

(continued)

If <value> does not represent strictly an integer, the digits to the right of the decimal point are ignored.

JMP <zero> causes the line containing it to be repeated.

Modify the program so that line 0 is:

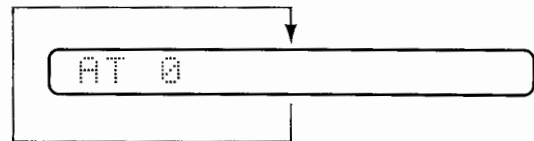
```
0: DSP "AT 0"; JMP 1+1.9F
```

Run the program again. The results should be the same, as 'JUMP 2.9' is equivalent to 'JUMP 2'.

Modify the program so that line 0 is:

```
0: DSP "AT 0"; JMP 0F
```

Run the program again. The result should be:



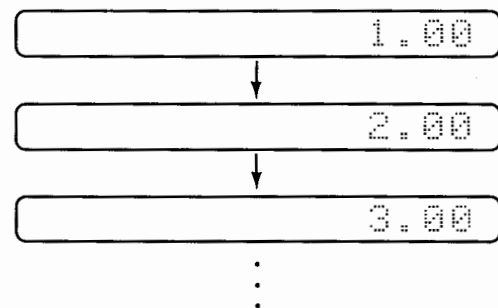
In contrast with the GO TO and GO TO SUB statements, when a JUMP statement is encountered it is executed immediately. For this reason, there is never any point in having other statements to the right of a JUMP statement; they will never be encountered.

Sample Exercise:

Store and run the following program:

```
0:
FXD 2:0+AF
1:
DSP A+1+A; JMP 0:
DSP "XXX"
2:
END F
```

Result:



Note that XXX is never displayed.

◆◆◆◆◆ THE SYNTAX ◆◆◆◆◆

Relational and mixed expressions are often combined with JUMP statements to form incrementing and decrementing loops.

Sample Exercise:

Enter and run the following program:

```
0:
FXD 2:0+A+
1:
PRT A;JMP (A+1+A
)=11+
2:
SPC 8;END +
```

Result:

```
0.00
1.00
2.00
3.00
4.00
5.00
6.00
7.00
8.00
9.00
10.00
```

————— RESTRICTIONS ON BRANCHING —————

With one exception, any branching statement will result in NOTE 08, as an error during execution, if the destination of the branch is not an existing line stored in the Calculator. The exception is this: The program line counter may be set to the line number which is one higher than the highest line number currently stored.

If the exceptional case should occur during the execution of a program, the Calculator will branch to that 'line' and treat it as if it were:

```
<m:>END +
```

Any attempt to branch (GO TO, GO TO SUB, JUMP, and IF statements) will result in NOTE 01 if it is attempted while the Calculator is in the Enter Mode.

The Calculator has to allow the program line counter to be set to one higher than the number of the highest currently stored line. Otherwise, you could never store the 'next line' of a program.

It is not possible to set it higher than that because to do so would require it to anticipate the place in memory where that line would eventually start. (The program line counter 'addresses' the start of a line, but there would be at least one missing line ahead of the designated line.)

 THE SYNTAX

 FLAG STATEMENTS

Literal Syntax:

SFG <value>

There are 16 flags, named flag 0 through flag 15. See "FLAGS", page 5-2, for a discussion of the role of flags.

Figurative Syntax:

SFG n

Sample Exercise:

```
SFG 0F
SFG 1F
-1→AF
SFG A+1→AF
```

Where $0 \leq n \leq 15$ denotes the desired flag.

A SET FLAG statement is used to set a flag (make its numeric value be 1).

If <value> is not strictly an integer, the digits to the right of the decimal point are ignored.

NOTE 05 will result, as an error during execution, if the integer part of <value> is less than 0, or greater than 15.

Press EXECUTE several more times. What's happening?

If, while the Calculator is running a program, SET/CLEAR FLAG N is pressed, the machine will detect this and automatically assume that flag 0 is to be set. Flag 0 will be set without interrupting the program.

Sample Exercise:

Load and run the following program:

```
0:
0→A;DSP "PRESS S
FG";" ";JMP FLG
0F
1:
CFG 0;DSP " THA
NKS !";JMP -((A+
1→A)=30)F
2:
END F
```

Literal Syntax:

CFG <value>

The mnemonic CFG is produced by pressing the SET/CLEAR FLAG N key twice in succession. Once the mnemonic CFG is present in the display, it cannot be separated into its original components of 'SFG SFG' through the use of BACK or DELETE; CFG is treated as is any other mnemonic.

Figurative Syntax:

CFG n

Where $0 \leq n \leq 15$ denotes the desired flag.

A CLEAR FLAG statement is used to clear a flag (make its numeric value be 0).

THE SYNTAX

If $\langle \text{value} \rangle$ is not strictly an integer, the digits to the right of the decimal point are ignored.

NOTE 05 will result, as an error during execution, if the integer part of $\langle \text{value} \rangle$ is less than 0, or greater than 15.

All flags are automatically cleared when an END statement (see page 5-36) is executed while the program line counter is within the mainline programming area (as opposed to an area defined by the User Definable Functions ROM).

Literal Syntax:

FLG $\langle \text{value} \rangle$

Figurative Syntax:

FLG n

Where $0 \leq n \leq 15$ denotes the desired flag.

This syntax is used to identify a particular flag in situations other than setting or clearing that flag.

If $\langle \text{value} \rangle$ is not strictly an integer, the digits to the right of the decimal point are ignored.

NOTE 05 will result, as an error during execution, if the integer part of $\langle \text{value} \rangle$ is less than 0, or greater than 15.

Sample Exercise:

```
FXD 0:0→A+
SFG A:CFG A+1:A+2→A
```

Press EXECUTE until the display is:

16

```
-1→A+
DSP FLG (A+1→A)
```

Press EXECUTE several times.

1

0

1

⋮

Continue pressing EXECUTE. What happens? Why?

```
FLG 0+FLG 2+FLG 4→X+
```

3

In addition to flag 0, other flags that have special properties are flags 13, 14, and 15.

There is an example of the use of flag 13 on page 5-21.

 THE SYNTAX

 FLAG STATEMENTS

(continued)

If flag 14 is set, certain arithmetic faux pas are no longer treated as errors during execution. They will set flag 15, however, regardless of the value of flag 14.

```
END F 0/0F
```

```
NOTE 10
```

```
SFG 14F 0/0F
```

```
9.999999999E 99
```

```
FXD 0;DSP FLG 15F
```

```
1
```

 IF STATEMENTS

Literal Syntax:

```
<n:~[ ... ; ] IF <value> ; <statement>
      [ ; <statement> ] ...
```

```
<n+1:~<line>
```

Figurative Syntax:

```
n: ... IF condition ; then ...
n+1: otherwise ...
```

IF statements are a powerful means of branching. If the condition following IF is true, then the remainder of the line is executed. Should the condition be false, however, the execution of the line is terminated at that point, and the *next* line is executed.

Most often <value> is a relational or mixed expression. The numeric value of the expression is used to determine if the condition is true or false. If the absolute value (in the arithmetic sense) of <value> is 1, or greater, the 'condition' is assumed to be 'true'; otherwise it is taken to be 'false'.

The line segment to the right of <value> usually contains a branching statement. Used in that way, an IF statement has the figurative meaning: if some condition is met, branch to somewhere, otherwise, don't branch there, but go to the next line instead.

The following are typical IF statements:

```
IF A=B;GTO 15
IF FLG 13;... or IF FLG 13=1;...
IF (A=B)(X=Y)=1;GTO 10
IF 0<A;-A+A
IF FLG 5=0;SFG 6
```

(continued)

 THE SYNTAX

Sample Exercise:

Enter and run the following program:

```

0:
SPC 4;ENT A,B;
1:
IF A>B;PRT "A>B"
;GTO 0;
2:
IF A=B;PRT "A=B"
;GTO 0;
3:
PRT "B>A";GTO 0;
4:
END ;
  
```

Enter various numbers for A and B, and see what happens. In particular, try letting A be .999999999988 and B be 1. What happens? Why? (Hint: see the bottom of page 5-4.)

 STOP STATEMENTS

Literal Syntax:

```
STP
```

Figurative Syntax:

Stop at end of current line.

STOP statements are used to interrupt or terminate the execution of a program.

If a STOP statement is located within the main-line programming area, *but not within a local subroutine*, the execution of the program can safely be resumed by pressing RUN PROGRAM — provided nothing has been done to alter pertinent conditions existing at the time the program halted.

However, if the STOP statement was located within a local subroutine, or within a subprogram written using the User Definable Functions ROM, consider the program as being terminated. The reason for this is that any editing or execution of a line, after the STOP, destroys the return branching information associated with the nesting of subroutines and subprograms.

Pressing STOP while a program is running will halt the execution of the program. The effect is the same as if the Calculator had encountered a STOP statement in the line it was executing.

Sample Exercise:

Load the following program.

```

0:
PRT "A";
1:
GSB "1";
2:
PRT "B";SPC 4;
3:
GTO 0;STP ;
4:
"1";PRT "1";SPC
2;STP ;
5:
RET ;
6:
END ;
  
```

(continued)

THE SYNTAX

STOP STATEMENTS

(continued)

Even though a STOP statement may be located in the interior of a line, it is not executed until after the completion of all other activity generated by that line.

When Calculator stops, the display will generally contain some information related to whatever happened before the STOP took effect.

Press:



Result:

A
1

Press:



Result:

B

Now, press:



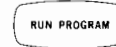
Result:

A
1

Press:



Press:



Result:

NOTE 07 IN 5

END STATEMENTS

Literal Syntax:

(n:)[...:] END F ← end of line

Figurative Syntax:

Clear all flags, go to line 0, and stop.

END statements are used in establishing certain initial conditions prior to running a program, and also as a means to terminate a program.

END statements are an important part of the mechanics of Model 20 operation. Chapter 3 contains explanations and examples of that aspect of using END statements.

Because storing or modifying a line containing an END statement deletes the lines of programming having higher line numbers, END statements are seldom used in the interior of a program. They are usually used only in the highest numbered line to be stored.

THE SYNTAX

A number of things happen when an END statement is executed or stored. They are:

- a. Executing an END statement sets the program line counter to zero.
- b. If an END statement is encountered during the execution of a program, the program is halted, and any nesting information is lost.
- c. All flags are cleared when an END statement is executed.
- d. When a line containing an END statement is stored, or, such a stored line is modified through the use of the editing ability, the existing subsequent lines in that programming area are deleted.

It is recommended that an END statement be executed before starting a program, because this accomplishes needed initialization, such as guaranteeing that no flags are set except for the ones deliberately set by the user. It also ensures that the program will be run with high speed branching.

RECORD STATEMENTS

Literal Syntax:

[[. . . ;] (ABSOLUTE GO TO or END statement) ;] REC F
↑
end of line

Figurative Syntax:

Record programming only.

This syntax is used to record programs on magnetic cards, and should be used only from the keyboard.

The record produced consists of a series of (compiled) lines; *the line numbers of those lines are not part of the record.* This permits a program, that is currently located in one part of the memory, to be recorded and later loaded back into a different part of the memory. The line numbers assigned to that programming will depend upon where the program line counter is set when the record is loaded.

When recording programs, a record starts with the line whose number is specified by the GO TO or END statement preceding REC. All subsequent lines, down to and including the highest numbered line currently stored in the mainline programming area, are made part of the record. It is possible to control where the record starts — but not where it stops. END and STOP statements within the programming being recorded exert no special effects upon the recording process.

Whenever a statement involving the mnemonics REC or LOD is executed, the card reader motor will start. It is a good practice to allow the motor to start before inserting any cards into the (upper) slot of the card reader. The card can be removed from the lower slot of the card reader once the card stops moving.

See pages 2-2 and 2-3 for additional information about using the magnetic card reader.

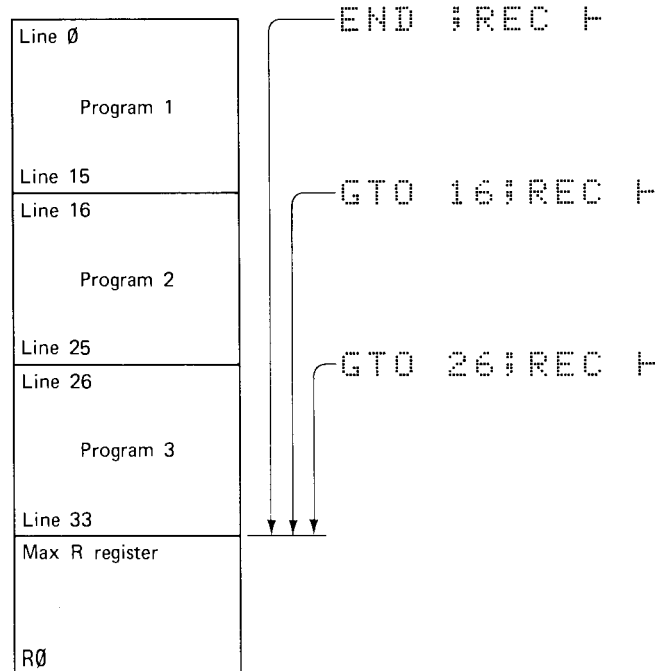


Figure 5-2. Recording Programs.

 THE SYNTAX

 RECORD STATEMENTS

(continued)

NOTE 13 will result from an attempt to record information on a side of a card which has been protected, by snapping out the protect tab for that side.

NOTE 14 occurs whenever another side is needed to continue making the record. Press EXECUTE and then insert another side.

Identify the sides, so that they can later be loaded back into the Calculator in the sequence in which they were recorded.

Any RECORD operation can be terminated by pressing STOP.

A record of programming information includes the type, and configuration of, any plug-in ROM's that are installed in the Calculator when the record is made.

Suppose the mainline programming area has three programs stacked in it, represented by Figure 5-2.

Program 3 can be recorded as an individual program, simply by executing:

```
GTO 26;REC F
```

However, to record program 2 as an individual program, you must first delete program 3. Do this by storing an END as the first line of program 3.



(At this point you may wish to delete the END statement just stored, as it will be considered part of program 2.)

In the same way, program 1 cannot be recorded as an individual program until programs 2 and 3 have been deleted.

 "SECURE" RECORD STATEMENTS

Literal Syntax:

```
[[[...]](ABSOLUTE GO TO or END
statement)]REC "SE"↑
end of line ↑
```

Figurative Syntax:

Record private programming.

This syntax is a means of recording private programming onto magnetic cards. If such a record is loaded back into the machine, using the LOAD statement (see pages 5-41 and 5-43), *any attempt to list the program, record the program on magnetic cards, or, view or edit any lines of the program, will result in destruction of all programming stored in the User's RWM.* Numerical data stored in memory is not affected.

Insofar as recording cards is concerned, REC "SE"↑ has all the properties of REC F. The only difference is that the record is identified as containing "secure" programming.

Sample Exercise:

Key in the following program:

```
0:
PRT "THIS IS A S
ECURE"; "    PROG
RAM!"↑
1:
SPC 1↑
2:
PRT "    TO RUN TH
IS "; "PROGRAM YO
U MUST"; "KEY IN
THE RIGHT"↑
3:
PRT "    CONTROL CO
DE!"↑
4:
SPC 2↑
```

THE SYNTAX

If a secure program is stacked in the memory along with other programs that are not secure, all the programming in the Calculator is treated as secure programming.

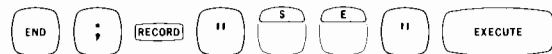
Secure programs can be run in the Trace Mode.

There is no way to key in a program and have it treated as a secure program. To become secure, a program must first be placed in the memory by conventional means, and then recorded with REC "SE"†. Then the record will contain the secure program; however, the programming still in the Calculator will not be secure, and can still be listed and recorded, etc.

```

5:
PRT "++++++"
++++"ISPC 8†
6:
ENT "      CODE =
?"†;A†
7:
IF A=1234;GTO +2
†
8:
DSP "      SORRY
!"†;DSP ;DSP ;
DSP ;DSP ;REC †
9:
DSP "      E=MC†2
"†;DSP ;DSP ;DSP
;DSP ;DSP ;DSP ;
DSP ;DSP ;DSP ;
DSP ;GTO 6†
10:
END †
    
```

Record the program on a card by pressing:



Now insert the card.

To load this secure program, press:



Insert the recorded side of the card. (The original programming wasn't secure – only the programming on the card is secure.)

To run the program, press:



The 'control code' mentioned in this program is 1234. If any but the correct digits are entered by the user, the program will (what else?) self-destruct. That is done by branching to a REC in the program (line 8).

5-40 THE MODEL 20 LANGUAGE

THE SYNTAX

RECORD "DATA" STATEMENTS

Literal Syntax:

```
REC "DA";R<value>
```

Figurative Syntax:

Record R0 through Rn, inclusive.

