HP 9825 Desktop Computer
Operating and Programming Reference

Manual Part No. 09825-90200
Microfiche No. 09825-99200

Hewlett-Packard Desktop Computer Division
3404 East Harmony Road, Fort Collins, Colorado 80525

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

May 1980...First Edition.
June 1980...Updated pages: D-5, D-6, Disc Programming insert.
November 1980...Second Edition. Revised pages: v, 1-8, 1-10, B-1 thru B-23, C-1 thru C-10, D-9, D-10.
June 1983...Third Edition.

Your Operating and Programming Reference

This reference describes installing, operating and programming an HP 9825A or 9825B Desktop Computer. The 9825B contains all features of its predecessor, the 9825A. In addition, the 9825B has many optional language modules (ROMs) built-in and can be configured with up to 62 Kbytes of read/write memory.

This reference replaces these earlier 9825A manuals:

- 9825A Operating and Programming (09825-90000)
- String Variables Programming (09825-90020)
- Advanced Programming (09825-90021)
- Systems Programming (09825-90027)

Although the information is the same, it's arranged here for easy access and allows us to provide better documentation updating in the future. You'll find a complete index to topics in both this reference and the I/O Control Reference at the back of each binder.

This reference also provides room for the optional language ROM manuals currently useable with the 9825A and 9825B:

- Matrix Programming (09825-90022)
- Disk Programming (09885-90000 or 09825-90220).

Since the 9825A and 9825B are often referred to as calculators, computers and desktop computers, these terms are used interchangably throughout this reference.

We welcome your comments and suggestions for improving HP user documentation. You'll find a card at the back of this reference. If it's missing, address your comments to:

Hewlett-Packard Company
3404 E. Harmony Road
Fort Collins, CO 80525
ATTN: User Documentation

Reference Preview

Chapter 1: Installation

Covers installing your new desktop computer and describes accessories and services available for your computer.

Chapter 2: Keyboard Operations

Introduces you to the keyboard functions including editing keys, math operations, special function keys and system command keys. If you are not familiar with the 9825, please read this chapter before starting to program.

Chapter 3: HPL Programming

Describes the standard 9825 High-speed Programming Language (HPL). Each statement and function is presented, along with typical example program lines. You'll also find a brief introduction to programming in HPL.

Chapter 4: Advanced Programming

Explains the advanced programming language: for-next loops, subprograms with parameter passing, split and integer data storage and program cross-referencing. Each statement and function is covered, accompanied by many example program sequences.

Chapter 5: Tape Cartridge Operations

Shows how to use the built-in tape drive for program and data storage. The statements and commands covered here can also be used to control external 9875A Tape Drives.

Chapter 6: String Variables

Describes the statements and functions available for handling alphanumeric data, using either simple string variables or string arrays.

Chapter 7: Systems Programming

Covers the language extensions available with the large memory (9825T), including remote keyboard operation, terminal emulation, and program self-modification.

You'll find reference tables, a complete list of HPL syntax, all error codes and an index at the back of the reference. For a table of contents to each chapter, look under the appropriate tabbed divider.

9825B User Documentation

The standard set of 9825B manuals is listed here. The first three manuals can be ordered as the 9825B Manual Kit, 09825-87901.

Operating and Programming Reference (09825-90200) – Explains installation, keyboard and tape cartridge operations, and the HPL programming language. Additional chapters cover the Advanced Programming, String Variables and Systems Programming language extensions.

I/O Control Reference (09825-90210) – Describes the interfacing and peripheral-control operations built into the 9825B: General I/O, Extended I/O, and HP 9862A/9872A Plotter control. the 9825 Interfacing Concepts Guide is included with this reference. Space is provided for keeping interface manuals and interface operating notes.

9825A/B Pocket Reference (09825-90012) – Lists all HPL syntax and error codes in a handy, pocket-size format.

9825A/B System Test Booklet (09825-90037) – Explains how to run each mainframe and peripheral test supplied on the 9825 System Test Cartridge.

9825A/B Error Codes Booklet (09825-90015) – Error codes listed in a small booklet kept under the computer's paper-access lid.

Matrix Programming (09825-90022) – Describes the HPL language extensions available with the optional Matrix ROM.

Disc Programming (09825-90220) - Explains controlling HP Disc Drives via the HPL language extensions supplied with the optional 98217A or 98228A Disc ROM. This manual replaces the 9885 Disc Programming Manual, 09885-90000.

Peripheral Operating Notes

Each of the following notes is shipped when you order the appropriate interface card or HP computer peripheral. Each 98032A Interface note shows the interface wiring configuration for a particular interface application. Most notes contain detailed programming instructions for the system application. These operating notes are currently available:

- 9863A Tape Reader Operating Note (09825-90041)
- 9864A Digitizer Operating Note (09825-90042)
- 9866A/B Printer Operating Note (09825-90043)
- 9869A Card Reader Operating Note (09825-90044)
- 9871A Printer Operating Note (09825-90045)
- 9883A Tape Reader Operating Note (09825-90046)
- 9884B Tape Punch Operating Note (09825-90047)
- 9881A Printer Operating Note (09825-90048)
- 6940A Multiprogrammer Operating Note (09825-90049)
- 98035A Real Time Clock Operating Note (09825-90054)
- 9875A Tape Cartridge Memory Operating Note (09825-90075)

Interface Manuals

These 9800-series interfaces and manuals are currently available:

- 98032A Parallel I/O Interface Installation and Service (98032-90000)
- 98033A BCD Interface Installation and Service (98033-90000)
- 98034A HP-IB Interface Installation and Service (98034-90001)
- 98035A Real Time Clock Installation and Service (98035-90000)
- 98036A Serial I/O Interface Installation and Service (98036-90001)
- HP 9878A I/O Expander Installation and Service (09878-90000)

A brief description of each interface is in your 9825B I/O Control Reference. More complete information can be found in the Interfacing Concepts guide supplied with the I/O Control Reference.

Chapter 1
Table of Contents

Notes

Chapter 1

Installation

Inspection Procedure

The individual parts of your computer system were thoroughly inspected before they were shipped to you. All equipment should be in good operating order. Carefully check the computer, plug-in ROMs and peripheral equipment for any physical damage sustained in transit. Notify HP and file a claim with the carrier if there is any such damage.

Please check to ensure that you have received all of the items which you ordered and that any options specified on your order have been installed. The options installed are listed on a label under the computer's paper-access cover.

NOTE

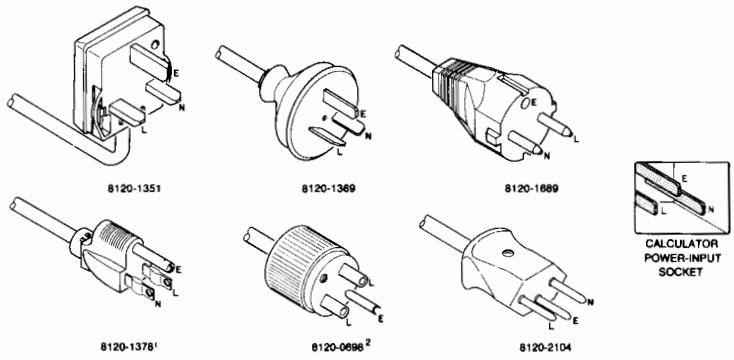
The standard 9825B is configured with 24 Kbytes of read/write memory and 9872 Plotter operation. If you wish to configure the system for 9862A Plotter operation or a larger memory, contact your HP Service Representative for assistance.

Also inventory the items in the Manuals Kit (09825-87901) and the Miscellaneous Kit (09825-80003). A pack list is supplied in each kit.

If you have any difficulties with your system, if it is not operating properly, or if any items are missing, please contact your nearest HP Sales and Service Office.

Power Cords

Power cords with different plugs are available for the calculator; the part number of each cord is shown below. Each plug has a ground connector. The cord packaged with each calculator depends upon where that calculator is to be delivered. If your calculator has the wrong power cord for your area, please contact your local HP sales and service office.



L = Line or active Conductor (also called "live" or "hot").
N = Neutral or Identified Conductor.
E = Earth or Safety Ground.

To protect operating personnel, we recommend that the computer be properly grounded. The computer is equipped with a three-conductor power cable which, when connected to an appropriate power receptacle, grounds the computer. Do not operate the computer from an ac power outlet which has no ground connection.

[1]UL and CSA approved for use in the United States of America and Canada with calculators set for either 100 or 120 Vac operation.

[2]UL and CSA approved for use in the United States of America and Canada with calculators set for either 220 or 240 Vac operation.

Power Requirements

The 9825 Computer has the following power requirements.

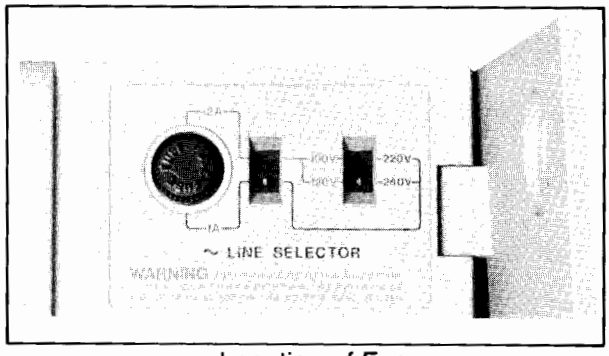
- Line Voltage: 100 Vac + 5%, −10%
  120 Vac + 5%, −10%
  220 Vac + 5%, −10%
  240 Vac + 5%, −10%
  } Switch Selectable
- Line Frequency: 48 to 66 Hertz
- Power Consumption: 100V @ 2.0A
  120V @ 1.8A
  220V @ 0.8A
  240V @ 0.8A

Fuses

For 100 or 120 Vac operation, use a 3A fuse; for 200 or 220 Vac operation use a 1.5A fuse.

WARNING

TO AVOID THE POSSIBILITY OF SERIOUS INJURY, DISCONNECT THE AC POWER CORD BEFORE REMOVING OR INSTALLING A FUSE.

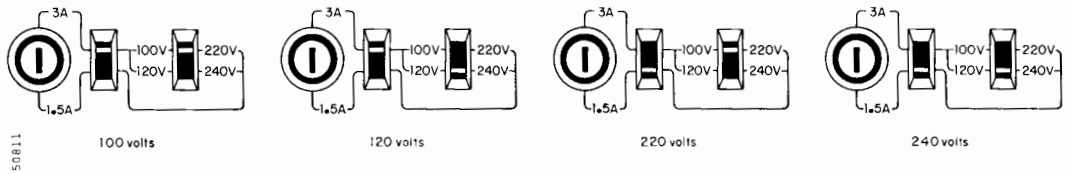


Location of Fuse

The figure shows the location of the fuse under the paper cover. To change the fuse, first disconnect the power cord to the calculator. Then remove the fuse cap by pressing inward while twisting it counterclockwise. Remove the fuse from the cap and insert the correct replacement fuse (either end) into the cap. Finally, put the fuse and cap back into the fuse holder. Press on the cap and twist it clockwise until it locks in place.

Initial Turn-On Instructions

1. With the calculator disconnected from its ac power source, check that the proper calculator fuse has been installed for the voltage in your area (see previous section).
2. Next, ensure that the two voltage selector switches under the paper cover are set for the correct powerline voltage. The figure below shows the correct settings for each nominal line voltage. If it is necessary to alter the setting of either switch, insert the tip of a small screwdriver into the slot on the switch. Slide the switch so that the position of the slot corresponds to the desired voltage, as shown below.

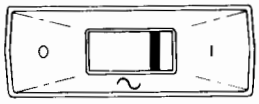


Nominal Line Voltage Settings

3. The operating system module on the right-hand side of the 9825A calculator must be inserted so that it is even with the side of the calculator.
4. Install the desired ROM cards and interface cards. See the next page and refer to the appropriate manual for interface installation.

> **CAUTION**
> ALWAYS TURN OFF THE CALCULATOR WHEN INSERTING OR REMOVING ROMS AND INTERFACES. FAILURE TO DO SO COULD DAMAGE EQUIPMENT.

5. Connect the power cord to the power input connector on the back of the calculator. Plug the other end of the cord into the ac power outlet.
6. Switch the calculator on using the switch on the right-hand side of the calculator.



Computer Testing

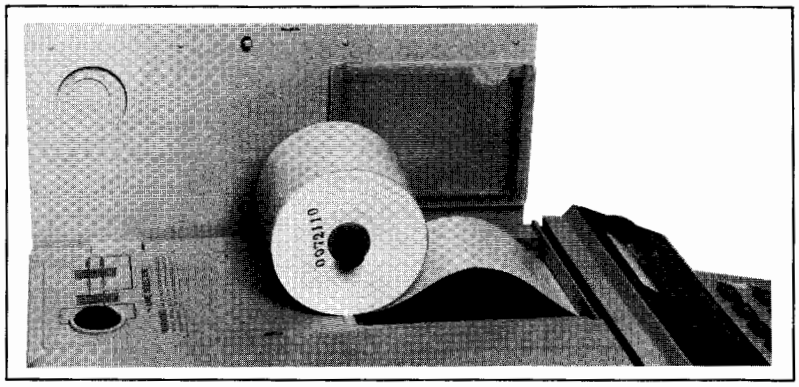
If you wish to test your calculator, or if there is any doubt that your calculator is operating correctly, refer to the System Test Booklet for the calculator test procedure.

Loading Printer Paper

The internal printer uses special heat-sensitive (thermal) paper. When ordering paper, specify the six-roll pack, HP part number 9270-0479.

To load a roll of paper:

1. Lift the paper cover and remove the paper spindle. Discard the old paper core and remove any paper left in the printer using the paper advance wheel.
2. Install the new roll as shown in the following figure.
3. Insert the free end of the paper and advance it through the printer using the paper advance wheel.



Loading Printer Paper

CAUTION

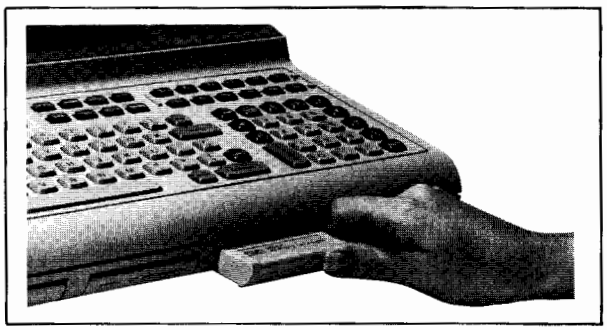
HP THERMAL PRINTER PAPERS ARE DESIGNED SPECIFICALLY FOR USE WITH HP DESKTOP COMPUTERS. USE OF OTHER PAPERS MAY DAMAGE THE PRINTER. TO MAINTAIN A VALID WARRANTY OR SERVICE CONTRACT AND ENSURE PROPER PRINTER OPERATION, USE ONLY HP THERMAL PAPER.

Accessory ROMs

Several ROMs (Read Only Memories) are available for your computer; each provides additional language capabilities to perform specific tasks such as plotting, controlling peripherals or extending the programming capabilities. One or more ROMs are packaged in a ROM card.

ROM Installation

A ROM card can be plugged into any one of the four ROM slots on the bottom front of the calculator as shown below.



ROM Installation

To install a ROM, first turn off the calculator. Then slide the ROM, with the label right-side-up, through the ROM slot door. Press it in so that it is even with the front of the calculator.

The ROMs listed below are an internal part of the 9825B Computer. They can be purchased in various combinations for the 9825A.

String Variables ROM

This ROM enables the calculator to recognize and operate on letters and words ("strings") in much the same way that it recognizes and operates on numbers. Some of the capabilities which are provided include: single strings and string arrays, numeric value of a string of digits, concatenation, and displaying or printing all special characters.

Advanced Programming ROM

This ROM extends the programming capabilities of the 9825 Calculator. For/next looping, split and integer precision number storage, multiparameter functions and subroutines, and the cross reference statement are the operations provided by the Advanced Programming ROM.

9862A and 9872A Plotter ROMs

These ROMs enable the 9825 to control HP 9862A and 9872A Plotters. Axes can be drawn and labeled; functions can be plotted; and in the "typewriter" mode, characters can be printed as you type them from the keyboard. More than one plotter can be operated at the same time with each ROM.

General I/O ROM

The General I/O ROM provides basic I/O capability with formatting. Most 9800 series peripherals (not the 9862A Plotter) can be controlled using this ROM. Binary I/O, status checking, and limited control of instruments via the HP Interface Bus are also provided.

Extended I/O ROM

The Extended I/O ROM extends the I/O capability of the calculator by providing complete HP-IB control, bit manipulation and testing, auto-starting, error trapping, and interrupt capabilities.

These ROMs are available for 9825A and 9825B Computers:

Matrix ROM

The Matrix ROM extends the language to include statements for manipulating matrices and arrays. Addition, subtraction, multiplication, and division of arrays, as well as inversion, transposition, and determinants of matrices are only some of the capabilities provided by this ROM.

Disk ROMs

The HP 98217A Disk ROM adds HPL language statements and functions for controlling HP 9885M and 9885S Flexible Disk Drives. Each 9885 Drive handles a ½ megabyte flexible disk. Both data and programs can be stored in a random-access, file-by-name structure. Up to eight 9885M (master) drives can be accessed. Up to three 9885S (slave) drives can be accessed via each 9885M.

The HP 98228A Disk ROM provides HPL language for controlling both HP 9885 and HP 9895 Disk Drives. Each 9895 handles one or two 1.2 megabyte flexible disks. The 98228A ROM can be used only with a 9825T computer.

Systems Programming ROM

This ROM add capability for remote keyboard operation, program self-modification, intelligent terminal emulation and run-time memory allocation. This ROM is available as the 98224A plug-in card for 9825A. The ROM is added to the 9825B with the large memory option (9825T).

Cleaning the Computer

The computer case has been painted with a long lasting, water-based paint. It is both non-toxic and environmentally safe. It will preserve the appearance of your computer for many years. When you want to clean the case, follow the instructions below to sustain the quality finish. If the case finish should become damaged, ask your local Hewlett-Packard sales and service office for touch-up paints.

CAUTION

CHEMICAL SPRAY-ON CLEANERS USED FOR APPLIANCES AND OTHER HOUSEHOLD OR INDUSTRIAL APPLICATIONS MAY DAMAGE THE CASE FINISH. DO NOT USE DETERGENTS THAT CONTAIN AMMONIA, BENZENES, CHLORIDES OR ABRASIVES.

Before cleaning the computer, disconnect the power cord and any interconnecting cables. Dampen a clean, soft, lint-free cloth in a solution of clean water and mild soap. Wipe the soiled areas of the case, ensuring that no cleaning solution gets inside the unit. For cleaning more heavily soiled areas, a solution of 80% clean water and 20% isopropyl alcohol may be used. Then dry the case with a dry, soft, clean cloth. A non-abrasive eraser may be used to remove pen and pencil marks.

Prerecorded Programs

Tape cartridges containing programs for solving problems from many disciplines are available. A utility program cartridge is supplied with each calculator. For a complete list of prerecorded programs and for pricing information, contact any HP sales office.

Service Contracts

When you buy a Hewlett-Packard desk-top calculator, service is an important factor. If you are to get maximum use from your calculator, it must be in good working order. A HP Maintenance Agreement is the best way to keep your calculator in optimum running condition.

Consider these important advantages:

- **Fixed Cost**– The cost is the same regardless of the number of calls, so it is a figure that you can budget.
- **Priority Service**– Your Maintenance Agreement assures that you receive priority treatment, within an agreed upon response time.
- **On-Site Service**– There is no need to package your equipment and return it to HP. Fast and efficient modular replacement at your location saves you both time and money.
- **A Complete Package**– A single charge covers labor, parts, and transportation.
- **Regular Maintenance**– Periodic visits are included, per factory recommendations, to keep your equipment in optimum operating condition.
- **Individualized Agreements**– Each Maintenance Agreement is tailored to your support equipment configuration and your requirements.

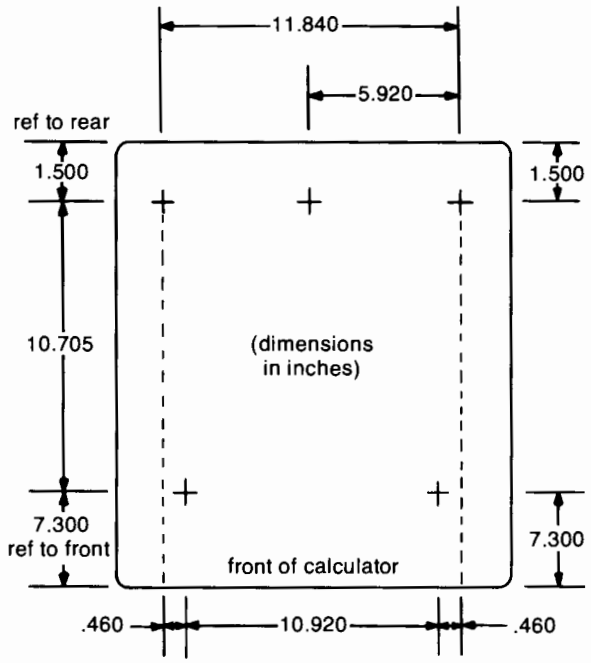
After considering these advantages, we are sure you will agree that a Maintenance Agreement is an important and cost-effective investment.

For more information please contact your local HP calculator sales and service office.

Table Mounting

Your calculator can be mounted to the top of a desk or table by following these steps:

1. Drill 5 holes in the top of your desk or table to accommodate #6-32 (National Coarse) screws according to the diagram below.
2. Remove the Phillips head #6-32NC screws that hold the rubber feet to the bottom of the calculator.
3. Use screws that are ½ inch longer than the thickness of the table top. This ½ inch allows for the thickness of the rubber feet and the hole for the screw in the bottom of the calculator.



Chapter 2
Table of Contents

Chapter 2

Keyboard Operations

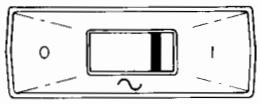
This chapter introduces some of the operating characteristics of the 9825 Desktop Computer. The keyboard, display, and range are a few of the topics covered.

Before Using the Calculator

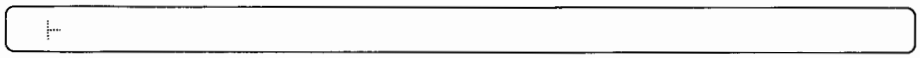
There are a few things you should check each time you turn on the calculator.

If the calculator is turned off:

- Set the power switch on the right-hand side of the calculator to the "1" position:



- When the following display appears, the calculator is ready for use:



If the calculator is turned on and the display is blank:

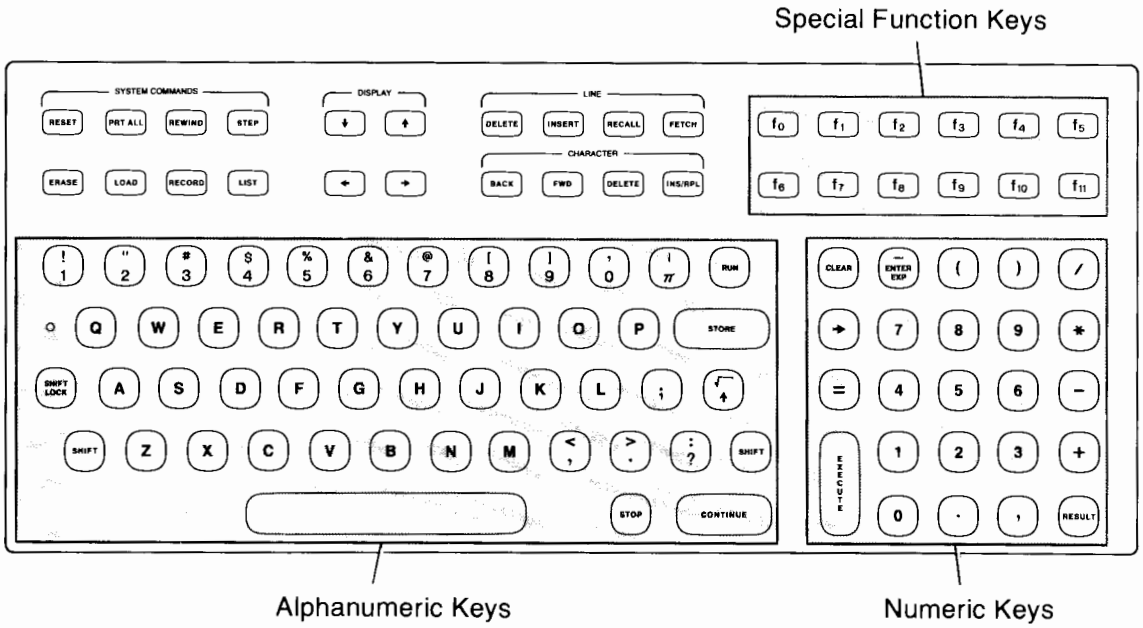
- Press CLEAR or STOP

If the display still remains blank, first check the power connection and fuse as described in chapter 1. If you still have a problem, call your HP sales and service office listed in the back of this manual.

If the calculator is on and the display shows the "lazy T", you can do keyboard operations or arithmetic or you can enter programs and run them.

General Information

The Keyboard



- **Alphanumeric Keys** - This area is very much like a standard typewriter keyboard. For instance, to display a capital A, press the shift key and (A) at the same time; or to display a percent sign, %, press the shift key and (% 5) at the same time.
- **Numeric Keys** - All the keys needed to enter numbers and do simple arithmetic are located in this block. The numeric keys in the alphanumeric section of the keyboard can also be used to enter numbers. The exponentiation and square root key, (√ ↑), is located in the alphanumeric key section.
- **Special Function Keys** - The keys in the upper right section of the keyboard, namely (f0) through (f11), provide additional calculator abilities. These keys are explained later in the chapter.

Keys of the same color have similar functions. For example, all the alphanumeric keys are the same beige color; gold colored keys are control keys used to run programs, store lines, erase programs, etc.

Below are a few more topics related to keyboard operations:

- **Spacing** - In general, spaces are not important. It makes no difference, for example if you key in:

  A+B or A + B

  Both are interpreted the same. Spacing, however, is important when using text (characters within quotes) and when printing and displaying messages.

- **Repetition of Keys** - When a key is held down, its operation is repeated rapidly. This is an especially useful feature with the editing keys.

- **The ⊢ Symbol** - When the display is clear and awaiting inputs, the "lazy T" symbol appears in the leftmost character of the display. This symbol also indicates the end of a stored line.

- **The Run Light** - A small red light in the left end of the display lights when a program is running.

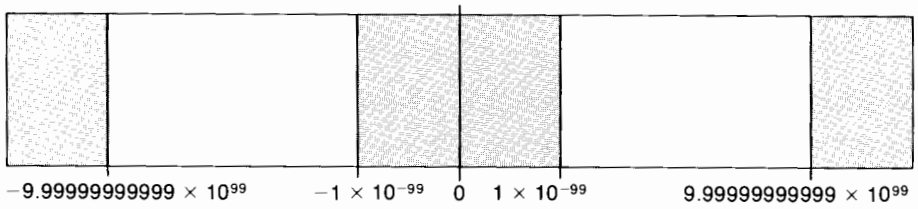
Display and Line Length

The 9825 Calculator has a 5 × 7 dot matrix, 32-character display. Even though only 32 characters can be displayed at one time, up to 80 characters can be keyed into the display. After the 32nd character, additional characters which are keyed in cause the displayed line to shift to the left. After 67 characters are keyed, a beep indicates that only thirteen more characters can be entered. Up to 73 characters can be stored. This includes any spaces or parentheses which the calculator may automatically insert in the line.

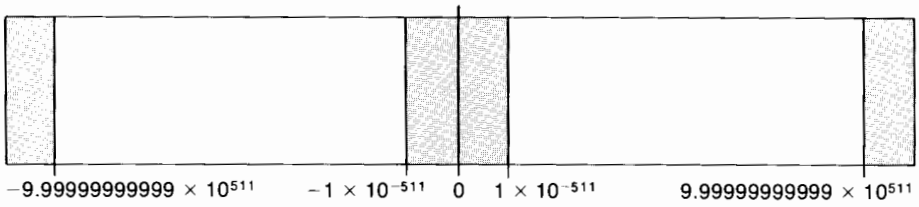
Range

The range of values which can be entered or stored is $-9.99999999999 \times 10^{99}$ through -1×10^{-99}, 0, 1×10^{-99} through $9.99999999999 \times 10^{99}$. However, the range of calculations is from $-9.99999999999 \times 10^{511}$ through -1×10^{-511}, 0, and 1×10^{-511} through $9.99999999999 \times 10^{511}$.

Storage Range



Calculating Range



out of range ▨ within range □

The extended calculation range is useful for calculations which have intermediate results outside of the storage range, but which have final results within the storage range. For instance:

$$(9.2 \times 10^{23} \times 8.6 \times 10^{80})/(1 \times 10^{24})$$

When the first two values are multiplied their result is:

$$(7.912 \times 10^{104})$$

This intermediate result cannot be stored, but the final result, 7.912×10^{80}, can.

Significant Digits

All numbers are stored internally with 12 significant digits in the mantissa and a two digit exponent. The format used to display or print numbers (such as fxd 2) has no effect on the internal representation of a number.

Memory

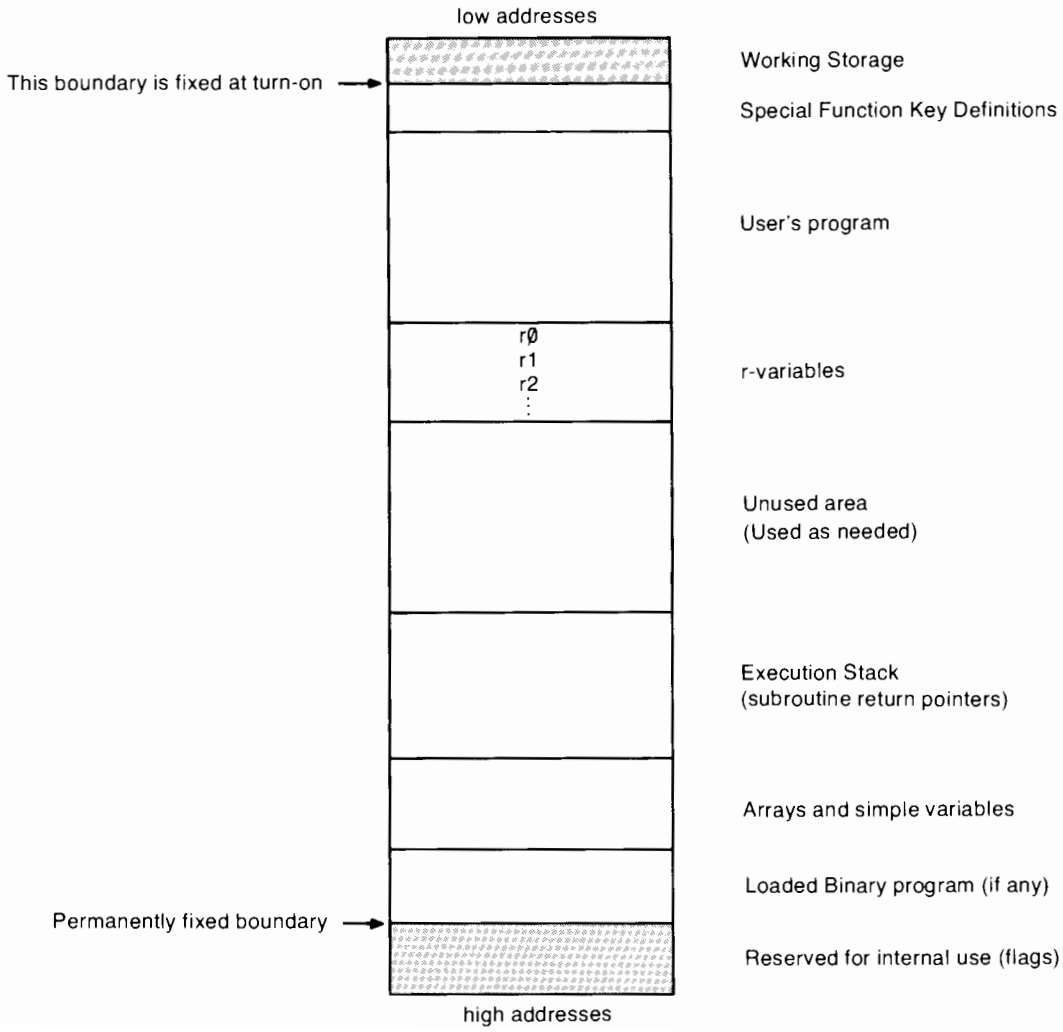
The 9825 Calculator uses two types of memory; Read/Write Memory, and Read Only Memory. Read/Write Memory is used to store programs and data. When you store a program or data, you "Write" into the memory. When you access a line of your program or a data element, you "Read" from memory; thus the term Read/Write.

Read Only Memory differs in that it is permanent. When the calculator is turned off, the contents of the Read/Write memory are lost, whereas the Read Only Memory is unaffected. ROM (for Read Only Memory) cards can be plugged into the ROM slots on the front of the calculator. This makes it possible to expand the language.

Programs and data in Read/Write memory can be saved for future use by recording the information on the tape cartridge.

A small amount of memory is sometimes required by a plug-in ROM. This area is called "working storage".

Read/Write Memory Organization



Language

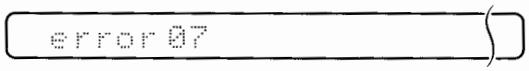
The language used by the HP 9825 Calculator is called HPL. The basic programming unit is the statement. Statements are typed using lower case abbreviated mnemonics, such as prt for print. Multi-statement lines can be stored by separating statements with semicolons.

Two other characteristics of this language are implied multiplication and the assignment operator. Implied multiplication is a standard algebraic notation, such as 5X. The assignment operator → points to the variable being assigned a value, such as 5 → D.

More mnemonics can be added to the language by adding ROM cards which plug into the ROM slots on the front of the calculator.

Error Messages

When an error occurs, the calculator beeps and displays an error number. The number references a description that will help pinpoint the cause of the error. For example:



Indicates a syntax error.

If an error message is displayed during an attempt to run a program, the program line number where the error occurs will also be displayed. For example:

Indicates that a parameter is out of range in line 3.

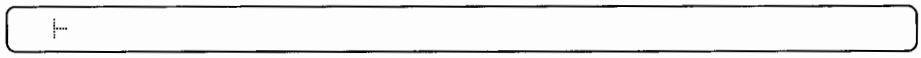
Pressing RECALL after some error messages will bring the line containing the error into the display with a flashing cursor indicating the location of the error.

A complete list of the error codes is at the back of this manual.

System Keys

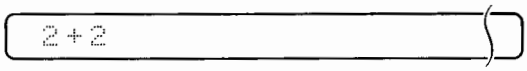
The following keys are used often for keyboard operations and programming.

CLEAR Clears the display; the ⊢ symbol remains to show that the calculator is ready for further instructions:

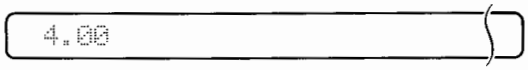


EXECUTE Performs the operation in the display. For example, to add 2 + 2:

Press: 2 + 2

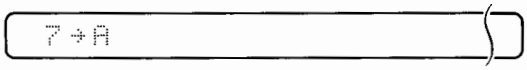


Press: EXECUTE

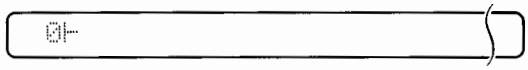


STORE Stores program lines in the memory. For example, to store a program line:

Type in: 7 → SHIFT* A



Press: STORE



This program line will assign the value 7 to the variable A.

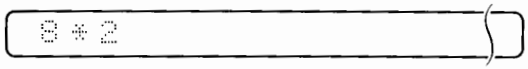
RUN Runs the program in memory from line 0.

*The SHIFT indicates that the following key is shifted.

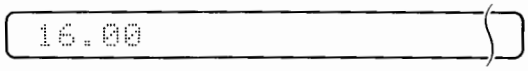
Keyboard Arithmetic

The six basic arithmetic operations in the 9825 are: addition (+), subtraction (−), multiplication (*), division (/), exponentiation (↑), and square root (√).

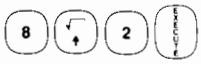
To perform a math operation, such as 8 × 2, first you key in the expression as follows:

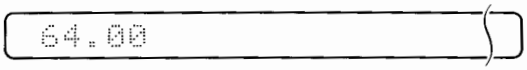


Then press: 



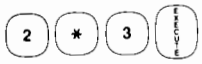
To raise a number to a power, such as 8^2, press:

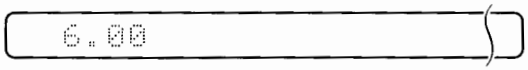


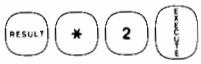


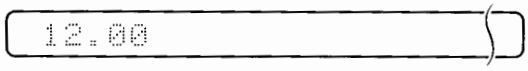
Notice that an operation such as 8^{-2} must appear as: 8↑(−2).

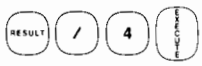
The value which is displayed after pressing the execute key is stored in a location called "result". This value can be used in other calculations. For example:

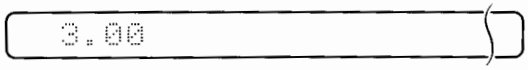












If you execute an operation involving large numbers, such as:

the calculator displays the result in scientific notation, with 9 digits to the right of the decimal point:

```
5.400000000e 13
```

This is because the number is too large for the fixed 2 notation which is set when you switch on the calculator.

Arithmetic Hierarchy

When an expression has more than one arithmetic operation, the order in which the operations take place depends on the following hierarchy:

√	square root	performed first
↑	exponentiation	↓
no operator	implied multiplication	↓
* /	multiplication and division	↓
+ −	addition and subtraction	performed last

An expression is scanned from left to right. Each operator is compared to the operator on its right. If the operator to the right has a higher priority, then that operator is compared to the next operator on its right. This continues until an operator of equal or lower priority is encountered. The highest priority operation, or the first of the two equal operations, is performed. Then any lower priority operations on the left are compared to the next operator to the right. If parentheses are encountered, the expression within the parentheses is evaluated before the left-to-right comparison continues. This comparison continues until the entire expression is evaluated. For example:

2+3*6↑2(4)/(6−2)↑2	exponentiation
2+3*36(4)/(6−2)↑2	implied multiplication
2+3*144/(6−2)↑2	multiplication
2+432/(6−2)↑2	evaluate parenthesis
2+432/4↑2	exponentiation
2+432/16	division
2+27	addition
29	result

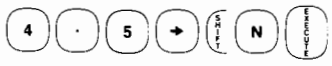
Variables

A variable is a name of a location where data is stored. There are two types of variables: numeric variables and string variables. Each data type can be stored in either simple or array form. Numeric data can also be stored in r-variables.

Simple Variables

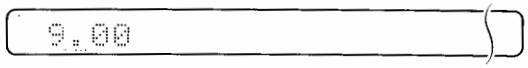
Twenty-six simple variables, named A through Z, are used on the 9825 Calculator. Only the upper case letters can be used for simple variable names.

To assign a value to a variable, the assignment operator is used. For instance, to assign the value 4.5 to N, press:

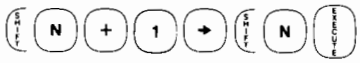


The number always appears on the left, and the variable appears on the right side of the assignment operation.

Now, N can be used in calculations. For instance, to multiply N by 2, press:



N is not changed. New values can be assigned to variables, such as:



r-Variables

r-variables are designated by a lower-case "r" followed by a number (e.g., r18). They are useful for one dimensional arrays and can be used in addition to the 26 simple variables.

In the following two examples, the value 12 is assigned to r10. Then the value 20 is assigned to the register designated by the value of r10 (this is called indirect storage).

12 → r10	The value 12 is assigned to r10 directly.
20 → rr10	The value 20 is assigned to r12 indirectly.

For more information about variables, see the next chapter and the String Variables chapter.

Operating Modes

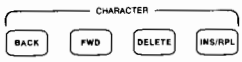
The calculator can operate in any of three modes: the calculator mode, the program mode, or the live keyboard mode.

- In calculator mode, no program is running, and the calculator is awaiting inputs or calculating keyboard entries.

- In the program mode, a program is running.
- In live keyboard mode, you can perform many calculator operations while a program is running.

Basic Editing

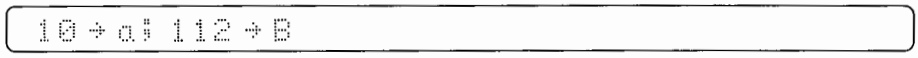
If you make a mistake while entering lines into the display, you can use the character editing keys for changing the line.



For instance, suppose you want to type in this line:

10 → A; 12 → B

But, instead you type:



To correct this, simply press [BACK] until a flashing cursor appears over the "a".

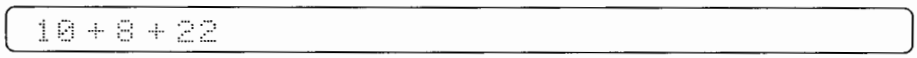
Then type in an A. To delete a "1" in 112, press [FWD] once and press character [DELETE]. The resulting display would be:

with a flashing cursor on the "1" of 12. To execute the line, press: [ENTER]

As another example, maybe you want to execute this line:

10 + 18 + 22

But you typed this:

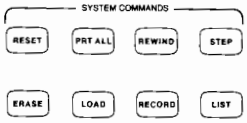


To insert a one in front of the 8, press the [BACK] key 4 times. The flashing replace cursor ▒ will be positioned on the 8. Next, press the [INS/RPL] key. This changes the replace cursor to the insert cursor ┤. Now, type in a 1. The display will be:

```
10+18+22
```

Note that the rest of the line shifted to the right 1 character. The insert cursor ┤ will still be flashing over the 8 indicating that more characters could be inserted if desired. To execute the line, press [ENTER].

System Command Keys



RESET Returns the calculator and I/O cards to the power-on state without erasing programs or variables. [RESET] is executed immediately when it is pressed; it does not have to be followed by [ENTER]. All calculator activity is halted and the line number of the current location in a program is displayed if a program is running. The reset key should be used to reset the calculator when no other key, such as [CLEAR] or [STOP], will bring the calculator to a ready state.

PRT ALL Sets the print-all mode on or off. When it is pressed once, the word on appears in the display. When it is pressed again, the word off appears in the display. In print-all mode, displayed results, executed lines, and stored lines are printed.

While a program is running in print all mode, all displayed messages and error messages are printed. Print-all mode can be turned on or off while a program is running.

REWIND Automatically rewinds the tape cartridge to its beginning. Other statements and commands can be executed immediately without waiting for the cartridge to completely rewind. If [REWIND] is pressed while a program is running or while a line is executing from the keyboard, the cartridge rewinds at the end of the current line.

STEP Executes a program, one line at a time. Then, the line number of the next line to be executed is displayed. When [STEP] is pressed just after stopping a program, only the line number of the next line to be executed is displayed. The next time [STEP] is pressed, that line is executed.

To step from a specific line, execute a gto X, where X is the line to start stepping from. For example, to begin stepping through your program from line 30, type in gto 30 and press [EXECUTE]. Then use the step key.

ERASE This typing aid is used to erase all or part of the Read/Write memory.

[ERASE] [A] [EXECUTE]	Erases the entire calculator memory.
[ERASE] [V] [EXECUTE]	Erases only the variables.
[ERASE] [K] [EXECUTE]	Erases all the special function keys.
[ERASE] [EXECUTE]	Erases programs and variables.
[ERASE] [fn]	Erases the special function key represented by "n".

The Reset Table in the Reference Tables appendix lists things affected by the erase command.

LOAD This typing aid is used to load programs and data from the tape cartridge. For example to load a program which is on file 3:

[LOAD] [3] [EXECUTE]	Loads the program from file 3 into the calculator.

The display shows ldf (for "load file") when this key is pressed. See the load file statement in the Tape Cartridge chapter.

RECORD This typing aid is used to record programs and data on the tape cartridge. Before recording on the tape cartridge, files must be marked (see the Tape Cartridge chapter). In the following example, it is assumed that the file has been marked:

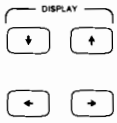
[RECORD] [6] [EXECUTE]	Record the calculator program on file 6 of the tape cartridge.

The display shows rcf (for "record file") when this key is pressed (see the record file statement in the Tape Cartridge chapter).

LIST This typing aid is used to list programs, sections of programs, all special functions keys, or individual special function keys. For example:

LIST EXECUTE	Lists the entire program.
LIST K EXECUTE	Lists all defined special function keys in numerical order.
LIST f_0	Lists special function key, f_0.
LIST 2 0 EXECUTE	Lists the program from line 20 to the end.
LIST 9 . 1 3 EXECUTE	Lists the program from line 9 to 13, inclusive.

Display Control Keys



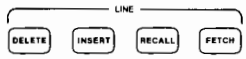
↓ Brings the line with the next higher-valued line number into the display. If there are no more lines in the program, ↓ clears the display and allows new program lines to be appended to the end of the program.

↑ Brings the line with the next lower-valued line number into the display. If a line number is in the display, ↑ brings that line into the display. If a stop statement is executed from a program, ↑ brings the line following the line with the stop statement into the display. After a program error, ↑ brings the line containing the error into the display for editing.

← Moves the line in the display to the left. This allows all the characters in a line to be moved into the display. Each time it is pressed, the displayed line moves 8 characters.

→ Moves the line in the display to the right for viewing all the characters in a line. Each time this key is pressed the displayed line moves 8 characters.

Line Editing Keys



FETCH This typing aid is used to bring program lines into the display and to fetch special function keys. For example:

FETCH 2 0 ENTER — Brings line 20 into the display.

FETCH f4 — Accesses special function key f4. If f4 is defined, its definition is displayed. Otherwise ↑4 is displayed.

DELETE Deletes the program line in the display from the program. If no program line is in the display, the calculator beeps and the key is ignored. To delete a program line, fetch the line into the display and press DELETE. When a line is deleted from a program all subsequent line addresses and all relative and absolute go to and go sub statements are renumbered to reflect the deletion.

This is not the same key as the character delete key explained later. To delete several program lines, the delete (**del**) command can be used. The delete command is explained later.

INSERT Inserts a line into a program. The inserted line is inserted before the fetched line. The fetched line and higher line numbers are renumbered. The FETCH, ↑, or ↓ keys can be used to fetch a line into the display. For example:

To insert the line: ⌜A→B between lines 20 and 21:

```
20: A+1→A
21: gto 25
```

Press: FETCH 2 1

Type in: ⌜A→B

Press: INSERT

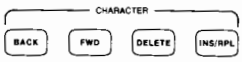
```
20: A+1→A
21: ⌜A→B
22: gto 26
```

When a line is inserted into a program, the branching addresses of all relative and absolute go to and go sub statements are adjusted to reflect the insertions as in line 22 above.

RECALL Brings back, into the display, one of the two previous keyboard entries. Pressing [RECALL] once brings back the most recent keyboard entry. Pressing it twice brings back the previous keyboard entry.

Press [RECALL] after errors resulting from keyboard operations to recall the line containing the error. For many errors, a flashing cursor indicates the location of an error in the line.

Character Editing Keys



Lines which are fetched into the display using [↑], [↓], [RECALL], or [FETCH], and lines which are typed into the display can be edited using the character editing keys.

Two flashing cursors are associated with these keys: the replace cursor ▒ and the insert cursor ⫞.

BACK Moves the flashing replace cursor ▒, or the flashing insert cursor ⫞, from its current position in the line in the display, toward the beginning (left) of the line. If the cursor is not visible, [BACK] causes the cursor to appear on the right-most character in the line.

FWD Moves the flashing replace cursor ▒, or the flashing insert cursor ⫞, from its current position in the display, towards the last character in the line. For a line which has just been fetched or typed into the display, pressing [FWD] causes the flashing cursor to appear on the left-most character in the display.

DELETE Deletes individual characters which are under the insert or replace cursor. This is not the same key as the line delete key explained previously.

INS/RPL The insert/replace key is used to change the flashing replace cursor to a flashing insert cursor and vice versa. Use the [BACK] or [FWD] key to position the cursor in the display. When the insert cursor is flashing, any characters entered from the keyboard are inserted to the left of the cursor and the characters under and to the right of the cursor shift to the right.

When the replace cursor is flashing, any character entered replaces the existing display character at the location of the cursor and the cursor moves to the character on the right.

Calculator Control Keys

RUN This key is an immediate execute key which runs the program in the calculator beginning at line zero. All variables, flags, and subroutine return pointers are cleared when (RUN) is pressed. The run light at the left end of the display indicates a running program.

The Reset Table in the Reference Tables appendix lists things which are affected by pressing (RUN).

STORE Stores individual program lines. Also, when a special function key is fetched and defined, (STORE) is used to store the key's definition. A program line can be a single statement or several statements separated by semicolons. When an error occurs while attempting to store a line, (RECALL) brings that line back into the display. A flashing cursor usually shows where the error was encountered in the line.

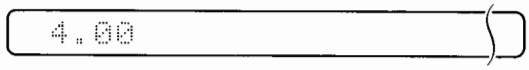
SHIFT and **SHIFT LOCK** are used to obtain shifted keyboard characters, such as A, #, and [. When (SHIFT LOCK) is pressed, the small light above the key lights. (SHIFT LOCK) locks the keyboard for shifted characters. Press (SHIFT) to release shift lock.

STOP Stops the program at the end of the current line. The number of the next program line to be executed is displayed. When (STOP) is pressed, list, tlist, and wait statements are aborted but the rest of the line is executed. When (STOP) is pressed in an enter statement, flag 13 is set and the enter statement is terminated.

There is also a stop statement. For details, see the next chapter.

EXECUTE Executes the single or multi-statement line which is in the display. The two most recently executed (or stored) keyboard entries are temporarily stored and can be recalled by pressing (RECALL) once or twice. The result of a numeric keyboard operation which is not assigned to a variable is stored in Result (see (RESULT) key). For example:

(2) (+) (2) (EXECUTE)



Pressing (EXECUTE) displays and stores the result. Pressing the execute key again repeats the same operation.

Although multiple expressions such as:

2+2 :r4 (ENTER)

are allowed, only the result of the last expression in the line is displayed and stored in Result. In print-all mode, both results are printed.

CONTINUE Automatically resumes a program from where it was stopped. When a line number is in the display (such as after pressing (STOP)) (CONTINUE) resumes the program from that line. However, after pressing (RESET), or after editing the program, the program continues at line 0 when (CONTINUE) is pressed. Pressing (CONTINUE) after an error also causes the program to continue from line 0.

In an enter statement, (CONTINUE) is pressed after entering data. If no data is entered and (CONTINUE) is pressed, the variable maintains its previous value and flag 13 is set. See also the continue command on page 2-24.

RESULT Accesses the storage location of the result of a numeric keyboard operation which was not assigned to a variable. For example:

Press: (3) (*) (6)

```
3*6
```

Press: (ENTER)

```
18.00
```

The answer, 18, is also stored in Result and can be used in other operations, such as:

Press: (RESULT) (/) (2)

```
res/2
```

Press: (ENTER)

```
9.00
```

In a program, values cannot be stored in Result; but the value in Result can be assigned to variables or used in computations.

For example:

```
0: 20→res
1: res+2→A
```

This is not allowed.

This assigns the value of Result +2 to the variable A.

CLEAR Clears the display. If the clear key is pressed during the enter statement, a question mark appears in the display, indicating that an entry is still expected. If this key is pressed after a special function key has been fetched, the key number (e.g., f8) appears in the display.

→ The assignment operator is used to assign values to variables (this is not the same as the right arrow used for display control.) For example:

Press: [SHIFT] [√] [5] [→] [SHIFT] [X] [EXECUTE] This stores the square root of 5 in X.

[π] To enter the value of π, this key is pressed. The value entered is 3.14159265360.

[(-) ENTER EXP] This key enters a lower case e into the display, representing an exponent of base 10. The unshifted [E] key can be used in place of [(-) ENTER EXP]. For example:

Press: [1] [(-) ENTER EXP] [9] [9] [EXECUTE]

```
1.000000000e 99
```

Press: [4] [E] [2] [3] [EXECUTE]

```
4.000000000e 23
```

Note that there is no difference between pressing [(-) ENTER EXP] and pressing [E].

Special Function Keys

[f0] [f1] [f2] [f3] [f4] [f5]

[f6] [f7] [f8] [f9] [f10] [f11]

There are 12 special function keys, which provide 12 unshifted functions and 12 shifted functions. The special function keys can be used as typing aids, one line immediate execute keys, or as immediate continue keys.

To define a special function key, press [FETCH] and the special function key to be defined. Then enter a line in the display. Press [STORE] to store the definition of the key and to exit key mode. For example:

Press: [FETCH] [f0] `f0` is displayed if the key was not previously defined.

Type-in: `prt` Enters `prt` in the display.

Press: [STORE] This stores `prt` under f_0, for use as a typing aid.

If you decide not to define a special function key after fetching one, the [STOP] key can also be used to exit key mode.

To list all of the defined special function keys in numerical order, type in: `list k` and press [EXECUTE].

To list individual special function keys, press [LIST] and then the special function key to be listed.

Immediate Execute Special Function Keys

If a line to be stored under a special function key is preceded by an asterisk (*), it is an immediate execute key. This means that when the key is pressed, the contents of the key are appended to the display and the line in the display is executed automatically.

For example:

Press: [FETCH] [SHIFT] [f11]	Accesses f_{23} (shifted f_{11}).
Type-in: *prt "π", π	The asterisk makes this an immediate execute key.
Press: [STORE]	This stores the line entered in the display under f_{23}.

Whenever [SHIFT] [f11] is pressed and the display is clear, the following is printed:

```
π              3.14
```

Immediate execute keys are useful for executing selected segments of a program. Using the continue command followed by a line number, you can make several entry points in your programs. For example:

```
[f0]: *cont 5
[f10]: *cont 10
```

Each time [f0] is pressed, the program continues at line 5, or at line 10 if [f10] is pressed.

Immediate Continue Special Function Keys

If a line to be stored as a special function key is preceded by a slash (/), it is an immediate continue key for use with the enter statement. "Immediate continue" means that when the key is pressed, the contents of the key are appended to the display and continue is executed automatically. Immediate continue keys are used to enter often used values in enter statements. For example:

Press:	FETCH f_{10}	Fetches special function key f_{10}.
Type-in:	`/2.71828182846`	This enters the value of e, the base of the natural logarithms, into the display.
Press:	STORE	This stores the line in the display under f_{10}.

Whenever an enter statement is waiting for a value and the f_{10} key is pressed, the approximate value for e (i.e., 2.71828182846) is entered and the program continues (see enter statement in the next chapter).

Keys with Multiple Statements

By separating statements with semicolons, several statements can be stored under one special function key. As an example, suppose you want to convert inches to centimeters. The following line is stored under special function key f_0.

Press: FETCH f_0

Type-in: `**→R; dsp R, "in. =", 2.54R, "cm."`

Press: STORE

Then key in a number, such as 6, and press f_0. The display will show:

```
6.00 in. =   15.24 cm.
```

Commands

Five commands are explained in this section. Commands can be executed only from the keyboard; they cannot be stored as part of a program.

The Run Command

run [line number or label]

The run command clears all variables, flags, and subroutine return pointers and then starts program execution. If a line number or label is specified, the program begins execution at the specified line number or label. Since [RUN] is an immediate execute key equivalent to run 0 [EXECUTE], the word run must be keyed in to run from a line number or label.

Examples:

run [EXECUTE]	Run beginning at line 0. This is the same as pressing [RUN].
run 20 [EXECUTE]	Run, beginning at line 20.
run "third" [EXECUTE]	Run, beginning at the label "third".

The Continue Command

cont [line number or label]

The continue key (**cont**) command continues the program without altering variables, flags, or subroutine return pointers. If no line number is specified, then the program continues from the current position of the program line counter. When a line number or label is specified, the program continues at the specified line or label. If the program has been edited or an error has occurred since the program ran, continue without parameters causes execution to begin at line 0. Since [CONTINUE] is an immediate execute key equivalent to cont 0 [EXECUTE], the word cont must be keyed in to continue at a line number or label.

Examples:

`cont` [EXECUTE]	Continue from current position of program line counter. This is the same as pressing [CONTINUE].
`cont 3` [EXECUTE]	Continue from line 3.
`cont "loop"` [EXECUTE]	Continue from the label "loop".

The Delete Line Command

`del` beginning line number [, ending line number] [,*]

The delete (**del**) command is used to delete lines or sections of programs. When one line number is specified , only that line is deleted. When two line numbers are specified, all lines in the block are deleted. To delete an entire program, and leave the variables, `del 0, 9999` can be executed.

Examples:

`del 28` [EXECUTE]	Delete line 28.
`del 13, 20` [EXECUTE]	Delete lines 13 through 20.
`del 18, 9999` [EXECUTE]	Delete program from line 18 to the end. (This does not affect variables.)

An attempt to delete lines that are destinations of relative or absolute go to or go sub statements (except labels) will cause error 36. To delete these lines, the delete command with the optional asterisk parameter can be used. When the asterisk is used, any go to or go sub statements which reference deleted lines are adjusted to reference the first line after the deleted section. For example to delete line 24 in this program segment:

```
22: ent U;if
 U=0;gto 24
23: U+T→T;C+1→C;
 gto 22
24: prt "Avg.
 Usage",T/C
25: prt "Total
 Usage",T
```

Type-in: del 24,*

Press: [ENTER]

Press: [LIST] 2 2 . 2 4 [ENTER]

```
22: ent U;if
 U=0;sto 24
23: U+T→T;C+1→C;
 sto 22
24: prt "Total
 Usage",T
```

The Erase Command

erase [a or v or k or special function key]

The erase command is used to erase programs, variables, and special function keys as shown below.

Command	Meaning
erase	Erases program and variables.
erase a	Erases everything (like switching the calculator off and then on again).
erase v	Erases all variables.
erase k	Erases all special function keys.
erase [fn]	Erases the indicated special function key.

Things affected by the erase command are listed in the Reset Table (see the Reference Tables appendix).

The Fetch Command

fetch [line number or special function key]

The fetch command brings individual program lines into the display. This is useful for editing lines or for viewing individual program lines. Fetching a special function key displays the definition of the key or f followed by the key number if the key is undefined. Executing fetch alone, fetches line 0.

Examples:

fetch 10	Fetch line 10.
fetch f1	Fetch special function key f1.

Live Keyboard

The calculator's live keyboard mode provides additional power for executing single or multi-statement lines while a program is running. Among other things, you can perform math operations, monitor program activity, and alter program flow in live keyboard mode. Two statements described in this section permit the live keyboard mode to be turned on or off.

How Live Keyboard Works

While a program is running, a live keyboard operation is executed as follows:

- The live keyboard operation is keyed into the display and [EXECUTE] is pressed.
- At the end of the current program line, the live keyboard line is executed.
- The live keyboard operation is executed entirely before the program continues.

Live Keyboard Math

Any math operations can be executed from live keyboard. Thus, when a program is running and a few calculations need to be made, key in the operation and press [EXECUTE].

Statements in Live Keyboard

Math operations are just a small part of what can be done from live keyboard. If you want a listing of the current program, press [LIST] [EXECUTE].

To check a variable in the program, key in the variable name, such as A or B[4] and press [EXECUTE]. The current value of the variable will be displayed.

To change a variable from live keyboard, enter the new value and assign it to the variable to be changed. For example to reset a counter such as C + 1 → C to 0, key in 0 → C and press [EXECUTE].

Subroutines from Live Keyboard

Parts of a program can be executed from live keyboard as subroutines using the go sub statement. For example, the following section of a running program is used to monitor the variables used in the program:

```
11: "check":prt
 A,B,C,D;ret
```

By executing gsb "check", the values of the variables are printed and control returns to the program.

After a subroutine is finished, control returns to the main program when the return (ret) or stop (stp) statement is executed or when a stop flag at the beginning of a line is encountered.

Special Function Keys in Live Keyboard

Although the special function keys f_0 through f_{23} cannot be defined from live keyboard, they can be used from live keyboard. In this example, the special function keys are used to alter the flow of the running program.

The special function keys are defined as follows:

f_0	f_1	f_2
*1 → F	*2 → F	*3 → F

The program is:

```
0: "Wait":dsp
 "waiting";wait
  100;jmp F
1: gto "first"
2: gto "second"
3: gto "third"
4: "first":prt
 "first";0→F;
 gto "Wait"
5: "second":prt
 "second";0→F;
 gto "Wait"
6: "third":prt
 "third";0→F;
 gto "Wait"
```

When the program is run, `waiting` is displayed until one of the immediate execute (line preceded by *) special function keys is pressed. Then the program branches to the line where either `first`, `second`, or `third` is printed. Although this is a simple example, it shows one way that special function keys can be used in live keyboard mode.

The Stop Key in Live Keyboard

If (STOP) is pressed during a live keyboard operation, the live keyboard operation is stopped, but the program continues. Pressing (STOP) a second time will stop the program.

Live Keyboard Limitations

Operations that modify the stored program or special function keys and operations that directly affect the execution of the program are not allowed in live keyboard mode. These operations include the following:

Mnemonic	Error
Commands:	
run	error 03
cont	error 03
fetch	error 03
erase	error 03
del	error 03
Statements:	
ent	error 13
end	error 09
gto (allowed in a live keyboard subroutine)	error 09
ldp	error 64
ldk	error 64
ldf (program file)	error 64

In addition, the following keys cause a beep and are ignored when pressed in live keyboard mode.

(STEP) (DELETE) (INSERT) (RUN) (STORE) (CONTINUE)

The Display

Lines which are typed in live keyboard mode will disappear from the display if the running program uses the display. The live keyboard line is re-displayed after each keystroke so that the line with the new character added can be seen.

If the running program continually uses the display, the live keyboard lines will not be visible while the line is being typed. In this case, the line that is currently being typed, or the line accessed by [RECALL] can be held in the display by pressing [↑] or [↓]. These keys will suspend the running program for one second and display the line. If the key is kept depressed, the program will be halted for one second after it is released. After the line is executed, the [↑] or [↓] key will not re-display the line unless [RECALL] is pressed first. For example, suppose the following program is running in the calculator:

```
0: dsp "Live
 Keyboard";wait
 100
1: gto 0
```

When the following line is typed in live keyboard, it will not be visible:

```
prt 25→A
```

Press [↑] or [↓] and the line will be displayed for about one second. When [ENTER] is pressed, the line will be executed and 5 will be stored in A and printed.

Results of calculations performed in live keyboard disappear from the display if a running program uses the display. The [↑] or [↓] keys only hold the live keyboard line in the display and not the result of the execution of a line. The result can be held in the display by appending a wait statement to the end of the line (e.g. `10+12; wait 1000`).

A special function key can be defined to preserve the displayed result long enough to be viewed as in this example:

Press: [FETCH] [f0]
Type in: `*; wait 1000`
Press: [STORE]

As you type in a calculation such as `5*6`, press [f0] instead of [ENTER]. The result of the calculation will remain in the display for about one second.

The Live Keyboard Enable Statement

```
lke
```

The live keyboard enable (**lke**) statement enables the live keyboard mode. For example:

```
31: lke
```

Enable live keyboard.

Live keyboard is automatically enabled when the calculator is turned on, `erase a` is executed, or [RESET] is pressed. To disable live keyboard, the live keyboard disable (lkd) statement is used.

The Live Keyboard Disable Statement

```
lkd
```

The live keyboard disable (**lkd**) statement disables live keyboard mode. For example:

```
0: lkd
```

The first line of this program disables live keyboard.

To re-enable live keyboard during a program it is necessary to execute the live keyboard enable (lke) statement from the program.

[STOP], [REWIND], and [RESET] are the only keys recognized while a program is running with live keyboard disabled.

During cartridge operations, the keyboard is disabled and all keys except [RESET] are ignored.

Chapter 3
Table of Contents

Chapter 3

HPL Programming

This chapter introduces the statements, functions and operators comprising the HPL language.

Programming Concepts

There are five basic steps in creating a program:

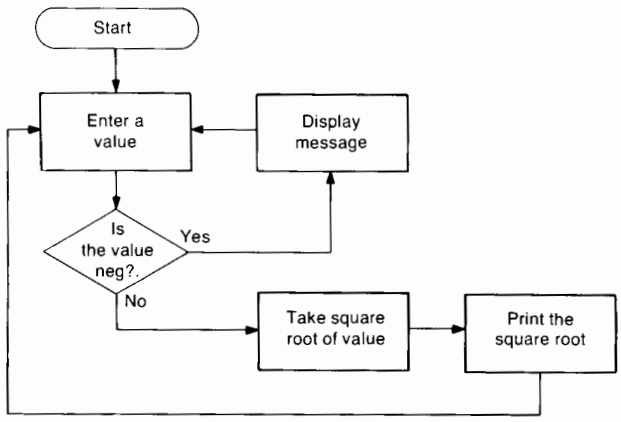
1. Define the Problem.
2. Decide how the problem is best solved.
3. Write out the statements for the program.
4. Key the statements into the computer memory.
5. Debug (correct) and run the program.

Step 1:

As a simple example, suppose you want to print the square root of each value that you enter. Then, if the value entered is negative, print a message and continue on.

Step 2:

A common method used to solve a problem is flowcharting*. Using a few basic flowcharting symbols, explained at the end of this chapter, we will flowchart the problem.



* Another method suitable for simple problems is to key in a few statements and try them out.

Step 3:

From the flowchart, write down the statements for the program:

Program	Comments
"start": ent V	V is the value to be entered.
if V<Ø; dsp "neg. V"; gto"start"	Decide if V is negative: if so, display the message and go back to the beginning.
√V→S	S is the square root of the value.
prt S	Print the square root.
gto "start"	Go to "start" for another value.

Note that the second line contains three statements separated by semicolons. All of the statements used are discussed later.

Step 4:

The next step is to clear the calculator by executing erase a. Then type in the program exactly as above, one line at a time. Press (STORE) at the end of each line to store that line in the calculator memory. If you make a mistake before you store the line, press (CLEAR) and type the line over.

Step 5:

After the program is stored, press (LIST) (↑) to get a printed listing. Then, to run the program press (RUN). Each time that V? is displayed, type in a value and press (CONTINUE). The calculator will print the square root of each value.

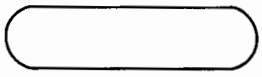
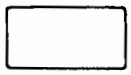
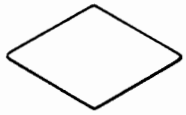
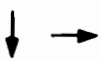
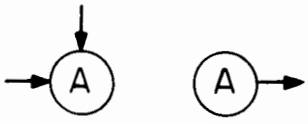
```
 0: "start":ent V
 1: if V<0;dsp
  "neg. V";sto
  "start"
 2: √V→S
 3: prt S
 4: gto "start"
```

For positive values, the program runs as expected, but if you enter a negative value you won't see the message displayed. This is because the message is displayed for a very short period of time before another display (i.e., V?) appears. Use a **wait** statement after the display statement in line 1. This statement causes the program to pause long enough for you to see the message. To change the program, press: [STOP] [FETCH] [1] [EXECUTE]. Then press the [BACK] key until it is positioned on the semicolon just before the **gto** statement. Press [INS RPL] and key in ;wait 500. Press [STORE] to store the new line at line 1. Then press [RUN]. Here is a listing of the completed program:

```
0: "start":ent V
1: if V<0;dsp
 "neg. V";wait
 500;gto "start"
2: rV→S
3: prt S
4: gto "start"
```

Since the program is a continuous loop, press [STOP] to stop the calculator. Then, to do another program key in erase a and press [EXECUTE]. This clears out the calculator memory.

Commonly Used Flowchart Symbols

	Meaning
	Program beginning or end.
	Program segment; usually one statement.
	Decision block indicates that a decision for a branch is made. Usually an **if** statement is used for a decision.
	Flowlines indicate the program flow.
	Connectors indicate that the lines going to or from them are connected.

Syntax Conventions

The statements, functions, and operators explained in this chapter are all programmable. Most of these instructions can also be used in calculator mode.

Statements can be programmed or executed. Operators and functions must be part of a statement in order to be programmed. This means that operations, such as **10 + 32** or $\sqrt{63}$, which can be executed from the keyboard, must be part of a statement in order to be programmed. Thus, `10+32→X` or `prt √63` are valid statements.

The instructions explained throughout this manual use the following syntax conventions. A complete list of syntax is near the back of the manual.

`[ ]`	- items within square brackets are optional.
`dot matrix`	- items in dot matrix must appear as shown.
...	- three dots indicate that the previous item can be duplicated.

Numeric Variables

The calculator uses two types of variables, numeric and string. Numeric data can be stored in simples variables, array variables, and r-variables. As numeric variables are allocated, they are initially assigned the value 0. Numeric variable elements each require 8 bytes* of memory. String variables are covered in chapter 6.

Simple Variables

There can be 26 simple variables, named A through Z. A simple variable must appear in upper case. Each simple variable can be assigned one value. For example:

```
0: 12→A
1: prt A
```

Assigns the value 12 to A.
Prints the value of A on the printer.

Array Variables

There can be 26 arrays, named A through Z. Array names are followed by square brackets which enclose the subscripts of the array (e.g., `L[3]`).

* A byte is the basic unit of data in the 9825. Eight bytes are required to store a number.

Before an array element can be used, the array must be declared in a dimension (**dim**) statement. This reserves memory for the array and initializes all elements in the array to zero. In the dimension statement, each dimension of an array can be specified either by specifying the upper bound, in which case the lower bound is assumed to be one, or by specifying both the lower and upper bounds. For example:

```
dim A[4,5]
```
Reserves memory for the 20 elements of the two-dimensional array A.

```
dim P[-2:1,-2:2]
```
Reserves memory for the 20 elements of the two-dimensional array P. (Lower and upper bounds specified.)

An array can have any size and any number of dimensions within the limits of the memory size and line length. The bounds must be between −32767 and 32767.

An individual element of an array is accessed by specifying the subscripts of the element. For example:

```
4→A[1,5,4,6]
```
4 is assigned to element 1,5,4,6 of array A.

```
3→P[-2,1]
```
3 is assigned to element −2,1 of array P.

Another Example:

```
0: dim Q[10,10]
```
Reserves memory for 100 elements of array Q.

```
1: 3→Q[7,1]
```
Q[7,1] is assigned the value 3.

```
2: 5→Q
```
The value 5 is assigned to the simple variable Q. There is no connection between the simple variable Q and array Q[10,10].

```
3: 2→Q[1,Q]
```
Q[1,5] is assigned the value 2.

r-Variables

r-variables are specified by a lower case "r" followed by a value or expression. When an r-variable is encountered, memory is reserved for all r-variables with smaller subscripts which have not been allocated. As r-variables are allocated, they are assigned the value 0. Thus if r10 is assigned a value, r0 through r9 are also automatically allocated and assigned the value zero if they have not been previously allocated.

Examples:

```
0: 4→r0
```

4 is assigned to r-variable 0.

```
1: 2→rr0
```

2 is assigned to r-variable 4. r0= 4, therefore 2 → r4. This is known as indirect storage.

Variable Allocation

Simple variables and r-variables are allocated when a statement containing either is executed. Array variables must be allocated using a dimension statement.

Before a variable is allocated, three cases are checked:

1. Before a variable is allocated by the dimension statement, a check is made to see if it is already allocated. If so, an error results and execution stops.
2. When a simple variable is referenced in any other statement, a similar check is made as to whether it has been allocated. If not, it is allocated.
3. When an array element is referenced in any other statement, a similar check is made as to whether the array has been dimensioned. If not, an error results.

Within one statement, variables are allocated in the same left-to-right order as they occur in the statement.

Number Formats

Numbers can be displayed or printed in floating-point format (scientific notation) or in fixed-point format. The calculator's internal representation of numbers is unaffected by number formats, therefore, accuracy is not changed.

When the calculator is turned on, [RESET] is pressed, or `erase a` is executed, the number format is fixed 2 (**fxd 2**), and for very large numbers, the calculator temporarily prints and displays in float 9(**flt 9**).

The Fixed Statement

fxd [number of decimal places]

The fixed (**fxd**) statement sets the format for printing or displaying numbers. In fixed-point format, the number of digits to appear to the right of the decimal point is specified. Fixed 0 through fixed 11 can be specified.

To set the number format from floating-point to the current fixed-point setting, fxd without parameters is executed.

When a number of the form:

$$A = N \times 10^{E}$$

where: $1 \leq N < 10$, or $N = 0$

is too large to fit in the fixed-point format, the number format temporarily reverts to the previously set floating-point (float 9 if no other floating-point format has been set) if:

$$D + E \geq 14$$

where: D is the number of decimal places specified in the fixed statement.
E is the exponent of the number.

To illustrate the reversion to a previous float 9 setting, run this program ⏵

```
0: ent A
1: fxd 0;prt A
2: fxd 1;prt A
3: fxd 2;prt A
4: fxd 3;prt A
5: end
```

If the value 125e10 is entered when A? appears in the display, this is printed ⏵

```
  1250000000000
1250000000000.0
1.2500000000e 12
1.2500000000e 12
```

For numbers too small to fit in the fixed-point format, zeros are printed or displayed for all decimal places, with a minus sign if the number is negative. For example:

```
0: fxd 3;dsp -
 .000125;wait
 2000
1: fxd 2;dsp
 .00204
```

```
-0.000
```

```
 0.00
```

Here are some numbers and their output format if `fxd 3` is executed:

Number	Fixed 3 Output
18	18.000
−.000006	-0.000
−2.7532	-2.753
4.5678	4.568
5.3111e3	5311.100
1234567891234.5	1.234567891e 12 (float 9 previously set)

The Float Statement

`flt` [number of decimal places]

The float (**flt**) statement sets floating-point format which is scientific notation. When working with very large or very small numbers, floating-point format is most convenient. Float 0 through float 11 can be specified. To set the number format from fixed-point to the current floating-point setting, `flt` without parameters is executed.

A number output in floating-point format has the form:

−D.D...De−DD

- The left-most non-zero digit of a number is the first digit displayed. If the number is negative, a minus sign precedes this digit; if the number is positive or zero, a space precedes this digit.
- A decimal point follows the first digit; except in **flt 0**.

- Some digits may follow the decimal point; the number of digits is determined by the specified floating-point format (e.g., in **flt 5**, five digits follow the decimal point).
- Then the character e appears, followed by a minus sign or space (for non-negative exponents) and two digits. This is the exponent, representing a positive or negative power of ten. The exponent indicates the direction and the number of places that the decimal point would have to be moved to express the number in fixed-point format.

Here are some numbers as they would appear if `flt 2` is executed:

Number	Float 2 Output
−3.2	−3.20e 00
271	2.71e 02
26.377	2.64e 01
.000004	4.00e−06
2.482e33	2.48e 33

Significant Digits

All numbers are represented internally with 12 significant digits regardless of the number format being used. To illustrate this, execute `fxd 5` then key in the number:

```
123456789.56789
```

then press (↑) and note the display:

```
123456789.56700
```

The 13th and 14th digits, 8 and 9, are not stored and zeros are displayed for those digits.

Rounding

A number is rounded before being displayed or printed if there are more digits to the right of the decimal point than the number format allows. The rounding is performed as follows: the first excess digit on the right is checked. If its value is 5 or greater, the digit to the left is incremented (rounded-up) by one; otherwise it is unchanged. The number remains unchanged internally. For instance:

```
0: fxd 2
1: dsp 1.235;
 wait 1000
2: dsp 2.404
3: end
```

```
1.24
```

```
2.40
```

The Display Statement

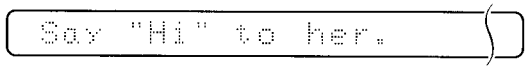
dsp [any combination of text or expressions]

The display (**dsp**) statement displays numbers or text on the calculator display. Commas are used to separate variables or text (e.g., dsp "No.", N, B).

Quotes are used to indicate text. To display quotes within text, it is necessary to press [shift] [" 2] twice for each quote to be displayed. For example:

Type in: dsp "Say ""Hi"" to her."

Press: [ENTER]



Displayed lines longer than 32 characters can be viewed using the display control keys, [←] and [→].

Numbers and text which are displayed remain in the display until another display operation (such as enter (**ent**) with a prompt) clears it.

The Print Statement

prt [any combination of text or expressions]

The print (**prt**) statement is used to print numbers or text on the calculator printer. For example:

```
prt 6
prt "One", 1
prt "This one"
```

```
                6.00
One             1.00
This one
```

If an expression is to be printed, such as:

prt 6*7→X

the expression is evaluated and the equivalent value is printed (and also stored in X in this case).