This syntax is used to record the contents of R registers onto magnetic cards. The record will consist of R0, up through, and including, the R register specified by <value>.

If <value> is not strictly an integer, the digits to the right of the decimal point are ignored.

If R <value> is missing, the entire statement will be ignored.

NOTE 02 will result if the integer portion of <value> is less than zero, or greater than the number of available R registers.

NOTE 13 and NOTE 14 have the same meanings for the RECORD DATA statement as they have for the other RECORD statements.

Unlike records of programming, where the line numbers are not part of the record, the names of individual registers involved are part of a record of data. Thus, if R0 through R10 are recorded, they *cannot* be loaded back in as R20 through R30. Data will automatically be loaded back into the same registers from which it was recorded.

Sample Exercise:

First, use the following program to put data into R0 through R100.

```
0:
0→A↑
1:
A→RA;IF 100>A;A+
1→A;JMP 0↑
2:
END ↑
```

To record the data, execute the line:

```
REC "DA";R100↑
```

Insert the first side. After it has been recorded, you will see:

NOTE 14

Press EXECUTE and insert the next side. Continue doing this until the display returns as:

↑

Now press MEMORY ERASE, and load the following program:

```
0:
-1→A;FXD 2↑
1:
PRT R(A+1+A);
JMP A=100↑
2:
END ↑
```

Now press LOAD EXECUTE. Insert the first side of the previously recorded data. When NOTE 14 appears, press EXECUTE again, and insert the next side. After all sides have been inserted the display will show only the end-of-line symbol. Then press END RUN PROGRAM.

THE SYNTAX

REC "DA" does not affect the setting of the program line counter, nor does that setting affect the operation of recording data.

REC "DA" is programmable, and can be located anywhere within a line.

A record of data contains no information about the configuration of plug-in ROM's installed in the Calculator.

There is no way to make a record of data be secure.

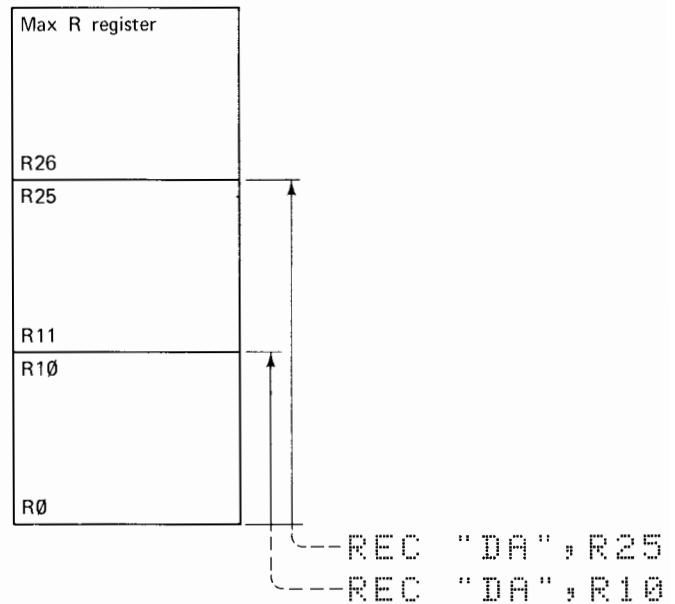


Figure 5-3. Recording Data.

KEYBOARD LOAD STATEMENTS

Literal Syntax:

[... ; {ABSOLUTE GO TO or END statement} ;] LOD F

Figurative Syntax:

Load one record of information from magnetic cards.

This syntax is used, from the keyboard only, to load whatever information has been recorded. The Calculator will recognize the type of information on the card because on each side there is a recorded heading that specifies that (and other) information. Be sure to insert the cards into the card reader in the sequence in which they were recorded.

NOTE 12 will occur if there is more information in the record than can be stored in the memory.

NOTE 14 has the same meaning for LOAD statements as it does for RECORD statements — press EXECUTE and insert the next side.

A LOAD operation can be terminated by pressing STOP.

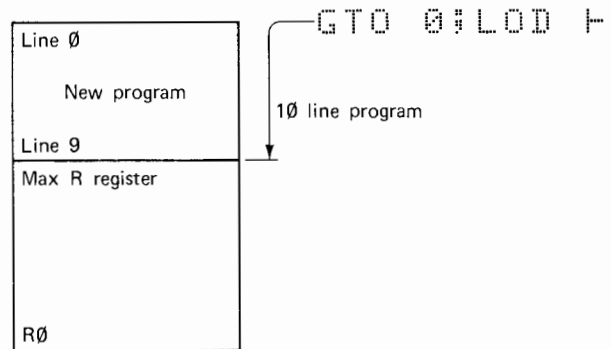


Figure 5-4. Loading a Single Program.

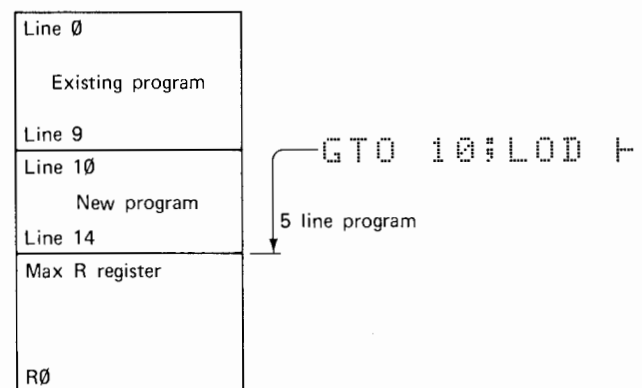


Figure 5-5. Stacking Programs.

(continued)

THE SYNTAX

KEYBOARD LOAD STATEMENTS

(continued)

NOTE 15 will occur, after a record of programming has been loaded, if any plug-in ROM's are not in the same slots they were in when the program was recorded. However, other ROM's installed in slots that were empty when the program was recorded will not cause NOTE 15.

When loading a record of programming, precede the LOD with an END or ABSOLUTE GO TO statement. The sequence of line numbers assigned to the lines of programming will begin with the line number specified by the END or ABSOLUTE GO TO statement, say, n. The second line of programming will become line number n+1, etc.

No addressing statement needs to precede LOD when the record to be loaded consists of data. The data will automatically be placed back into the same registers from which it was recorded.

NOTE

If a program was recorded with the User Definable Functions ROM installed, be sure that the ROM is again installed before the program is loaded back into the Calculator, even though the program in question may not actually use capabilities of the UDF ROM! Otherwise, the END and RETURN statements in the program will be incorrectly loaded into the memory. The mnemonic END will be replaced by an arbitrary character.

See the Operating Manual for the User Definable Functions ROM for further information.

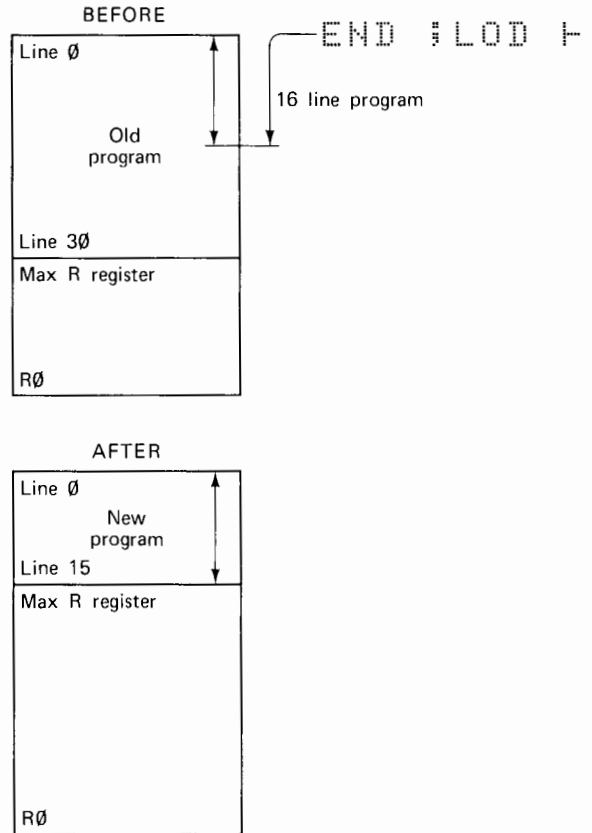


Figure 5-6. Loading a New Program over an Old Program.

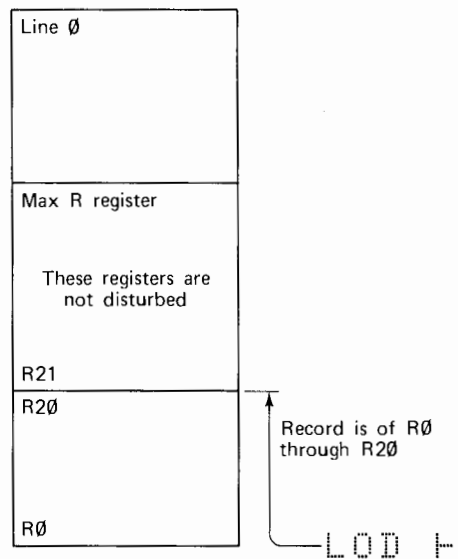


Figure 5-7. Loading Data.

THE SYNTAX

STORED LOAD STATEMENTS

Literal Syntax:

(!:{...;}{any GO TO statement}; LOD F
 end of line ↗

Figurative Syntax:

GTO n; LOD ;GTO n RUN PROGRAM

This syntax is used, *in the context of a program only*, to load data or mainline programming, and then automatically continue the execution of the program.

The GO TO statement can be any type of GO TO statement: absolute, relative, or symbolic.

When LOD is reached, program execution is halted, and the card reader motor starts. As cards are inserted, NOTE's 12, 14, and 15 have their usual meanings, although NOTE 12 can have some additional meanings.

NOTE 12 will occur if the GO TO statement is missing, or, if the GO TO statement specifies a negative line number, or a line number greater than one higher than is currently stored.

If programming is to be loaded, the lines are loaded starting at the line number specified by the GO TO statement, and program execution *will automatically resume at that same line number*, once the record has been loaded.

NOTE

The GO TO statement preceding LOD is needed even though the record to be loaded consists of numerical data.

Even though the record to be loaded consists of data, the GO TO statement will perform its designated function. Be sure you choose a parameter for the GTO which causes the execution of the program to resume in the appropriate place.

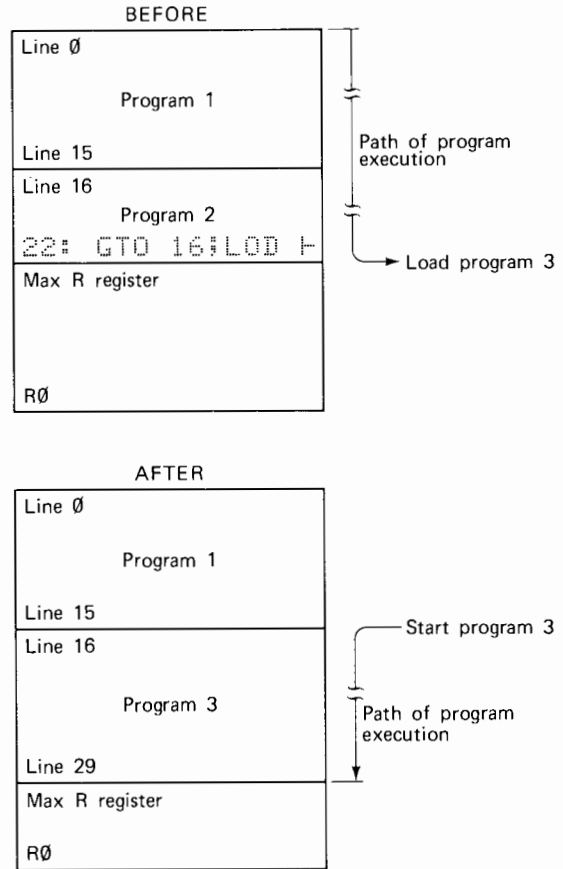


Figure 5-8. Automatic Linking of Programs.

5-44 THE MODEL 20 LANGUAGE

THE SYNTAX

STORED LOAD STATEMENTS

(continued)

Literal Syntax:

```
<I>[. . . ;] <any GO TO SUB statement> ; LOD F  
                                     ↗  
                                     end of line
```

Figurative Syntax:

```
GTO n ; LOD ; GTO 0 RUN PROGRAM
```

Except for one difference, this syntax behaves in a manner identical to the previous one. The difference is this: After a program has been loaded, beginning at the line number specified by the GO TO SUB statement, the execution of the program *automatically resumes at line 0*, rather than at the point where the new lines were loaded.

NORMAL AND TRACE STATEMENTS

Literal Syntax:

```
[. . . ;] TRC F  
                ↗  
                end of line
```

Figurative Syntax:

Establish the Trace Mode.

Literal Syntax:

```
[. . . ;] NOR F  
                ↗  
                end of line
```

Figurative Syntax:

Cancel Trace Mode Operation.

These syntaxes are used to establish and cancel the Trace Mode. While in the Trace Mode, the Calculator makes a printed record of the results of its activities.

When the Calculator is not in the Trace Mode, it is said to be in the Normal Mode.

Pressing TRACE while the Calculator is running a program will establish the Trace Mode (provided the program does not execute its own NORMAL statement) without halting the program. NORMAL has a similar ability.

See "TRACE MODE OPERATION", page 3-14, for a complete description of the Trace Mode.

THE SYNTAX

MISCELLANEOUS

Literal Syntax:

R⟨quantity⟩ or R (⟨expression⟩)

Figurative Syntax:

R_n

Where $0 \leq n < n_{max}$.

This form is used to designate R registers. If ⟨quantity⟩ or ⟨expression⟩ is not strictly an integer, the digits to the right of the decimal point are ignored.

Of course, n_{max} is the number of available R registers, and n itself must be less than n_{max} because R register numbering starts at 0, rather than at 1.

NOTE 05 will occur if n is less than zero, or greater than or equal to n_{max} ; n is the integer part of ⟨quantity⟩ or ⟨expression⟩.

The syntax:

⟨GO TO ...⟩ ⟨IF ...⟩ ⟨GO TO ...⟩
 ⟨IF ...⟩ ⟨GO TO ...⟩

is often useful when it is necessary to branch to one of several places from one line, but it is difficult or impossible to use the computing ability of the JUMP statement for that purpose. It's called the 'n-way branch'.

Briefly, it works like this: A GO TO statement is not executed until the end of the line, or until the line is terminated, whichever comes first. Then if more than one GO TO statement has been encountered by that time, it is only the latest one that gets executed. A 'met' IF statement allows the execution of the line to continue from left to right, but a 'failed' IF statement will terminate the line at that point. Then the previous GO TO statement will be executed. The syntax can be thought of as a series of 'GO TO IF NOT's'.

R- will cause a syntax error (NOTE 01). Use R (- instead.

The most general form for R is:

R (⟨value⟩)

Since R (⟨value⟩) is itself a value, it follows that the following forms are also valid:

RRR5 RR(A+RR5)

Remember that R is obtained by pressing the R () key, not the half-key called R.

See "DESIGNATING A REGISTER", page 4-2, for a detailed explanation of R register designation.

Consider the following line:

```
25: GTO 10;IF A>10;GTO 12;
IF A>20;GTO 14;IF A>30;GTO 16-
```

This line branches to: line 10 if A is less than or equal to 10; to line 12 if A is greater than 10 but less than or equal to 20; to line 14 if A is greater than 20 but less than or equal to 30; and to line 16 if A is greater than 30.

 THE SYNTAX

 MISCELLANEOUS

(continued)

The syntax:

```

<n> [....]
    <GO TO SUB statement> ; <JUMP statement> ;
                                     ↑
                                     end of line
  
```

can be thought of as a 'JUMP TO SUB' or 'COMPUTED GO TO SUB' statement.

Here is what happens: The GO TO SUB statement will not be executed until the end of line. But the end of the line contains a JUMP statement which overrides the GSB. The destination of the branch will be determined by the JUMP statement, but the GO TO SUB statement will still cause an automatic return branch to be established for a corresponding RETURN statement.

The parameter for the GO TO SUB statement does not determine any of the branching (its a *dummy* parameter), yet it still must be an otherwise valid parameter, or NOTE 08 will result as an error during execution.

Consider the following program:

```

0:
PRT "0"
1:
PRT "1"
2:
PRT "2" ; GSB -1 ;
JMP 5
3:
PRT "3"
4:
PRT "4"
5:
PRT "5"
6:
PRT "6" ; STP
7:
PRT "7"
8:
PRT "8"
9:
PRT "9"
10:
PRT "10"
11:
RET
12:
END
  
```

When run, it produces the following results:

```

0
1
2
3
4
5
6
  
```

◆◆◆◆◆ THE SYNTAX ◆◆◆◆◆

Sometimes you might wish to store a blank line, like:

0: ┆

To do that it is only necessary to press STORE while the display is either blank, or has just been cleared with CLEAR.

Sample Exercise:

Press END EXECUTE CLEAR STORE.

0: ┆

There is a special use of CLEAR which is useful when replacing one line of programming with another line.

CLEAR will maintain the program line counter at n , if it is pressed while a replica of the line numbered n is in the display.

Then the new line can be written, and stored. It will replace the old line numbered n . If CLEAR were not pressed, and the new line written and stored anyway (the replica will vanish as the first key of the new line is pressed), the line would be stored as the line numbered $n+1$.

Let's say that you wished to replace line 5 with an altogether different line. Here are three ways to do this according to the procedures of Chapter 3:

- a. Press GO TO 5 EXECUTE, write the line and press STORE.
- b. Press GO TO 5 RECALL, then use BACK to get to the front of the line, write the new line and then press STORE.
- c. PRESS GO TO 5 RECALL DELETE, then set program line counter back to line 5, write the new line, and press INSERT STORE.

Using CLEAR, you would press: GO TO 5 RECALL CLEAR, then write the new line, and then press STORE.

◆◆◆◆◆ REVIEW ◆◆◆◆◆

The following concepts were presented in this chapter.

1. A constant is a number that is expressed in an explicit manner, as part of a line.

A real constant is an ordinary number.

A 'constant' in the problem solving sense is not necessarily the same thing as a constant in the syntactical sense.

2. A variable is a unit of memory. We don't generally distinguish between the name of that unit of memory, and the thing that's stored in it.

A real variable is a register.

3. A quantity is any variable or constant.
4. A literal is a series of characters in quotation marks. Literals make possible labels and symbolic addressing.
5. A flag is a variable that is used to represent a condition. A flag can be either 'set' or 'clear', which correspond to the numeric values 1 and 0, respectively. There are 16 flags, named flag 0 through flag 15.

Certain flags have special properties:

- a. Flag 0 can be set while a program is running by pressing the SET/CLEAR FLAG N key.
- b. Flag 13 denotes that no entry was made during an ENTER statement.

REVIEW

- c. Flag 14 permits certain arithmetic faux pas.
 - d. Flag 15 is automatically set by such a faux pas.
6. An operator is a symbol that indicates some computational activity, or a relationship, between two values. The arithmetic and relational operators are subject to a hierarchy.
 7. Supplied functions are those functions whose properties are determined by the design of the machine. Using the User Definable Functions ROM, you can write special functions of interest to you. They are called definable functions. In general, both types are used in the same way.
 8. A mathematical expression contains no operators other than arithmetic operators, and is used for general computational purposes.

Relational expressions involve relational operators, and are most often used in forming tests for certain conditions.

Every mathematical or relational expression has a numeric value. The numeric value of a mathematical expression is obtained simply by performing the computations it describes. The numeric value of a relational expression is 1 if the expression is 'true', and 0 if it is 'false'.

Complex relational expressions are evaluated by replacing the component relational expressions with their individual numeric values. Mixed expressions are evaluated in the same way.

9. A value is any quantity or expression.
10. An instruction is a command, other than an operator or function, to do something. 'PRINT' is an instruction.
11. A statement is a complete syntactical unit, and specifies an activity that can be performed.
12. A line is a sequence of statements, separated by semi-colons. The statements in a line are generally executed from left to right, but

certain types of statements are not executed until the line is otherwise finished. These are STOP, GO TO, and GO TO SUB statements.

13. A label is a means of identifying a line; it is a literal in the left-most position of a line.
14. Parameter lists are used by several instructions to specify the exact activity that is to be performed.

For example:

```
PRT "A=", A, "B=", B
```

15. A program is a sequence of stored lines. Each line in a program has a line number, which was automatically assigned by the Calculator at the time the line was stored. Line numbers are the consecutive unsigned integers 0, 1, 2, 3, ... Line numbers always start at 0. The natural order of execution for lines is by line numbers, in increasing numerical order.
16. Branching and addressing are terms that describe a program's ability to alter the natural order in which the lines of a program are executed.
17. Relative addressing is a means of branching from one line to another line, by indicating that the destination is so many lines ahead or behind the current line. The actual line numbers are not involved.
18. Symbolic addressing is a means of branching to a line identified as possessing a certain label. Neither the line numbers of the lines, nor their relative positions in the program, have any effect on the branch.
19. A record is a series of one or more sides of magnetic cards upon which are preserved some programming or some data. Programs and data cannot be preserved together on the same record.
20. A subroutine is a program segment which performs some useful task for another program. The type of subroutines that can be written with the basic Calculator are called local subroutines.

REVIEW

21. A literal syntax describes the technical details of the form that a given type of statement must have. In other words, it specifies what is permissible in terms of the various language elements. However, it doesn't necessarily say what the statement will do when it is executed.

A figurative syntax is a representation of what the statement will do, or of what certain parts of the statement mean.

22. **FIXED** and **FLOAT** statements are used to specify what form printed and displayed results will assume.

Numbers too large to appear in the current fixed point specification appear under the previous floating point specification.

FIXED and **FLOAT** statements do not affect the appearance of literals.

23. The **DISPLAY** statement is used to place numerical information or messages in the display.

24. **PRINT** and **SPACE** statements are used to print information with the printer. To print alpha information, put a literal in the list following the **PRINT**. **SPACE** statements use the **SPACE N** key, rather than the half-key called **SPACE**.

25. Assignment statements are used to put numbers into registers.

Multiple assignment statements are used to put the same number into several registers, and to save intermediate results of calculations.

```
0→A→B→C←      10(A+B→C)+X→Y←
```

Because of the implied store into Z, a line like

```
A+B is stored as A+B→Z
```

The implied store into Z never affects the elements of a parameter list.

26. **ENTER** statements are the primary means used to enter data into the Calculator while a program is in progress.

Press **RUN PROGRAM** to cause the Calculator to accept an entry.

An entry must appear 'on the left' in the display; an executed entry will not be entered, even if **RUN PROGRAM** is pressed next.

ENTER statements cannot be nested, nor executed from the keyboard.

27. The **ABSOLUTE GO TO** statement is used to branch from one line to the beginning of another line explicitly identified by its line number. There is no 'computed GO TO', nor is there any way to branch into the middle of a line.

28. The **RELATIVE GO TO** statement is a means of relative addressing. The parameter must be an integer constant.

29. The **LABELED GO TO** statement is a means of symbolic addressing.

30. **GO TO SUB** statements branch to local subroutines, and can have any form permitted for a **GO TO** statement. Local subroutines can be nested no deeper than 31 levels, although the limit may be lower than that, depending upon the circumstances.

31. A **RETURN** statement causes a branch back to the line following the one which branched to the subroutine.

32. **JUMP** statements are a means of relative branching by either an explicit or computed amount.

33. **GO TO** and **GO TO SUB** statements can use the high speed mode of branching; **JUMP** statements cannot. An **END** statement must be executed prior to starting the program, or all branching will be done in the slow speed mode.

34. No branching statements can be executed while the Calculator is in the Enter Mode.

35. **SFG** is used to set a flag.

CFG is used to clear a flag. **CFG** is generated

REVIEW

by pressing the SET/CLEAR FLAG N key twice in succession.

FLG is used to designate a flag for purposes other than setting or clearing it.

Executing an END statement clears all the flags.

36. IF statements are used to branch according to some condition.

An IF statement is 'met' if the numeric value of the <value> following it has an absolute value of 1 or greater, otherwise it is 'failed'. If the IF is met, the execution of the line is continued; should the IF be failed, the line is terminated at that point.

37. STOP statements are used to interrupt or terminate the execution of a program. If the STOP occurs within any nesting, consider the program terminated. Otherwise it may be restarted, provided no other vital conditions (such as the setting of the program line counter or the values of pertinent variables) have been altered.

38. END statements are used to terminate the execution of a program, and also in establishing certain initial conditions prior to running a program.

Executing an END statement sets the program line counter to 0, and clears all flags.

Storing an END statement deletes all subsequent lines of existing programming.

39. When programming is recorded, the line numbers are not made part of the record. This allows that programming to later be loaded back into the memory and have different line numbers.

When recording programs, the process starts with the line number specified by the GO TO or END statement accompanying the REC F. It ends when the last line in the mainline programming area has been recorded.

40. An attempt to duplicate on magnetic cards, edit, list, or otherwise learn the contents of a secure program will destroy the program.

Data cannot be made secure.

A program cannot be keyed in and then, at that time, be treated as a secure program by the Calculator.

41. A record of data does not contain the names of the registers from which the data was recorded.
42. A LOAD statement puts the information of a record into the memory of the Calculator.

The Calculator automatically knows where to put the information of a data record; the necessary information is on the card.

To load a record of programming, the Calculator must know at which line to start loading the memory. This piece of information comes from a GO TO or GO TO SUB statement preceding the LOD F.

EXERCISES

Answers are on page A-2.

1. Find at least one error in each of the following lines.

- a. (A+B)/-2.2X→Z;GTO A
- b. "START;ENT "N=?",X+1→Y
- c. PRT ;r(-AA+BB)→X,SPC
- d. (5X+3YY)/2B→5A+1.3→R10
- e. ENT ,"A=?";A,"B=?";B
- f. r(-Y↑2+(A-B)↑-3)→X;GTO 2.0
- g. A+1→A;JMP A+B;C-B→X
- h. REC "DA";R25
- i. 4: FXD ;PRT "SUM=";A;LOD F
- j. 0→FLG 5;SFG FLG 5

2. What is the largest number that can be displayed in fixed point? The smallest? The closest to zero? What is the largest number that can be displayed under a fixed 5 specification?

3. Write a relational expression that has a value of 1 if A equals B and B equals C.

4. Write a statement that assigns X the value 2A+B if Y is greater than 100, and the value 2A-B if Y is less than or equal to 100.

5. Write an ENTER statement that assigns a value to A, and also to B and C at the same time.

6. Write a line that adds 1 to the value of C until C equals 25, and then branches to line 10. Write it once using IF, and once without IF. (Assume an initial value of -1 for C, and that the line to be written will have line number 3.)

7. Write a statement that assigns the number in the R register designated by the contents of R5 to the R register designated by twice the number stored in B.

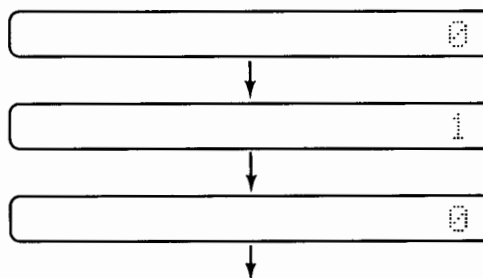
8. Write and execute the line:

```
FXD 0;1+Z↑
```

Now write:

```
Z=0
```

Press EXECUTE several times:



Explain the results in the display.

9. What is the difference between the following pairs of statements:

- a. DSP FLG A+1→A
DSP FLG (A+1→A)

Will the top line turn into the bottom one if it is stored?

- b. X+RAB→Z
X+RA*B→Z
- c. (For Math ROM enthusiasts)
LOG 10↑2
LOG TN↑2



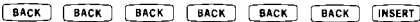

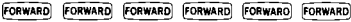
Hint: TN↑ is a single mnemonic designating the function 'anti log base 10'.




10. Write a JUMP statement that is equivalent to the following n-way branch:

```
GTO -5;IF X>25;GTO -8;  
IFX>50;GTO +3
```

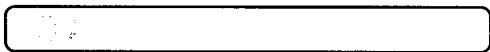

ANSWERS TO THE EXERCISES

CHAPTER 3


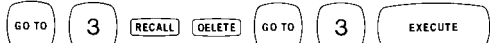



1. Press BACK until you see that part of the line. Then press FORWARD to return to the end of the line. Don't worry about pressing FORWARD too many times – it won't advance past the end of the line.
2. The line is too long. Press CLEAR and write a new line. Do not attempt to salvage the old line.
3. The comma should be a semicolon.
4. Press:  
5. 


6. The first four lines of the only program stored in the Calculator *must* be the lines numbered 0, 1, 2, and 3. You could set the program line counter to 4, but not to 5.
7. Press RECALL.

8. 

 OR


until the display is:



Then,


 OR


 OR



9. NOTE 01 would appear in the display, as PRT A, is an incomplete statement. The B,C will not be considered part of the line until they are again made part of the display.
10. There is not enough memory for the longer version produced by the modification. Lines 32 through 59 were unable to move 'down' far enough to make room for the new line 31.
11. Remember that when a line containing an END is stored, all lines following that line are deleted. As you use a series of consecutive FORWARD's to step through the program, each 'completed replica' is restored, as the machine cannot be sure that you did not change something while the replica was at your disposal. Hence the missing lines.

A series of consecutive RECALL's does not do this, as each RECALL only brings the next replica into the display – what is already in the display is ignored.

CHAPTER 4

1. a. $A(B+C)$ or $AB+AC$
- b. $-(A/2)$ or $-A/2$
- c. $(X*Y)/(A*B)$ or XY/AB
- d. $(X/Y)/(A/B)$ or XB/YA
- e. $r(-(A-B))$ or $r(B-A)$
- f. $r((X-Y)(X-Y)+(A-B)(A-B))$
 or
 $r((X-Y)\uparrow 2+(A-B)\uparrow 2)$
- g. $A*Y+BX+C*Y$ or $X(AX+B)+C*Y$
- h. $r256\div A$ or $16\div A$
- i. Give yourself a gold star if you noticed that $-a^b$ is ambiguous.
 $(-a)^b = (-A)\uparrow B$
 $-(a^b) = -A\uparrow B$

ANSWERS TO THE EXERCISES

j. $(A+B) \uparrow (1/4)$
 or
 $(A+B) \uparrow .25$
 or
 $\uparrow \uparrow (A+B)$

k. EXP (A+B)



l. SIN (-30)

m. SIN (XX) or SIN (X↑2)

n. (SIN X)↑2 or (SIN X)(SIN X)

o. SIN SIN X

2. a. Minus must not be the next item after a function – use parentheses.
- b. Missing right parenthesis; minus and divide side by side.
- c. Parentheses imbalance.
- d. R2 cannot go into 5Y; it should be R2+Y5÷Z.
- e. E must be followed by at least one digit.
- f. The number following E cannot have a decimal point.
- g. A number cannot contain a comma.
- h. An exponent cannot contain register names.
- i. The inner-most parentheses are switched.

3. The mnemonic E results from pressing , not .

4. 1.4

5. $-.000000000703$, or $-7.03E-10$. No, the result of executing the line is zero. Yes, $-7.03E-10$ is well within the range of the machine. The correct answer is not achieved because the difference between the two numbers does not occur until the sixteenth significant digit – and calculations are only performed to 12.

6. The number in R24 is 24.9999999997, which is displayed as 25 because of round-off. However, in R register designation, the digits to the right of the decimal point are dropped. Thus, R24.9999999997 is simply R24.

7. The subtraction introduces a loss of significance which is cumulative. (The multiplication by 10 only adjusts the exponent, not the sequence of significant digits).

$$x_1 = \left\{ \begin{array}{l} 1.1111111111 \\ -1 \\ \hline 0.1111111111 = 1.11111111110E-01 \\ \times 10 \\ \hline 1.11111111110 \\ -1 \\ \hline 0.11111111110 = 1.111111111100E-01 \\ \times 10 \\ \hline 1.11111111100 \\ -1 \\ \hline 0.11111111100 = 1.111111111000E-01 \\ \times 10 \\ \hline 1.11111111000 \end{array} \right.$$

As can be seen from the above x_1 and x_2 are not equal to $1.1111111111E-01$ (which is 1/9 to 12 significant digits).

CHAPTER 5

1. a. An absolute GO TO statement cannot contain a variable.
- b. Missing quote marks after "START.
- c. A PRINT statement must have at least one parameter, or the printed results will be unpredictable. The comma should be a semicolon.
- d. The first item to the immediate right of an assignment instruction must be a real variable.
- e. The left-most comma should not be there at all, while the semicolon should be a comma.
- f. The minus three should be in parentheses, and the parameter of a GO TO statement cannot contain a decimal point.

◆◆◆◆◆ ANSWERS TO THE EXERCISES ◆◆◆◆◆

- g. There is no point in writing anything to the right of a JUMP statement.
- h. The semicolon should be a comma.
- i. No GO TO or GO TO SUB statement preceding the LOAD statement.
- j. The only way the value of a flag can be changed is through the use of CLEAR FLAG, SET FLAG and END statements.
2. The largest is 9999999999.99, displayed in fixed 0. The smallest is -9999999999.99, displayed in fixed 0. The closest to zero is either of E-99 or -E-99, in any fixed point setting.

The largest number that can be displayed under fixed 5 is 99999.9999999.