To print a quote within text press (shift) (") twice for each quote to be printed. For example:

Type in: `prt "Ent ""1"" or ""0"""`

Press: (EXECUTE)

```
Ent "1" or "0"
```

Commas are used to separate variables or text. For example:

Type in: `prt "First", 1, "Next", 2`

Press: (EXECUTE)

```
First          1.00
Next           2.00
```

When printing lines of text and values, the printout follows this format:

- Text followed by a numeric is printed on the same line if it fits; otherwise the text is printed and the number is printed on the next line.
- Each line of text separated by commas begins on a new line and folds over on successive lines if it is longer than 16 characters.
- Numerics separated by commas are printed one per line unless the format is **flt 10** or **flt 11** which requires two lines each.

When `prt` is specified without parameters, no operation takes place. To space one line, use the space statement.

The Enter Statement

`ent` [prompt ,] variable [, [prompt ,] variable...]

The enter (**ent**) statement is used to assign values to variables from the keyboard during a program. The variable can be a simple variable, array variable, or an r-variable. For example:

```
 4: ent Q
 5: ent A,B[3],
  r11
```

When an enter statement is encountered in a program, key-in a number, variable (such as `r30`), or expression (such as `√5`) and press (CONTINUE).

When many items are entered from the keyboard, it is often helpful to have a message called a "prompt" displayed representing the variable being assigned a value. For instance:

```
 0: ent "Amount",
  A
 1: ent "Temperat
  ure",T
```

```
Amount
```

```
Temperature
```

If no prompt is given, the calculator uses the name of the variable as the prompt. For example:

```
3: ent A[7]
```

```
A[7]?
```

If a null quote field is given as a prompt, such as `10: ent " ",A` the calculator retains any previously displayed message, unless a print operation is between the display statement and the enter statement. This is useful for variable prompts using the display statement. For example:

```
7: 1976→Y;fxd 0
8: dsp "July,",Y
9: ent "",A
```

```
July, 1976
```

You can calculate values from the keyboard while the program waits in the enter statement. This is done simply by entering the calculation and pressing [EXECUTE]. If the value to be entered is the result of pressing [EXECUTE], press [RESULT] or [RECALL] then press [CONTINUE]. Pressing [EXECUTE] immediately before pressing [CONTINUE] causes a default condition as if [CONTINUE] were pressed without entering a value.

Complex lines can be entered as the response to an enter statement. For instance, run this program:

```
0: ent B
1: ent A
2: prt A
3: end
```

When the display is:

```
B?
```

enter a value for B. Then when the display is:

```
A?
```

Type in: `20: if B>20: 40`

Then press [CONTINUE]. If the value that you entered for B is greater than 20, then `40` is printed, otherwise `20` is printed.

If [CONTINUE] is pressed without entering a value, the variable maintains its previous value and flag 13 is set. When a value is entered, flag 13 is cleared. See flags later in this chapter.

To terminate a program during an enter statement, press [STOP]. The rest of the program line is completed before the calculator stops.

Commands, such as `fetch` or `run`, are not allowed during an enter statement and cause error 03.

The following example illustrates a unique case using the enter statement. Run the short program:

```
0: dim A[20]
1: 4→I
2: ent I,A[I]
```

```
I?
```

Type in: 8

```
A[4]?
```

Press: CONTINUE

Notice that the value of I when the enter statement is encountered is used, not the entered value of I. To use the entered value of I as the subscript, use another enter statement. For the above example, change line 2 to:

```
2: ent I;ent
 A[I]
```

Even though you can have one enter statement that enters values for several variables, only one value can be supplied at a time. For example:

```
0: ent A,B
```

type in a value for A when a A? appears in the display and press CONTINUE, then do the same when B? appears in the display.

The Enter Print Statement

enp [prompt ;] variable [;[prompt ;] variable...]

The enter print (**enp**) statement is the same as the enter statement except that prompts and the entered values are printed and displayed as they are encountered.

For example, type in this short program to calculate the area of a circle:

```
0: enp "radius",
 R
1: πRR→A
2: prt "area",A
3: end
```

If 2 is entered for R when the program is run, the printout will be:

```
radius
2
area          12.57
```

The Space Statement

spc [number of blank lines]

The space (**spc**) statement causes the printer to output the number of blank lines indicated. The number of lines can be an expression with a range of 0 through 32767. If no parameter is specified, one blank line is output.

Examples:

```
0: spc A+B
```
Space the number of lines specified by A + B.

```
1: spc 5
```
Space 5 lines.

```
2: spc
```
Space one line.

The Beep Statement

beep

The beep statement causes the calculator to output a beep. For example, the calculator normally beeps, displays error 67, and stops when the argument of the square root (√) function is negative. In the following short program, the value entered for A is tested. If it is negative, the calculator beeps and displays a message, but the program continues entering values.

```
0: fxd 4
1: "start":ent
 "Argument";A
2: if A<0;gto
 "error"
3: prt √A;gto
 "start"
4: "error":beep
5: dsp "√ of
 neg. no."
6: wait 2000
7: gto "start"
```

The Wait Statement

wait number of milliseconds

The wait statement causes a program to pause the specified number of milliseconds (thousandths of a second). The wait statement is often used with display or enter statements to display a message for a specified time. The number of milliseconds can be an expression. The maximum wait is around 33 seconds, which is specified by the value 32767.

Since the wait statement takes time to be executed, small values in the wait statement are actually longer than a millisecond. This becomes evident in a loop which is executed many times.

Examples:

```
wait 2000
```

Pauses for 2 seconds.

```
 2: wait 2*I
```

Pauses for 2*I milliseconds.

In the next example, a display statement is followed by an enter statement. To preserve the first display for one second, the wait statement is used.

```
10: dsp "Please
 enter";wait
 1000
11: ent "value
 of X",X
```

The first display remains one second before the next display.

The Stop Statement

```
stp
```

The stop (**stp**) statement stops program execution at the end of the line in which it is executed. Pressing [CONTINUE] continues the program at the next program line. [STEP] can also be used to "step" through the program one line at a time. If any editing is performed after the program stops, [CONTINUE] and [STEP] cause the program to continue from line 0.

The stop statement can also be used for debugging. See the section on debugging statements for details.

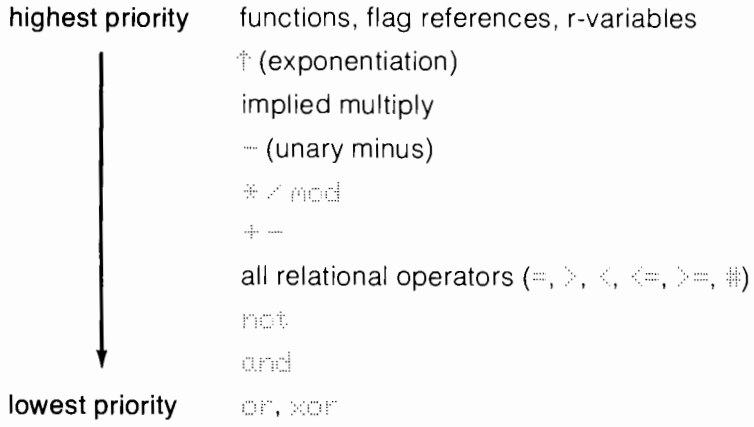
The End Statement

```
end
```

The end statement causes the program to stop like the stop statement. However, the end statement resets the program line counter to line 0 and resets all subroutine return pointers (see go sub statement). The end statement is usually put at the end of a program. The end statement cannot be executed during an enter statement, nor in live keyboard mode.

Hierarchy

In a statement containing functions, arithmetic operations, relational operations, logical operations, imbedded assignments, or flag operations, there is an order in which the statement is executed. This order is called the hierarchy, which is:



An expression is scanned from left to right. Each operator is compared to the operator on its right. If the operator on the right has a higher priority, then that operator is compared to the next operator on its right. This continues until an operator of equal or lower priority is encountered. The highest priority operation, or the first of the two equal operations, is performed. Then any lower priority operations on the left are compared to the next operator to the right. If parentheses are encountered, the expression within the parentheses is evaluated before the left-to-right comparison continues. This comparison continues until the entire expression is evaluated. In the following example, S_1, S_2, S_3... indicate intermediate results:

2A = B + C – D mod E exp (not F)	implied multiplication
S_1 = B + C – D mod E exp (not F)	addition
S_1 = S_2 – D mod E exp (not F)	evaluate parenthesis
S_1 = S_2 – D mod E exp S_3	exp function
S_1 = S_2 – D mod E S_4	implied multiplication
S_1 = S_2 – D mod S_5	mod operator
S_1 = S_2 – S_6	subtraction
S_1 = S_7	equality relation
S_8	final result

Operators

The four groups of mathematical or logical symbols, called operators, are: the assignment operator, arithmetic operators, relational operators, and logical operators.

Assignment Operator

expression → variable

The assignment operator is used to assign values to variables. For example:

`1.4→A`	The value 1.4 is assigned to the variable A.
`3: B→A`	The value of B is assigned to the variable A.

There are other ways to assign values to variables such as the enter (**ent**) statement or the load file (**ldf**) statement.

To assign the same value to many variables, the assignment operator can be used as in this example.

```
32→A→B→X→r4
```

Multiple assignments can also take the form `(25→A)+1→B` (which is the same as `25→A: A+1→B`). This is called an imbedded assignment.

Arithmetic Operators

There are six arithmetic operators as follows:

+	Add (if unary, no operation)	A + B or +A
−	Subtract (if unary, change sign)	A − B or −A
*	Multiply	A*B
/	Divide	A/B
↑	Exponentiate	A^B
mod	Modulus	A mod B is the remainder of A/B when A and B are integers. A mod B is the same as A − int (A/B)*B.

When A is much larger than B, there is a chance that a value of 0 could be returned for AmodB. This condition can be caught by examining the exponent of A/B when it is represented in floating point notation with one digit to the left of the decimal point. If the exponent is greater than 8, AmodB results in a value of 0.

Besides the * symbol for multiplication, implied multiplication can be used. In the following instances, implied multiplication takes place:

- Two variables together (like AB).
- A variable next to a number (like 5A).
- A variable or number next to a parenthesis [like 5(A + B)].
- A parenthesis next to a parenthesis [like (A + B) (X + Y)].
- A variable, number, or parenthesis preceding a function name (like 32 sinA).

For example:

AB→X	A times B is stored in X.
(5)5→X	5 times 5 is stored in X.
A(B+C)→B	A times the sum B + C is stored in B.
5 abs B	5 times the absolute value of B.

Relational Operators

There are six relational operators as shown in the following table.

Symbols	Meaning
=	Equal to.
>	Greater than.
<	Less than.
=> or >=	Greater than or equal to (either form is acceptable).
=< or <=	Less than or equal to (either form is acceptable).
# or < > or ><	Not equal to (either form is acceptable).

The result of a relational operation is either a one (if the relation is true) or a zero (if it is false). Thus if A is less than B, then the relational expression A<=B, is true and results in a value of one. All comparisons are made on 12 significant digits, signs, and exponents.

The relational operators can be used in any statement which allows expressions as arguments. For example:

`A=B → C`	Assignment statement. If A and B are equal, a 1 is stored in C; otherwise, a 0 is stored in C.
`if A>B:...`	If statement. If A is greater than B, then continue in the line; but if A is less than or equal to B, go to the next line.
`...:jmp A>3`	Jump statement. If A is greater than 3, jump 1 line, otherwise jump to the beginning of the line (jmp 0).
`prt A(A>B)+B(A<B)`	Print statement. If A is greater than B, the value of A is printed. If A is less than B, then the value of B is printed. If A equals B, then 0 is printed.

Logical Operators

The four logical operators, **and**, **or**, **xor** (exclusive or), and **not** are useful for evaluating Boolean expressions. Any value other than zero (false) is evaluated as true. The result of a logical operation is either zero or one.

Operation	Syntax	Truth Table

AND — expression `and` expression

A	B	A and B
F	F	0
F	T	0
T	F	0
T	T	1

OR — expression `or` expression

A	B	A or B
F	F	0
F	T	1
T	F	1
T	T	1

Operation	Syntax	Truth Table
Exclusive OR	expression xor expression	A | B | A xor B F | F | 0 F | T | 1 T | F | 1 T | T | 0
NOT	not expression	A | not A F | 1 T | 0

For example:

Program:

```
0: .1→A:0→B
1: prt "A and
 B",A and B
2: prt "A or B",
 A or B
3: prt "A xor
 B",A xor B
4: prt "not A",
 not A
5: end
```

Printout:

```
A and B      0.00
A or B       1.00
A xor B      1.00
not A        0.00
```

Math Functions and Statements

The math functions and math statements are explained in this section.

Parentheses must enclose the argument of a function when a "+" or "−" sign precedes the argument. In the examples, parentheses are shown only where they are required.

General Functions

Syntax	Description	Examples (fxd 5)
√ expression	Returns the square root of a non-negative expression. For negative expressions, see the section on math errors.	√64 = 8.00000 √π = 1.77245
abs expression	Determines the absolute value of the expression.	abs (−3.09) = 3.09000 abs 330.1 = 330.10000

Syntax	Description	Examples (fxd 5)
sgn expression	The sign function returns a −1 for negative expressions, 0 if the expression equals 0, and 1 for a positive expression.	sgn (−18) = −1.00000 sgn 0 = 0.00000 sgn 34 = 1.00000
int expression	Returns the largest integer less than or equal to the expression. This is often referred to as the "floor" integer value of the expression.	int 2.718 = 2.00000 int (−3.24) = −4.00000
frc expression	Gives the fractional part of a number. It is defined by: expression − int expression	frc 2.718 = 0.71800 frc (−3.24) = 0.76000
prnd (expression, rounding specification)	Returns the value of the argument rounded to the power-of-ten position indicated by the rounding specification.	prnd (127.375, −2) = 127.38000 127.375 is rounded to the nearest hundredth (10^{-2})
drnd (expression, number of digits)	The digit round function rounds the argument to the number of digits specified. The leftmost significant digit is digit number 1.	drnd (73.0625,5) = 73.06300 drnd (−65023,1) = −70000.00000 drnd (.058,1) = 0.06000
min (list of expressions and arrays)	Returns the smallest value in the list. An entire array can be specified by substituting an asterisk for the array subscript list (such as B[*]).	0: dim A[3]: 2→A[1] 1: 9→A[2]: 3→A[3] min (A[*]) = 2.00000 min (2, −3, −3, 4) = −3.00000
max (list of expressions and arrays)	Returns the largest value in the list. An entire array can be specified by substituting an asterisk for the array subscript list (such as B[*]).	0: dim A[3]: 2→A[1] 1: 9→A[2]: 3→A[3] max (A[*]) = 9.00000 max (5, 4, −3, 8) = 8.00000
rnd [−] expression	The random number function generates a pseudo-random number greater than or equal to 0 and less than 1. When the argument is positive, the starting seed is π/180 (which is .0174532925200). This seed is initialized when the calculator is turned on, erase a is executed, or RESET is pressed. Each subsequent access to the **rnd** function with a positive argument uses a seed based on the previous result of the function.	rnd 1 = 0.67822

(continued)

Syntax	Description	Examples (fxd 5)
rnd (cont'd)	To specify a starting seed other than $\pi/180$, use a negative argument. The fractional part of the absolute value of the argument is used as the seed. To obtain a good seed use a number less than 0 and greater than −1. The more non-zero digits in the number, the better. Last digits of 1, 3, 7, or 9 are preferable.	0: wait rnd(-.31317) 1: rnd 1+A Note that the wait statement is used instead of an assignment statement to initialize the starting seed. Line 1 generates a random number based on .31317 instead of $\pi/180$.

Logarithmic and Exponential Functions

Syntax	Description	Examples (fxd 5)
ln expression	The natural logarithm function calculates the logarithm (base e) of a positive valued expression.	ln 8001 = 8.98732 ln .0026 = −5.95224
exp expression	The exponential function raises the constant, naperian e, to the power of the computed expression. The range of the argument is approximately from −227.95 through 230.25.	exp 1 = 2.71828 exp (−3) = .04979
log expression	The common logarithm function calculates the logarithm (base 10) of a positive valued expression.	log 305.2 = 2.48458 log .0049 = −2.30980
tnt expression	The ten-to-the-power function raises the constant, 10, to the power of the computed expression. The range of the argument is approximately from −99 through 99.999. This function executes faster than: 10↑ expression.	5 tnt 2 = 500.00000 tnt (−3) = 0.00100

The math errors and default value associated with the **log** and **ln** (natural log) functions are explained in detail in the next section.

Trigonometric Functions and Statements

The angular units: degrees, radians, or grads, are set by statements explained in this section. Degrees are automatically set when the calculator is switched on, `erase a` is executed, or (RESET) is pressed.

`deg` This statement sets degrees for all calculations which involve angles. A degree is 1/360th of a circle.

`rad` This statement sets radians for all calculations which involve angles. There are 2π radians in a circle.

`grad` This statement sets grads for all calculations which involve angles. A grad is 1/400th of a circle.

`units` This statement displays the current angular units.

Syntax	Description	Examples (fxd 5)

`sin` expression	Determines the sine of the angle represented by the expression in the current angular units.	`deg: sin 45` = 0.70711
		`rad: sin (π/6)`
		= 0.50000
		`grad: sin (-70)`
		= −0.89101

`cos` expression	Determines the cosine of the angle represented by the expression in the current angular units.	`deg: cos 45` = 0.70711
		`rad: cos (π/6)`
		= 0.86603
		`grad: cos (-70)`
		= 0.45399

`tan` expression	Determines the tangent of the angle represented by the expression in the current angular units.	`deg: tan 45` = 1.00000
		`rad: tan (π/4)`
		= 1.00000
		`grad: tan 50`
		= 1.00000

`asn` expression	Returns the principal value of the arcsine of the expression in the current angular units. The range of the argument is −1 through +1. The range of the result is $-\pi/2$ to $+\pi/2$ (radians), −90 to +90 (degrees), or −100 to +100 (grads).	`deg: asn .8` = 53.13010 `rad: asn .8` = 0.92730 `grad: asn .8` = 59.03345

(continued)

Syntax	Description	Examples (fxd 5)
acs expression	Returns the principal value of the arccosine of the expression in the current angular units. The range of the argument is −1 through +1. The range of the result is 0 to π (radians), 0 to 180 (degrees), or 0 to 200 (grads).	deg: acs (−.4) = 113.57818 rad: acs (−.4) = 1.98231 grad: acs (−.4) = 126.19798
atn expression	Calculates the principal value of the arctangent of the expression in the current angular units. The range of the result is $-\pi/2$ to $+\pi/2$ (radians), −90 to +90 (degrees), or −100 to +100 (grads).	deg: atn 20 = 87.13759 rad: atn 20 = 1.52084 grad: atn 20 = 96.81955

Math Errors

Errors 66 through 77 are displayed when a math error occurs. In this section, the default values of math operations which result in an error are explained. Whenever a math error occurs, flag 15 is set automatically. If you set flag 14, math operations which normally cause an error to be displayed, result in a default value.

When printing, displaying, or storing a default value outside the storage range, the value is converted to an appropriate value of ±9.99999999999e 99.

error 66 Division by zero. The default value is +9.99999999999e 511 if the dividend is positive and −9.99999999999e 511 if the dividend is negative. For example:

−9.5/0 = −9.99999999999e 511

A mod B with B equal to zero. The default value is 0. For example:

32 mod 0 = 0

error 67 Square root of a negative number. The default value is $\sqrt{(\text{abs (argument)})}$. For example:

√(−36) = 6.

error 68 Tangent of (n×π/2 radians);
Tangent of (n×90 degrees);
Tangent of (n×100 grads);
where n is an odd integer. The default value is 9.99999999999e 511 if n is positive; and −9.99999999999e 511 if n is negative. For example:

```
radi tan (-π/2) = -9.99999999999e 511
degi tan 270 = 9.99999999999e 511
gradi tan 300 = 9.99999999999e 511
```

error 69 ln or log of a negative number. The default is:
ln (abs (argument)) or log (abs (argument))
respectively. For example:

```
ln (-301) = 5.70711
log (-.001) = -3.00000
```

error 70 ln or log of zero. The default value is −9.99999999999e 511. For example:

```
ln 0 = -9.99999999999e 511
log 0 = -9.99999999999e 511
```

error 71 asn or acs of a number less than −1 or greater than 1. The default value is
asn (sgn (argument)) or acs (sgn (argument))
respectively. For example (in degrees):

```
degi asn (-10) = -90
degi acs (1.5) = 0
```

error 72 Negative base to a non-integer power. The default value is
(abs (base)) ↑ (non-integer power) For example:

```
(-36) ↑ (.5) = 6
```

error 73 Zero to the zero power (0↑0). The default value is 1.

error 74 Storage range overflow. The default value is 9.99999999999e 99 or −9.99999999999e 99. For example:

```
(1e 62) * (1e 38) → A; A will equal 9.99999999999e 99.
(-1e 25) ↑ 5 → B; B will equal -9.99999999999e 99.
```

error 75 Storage range underflow. The default value is zero. For example:

```
(1e-66) * (4e-35) → A; A will equal 0
```

error 76 Calculation range overflow. The default value is 9.99999999999e 511 or −9.99999999999e 511. For example:

```
(1e 99) ↑ 6 = 9.99999999999e 511
(-1e 99) ↑ 7 = -9.99999999999e 511
```

error 77 Calculation range underflow. The default value is zero. For example:

```
(1e-10) ↑ 60 = 0
```

Flags

Flags are programmable indicators that can have a value of one or zero. When a flag is set, its value is one; when it is cleared, its value is zero. There are 16 flags, numbered 0 through 15. The following flags have special meanings:

Flag 13 - is automatically set when CONTINUE is pressed without entering data in an enter statement or when STOP is pressed in an enter statement. Flag 13 is automatically cleared when data is supplied in an enter statement.

Flag 14 - when flag 14 is set, the calculator ignores math errors such as division by zero and supplies a default value shown in the preceding Math Errors list.

Flag 15 - is automatically set whenever a math error occurs, regardless of the setting of flag 14.

The Set Flag Statement

```
sfg [flag number₀...]
```

The set flag (**sfg**) statement sets the value of the specified flags to one. The flag number can be a value or an expression. If a non-integer flag number is specified, the value is rounded to an integer. If sfg is executed with no flag number specified, all flags (0 through 15) are set. For example:

```
sfg 2
```
Set flag 2.

```
 0: sfg A+1
```
Set the flag designated by A + 1.

```
 1: sfg 1,X
```
Set flag 1 and the flag designated by X.

The Clear Flag Statement

cfg [flag number,...]

The clear flag (**cfg**) statement clears the specified flags to zero. The flag number can be a value or expression. If a non-integer flag number is specified, the value is rounded to an integer. If cfg is executed with no flag numbers specified, all flags (0 through 15) are cleared.

Examples:

`cfg 14`	Clear flag 14.
`3: cfg fl*2`	Clear the flag designated by the value of flag 2 (either flag one or flag zero will be cleared).
`4: cfg`	Clears all flags.

The Complement Flag Statement

cmf [flag number,...]

The complement flag (**cmf**) statement changes (toggles) the value of the flags specified. If a set flag is complemented, its new value is zero. If a cleared flag is complemented, its new value is one. A value or expression can be given for the flag number. If a non-integer flag number is specified, the value is rounded to an integer. To complement flags 0 through 15, cmf is executed without paramenters.

Examples:

`cmf 1`	Complement flag 1.
`0: cmf X-1`	Complement the flag designated by X−1.
`1: cmf 3,4,5`	Complement flags 3, 4, and 5.

The Flag Function

```
flg  flag number
```

The flag (**flg**) function is used to check the value of a flag. The result of the flag function is zero or one. One indicates a set flag; zero indicates a cleared flag.

Examples:

```
4: if flg2;jmp 5
```

If flag 2 is set, jump 5 lines.

```
5: flg15→A
```

If flag 15 is set, 1→A; if flag 15 is cleared, 0→A.

Branching Statements

Branching statements are used to alter the sequential flow of a program. Branching is used for such operations as looping through a section of a program, executing a subroutine program, and branching to different parts of a program based on a decision (**if**) statement. There are three statements used for branching: the go to (**gto**) statement, the jump (**jmp**) statement, and the go sub (**gsb**) statement.

The following three types of branching may be used for both go to and go sub statements:

Absolute Branching - branch to the specified line number (such as `gto 10`).

Relative Branching - branch forward or backward in the program the specified number of lines relative to the current line (such as `gsb -3`).

Labelled Branching - branch to the indicated label. This type of branching is generally the most convenient to use since the programmer doesn't have to know line numbers for a branch (such as `gto "First"`).

Line Renumbering

Line numbers are automatically renumbered when a program line is inserted or deleted. As lines are inserted or deleted in a program, the line numbers of relative or absolute go to or go sub statements are changed as required to reflect the insertion or deletion. The address in the jump statement is not changed. The entire program is checked before any deletion is made. If a line being deleted is the destination of a relative or absolute go to or go sub statement, an error is displayed and no deletion occurs, unless an asterisk (*) is used in the delete command.

An error message is not displayed when the line containing a label name in a gto statement is deleted.

If a line becomes too long due to line renumbering, the line number for that line will appear followed by a ? when the line is displayed or listed. For example:

```
8: ...goto 99.
```

Line 8 was stored with 73 characters.

Inserting a line at line 7 causes line 8 to be renumbered such that the branch is to line 100. The line will appear as:

```
9: ?...
```

To view the entire line, delete an appropriate line to recover the original line numbering. The fact that a line is too long to display or list does not affect the operation of the program when the program is run.

More information on line renumbering is in the Program Debugging section.

Labels

Labels are characters within quotes located either at the beginning of a line, after a go to or go sub statement, or after a run or continue command. Labels at the beginning of a line must be followed by a colon.

Labels are used for branching and for remarks within a program. When used for branching, the label in the go to or go sub statement is compared to the line labels in the program until a match is found. Then, at the end of the line, a branch is made to the line containing the label. The first time a branch is made to a label, the program is scanned beginning at line 0 until a matching label is found. From then on, the branch is directly to the line with that label. When comparing labels for branching, a comparison is made on all characters in the label, including blanks.

Labels are often used to make remarks in a program for documentation purposes.
For example:

```
7: "Area":
8: πR↑2→A
```

Note that a colon must follow a label even if nothing else is in the line.

The Go To Statement

The go to (**gto**) statement causes program control to transfer to the location indicated. When a line contains more than one go to statement, only the last one encountered is executed.

Absolute Go To

gto line number

An absolute go to statement is used to branch to the indicated line. The line number must be an integer (such as 5 or 13).

When an absolute or labelled go to statement is executed from the keyboard in calculator mode, the program line counter is set to the specified line number. To view the line, press the ↓ key.

Relative Go To

gto + number of lines
gto − number of lines

A relative go to statement is used to branch forward (+) or backward (−) the specified number of lines, relative to the current line. The number of lines must be an integer.

Examples:

```
20: gto +1
21: gto -3
22: gto +0
23: gto -0
```

Go forward 1 line.

Go back 3 lines.

Go to the beginning of the current line.

Labelled Go To

gto label

A labelled go to statement is used to branch to the line with the indicated label (see section on labels). This is the most convenient type of branching since no line numbers have to be considered.

Example:

```
gto "Avg."
```

Go to the line labelled by "Avg.".

When a labelled go to statement is executed from the keyboard in calculator mode, the program line counter is set to the specified line number. To view the line, press the ↑ key.

Multiple go to statements in a line are useful for N-way branching when used with an if statement. N-way branching is explained later.

The Jump Statement

jmp number of lines

The jump (**jmp**) statement allows branching from the current line by the number of lines specified. This statement is similar to the relative go to statement except that the number of lines can be an expression. If the number of lines is positive, the branch is forward in the program. If the number of lines is zero, the branch is to the beginning of the current line. If the number of lines is negative, the branch is backward in the program. If the number of lines is not an integer, then it is rounded to an integer.

The go to statement executes faster than the jump statement. The jump statement can only be at the end of a line, otherwise error 07 is displayed when you try to store or execute the line.

Examples:

```
10: jmp 10
```

Jump forward 10 lines.

```
19: jmp A
```

Jump the number of lines designated by the value of A.

```
28: jmp (Z=2)2
```

Jump forward 2 lines if Z=2; otherwise jump to the beginning of the current line.

```
33:    ... ;jmp(B+
   1→B)>20
```

Increment B and jump to the next line if B is greater than 20; otherwise jump to the beginning of the current line.

The Go To Subroutine and Return Statements

The go to subroutine (**gsb**) statement allows branching to subroutine portions of a program. Subroutines are useful when the same routine will be executed many times and called from different places in the program. A return pointer is set up when the go sub statement is executed. This pointer points to the next line after the line containing the go sub statement. The return (**ret**) statement returns the program execution to the pointer location. The return statement is the last statement executed in the subroutine and must be the last statement in a line. The depth of subroutine nesting is limited only by the amount of available memory. Each subroutine return pointer requires eight bytes of memory. Subroutines should be entered only by a **gsb** statement and should be exited only by a **ret** statement.

When a line contains more than one go sub statement, only the last one encountered is executed. There are three types of go sub statements: absolute, relative, and labelled.

Absolute Go Sub

gsb line number

An absolute go sub statement is used to go to the subroutine at the specified line number. The line number must be an integer.

Example:

```
7: gsb 15
```

Go to the subroutine at line 15.

```
18: ret
```

End subroutine with return statement (program returns to line 8).

Relative Go Sub

gsb + number of lines
gsb − number of lines

A relative go sub statement provides forward (+) or backward (−) subroutine branching the specified number of lines, relative to the current line number. The number of lines must be an integer.

Examples:

```
 7: gsb +5
```

Go to the subroutine at line 12.

```
 8: gsb -3
```

Go to the subroutine at line 5.

Labelled Go Sub

gsb label

A labelled go sub statement is used to branch to the subroutine at the indicated label. This is the most convenient form of subroutine branching since no line numbers need to be considered.

Example:

```
 3: gsb "sub1"
```

Go to the subroutine at the line labelled by "sub1".

Multiple go sub statements in a line are useful for N-way branching when used with the if statement. N-way branching is explained later.

Calculated Gosub Branching

By using the jump statement and the go sub statement together, calculated branching to subroutines is possible. This form of subroutine branching is called the calculated go sub and has the form:

gsb dummy location [jmp expression

The dummy location can be a line number, + or − a number of lines, or a label, but the calculator branches to the subroutine designated by the computed jump expression. For example:

```
0: ent N
1: gsb "X";jmp N
2: prt "end"
3: "X":end
4: prt "sub1";
 ret
5: prt "sub2";
 ret
6: prt "sub3";
 ret
```

If a 3 is entered for N, the program branches to the subroutine at line 4.

The If Statement

if expression

The if statement is used to branch based on a logical decision. When an if statement is encountered, the expression following it is evaluated. If the computed expression is zero (false), program control resumes at the next program line (unless the preceding statement was a go to or go sub statement as explained later under N-Way Branching). If the computed expression is any other value, it is considered true, and the program continues in the same line. The if statement is most often used with expressions containing relational operators or flags.

Example:

```
0: ent A,B
1: if A=B;gto
 "one"
2: gto "zero"
3: "one":dsp
 "A=B"
4: stp
5: "zero":dsp
 "A#B"
6: end
```

Enter a value for A and B.

If A=B, go to "one"; otherwise go to 'zero".

At label "one", display A=B; then stop.

At label "zero", display A#B; then end the program.

Whenever A and B are equal, A = B is displayed. All other times, A # B is displayed.

The if statement can be used with other statements besides the go to statement used in the above example. The previous example could be shortened to:

```
0: ent A,B
1: if A=B;dsp
 "A=B";stp
2: dsp "A#B"
3: end
```

Note that no go to statements are used.

N-Way Branching

The if statement used with a go to or go sub statement makes it possible to branch to any of several locations. This type of branching is referred to as n-way branching, and has the following forms:

```
gto...; if...; gto...
```

or

```
gsb...; if...; gsb...
```

If the first if statement is false, then the branch is determined by the first go to or go sub statement. If the first if statement is true, the second go to or go sub statement determines the branch. Go to and go sub statements can be mixed in the same line.

When a line contains more than one go to or go sub statement, only the last one encountered is used. An if statement whose expression is zero can abort execution of the remainder of a line (before subsequent go to or go sub statements are encountered).

Example:

```
20: gto 24;if
 X>30;gto 32;if
 X>40;gto "max"
21:
```

If X is less than or equal to 30, the program branches to line 24. If X is greater than 30 and less than or equal to 40, the branch is to line 32. If X is greater than 40, the branch is to the line labelled "max".

The Dimension Statement

dim item [, item, ...]

item may be: simple variable
array variable [dimension [, dimension, ...]]

The dimension (**dim**) statement reserves memory for simple and array variables, and initializes the indicated variables to zero. r-variables can not be dimensioned in a dimension statement.

In the dimension statement, the dimensions of an array can be specified by expressions. For example:

```
0: ent N,I,r2
1: dim A[N,I],
   B[r2],C[3,2*N]
```

Variables are used to specify dimensions.

Variables are allocated in the order that they appear. If a variable is allocated already, an error results. All the variables dimensioned in any one dimension statement are stored in a contiguous block of memory. This is important when recording data.

Dimension statements may appear anywhere in a program but any dimension statement can only be executed once during a program. The number of dimension statements is limited by memory size. The number of dimensions and the size of the dimensions of an array is limited only by memory size and line length. For example:

```
0: dim N[2,2,2,
  2,2,2,2]
1: dim M[1000]
```

Reserves 128 array elements.

Reserves 1000 array elements.

Specifying Bounds for Dimensions

A dimension may be specified by giving lower and upper bounds. The lower bound must be specified before the upper bound. The two are separated by a colon. The bounds must be in the range from −32767 through 32767. For example:

```
0: dim S[-3:0,
  4:6]
```

Reserves 12 array elements.

This statement reserves the same amount of memory as:

```
0: dim X[4,3]
```

Reserves 12 array elements.

The elements of array S are referenced as:

S[−3,4]	S[−3,5]	S[−3,6]
S[−2,4]	S[−2,5]	S[−2,6]
S[−1,4]	S[−1,5]	S[−1,6]
S[0,4]	S[0,5]	S[0,6]

If a lower bound is not specified, as in X[4,3], it is assumed to be 1, the same as X[1:4,1:3].

The Clear Simple Variables Statement

```
csv
```

The clear simple variables (**csv**) statement clears any allocated simple variables to zero. The clear simple variables statement does not de-allocate variables. Therefore, an error results when the following line is executed:

```
0: 7→A!csv!dim A
```
Not allowed. Cannot allocate A twice.

The List Statement

```
list [beginning line number [, ending line number ]]
list special function key
list k
```

The list statement is used to obtain a printed listing of a stored program, section of a program, or special function keys. If no parameter follows the list statement, the entire program is listed. If one line number is specified, the program is listed from that line to the end. If two line numbers are specified, the program segment between the two line numbers is listed. To list all of the special function keys, execute `list k` (for list keys). When list is followed by pressing an individual special function key, then only that key is listed (this is not programmable). The list statement must be the last statement in a line.

Examples:

`0: list`	Lists the entire program.
`list 10,15`	List lines 10 through 15.
`list 4,4`	List line 4.
`list k`	List the special function keys.
`list` f10	List special function key f10 (not programmable).

At the end of a listing, a checksum is printed. This checksum is useful for detecting interchanged or omitted lines and characters. Any difference in the programs generates a different checksum. In the following two programs, only the characters `rt` in line 1 are interchanged. Note that the checksums are different. There is no change in checksum from machine to machine, with different memory sizes, nor with different ROMs.

```
0: ent N
1: prt "sqrt.
 of",N
2: prt "is",ΓN
3: end
*23740
```

```
0: ent N
1: prt "sqtr.
 of",N
2: prt "is",ΓN
3: end
*23736
```

Different Checksums

Used and Remaining Memory

After a list operation, two numbers are displayed. The first number is the total length of the program in bytes*. This number doesn't include variables, subroutine return pointers, etc. The second number is the unused memory in bytes. For example:

```
468                6246
```

Program Length (in bytes) Unused Memory

*A byte is the basic unit of data in the 9825. 8 bits make up one byte. 8 bytes are required to store a number.

Program Debugging

Debugging is the process of refining a program by editing, correcting, and updating. Like programming, it is a creative process. Many operations are involved such as deleting and inserting lines and changing, inserting, and deleting characters. Selective tracing and selective stopping are useful for locating lines which require changes. [STEP] is useful for going through a program one line at a time. This chapter explains some of the steps in editing a program.

Finding the Problem

The first step in debugging is to find the lines which require changes. This can be done in several ways. One way is to step through a program by pressing [STEP] once for each line to be executed. Then check the results after each executed program line.

Another way is to use the trace, stop, and normal statements. When program lines are traced, the line number, and variables and flags which are assigned values are printed. This allows you to monitor program activity in individual program lines. Using the stop statement, the program can be stopped whenever a specified program line is encountered. The normal statement is used to terminate tracing and stopping. Stop, trace,and normal statements are explained later.

Fixing the Problem

The next step in debugging is fixing the problem. In many cases, this is as simple as changing one character. Fixing the problem could, however, require rewriting many program lines.

To modify characters within a line, fetch the line by pressing the [FETCH] key followed by the line number of the line requiring the change. Then press [ENTER]. The line will appear in the display. Next press either [BACK], if the change is closer to the end of the display, or [FWD], if the change is closer to the front. Once a flashing cursor is over the location needing correction, you can either insert characters, delete characters, or write over the existing characters. To insert characters, press the [INS·RPL] key. This changes the flashing ▒ to a flashing ◂. Characters that are typed-in are inserted at the left of this cursor. To delete characters, press character [DELETE] for each character to be deleted. To replace characters, be sure the ▒ cursor is in the display (if the ◂ is in the display, press [INS·RPL] to get ▒) and then enter the necessary characters.

To modify lines within a program, use the [FETCH] key or the [↑] and [↓] keys to bring the line into the display. To delete the line, press the line [DELETE] key.

If a line being deleted has a line number referenced by a go to or go sub statement, an error 36 will occur. Either execute the delete command with the optional asterisk (*) parameter or adjust the line reference in the go to or go sub statement accessing that line. In the following example, line 25 is to be deleted; but line 25 is referenced from line 27. Two alternatives are shown.