3. (A=B) (B=C), but not (A=B) = (B=C),
A=B (B=C), or A=B=C.
4. (2A+B) (Y>100) + (2A-B) (Y≤100) + X
or
2A-B+X; IF Y>100; 2A+B+X
or
2A+B-2B (Y≤E2) + X

5. ENT A+B+C
6. 3: C+1+C; IF 25>C; GTO -0F
4: GTO 10F
3: JMP 7 ((C+1+C)=25)F

7. RR5→R(2B)

8. Because of the implied store in Z, the relational value is stored into Z each time the line is executed. When Z is zero, that value is 1, which is stored in place of the zero. But now Z no longer equals zero, and a 0 replaces the 1, and the process repeats itself.

9. a. The top statement means 'display the value obtained from adding the value of

flag A to 1, and meanwhile, store that sum in A.'

The bottom statement means 'display the value of the flag denoted by A once A has been incremented by 1.'

The top statement will not turn into the bottom one when it is stored.

- b. There is no difference. In each case the meaning is:

$$X + (RA) * B + Z$$

- c. LOG 10↑2 = 1

$$\text{LOG TN} \uparrow 2 = 2$$

The reason for the difference is hierarchy. The top line contains a function and an exponentiation. The bottom line is the composition of two functions.

$$\text{LOG } (10 \uparrow 2) = 2$$

10. JMP -5 (X≤25) - 8 (X≤50) (X>25)
+ 3 (X>50)

or

$$\text{JMP } -5 - 3 (X > 25) + 11 (> 50)$$

11. The way the program is arranged, it is possible to get to line 5 from line 4, and thus encounter a RETURN statement while not actually executing line 5 as a subroutine. When the RET is executed, the Calculator tries to go back to the line after the one from which it came (via GSB). Only now there was no such branch via GSB.

12. a. one

b. one

c. zero

d. zero

e. one

◆◆◆◆◆ THE DIAGNOSTIC NOTES ◆◆◆◆◆

Table I-1 is a comprehensive list of the meanings of the diagnostic NOTE's for the basic Model 20 Calculator. Within practical limits, Table I-1 attempts to indicate as many of the distinct causes for each NOTE as possible.

The discovery of an error during execution does not immediately interrupt the execution of the line. The offending operation is either simply not performed, or an alternate operation is performed in its place. The NOTE will appear in the display when the line is eventually terminated.

Only one NOTE can appear in the display at one time. If a statement contains two or more distinct errors, it is difficult to predict which one will be indicated by the NOTE.

If a line has two or more statements each of which contain errors, the NOTE that appears in the display will be the one for the last statement which had an error.

Table I-1. Diagnostic Notes.

INDICATION	MEANING
NOTE 01	<p>(NOTE 01 results from general syntax errors, however, certain specific syntax errors have their own notes.)</p> <ol style="list-style-type: none"> a. Missing ; between statements (usually results in one of the other conditions causing NOTE 01). b. Attempt to begin an expression with improper character, e.g., *, /,), ÷, =, ≠, etc. c. Two instructions side by side. d. Missing operand (resulting in two side by side operators), or, an argument of a function is missing. f. Numeric constant contains a comma, extra decimal point, or quote mark, etc. g. Exponent contains an improper character, e.g., decimal point, parenthesis, variable, * or /, etc. h. Exponent contains no digit. i. Function immediately followed by any operator except +. j. Empty parentheses, i.e., (). k. R- l. Variable on the right of ÷ is missing or not completely specified. m. ÷ immediately followed by -, *, /, %, etc. n. Missing quote mark. o. GTO or GSB followed by anything except a literal or a signed or unsigned series of digits. Decimal points, parenthesis, variables names, etc., are not allowed. p. Key is not permitted during the Enter Mode.

◆◆◆◆◆◆◆◆◆◆ **THE DIAGNOSTIC NOTES** ◆◆◆◆◆◆◆◆◆◆

Table I-1. Diagnostic Notes. (cont'd)

INDICATION	MEANING
NOTE 12	<ul style="list-style-type: none"> a. Attempt to store a line when there is insufficient memory to accommodate it. <ul style="list-style-type: none"> i. Current program is simply too long. ii. An old (and probably long) program is still in memory, and while the new program is being manually keyed in, the remnant of the old one is still being carried — using up memory. To salvage the situation, press CLEAR, END, STORE, DELETE, and continue storing the program. b. Information on magnetic cards requires more memory than is available. c. No GTO or GSB preceding LOD when loading information under the control of a program. d. GTO or GSB preceding LOD has parameter of illogical value.
NOTE 13	Attempt to record a protected side of a magnetic card.
NOTE 14	Magnetic card reader operation is not completed; press EXECUTE and insert the next side.
NOTE 15	Configuration code error — the plug-in ROM's are not arranged in a suitable manner for a program being loaded from a magnetic card.
NOTE 16	Attempted printer operation while the printer is out of paper.

◆◆◆◆◆◆◆◆◆◆ **MODEL 20 INTERNAL STRUCTURE** ◆◆◆◆◆◆◆◆◆◆

This section is a brief sketch of the internal structure of the Model 20. The intent is to answer some of the more frequently asked questions about what happens when certain keys are pressed, and especially, what the display represents.

The account is based on the diagram in Figure I-1 on page A-9. The diagram is more or less an accurate reflection of reality, although certain liberties have been taken in order to illustrate the concepts involved.

THE KEYBOARD

As each key is pressed, the keyboard generates a keycode that represents that key. Keycodes are eight digit binary-coded-octal numbers.

Most keys represent a symbol or mnemonic that is to be placed in the display. The keycodes for these keys are stored in the left-hand side of the instruction buffer, to be acted on later. Keys that do not place a symbol or mnemonic in the display are those keys concerned with editing. Those keys

are detected and acted upon without their keycodes being placed in the instruction buffer.

THE INSTRUCTION BUFFER

The left-hand side of the instruction buffer holds (in left-to-right order) the 'source information' that is to be dealt with. This information (which is a line) comes either from the keyboard, as just described, or, from the uncompiler, which reconstructs earlier information which has since been stored with the STORE key.

MODEL 20 INTERNAL STRUCTURE

The right-hand side of the instruction buffer holds compiled information. This is discussed under "THE COMPILER".

THE DISPLAY

The display shows the line in the left-hand side of the instruction buffer, or, the contents of the result register.

An important thing to remember about the display is that *the display is simply a means to make visible the information supplied to it – it has no memory of its own.*

The information presented by the display is controlled by the setting of 'switch' (3). It really isn't a switch in the physical sense, although it functions almost as if it were. The setting of this switch is controlled by the Calculator itself, as it performs its internal activities; the user does not have direct control of its setting.

THE COMPILER

The Model 20 has an interpretive compiler. It rearranges and supplements the incoming information of the left-hand side of the instruction buffer. The compiled information is placed in the right-hand side of the instruction buffer (in right-to-left order).

The compiled line has been rearranged, such that the arithmetic unit can execute it as a series of operations from right to left; the arithmetic unit can then perform each operation without having to know anything about any other operations that might be further to the left in the compiled line.

The compiled line is formed as the original line is keyed in. The compiled line and the original start from opposite ends of the instruction buffer and work towards the center. NOTE 09 results if the two ever overlap each other. Due to the nature of the compilation process, the relationship between the length of the original line and the length of its compiled counterpart is not constant. This is why maximum line lengths are not predictable.

The compiler detects any syntax errors in the line, and sends error information to the note generator.

THE ARITHMETIC UNIT

The arithmetic unit performs the actual activity specified by the line. It sequentially executes the operation codes of the line, and places the result in the result register; it also sends any information about errors during execution to the note generator.

The compiled line that the arithmetic unit executes can come either from the right-hand side of the instruction buffer, or from the User's RWM.

THE NOTE GENERATOR

The note generator determines which NOTE is to appear in the display. If it is an error during execution, it also determines the line number to be indicated. This information is then placed in the result register so that it can be displayed.

THE RESULT REGISTER

The result register is used to contain the answer produced by the arithmetic unit or the diagnostic information produced by the note generator.

EXECUTE

Pressing EXECUTE connects the compiled line to the arithmetic unit. Then the contents of the result register are displayed.

The compiled line remains in the right-hand side of the instruction buffer. Nothing changes it. This is why a line can be repeatedly executed, or executed and then stored, or vice versa, even though it is no longer visible in the display. (However, an 'escapement mechanism' prevents a line from being repeatedly stored by a series of consecutive STORE's.)

STORE

Pressing STORE places the compiled line into memory as the next line, beginning at the place in memory specified by the program line counter. The line number is not stored as part of the line. Instead, there is a special symbol to mark the start of each line, as well as symbols to identify the boundaries of the programming area.

MODEL 20 INTERNAL STRUCTURE

STORE (cont'd)

This explains why GO TO and GO TO SUB statements work (in the slow speed mode) by counting, and why line numbers are not part of a record of programming. Line numbers simply aren't stored as part of the line.

(Incidentally, a GO TO or GTO SUB statement generally occupies slightly more memory than a JUMP statement. The compiler leaves extra room in the compiled statement for the internal address of the destination — for high speed branching.)

When a compiled line is stored, the compiled information is also sent to the uncompiler.

THE UNCOMPILER

The uncompiler reconstructs the original of a compiled line, and adds the line number information; that is, the uncompiler makes a replica of the line.

The replica is sent into the left-hand portion of the instruction buffer, where it is made visible by the display mechanism.

The reason the Model 20 has both a compiler and an uncompiler is for speed, convenience, and economy. In order to keep computational speed high, no original lines should be permanently stored in memory — each line should already be compiled before it is time for program execution to begin. However, to compile an entire program at once requires a great deal of memory. Hence, the Model 20 compiles a line at a time as the lines are stored. But this means the original is lost, once the next line is keyed in. As a consequence, editing operations use the uncompiler to reconstruct an original line. Of course, extra parentheses which were removed by the compiler cannot be reconstructed by the uncompiler — it never knows they were there.

RECALL

Pressing RECALL sends a duplicate of a compiled line, identified by the program line pointer, to the uncompiler. There it is made into a replica that can be displayed, and if necessary edited and restored.

RUN PROGRAM

Pressing RUN PROGRAM establishes a mode of operation during which compiled lines in the program memory are automatically executed, in order, by the arithmetic unit. The arithmetic unit has control of the program line counter, so that it can perform any branching instructions it encounters during its execution of a line.

OTHER OPERATIONS

Certain other operations can be understood, or at least partially explained, on the basis of Figure I-1. For instance, BACK and FORWARD can be thought of as controlling pointers ① and ②. Branching statements control the setting of the program line pointer, ④. It is possible to explain why the program line pointer cannot be set to a line number higher than one greater than the highest existing line number: that place cannot be explicitly identified unless the line ahead of it is in existence.

A final word about the pointers ①, ②, and ④. They are simply locations in memory whose contents are the internal address of the part of memory being 'pointed' at.

MODEL 20 INTERNAL STRUCTURE

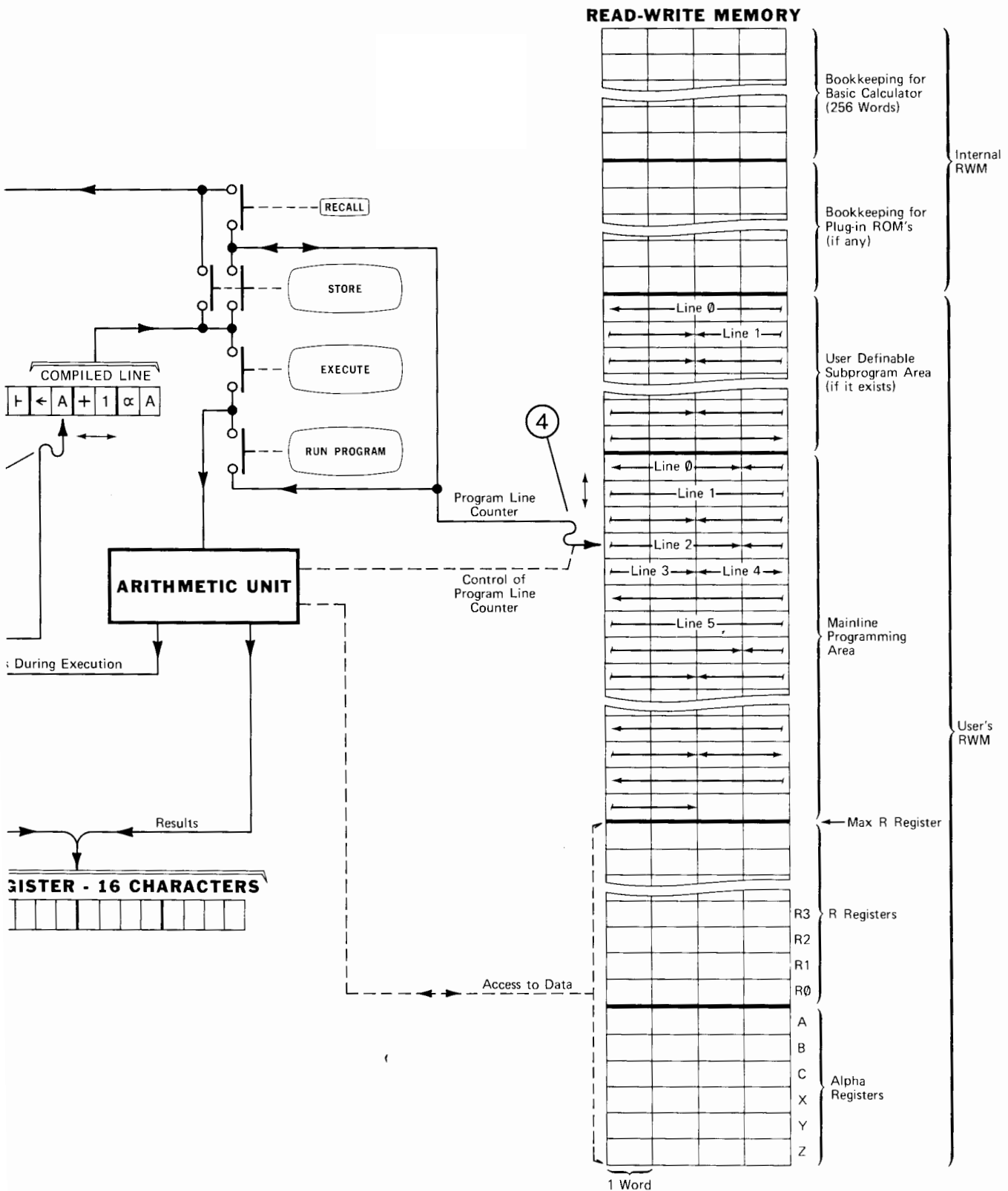
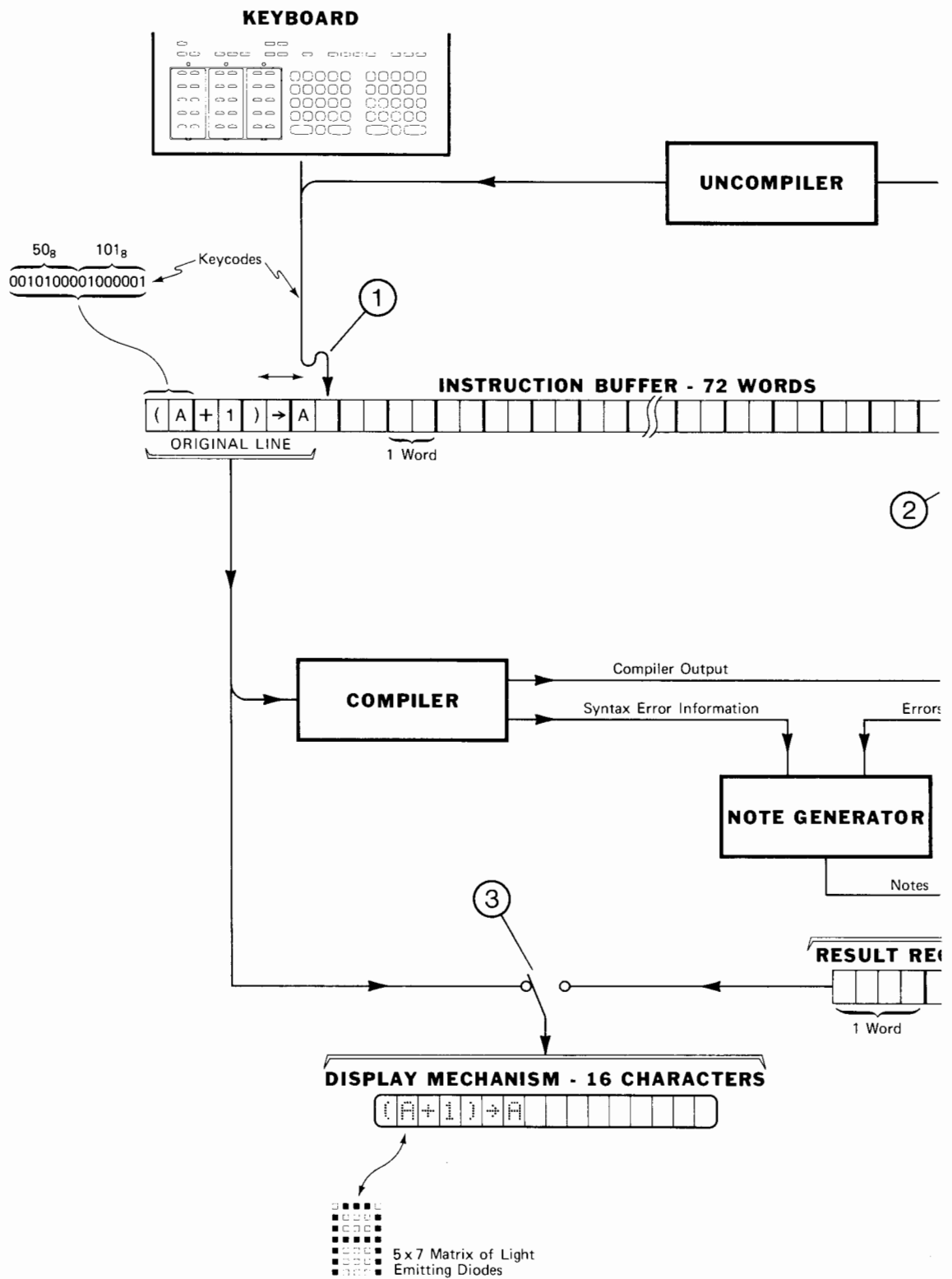


Figure I-1. The Internal Structure of the Model 20.