Program section:

```
25: prt "number"
 ,N
26: if N=27;stp
27: N+1→N;gto 25
```

Alternative 1:

Type in: `del 25, *`

Press: [ENTER]

Deletes line 25 only. The go to statement in line 27 still addresses line 25.

Alternative 2:

Change line 27 to:

```
27: N+1→N;gto 26
```

Then fetch line 25 and press line [DELETE],
or execute `del 25`.

To insert a line, fetch the line that the inserted line is to precede. Then type the new line into the display and press the line [INSERT] key to store it. All the lines from the fetched line on are automatically renumbered (incremented by one). When a line is inserted, the line references of go to or go sub statements are incremented to reflect the new line numbering. If the line being inserted contains an absolute go to or go sub statement, it is assumed that the line numbers reference the lines before they are renumbered. Thus, if a line inserted before line 30 contains a `gto 45` statement, it will be renumbered to `gto 46`. (The old line 45 is renumbered to line 46.)

In this example, a line is inserted between lines 14 and 15 ▶

```
14: prt "number
 of days",D
15: gto 19
```

First, fetch line 15, then type the line to be inserted into the display ▶

```
prt "number of weeks";W
```

Then press the line (INSERT) key. The display will be ▶

```
15▷
```

To see where the line was inserted, execute: `list 14,16`

```
 14: prt "number
  of days",D
 15: prt "number
  of weeks",W
 16: gto 20
```

Note that the line number in the go to statement in line 16 is incremented since old line 19 is now line 20.

The branching address of the jump statement is not affected by adding or deleting lines in a program.

The Debugging Statements

The trace, stop, and normal statements are used for debugging programs. The three statements have dual roles in that their action depends upon whether any parameters are specified.

To effectively use the trace, stop, and normal statements, the internal operation should be understood. There is one master flag which enables and disables overall tracing and stopping. In addition, each line has two flags. The trace flag enables and disables tracing of the line. The stop flag enables and disables selective stopping at a line. These flags are unrelated to flags 0 through 15 explained earlier in this chapter.

The Trace Statement

trc [beginning line number [, ending line number]]

The trace (**trc**) statement sets the master trace flag. If line numbers are specified in the trace statement, then the individual line trace flags are set on the designated lines. One line number specifies that line only and two line numbers specify the block of lines from the beginning line number through the ending line number.

During the execution of the program, a specific line is traced if both the master trace flag and the individual line trace flags are set. When a line is traced, the number of the line is printed as well as information describing any variable assignments and flag operations (involving flags 0 through 15).

The Stop Statement

stp beginning line number [, ending line number]

The stop (**stp**) statement with line numbers sets the master trace flag and stop flags on the designated lines.

Before each program line is executed, the stop flag for that line is checked. If this flag and the master trace flag are set, the program is stopped before the line is executed. The number of the current program line is displayed when the program is stopped. Execution of the program will continue from this line if CONTINUE or STEP is pressed (see description of CONTINUE and STEP keys).

The Normal Statement

nor [beginning line number [, ending line number]]

The normal (**nor**) statement clears the trace and stop flags of the lines specified by the line numbers. If no line numbers are specified, the normal statement clears the master trace flag.

The use of a master trace flag in addition to individual line trace and stop flags makes it easy to enable or disable selective tracing or stopping of parts of a program. This process is shown in the following example.

A 100 line program contains three sections in which critical operations are performed. These sections can be traced by executing the following statements:

```
trc 5,15
trc 40,50
trc 70,85
```

The program is run and the tracing printout indicates that line 45 contains an error. The line is modified and `nor` is executed to clear the master trace flag. The program is again run, but this time the assignments are not printed. At the conclusion of the program it becomes obvious that the program still contains an error. The three critical sections of the program are again traced by executing `trc`. This sets the master trace flag so that the lines 5-15, 40-50, and 70-85 are traced (the trace bits are still set on these lines). After the program is totally debugged, the individual line trace flags are cleared by executing `nor 0,9999`.

The individual line trace and stop flags are not normally stored on the cartridge when a program is recorded by the record file statement. These flags can be recorded on the tape cartridge along with the program by including the optional debug ("DB") parameter in the record file statement. The master trace flag is not recorded. To have the program automatically trace the lines when the program is loaded back into the calculator, put trc in line 0 to set the master flag.

Programming Hints

There are usually several ways to write a program or section of a program to perform a specific job, and the programmer is often faced with the choice of which of several methods to use. Usually the goal is to save program space and execution time and at the same time maintain readability. However, these goals are sometimes conflicting and the programmer must decide which is the overriding concern.

This appendix is not intended to discuss programming techniques in general but to describe a collection of hints for the programmer who wishes to save space or time. While by no means complete, this list describes some of the trade-offs which are "machine dependent" and therefore not necessarily obvious.

In most cases, the time savings are small and are not observable unless the statement is executed thousands of times. The space savings usually only amount to a few bytes. To check the amount of space used by a statement, execute `list -l` after storing the statement.

Method A	Method B	Method Requiring Less Program Storage	Method With Faster Execution Time
Simple Variables	r-variables	A	A
r-variables	one-dimensional array variables	Same	A
Multiple statements per line	One statement per line	A	A
gto +5	gto 5	Same	Same
gto −5	gto 5	Same	Same
gto "5" (one or two character label)	gto 5	Same	Same
gto +5	jmp 5 (Note 1)	B	A
$\sqrt{X}$	X↑.5	A	A
XX	X↑2 (Note 2)	A	Same

Method A	Method B	Method Requiring Less Program Storage	Method With Faster Execution Time
implied multiply	explicit multiply	Same	Same
π	3.14159...	A	A
if flg 2=1	if flg 2	B	B
if flg 2=0	if not flg 2	B	B
if A#0	if A	B	B
if (A<B) or (B<C)	if (A<B) + (B<C)	Same	A
if (A<B) and (B<C)	if (A<B) * (B<C)	Same	A
J+5→K; K−3→L	(J+5→K)−3→L	B	B
J+1→J; if J<5	if (J+1→J)<5	B	B
Specify lower bounds for array dimensions.	Use default lower bounds.	B	Same
Use simple variable as a flag (as 1→A).	flag	(Note 3)	B
Using both tracks alternately.	Using one track at a time, sequentially.	Same	A

Note 1: For computed branching, only jump statement can be used.

Note 2: X↑Y is done by repeated multiplication if Y is an integer.

Note 3: If only one test is made, the flag method takes less room. If two tests are made, both methods are the same. For more than two tests, the simple variable method takes less room.

Notes

Chapter 4
Table of Contents

Notes

Chapter 4

Advanced Programming

The Advanced Programming statements and functions enables you to –

- Use for/next loops to repeat sections of a program
- Pass parameters to subprograms including subroutines and functions.
- Store numbers in split and integer precision formats to conserve memory.
- Generate a list of the variables used in a program and the line numbers in which they occur using the cross reference statement.

Advanced Programming (AP) is available in a plug-in ROM card for the 9825A and S Computers. The ROM card uses four bytes of read/write memory. AP is a permenent part of the 9825B Computer.

For/Next Loops

The for and next statements enable you to repeat a group of statements within a program as many times as necessary.

```
for simple variable = initial value to final value [by step size value]
  •
  •
  •
  •
next same simple variable
```

The for and next statements, including the statements between them, form a loop within a program. The for statement defines the beginning of the loop and the number of times the loop is to be performed. The variable that follows the for and next statements can be any one of the simple variables A through Z.

The initial, final and step size values can be expressions. If the step size value is not specified, the default value is 1.

Here's an example of a for/next loop—

```
•
•
5: for I=1 to 5
•
•
•
•
10: next I
•
•
```

This for/next loop would be executed five times - when I = 1,2,3,4 and 5. Each time the next statement is executed, the value of I is incremented by one, the default step size value. When the value of I exceeds the final value (when I = 6)*, the loop is finished and the program continues at the statement following the next statement.

The advantages of using for/next looping instead of an if statement are shown in the following examples where the numbers 1 through 10000 are displayed in succession.

if statement

```
0: 1→I
1: dsp I
2: if (I+1→I)<=1
 0000;goto 1
3: beep
4: end
```

for/next loop

```
0: for I=1 to
 10000
1: dsp I
2: next I
3: beep
4: end
```

The program that uses the for/next loop is easier to key in, takes less calculator memory (40 bytes) and is executed faster (25 seconds). With the if statement, the program uses 48 bytes of memory and is executed in 32 seconds.

The initial value of the variable assigned in the for/next loop does not have to be 1. The following example totals the integers, 90 through 100, and prints the total (1045).

```
0: 0→A
1: for I=90 to
 100
2: I+A→A
3: next I
4: prt "Total=";
 A
5: end
```

```
Total=    1045.00
```

* This is an often overlooked aspect of for/next loops and is covered on the next page.

The next example illustrates that variables can be used in the for statement. The variables B and C are assigned values in the enter statement in line 1 and are used in the for statement in line 3.

```
0: 0→A
1: ent B,C
2: prt "B=",B,
 "C=",C
3: for I=B to C
4: I+A→A
5: next I
6: prt "Total=",
 A
7: end
```

```
B=              1.00
C=              3.00
Total=          6.00

B=              5.50
C=              8.50
Total=         28.00
```

If B = 1 and C = 3, the total of 1,2 and 3 (6) is printed. If B = 5.5 and C = 8.5, the total of 5.5, 6.5, 7.5 and 8.5 (28) is printed. In either case, the value of I is incremented by one after each loop. If the value of B is greater than the value of C, the loop is not executed and the program continues at the first statement following next I, in this example the print statement in line 6.

The following example illustrates an often overlooked aspect of for/next looping. After each loop is performed, the next statement increments the value of I by 1. Then the incremented value is compared with the final value. If the incremented value is not greater than the final value, the loop is repeated. When the incremented value is greater than the final value (when I = 11) the loop is no longer repeated and the statement following the next statement (spc) is executed.* Although the final loop is performed when I = 10, the last incremented value for I is 11 and the calculator retains this as the value of I.

```
0: for I=1 to 10
1: prt I
2: next I:spc
3: prt I
4: end
```

```
             1.00
             2.00
             3.00
             4.00
             5.00
             6.00
             7.00
             8.00
             9.00
            10.00

            11.00
```

*Statements following a next statement are not executed until the entire loop is completed. If a gto or gsb statement precedes a next statement on the same line, the gsb or gto isn't executed until the loop is completed.

The next program shows how the for/next loop can be used to assign values to arrays. In this example, the array variables A[1] through A[4] are assigned values.

```
0: dim A[4]
1: for I=1 to 4
2: I↑2→A[I]
3: prt I,A[I];
 spc
4: next I
5: end
```

```
 1.00
 1.00

 2.00
 4.00

 3.00
 9.00

 4.00
16.00
```

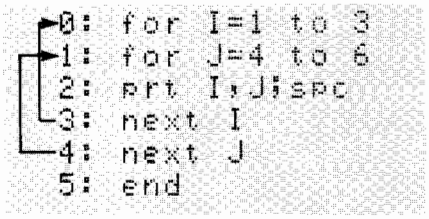
For/next loops can be nested or located inside one another up to a depth of 26 (one for each simple variable A through Z). However, one loop cannot overlap another. Before running the following programs, set the print all mode by pressing PRT ALL.

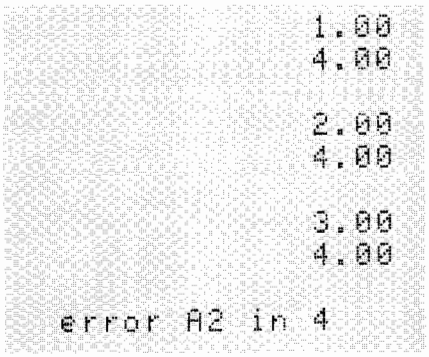
Correct Nesting

```
0: for I=1 to 3
1: for J=4 to 6
2: prt I,J;spc
3: next J
4: next I
5: end
```

```
 1.00
 4.00

 1.00
 5.00

 1.00
 6.00

 2.00
 4.00

 2.00
 5.00

 2.00
 6.00

 3.00
 4.00

 3.00
 5.00

 3.00
 6.00
```

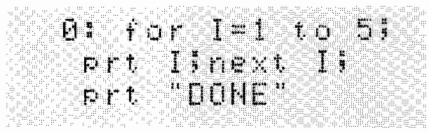
Incorrect Nesting





In the incorrect nesting example, the I loop is activated first and then the J loop is activated. The J loop is cancelled at the same time that `next I` is executed because it's an "inner loop". When the I loop is completed and `next J` is finally accessed, `error A2` is displayed. This is because the J loop was cancelled and was not reactivated after the last I loop.

For/next loops can be written in more than one line, as previously shown, or all in one line, like this–



When line 0 is executed, the numbers 1 through 5 are printed as I is incremented by one. When the final value of I is reached, the last statement in the line is executed and `DONE` is printed.

If (STOP) is pressed while the program is running, the program halts when the current line is completely executed. If a for/next loop is completely contained in one line and (STOP) is pressed, the calculator will not stop until the loop is completed. Only (RESET) can stop the execution of the line containing the loop, before its normal termination. This can be avoided by putting the `for` and `next` statements on separate lines.

Each `for` statement can have only one associated `next` statement. When a `for` statement is executed, and there is already an active loop using the same simple variable, then the previous loop is cancelled and the new loop becomes active. In the following example, the first I loop (in line 0) is activated and then cancelled when the second I loop is activated in line 2. When line 4 is executed, control returns to the latest I loop (in line 2).

```
0: for I=1 to
 100
1: prt "I1=";I
2: for I=2 to 10
3: prt "I2=";I
4: next I
```

```
I1=          1.00
I2=          2.00
I2=          3.00
I2=          4.00
I2=          5.00
I2=          6.00
I2=          7.00
I2=          8.00
I2=          9.00
I2=         10.00
```

The optional step size value enables you to specify a step size other than 1, the default step size value. For example–

```
0: for I=0 to
 50 by 10
1: prt I
2: next I
```

```
          0.00
         10.00
         20.00
         30.00
         40.00
         50.00
```

By adding the optional step size value to the `for` statement, the simple variable will be incremented by that value each time the `next` statement is executed. In the previous example, the loop is executed six times – when I = 0,10,20,30,40 and 50. As soon as the incremented value is greater than the final value, the loop is exited.

For/next loops can be decremented by using negative values for the optional step size value. For example–

```
0: for I=50 to
 0 by -10
1: prt I
2: next I
```

```
         50.00
         40.00
         30.00
         20.00
         10.00
          0.00
```

The step size value does not have to be an integer; fractional numbers are allowed. For example–

```
for I = 1 to 10 by .5
```

The initial value, the final value and the step size value can be variables or expressions. For example–

```
5: for I=A to B
 by (B-A)/100
•
•
•
10: next I
```

Once the for statement is executed, the initial, final and step size values can be changed without affecting the number of times the loop is repeated. In the following example, the variables A and B can be used within the loop for other purposes, but the loop itself is repeated only six times.

```
0: 1→A;6→B
1: for I=A to B
2: A-1→A
3: B+1→B
4: prt I,A,B;
 spc
5: next I
6: end
```

```
 1.00
 0.00
 7.00

 2.00
-1.00
 8.00

 3.00
-2.00
 9.00

 4.00
-3.00
10.00

 5.00
-4.00
11.00

 6.00
-5.00
12.00
```

Subprograms

A subprogram is a programming routine that enables you to repeat an operation many times substituting different values each time the subprogram is called. There are two types of subprograms – subroutines* and functions.

Subroutines

A subroutine subprogram consists of one or more lines of programming which perform a specific task. A subroutine is accessed using a call (`cll`) statement followed by the name of the subroutine, enclosed in single quotes (apostrophes). As many parameters as needed can be used, within the limits of line length.

```
cll "name" [(parameter 1[, parameter 2, ...])]
  •
  •
  •
"name":
  •
  •
  •
ret
```

The first statement in the subprogram is its name, written as a label (enclosed in quotation marks and followed by a colon). The last statement executed in a subprogram is always a return (`ret`) statement.

*Subroutine subprograms are similar to standard subroutines called by the gosub (`gsb`) statement within a mainframe program. To eliminate confusing the two, subroutine subprograms will be referred to as subroutine subprograms and standard subroutines will be referred to as mainframe subroutines in this chapter.

Here's a program with a mainframe subroutine which prints the sum of two numbers–

```
0: 1→A;2→B
1: esb "name";
   prt "DONE"
2: end
3: "name":
4: prt A+B
5: ret
```

And here's a program that uses a subroutine subprogram to do the same–

```
0: 1→A;2→B
1: cll 'name';
   prt "DONE"
2: end
3: "name":
4: prt A+B
5: ret
```

A look at both programs shows that the subroutines are identical, but the calling statements are different. A esb statement, followed by the name of the subroutine enclosed in quotes, is used to access the mainframe subroutine, while a cll statement, followed by the name of the subprogram enclosed in apostrophes, is used to access the subroutine subprogram.

There's another difference between the two. The subroutine subprogram is executed immediately, but execution of the mainframe subroutine is delayed until all other statements in that line are executed, as shown by the following printouts.

Mainframe Subroutine

```
DONE
                3.00
```

Subroutine Subprogram

```
                3.00
DONE
```

With the mainframe subroutine, DONE is printed before the routine is accessed and executed and program control returns to the line following the one containing the esb statement.

The subroutine subprogram is accessed and executed immediately so the sum is printed first. Program control then returns to the statement following the call statement and DONE is printed.

In addition to the immediate execute feature, the call statement can pass parameters to the subroutine. In a subprogram, parameters are represented by p-numbers (parameter numbers). This enables you to call the subprogram repeatedly using different values for the parameters each time. Here's an example of this based on the previous two programs–

```
0: ent A,B
1: cll 'name'(A,
 B);prt "DONE"
2: end
3: "name":prt
 p1+p2
4: ret
```

```
1: cll 'name'(A,B)
•
•
•
3: "name":prt p1+p2
```

Passing Parameters

Before covering functions, here's some general information about parameters. A detailed explanation of parameters (p-numbers) is found later.

Parameters that follow the call statement are always enclosed in parentheses and as many parameters as the length of the line allows can be used. These parameters can be constants, simple variables, expressions, r-variables or single elements of an array; entire arrays, strings, string arrays and text cannot be used as parameters. In the preceding example, p1 and p2 in line 3 correspond to parameters A and B.

Parameters can be passed back from subroutines to main programs by assigning a value to a p-number which corresponds to a variable. For example, lines 1 and 3 in the previous program can be changed to–

```
1: cll 'name'(A,
 B,C);prt C
•
•
•
3: "name":p1+
 p2→p3
```

Subprograms can be nested (called by another subprogram) as deeply as the calculator memory allows. Each call statement requires a minimum of 26 bytes of memory when executed. That memory is returned when `ret` is executed. If parameters are passed, additional memory is required.

Functions

A function subprogram consists of one or more lines of programming which perform a specific task. A function is accessed using the name of the function enclosed in single quotes (apostrophes) within an expression or statement in the program. As many parameters as needed can be used, within the limits of line length.

```
'name' [(parameter 1[; parameter 2; ...])]
   •
   •
   •
"name":
   •
   •
   •
ret parameter
```

The first statement in the function itself is its name, written as a label (enclosed in quotation marks and followed by a colon). The last statement executed in a function is always `ret` followed by a return parameter. The return parameter, like a parameter that follows call statements, can be a simple variable, a constant, an expression, an r-variable or an element of an array. In addition, a return parameter can be an array, a string, a string array or text.

Here's an example of a function based on the previous programs—

```
0: 1→A;2→B
1: prt 'name';
   prt "DONE"
2: end
3: "name":
4: ret A+B
```

```
                    3.00
DONE
```

When the program is run, the function is accessed as line 1 is executed. The result of the function is automatically returned and substituted for the name of the function in the statement (`prt 'name'`). This causes the value of A + B to be printed.

Like a subroutine, a function is executed immediately and program control returns to the function (`'name'`). A function subprogram can be used in a program wherever an expression can be used.

A parameter which follows a function call can be a simple variable, a constant, an r-variable, an expression or a single element of an array. (Entire arrays, strings, string arrays and text can't be parameters in a function call.) Parameters following a function call are always enclosed in parentheses and as many parameters as the length of the line allows can be used.

Here's an example of a function that uses parameters–

```
0: ent A,B
1: prt 'name'(A,
 B);prt "DONE"
2: end
3: "name":
4: ret p1+p2
```

If the return parameter is omitted from a function subprogram, `error 84` results; if a return parameter follows `ret` in a subroutine subprogram or a mainframe subroutine, it's ignored and no error is displayed.

Functions, like subroutines can be nested as deeply as the calculator memory allows. Each function call requires a minimum of 26 bytes of memory when executed. That memory is returned when `ret` is executed. If parameters are passed, additional memory is required.

A function subprogram can be used within another subprogram or within an expression. When the function call is placed in the expression, the value returned by the function is used directly in the expression.

Here's an example of a function subprogram that computes the factorial of a number (lines 7 and 8) and uses it in the calculation in line 4 to find the number of combinations of N items taken R at a time.

```
0: fxd 0
1: ent "No. of
 items?",N
2: ent "No. take
 n at a time?",R
3: prt "Combinat
 ions of",N,"ite
 ms taken",R,
 "at a time="
4: prt '!'(N)/
 ('!'(R)*'!'(N-
 R));spc 3
5: end
6: "!":
7: 0→p2;1→p3
8: if p1#p2;p2+
 1→p2;p2p3→p3;
 jmp 0
9: ret p3
```

For 12 items taken 3 at a time the number of combinations is –

```
Combinations of
                12
items taken      3
at a time=
               220
```

P-Numbers

A subprogram (subroutine or function) enables you to repeat an operation using different values each time the subprogram is called. This is accomplished by following the subprogram call with a list of parameters. When these parameters are passed to the subprogram, a parameter number or p-number is assigned to each parameter in the list. The p-numbers are assigned to the parameters consecutively, starting with p1. The subprogram operation is then performed using the values passed by the subprogram call.

In addition to passed parameters, there are local p-numbers. When allocated, a local p-number is initialized to zero. Local p-numbers are used in a subprogram as needed. Here's an example that uses passed parameters and a local p-number.

```
0: ent A,B
1: prt 'name'(A,
 B)
2: end
3: "name":p1p1+
 p2p2+p6;ret rp6
```

When this program is run, p1 and p2 correspond to the passed parameters A and B, but p6 is a local p-number which, when allocated, is initialized to zero. When the subprogram operation is performed using p1 and p2, the result of the function (rp6) is returned and printed.

P-numbers are assigned to parameters consecutively, starting with p1. If you use a local p-number that doesn't follow the passed p-numbers in consecutive order, all p-numbers in between are automatically allocated as local p-numbers. When allocated, these p-numbers are initialized to zero. In the previous example, p3, p4 and p5 are initialized when p6 is allocated and require memory space, even though they are not used.

P0 is also a local p-number but it isn't initialized to zero. Instead, when the subprogram is called, p0 is initialized to the number of parameters passed to the subprogram.

Subprograms can be nested (called by another subprogram) as deeply as the calculator memory allows. In addition, a function subprogram can be used as the parameter for another subprogram (function or subroutine) like this—

```
•
•
20: cll 'SUB'('F
 UN'(A,B))
•
•
```

In the line above, A and B are parameters for the function 'FUN' and the result of the function is the parameter for the subroutine 'SUB'.

When subprograms having parameters are nested, each set of p-numbers is independent of the p-numbers in the next subprogram or level, even though the same p-numbers may be used in each. To illustrate independent p-numbers in nested subprograms, the following example converts a Fahrenheit temperature to Celsius and then outputs both temperatures. Notice that each subprogram uses p1 without affecting the value of the other.

```
0: fxd 0
1: "L":ent "Temp
 erature(F)?",T
2: cll 'Output'(
 T,'C'(T));gto
 "L"
3: "Output":
4: prt "Fahrenhe
 it=",p1,"Celsiu
 s=",p2
5: spc ;ret
6: "C":
7: p1-32→p2
8: ret 5p2/9
```

When the trace mode is established (`trc 0,8`) to monitor the activity of the running program, value assignments for each p-number used are not printed as they are for each simple variable. Instead, as in line 7 of the following traced printout, all p-numbers are referenced by `p=` without indicating the specific p-number.

```
0:
1:
T=              32
2:
6:
7:
p=               0
8:
2:
3:
4:
Fahrenheit=     32
Celsius=         0
5:

2:
1:
```

If a p-number is used as a parameter in a nested subprogram call, there may be some interaction between the p-numbers used in each subprogram. The following program uses nested subprogram calls with parameters to illustrate what happens to p-numbers, variables, expressions and constants in a parameter list when their values are changed in a subprogram.

```
0: fxd 0;2→A
1: cll 'Sub-1'(A
 ,5,1*A)
2: prt "Main",
 "A=",A;stp
```

```
3: "Sub-1":cll
 'Sub-2'(A,p1,
 p0,5,1*A)
4: 2p1→p1
5: 2p2→p2;2p3→p3
6: prt "Sub-1",
 p1,p2,p3;spc ;
 ret
```

```
7: "Sub-2":
8: 3p1→p1
9: 3p2→p2;3p3→p3
 ;3p4→p4;3p5→p5
10: prt "Sub-2",
 p1,p2,p3,p4,p5;
 spc ;ret
```

The main program (lines 0 through 2) contains the call for Sub-1 with three parameters – A, 5 and 1×A. Sub-1 (lines 3 through 6) calls Sub-2 which has five parameters – A, p1, p0, 5 and 1×A. Sub-2 (lines 7 through 10) triples the value of each parameter and then prints the values. Program control returns to line 4 (Sub-1) and the current value of each parameter is doubled and printed.

Here's a chart that shows the values of the parameters during program execution. The shaded chart below duplicates the chart at the top and shows values before Sub-2 is called.

Sub-1

Passed Parameters	Initial Values	Corresponding p-numbers
A	2	p1
5	5	p2
1×A	2	p3

Sub-2

Passed Parameters	Initial Values	Corresponding p-numbers	Values after line 8	Values after line 9
A	2	p1	6	18*
p1	2	p2	6*	18
p0	3	p3	3	9
5	5	p4	5	15
1×A	2	p5	2	6

```
Sub-2          18
               18
                9
               15
                6
```

*Since A and p1 (in Sub-1) and p1 and p2 (in Sub-2) are all different names for the same value, when p1 (in Sub-2) is tripled in line 8, A and p1 (in Sub-1) and p2 (in Sub-2) are also tripled. The same is true in line 9 when p2 is tripled.

Sub-1 (Results before calling Sub-2)

Passed Parameters	Initial Values	Corresponding p-numbers
A	2	p1
5	5	p2
1×A	2	p3

Results after return from Sub-2

Values after Sub-2 execution	Values after line 4	Values after line 5
18	36	36
5	5	10
2	2	4

```
Sub-1          36
               10
                4
```

When program control returns to the main program, the final value of A is printed.

```
Main
A=          36
```

Although p-numbers can be used only within subprograms, they can be accessed in the live keyboard mode or by stopping execution during a subprogram. A stop statement can be used in a subprogram to stop execution of the subprogram. The current value of any of the p-numbers in the subprogram can be displayed or changed, but new p-numbers can't be created.

Split and Integer Precision Storage

With the AP and String Variables ROM installed in your HP 9825, you can compactly store values in split and integer precision formats using string variables. In stored form, the values cannot be used directly in calculations, although they can easily be converted back to numeric values for that purpose. This enables you to store large amounts of data using half (split precision) or one fourth (integer precision) as much memory as full precision storage requires.

Split Precision Storage

Using split precision format, full precision numbers (twelve digit mantissa with sign and exponent) are rounded to six digits and stored in string variables. Only values with exponents in the range of ±63 can be stored using split precision format.

The full to split (f t s) function stores a value in split precision format by encoding the value into four characters* (or bytes) which can then be stored in a previously dimensioned string variable. The location within the string variable (first and last characters) where the encoded value is to be stored should always be specified to eliminate truncation of the rest of the string. The value to be stored must be enclosed in parentheses.

f t s (expression)

*The first character contains the exponent and sign. Each of the three remaining characters contain two BCD (Binary Coded Decimal) digits.

To unpack the value, the split to full (stf) function is used. The string variable must also be enclosed in parentheses.

stf (string variable)

Here's a program that uses the fts function to store a list of ten random numbers. (The rnd function in line 4 generates the random numbers.) The numbers are packed into a string array consisting of ten strings, each four characters long.*

```
0: fxd 11
1: dim A$[10,4]
2: prt "STORING"
3: for I=1 to 10
4: rnd(1)→A;prt
 A
5: fts (A)→A$[I]
6: next I
7: spc
```

The rest of the program unpacks the stored values using the stf function and then prints the numbers. The values being recovered are six digit numbers because they were rounded before they were stored using the fts function.

```
8: prt "RECOVERI
 NG"
9: for J=1 to 10
10: stf(A$[J])→A
 ;prt A
11: next J
12: end
```

Now press (RUN) to start the program and compare these printouts with yours. (Press (RESET) before running any of the example programs in this chapter to get printouts identical to those shown.)

```
STORING
   0.67821900935
   0.38218685905
   0.41914846259
   0.50385703791
   0.74376887685
   0.50962543530
   0.59499108444
   0.38750201234
   0.88919237916
   0.81079087248
```

```
RECOVERING
   0.67821900000
   0.38218700000
   0.41914800000
   0.50385700000
   0.74376900000
   0.50962500000
   0.59499100000
   0.38750200000
   0.88919200000
   0.81079100000
```

*Normally the first and last characters of the string variable being used for storage (i.e., A$[I,1,4]) must be specified, otherwise the remainder of the string may be truncated after the last character stored. However, in this program it's not needed, since each string is only four characters long.

All values are rounded to six digits before they are stored. If you attempt to store a number with an exponent outside the range of −63 to +63 (and flag 14 is clear), `error A8` is displayed and flag 15 is set (to 1)*. To avoid this error, you can set flag 14 before the `fts` function is executed. This causes a default value to be substituted and stored. If the exponent is less than −63, the underflow default value is 0; if the exponent is greater than +63, the overflow default value is ±9.99999e63. Flag 15 is set regardless of whether flag 14 is set or not.

To illustrate what happens when the exponent is less than −63 (underflow), execute these statements−

```
erase a
dim A$[4]:fts(1.2345678e-65)→A$
```

And the display shows−

```
error A8
```

Then set flag 14, and the underflow value is automatically substituted. Key in and execute these statements−

```
sfg14
flt9:fts(1.2345678e-65)→A$
stf(A$)
```

Which substitutes, stores and displays−

```
0.000000000e 00
```

To illustrate what happens when the exponent is greater than +63 (overflow), execute the following statements−

```
erase a
dim A$[4]:fts(1.2345678e65)→A$
```

And the display shows−

```
error A8
```

*Remember that flag 15 is set when ***any*** math error occurs.

By setting flag 14 first, the overflow default value is substituted. Key in and execute these statements–

```
sfg14
flt9;fts(1.2345678e65)→A$
stf(A$)
```

Which substitutes, stores and displays–

```
 9.999990000e 63
```

The next example uses split precision format to store four full precision numbers in each simple string in a string array. As many numbers as the size of the memory and the size of the string array allow, can be stored in split precision format. This means that you can use a string array just like a chart or a table to store data (part numbers, temperatures, etc.) for easy reference. This program also uses the rnd function in line 4, to generate the values to be stored.

```
0: fxd 11
1: dim A$[4,16]
2: for I=1 to 4
3: for J=1 to 4
4: rnd(1)→A;prt
 A
5: fts (A)→A$[I,
 4(J-1)+1,4J]
6: next J;spc
7: next I
8: spc 3
```

Notice that in line 5 three expressions are used to position the value in the appropriate string – the string used for storage (I), the beginning character of the string where the value is to be stored (4(J-1)+1) and the end character where the value is to be stored (4J).

To recall the numbers from split precision format, add these lines to the program and run it.

```
9: for K=1 to 4
10: for L=1 to 4
11: stf(A$[K,
 4(L-1)+1,4L])→A
 ;prt A
12: next L;spc
13: next K
14: end
```

And the printout looks like this—

```
0.67821900935                0.67821900000
0.38218685905                0.38218700000
0.41914846259                0.41914800000
0.50385703791                0.50385700000

0.74376887685                0.74376900000
0.50962543530                0.50962500000
0.59499108444                0.59499100000
0.38750201234                0.38750200000

0.88919237916                0.88919200000
0.81079087248                0.81079100000
0.87512375514                0.87512400000
0.97907806819                0.97907800000

0.40465534756                0.40465500000
0.31514729971                0.31514700000
0.03887905206                0.03887910000
0.69728278019                0.69728300000
```

Some applications require that data be stored in a linear array. By storing data in a single string instead of string arrays, numbers can be stored even more compactly by saving the bytes of memory that would have been allocated for the setting up (overhead) of a string array.

The following example stores numbers in a simple string using the rnd function to generate the values to be stored.

```
0: fxd 9
1: dim A$[80]
2: for I=1 to
 77 by 4
3: rnd(1)→A;prt
 A
4: fts (A)→A$[I,
 I+3]
5: next I
6: spc
```

To recover the numbers, add these lines and run the program.

```
7: for J=1 to
 77 by 4
8: stf(A$[J,J+
 3])→A;prt A
9: next J
10: end
```

To get these printouts–

```
0.678219009
0.382186859
0.419148463
0.503857038
0.743768877
0.509625435
0.594991084
0.387502012
0.889192379
0.810790872
0.875123755
0.979078068
0.404655848
0.315147300
0.038879052
0.697282780
0.414818142
0.862057758
0.990574867
0.073462872
```

```
0.678219000
0.382187000
0.419148000
0.503857000
0.743769000
0.509625000
0.594991000
0.387502000
0.889192000
0.810791000
0.875124000
0.979078000
0.404655000
0.315147000
0.038879100
0.697283000
0.414818000
0.862058000
0.990575000
0.073462900
```

Integer Precision Storage

Using integer precision format, numbers in the range −32768 to +32767 can be stored as integers in string variables even more compactly than split precision format.

The full to integer (`fti`) function rounds a value to an integer and stores it in integer precision format by encoding the value into two characters (or bytes) which can then be stored in a previously dimensioned string variable. The location within the string variable (first and last characters) where the encoded value is to be stored should always be specified to eliminate truncation of the rest of the string. The value to be stored must be enclosed in parentheses.

`fti` (expression)

To recover or unpack the value, the integer to full (`itf`) function is used. The string variable must also be enclosed in parentheses.

`itf` (string variable)

The following program uses the `fti` function to store a list of ten random numbers. (The `rnd` function in line 4 generates the random numbers.) The numbers are packed into a string array consisting of ten strings, each two characters long.*

```
0: fxd 2
1: dim A$[10,2]
2: prt "STORING"
3: for I=1 to 10
4: 250rnd(1)→A;
 prt A
5: fti (A)→A$[I]
6: next I
7: spc
```

The rest of the program unpacks the stored values using the `itf` function and then prints the numbers. The values being recovered are integers within the range previously stated because they were rounded before they were stored.

```
8: prt "RECOVERI
 NG"
9: for J=1 to 10
10: itf(A$[J])→A
 ;prt A
11: next J
12: end
```

*Normally the first and last characters of the string variable being used for storage (i.e., A$[I,1,2]) must be specified, otherwise the remainder of the string may be truncated after the last character stored. However, in the following program it's not needed since each string is only two characters long.

Now press (RUN) to start the program and compare the listings.

```
STORING
          169.55
           95.55
          104.79
          125.96
          185.94
          127.41
          148.75
           96.88
          222.30
          202.70
```

```
RECOVERING
          170.00
           96.00
          105.00
          126.00
          186.00
          127.00
          149.00
           97.00
          222.00
          203.00
```

If you attempt to store a number outside the range −32768 to +32767 using integer precision format (and flag 14 is clear) `error A8` is displayed and flag 15 is set.*

To avoid `error A8`, you can set flag 14 before the `fti` function is executed. This causes an overflow default value (−32768 or +32767) to be substituted. Flag 15 is set regardless of whether flag 14 is set or not.

To illustrate overflow, execute these statements−

```
erase a
dim A$[2]:fti (-54321)→A$
```

And the display shows−

```
error A8
```

By setting flag 14 first, the overflow default value is substituted without displaying an error. Key in and execute these statements−

```
sf 14
fxd 0:fti (-54321)→A$
itf (A$)
```

And the default value is automatically substituted, stored and displayed−

```
-32768
```

*Remember that flag 15 is set when ***any*** math error occurs.

If the value to be packed is between −.5 and .5, then it is rounded to zero as shown here−

```
fxd 11;fti(.12345)→A$;itf(A$)
```

```
 0.00000000000
```

Here's an example that uses integer precision format to store eight values in each simple string of a string array. As many numbers as the size of the memory and the size of the string array allow, can be stored in integer precision format. This means that you can use a string array to store data in a table or chart for easy reference. This program also uses the rnd function to generate the values to be stored.

```
0: fxd 2
1: dim A$[4,8]
2: for I=1 to 4
3: for J=1 to 4
4: 250rnd(1)→A;
 prt A
5: fti (A)→A$[I,
 2(J-1)+1,2J]
6: next J;spc
7: next I
8: spc 3
```

Notice that in line 5, three expressions are used to position the value in the appropriate string - the string being used for storage (I), the beginning character where the value is to be stored (2(J-1)+1) and the last character where the value is to be stored (2J).

To recall the numbers from integer precision format, add these lines to the program and run it.

```
9: for K=1 to 4
10: for L=1 to 4
11: itf(A$[K,
 2(L-1)+1,2L])→A
 ;prt A
12: next L;spc
13: next K
14: end
```

And the printout looks like this—

```
169.55                170.00
 95.55                 96.00
104.79                105.00
125.96                126.00

185.94                186.00
127.41                127.00
148.75                149.00
 96.88                 97.00

222.30                222.00
202.70                203.00
218.78                219.00
244.77                245.00

101.16                101.00
 78.79                 79.00
  9.72                 10.00
174.32                174.00
```

Some applications require data storage in a string or linear array. By storing data in a single string instead of string arrays, numbers can be stored even more compactly by saving the memory that would have been allocated for the setting up (overhead) of a string array.