IDIOSYNCRASIES

The Model 20 has a few operating characteristics that differ from what you would normally expect. In some instances, an illogical or prohibited operation will produce an unusual result, rather than a NOTE, or, in addition to a NOTE.

- a. An attempt to store or execute a line that is too long may place strange characters in the display, the most usual being GSB L↑H →* ... Press CLEAR and write a shortened version of the line.
- b. Subtracting a number from itself, when the number has an absolute value of less than E-87, will result in NOTE 10 if flag 14 is a 0. This will not occur, and the correct answer is obtained if flag 14 is set to a 1.
- c. Unpredictable things can occur while storing or executing lines containing floating point constants whose exponents contain more than two significant digits. Most of the time this will result in NOTE 06 as an error during execution, but under the proper circumstances it can erase the memory.
- d. If, while in the Trace Mode, you execute a line from the keyboard, then 'back' into the line and shorten it, and execute it again, the line will be executed correctly, but printout will contain the remnant of the previous line. You can avoid this by pressing CLEAR, and then rewriting the line.
- e. An attempt to designate a (definitely non-existent) R register in the range 'R8591' to 'R16784' may not produce the customary

NOTE, and the 'contents' of such a non-existent register are unpredictable.

- f. Don't attach significance to the sign of a fixed point number printed or displayed as zero. If, when printing or displaying in fixed point, a number is close enough to zero to appear as all zeros (.000...), the sign attached to the number will not necessarily be correct; it will be the same as the sign of the previously displayed or printed number.
- g. Unexpected displays can result from executing FIXED and FLOAT statements if:
 - i. The previously executed statement was a DISPLAY or PRINT statement, and,
 - ii. The last parameter in the list for that previous statement was a computed value, such as A+B.

Under these circumstances, a statement like FXD 5 would result in a display of the number 5, regardless of what was already in the display.

- h. Some operations require a noticeable amount of time when they are executed. For instance, executing an END statement when the entire memory is filled with mainline programming may take approximately one or two seconds, depending upon the size of the memory. The same thing applies to STORE. While the display is blank during these type of activities, do not press other keys, especially STORE and EXECUTE. The result can be unpredictable.

THE PROGRAMMING FORM

Figure I-2 is an illustration of the Model 20 programming form. The form is a convenient work sheet during the writing of a program. Also, many people find it easier to key in a program from a programming form than from a printed listing.

The back of the form (not shown in the illustration) is a 'map' of the memory — use it to record the significance or purpose of each of the various registers used for storage.

If you wish, ignore the boxes into which each line is divided. They are handy when planning the appearance of a literal, however.

The shaded area on the right-hand part of the form is to remind you of the considerations surrounding maximum line length.

When filling out a programming form, it is an excellent practice to distinguish between "O's" and "zeroes", and between "I's" and "ones". See Figure I-2. Also, notice the use of the symbol δ to denote a use of the SPACE half-key.

RECOMMENDED ROM CONFIGURATIONS

For the sake of compatibility between pre-recorded programs, certain configurations are, or will be, recommended for the installation of plug-in ROM's.

As this is written, it is recommended that each of the User Definable Functions ROM, the Mathematics ROM, and Peripheral Control I, if present, be installed in the slot indicated by Figure I-3.

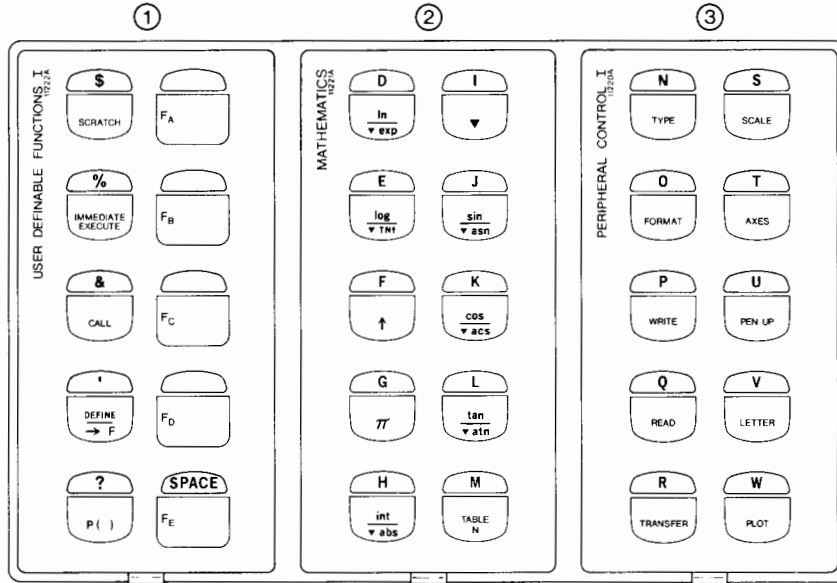
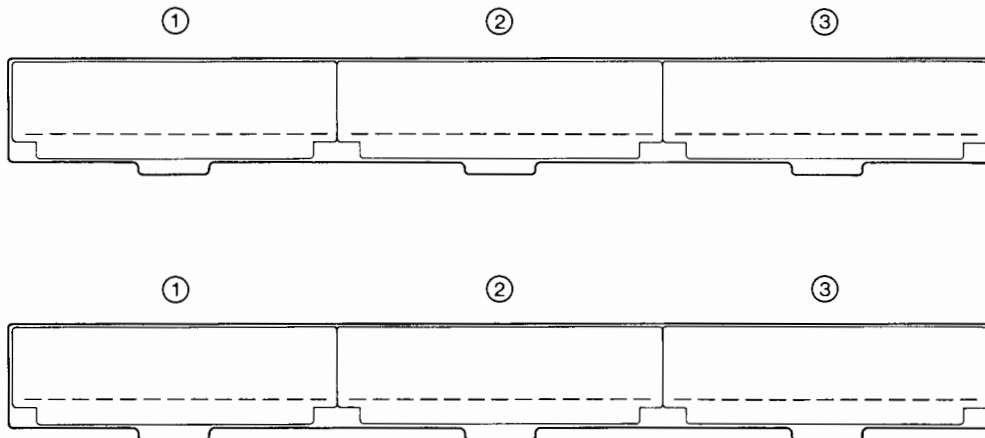
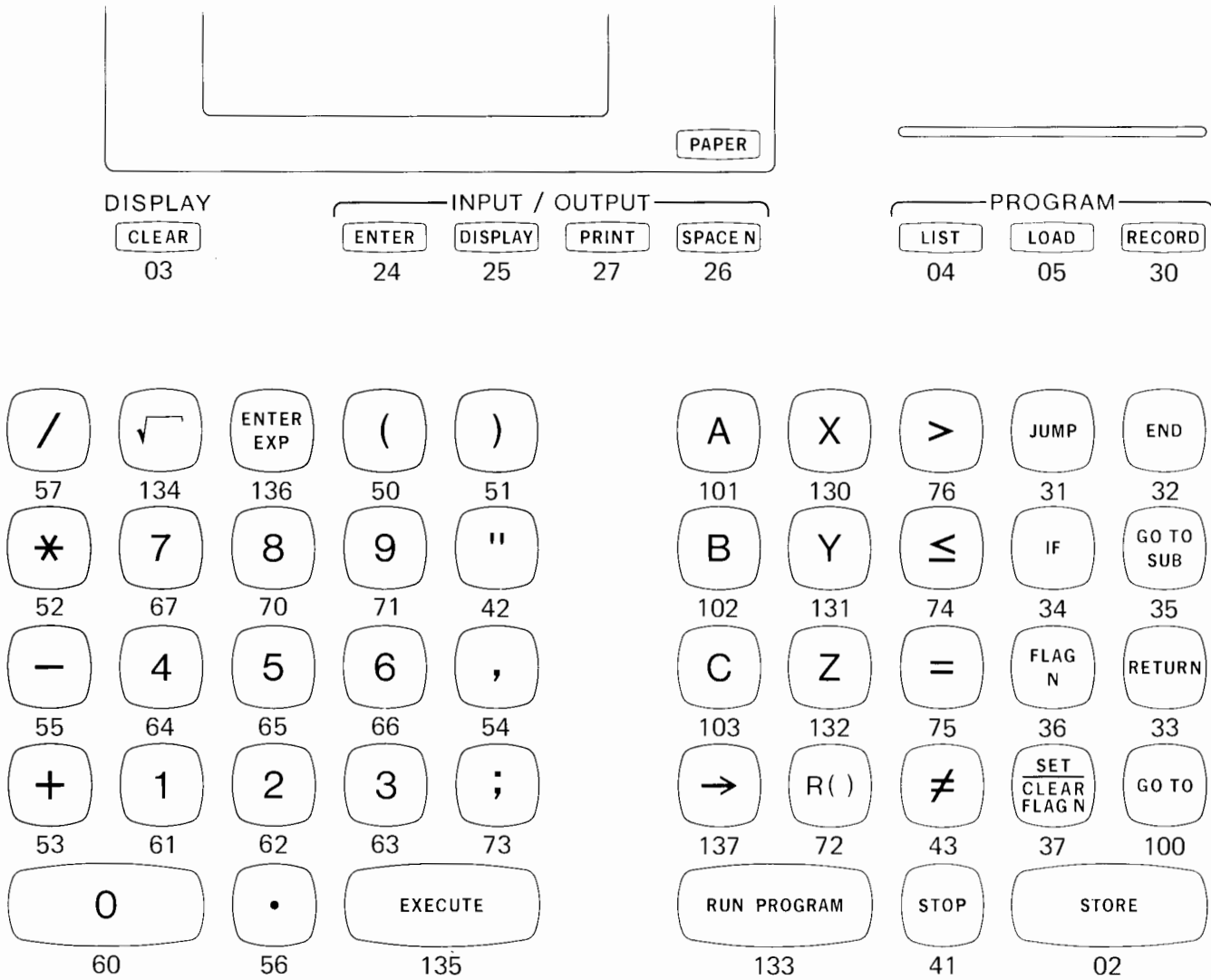


Figure I-3. The Recommended Configuration for ROM's.

The following figures can be filled out to record new recommended configurations, or other configurations that play an important role in your use of the Calculator.



MODEL 20 KEYBOARD



NOTES

1. The numbers under the keys are keycodes.
2. PAPER is mechanical, and does not have a keycode.
3. The key codes for the half-keys are always the same – regardless of which ROM's are installed.

Figure I-4. The Model 20 Keyboard.

MEMORY

ERASE
145

DECIMAL
FIXED N 22 FLOAT N 23

EDIT

NORMAL
20

TRACE
21

DELETE
141

INSERT
142

RECALL
140

BACK
144

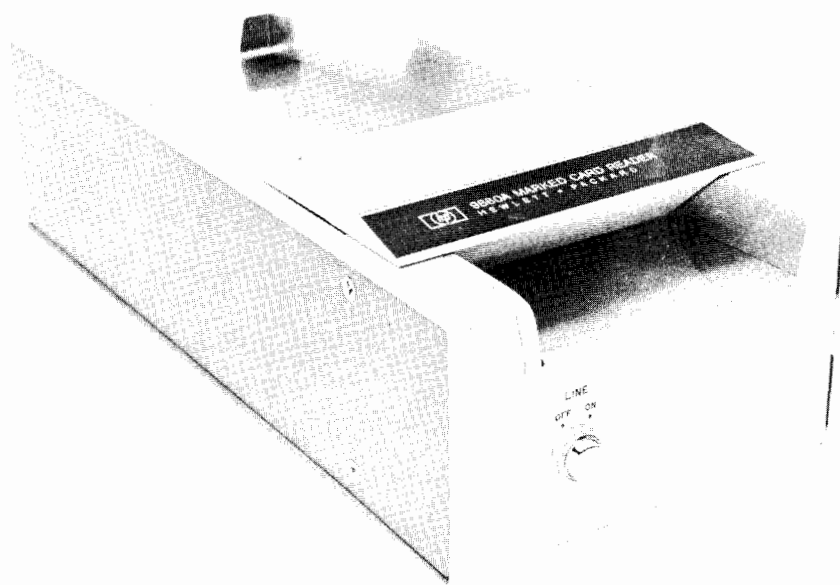
FORWARD
143

①

②

③

\$		D	I	N	S
44	10	104	111	116	123
%		E	J	O	T
45	11	105	112	117	124
&		F	K	P	U
46	16	106	113	120	125
'		G	L	Q	V
47	17	107	114	121	126
?	SPACE	H	M	R	W
77	40	110	115	122	127



**HEWLETT-PACKARD
MODEL 9860A MARKED CARD READER**

GENERAL INFORMATION

DESCRIPTION

The Hewlett-Packard Model 9860A Marked Card Reader, herein called the Model 60, or simply the 'Reader', provides a means of executing or storing lines that have been encoded on special cards. These cards are fed through the Model 60, which, using an optical technique, senses the various combinations of marks on the card. There is a combination of marks to represent each key on the Model 20 keyboard, and as a combination is detected, it is as if the associated key has been pressed. A card is encoded to represent a series of keys by marking various combinations of boxes on the card with an ordinary black lead pencil.

No peripheral control ROM is required to operate the Model 60 in a Model 20 Calculator System.

The accessories and equipment supplied with the Model 60 are listed in Table II-1.

WARRANTY

Your Model 60 was carefully inspected, both mechanically and electrically, before shipment. It should be free of mars or scratches and in perfect electrical order upon receipt. Carefully inspect the Reader for damage caused in transit and check the material listed in Table II-1. Also, check the electrical performance of the Reader by performing the electrical inspection procedure described in Supplement A of the 9820A CALCULATOR SYSTEM ELECTRICAL INSPECTION BOOKLET. (Supplement A is supplied with the Reader, and should be installed between pages 36 and *i* of the Inspection Booklet.)

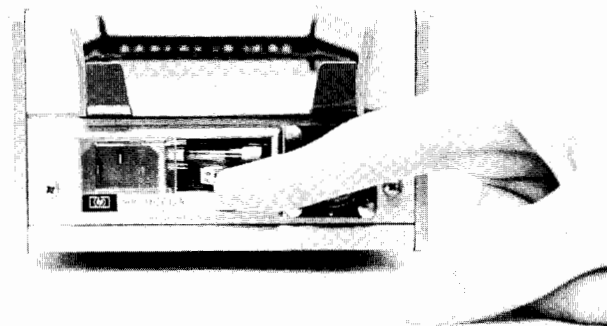
Service contracts are available for your Reader. Contact your local Hewlett-Packard Sales and Service Office for further information.

INSTALLATION

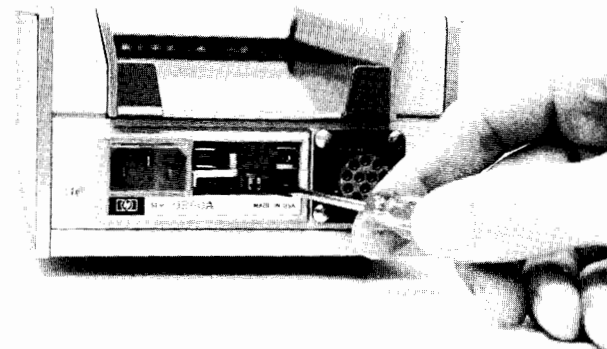
With the power cord removed from the Model 9860A, look through the clear plastic window of the power module located on the rear of the Reader. The small arrow on the switch should point to the line voltage (120 or 240) which will be used. If the switch is not set to the correct voltage, remove the fuse (see Figure II-1) and position the switch so that the arrow points to the correct voltage to be used. Replace the fuse and connect the power cord to the rear of the Reader.