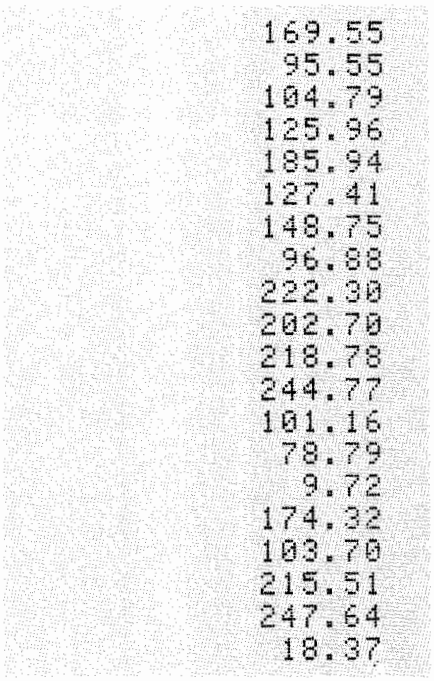
The following example stores numbers in a single string using the rnd function to generate the values to be stored.

```
0: fxd 2
1: dim A$[40]
2: for I=1 to
 39 by 2
3: 250rnd(1)→A;
 prt A
4: fti (A)→A$[I,
 I+1]
5: next I
6: spc
```

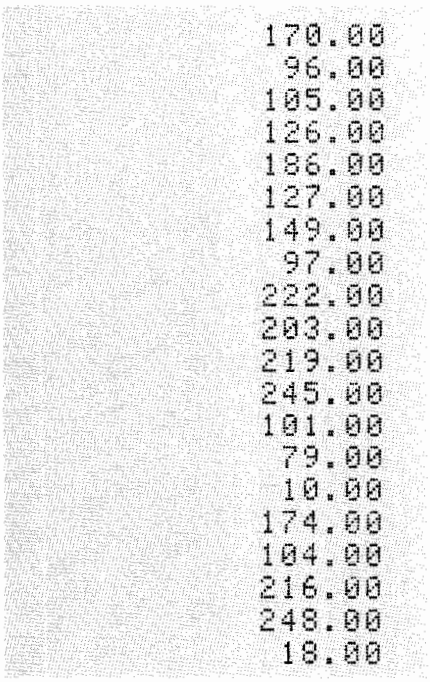
To recover the numbers, add these lines and run the program:

```
7: for J=1 to
 39 by 2
8: itf(A$[J,J+
 1])→A;prt A
9: next J
10: end
```

And the printout shows–



```
169.55
 95.55
104.79
125.96
185.94
127.41
148.75
 96.88
222.30
202.70
218.78
244.77
101.16
 78.79
  9.72
174.32
103.70
215.51
247.64
 18.37
```



```
170.00
 96.00
105.00
126.00
186.00
127.00
149.00
 97.00
222.00
203.00
219.00
245.00
101.00
 79.00
 10.00
174.00
104.00
216.00
248.00
 18.00
```

Summary

Full precision numbers (twelve digit mantissa plus exponent and sign) can be compactly stored in strings or in string arrays using one of two possible storage formats. Split precision format packs data in half the memory space that full precision storage requires and integer precision format packs data in one fourth the memory space that full precision storage requires.

Storing a number using full precision format requires eight bytes of memory. Using split precision format, only four bytes of memory are required to store a number. This is accomplished by limiting the range and precision of the numbers that can be stored. Using split precision format, the number is rounded to six digits before storage. In addition, the exponent must be in the range −63 to +63. If it's not in that range, then flag 15 is set (to 1) and `error A8` is displayed (if `flg 14` is clear). To avoid `error A8`, you can set flag 14 before executing the `fts` function, causing a default value to be substituted and stored. For an overflow error, the default value is ±9.99999e63; if it's an underflow error the default value is 0.

The following program illustrates how the drnd function internally rounds the value to be packed to six digits before storage in split precision format.

```
0: dim A$[4]
1: for I=1 to 10
2: rnd(1)→A
3: fts (A)→A$
4: if drnd(A,
 6)#stf(A$);prt
 A,"Different";
 stp
5: next I
6: prt "ALL OK";
 spc 2
7: end
```

```
ALL OK
```

Using integer precision format, only two bytes of memory are required to store a number. Integers in the range −32768 to +32767 can be stored using integer precision format. If you attempt to store a number that's outside of this range using integer precision format, flag 15 is set and error 88 is displayed (if flg 14 is clear). To avoid error 88, you can set flag 14 before executing the fti function, causing an overflow default value (−32768 or +32767) to be substituted and stored. If the value to be packed is between −.5 and .5, then it is rounded to zero.

This program shows how the prnd function internally rounds the value to be packed to the nearest integer value before storage in integer precision format.

```
0: dim A$[2]
1: for I=1 to 10
2: 32767rnd(1)→A
3: fti (A)→A$
4: if prnd(A,
 0)#itf(A$);prt
 A,"Different";
 stp
5: next I
6: prt "ALL OK";
 spc 2
7: end
```

```
ALL OK
```

When storing numbers in a string variable using the fts or fti functions, the locations where storage begins and ends within the string variable must be specified; otherwise the string may be truncated after the last character stored.

Cross Reference Statement

The cross reference (`xref`) statement prints each variable used in your program followed by the line numbers in which it appears.

```
xref
```

For programs with many variables, the `xref` statement aids in keeping track of these variables and their locations in your program. The `xref` statement can be executed from the keyboard, in the live keyboard mode or within a program. The variables used in the program – simple, numeric array, string and r-variables – are printed, in that order. Within each type, the variables are arranged alphabetically.

When `xref` is executed, it searches the program once for each of the 79 possible variables (26 simple, 26 numeric array, 26 string and r-variables*). The `xref` statement does not list references to p-numbers or variables used in Matrix ROM statements (see the Matrix ROM Programming Manual).

*All r-variables are considered as one for this statement and they appear together at the end of the cross reference listing.

The following program finds prime numbers and their logarithms using simple, numeric array, string and r-variables.

```
0: dim P$[1000→r
 0],L[r0/2→r1],
 D$[16]
1: 0→I;2→X
2: "Loop":
3: if not 'Prime
 '(X);sto "Not
 Prime"
4: I+1→I
5: fti (X)→P$[2(
 I-1)+1]
6: log(X)→L[I]
7: fxd 0;"Prime"
 &str(I)&"="&str
 (X)→D$;fxd 4;
 dsp D$,",Log=";
 L[I]
8: if I<r1;sto
 "Not Prime"
9: dsp "Done";
 end
10: "Not Prime":
11: X+1→X;sto
 "Loop"
12: "Prime":
13: rp1→p2
14: for J=1 to I
15: if (itf(P$[2
 (J-1)+1])→p3)>p
 2;ret 1
16: if Xmodp3=0;
 ret 0
17: next J;ret 1
```

By executing the `xref` statement, these variables are listed–

```
I     1     4     4
5     6     7     7
8     14

J     14    15    17

X     1     3     5
6     7     11    11
16

L[*]0       6     7

D$    0     7     7

P$    0     5     15

r     0     0     0
8
```

Chapter 5
Table of Contents

Chapter 5

Tape Cartridge Operations

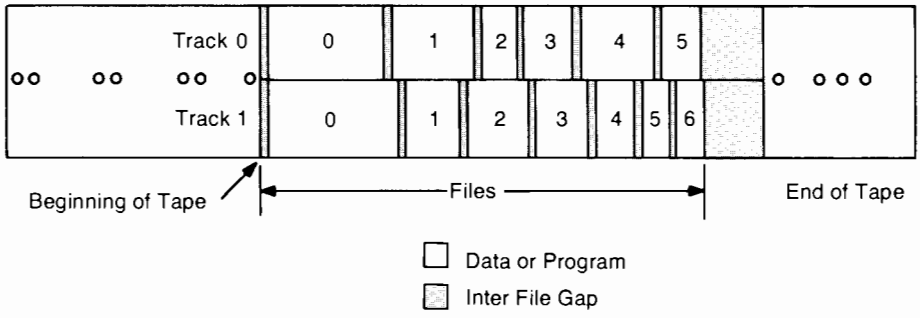
The tape cartridge used with 9825 Calculator is a high quality, high density, digital storage medium. The structure, care, and use of the tape cartridge are detailed in this chapter.

Specifications

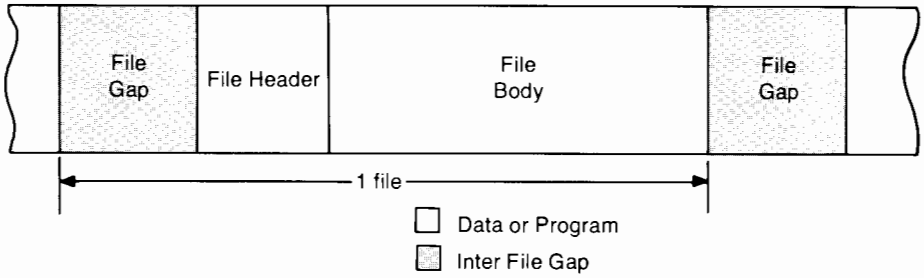
Typical data transfer rate (the rate at which information is loaded from or recorded on the tape cartridge)	2750 bytes per second
Typical access rate (the rate at which information passes over the tape head when searching for a file)	14300 bytes per second
Typical rewind time (from end to end)	19 seconds
Typical erase time (one entire track)	40 seconds
Usable tape length (typical)	42.67 meters (140 ft.)
Number of tracks	2

Tape Structure

The structure of the tape is diagrammed below:

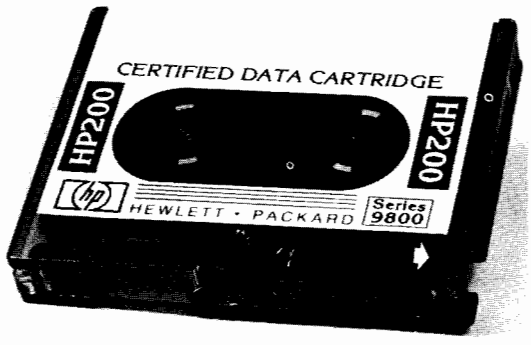


An individual file has the following format:



Tape Cartridge

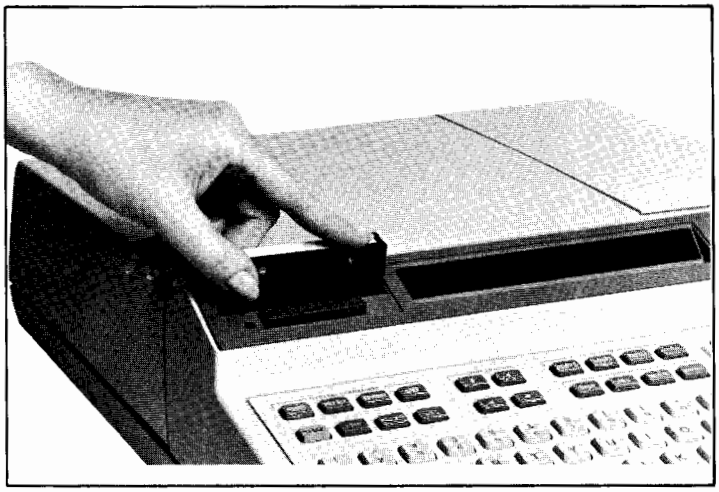
The tape cartridge, shown below, is used to store programs, data, and the defined special function keys.



To record on the tape cartridge, the record slide tab must be in the rightmost position, that is, in the direction of the arrow (as shown).

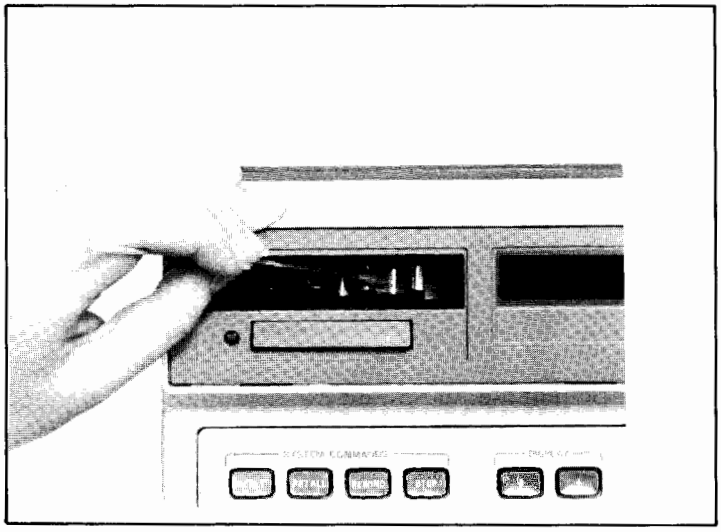
Inserting the Cartridge

Insert the tape cartridge so that the label on the cartridge faces the back of the calculator as shown.



Inserting the Tape Cartridge

Tape Care



Cleaning the Tape Head and Capstan

Dirt and dust are by far the greatest cause of cartridge related-errors. Several basic precautions can reduce such problems substantially.

- Clean the tape head and capstan (drive wheel) of the tape transport after at least every eight hours of use, or more frequently in dirty environments.
- Rewind the cartridge after each use.
- Keep the tape transport door clean.
- Keep the cartridge in the plastic container supplied with it.

Two other factors can affect the reliability of the tape cartridge. Strong magnetic fields can erase data and programs stored on the cartridge. Physical damage to the tape, such as wrinkled or folded tape can also cause record and load problems. A back-up copy should be maintained for critical programs or data on a separate tape cartridge.

Information on tape error recovery is at the back of this chapter.

The Rewind Statement

```
rew
```

The rewind (**rew**) statement is used to rewind the tape cartridge to its beginning. This statement has the same function as [REWIND]. Operations which do not use the tape cartridge can take place while the tape rewinds. To stop a tape while it is rewinding, press the tape cartridge ejection bar. The rewind statement must be executed before marking a new tape.

The Track Statement

```
trk track number
```

The (**trk**) statement sets track 0 or track 1 of the tape cartridge. When the track statement is executed, any following cartridge operations are performed on that track. Track 0 is automatically set whenever the machine is switched on, [RESET] is pressed, or `erase a` is executed. The track does not change when the cartridge is removed nor when [RUN] is pressed.

The track number can be an expression with a value of 0 or 1, only.

CAUTION

THE TRACK IS AUTOMATICALLY SET TO 0 WHEN RESET IS PRESSED, `erase a` IS EXECUTED, OR WHEN THE CALCULATOR IS SWITCHED ON. UNLESS A SUBSEQUENT TRACK STATEMENT SPECIFIES TRACK 1, CARTRIDGE OPERATIONS WILL BE PERFORMED ON TRACK 0. IF YOU ARE UNAWARE OF THIS, YOU COULD LOSE IMPORTANT PROGRAMS OR DATA.

The Identify File Statement

`idf` [file number [, file type [, current file size [, absolute file size [, track number]]]]]

The identify file (**idf**) statement is used to load the contents of the current file header into the return variables specified. After the identify file statement is done, the tape is positioned in front of the file just identified. Thus, the tape is positioned for easy loading or recording of the identified file.

All five of the parameters are optional return variables. That means that a value is returned to the variable specified when the statement is executed. If one variable is specified, as in: `idf A`, then only the file number is returned. Two variables must be specified to get the file type; three variables to get the current file size in bytes; four variables to get the absolute file size in bytes; and five variables to get the track number. The return variables can be any variable type.

The file type can be one of the following:

0 null* file
1 binary program
2 numeric data
3 string or string and numerics
4 memory file (from record memory statement)
5 key file
6 program file
7 track dump error recovery (disk)
8 single file dump (disk)
9 entire disk dump

*A null file has an absolute size of zero.

The tape position becomes unknown when a tape cartridge is inserted into the tape drive, the track is changed, (RESET) is pressed, or erase a is executed. If the tape position is unknown such as after switching tracks, at least one return variable must be specified or error 45 will occur.

Example:

```
idf A, B, C, D, E
```

Identify the current file and return the file number, file type, current file size, absolute file size, and track number to A, B, C, D, and E, respectively.

```
idf A, A, A
```

Return the current file size to A.

The Find File Statement

fdf [file number]

The find file (**fdf**) statement is used to find the specified file on the current track of the tape cartridge. The tape is positioned at the beginning of the file specified. The file number can be an expression. A find file statement without parameters finds file 0. Other statements can be executed while the find file statement is executing.

> **NOTE**
>
> If a file number which does not exist is specified, the ***next*** cartridge statement executed (except find file or rewind) will result in error 65.

Examples of the find file statement:

```
fdf 8
```

Find file 8.

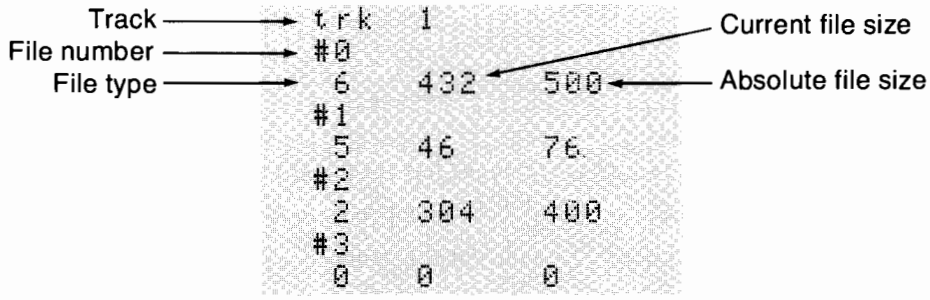
```
 4: fdf A[3]
```

Find the file specified by the value of A [3].

The Tape List Statement

```
tlist
```

The tape list (**tlist**) statement is used to identify the files on the tape cartridge. Starting from the tape's current position, the track, file number, file type, current file size in bytes, and absolute file size are printed as shown below.



The file type can be one of the following:

0 null* file
1 binary program
2 numeric data
3 string or mixed string and numeric data
4 memory file (from record memory statement)
5 key file
6 program file

*A null file has an absolute size of zero.

If (STOP) is pressed while tlist is being executed, the tlist will terminate. Otherwise it will halt when the last file (null file) is reached.

A convenient way to determine the current track setting is to execute "tlist" then press (STOP). Alternately, use the identify file statement as in: idf T; T; T; T; T; T (ENTER).

Marking Tapes

The Mark Statement

mrk number of files; file size in bytes [; return variable]

The mark (**mrk**) statement reserves file space on the tape cartridge. A file must be reserved before a program or data can be recorded. One file more than the number of files specified is marked. This file is the null file and is used as the starting point when marking more files. The null file has an absolute size of zero.

The file size is specified in bytes. If an odd number of bytes is specified, one more byte is automatically marked. For example, if 111 bytes are specified, 112 bytes are marked.

In order to mark files, the position of the tape must be known. If the position is unknown, execute a find file, or rewind statement to position the tape where you are going to start marking. Executing a mark statement where the first two parameters are zero (e.g., mrk 0,0) is a special case and is explained in the Tape Cartridge Errors section.

The number of files and the file size can both be expressions. If a return variable is specified, the file number of the last usable file marked is stored in it. If the value of the return variable is positive, all the files specified were marked. If the value is negative, an end-of-tape (eot) condition occurred before all the requested files were marked. In either case, the absolute value of the return parameter is the last usable file marked. The null file is one file beyond.

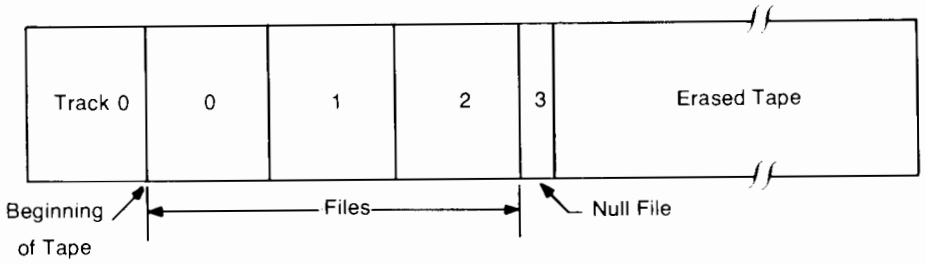
Example:

A tape is to be re-marked for 3 files with a length of 320 bytes each on track 0. The following short program performs this operation.

```
 0: rew
 1: trk 0
 2: mrk 3,320,X
 3: ert X+1
 4: end
```

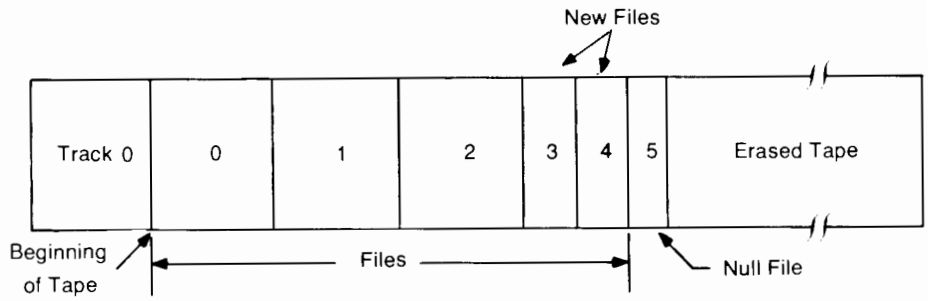
Rewind the cartridge.
Set to track 0.
Mark 3 files, 320 bytes long.
Erase the rest of track 0.
End the program.

The tape will be positioned at the beginning of file 3 and the resulting tape structure will be:



Then, 2 files with a length of 80 bytes are to be marked. Execute: `mrk 2,80`

New tape structure:



To mark 2 files, 300 bytes long beginning at file 4, execute: `fdf 4; mrk 2,300`

Determining Size to Mark a File

Program Files

When marking a file for a program which is currently in the calculator, execute `list -1`. The number in the left-hand portion of the display is exactly the number of bytes needed to record the program. It is advisable to mark the file larger to accommodate any future program changes.

Data Files

Data files require 8 bytes for each data element to be recorded. For example, to record data which is stored in the variables A and B, mark a file 16 bytes long.

Special Function Key Files

Special function key files require 1 byte for each character under the keys, plus 2 bytes for each defined key. If the number of bytes for each key is odd, add one byte. The sum for all keys is the minimum size to mark the file.

Memory Files

For a memory file (using record memory statement), mark the file for the size of your computer's available read/write memory. Refer to the label under the computer's paper-access lid.

Tape Capacity

Table of Typical Storage Capacities

File size (bytes)	Typical number of files per track	Bytes per track
50	827	41350
100	656	65600
250	404	101000
500	239	119500
750	170	127500
1000	131	131000
2500	56	140000
5000	28	140000
7500	19	142500
10000	14	140000

Due to the overhead required by each file, the number of bytes per track is not the same for different size files.

Tape Capacity Calculations

The number of files which can be stored on the tape cartridge depends on the size of the file. Using the following calculations, the number of files that can be stored on the tape cartridge can be calculated.

$$L = 1.278 + .209 \text{ int } (A/256 + .999) + .0105A$$

where: A = absolute file size in bytes.
L = length of the file in inches.

a) For **typical** capacity per track:
Number of files per track = int (1665/L)

b) For **minimum** capacity per track:
Number of files per track = (1498/L)

The following program can be used to mark more files and calculate the percentage of a track used.

```
0: rew;fxd 0
1: ent "Track,
 0 or 1?",T;trk
 T
2: ent "New tape
 ? Yes=1 No=0",N
3: if N;gto "Mar
 k"
4: 0→F→L
5: fdf F;idf A,
 A,A,A
6: if A=0;gto
 "Mark"
7: gsb "L"
8: Q+L→L;F+1→F;
 gto 5
9: "Mark":prt
 "% of Tape"
10: prt "    Mark
 ed",L/1665*100→
 r0
11: if r0>=100;
 prt "All Marked
 ";gto "out"
12: ent "Mark
 more files?
 Yes=1 No=0",M
13: if M=0;gto
 "out"
14: ent "Length
 of files",A;
 gsb "L"
15: dsp "Number
 of files",A,
 "long?"
16: ent "",I;I*
 Q+L→r1
17: if (r1/1665→
 Z)>=1;prt "Too
 long...","That
 is %",Z*100;
 gto "out"
18: r1→L;mrk I,
 A,R;if R<0;dsp
 "100% Marked";
 gto "out"
19: gto "Mark"
20: "out":dsp
 "That's it";end
21: "L":1.278+
 .209int(A/256+
 .999)+.0105A→Q;
 ret
```

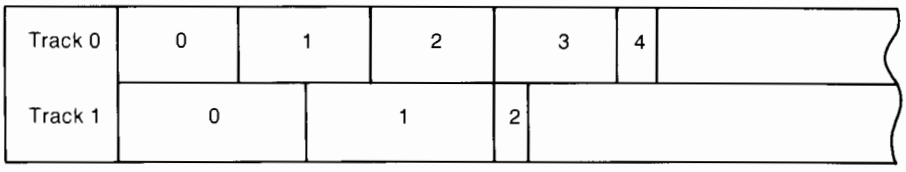
Marking New Tapes

Since no files are marked on a new tape, the rewind statement must be used to position the tape before marking files. For example, to mark 4 files 200 bytes long on a new tape, execute the following:

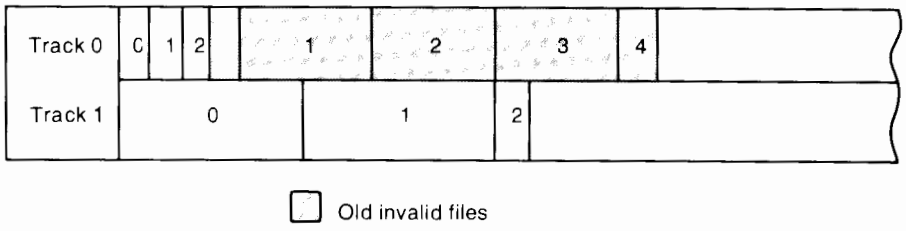
```
rew
mrk 4, 200
```

Marking Used Tapes

When re-marking a used tape, it is possible that some old files may remain on the tape. These files can be accessed accidentally by changing tracks. For example, suppose track 0 has 4 files of 1000 bytes and track 1 has 2 files of 1500 bytes:

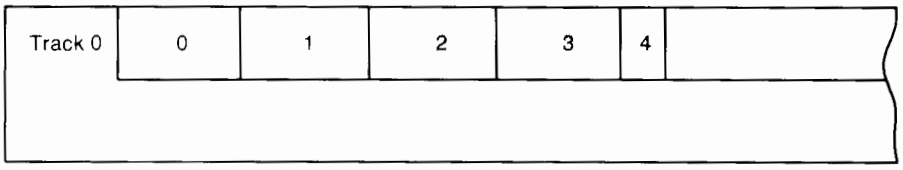


Then track 0 is re-marked from the beginning to contain 2 files of 200 bytes each:

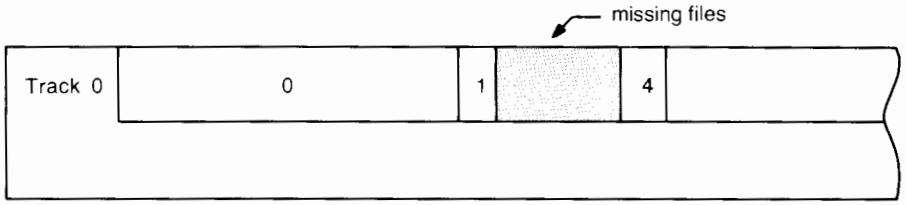


If the tape is positioned at file 1 on track 1, and `trk 0` is executed, the tape will be positioned in an old section of tape. Accessing file 1 on track 0 will result in using **old** file 1.

With slightly different conditions, it is possible to have missing files rather than multiple files. For example, suppose that track 0 has 4 files of 1000 bytes each:



If the tape is rewound and `mrk 1, 3000` is executed, the tape would have a gap of missing files:



To remove the old files, use the erase tape (ert) statement. For the first example:

`rew`	Rewind the tape on track 0.
`mrk 2,200, A`	Mark 2 files, 200 bytes each.
`ert A+1`	Erase the tape starting at the null file.

> CAUTION
>
> WHEN MARKING OVER A PREVIOUSLY MARKED TAPE, USE THE ERASE TAPE STATEMENT TO REMOVE OLD FILES.

The Erase Tape Statement

ert file number

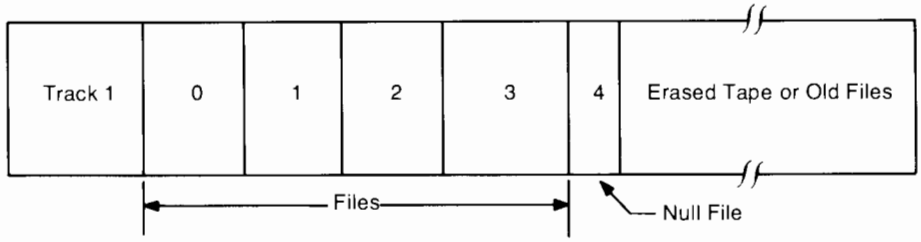
the erase tape (ert) statement is used to erase everything on the current track starting from the file number specified. It is usually executed after a mark statement (see Marking Used Tapes).

The erase tape statement:

1. Positions the tape in front of the file specified.
2. Marks that file as a null file.
3. Then, erases the track from the null file to the end of the track.
4. Finally the tape is positioned in the file gap in front of the null file.

The file number can be an expression.

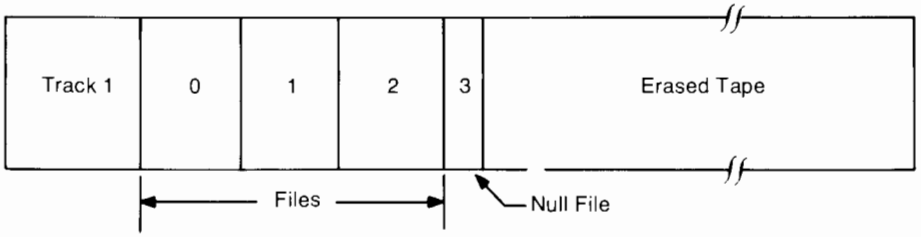
For example, a tape has the following structure on track 1:



To erase everything on track 1 starting at, and including file 3, the following program is used:

```
0: trk 1
1: ert 3
2: end
```

After running this program, the tape's structure is:



Track 0 is not altered.

The Record File Statement

The record file statement is used to store both data and programs. The syntax for each is explained below.

Recording Programs

rcf [file number [, beginning line number [, ending line number]] [, "SE" or "DB"]]

To record a program or a section of a program the record file (**rcf**) statement is used. If no file number is specified, the file is assumed to be file zero. If no line numbers are specified, the entire program is recorded on the specified file. If the beginning line number is specified, then the program from that line number to the end is recorded. If both line numbers are specified, that program section is recorded from the first line number to the second line number, inclusive.

The file number and ending line number parameters can both be constants, variables, or expressions. The beginning line number can be a constant, or expression (such as 1A), but must not be a variable. Using a variable as in rcf 1,A records the value of "A" as data. To record the program beginning at the line whose value is A, use rcf 1,1A.

If "SE" (for secure) follows at the end of the statement, the program is secured when stored on tape. When the secured program is loaded back into the calculator, the program cannot be listed or displayed, but can be re-recorded on a tape cartridge.

When "DB" (for debug) follows the end of the statement, any trace or stop flags are recorded with the program (see Program Debugging in chapter 3).

The tape file must be marked before recording a program. The file size must be greater than or equal to the size of the program being recorded.

Example:

```
7: rcf 8,3
```

Record the program on file 8, starting at line 3 through the end.

Recording Data

rcf file number , data list

The record file (**rcf**) statement is used to record data when this syntax is used. The data list can consist of simple variables, array variables, or r-variables. r-variables are stored in a different area in memory which is not contiguous with array or simple variables. Due to this, r-variables cannot be mixed with simple or array variables in the record file statement.

To record an entire array, the array name is followed by an asterisk in brackets. For example:

```
rcf 2, S[*]
```

Record the entire array S on file 2.

Simple and array variables must appear in the data list in the same order as allocated. If the variables appear in a dimension statement, then they must appear in the same order in the record file statement.

Example:

```
 0: dim A[10,10]

 1: 0→X
 2: X+1→X
 3: 1→I
 4: rcf 5,A[*],X,
    I
```

The array A is allocated 100 elements (800 bytes).
The variable X is allocated 8 bytes.
Doesn't affect memory allocated to X.
The variable I is allocated 8 bytes.
The array A, and variables X and I are recorded in the same order as allocated (contiguously) on file 5 (total of 102 numbers or 816 bytes).

If one r-variable is specified in the data list, all r-variables from r0 to that r-variable are recorded. If two r-variables are specified, all r-variables from the first through the second are recorded.

Considerations for Recording Data

When recording data on the tape cartridge, the variables being recorded must be listed in the same order as they are allocated in memory. For example:

```
 0: ent A
 1: 2*A→B
 2: dim C,X,Y,Z
    •
    •
    •
15: rcf A,B,C,X,
    Y,Z
```

In the program, the variables A and B are allocated outside a dimension statement. Variables C, X, Y, and Z are allocated in a dimension statement. But, if B were allocated before A in the program, line 15 would cause error 56 to be displayed since the variables must be listed in the same order as they are allocated. Because lines are not necessarily executed in numerical order, it is sometimes difficult to know the order in which variables are allocated. For this reason, when a group of simple or array variables is to be recorded on a single file, it is recommended that they all be allocated in one dimension statement.

The Load Program Statement

ldp [file number [, line number$_1$ [, line number$_2$]]]

The load program (**ldp**) statement is used to load a program from the specified file on the current track and **run** it automatically. The automatic run implies that all variables are erased, all subroutine return pointers are cleared, and all flags (0 through 15) are cleared.

When the file number only is given, the program is loaded from the file, beginning at line zero, and the program automatically **runs** from line zero. If the file number and the first line number are specified, the program is loaded from that file, beginning at the specified line number and **runs** from that line number. When all three parameters are specified, the program is loaded from the specified file number beginning at the first specified line number and is **run** beginning at the second specified line number. If no parameters are specified, zeros are assumed for all three. All three parameters can be expressions.

If a program is loaded at the end of an existing program, go to and go sub branching line numbers are not renumbered.

The load program statement can only be stored as the last statement in a line. This statement is not allowed in live keyboard mode nor during an enter statement.

Examples:

`ldp 2`	Loads the program from file 2 beginning at line 0 and runs from line 0.
`ldp 8,2`	Loads the program from file 8 beginning at line 2 and runs from line 2.
`ldp 16,3,0`	Loads the program from file 16 beginning at line 3 and runs from line 0.

The Load File Statement

The load file (**ldf**) statement is used to load both data and program files into the calculator memory.

> CAUTION
>
> THE LDF STATEMENT LOADS THE PROGRAM OR VARIABLE AREA OF MEMORY DEPENDING ON THE FILE TYPE ACCESSED. BUT THE LDP AND RCF STATEMENTS LOAD OR RECORD A SPECIFIED PART OF MEMORY DEPENDING ON THE STATEMENT. THUS, WITH THE LDF STATEMENT IT IS POSSIBLE TO ACCIDENTALLY LOAD A PROGRAM WHEN THE INTENT WAS TO LOAD VARIABLES OR VICE VERSA.

Loading Programs

ldf [file number [, line number$_1$ [, line number$_2$]]]

The load file (**ldf**) statement loads programs from the specified file on the current track into the calculator memory.

This statement is like the load program (ldp) statement except that ldf can be used to continue a program, while the ldp statement causes the program to run.

From the Keyboard

This statement is executed from the keyboard as follows: When no parameters are given, the program on file zero is loaded, beginning at line 0. If the file number is given, that file is loaded beginning at line 0. If the file number and a line number are specified, then that file is loaded beginning at the specified line number. When all three parameters are given, the specified file is loaded beginning at the first line number, and the program automatically **continues** at the second line number (all variables are preserved whereas ldp destroys the old variables; see The Continue Command in chapter 3.
If a program is loaded at the end of an existing program, go to and go sub branching line numbers are not renumbered.

In a Program

The ldf statement is executed in a program as follows: When no parameters are specified, the program on file zero is loaded beginning at line zero and continues at line zero. If the file number is specified, then the program is loaded from the specified file beginning at line zero and continues at line zero. When the file number and a line number are given, the specified file is loaded beginning at the specified line number and the program continues from that line

number. When all three parameters are given, the statement is executed the same as from the keyboard. That is, a "continue" is performed from the second line number. All three parameters can be expressions.

This statement is not allowed in live keyboard mode nor during an enter statement to load a program file. However, the ldf statement can be used to load a data file in live keyboard.

Example:

```
ldf 1
```

Load file 1 beginning at line 0 (executed from keyboard).

```
13: ldf 2
```

Load file 2 beginning at line 0 and **continue** from line 0.

Linking Programs

Programs too long to store in the calculator memory can be segmented and stored in separate files on the tape cartridge. Each segment can be loaded as needed by the program, and, using the ldf statement, variables, flags, and subroutine return pointers can be preserved for each segment.

In the following basic example, three segments are used. Each segment is loaded as it is needed by the program. The first segment loads the second and the second loads the third.

Program Segment on file 0 ▶

```
 0: prt "file 0";
   π→A
 1: ldf 1
```

Program Segment on file 1 ▶

```
 0: prt "file 1";
   prt A
 1: ldf 2
```

Program Segment on file 2 ▶

```
 0: prt "file 2"
 1: end
```

Press: [LOAD] [0] [EXECUTE] [RUN]

```
file 0
file 1
                3.1416
file 2
```

Loading Data

ldf [file number [, data list]]

The load file (**ldf**) statement loads data from the specified file on the current track. The data list contains the names of variables separated by commas. r-variables cannot be in the same load file statement with simple and array variables.

If no list is specified, data begins filling the r-variables from r0 until all the data has been loaded. If one r-variable is specified, then the data begins filling r-variables from that r-variable until all the data has been loaded into higher r-variables. If two r-variables are specified, the data starts filling from the first location specified (lower r-variable) to the second, higher, r-variable. If there is more data than available or specified r-variables, no data is loaded.

When simple or array variables are specified, data begins filling the first variable until all variables have assigned values. If there is more data than variables, no data is loaded. If there is less data than variables, the data is loaded until all data is used. Variables must be contiguous.