CAUTION

DO NOT APPLY OPERATING POWER TO THE MODEL 9860A MARKED CARD READER UNLESS THE LINE VOLTAGE SWITCH ON THE REAR PANEL IS IN THE PROPER POSITION, OTHERWISE, DAMAGE TO THE POWER TRANSFORMER MAY RESULT.



- A. Remove the power cord and slide the window over. Then pull the lever to release the fuse.



← 240V / 120V →

- B. Set the Line Voltage Switch to the proper position.

Figure II-1. Setting the Line Voltage Switch.

The 9860A can be rewired to operate on either 100V or 220V (+5% -10%) line voltages. For further information, contact your nearest Hewlett-Packard Sales and Service Office.

GENERAL INFORMATION

Table II-1. Accessories and Equipment Supplied with the Model 60 Card Reader.

QTY.	ITEM	PART NO.
1	Interface Cable Assembly	11200A
1	AC Power Cord	8120-1575
1	Spare Lamp*	09160-67901
1	Supplement A**	09820-90050
50	Model 20 Program and Data Card	9320-2885
2	Operating Manual†	09860-90001
1	Diagnostic Card†	09860-90002
25	Program Card†	9320-2085
25	Data Card†	9320-2088

*Located in the instrument, under the right side cover. See "CHANGING THE LAMP".

**To be added to the Model 20 ELECTRICAL INSPECTION BOOKLET.

†This material is for use in Model 10 Calculator Systems. In general, it is not applicable to a Model 20 System.

To protect operating personnel, the NATIONAL ELECTRICAL MANUFACTURER'S ASSOCIATION (NEMA) recommends that the Marked Card Reader cabinet be grounded. The Marked Card Reader is equipped with a three-conductor power

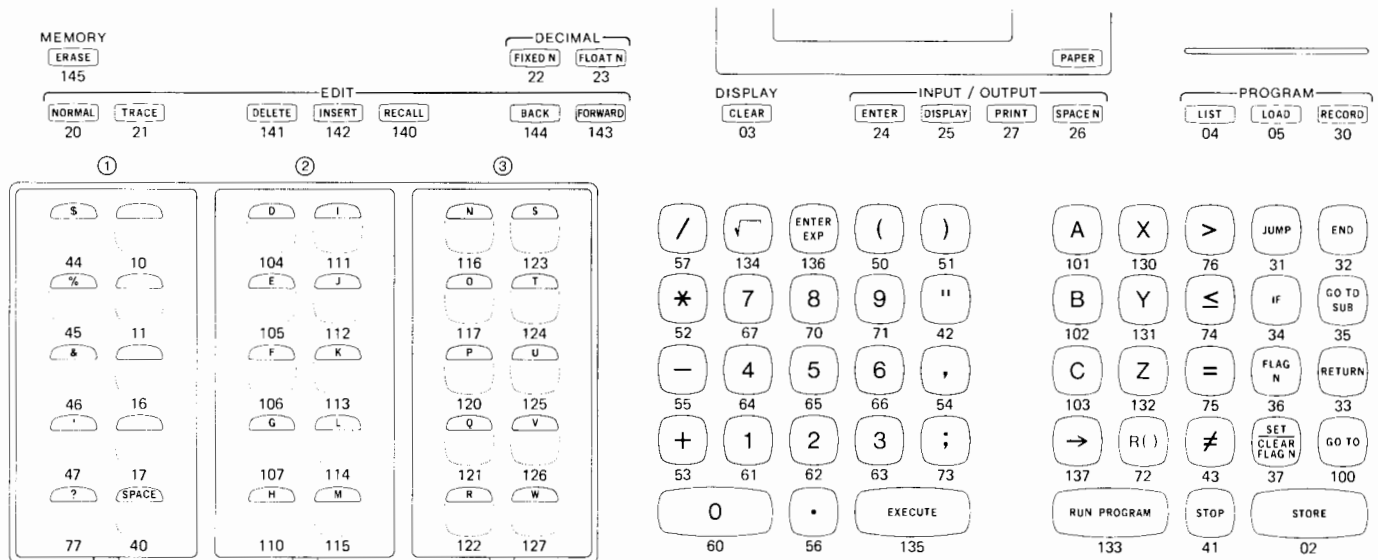
cable which, when plugged into an appropriate receptacle, grounds the cabinet of the Marked Card Reader. The center pin on the power cable's three-pronged connector is the ground connection.

Connect the I/O Cable to the rear of the Reader by aligning the keys of the I/O Cable connector with the keyways on the instrument's connector. Then, rotate the knurled barrel on the I/O connector clockwise. This will draw the two connectors together.

Turn the calculator OFF and insert the I/O Card into the rear of the Calculator. The card is keyed and cannot be inserted incorrectly. It may be inserted into any one of the four I/O connectors on the rear of the Calculator. Connect the Reader's power cord to one of the power outlets on the rear of the Calculator.

The Model 60 is switched on by pressing the button on the front of the instrument. When power is applied, the button will light. Power is removed by pressing the button again.

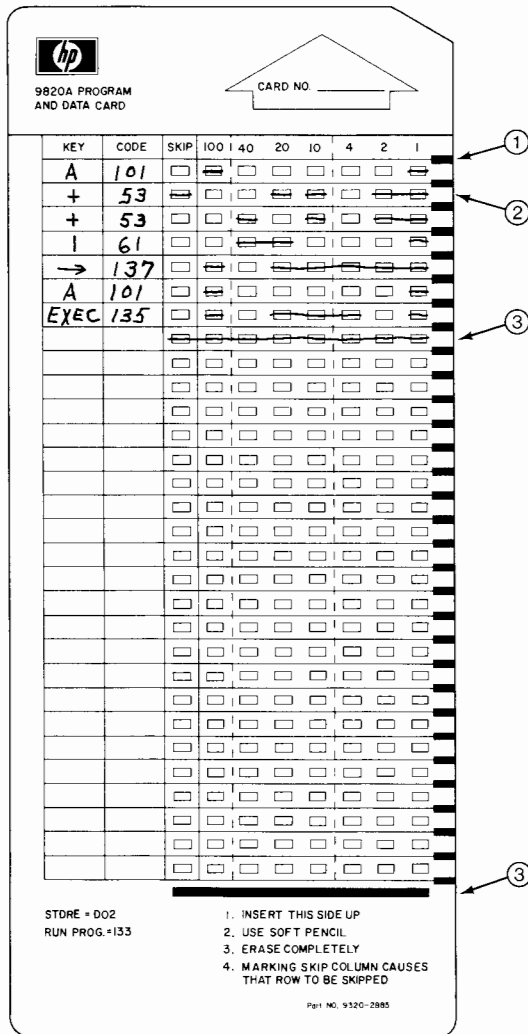
MARKING MODEL 60 CARDS



1. The numbers under the keys are keycodes.
2. PAPER is mechanical, and does not have a keycode.
3. The key codes for the half-keys are always the same - regardless of which ROM's are installed.

Figure II-2. Model 9820A Key-Codes

MARKING MODEL 60 CARDS



- ① Strobe marks.
- ② SKIP is marked to cover an error.
- ③ "SKIP 177's" turn the Reader "off". SKIP 177 is not a key-code — it is an instruction to the Model 60 only.

Figure II-3. The 9820A Program and Data Card for the Model 60.

The two columns of the card marked KEY and CODE are used to record an instruction (i.e. RUN PROGRAM) and its key-code (i.e. 133). See Figures II-2 and II-3.

The columns marked 100, 40, 20, 10, 4, 2 and 1 are for marking the key-code for an instruction. These boxes are marked so that the value of the boxes marked on a row, when totaled, equals the key-code of the desired instruction. For example,

if the instruction was EXECUTE (key-code 135), you would mark the 100, 20, 10, 4 and 1 boxes in that row (100 + 20 + 10 + 4 + 1 = 135). Figure II-2 shows the key-codes for your Model 9820A Calculator.

The SKIP column, if marked, will cause the Model 60 to ignore any boxes that are marked on that row. Marking the SKIP column can be used as an alternative to erasing; should you make an error in marking a row, you may either erase the error and correct it, or mark the SKIP column.

Near the bottom of the card, there is a pre-printed mark that intersects all of the channels on the card. This mark, interpreted by the Reader as SKIP 177, causes the Reader to stop reading the card. When a SKIP 177 is seen by the Reader, no information beyond that point on the card is transferred to the Calculator. This prevents the possibility of any marks present near the bottom of the card (e.g., program remarks, etc.) being interpreted by the Reader as information to be transferred to the Calculator.

Always use a blunt, soft-graphite pencil to mark a card. If the pencil is sharp, you may gouge the surface of the card, making any required corrections impossible. A felt tip or ball point pen should never be used; marks made with some of these devices dry with a shiny surface. The photo cells in the Reader respond only to a dull finish.

It is not necessary for you to completely darken an entire box that you wish to mark. The photo cells can 'see' a single mark as small as .5 mm (.02") in width. A single line drawn through the box with a blunt pencil will be seen. Be sure, however, that the line extends completely through the box and slightly extends through the vertical sides of the box.

If you wish to mark two or more adjacent boxes on the same row, simply draw one continuous line through the boxes.

Never mark between the STROBE marks along the edge of the card, because these marks will be read as STROBE marks and will cause incorrect operation.

MARKING MODEL 60 CARDS

Errors may be corrected by completely erasing incorrect marks. Use a soft eraser. Hard ink erasers may roughen the surface of the card. A rough card surface will reflect less light than a smooth card surface and may be seen by the photo cells as a mark.

Never use a card that has become soiled, curled, or creased. The photo cells are sensitive and could interpret a smudge or crease as a mark.

USING THE READER

WHAT TO PUT ON THE CARD

Think of the card as a means to automatically cause a series of keys to be pressed. The marks on the card determine which keys. Considered in this way, you can see that you can do many things through the Model 60 that you can do at the keyboard.

There is one very important thing to keep in mind, however. Some of the keys on the keyboard do not simply put a mnemonic into the display, but cause immediate activity of some other type (for instance, BACK, or RECALL). These keys can take more time to perform than there is between the reading of consecutive rows of boxes on the card. If a new key-code is given to the Model 20 before it has had time to finish completely with the previous key-code, the results can be unpredictable, even though a resulting display may seem correct. The keys that have execution times long enough to cause this problem are:

ERASE	FORWARD
DELETE	LIST
INSERT	EXECUTE
RECALL	RUN PROGRAM
BACK	STORE

For this reason, the Model 60 is intended primarily to enter lines and then do one of these things: EXECUTE, STORE, or RUN PROGRAM. We *strongly* recommend that no more than one of the keys in the list above be encoded on a card, and that such a key be the last key-code encoded on the card.

It is a good practice for the last mark encoded on the card to be 'SKIP 177', see Figure II-3. This will help prevent stray marks on the card from causing erroneous entries.

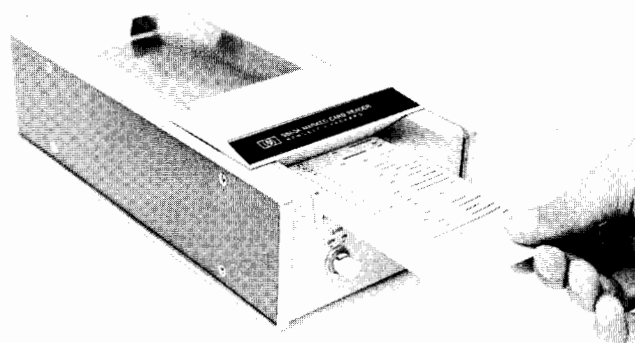


Figure II-4. Inserting a Card into the Reader.

If a line is longer than can be encoded on a single card, simply encode the first part of the line on one card, and the second part on another. Neither card needs any special encoding; remember: "the card is simply a series of keystrokes". Insert the cards, one after the other, in the obvious order.

NOTE

Encoding an undefined key-code (i.e., one that does not correspond to a key) can result in unpredictable results.

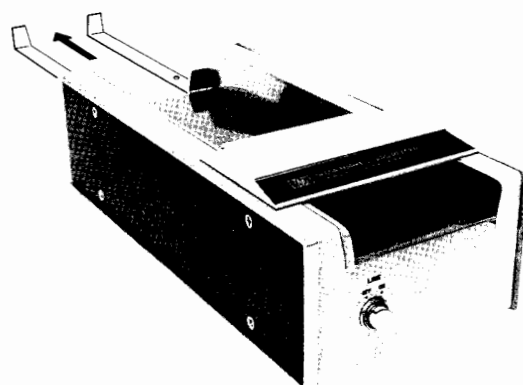


Figure II-5. The Card Stop in its Extended Position.

USING THE READER

To accommodate long cards, the Reader is equipped with an adjustable card stop. When the card stop is placed in its extended position, the Reader will accept a longer card. To extend the card stop, simply pull the stop into its extended position.

THE DATA CARD

Figure II-6 shows a special data card for entering data during programmed ENTER statements. The last key-code on the card is RUN PROGRAM.

The format of the card is:

*Items in **BOLD** are pre-marked on the card. Only the digits need to be filled in.

Minus
Minus
 Digit
 Digit
 .
 .
Decimal Point
 Digit
 Digit
 .
 .
Enter Exponent
Minus
Back
Skip
Skip
Skip
Skip
 Digit
 Digit
Run Program

The entire number is made negative by marking what is really the SKIP channel for the second Minus at the top of the card. If the number is to be entered as a fixed point number, simply leave all the exponent information blank. This causes the exponent to be entered as E00, which has no effect on a fixed point number.

If an exponent is to be made negative, the SKIP for the Back is marked. Normally, the Back removes the preceding Minus; by marking the SKIP, it stays. The four SKIP's following the Back are to allow time for the Back to be performed before the digits of the exponent are entered.

9820A DATA CARD
PART NO. 9320-1641

CARD NO. _____

8 4 2 1

(-) ? →

1. WRITE THE NUMBER AROUND THE EXISTING DECIMAL POINT.
2. MARK THE BOX IF THE NUMBER IS NEGATIVE.

-1
2
3
•
4
5
6

1. AN UNMARKED EXPONENT IS IGNORED.
2. MARK THE BOX IF THE EXPONENT IS NEGATIVE.

E
(-) ? →

RUN PROGRAM →

1. INSERT THIS SIDE UP
2. USE SOFT PENCIL
3. ERASE COMPLETELY
4. MARK THE BOXES SO THAT THE SUM OF THE COLUMNS EQUALS THE NUMBER.

0 =	8	4	2	1	5 =	8	4	2	1
1 =	□	□	□	□	6 =	□	□	□	□
2 =	□	□	□	□	7 =	□	□	□	□
3 =	□	□	□	□	8 =	□	□	□	□
4 =	□	□	□	□	9 =	□	□	□	□

This card has been marked to enter the number -123.456.

Figure II-6. The Model 20 Data Card, Part Number 9320-1641.

USING THE READER

HOW THE READER WORKS

Inserting a card into the Model 60 will automatically start the motor, which will draw the card through the instrument. As the card is drawn through the instrument, it is viewed, across its width, by nine light-sensitive photo cells located in the instrument's cover. As the card moves through the Model 60, each photo cell independently monitors the light reflected from its column. If a box is marked, the light reflected

from the card suddenly diminishes. The photo cell 'sees' this reduction in reflected light and remembers that a box has been marked. Thus, the card is read one frame (one row of boxes) at a time.

As the STROBE marks, located between the frames, pass under the STROBE photo cell, the Reader sends the information of the previous frame to the Calculator, provided SKIP was not marked.

CHANGING THE LAMP

Occasionally the lamp which lights the card will require replacing. A defective lamp will prevent the motor, which draws the card through the Card Reader, from operating. Always check first for a burned out lamp when cards fail to feed properly. Light from a correctly functioning lamp can be seen when looking into the card slot with the Calculator and Card Reader turned on.

If the card fails to feed and the lamp glow is not visible in the card slot, replace the lamp by following these instructions.

Access to the lamp in the card reading assembly is obtained by grasping the top of the card slot with your hand and pulling upward. The top of the Card Reader is hinged and will pivot up to reveal the lamp.

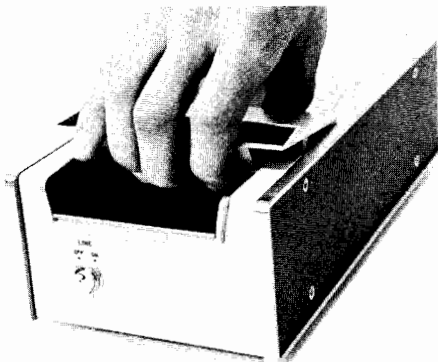


Figure II-7. Exposing the Lamp.

A replacement lamp is located inside the Card Reader behind the right side panel. Remove the sponge plastic cushion which protects the spare lamp.

CAUTION

THE LAMP IS VERY FRAGILE. ONE SIDE OF THE CUSHION HAS A SLIT CONTAINING THE LAMP.

Pull the cushion toward the front of the Card Reader at a very shallow angle. Pulling the cushion at right angles to the side frame will break the lamp.

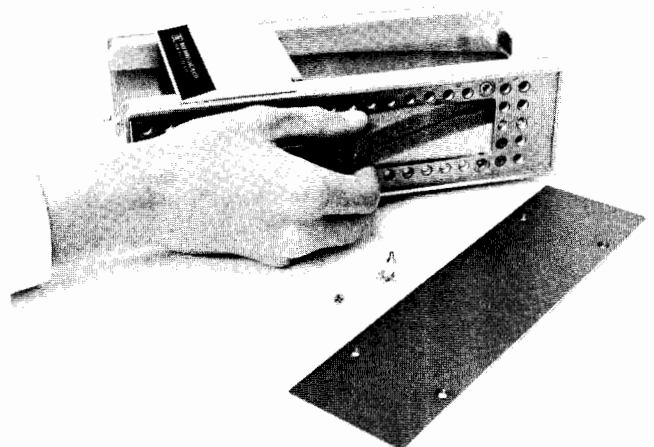


Figure II-8. Removing the Spare Lamp.

CHANGING THE LAMP

Remove the lamp from the card reader assembly. To prevent breakage, do not grasp the lamp by the glass cylinder. Instead, using a screwdriver or similar instrument, placed against the lamp's left metal cap, push the lamp toward the right until both metal caps clear their spring clamps and the lamp is released.

To install the replacement lamp, position each metal cap over its spring clamp with the small spring in the glass cylinder to your left; then, simultaneously press both metal caps into the spring clamps. Do not press against the glass cylinder; that may break the new lamp.

Return the top cover to its original position. Replace the cushion, the side cover and the four retaining screws. You will probably want to order a new lamp at this time from your local Hewlett-Packard office. Then you will always have a spare lamp on hand.



Figure II-9. Removing the Old Lamp.

After replacing the lamp run the Card Reader exerciser to verify proper operation.

INDEX

This is an index of the operating and programming information for the Model 20 Calculator. Information concerning the Model 60 Marked Card Reader (pages A14 through A21) is not included in the index. Page numbers in bold are primary references.

A

absolute go to statements, 3-7, **5-22**
accuracy, 4-4
addition, **4-4**, 4-5, 5-3
addressing, 3-9, **5-8**
algebraic language, 2-5
alpha registers, 1-2, **2-8**, 4-2
ambivalent group, 2-9
argument, **4-7**, 5-3
arithmetic impossibilities, 3-13, **4-11**
arithmetic operators, **4-4**, **4-5**, 5-3
arithmetic unit, A-7 thru A-9
assignment statements, 3-2, 4-5, 4-12, **5-16**, 5-20

B

back, 3-2 thru 3-7, 3-12, 3-13, A-10
binary operators, 5-3
blank, 2-10, 5-47
branching, 3-9, **5-8**

C

card reader
– magnetic, see magnetic card reader
– marked, see marked card reader
CFG, 2-10, **5-2**, 5-32
character, 2-0, 2-5, **2-9**
cleaning the calculator, 1-5
clear, 2-0, **2-1**, 3-1 thru 3-5, 3-12, 5-47, A-10
clear flag
– key, 2-10, A-13
– statements, 5-2, **5-32**
colon, 2-10
comma
– use, 4-1, **5-8**
– key, 2-10, A-13
compilation, 3-3, **A-7**
compiler, 3-7, **A-7**, A-9
computed go to sub, 5-46
conditional branching, 5-8, 5-34
constants, **4-1**, 4-7, 5-1
also see real constants
convenience outlets, 1-3

D

data storage, 2-8
also see assignment statements
decimal point
– use, 4-1 thru 4-4
– key, 2-10, A-13
also see fixed statements and float statements

definable function area, 2-8, A-9
definable functions, 4-7
also see user definable functions ROM
definable functions ROM, 4-7, 5-3, 5-10
delete, 3-2 thru 3-4, 3-11
diagnostic notes, 3-6, A-4
also see the various notes
display
– the, 2-1, 3-2, 3-3, 3-5, 3-7, A-7, A-9
– key, 2-10, A-13
– statements, 5-13
– reversion to floating point, 4-4, 5-13
– idiosyncrasy, A-10
divide
– key, 2-10, A-13
– operation, **4-5**, 5-3
division by zero, 3-13, **4-11**
DSP, 2-10, 5-13

E

E, 2-10, 4-1 thru 4-4
editing
– within lines, 3-3
– upon programs, 3-10, 3-11
electrical inspection, 1-2
END, 2-10, 5-36
– key, 2-10, A-13
– statements, 3-7, 3-8, 3-10, 3-13, 5-28 5-31, 5-33, **5-36**, 5-42, A-10
end-of-line (\dagger), 2-1, 3-4 thru 3-6, 3-10, 3-14
ENT, 2-10, 5-18
enter
– key, 2-10, A-13
– mode, 5-18
– statements, 2-1, 3-13, **5-18**, 5-27
also see, 2-1
enter exponent, 4-1
equal to
– key, 2-10, A-13
– use, 5-3, 5-34, 5-35
erase, see memory erase
errors
– during execution, 3-5, 3-13
– syntax, 3-5, 3-13
exclamation point, 2-10
execute, 2-0, 2-6, 3-2, **3-4**, A-7, A-10
exerciser program, 1-2
exponent, 4-1, A-10
exponentiation, 4-7, 4-11, 5-3
expression, 4-7, 5-4

F

fan filter, 1-5
 figurative syntax, 5-11
 fixed
 – point, 4-1, 4-4, 5-11
 – statements, 5-11, A-10
 fixed n key, 2-1, 2-9, 4-4, 5-11
 flag
 – key, 2-10, 5-2, A-13
 – statements, 5-32, 5-33
 flags, 3-1, 5-2, **5-32**, 5-36
 flag 0, 5-32
 flag 13, 5-2, 5-21
 flag 14, 4-11, 5-2, 5-34, A-10
 flag 15, 4-11, 5-2
 FLG, 2-10, 5-2
 float
 – point, 4-1, 4-4, 5-11
 – statements, 5-11, A-10
 float n key, 2-1, 2-9, 2-10, 4-4, 5-12, A-13
 float 9, 3-1, 5-12
 FLT, 2-10, 5-11
 forward, 3-3 thru 3-6, 3-10, 3-12, A-13
 functions, 4-1, **4-6**, 4-10, 5-3
 fuses, 1-3, 1-4
 FXD, 2-10, 5-11

G

gazinta, 5-6
 go to
 – key, 2-10, 3-7, A-13
 – statements, 5-18, 5-22, 5-23, 5-27, 5-28, 5-43, A-8
 go to sub statements, 5-10, 5-18, 5-24, 5-27, 5-28, A-8
 greater than
 – key, 2-10, A-13
 – use, 5-3
 grounding requirements, 1-3
 GSB, 2-10, 5-24
 GTO, 2-10, 5-22, 5-23
 guard digits, 4-4

H

half keys, 2-1, 2-4
 hierarchy, 4-8, 4-10
 high speed branching, 5-28, 5-29, 5-37

I

I/O card, 2-4
 I/O expander, 2-5
 idiosyncrasies, A-10
 IF, 2-10, 5-34
 if statements, 5-34
 implied multiply, **4-5**, 4-10, 5-3

implied store into Z, **4-9**, 5-17, 5-28

initialization
 – initial turn on procedure, 1-4
 – during turn on, 3-1
 – of program memory, 3-8

insert
 – ing keys, 3-4
 – ing lines, 3-11

inspection procedure, 1-2
 instruction buffer, A-6, A-7, A-9
 instructions, 5-6, 5-8

J

JMP, 2-10, 5-29
 jump statements, 5-9, 5-29, A-8
 jump to sub, 5-46

K

key codes, A-6, A-9, A-13, A-16, A-17
 keyblocks, 2-0, 2-1, 2-3
 keyboard
 – magazine, 1-2
 – drawing, 2-0, A-13
 – discussion, 2-1, A-6
 – load statements, 5-41

L

labeled go to statements, 5-23
 labels, 5-2, 5-7, 5-9
 lazy t (⌊), see end-of-line
 LED's, 2-2
 less than (does not exist), see 5-3
 less than or equal to
 – key, 2-10, A-13
 – use, 5-3
 line frequency, 1-3
 line length, 3-2, A-7
 line numbers, **2-7**, 3-7, 3-9, 3-11, 3-12, 5-8
 line voltage, 1-3
 lines, 2-5, **2-6**, 2-9, 3-1, 5-7
 linking programs, 5-43, 5-44
 list
 – (definition), see lists
 – key, 3-9, A-13
 listing programs, 2-9, 3-9
 lists, 5-8
 literal syntax, 5-11
 literals, 2-4, **5-2**, 5-6, 5-7, 5-12 thru 5-14, 5-19
 load statements, 5-41 thru 5-43
 local subroutines, **5-10**, 5-20, 5-26, 5-27, 5-35
 LOD, 5-10, 5-41 thru 5-44
 logarithms, 4-6, 4-8, 4-11

INDEX

M

magnetic card holder, 1-2
magnetic card reader, 2-2
 also see load statements and record statements
magnetic cards, 1-2, 2-3
mainline programming area, 2-8, 3-7, 3-9, A-9
marked card reader, 2-4, A-14 thru A-21
math ROM, 4-1, 4-6 thru 4-8
mathematical expressions, 5-3, 5-4
mathematical impossibilities, 3-13, 4-11
max r register, **2-8, 2-9**, 3-8, 3-10 thru 3-12, 5-37, 5-45
maximum line length, 3-2
maximum line number, **3-7**, 5-31, 5-37
mechanics group, 2-9
memory
 – size, 1-1
 – types, 2-7
 – drawings, 2-8, 3-8, 3-9, 5-37, 5-41 thru 5-43, A-9
 – map, A-10
memory erase, 2-1, **3-1**, 3-8, 3-9, 5-2
messages, 2-2
minus, **4-4, 4-5**, 5-3
mixed expressions, 5-5
mnemonics, 2-0, 2-9, 3-1, 3-2
model 60 marked card reader, A-14 thru A-21
modifying a line, 3-3, 3-10
modifying programs, 3-10 thru 3-12
multiple assignment statements, **4-5**, 4-12, 5-16
multiplication, **4-5**, 5-3

N

N max, 5-45
 also see max r register
nesting
 – parentheses, 4-8
 – subroutines, 5-26, 5-35, 5-37
normal
 – key, 2-9, A-13
 – mode, 3-1, 3-14
not equal to, 2-10, 5-3
notes, 3-1, 3-5, **3-6**, 3-13, 4-10, **A-4**
note 01, 3-6, 3-13, 4-1, 4-3, 5-22, 5-23, 5-31, A-4
note 02, 3-6, 5-12, 5-15, 5-40, A-5
note 03, 3-6, A-5
note 04, 3-6, A-5
note 05, 3-6, 4-3, 5-16, 5-32, 5-33, 5-45, A-5
note 06, 3-6, 4-6, 5-16, A-5, A-10
note 07, 3-6, 5-2, 5-36, A-5
note 08, 3-6, 5-29, 5-31, 5-46, A-5
note 09, 3-2, 3-6, 5-26, A-5, A-7
note 10, 3-6, A-5, A-10
note 11, 2-1, 3-6, 5-18, 5-20, A-5
note 12, 3-6, 3-8, 5-41, 5-43, A-6
note 13, 2-3, 3-6, 5-38, 5-40, A-6
note 14, 2-3, 3-6, 5-38, 5-40, 5-41, 5-43, 5-45, A-6

note 15, 2-4, 3-6, 5-42, 5-43, A-6
note 16, 1-5, 3-6, A-6
note generator, A-7
null program, 2-7, 3-1
number entry, 4-2
 also see enter statements
n-way branch, 5-45

O

operands, 5-3
operations (numerical), 4-1
operators, 4-4, **4-5**, 4-8, 4-10, **5-3**
options
 – 001 (memory), 1-1, 1-2, 2-7, 2-8
 – power cord, 1-2
overflow, 4-11
overlays, 2-3, 2-4, A-12

P

paper
 – button, 2-1, 5-15
 – printer, 1-1, 1-4
parameters, 5-8
parenthesis, 3-7, 4-3, 4-5, **4-8**, 4-10, 5-17, A-8
peripheral control ROM's, 2-4
peripherals, 2-4
plug-in ROM's, 1-1, 2-1, **2-3, 2-4**, 2-7, 2-9, 3-7
 – configurations, 2-4, **A-12**
 – initialization, 3-1
plus, 4-4, 4-5, 5-3
power cords, 1-2
power requirements, 1-3
pre-recorded programs, 1-2
pre-written programs, 1-2
print
 – key, 2-10, A-13
 – statements, 2-6, **5-14**
printer
 – loading, 1-4
 – paper, 1-1, 1-4
 – the, 2-2
 – window, 1-5
program, 2-7, 5-8
program execution, 2-7, **3-13**, 3-14
program line counter, **2-7**, 3-7, 3-10 thru 3-12, 5-8, 5-37, A-7 thru A-9
program storage (memory aspects) 2,8, 2-9
programming form, A-10
PRT, 2-10, 5-14

Q

qualifiers, see if statements
quantities, 4-7, 5-2
quote fields, 2-1
 also see literals

R

R (), 2-8, 2-10
 also see R registers, 4-2
 R registers, 2-8, 3-8, 3-11, 4-2, 5-45, A-10
 real constants, 4-1, 5-1, A-10
 real variables, 4-2, 5-1, 5-2
 rear panel, 1-3
 REC, 2-10, 5-37, 5-40
 recall, 3-7, 3-11 thru 3-13, A-8
 record "data" statements, 5-40
 record statements, 5-37
 recording programs, 5-37, 5-38
 records, 2-2, 5-9
 registers, 1-1, 1-2, 2-8
 also see R registers and alpha registers
 relational expressions, 5-3, 5-4
 relational operators, 4-8, 5-3
 relative addressing, 3-9, 5-9
 relative go to statements, 5-9, 5-22
 replacing fuses, 1-4
 replicas, 3-7, 3-10 thru 3-13, A-8
 result register, A-7
 RET, 5-10, 5-24
 return
 – key, 2-10, A-13
 – statements, 5-10, 5-24
 right arrow, 2-10
 also see assignment statements
 ROM
 – internal, 2-7
 – plug-in, see plug-in ROM's
 – slot, 2-3
 rounding, 5-4, 5-12, 5-35
 rules of mathematical combination, 4-10
 run program, 3-10, 3-13, 5-18, 5-35, A-8
 running a program, 3-13
 – in the trace mode, 3-14
 RWM, see memory

S

"secure" programs, 5-38
 "secure" record statements, 5-38
 segments, 3-3
 select code, 2-5
 semi-colon, 2-6, 2-10, 3-2, 5-7
 service contracts, 1-2
 set flag
 – key, 2-10, 5-2, A-13
 – statements, 5-32
 SFG, 2-10, 5-32
 sides, 2-3
 significant digits, 4-3
 slash, 2-10, 4-5
 slide switches, 1-3, 1-4
 slot, see ROM slot
 slow speed branching, 5-28

space key, 2-10, A-13
 space N
 – key, 2-1, 2-10, A-13
 – statements, 5-15
 square root, 4-6, 4-11
 stacking programs, 3-8, 3-9, 5-9, 5-41
 statements, 2-6, 5-6, 5-7
 stop
 – key, 2-9, 2-10, 3-9, 3-10, 3-13, A-13
 – statements, 3-10, 3-13, 5-27, 5-35
 STP, 2-10, 5-25, A-13
 store, 2-10, 3-4, 3-7, 3-9 thru 3-12, A-7, A-10
 storing programs
 – (with STORE etc.), 3-6, 3-8, 3-9
 – in the trace mode, 3-14
 – with mag cards, 5-41, 5-43
 subtraction, 4-4, 4-5, 5-3, A-10
 subroutines, 5-10
 supplied functions, 4-7, 5-3
 symbol, 2-0, 2-9, 3-1
 symbolic addressing, 3-9, 5-2, 5-7, 5-9
 syntax, 2-5
 also see chapter 5
 syntax errors, 3-5, 3-13
 syntax group, 2-9

T

times sign, 4-5
 also see multiplication
 trace
 – key, 2-9, 2-10, 3-19, 5-44
 – mode, 3-14, A-10
 – statements, 5-44
 TRC, 2-10, 3-14, 5-44