Examples:

```
ldf 4, r2, r10
ldf r12, A, B[*]
```

Load r2 through r10 from data file 4.
Load the data file designated by r12 into the variable A and array B.

Array and r-variable Recording

Array variables are recorded in the opposite order of r-variables. Thus, if r-variables are recorded, then loaded back into an array, they will be in the opposite order. For example:

```
0: 1→r1;2→r2;
 3→r3;4→r4;5→r5
1: rcf 0,r5
2: dim A[6]
3: ldf 0,A[*]
4: 1→I
5: prt A[I];jmp
 (I+1→I)>6
6: end
```

```
r5= A[1]    5.0000
r4= A[2]    4.0000
r3= A[3]    3.0000
r2= A[4]    2.0000
r1= A[5]    1.0000
r0= A[6]    0.0000
```

In line 1, r-variables 0 through 5 are recorded on file 0. Then in line 3, the array A is loaded from file 0. A[6] is loaded first; A[1] is loaded last.

The Record Keys Statement

rck [file number]

The record keys (**rck**) statement is used to record all the special function keys on the specified file on the current track. If the file number is omitted, file zero is assumed. The file number can be an expression. The specified file must be marked before the record keys statement is executed.

Examples:

```
rck 2
   3: rck A[12]
```

Record the special function keys on file 2.
Record the special function keys on the file designated by the 12th element of array A.

The Load Keys Statement

ldk [file number]

The load keys (**ldk**) statement is used to load the special function keys exactly as they were recorded from the specified file on the current track. If the file number is omitted, file zero is assumed. The file number can be an expression. Executing the load keys statement from the keyboard causes subroutine return pointers to be reset and causes the program counter to reset to line zero.

This statement is not allowed in live keyboard nor during an enter statement.

Example:

```
ldk 4
```

Load the special function keys from file 4.

The Record Memory Statement

rcm [file number]

The record memory (**rcm**) statement records the entire read-write memory (program, data, keys, pointers, etc.) on the specified file on the current track of the tape cartridge.

If the file number is omitted, file 0 is assumed. The file number can be an expression.

The presence of a binary program in memory is a special case. The information supplied with each binary program explains the calculator operation when the record memory statement is executed.

The Load Memory Statement

ldm [file number]

The load memory (**ldm**) statement is used to load a previously recorded memory file. When the load operation is complete, the calculator is in the same state it was in when memory was recorded. If the file number is omitted, file 0 is loaded.

If a program was running when the record memory statement was executed, that program will continue with the next statement after the record memory statement when the load memory statement is executed.

The record memory and load memory statements can be executed from live keyboard or from a special function key. The record memory statement can be used to "freeze" the state of the system without interrupting the running program. These statements can be especially useful in areas where frequent power interruptions occur.

The file number can be an expression.

The Load Binary Program Statement

ldb [file number]

The load binary program (**ldb**) statement loads binary programs* into the calculator's read/write memory from the specified file on the current track of the tape cartridge. Binary programs can be loaded over other binary programs of equal or greater length at any time.

If no file number is specified, file 0 is assumed. The file number can be an expression.

Example:

```
ldb 2                    Load the binary program from file 2.
```

* A binary program is a machine language program which cannot be listed or displayed.

Since binary programs occupy a special place in memory, certain rules must be followed when loading them:

1. Any binary program can be loaded at any time (from the keyboard or a running program) provided there is room in memory for it and no simple or array variables are allocated.
2. Once simple or array variables are allocated, a binary program cannot be loaded unless space has been allocated for it by a previous binary program load operation.

The following procedure is suggested: Before any simple or array variables are referenced, load the largest binary program file that will be needed. Then any variables can be allocated and other binary programs can be loaded without concern about room for the binary program.

File Verification

File verification is used to compare a tape file against the calculator memory to detect recording errors without losing the information in memory. If you get a verify error (error 44), try re-recording the file. Repeated verify errors on a file may indicate damaged tape.

File verification requires a stronger tape signal than load; thus, it increases confidence that a file will load properly at a later time.

When the calculator is turned on, `erase a` is executed, or [RESET] is pressed, the calculator automatically verifies files on **all** record operations. Two statements are used to control automatic verification.

The Auto-Verify Disable Statement

```
avd
```

The auto-verify disable (**avd**) statement turns off file verification. For example:

```
12: avd
```

Turn-off automatic file verification.

The Auto-Verify Enable Statement

```
ave
```

The auto-verify enable (**ave**) statement turns on automatic file verification. After ave is executed, all record operations are automatically followed by a verify. When the calculator is turned on, RESET is pressed, or erase a is executed, automatic file verification is again enabled.

Example:

```
ave                                    Turn on Automatic file verification.
```

The Verify Statement

```
vfy [return variable]
```

The verify (**vfy**) statement is used to compare a tape file with the calculator memory. If the calculator memory is identical to the tape file, the value of the return variable is 0 after the operation. If the two are different, the return variable is one. If no return variable is specified and if the memory and tape file are not identical, error 44 occurs. The return variable can be either a simple variable, array variable, or r-variable.

The verify statement can follow any record operation except the record memory (rcm) statement. The record memory statement followed by vfy will result in error 44. Memory files can only be verified using automatic verification.

With the verify statement, you can selectively verify files. This can be useful to save time when recording many files. Another important use is recovery from verify errors using the return variable parameter.

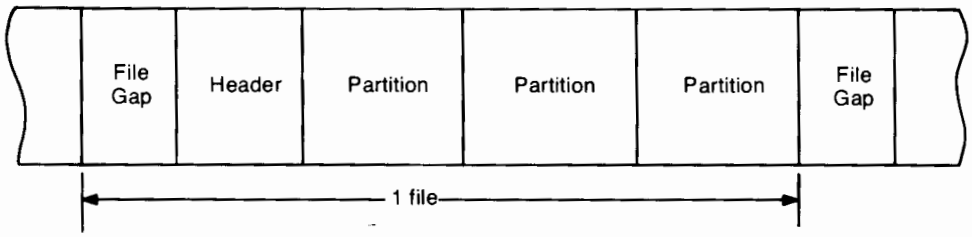
This statement does not alter the calculator memory.

Tape Cartridge Errors

File Body Read Error

If a file body read error (error 46) occurs, first clean the tape head and drive wheel as explained earlier. Then execute the statement which caused the error again. If an error still occurs, the next step depends on the type of file being loaded.

It will be informative at this point to explain something about the file structure of the tape. A file is made up of one or more "partitions". This structure makes it possible to recover portions of a file even though a loading error has occured. Error 46 indicates that one or more partitions may be erroneous.



Loading A Program File

If error 46 occurs while loading a program file, one or more program lines may be lost. The place where this error occured is indicated by a line of asterisks (*) inserted in the program at the point where the program lines are missing. These lines can be replaced by referring to a previous listing.

Note that go to and go sub statement addresses are not adjusted during this editing. Thus, it may be necessary to re-adjust the go to and go sub addresses after inserting the lost lines.

Loading a Data File

If error 46 occurs while loading numeric data, the partition in question is marked by a single number replaced by ?.??????????E00 (in float 11 format). A partition in a numeric data file always contains 32 numbers. With one entry replaced by ?.??????????E00, there are 31 numbers remaining which may be incorrect. To determine the bounds of the affected partition:

- For r-variables, the 31 higher numbered r-variables may be incorrect.
- For simple and array variables, determine the order in which the variables in question were allocated (see dimension statement). From the element that is replaced by ?.????????????e-00, go from right to left in the parameter list of the dimension statement. For an array in the list, the first element in the lost partition will have the largest subscripts. Decreasing the leftmost subscript first for an array reveals the missing values. For example, a partition is lost and the dimension statement was:

```
0: dim A,B,C,
 D[3,10]
```

The value D[3,10] contains question marks. All questionable values can be accessed in this order:

→

D[3,10],	D[2,10],	D[1,10],	D[3,9],	D[2,9],
D[1,9],	D[3,8],	D[2,8],	D[1,8],	D[3,7],
D[2,7],	D[1,7],	D[3,6],	D[2,6],	D[1,6],
D[3,5],	D[2,5],	D[1,5],	D[3,4],	D[2,4],
D[1,4],	D[3,3],	D[2,3],	D[1,3],	D[3,2],
D[2,2],	D[1,2],	D[3,1],	D[2,1],	D[1,1],
C,	B			

File Header Read Error

If a file head read error (error 47) occurs, proceed as follows:

1. Clean the tape head and drive wheel as explained in the Tape Care section. This may solve the current read error and prevent future read errors.
2. Execute the statement that caused the error again.

> **CAUTION**
>
> RE-MARKING A FILE HEADER IS A "LAST RESORT" OPERATION, SINCE ALL INFORMATION ON A FILE WITH A RE-MARKED HEADER IS LOST AND THAT FILE CAN NO LONGER BE USED. HOWEVER, THIS DOES PERMIT YOU TO ACCESS FILES BEYOND THE BAD FILE.

3. If, after steps 1 and 2, the error still occurs, re-mark the tape-file header.

To re-mark the head of file N (file which cannot be loaded), execute:

```
fdf N-1    Positions the tape.
mrk0,0     Re-marks file header of file N.
```

For file 0, execute:

`rew`	Positions the tape.
`mrk0,0`	Re-marks file header of file 0.

After the file header has been re-marked the absolute size of the file is 2 bytes.

Conditioning the Tape

Repeated operations over a short length of tape (usually less than 4000 bytes or 5 ft.) can cause slack. (Extreme changes in temperature can also cause this.) The outer layer of tape can slip and rub on the cartridge, causing damage to the tape. If operation continues, the tape may jam and be ruined.

> NOTE
>
> This condition is most likely to occur if exclusive use is made of one file or two adjacent files near the beginning or end of tape.

If a particular application requires such operation, this slack can be prevented by moving the tape periodically 15 feet or more toward midtape. For example, for a tape with 80 files where only files 0 and 1 are used, execute the following program segment after every 200 operations on file 0 or 1:

```
18: fdf 80
19: rew
```

Tape Life

The tape cartridge does not have an infinite life span. Many factors increase wear and decrease life. A high resistance to turning and continuous use for long periods of time (½ to 3 hours) both result in increased temperature in the cartridge. High humidity, high temperature (above 45°C, 113°F for the cartridge itself) and a high duty cycle (percent of the time the tape is accessed during the total time the computer is used) all increase wear.

Several things start happening to the cartridge which are danger signs:

- The tape begins to wear out and lose information.
- The capstan develops dark bumps due to slippage.
- The cartridge can stall, causing the capstan to wear a flat spot on the drive pulley.
- The cartridge sounds rattly, rather that making a constant hum when the tape moves.
- Errors 43 (indicating tape transport failure), 46 and 47 occur more frequently.

If any of these occur, replace the cartridge at once. If you continue to use it, you could lose all the information on the tape and damage the drive itself.

> CAUTION
>
> NEVER OVERRIDE `error 43` IN YOUR PROGRAMS. BY OVERRIDING A TRANSPORT ERROR, YOU CAN EASILY DAMAGE THE TRANSPORT AND BE FORCED TO REPLACE IT.

Notes

Chapter 6
Table of Contents

Notes

Chapter 6

String Variables

Introduction

The string Variable ROM enables you to –

- Do character manipulations, such as editing portions of strings.
- Use portions of strings as variables in arithmetic calculations.
- Perform character by character comparisons of these strings.
- Use alphanumeric data in input and output operations.

The String Variable ROM is a plug-in accessory which uses 52 bytes of read/write memory when installed in a 9825 A or S Computer. This ROM is a permanent part of the 9825B Computer.

A string is a series of characters, like "ABC! 123*". Any number of characters, within the limits of available memory, can be stored in a string variable. Each character requires one byte of memory and some overhead, as explained later.

Naming Strings

String variables are given names, like A$ or Z$. The dollar sign following the string variable name differentiates strings from numeric variables. Up to 26 string variables, one for each letter of the alphabet, can be used in one program. There are two kinds of string variables – simple strings and string arrays. Each string of a string array can be used in exactly the same way as a simple string.

In the diagram below, A$ is a simple string, nine characters long and Z$ is a 4 (row) by 10 (column) string array containing four strings, Z$[1] through Z$[4]. Each of the strings in this example can be up to ten characters long. Strings have a dimensioned length and a current or "logical" length. The dimensioned length of Z$[1] is 10; its current length is 6.

	1	2	3	4	5	6	7	8	9	10
A$	-	A	B	C	!	1	2	3	*	
Z$[1]	K	e	e	p	e	r				
Z$[2]	F	u	n	c	t	i	o	n		
Z$[3]	C	a	l	c	u	l	a	t	o	r
Z$[4]										

Dimensioning Strings

Before characters can be stored in a string variable, the string variable must be dimensioned. The `dim` statement defines the type of string variable and the size of the string or strings. The `dim` statement also reserves storage space in memory for the string variable. To dimension the strings shown in the previous example, execute –

```
dim A$[10], Z$[4, 10]
```

The subscript 10 indicates that up to ten elements can be assigned to the simple string named A$. The subscripts 4 and 10 indicate that up to 4 strings, each 10 characters in length, (40 elements) can be assigned to the string array named Z$. Exceeding dimensioned limits by assigning more characters than dimensioned, results in `error 59`. Both string and numeric arrays can be included in the same `dim` statement.

Storing Strings

Characters in quotation marks ("text") are assigned to string variables in the same way that values are assigned to numeric variables. To store the characters from the previous example in the dimensioned strings, execute –

```
"-ABC!123*"→A$
"Keeper"→Z$[1]
"Function"→Z$[2]
"Calculator"→Z$[3]
```

Text can't be stored in a string array. Executing

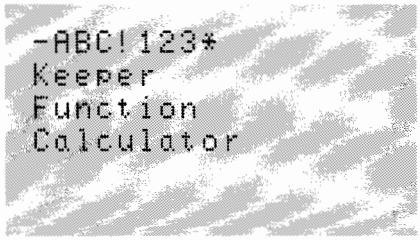
```
"XXX" → Z$
```

results in error 55.

Printing Strings

Using the print statement, strings and portions of strings can be printed. To print the strings just stored, execute –

```
prt A$, Z$[1], Z$[2], Z$[3]
```

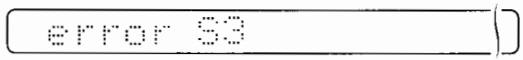


Each string in the string array must be specified to be printed. To specify an entire array of strings, a program loop of some kind must be used. Most programs in this chapter use the Advanced Programming capability of for/next loops where needed.

Specifying a string longer than its dimensioned length results in error 53. For example, execute –

```
Z$[1, 20]
```

And the display shows –



Substrings

A substring is one or more contiguous characters within a string. Using the previous example, ABC is a substring of A$ starting at the second character and ending with the fourth. This substring is indicated by –

```
A$[2, 4]
```

With string arrays, an extra subscript is required to indicate which string in the array is specified. Expressions may be used as subscripts in strings or string arrays. So per, which is a substring of Keeper from the fourth character of Z$[1] to the sixth, can be specified by–

```
Z$[1,4,6]
```
or
```
Z$[1, A, A+2]
```
where the numeric variable A has the value 4.

Since there are no characters following per in Z$[1], this substring (from the fourth character to the end of the current length of the string) can also be specified by –

```
Z$[1, 4]
```

To specify the substring Fun in Z$[2], these subscripts are used –

```
Z$[2, 1, 3]
```

The u in Calculator can be specified by executing –

```
Z$[3, 5, 5]
```

Since a different number of subscripts are required to specify different things, it's a good idea to keep a record of the simple strings and string arrays you're using in a program. (The dim statement can be used in this way.)

Null String

The last string, Z$[4], in the string array is the null string, since it contains no characters. This null string can be specified by executing –

```
Z$[4]
```
or
```
Z$[4, 4, 3]
```
or
```
""
```

Two quotation marks with no intervening characters are considered a null string, but a string that is assigned spaces ("Δ"→A$[1, 80], where Δ represents a space) is not a null string.

The null string can be used when adding more characters to the current length of a string. Using A$ from the previous examples, execute –

```
"X"→A$[10, 10]
A$
```

And the display shows –

The null string can also be used to clear a string. For example, to clear A$ (from the previous example) execute –

```
""→A$[1, 10]
```

String Variable Modification

A string or a substring can be modified by another string or substring. For example, a part of a string can be changed or characters can be added or deleted. The string containing the modification is called the **modifying** string; the string to be modified is called the **destination** string. For example, in the statement M$→D$, M$ is the modifying string and D$ is the destination string. The modifying string can be a string, a substring or text.* The destination string can be a string or substring; it cannot be text.

The length and content of the destination string after modification depend not only on the characteristics (length and content) of the modifying string, but also on the number of subscripts given for the destination string.

Each element or string of a string array can be used in exactly the same way as a simple string. Therefore, the following examples show simple string modification only; string array modifications are done in exactly the same way with an extra subscript (the first) to show which string in the array is being modified.

(If you've just executed the statements from the previous section, press ERASE A EXECUTE before executing the statements that follow.)

Any modification made to a string or substring requires a temporary storage area as large as the modifying string.

*Text is defined as characters within quotation marks.

Destination Strings without Subscripts

When the destination string has no subscript (or one subscript for string arrays) the entire destination string is replaced by the modifying string or substring, and its length and content after modification are the same as those of the modifying string or substring. To illustrate this, execute these statements –

```
dim A$[7],B$[7]
"Atwater"→A$
"Belcher"→B$
A$→B$
B$
```

```
Atwater
```

When the destination string is longer than the modifying string, the modifying string replaces the destination string. All characters of the destination string which are not replaced are truncated. For example –

```
dim C$[11]
"Bellweather"→C$
A$→C$
C$
```

```
Atwater
```

The current length of C$ is now 7 characters; the dimensioned length is 11 characters.

If the modifying string is longer than the dimension of the destination string, `error 59` occurs. For example –

```
"Bellweather"→C$
C$→A$
```

```
error 59
```

String arrays require one extra subscript to indicate which string in the array is to be modified.

Destination Strings with One Subscript

When the destination string has one subscript (or two subscripts for string arrays) the substring is replaced by the modifying string or substring.

(If you've just executed the statements from the previous section, press ERASE A ↵ before executing the statements that follow.)

If the destination substring is equal in length to the modifying string or substring, the modification will not affect the length of the destination string. When executed, these statements illustrate this –

```
dim A$[7]
"Atkins"→A$
"wood"→A$[3]
A$                                  Atwood
```

If the destination substring is longer than the modifying string, the modification causes the destination string to be shortened; the characters not replaced by the modifying string are truncated. These statements, when executed, illustrate this –

```
dim B$[7]
"Atwater"→B$
"kins"→B$[3]
B$                                  Atkins
```

If the destination substring is shorter than the modifying string (or substring), the modification causes the destination string to be lengthened (within the dimensioned length of the destination string). To illustrate this, execute these statements –

```
dim C$[8]
"Atwater"→C$
"kinson"→C$[3]
C$                                  Atkinson
```

Any attempt to lengthen a string beyond its dimensioned length causes error 59. For example –

```
"kinsons"→C$[3]
```

```
error 59
```

If the destination substring is a null string (the current length plus one), the modifying string is attached to the end of the destination string. For example –

```
"Atkin"→C$
"son"→C$[6]
C$
```

```
Atkinson
```

When characters of a modifying string (or substring) are added to a destination string (or substring), they must be contiguous, that is, they must immediately follow the destination string without any unassigned character spaces. If they are non-contiguous, error 53 occurs. For example –

```
dimD$[20]
"Andy"→D$
"Atkinson"→D$[8]
```

```
error 53
```

By assigning blank characters to the string, error 53 is avoided. (Δ represents a space.) For example, execute these statements –

```
"ΔΔΔAtkinson"→D$
[5,15]
D$
```

```
Andy    Atkinson
```

String arrays require an extra subscript to indicate which string in the array is to be modified.

Destination Strings with Two Subscripts

When the destination string has two subscripts (or three subscripts for string arrays), the substring is replaced by the modifying string or substring. Since two subscripts define the beginning and ending characters of the substring to be replaced, truncation of the remaining (unreplaced) characters does not occur. Using an example from the previous section, execute –

```
C$                                  Atkinson
"ken"→C$[3,5]
C$                                  Atkenson
"kin"→C$[3]
C$                                  Atkin
```

(Press ERASE A ↵ before executing the statements that follow.)

When the destination substring is equal in length to the modifying string or substring, the modification won't affect the length of the destination string; the destination substring characters are replaced. For example –

```
dimA$[8]
"Loveland"→A$
"Hom"→A$[1,3]
A$                                  Homeland

"town"→A$[5,8]
A$                                  Hometown

"Down"→A$[1,4]
A$                                  Downtown
```

When the destination string has two subscripts, its length after modification will either be greater than before, or remain unchanged. When the value of the second subscript is greater than the current length of the destination string, the modification results in a lengthened string (within the dimensioned length of the string).

Here's an example that illustrates this. (Δ represents a space.) Execute these statements –

```
dimB$[32]
"AndrewΔ"→B$
"AtwaterΔ"→B$[8,
15]
"Atkinson"→B$[16
,32]
B$
```

```
AndrewΔAtwaterΔAtkinson
```

When characters of a modifying string (or substring) are added to a destination string (or substring), they must be contiguous, that is, they must immediately follow the destination string without any unassigned character spaces. If they are non-contiguous error 53 occurs. For example –

```
"Andrew"→B$
"Atkinson"→B$[16
,23]
```

```
error 53
```

By assigning blank characters to the string, error 53 is avoided. (Δ represents a space.) For example, execute these statements –

```
"Δ"→B$[7,15]
"Atkinson"→B$[16
,23]
B$
```

```
AndrewΔΔΔΔΔΔΔAtkinson
```

String arrays require an extra subscript to indicate which string in the array is to be modified.

String Variable Functions

A string function returns a numeric or string value to an expression. String functions enable you to determine the length and analyze the contents of a string. This is useful when strings with different characteristics (length and content) are processed at the same time, as in entering strings from the keyboard.

Length Function

The length (len) function returns the number of characters in a string or substring.

len (string variable or text)

The current length of the string (or substring) is returned, which is not necessarily equal to the length defined in the dim statement.

The following example uses a simple string to illustrate the length function. If you've run any of the examples from the previous chapter, press [ERASE] [A] [EXECUTE] before executing the following statements –

```
dim A$[32]

"length*width="→
A$

len("length*widt
h=")                    13.00

len(A$)                 13.00
```

In the following example, the last character of A$ is replaced –

```
"*height"→A$[len
(A$)]

A$
```

```
length*width*height
```

Here an addition to A$ is made –

```
"=volume"→A$[len
(A$)+1]

A$
```

```
length*width*height=volume
```

The length of a part of A$ can be found using the length function –

```
len(A$[21])
```

```
6.00
```

String arrays require an extra subscript to indicate which string in the array is being used in the function.

Position Function

The position (pos) function determines the position of a substring within a string.

pos (in string variable or text , of string variable or text)

If the second string is part of the first, the value of the function is the position of the beginning character of the second string within the first starting from where you began searching. If the second string is not a substring of the first string or if the second string is the null string, the value of the function is zero.

Simple strings are used in the following example to illustrate the position function. Execute these statements –

```
A$
                    length*width*height=volume

pos(A$,"width")                                  8.00

dim C$[10]
"width"→C$
pos(A$,C$)                                       8.00

"*width=area"→A$
[pos(A$,"*")]

A$                                   length*width=area

pos(A$,"area")                                  14.00
```

String arrays require an extra subscript to indicate which string in the array is being used in the function.

Value Function

With the value (val) function, the numerical value of a string or a substring of digits can be used in calculations. (Normally the elements in a string are not recognized as numeric data and can't be used in calculations.)

val (string variable or text)

The first character to be converted in a string using the value function can be a digit, a plus or minus sign, a decimal point or a space. Leading plus signs are ignored; leading minus signs are counted. An odd number of minus signs is equal to a minus sign; an even number of minus signs is equal to a plus sign.

Numerical data entries can be combined logically with input text. All contiguous numerics are considered a part of the number until a non-numeric* is reached in the string. This means that a string can contain more than one number. The first character after leading spaces, plus signs or minus signs must be a digit or a decimal point. If the leading part of a string is not valid as a number according to the rules of the enter statement, an error occurs, unless flag 14 is set. If flag 14 is set, the default value (zero) is substituted for the number to be returned and flag 15 is automatically set (to 1) to indicate the substitution.

The value function requires a temporary storage area equivalent to the size of the portion of the string used. If there is not enough memory for this temporary storage area, error 40 will result.

Simple strings are used in the examples below to illustrate the value function. The simple string (E$) contains a name (Atkinson), a social security number (094-30-6441) and a balance due ($250).

```
dim E$[32]

"Atkinson+094306
441*+250"→E$

val(E$[21])
```

```
 250.00
```

*The * character is recognized as the "exponent of base 10", when it follows a numeric.

After a payment of $100, the balance due is $150, as shown below.

```
val(E$[21])-100
```

```
 150.00
```

Other operations can be performed, using the value function –

```
val(E$[20])*val(
E$[pos(E$,"*")+3
])
```

```
 12500.00
```

Alpha characters cannot be converted using the value function –

```
val(E$)
```

```
 error 54
```

```
sfg14;val(E$)
```

```
 0.00
```

```
flg15
```

```
 1.00
```

String arrays require an extra subscript to indicate which string in the array is being used in the function.

The string function, covered next, is the opposite of the value function.

String Function

The string (str) function converts a numeric value into an equivalent ASCII* string using the current fixed or float setting. If the numeric value is positive, the resulting string has a leading blank. The string function is the opposite of the value function.

str(expression)

The string function is illustrated using the simple string from the value function examples on the previous pages. The examples use E$ which contains a name (Atkinson), a social security number (094-30-6441) and an amount due ($250). Using the value function, the amount due is located in E$. Then a payment of $100 results in a balance due of $150. To return the $250 to E$, the string function is used. Execute these statements –

```
str(250)→E$[21]
E$
```

```
 Atkinson→094306441*→ 250.00
```

or

```
250→X
str(X)→E$[21]
E$
```

```
 Atkinson→094306441*→ 250.00
```

The destination string must be dimensioned to accommodate the digits added due to the current flt or fxd setting and for the + or – sign.

* American Standard Code for Information Interchange.

Character Function

The character (char) function converts a numeric value (modulo 256) to an ASCII character. Any of the 94 alphanumeric characters and symbols from the 9825 keyboard, and 34 other characters which are not on the keyboard can be output by executing the character function. (See the complete character set in the Appendix.)

char (expression)

All of the 128 characters can be displayed using the following program.

```
0: fxd 0
1: for I=0 to
 127
2: dsp char(I)
3: wait 500
4: next I
5: end
```

The character function can also be used to output control characters to devices like a tape reader, 9871A printer or a digital voltmeter.

The character function is the opposite of the numeric function, which is covered next.

Numeric Function

The numeric (num) function returns the ASCII decimal value of a single character. The numeric function is the opposite of the character function.

num (single character of a string variable or text)

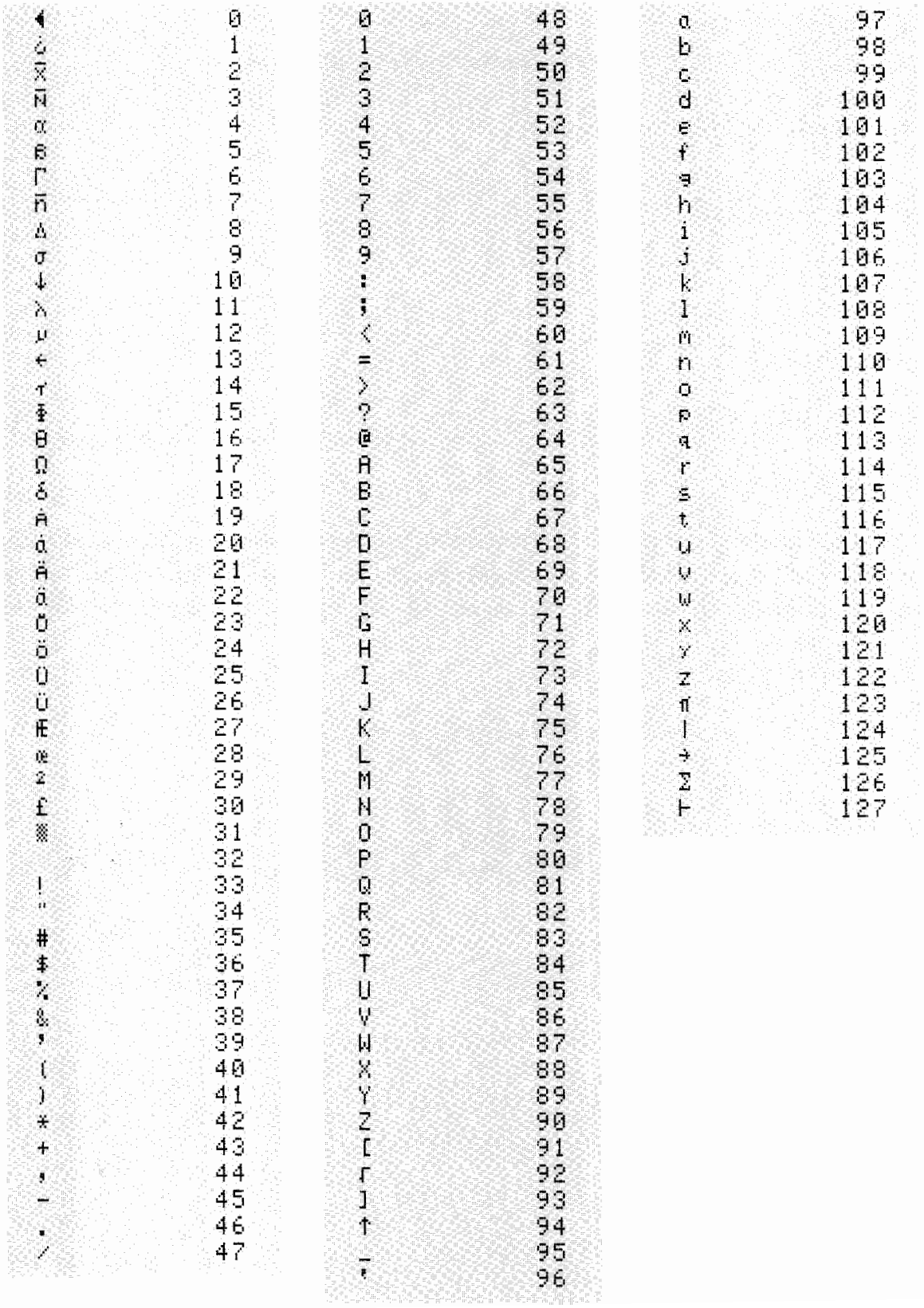
For example, execute these statements –

```
num("A")
65.00
num("B")
66.00
num("C")
67.00
```

All 94 alphanumeric characters and symbols from the 9825 keyboard, and 34 others not on the keyboard, have a decimal value (or code) for their binary equivalent. These decimal values and the equivalent character or symbol are printed using the character function in the following program.

```
0: fxd 0
1: for I=0 to
127
2: prt char(I),I
3: next I
4: end
```

Here's the printout of the internal character set and equivalent decimal codes. The internal character set for decimal codes 128 through 255 is the same as for decimal codes 0 through 127, but are displayed with a flashing cursor.



The character and numeric functions can be used to store and retrieve numbers in the range 0 to 255 using only 1 byte of memory per number. Each number is represented in memory by one of the 256 characters of the internal printer character set. For example, in the following program the five test scores on the right are stored in T$ –

```
0: dim T$[5]
1: for I=1 to 5
2: ent X
3: char(X)→T$[I,
  I]
4: next I
5: end
```

Student	Score
1. Andy Atkins	88
2. Tom Atwater	78
3. Jim Belcher	65
4. Jerry Hafford	100
5. Rob Rood	99

To recover a test score, the numeric function is used. For example, the last score can be displayed by executing –

```
num(T$[5,5])
```

```
99.00
```

The length of the string used to store the values is limited by memory size only. However, only values between 0 and 255 can be stored in this way.

String arrays require an extra subscript to indicate which string in the array is being used in the function.

Capital Function

The capital (cap) function converts lower case alphabetic characters to upper case without modifying the original string. This enables you to compare strings for sorting or alphabetizing without regard to upper or lower case characters.

cap (string variable or text)

Strings or substrings of string variables can be converted using the capital function. For example –

```
dim X$[3],Y$[3]
"yes"→X$
cap(X$)
```

```
YES
```

```
cap("yes")
```

```
YES
```

String arrays require an extra subscript to indicate which string in the array is being used in the function.

A temporary storage area is required by the cap function equivalent in size to the portion of the string used. If there is not enough memory for this temporary storage area, error 40 will result.

String Variable Operations

Relational Operators

The if statement allows comparison of strings or substrings. All of the relational operators allowed in numerical comparisons can be used for string comparisons.

=	equal to
>	greater than
<	less than
>= or =>	greater than or equal to
<= or =<	less than or equal to
# or <> or ><	not equal to

Here's an example that uses a relational operator to illustrate conversational programming –

```
 0: dim A$[10]
 1: ent "Do you
 wish to continu
 e?",A$
 2: if cap(A$)="Y
 ES";sto 4
 3: stp
 4: dsp "Continue
 ...."
```

When the enter statement prompt is displayed, the answer is keyed in.

Within the computers memory, each character contained in a string is represented by a standard ASCII* decimal equivalent code (see the Reference Tables appendix). When two string characters are compared, the smaller of the two characters is the one whose decimal code is smaller. For example 2 (decimal code 50) is smaller than R (decimal code 82).

* American Standard Code for Information Interchange.

In some cases, such as alphabetic sequencing problems, strings must be compared for conditions other than "equal to" or "not equal to". For example, to arrange a number of strings in alphabetical order, the following type of string comparison is used –

```
0: dim A$[20],
 B$[20],Z$[20]
1: ent A$,B$;if
 A$<B$;sto 3
2: B$→Z$;A$→B$;
 Z$→A$
3: prt A$,"is
 less than",B$
```

Strings are compared, character by character, from left to right until a difference is found. If one string ends before a difference is found, the shorter string is considered smaller. For example, execute these statements –

```
"Atkinson">"Atki
ns"
```

```
 1.00
```

Concatenation Operator

The concatenation operator (&) links strings or substrings in order from left to right.

string variable or text & string variable or text [&...]

The resulting string is the total number of characters in all of the strings. Strings or substrings can be linked. Press [ERASE] [A] [(] before executing the following statements –

```
dimA$[15],B$[15]
,C$[15],D$[45]

"Andrew"→A$

"ΔAtwater"→B$

"ΔAtkinson"→C$

A$&B$&C$→D$

D$
```

```
 Andrew Atwater Atkinson
```

In general, the concatenation of N strings requires a temporary storage area equal to $2*(\text{len}(1_{st}) + \text{len}(2_{nd}) + \ldots) + \text{len}(N_{th} - 1)) + \text{len}(N_{th})$ strings. If insufficient memory is available for this temporary storage, error 40 will result.

String Input and Output

Enter Statement

The enter (ent) statement allows string variables to be input from the calculator keyboard during program execution. Up to 80 characters may be entered into a string at one time. To enter a string longer than 80 characters, several substrings of up to 80 characters each may be entered. For example –

```
 0: dim A$[160]
 1: ent A$[1,80]
 2: ent A$[81,
160]
```

After the first 80 characters are entered, a second data request - A$[81, 160]? - is displayed so the second 80 characters may be entered. A for/next loop (Advanced Programming) can be used to enter very long strings.

String arrays require an extra subscript to indicate the appropriate string in the array.

Strings and numeric variables can be used together in an enter statement –

```
 0: dim C$[10],
  D$['0]
 1: ent C$,D$,C,D
```

In conversational programming, the enter statement assigns text to string variables from the keyboard while a program is running. The destination string follows the same rules as the destination string in string assignment and modification. For example –

```
0: dim A$[160]
1: ent "NAME?",
 A$
```

When line 0 is executed, the prompt NAME? is displayed. When the user types in a name and presses CONTINUE, that name is assigned to A$.

If a literal prompt is not given, the destination string variable is displayed as a prompt. For example –

```
1: ent A$[3,3]
```

When the line above is executed, the display shows –

```
A$[3,3]?
```

When the user types in the appropriate data and presses the CONTINUE key, the data is assigned to A$[3, 3].

Print Statement

The print (prt) statement can be used to print string characters. With the print statement, an automatic carriage return-line feed (cr/lf) occurs when a new string is output. The string characters are left justified when output.

Here's a program that prints out the internal printer character set using the print statement.

```
0: fxd 0
1: dim Z$[128]
2: for I=0 to
 127
3: char(I)→Z$[I+
 1,I+1]
4: next I
5: prt Z$
6: end
```

When executed, this is printed –

```
◀¿ÑαßΓπΔσ↓λμ÷τ‡
ΘΩδÄäÄäÖöÜüß²£▒
 !"#$%&'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[Γ]↑_
'abcdefghijklmno
pqrstuvwxyzπ|→Σ⊢
```

Strings, the rows of string arrays, numeric variables and constants may be included together is a print statement, as shown in lines 6 and 7 of the following temperature conversion program. (A represents the month, B represents the day and F represents the Fahrenheit temperature.)

```
0: fxd 0
1: dim C$[20],
 F$[2,20]
2: ent A,B,F
3: "Celsius"→C$
4: "Fahrenheit"→
 F$[1]
5: 5(F-32)/9→C
6: prt "Today
 is:",A,B,1977;
 spc
7: prt "Temperat
 ure is:",F$[1],
 F,C$,C
8: end
```

```
Today is:        3
                15
              1977

Temperature is:
Fahrenheit      55
Celsius         13
```

Display Statement

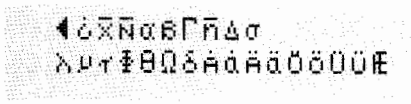
The display (dsp) statement can be used to display a maximum of 80 characters. Only 32 of these characters can be viewed at one time. The display control keys (← and →) are used to shift the display to the left and right to view all 80 characters.

Write Statement

The write (wrt) statement is a General I/O ROM statement which allows strings to be output to external devices, as well as to the internal printer.