U

unary minus, 4-4, 4-10, 4-11
 unary operators, 5-3
 unary plus, 4-4, 4-10, 4-11
 uncompiler, A-6, A-8, A-9
 unconditional branching, 5-8
 underflow, 4-11
 up arrow, 2-10, 4-7
 user definable functions ROM, 2-4, 5-10, 5-20, 5-26, 5-33, 5-35, 5-42
 user's RWM, 2-7
 also see memory

V

values, 4-7, 5-6
 variables, 4-1, 4-2, 5-1
 volt-amps, 1-3

W

wattage, 1-3
 words, 2-9, A-9

**ELECTRONIC
SALES & SERVICE OFFICES**

UNITED STATES

ALABAMA

P.O. Box 4207
2003 Byrd Spring Road S.W.
Huntsville 35802
Tel: (205) 881-4591
TWX: 810-726-2204

ARIZONA

2336 E. Magnolia St.
Phoenix 85034
Tel: (602) 252-5061
TWX: 910-951-1390

5737 East Broadway
Tucson 85716
Tel: (602) 298-2313
TWX: 910-952-1162

CALIFORNIA

1430 East Orangethorpe Ave.
Fullerton 92631
Tel: (714) 870-1000

3939 Lankershim Boulevard
North Hollywood 91604
Tel: (213) 877-1282
TWX: 910-499-2170

1101 Embarcadero Road
Palo Alto 94303
Tel: (415) 327-6500
TWX: 910-373-1280

2220 Watt Ave.
Sacramento 95825
Tel: (916) 482-1463
TWX: 910-367-2092

9606 Aero Drive
San Diego 92123
Tel: (714) 279-3200
TWX: 910-335-2000

COLORADO

7965 East Prentice
Englewood 80110
Tel: (303) 771-3455
TWX: 910-935-0705

CONNECTICUT

12 Lunar Drive
New Haven 06525
Tel: (203) 389-6551
TWX: 710-465-2029

FLORIDA

P.O. Box 24210
2806 W. Oakland Park Blvd.
Ft. Lauderdale 33307
Tel: (305) 731-2020
TWX: 510-955-4099

P.O. Box 13910
6177 Lake Ellenor Dr.
Orlando, 32809
Tel: (305) 859-2900
TWX: 810-850-0113

GEORGIA

P.O. Box 28234
450 Interstate North
Atlanta 30328
Tel: (404) 436-6181
TWX: 810-766-4890

HAWAII

2875 So. King Street
Honolulu 96814
Tel: (808) 955-4455

ILLINOIS

5500 Howard Street
Skokie 60076
Tel: (312) 677-0400
TWX: 910-223-3613

INDIANA

3839 Meadows Drive
Indianapolis 46205
Tel: (317) 546-4891
TWX: 810-341-3263

COLORADO

7965 East Prentice
Englewood 80110
Tel: (303) 771-3455
TWX: 910-935-0705

LOUISIANA

P.O. Box 856
1942 Williams Boulevard
Kenner 70062
Tel: (504) 721-6201
TWX: 810-955-5524

MARYLAND

6707 Whitestone Road
Baltimore 21207
Tel: (301) 944-5400
TWX: 710-862-9157

P.O. Box 1648
2 Choke Cherry Road
Rockville 20850
Tel: (301) 948-6370
TWX: 810-828-9684

MASSACHUSETTS

32 Hartwell Ave.
Lexington 02173
Tel: (617) 861-8960
TWX: 710-326-6904

MICHIGAN

21840 West Nine Mile Road
Southfield 48075
Tel: (313) 353-9100
TWX: 810-224-4882

MINNESOTA

2459 University Avenue
St. Paul 55114
Tel: (612) 645-9461
TWX: 910-563-3734

MISSOURI

11131 Colorado Ave.
Kansas City 64137
Tel: (816) 763-8000
TWX: 910-771-2087

MARYLAND

148 Weldon Parkway
Maryland Heights 63043
Tel: (314) 567-1455
TWX: 910-764-0830

NEW JERSEY

W. 120 Century Road
Paramus 07652
Tel: (201) 265-5000
TWX: 710-990-4951

1060 N. Kings Highway
Cherry Hill 08034
Tel: (609) 667-4000
TWX: 710-892-4945

NEW MEXICO

P.O. Box 8366
Station C
6501 Lomas Boulevard N.E.
Albuquerque 87108
Tel: (505) 265-3713
TWX: 910-989-1665

156 Wyatt Drive
Las Cruces 88001
Tel: (505) 526-2485
TWX: 910-983-0550

NEW YORK

6 Automation Lane
Computer Park
Albany 12205
Tel: (518) 458-1550
TWX: 710-441-8270

1219 Campville Road
Endicott 13760
Tel: (607) 754-0050
TWX: 510-252-0890

82 Washington Street
Poughkeepsie 12601
Tel: (914) 454-7330
TWX: 510-248-0012

39 Saginaw Drive
Rochester 14623
Tel: (716) 473-9500
TWX: 510-253-5981

5858 East Molloy Road
Syracuse 13211
Tel: (315) 454-2486
TWX: 710-541-0482

1 Crossways Park West
Woodbury 11797
Tel: (516) 921-0300
TWX: 510-223-0811

NORTH CAROLINA

P.O. Box 5188
1923 North Main Street
High Point 27262
Tel: (919) 885-8101
TWX: 510-926-1516

OHIO

25575 Center Ridge Road
Cleveland 44145
Tel: (216) 835-0300
TWX: 810-427-9129

3460 South Dixie Drive
Dayton 45439
Tel: (513) 298-0351
TWX: 810-459-1925

1120 Morse Road
Columbus 43229
Tel: (614) 846-1300

OKLAHOMA

6301 N. Meridian Avenue
Oklahoma City 73122
Tel: (405) 721-0200
TWX: 910-830-6862

OREGON

Westhlis Mall, Suite 158
4475 S.W. Scholls Ferry Road
Portland 97225
Tel: (503) 292-9171
TWX: 910-464-6103

PENNSYLVANIA

2500 Moss Side Boulevard
Monroeville 15146
Tel: (412) 271-0724
TWX: 710-797-3650

1021 8th Avenue
King of Prussia Industrial Park
King of Prussia 19406
Tel: (215) 265-7000
TWX: 510-660-2670

RHODE ISLAND

873 Waterman Ave.
East Providence 02914
Tel: (401) 434-5335
TWX: 710-381-7573

***TENNESSEE**

Memphis
Tel: (901) 274-7472

TEXAS

P.O. Box 1270
201 E. Arapaho Rd.
Richardson 75080
Tel: (214) 231-6101
TWX: 910-867-4723

P.O. Box 27409
6300 Westpark Drive
Suite 100
Houston 77027
Tel: (713) 781-6000
TWX: 910-881-2645

231 Billy Mitchell Road
San Antonio 78226
Tel: (512) 434-4171
TWX: 910-925-5681

UTAH

2890 South Main Street
Salt Lake City 84115
Tel: (801) 487-0715
TWX: 910-925-5681

VERMONT

P.O. Box 2287
Kennedy Drive
South Burlington 05401
Tel: (802) 258-4455
TWX: 510-299-0025

VIRGINIA

P.O. Box 6514
2111 Spencer Road
Richmond 23230
Tel: (703) 258-3431
TWX: 710-956-0157

WASHINGTON

433-108th N.E.
Bellevue 98004
Tel: (206) 454-3971
TWX: 910-443-2303

***WEST VIRGINIA**

Charleston
Tel: (304) 768-1232

FOR U.S. AREAS NOT LISTED:

Contact the regional office nearest you: Atlanta, Georgia... North Hollywood, California... Paramus, New Jersey... Skokie, Illinois. Their complete addresses are listed above.
*Service Only

CANADA

ALBERTA

Hewlett-Packard (Canada) Ltd.
11748 Kingsway Ave.
Edmonton
Tel: (403) 452-3670
TWX: 610-831-2431

BRITISH COLUMBIA

Hewlett-Packard (Canada) Ltd.
4519 Canada Way
North Burnaby 2
Tel: (604) 433-8213
TWX: 610-922-5059

MANITOBA

Hewlett-Packard (Canada) Ltd.
513 Century St.
Winnipeg
Tel: (204) 786-7581
TWX: 610-671-3531

NOVA SCOTIA

Hewlett-Packard (Canada) Ltd.
2745 Dutch Village Rd.
Suite 206
Halifax
Tel: (902) 455-0511
TWX: 610-271-4482

ONTARIO

Hewlett-Packard (Canada) Ltd.
1785 Woodward Dr.
Ottawa 3
Tel: (613) 255-6180, 255-6530
TWX: 610-562-1952

Hewlett-Packard (Canada) Ltd.

50 Galaxy Blvd.
Rexdale
Tel: (416) 677-9611
TWX: 610-492-4246

QUEBEC

Hewlett-Packard (Canada) Ltd.
275 Hymus Boulevard
Pointe Claire
Tel: (514) 697-4232
TWX: 610-422-3022
Telex: 01-20607

FOR CANADIAN AREAS NOT LISTED:

Contact Hewlett-Packard (Canada) Ltd. in Pointe Claire, at the complete address listed above.

CENTRAL AND SOUTH AMERICA

ARGENTINA

Hewlett-Packard Argentina
S.A.C. S. de R.L.
Lavalle 1171 - 3°
Buenos Aires
Tel: 35-0436, 35-0627, 35-0431
Telex: 012-1009
Cable: HEWPACK ARG

BRAZIL

Hewlett-Packard Do Brasil
L.C. Ltda.
Rua Frei Caneca 1119
Sao Paulo - 3, SP
Tel: 288-7111, 287-5858
Cable: HEWPACK Sao Paulo

Hewlett-Packard Do Brasil

Praca Dom Feliciano 78
Salas 806/808
Porto Alegre
Rio Grande do Sul (RS)-Brasil
Tel: 25-8470
Cable: HEWPACK Porto Alegre

Hewlett-Packard Do Brasil

L.C. Ltda.
Rua da Matriz 29
Botafogo ZC-02
Rio de Janeiro, GB
Tel: 246-4417, 246-2919
Cable: HEWPACK Rio de Janeiro

CHILE

Héctor Calcagni y Cia, Ltda.
Casilla 16,475
Santiago
Tel: 423 96
Cable: CALCAGNI Santiago

COLOMBIA

Instrumentación
Henrik A. Langebaek & Kier S.A.
Carrera 7 No. 48-59
Apartado Aereo 6287
Bogota, 1 D.E.
Tel: 45-78-06, 45-55-46
Cable: AARIS Bogota
Telex: 44400INSTCO

COSTA RICA

Lic. Alfredo Gallegos Gurdían
Apartado 10159
San José
Tel: 21-86-13
Cable: GALGUR San José

ECUADOR

Laboratorios de Radio-Ingeniería
Calle Guayaquil 1246
Post Office Box 3199
Quito
Tel: 212-496; 219-185
Cable: HORVATH Quito

EL SALVADOR

Electronic Associates
Apartado Postal 1682
Centro Comercial Gigante
San Salvador, El Salvador C.A.
Paseo Escalon 4649-4° Piso
Tel: 23-44-60, 23-32-37
Cable: ELECAS

GUATEMALA

IPESA
5a via 2-01, Zona 4
Guatemala City
Tel: 63-6-27 & 64-7-86
Telex: 4176 Mahohegu

MEXICO

Hewlett-Packard Mexicana, S.A.
de C.V.
622 Adolfo Prieto
Col. del Valle
Mexico 12, D.F.
Tel: 543-4232; 523-1874
Telex: 017-74-507

NICARAGUA

Roberto Terán G.
Apartado Postal 689
Edificio Terán
Managua
Tel: 3451, 3452
Cable: ROTERAN Managua

PANAMA

Eléctrico Balboa, S.A.
P.O. Box 4929
Ave. Manuel Espinosa No. 13-50
Bldg. Alina
Panama City
Tel: 230833
Telex: 3481003, Curundu.
Canal Zone
Cable: ELECTRON Panama City

PARAGUAY

Z.T. Melamed S.R.L.
Division: Aparatos y Equipos
Médicos
Salón de Exposición y Escritorio:
Chile 482
Edificio Victoria—Planta Baja
Asunción, Paraguay
Tel: 4-5069, 4-6272
Cable: RAMEL

PERU

Compañía Electro Medica S.A.
Ave. Enrique Canaval 312
San Isidro
Casilla 1030
Lima
Tel: 22-3900
Cable: ELMED Lima

PUERTO RICO

San Juan Electronics, Inc.
P.O. Box 5167
Ponce de Leon 154
Pda. 3-PTA de Tierra
San Juan 00906
Tel: (809) 725-3342, 722-3342
Cable: SATRON/CS San Juan
Telex: SATRON 3450 332

URUGUAY

Pablo Ferrando S.A.
Comercial e Industrial
Avenida Italia 2877
Casilla de Correo 370
Montevideo
Tel: 40-3102
Cable: RADIUM Montevideo

VENEZUELA

Hewlett-Packard De Venezuela
C.A.
Apartado 50933
Caracas
Tel: 71.88.05, 71.88.69, 71.99.30
Cable: HEWPACK Caracas
Telex: 21146 HEWPACK

FOR AREAS NOT LISTED,

CONTACT:
Hewlett-Packard
INTERCONTINENTAL
3200 Hillview Ave.
Palo Alto, California 94304
Tel: (415) 493-1501
TWX: 910-373-1267
Cable: HEWPACK Palo Alto
Telex: 034-8300, 034-8493

The Symbols and Mnemonics for the Keys of the Basic Model 20

KEY	IN QUOTE FIELD?		KEY	IN QUOTE FIELD?		KEY	IN QUOTE FIELD?		KEY	IN QUOTE FIELD?	
	NO	YES		NO	YES		NO	YES		NO	YES
1	1	1	R()	R	#	LOAD	LODb	NOTE 2	S	NOTE 3	S
2	2	2	=	=	=	RECORD	RECb	NOTE 2	T	NOTE 3	T
3	3	3	≠	≠	≠	A	A	A	U	NOTE 3	U
4	4	4	>	>	>	B	B	B	V	NOTE 3	V
5	5	5	≤	≤	≤	C	C	C	W	NOTE 3	W
6	6	6	GO TO	GTO b	@	X	X	X	\$	NOTE 3	\$
7	7	7	GO TO SUB	GSS b	NOTE 2	Y	Y	Y	%	NOTE 3	%
8	8	8	RETURN	RET b	NOTE 2	Z	Z	Z	&	NOTE 3	&
9	9	9	STOP	STP b	!	D	NOTE 3	D	'	NOTE 3	'
0	0	0	END	END b	NOTE 2	E	NOTE 3	E	?	NOTE 3	?
.	.	.	JUMP	JMP b	NOTE 2	F	NOTE 3	F	SPACE	NOTE 3	<i>b</i>
ENTER EXP	E	↑	IF	IF b	NOTE 2	G	NOTE 3	G	These keys are the Mechanics Group and have no Mnemonics or Symbols.		
+	+	+	FLAG N	FLG b	NOTE 2	H	NOTE 3	H	EXECUTE	—	—
-	-	-	SET CLEAR FLAG N	SFG b	NOTE 2	I	NOTE 3	I	STORE	—	—
*	*	*	SET CLEAR FLAG N SET CLEAR FLAG N	CFG b	NOTE 2	J	NOTE 3	J	RUN PROGRAM	—	—
/	/	/	FIXED N	FXD b	NOTE 2	K	NOTE 3	K	CLEAR	—	—
√	√	√	FLOAT N	FLT b	NOTE 2	L	NOTE 3	L	ERASE	—	—
(((ENTER	ENT b	NOTE 2	M	NOTE 3	M	BACK	—	—
)))	DISPLAY	DSP b	NOTE 2	N	NOTE 3	N	FORWARD	—	—
;	;	;	PRINT	PRT b	NOTE 2	O	NOTE 3	O	DELETE	—	—
,	,	,	SPACE N	SPC b	NOTE 2	P	NOTE 3	P	INSERT	—	—
"	"	NOTE 1	NORMAL	NOR b	NOTE 2	Q	NOTE 3	Q	RECALL	—	—
→	→	→	TRACE	TRC b	NOTE 2	R	NOTE 3	R	LIST	—	—

NOTES

1. The " character never occurs inside a quote field; it is used exclusively to begin or terminate a quote field.
2. This key produces one character with an arbitrary pattern. Sometimes the pattern will vary according to which plug-in ROM's are installed.

3. If this key is used outside of a quote field, NOTE 11 will result unless the key is defined by a plug-in ROM, in which case the mnemonic or symbol is determined by the ROM.
4. The character *b* denotes a blank space.



PART NO. 09820-90001
MICROFICHE NO. 09820-99001

PRINTED IN U.S.A.
SEPT 1972