The write statement works like the print statement, without automatic carriage return-line feeds. The write statement outputs the entire string (as does prt). Using the example from the prt statement description, the write statement replaces print in line 5. The number following wrt is the select code of the output device being used (16 is the internal printer select code and 0 is the display select code). Compare the outputs –

```
0: fxd 0
1: dim Z$[128]
2: for I=0 to
 127
3: char(I)→Z$[I+
 1,I+1]
4: next I
5: wrt 16,Z$
6: end
```



Characters 10 (line feed) and 13 (carriage return) cause a carriage return and line feed. When the internal printer reaches the tenth character, a line feed occurs. Then the next 16 characters are output (except the thirteenth, which is a carriage return). Since line feeds are not automatic, the characters following the carriage return are not printed.

The write statement can be used with free-field format or with format specifications. The following example uses the write statement without format specs. Execute these statements –

```
dimA$[10]
"ABCDEFGHIJ"→A$
wrt16,A$[5]
```

```
EFGHIJ
```

```
wrt0,A$[6,6]
```

```
F
```

```
wrt0,A$
```

```
ABCDEFGHIJ
```

When format specifications are used, the spec that sets the width of a character field is C followed by the width of the field in number of characters. (A value can precede C to indicate the number of strings to be output.) The strings are right justified when output. Execute these statements –

```
fmtc15;wrt16,A$
```

```
ABCDEFGHIJ
```

```
fmtc10;wrt16,A$
```

```
ABCDEFGHIJ
```

```
fmtc10;wrt16,A$[
1,5]
```

```
ABCDE
```

```
fmtc15;wrt16,A$[
1,5]
```

```
ABCDE
```

```
fmtc5;wrt16,A$[1
,5]
```

```
ABCDE
```

When the string is longer than the specified field, an overflow condition exists and dollar signs are output.

```
fmtc5;wrt16,A$
```

```
$$$$$
```

Complete information about the write and format statements can be found in the I/O Control Reference.

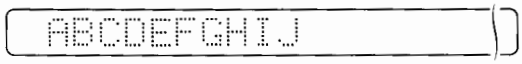
Read Statement

The read (red) statement is a General I/O ROM statement used to input data. The read statement can be used with free-field format or with format specifications.

Here's an example using free-field format that reads A$ using an input device that responds to select code 3 (the number following red), such as a tape reader. Assume this string is on tape –

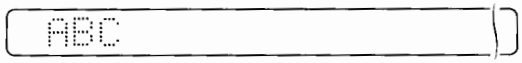
ABCDEFGHIJ (LF)

```
dim B$[10]

red 3, B$

dsp B$
```



When format specs are used, as with the write statement, parts of a string can be read. Using the previous example, the first three characters only can be read by executing these statements –

```
fmt c3

red 3, B$

dsp B$
```



Strings, string arrays, numeric variables and constants may be used together in a read statement. Complete information about the read and format statements can be found in the I/O Control Reference.

Loading and Recording Strings

Each character of a string requires 1 byte of memory for storage. Extra bytes of memory , called overhead, are required to store strings and string arrays on tape. For example, to store an array dimensioned A$[X, Y], it takes 6+2X bytes of overhead plus the data requirements of XY bytes (plus 1 byte if XY is odd) to store X strings.

Recording Data

To store string variables on tape, first mark the number and size of the file or files required and then use the record file (rcf) statement.

rcf file number ; string variable [; string variable...]

When the record file statement is executed, the strings found in the list of data items are recorded into the file number specified, on the current track. Strings in the list must be entire simple strings or entire string arrays. For example –

```
0: dim A$[5],
 B$[10,20],C
    •
    •
    •
3: rcf 3,A$,B$
```

When recording a list of items, such as A$, B$, X, the items must appear in the same order as allocated in a previously encountered dim statement. For example –

```
0: dim A$[5],X,
 B$[10,20],Y,Z
    •
    •
    •
3: rcf 5,A$,X,
 B$,Y,Z
```

Loading Data

To load string variables into the calculator's memory from tape, the load file (ldf) statement is used.

ldf file number, string variable [, string variable...]

When the load file statement is executed, the data in the file specified from the current track is loaded into the calculator's memory. A dim statement must be executed before the load file statement. The list of variables in the load file statement must be in the same order as previously allocated in the dim statement.

String variables can't be combined in the load file statement. For example, if A$ and B$ are dimensioned and then linked later in a program (A$&B$→C$), they must be recorded and loaded as dimensioned –

```
0: dim A$[5],
 B$[10],C$[15]
    •
    •
    •
4: A$&B$→C$
5: rcf 1,A$,B$
    •
    •
    •
9: ldf 1,A$,B$
```

Chapter 7
Table of Contents

Notes

Chapter 7

Systems Programming

The Systems Programming ROM extends the 9825 language to include capabilities for remote keyboard operation, program self-modification, run-time read/write storage allocation, and intelligent terminal emulation. This ROM is available as a plug-in acessory (98224A) for 9825A and S computers. The ROM is included with the 9825T.

The Systems Programming ROM uses 160 bytes of user read/write memory when installed in the 9825A or S. The Systems Programming ROM and the 98211A Matrix ROM cannot be used simultaneously in a 9825A or S. If the Matrix ROM is installed, it must be removed before installing the Systems Programming ROM. Both ROMs can be operated simultaneously with the 9825T.

9825A/S Requirements

Several of the Systems Programming statements require the presence of other ROMs. The relationships of the statements and their requirements are shown in the following table:

Mnemonic – ROM Option Requirements

Mnemonic	Description	ROM Option Necessary
on key	Keyboard Interrupt Routine enable	None
key	Key Buffer Empty function	None
kret	Keyboard Interrupt Routine return	None
asc	Keycode to ASCII Conversion function	None
bred	Read Transfer Buffer function	Extended I/O[1] and General I/O
eol	End-of-Line specification	General I/O
wsm	Write Serial Mode Word statement	General I/O
wsc	Write Serial Control Word statement	General I/O
rss	Read Serial Status Word function	General I/O
rkbd	Remote Keyboard Enable	General I/O
nal	Next Available Line function	None
%	Free Syntax prefix	None
store	Store String instruction	String[2]
avm	Available Memory function	None
cln	Current Line Number function	None

1 Extended I/O Binary Tape can not be used.

2 The String Programming ROM is not necessary if only literals are to be stored.

With the 9825A Option 003 (32K R/W memory), it is necessary to load Extended I/O as a binary tape. The Systems Programming ROM will not operate properly if the Extended I/O binary tape is loaded into the 9825A. Do not use the Systems Programming ROM and the Extended I/O binary tape concurrently, as erratic and unpredictable calculator operation will result.

Intelligent Terminals

An intelligent terminal should represent a logical extension of the capabilities of a basic data terminal. The minimum facilities of a basic data terminal usually include a keyboard for operator entry, a printer or CRT display for data records and communications link status information, and a serial interface to the central processor. An intelligent terminal should include the minimum terminal functions and be user programmable.

The programmability of an intelligent terminal allows the user to define key functions, set special formats, establish communication formatting, and in the case of the 9825, perform off-line computing as well. Some of the features an intelligent terminal makes possible include:

- Extension of the throughput capabilities of an overloaded central processing system;
- Faster effective turnaround time with much of the data processing done locally;
- Local formatting of input and output data records;
- Local concentration of data, with high speed block data transmission;
- Local content error correction and editing;
- Appending local, variant data, such as operator code, date and security information;
- Reduction of repeated communication link transfers due to local message correction and verification.

Use of the Systems Programming language can provide all of the features of an intelligent terminal and additional features that aren't usually available. The 9825 contains a high speed data cartridge for temporary off-line data storage if the communication link goes down, and an internal printer to list operator instructions and prompt messages.

The internal printer and the display can be treated as external devices by the program, and can be used to list two different message levels simultaneously. For example, the display could be used to list the data as typed by the operator and the printer utilized to update the communications link and system status.

Intelligent Terminal Instructions

The Intelligent Terminal Instructions facilitate segmentation of the internal 9825 computer "peripherals" into program controllable modules. With the three instructions "on key", "key", and "kret" the programmer can set up the 9825 keyboard as an external peripheral input device. The "asc" function returns the ASCII code equivalent of a 9825 keycode (which can be output to an ASCII coded printing device such as a teletype). The "eol" specification extends the generality of the communication format by allowing the programmer to specify output line delimiters other than the standard carriage return/line feed.

9825A/S External ROM Requirements

Mnemonic	Required ROM Option	Description
on key	None	Keyboard interrupt directive
key	None	Key buffer empty function
kret	None	Keyboard interrupt routine return
asc	None	keycode to ASCII conversion
bred	Extended I/O ROM and General I/O ROM	Read transfer buffer function
eol	General I/O ROM	Line delimiter specification

Keyboard Interrupt Service Enable

The "on key" statement enables the programmer to establish the 9825 keyboard as an external input device, operating on an interrupt service level.

on key "Routine Name" [; Flag Number]

The routine name parameter may be either a string or a literal, and the flag number parameter either a fixed value or an expression.

Routine Name: Specifies the label of the keyboard service routine that is to process keyboard interrupts.

Flag Number: (Optional) specifies which flag to set if the key buffer overflows. If a flag number is specified, error C5 will not be issued for a key buffer overflow. The flag number may specify any one of the 16 system flags, however flags 14 and 15 should not be used if any math processing is being performed.

When activated by an "on key" statement, a dedicated 16 character circular buffer is established, as well as a link to the "on key" service routine. This routine (specified in the Routine Name parameter) changes the status of the system keyboard from calculator controller to input device (with the exception of the RESET key).

Thereafter, when a key is pressed, the keycode is placed into the 16 character circular buffer and end-of-line interrupt service is requested. If no other interrupts are pending, program control is passed to the keyboard service routine for processing. If any interrupts other than a keyboard interrupt occur before the end of the current line, they will be processed in descending order by select code until all pending interrupts have been processed. (Refer to the "on key" execution chart, execution priority block diagrams, and program execution flowchart shown later.) The 16-character key buffer allows for execution of long program lines and multiple interrupt processing before the key buffer overflows.

A key buffer overflow results if more than 16 keys are pressed before program control transfers to the "on key"service routine. An overflow is indicated either by error C5 or by setting the "on key" flag (use the optional Flag Number parameter).

The "on key" statement specified without parameters disables the on key service routine, clears the key buffer, and returns the 9825 to normal keyboard operation. The "on key" optional flag (if used) is not affected, and it should be noted that "on key" cannot be disabled from live keyboard. (The "on key" statement effectively disables live keyboard.)

NOTE

Whenever the "on key" statement is executed the key buffer is cleared and any data remaining in the buffer will be lost. This applies to the "on key" statement with or without parameters.

NOTE

Do not execute a branch command (ldf, ldp, jmp, gto, etc.) from within the "on key" routine if program execution will branch from the routine without executing a kret. The result will be that no more keys will be processed from the keyboard.

Key Buffer Empty Function

```
key
```

Parameters are not required for the "key" function.

The "key" function returns the earliest entered unprocessed keycode in the key buffer. When all keycodes have been processed by the "on key" routine, `key` returns a value of zero and `kret` execution is allowed. If an exit from the subroutine is attempted (by a "kret") with any remaining keycodes in the key buffer, the "on key" routine will be restarted. (See the "kret" execution flowchart later in this chapter.)

Example:

```
0: on key "kbd"
1: ato +0
```

0,1: Enable on key service routine "kbd", and hang in loop.

```
2: "kbd":
```

2: on key routine label.

```
3: dsp chr(asc
  key));wait 500
```

3: Display each consecutive keycode in buffer.

```
4: kret
+9272
```

4: When buffer is empty, return.

> **NOTE**
>
> The "kret" will cause an immediate routine reentry unless the key buffer has been emptied.

Keyboard Interrupt Routine Return

```
kret
```

Parameters are not required for the "kret" syntax.

The "kret" statement serves to return program execution to the main program after emptying the key buffer. The reentry point of the main program is the program line that would have been executed before control was passed to the keyboard service routine.

If kret is executed before emptying the key buffer, control is not transferred to the main program, and the keyboard service routine is restarted. (See the on key execution flowchart.)

Keycode to ASCII* Code Conversion Function

The "asc" function provides a single statement conversion from 9825 keycodes to an ASCII equivalent code. It is useful when outputting 9825 keycodes to an external ASCII device.

```
asc  keycode
```

The keycode parameter may be either a fixed value or an expression.

The "asc" function returns the ASCII equivalent of a 9825 keycode, including the system control keys and special function keys. The value returned by the "asc" function for the shifted function keys will be greater than 127 decimal, and therefore out of range of the ASCII character set. If the Extended I/O ROM is present, the "asc" function will return an octal or decimal value depending on the oct/dec mode of the calculator. If the octal mode is set, the value returned by "asc" will be in octal, which is an improper format for the "char" function of the Strings ROM. (In this case use the octal-to-decimal function to restore the "asc" value to decimal; refer to the Binary I/O chapter of the I/O Control Reference.

Example:

Typing a key on the keyboard will result in the ASCII character and code shown in the left of the display and the 9825 internal character and code to the right.

```
0: on key "kbd"
1: ato +0
2: "kbd":
3: key→K:dsp
 char(asc K),
 asc K→char(K),
 K:wait 500
4: kret
*25604
```

* ASCII: American Standard Code for Information Interchange.

Read Transfer Buffer Function

The "bred" function facilitates use of the 9825 over a high speed data link, offering a means of reading an active interrupt input buffer without having to wait for the buffer transfer to run to completion.

```
bred ("Buffer Name")
```

Buffer Name: A string or literal parameter specifying the name of the transfer buffer to be emptied. The buffer specified must be an active*, interrupt type, byte input buffer (type 1) as implemented by the Extended I/O ROM. An error (C4) is displayed if "bred" is executed specifying a non-interrupt type or non-busy buffer. If the "bred" function is used to read a transfer buffer, the General I/O "red" statement should not be used. Using both "bred" and "red" on the same buffer disrupts the buffer pointers and incorrect data is read from the buffer. A more detailed discussion of the transfer buffer pointers is in the I/O Control Reference.

Use of the "bred" function in conjunction with the Extended I/O transfer buffer facilitates 9825 data communications on a high speed data link. The "bred" function allows the programmer to implement a high speed input buffer which is emptied at memory speed without having to run the buffer transfer to completion. This input scheme presents a broader data input window to incoming messages than does a double buffer input scheme of alternating input transfer buffers. The double buffer input method offers only limited control over the time window between buffer available periods, due to the necessity of completing the current program line before acknowledging a buffer completion interrupt. If a long program line is being executed when a buffer terminates, the time delay encountered before reenabling another input buffer may be too large to insure reception of all incoming data when operating at high data rates.

* The transfer operation must be in effect.

When high speed data communication is implemented on the 9825, use of the "bred" buffer read function on a frequent basis is suggested. Interrupts are disabled by "bred" for a time span dependent upon the number of bytes in the buffer to be read out, so it is suggested that the program be designed to execute a "bred" periodically. If a buffer overflow occurs, possible alternatives are to add more "bred" instructions to the program or to execute bred within a subroutine which is called from several program locations.

Example:

```
 0: dim B$[650]
 1: buf "Buff";
  200,1
 2: eir 12;4;oni
  12;"overrun"
 3: tfr 12,"Buff"
 4: "loop":
          •
          •
          •
          •
          •
          •
          •
          •
          •
15: bred("Buff")
  →B$[len(B$)+1]
16: gto "loop"
17: "Overrun":er
  t "More frequen
  t bred needed."
```

Line 1 establishes a type 1 buffer of 200 bytes ("Buff").

Line 2 enables the peripheral on select code 12 for a full buffer interrupt routine (eir 12,4).It also locates the proper service routine for this interrupt (oni 12, "overrun").

Line 3 starts the transfer operation into "Buff".

Line 4-14 are program lines that process the incoming data.

Line 15 initiates a bred operation on "Buff", specifying the contents of the buffer to go to B$.

Line 16 returns to "loop".

Line 17 is executed on a buffer completion interrupt. If this happens, "bred" must be executed more frequently.

End-of-Line Specification

The end-of-line sequence specification furnishes the programmer with a means of substituting any character sequence (up to seven characters) for the General I/O carriage-return/line-feed for tailoring output to the needs of the external device.

eol [eol Character] [, eol Character$_2$]...[, eol Character$_7$] [, − eol Sequence Delay]

From zero to seven eol Characters may be specified; each may be a fixed value or an expression. The Sequence Delay parameter (if specified) must be given as a negative value, and may be either a fixed value or an expression.

eol Character: Is the numeric value of each character code to be output as an end-of-line delimiter. The maximum value that may be specified for an eol character is 127 decimal, as only 7-bit characters are transmitted. The eol characters are fixed at the time the eol specification is executed, and the octal/decimal mode setting of the calculator will determine the interpretation of the eol character value. This value is not reevaluated when the octal/decimal mode is switched subsequent to the eol specification.

eol Sequence Delay: Specifies the milliseconds of delay between output of the last character of an eol sequence and the start of the next line of output. The maximum possible delay is 32768 milliseconds (decimal value), allowing a flexible approach to a peripheral's physical requirements. (For example, some teletype printers require about 200 msec after performing a carriage return before being ready for new characters.)

The end-of-line specification is useful for formatting output to specialized devices such as the HP 2640 Terminal. The 2640 terminal requires specific codes in an end-of-line sequence to keep the display in the special enhancement mode on the next display line. Since the "eol" sequence specification may be executed at any time, it is possible to extensively reformat output to a device by specifying tabs, spaces, double spaces, or whatever sequence is desired, as necessary.

In operation, the eol sequence is substituted for the carriage-return/line-feed delimiters of the General I/O format. This substitution affects output to any device using the statements "list#" and "wrt" (General I/O), and "cat" (mass storage).

The General I/O format statement ("fmt") is also affected by the eol sequence specification. The slashes (new line) will cause an eol sequence to be output to the specified device instead of a carriage-return/line-feed, and the suppress line-feed (z) will suppress an eol sequence output.

Examples:

```
eol 13,10,32,32,32,32,32
```

Changes format to carriage return, line feed, and five spaces.

```
fmt1,/,/,c20
```

This format will output two eol sequences and a twenty character string.

Serial Interface Control Instructions

9825A/S External ROM Requirements

Mnemonic	Required ROM Option	Description
wsm	General I/O	Write Serial Mode Word statement
wsc	General I/O	Write Serial Control Word statement
rss	General I/O	Read Serial Status Word function
rkbd	General I/O	Remote Keyboard Enable/Disable

Write Serial Control Word Statement

The "wsc" statement insulates the programmer from the complex control register access sequence for the 98036A Serial Interface. A single statement is all that is necessary to access the 98036A control word, making the implementation of specialized I/O formats a much simpler task with the Systems Programming ROM.

wsc Select Code , Control Word

Parameters specified may be either fixed values or expressions.

Select Code: Specifies a 98036A Serial Interface select code set to the range [2 ≤ select code ≤ 15]. If the interface specified by the select code is not a 98036A, or if no interface is set to the specified select code, error C9 is issued. Extended I/O device names are disallowed.

Control Word: Specifies a bit pattern to be written into the control register (R4D) of the 98036A Serial Interface. Note that the value of the control word (mod 256) follows the octal/decimal mode setting of the calculator (for Extended I/O ROM only), and is interpreted accordingly. (Bit 6 is masked out to avoid resetting the 98036A.)

Write Serial Mode Word Statement

The "wsm" statement accesses the mode register of the 98036A Serial Interface with a single statement, reducing the programming necessary to reconfigure the 98036A mode word. This function is useful when temporarily logging on to a serial I/O link which uses a word format different from the one set by the 98036A mode switches.

wsm Select Code , Mode Word [, Control Word]

Parameters specified may be either fixed values or expressions.

Select Code: Designates a 98036A select code with the same specifications and limitations as described for the "wsc" function.

Mode Word: Specifies a bit pattern to be written into the R4C register of the 98036A Serial Interface. Note that the value of the mode word follows the octal/decimal mode setting of the calculator (Extended I/O ROM only), and is interpreted accordingly.

Control Word: (Optional; default value = 5) If a value different from the default value is desired, it can be specified as a parameter to the "wsm" syntax. See the "wsc" syntax for the 98036A control word (R4D) details.

Read Serial Status Word Function

The "rss" function returns the contents of the 98036A status register, giving the programmer easy access to the current status of the serial I/O link.

rss Select Code

Parameters may be specified as either fixed values or expressions.

Select Code: Designates a 98036A select code with the same specifications and limitations as described for the "wsc" function.

The 98036A status word (register R4E) is accessed by the "rss" function and returned as a value interpreted according to the octal/decimal mode setting of the calculator. The following table describes the bit position functions of the R4E status word:

> **NOTE**
>
> When using the "wsc", or "wsm" commands, a parameter error could leave the 98036A in an undefined state. Use care when selecting the parameters for these functions, as data loss could result if the interface locks up. If this state is encountered, it is necessary to reset the 98036A.

Remote Keyboard Statement

The "rkbd" statement enables a remote keyboard to control the 9825A over a serial data link through the 98036A interface.

rkbd Select Code [, Code Type]

Parameters may be either fixed values or expressions.

Select Code: Must specify the select code of a 98036A Interface, and must be in the range [2 ≤ select code ≤ 7].

Code Type: Specifies the remote keyboard code interpretation as follows:

Code Type = 0: ASCII keyboard
Code Type = 1: 9825 keycode keyboard

If code type is not specified, a default value of zero is assumed (ASCII).

The "rkbd" statement operates in conjunction with the 98036A Serial Interface to enable full duplex remote operation of the 9825. This capability allows the 9825 to be used with a multiline display and to be controlled remotely while the calculator is operating in a hostile or inaccessible environment. All characters sent to the 9825 from the remote keyboard are echoed to the remote display, allowing contiual monitoring of the data link status.

When operating with a remote keyboard, the local keyboard is not disabled and characters generated by the 9825 keyboard are not transmitted to the remote display. Error messages are treated as local data and are not transmitted to the remote display. To enable error message monitoring on the remote display, the error trapping facility of the Extended I/O ROM must be used in conjunction with the "wrt" statement. (Note that an "rkbd" interface may be written to, but not read from, by the 9825.)

Example:

```
3: on err "trape
rr"
    .
    .
17: "traperr":wr
t 6,"error",
char(ron):ern
,line:erl
sto 3
```

The error recovery routine "traperr" outputs the error number and the line it occurs in to the remote keyboard set to select code 6.

Pressing the calculator "Reset" key will take the calculator out of the remote keyboard mode.

To prevent erroneous character transmission over the data link, the interface character format (#of stop bits, parity, #character bits) should be identical for the remote keyboard and the 9825. When the calculator is operating in the ASCII mode, the input characters are masked to seven bits. When operating in the 9825 keycode mode, the interface should be configured for 8 bit characters, or the shifted special function keycodes will be inaccessible.

Some peripherals, such as the HP 2640 Terminal, have block output capability and can transmit a line or more of characters at a time. If block transmission is to be used with a 9825 enabled for remote keyboard operation, a data rate of not higher than 110 baud should be used. (For large block transmissions use 50-75 baud.)

> NOTE
>
> Buffered I/O operations should not be used with a 98036A configured as a remote keyboard interface, as erratic calculator operation will result.

Limited editing of 9825 program lines is possible from the remote keyboard by using the "list#" statement to output selected program lines to the remote terminal, however the 9825 cursor position is not accessible and it is necessary to retype the entire program line. The remote edit sequence for line 7, interface select code 6 becomes:

`list#6,7,7`	(typed at remote keyboard)
(LF)	(line-feed = "execute")
`fetch 7`	(typed at remote keyboard)
(LF)	(line-feed = "execute")
(Retype edited version of line.)	(typed at remote keyboard)
(CR)	(carriage-return = "store")

Although remote control of 9825 operation is possible with the "rkbd" statement, remote keyboard editing is awkward (as demonstrated above) and not recommended for extensive program development.

The ASCII to keyboard function chart in the appendix relates ASCII control codes to 9825 functions, and is included for reference when using an ASCII coded remote keyboard with the 9825. ASCII control codes do not generate locally displayable characters, and it may be difficult to keep track of calculator operations. Typing out commands is therefore recommended so the operator can have a record of calculator operation for reference.

Power-Up Remote Keyboard Operation

Upon power-up, the 9825 checks select codes 2 through 7 (in descending order) for a 98036A Interface configured for remote keyboard operation. The interface with the highest select code configured for remote keyboard operation will be used for the system remote keyboard.

To set up the 98036A Interface for power up remote keyboard operation, two jumpers must be located and changed as described below:

1. Disassembly of the 98036A Interface:

    a. Remove the four screws that hold the rear housing to the front housing.

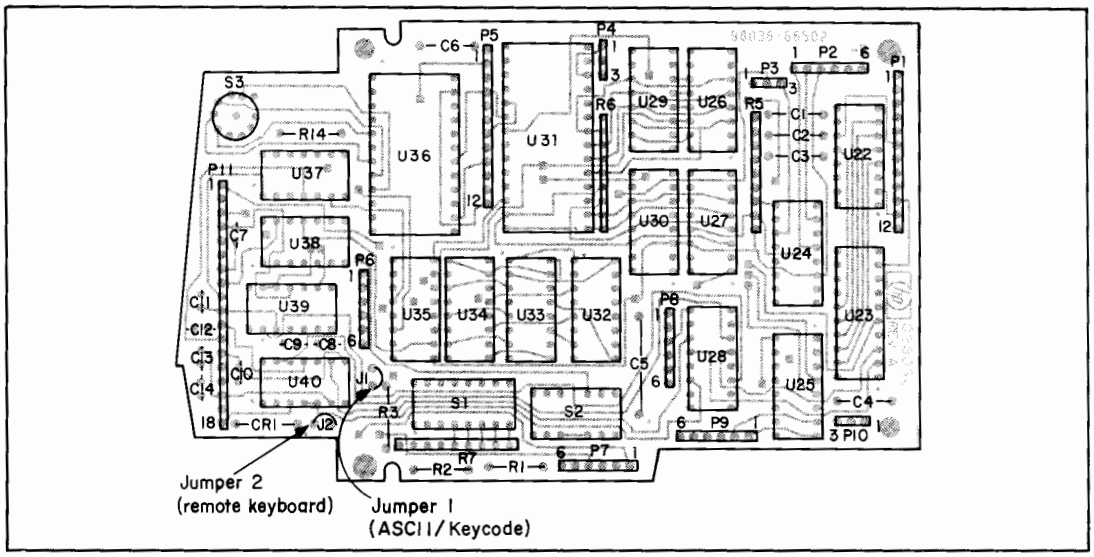
    b. Pull the rear housing off the front housing slightly, disconnect the cable connector from the PC assembly and remove the rear housing.

    c. Remove the remaining four screws in the front housing and separate the front housing cases.

    d. Carefully separate the printed circuit assemblies.

2. Locate the 98036-66502 printed circuit board and orient it as shown in the figure labeled "Component Side".
3. Locate and identify the two wire jumpers on the board corresponding to J1 and J2 in the figure.
4. For power-up remote keyboard operation, cut jumper 2 (J2) and slightly spread the wire pieces so no electrical contact is made.
5. If the remote keyboard is to be an ASCII coded keyboard, cut jumper 1 (J1). If the remote keyboard is a 9825 type keyboard, leave J1 connected.
6. To reassemble the interface, reverse procedures 1d through 1a, being careful that the pins on the A2 assembly are properly seated in the connectors of the A1 assembly.

These jumpers affect only power-up remote keyboard operation. Programmable remote keyboard using "rkbd" is independent of the jumper configuration.



98036-66502 Diagram

Pressing the reset key of the 9825 takes the calculator out of the remote keyboard mode, regardless of the state of the 98036A jumpers J1 and J2. Turning power off then on will put the 9825 back into remote keyboard mode (as set by jumpers J1 and J2).

Systems Programming Instructions

The System Programming Instructions extend the 9825's capability to generate or modify programs under program control. The "store", "%", and "nal" statements enable the 9825 to handle string text (regardless of its source) and store the text at designated program lines. The string text can be obtained from any source, such as mass memory, external systems, or another 9825. The "avm" function returns the amount of available memory remaining in user read/write memory, and "cln" returns the current program line number.

9825A/S External ROM Requirements

Mnemonic	ROM Option	Description
nal	None	Next available line.
%	None	Free-text prefix.
store	String ROM*	Store string statement.
avm	None	Available memory function.
cln	None	Current program line number.

* String ROM is not required for literals.

Store Statement

The "store" statement provides the capability of storing program lines from an executing program.

store String Name or "Literal" [: Line Number]

The string name parameter may be either a string or a literal. The line number parameter may be either a fixed value or an expression.

String Name: Names a string containing any valid HPL program line, specified as a string variable or a literal. If a string is specified, the String Programming ROM must be present in the system. If the syntax of the line to be stored is invalid, an error message is issued and program execution halted. It is possible, however, to recover from this type error and disable syntax checking by concatenating the free text prefix to the beginning of the line. A further discussion of this concept and an example are included under the "%" free text syntax.

Line Number: If included in the `store` statement, the line number must specify a line number less than the last program line number plus one. If the specified line number is greater than this value, the default (nal) value will be substituted. (Refer to the priority list below.)

To determine which program line the "store" text will actually be stored at, consider the following priorities:

(Highest Priority)

3. Line number* (parameter of "store" statement);

Example: `store "dsp A",5`

2. Line number* (prefix of text;)

Example: `store "5: dsp A"`

1. `nal` (default value if no others are specified;)

Example: `store "dsp A"`

(Lowest Priority)

There are four cases to consider in determining the actual program line number where the text is stored:

1. If the Line Number syntax parameter is not given, and no line number prefixes the program line text — the text will be stored at the default value (next available line).

* If a line number is specified, but is a number greater than the value of the last program line number plus one, the default value (`nal`) will be substituted.

2. If the Line Number syntax parameter is not given, but there is a line number prefix to the text — the text line number is compared to the value of the last program line number plus one ("nal"). If the line number is greater than the "nal" value, the line number prefix is stripped from the text and the text is stored at the next available program line. If the text line number prefix is within the program line limits, the text is stored at the specified program line.

3. If both a prefix Line Number and the line number parameter are given – the text is stored at the program line specified by the line number parameter, conditional on the parameter designating a line number less than or equal to the "nal" value. Otherwise, the text is stored at the next available line and the prefix line number is stripped from the text.

4. If there is no prefix Line Number, and the line number parameter is given – the line number parameter is checked against the "nal" value. If the line number is within the range of the program, it then specifies the program line at which to store the text. Otherwise the text is stored at the next available line ("nal").

The store instruction must be the last statement of an HPL program line, and can be executed from either an idle keyboard or a running program. It may not be executed from the live keyboard, or error C7 is issued. (This includes a "store" within a subroutine executed from live keyboard.)

There are some programming considerations to take into account when using the "store" instruction, as this instruction can significantly alter the execution flow of a running program.

- When a "store" is executed and the line is stored at a lower line number than any subroutines or interrupt routines, they will be disabled, as will any "for...next" links.

Example:

```
 0: dsp 8
 1: for I=1 to 5
 2: store "dsp
  char(66)";0
 3: next I
```

```
(error A2 in 3)
```

- Interrupts are disabled for a period of several hundred milliseconds when a "store" instruction is executing; "store" should not be used during high speed data transmissions.

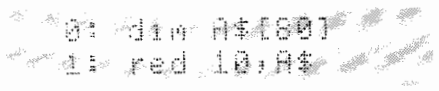
- When storing an executable expression or a string literal, the "store" instruction will actually store the interpreter representation of the expression or literal, and the resulting line will have "dsp" appended to the beginning of the text.

The store statement is a powerful programming tool, and should be used with discretion. The principal use for the store statement is in conjuction with the "nal" function given as a line number parameter. If the store statement is to be used to modify a running program, the potential consequences as mentioned above should be carefully considered.

> **CAUTION**
>
> USING THE STORE INSTRUCTION TO MODIFY THE PROGRAM AT A LINE NUMBER LOWER THAN THE CURRENTLY EXECUTING PROGRAM LINE CAN CAUSE UNPREDICTABLE PROGRAM EXECUTION.

An expanded example of the "store" capabilities is listed and explained in the appendix. The example provides the 9825 with externally stored program loading capability. A shorter example of the "store" statement used to input a program listing from an external source is included here to demonstrate the basic operations necessary.

Example:



Line 1 reads one line of text into A$.

Line 2 stores the text at the next available line.

Line 3 returns to read a new line of input text.

Next Available Line Function

```
nal
```

The `nal` function returns the value of the last program line number plus one. For example, if the resident HPL program has lines numbered 0 through 54, `nal` will return the value 55. When specified as a "store" statement parameter, the "nal" value overrides the line number prefix (if present) of the string to be stored, and the result is to store the line after the last program line.

Examples:

Ex. 1

```
0: ent "Append
Mod Routine?"r
A!if not A!gto
2
1: trk 1!ldf 12,
nal;2
2: "Main Prog"!
```

Ex. 2

Before Execution

```
0: store "0:
dsp A",nal
1: 5+A!gto 0
*25626
```

After Execution

```
0: store "0:
dsp A",nal
1: 5+A!gto 0
2: dsp A
3: dsp A
4: dsp A
5: dsp A
6: dsp A
7: dsp A
8: dsp A
```

Example 1 loads the specified file into program memory beginning with the next available program line number, allowing program editing (line insertion and deletion) without requiring modification of the load statement.

Example 2 demonstrates the use of `nal` to override the line number prefix of the literal, and the result is to store the literal at successive lines after the last program line.

Free Text Syntax Prefix

`%` String or text to be stored

Any text following a "%" symbol prefix is stored into program memory with no syntax checking performed. Note however, that the percent symbol prefix eliminates all blanks in the line except those occurring within quotation marks, and that a semicolon in the line masks all following statements in the line from the free text prefix protection. The semicolon specifies the end of the program line when it is encountered in a free text syntax. Execution continues at the next program line, not at the next statement. Use of the free text syntax prefix does not permit storing text with unmatched quotes.

Example 1:

```
2: % Thislineisi
 nvalidlysyntaxe
 d,butnasbeensto
 redwiththe"%"pr
 efix.
```

```
1: % "Note that
 all blanks are
 removed outside
  of quotes."
*16486
```

Line 0, the literal is stored but the blanks are removed because the interpreter causes blanks to be removed from the string.

Line 1, the blanks in the text are preserved by surrounding it with quote marks.

Example 2:

```
0: dim A$[80]
1: ent "Next
 Line?",A$
2: on err "inser
 t %"
3: store A$,nal
4: gto 1
5: "insert %":
6: "%"&A$->A$
7: gto 2
*26218
```

1 Enter the input line to A$.

2 Enable the error recovery routine "insert %".

3 Try to store the string.

4 Return to enter another line if no errors.

5 If an error occurred, append the free text prefix to the front of the string and return to line 2 to store the text. (This will not work if the statement contained an error after a semicolon. The example on page 7-32 resolve this problem by replacing all semicolons with % signs.)

Example 3:

The free syntax prefix enables the programmer to write end-of-line comments for a program.

```
0: % "   EXAMPLE OF COMMENTED HPL"
1: dim A$[85]
2: ent "NEXT LINE...",A$;% "          Input one line of text"
3: on err "insert %";% "              Enable the error recovery routine"
4: % "                                Store the line if possible"
5: store A$,nal
6: gto 2;% "                          Input another line"
7: "insert %":
8: "%"&A$→A$;gto 3;% "                Append the percent sign to invalid lines"
*39
```

Available Memory Function

avm

The "avm" function returns the number of unused bytes remaining in the 9825's read/write memory. This feature enables a program to allocate storage based on remaining memory. For example, a listing routine can use as much memory as possible in creating a list buffer, or an edit program can allocate as large an edit string as is currently available in the machine.

> **Note**
>
> Since the 9825 system memory requirements change during program execution, the amount of available memory is constantly changing and a several hundred byte safety factor should be allowed for (to prevent an insufficient-memory error) and subtracted from the avm value. (Useable memory = avm – safety factor.)

Current Line Number Function

```
cln
```

No parameters are required for the cln function.

The "cln" function returns the value of the current line number at the point of execution. Note, however, that the value returned by "cln" will be different when executed from within a program than when executed from live keyboard. When "cln" is executed within a program, it returns the line number of the current program line. When executed from live keyboard, "cln" will return the line number of the next program line to be executed. This is because "cln" is incremented after the end of the program line and before the live keyboard statement is executed.

The "cln" function makes possible an absolute computed gosub or go to, and a relative store. Examples of these functions follow:

Example 1, computed gosub to absolute line number:

```
6: ent "Subrouti
 ne Line Number?
 ",A
7: gsb 0;jmp A-
 cln
```

Line 6 "computes" the line number of the desired subroutine.

Line 7 executes the computed go sub to the line number in A.

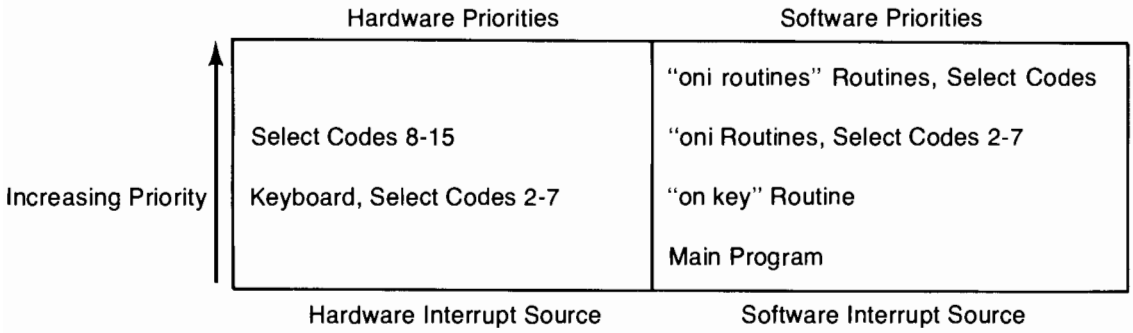
Example 2, "store" relative.

```
3: store A$,cln+
 4
```

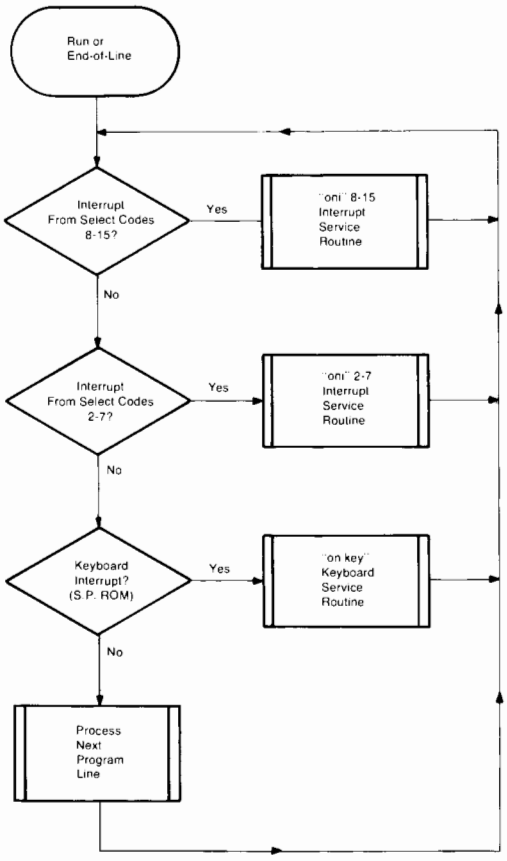
Line 3 stores the string A$ at the program line four lines down.

Example 1 enables a program to branch to an absolute line number that has previously been computed and placed into a variable. It is not necessary to perform a subroutine branch, as it is possible to simply jump to the computed location. Example 2 allows editing of the program (inserting or deleting program lines) before the "store" program line without having to modify the line number parameter of the "store" statement.

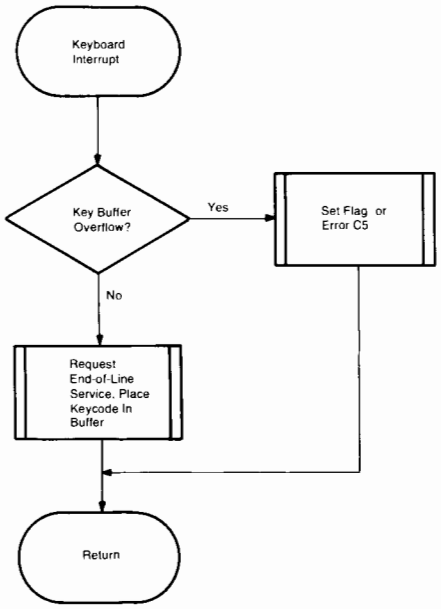
Execution Priority Diagram



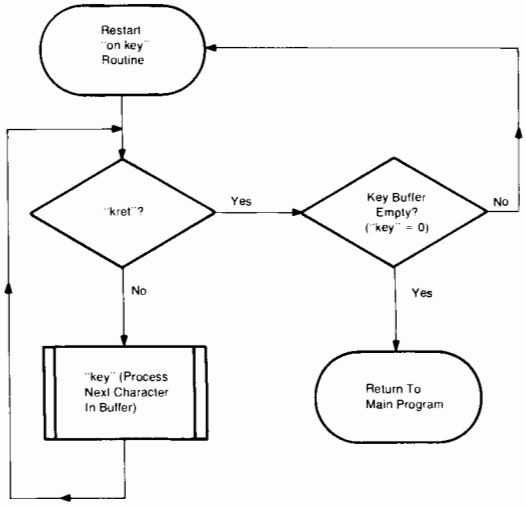
Program Execution Flowchart



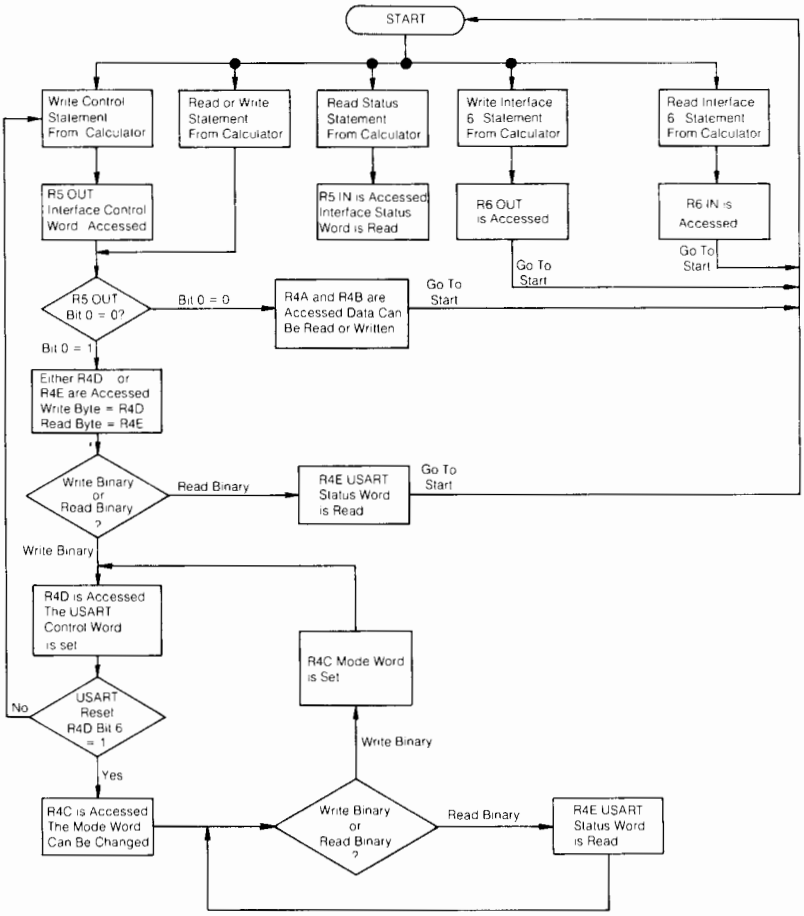
"on key" Execution Flowchart



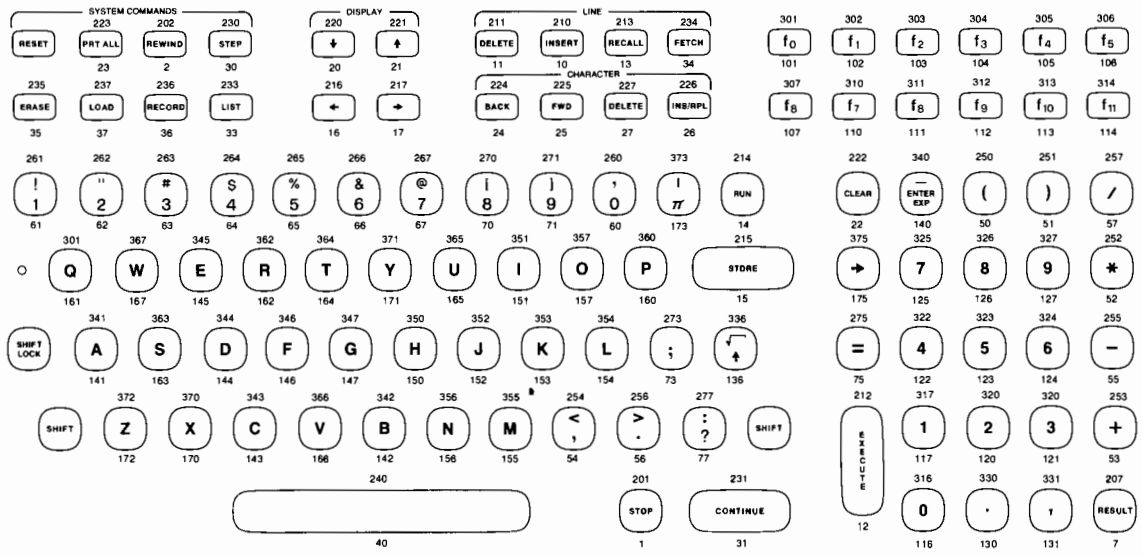
"on key" Service Routine and 'kret" Flowchart



98036A Register Access Flowchart



Octal Keycode Chart



"PTAPE"

This program is offered as an example of the capabilities of the "store" instruction when augmented by the free text prefix and error recovery facilities. The program takes input from an external source which has previously recorded a program in the "list#" format. It requests a cartridge track and file number for recording the input program, or it can mark a file the size of the input program (allowing 500 bytes for expansion) at the last unused file (null file) on the cartridge. (If a negative file number is entered, "PTAPE" will mark its own file.) The file number at which the program is recorded is printed out for future reference.

```
 0: "Program Loader or PTAPE":
 1: ent "Input Select Code = ?",S;ent "Record on Track# ?",T;ent "File# ?",F
 2: dim A$[85] ;avm→A;nal→N
 3: "input":red S,A$; if len (A$)<=2;gto +0
 4: if A$[1,1]="*";gto "out"
 5: if num(A$[1])=0;gto "input"
 6: on err "err";store A$,nal
 7: if avm<250; beep; dsp "INSUFFICIENT MEMORY";stp
 8: gto "input"
 9: "out":A-avm→A;trk T;if F>=0;gto "rec"
10: for F=0 to 9999
11: fdf F;idf F,Y,C,Q;if Q;next F
12: mrk 1,A+500,Z;if Z<0;beep;prt "Not enough tape,",A,"bytes needed";stp
13: "rec":rcf F,lN;prt "PROGRAM ON FILE#",F,stp
14: "err":"%"&A$→A$
15: if not (pos(A$,";")→X);gto 6
16: "%"→A$[X,X];gto -1
*4328
```

Line 1: Input interface select code and the cartridge track and file number for storing the program. If given a negative file number, the routine (lines 10-12) will search for the last cartridge file (null file) and mark it to the correct program size allowing 500 bytes for expansion.

Line 2: Saves the available memory and next available line values into variables A and X.

Line 3: Inputs one program line, rejects lines consisting of only carriage-return/line-feeds.

Line 4: Checks for the asterisk at the end of the program listing.

Line 5: Strips null lines from the input.

Line 6: Enables the error recovery routine "err" and attempts to store the program at the next available line.

Line 7: Checks for enough remaining memory to input more source program.

Line 8: Returns to line 3 for more input.

Line 9: Computes the source program size in bytes, and checks the file number specified for a negative value. If negative, it proceeds to find and mark the null file.

Lines 10-12: Find the null file (last file) and mark it to the size of the source program plus 500 bytes for modification and expansion.

Line 13: Records the program on either the specified or the marked file, and prints the file number used.

Line 14: The error recovery routine appends a "%" (free text prefix) to the beginning of the program line.

Lines 15 and 16: Check for semicolons in the source line, substituting them with %'s, because a semicolon will mask the following line statements from free text protection. This avoids a possible loop from illegal statements after a semicolon in the source line.

This program (with slight modification) makes possible an interesting method of program editing using the HP 2640 terminal's block output capability. It is possible to list a program to the HP 2640 terminal, inspect and edit it from the terminal as desired, then place the program back on the cartridge at the specified track and file number. The necessary modification is a write byte (wtb) statement inserted at line 3, which now becomes (assuming the 2640 set up for single line transmission blocks, with select code 2):

```
3: "input": wtb2,27,100,17: red S,A$: if len (A$)<=2:
   sto +0
```

The program now reads the text from the HP 2640 terminal a line at a time, beginning with the first character after the cursor, then records it on the specified track and file of the HP 9825 data cartridge as a program.

Technical Appendix On Asynchronous Data Formats

Asnychronous I/O is a serial mode of communication that in its simplest form requires no handshaking ("I'm ready, are you ready?") signals. This is made possible through special codes that are added to each character being sent. These extra codes are the "Start Bit" and the "Stop Bits". An additional bit, the "Parity Bit" may be added for purposes of error detection.

For example, the ASCII character "T" looks like this in binary:

(most significant bit) 1010100 (least significant bit).

When the start, parity and stop bits are added, the character "U" looks like this:

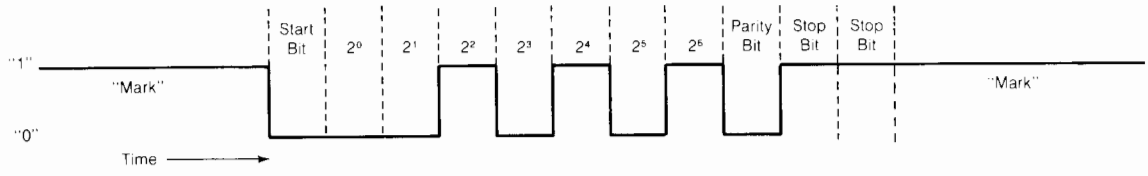
(msb) 11010101000 (lsb).

The number of bits per character is not changed by adding the start, parity, and stop bits, as these bits are not considered when looking at character bits.

The start bit is always a "0", and comes before the least significant bit of the character. The parity bit is set to a "1" or "0" to make the sum of the "1" bits of the character plus the parity bit either odd or even, depending on whether odd or even parity is selected. (The "1" character bits are added to the parity bit, yielding an odd or even sum.) The character "T" above has odd parity. The two leftmost bits of the above character are stop bits, and the stop bits are always a "1".

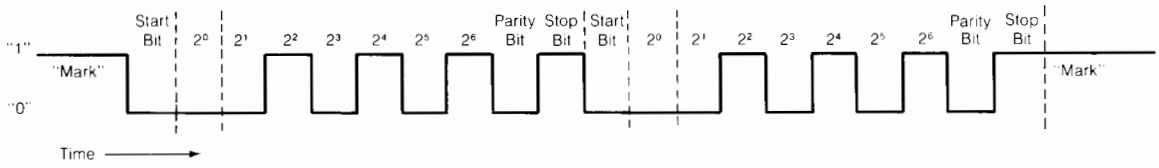
Each bit is transmitted at a specific time, controlled by an extremely accurate crystal timer. The rate at which bits are transmitted is referred to as the bit rate, sometimes known as the baud rate. The bits can be sent and received at the clock frequency, 1/16 the clock frequency, or 1/64 the clock frequency. The 1/64 rate provides the highest degree of accuracy in timing, and is used whenever error-free communications is a must.

A diagram of a single character ("T" again) being transmitted asynchronously looks like this:



The start bit is the first bit transmitted, and when received means "wake up, get ready for a new character". The next bits are the data bits of the character "T", beginning with the least significant (2^0) bit and ending with the most significant (2^6) bit. The next bit is the parity bit (odd parity), which is used for error checking. The last two bits are the stop bits, which mean "end of this character".

By using the 98036A Mode Word, we can change the format of the ASCII character, so let's send two "T" 's, but this time with only one stop bit selected. The diagram looks like this:

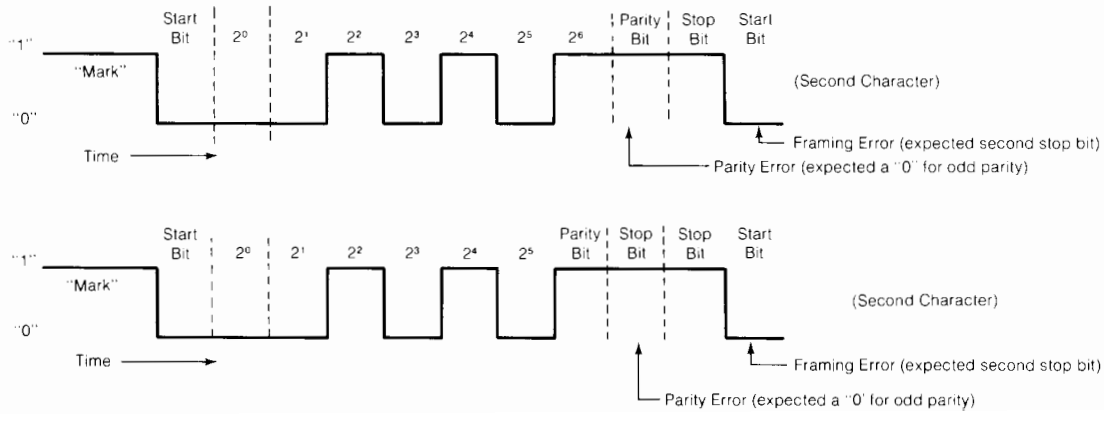


The 98036A Status Word can give us some clues about the incoming data on the serial I/O link. These are the "framing", "parity", and the "overrun" status bits of the 98036A Status Word. The "framing" bit will be set to a "1" if our interface doesn't find all the stop bits that it expects. There is no way for the interface to detect too many stop bits, but if too few are received then the framing error bit is set. (The interface looks for a "1" in the stop bit time slot.) Some causes of framing errors are incorrect number of data bits, no parity, or too few stop bits.

The "parity" bit of the 98036A Status Word will be set if the incoming parity bit is wrong. This can be caused by an incorrect number of data bits, having the wrong parity selected, or no parity bit being received.

A "overrun" error simply means that the incoming data is coming in faster than it is being taken from the interface. If the baud rate being operated at is too high, it may not be possible to read the data from the interface before a new character is received. A lower baud rate or buffered I/O can alleviate this problem. (The baud rates for the sender and the receiver must always be the same.)

Two examples showing how the same error can be generated in two completely different ways are shown below. Assume the interface is configured to expect seven data bits, odd parity, and two stop bits.



The first example is simple: the sender is sending the wrong parity and only one stop bit. Changing the interface parity and stop bits will clear the problem. The second example is also simple, but wouldn't be corrected by changing the parity and stop bit format. The fewer data bits sent (6) will always generate a framing error, and only sometimes generate a parity error. This is a difficult problem to track down from the receiver end.

Hopefully, this discussion has served to introduce the reader to the purpose of the changeable asynchronous data format, and to the necessity of accessing the status bits of the 98036A Serial Interface. In a typical system, both sender and receiver data formats are known and accessible, making interfacing a simple task.

A program which could be used to establish the correct number of stop bits and the correct parity setting is included as an example of how the 98036A control statements can be used. The program makes two assumptions for the purpose of simplification: first, an ASCII format is assumed – that is, seven data bits per character; second, it assumes that a parity bit is part of the character and not disabled (if no parity bit is present, this program cannot get a correct frame count).

Mode Word Finder Program

```
0: dim A$[80];
 0→F;2↑0+2↑1+
 2↑2+2↑3+2↑6+
 2↑7→M;2↑0+2↑1+
 2↑2+2↑5→C
1: ent "Select
 Code Is?",S;
 wsm S,M,C
2: "read1":rdb(S
 )→X;rsa S→X
3: if bit(5,X);
 sto "framing"
4: if F<2;2→F;M-
 2↑2+2↑4→M;wsm
 S,M,C;sto "read
 1"
5: if not bit(3,
 X);prt "Mode
 Word Is";M;red
 S,A$;prt A$[1,
 80];stp
6: if F<3;3→F;M+
 2↑5→M;wsm S,M,
 C;sto "read1"
7: if F<4;4→F;M+
 2↑2-2↑4→M;wsm
 S,M;sto "read1"
```

Line 0 sets the mode variable (M) to 8 data bits, no parity, 2 stop bits (to avoid parity checking).

Line 1 configures the interface mode.

Line 2 inputs one byte and reads the status.

Line 3 checks for framing error.

Line 4 reconfigures the interface to 7 data bits, odd parity.

Line 5 checks for parity error, and prints the mode value if no error.

Line 6 sets even parity if a parity error was detected.

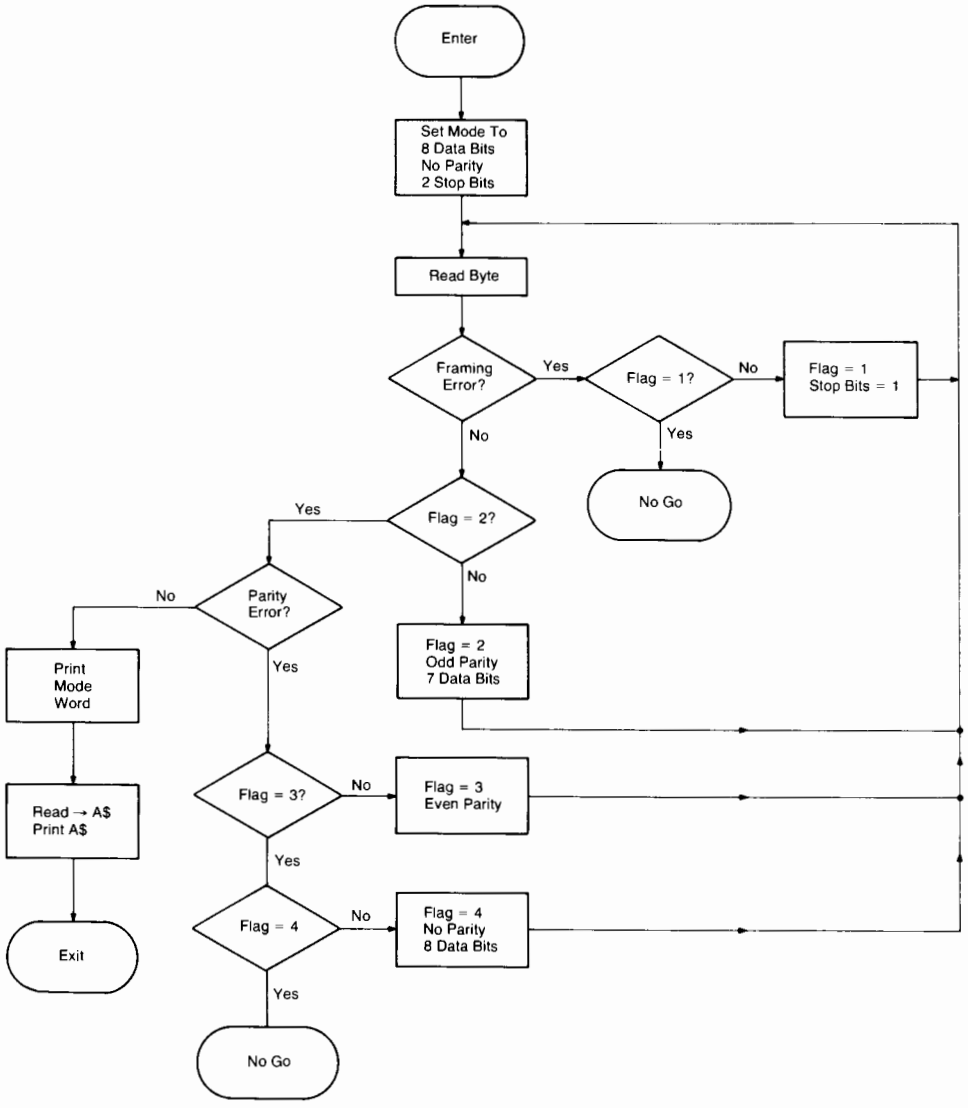
Line 7 resets to 8 data bits, no parity if a parity error still exists.

```
8: prt "No Go,
 Try New Baud
 Rate..."!stp
9: "framing"!if
 F<1;1+FIM-217+M
 !wsm S,M!sto
 "read1"
10: sto 8
*12072
```

Line 8 stops execution if all combinations have been tried.

Line 9 sets the interface to 1 stop bit on the first framing error.

Line 10 goes back to 8 if a framing error still exists with only 1 stop bit.



Flowchart For Mode Word Finder

Notes

Appendix A
Table of Contents

Notes

Appendix A

Reference Tables

Calculator Status Conditions

The following table shows the calculator status conditions when the indicated operations are performed. For details about the status condition of modes, variables, etc., see the appropriate section in the manual.

	Erase all or Power on	Reset	Erase	Run	Continue after editing	Continue
Variables	R	X	R	R	X	X
Flags 0 through 15	R	X	R	R	X	X
Result	R	X	X	X	X	X
Binary program	R	X	X	X	X	X
Subroutine return pointers	R	R	R	R	R	X
Print-all mode	R	R	X	X	X	X
Verify mode	R	R	X	X	X	X
Live keyboard mode	R	R	X	X	X	X
Secure mode	R	X	R	X	X	X
Cassette select code	R	R	X	X	X	X
Cassette track	R	R	X	X	X	X
Angular units for trig functions	R	R	X	X	X	X
Fixed/Float setting	R	R	X	X	X	X
Random number seed	R	R	X	X	X	X
Trace mode	R	R	X	X	X	X

R = Restored to power-on value
X = Unchanged

Extended I/O Status Conditions

The following table shows status conditions for various Extended I/O operations and modes. Notice that the Erase, Erase All-Power on, and Run columns from the previous table are combined into one column here. R = restored to power-on state; X = unchanged.

	Calculator Operation			
Extended I/O ROM Operation or Mode	Power On Erase Erase All Run	Reset	Continue (after edit)	Continue (after Stop)
Conversion and parity tables	R	X	X	X
Binary mode (reset to decimal)	R	X	X	X
I/O buffer area	R	X	X	X
Service name list	R	X	X	X
Equate name list	R	X	X	X
Buffer select code for tfr	R	R	R	X
Interrupt parameters	R	R	R	X
Error recovery routine	R	R	R	X
Timeout routine	R	X	X	X

ASCII Character Codes

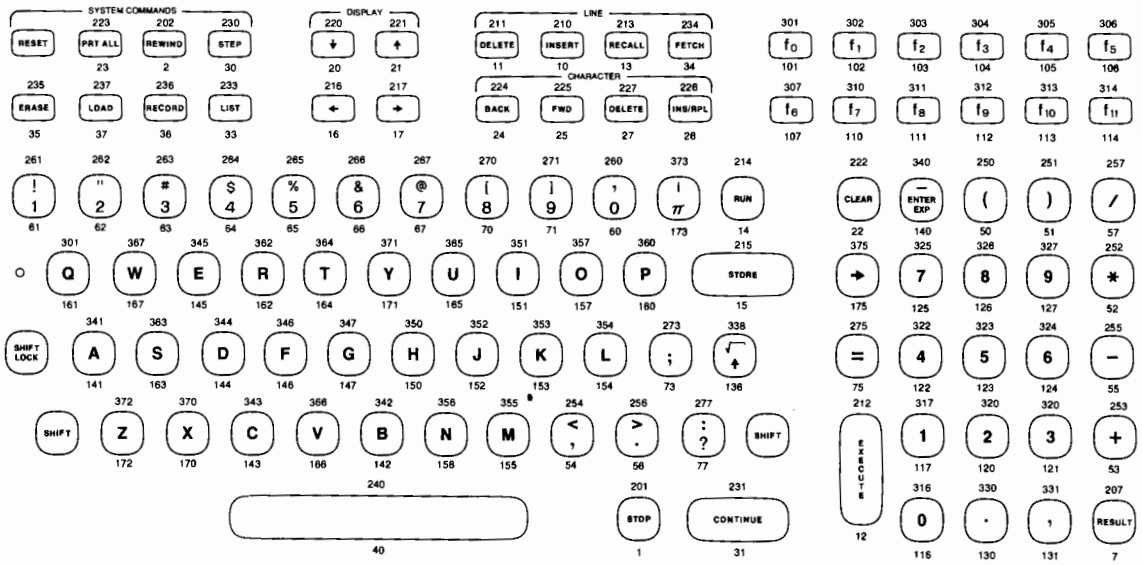
ASCII Char.	EQUIVALENT FORMS		
	Binary	Octal	Decimal
NULL	00000000	000	0
SOH	00000001	001	1
STX	00000010	002	2
ETX	00000011	003	3
EOT	00000100	004	4
ENQ	00000101	005	5
ACK	00000110	006	6
BELL	00000111	007	7
BS	00001000	010	8
HT	00001001	011	9
LF	00001010	012	10
VT	00001011	013	11
FF	00001100	014	12
CR	00001101	015	13
SO	00001110	016	14
SI	00001111	017	15
DLE	00010000	020	16
DC1	00010001	021	17
DC2	00010010	022	18
DC3	00010011	023	19
DC4	00010100	024	20
NAK	00010101	025	21
SYNC	00010110	026	22
ETB	00010111	027	23
CAN	00011000	030	24
EM	00011001	031	25
SUB	00011010	032	26
ESC	00011011	033	27
FS	00011100	034	28
GS	00011101	035	29
RS	00011110	036	30
US	00011111	037	31

ASCII Char.	EQUIVALENT FORMS		
	Binary	Octal	Decimal
space	00100000	040	32
!	00100001	041	33
"	00100010	042	34
#	00100011	043	35
$	00100100	044	36
%	00100101	045	37
&	00100110	046	38
'	00100111	047	39
(	00101000	050	40
)	00101001	051	41
*	00101010	052	42
+	00101011	053	43
,	00101100	054	44
−	00101101	055	45
.	00101110	056	46
/	00101111	057	47
0	00110000	060	48
1	00110001	061	49
2	00110010	062	50
3	00110011	063	51
4	00110100	064	52
5	00110101	065	53
6	00110110	066	54
7	00110111	067	55
8	00111000	070	56
9	00111001	071	57
:	00111010	072	58
;	00111011	073	59
<	00111100	074	60
=	00111101	075	61
>	00111110	076	62
?	00111111	077	63

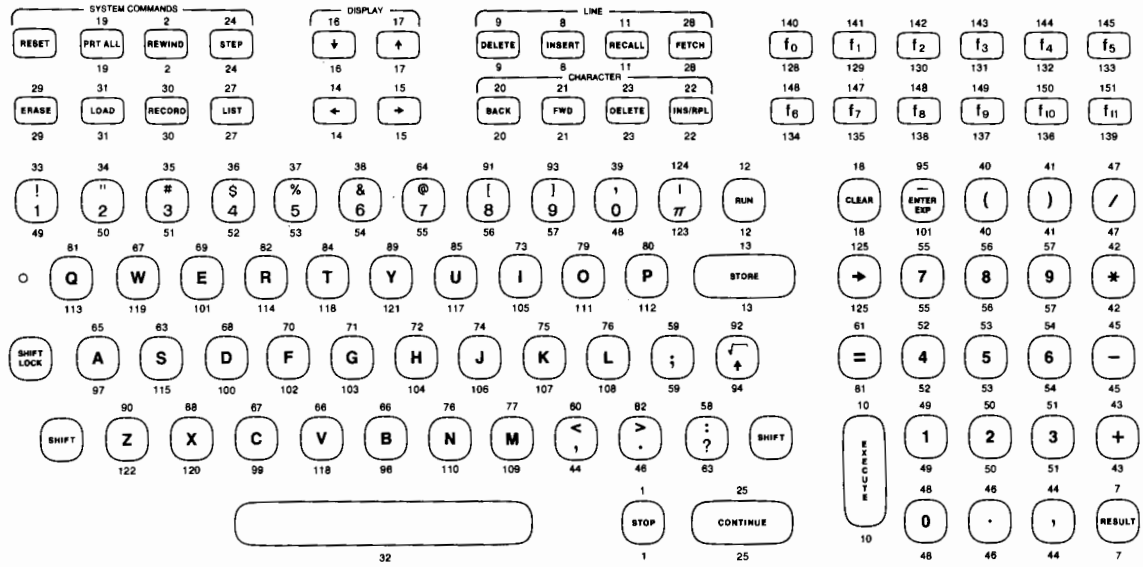
ASCII Char.	EQUIVALENT FORMS		
	Binary	Octal	Decimal
@	01000000	100	64
A	01000001	101	65
B	01000010	102	66
C	01000011	103	67
D	01000100	104	68
E	01000101	105	69
F	01000110	106	70
G	01000111	107	71
H	01001000	110	72
I	01001001	111	73
J	01001010	112	74
K	01001011	113	75
L	01001100	114	76
M	01001101	115	77
N	01001110	116	78
O	01001111	117	79
P	01010000	120	80
Q	01010001	121	81
R	01010010	122	82
S	01010011	123	83
T	01010100	124	84
U	01010101	125	85
V	01010110	126	86
W	01010111	127	87
X	01011000	130	88
Y	01011001	131	89
Z	01011010	132	90
[	01011011	133	91
\	01011100	134	92
]	01011101	135	93
^	01011110	136	94
_	01011111	137	95

ASCII Char.	EQUIVALENT FORMS		
	Binary	Octal	Decimal
`	01100000	140	96
a	01100001	141	97
b	01100010	142	98
c	01100011	143	99
d	01100100	144	100
e	01100101	145	101
f	01100110	146	102
g	01100111	147	103
h	01101000	150	104
i	01101001	151	105
j	01101010	152	106
k	01101011	153	107
l	01101100	154	108
m	01101101	155	109
n	01101110	156	110
o	01101111	157	111
p	01110000	160	112
q	01110001	161	113
r	01110010	162	114
s	01110011	163	115
t	01110100	164	116
u	01110101	165	117
v	01110110	166	118
w	01110111	167	119
x	01111000	170	120
y	01111001	171	121
z	01111010	172	122
{	01111011	173	123
|	01111100	174	124
}	01111101	175	125
~	01111110	176	126
DEL	01111111	177	127

Octal Keycode Chart*



ASC Keycode Chart*



*Unshifted code shown below key; shifted code shown above key.

Keyboard/ASCII Function Chart

Control*	9825A Command	Octal Code	Decimal Code
]	erase	35	29
_	ldf	37	31
↑	rcf	36	30
[	list	33	27
S	prt all	23	19
B	rewind	2	2
X	step	30	24
P	↓	20	16
Q	↑	21	17
N	←	16	14
O	→	17	15
I	del	11	9
H	ins	10	8
K	recall	13	11
\	fetch	34	28
T	back	24	20
U	fwd	25	21
W	del	27	23
V	ins/rep	26	22
L	RUN	14	12
M	STORE	15	13
Y	CONTINUE	31	25
A	STOP	1	1
J	EXECUTE	12	10
R	clear	22	18
G	result	7	7

Alternate ASCII Code Functions:

Carriage-Return (15 Octal): Store
Line-Feed (12 Octal): Execute

* The Control Key and the specified character key are pressed simultaneously.

ASC Conversion Values

Display Char	Key Code	ASCII Dec	ASCII Oct
@	0	0	0
A	1	1	1
B	2	2	2
C	3	3	3
D	4	4	4
E	5	5	5
F	6	6	6
G	7	7	7
H	8	8	10
I	9	9	11
J	10	10	12
K	11	11	13
L	12	12	14
M	13	13	15
N	14	14	16
O	15	15	17
P	16	16	20
Q	17	17	21
R	18	18	22
S	19	19	23
T	20	20	24
U	21	21	25
V	22	22	26
W	23	23	27
X	24	24	30
Y	25	25	31
Z	26	26	32
[	27	27	33
\	28	28	34
]	29	29	35
^	30	30	36
_	31	31	37
SP	32	32	40
!	33	33	41
"	34	34	42
#	35	35	43
$	36	36	44
%	37	37	45
&	38	38	46
'	39	39	47
(	40	40	50
)	41	41	51
*	42	42	52
+	43	43	53
,	44	44	54
-	45	45	55
.	46	46	56
/	47	47	57
0	48	48	60
1	49	49	61
2	50	50	62
3	51	51	63
4	52	52	64
5	53	53	65
6	54	54	66
7	55	55	67
8	56	56	70
9	57	57	71
:	58	58	72
;	59	59	73
<	60	60	74
=	61	61	75
>	62	62	76
?	63	63	77

Display Char	Key Code	ASCII Dec	ASCII Oct
`	64	64	100
a	65	128	200
b	66	129	201
c	67	130	202
d	68	131	203
e	69	132	204
f	70	133	205
g	71	134	206
h	72	135	207
i	73	136	210
j	74	137	211
k	75	138	212
l	76	139	213
m	77	0	0
n	78	48	60
o	79	49	61
p	80	50	62
q	81	51	63
r	82	52	64
s	83	53	65
t	84	54	66
u	85	55	67
v	86	56	70
w	87	57	71
x	88	46	56
y	89	44	54
z	90	90	132
{	91	91	133
|	92	92	134
}	93	93	135
~	94	94	136
DEL	95	95	137
↑	96	101	145
←	97	97	141
BLK	98	98	142
BLU	99	99	143
GRN	100	100	144
CYN	101	101	145
RED	102	102	146
MAG	103	103	147
YEL	104	104	150
WHT	105	105	151
↓	106	106	152
→	107	107	153
BS	108	108	154
TAB	109	109	155
LF	110	110	156
VT	111	111	157
FF	112	112	160
CR	113	113	161
SO	114	114	162
SI	115	115	163
DLE	116	116	164
DC1	117	117	165
DC2	118	118	166
DC3	119	119	167
DC4	120	120	170
NAK	121	121	171
SYN	122	122	172
ETB	123	123	173
CAN	124	124	174
EM	125	125	175
SUB	126	126	176
ESC	127	127	177

ASC Conversion Values

Display Char	Key Code	ASCII Dec	ASCII Oct
	128	0	0
	129	1	1
	130	2	2
	131	3	3
	132	4	4
	133	5	5
	134	6	6
	135	7	7
	136	8	10
	137	9	11
	138	10	12
	139	11	13
	140	12	14
	141	13	15
	142	14	16
	143	15	17
	144	16	20
	145	17	21
	146	18	22
	147	19	23
	148	20	24
	149	21	25
	150	22	26
	151	23	27
	152	24	30
	153	25	31
	154	26	32
	155	27	33
	156	28	34
	157	29	35
	158	30	36
	159	31	37
	160	32	40
	161	33	41
	162	34	42
	163	35	43
	164	36	44
	165	37	45
	166	38	46
	167	39	47
	168	40	50
	169	41	51
	170	42	52
	171	43	53
	172	60	74
	173	45	55
	174	62	76
	175	47	57
	176	39	47
	177	33	41
	178	34	42
	179	35	43
	180	36	44
	181	37	45
	182	38	46
	183	64	100
	184	91	133
	185	93	135
	186	0	0
	187	59	73
	188	0	0
	189	61	75
	190	0	0
	191	58	72

Display Char	Key Code	ASCII Dec	ASCII Oct
	192	0	0
	193	140	214
	194	141	215
	195	142	216
	196	142	217
	197	144	220
	198	145	221
	199	146	222
	200	147	223
	201	148	224
	202	149	225
	203	150	226
	204	151	227
	205	0	0
	206	48	60
	207	49	61
	208	50	62
	209	51	63
	210	52	64
	211	53	65
	212	54	66
	213	55	67
	214	56	70
	215	57	71
	216	46	56
	217	44	54
	218	0	0
	219	0	0
	220	0	0
	221	0	0
	222	92	134
	223	0	0
	224	95	137
	225	65	101
	226	66	102
	227	67	103
	228	68	104
	229	69	105
	230	70	106
	231	71	107
	232	72	110
	233	73	111
	234	74	112
	235	75	113
	236	76	114
	237	77	115
	238	78	116
	239	79	117
	240	80	120
	241	81	121
	242	82	122
	243	83	123
	244	84	124
	245	85	125
	246	86	126
	247	87	127
	248	88	130
	249	89	131
	250	90	132
	251	124	174
	252	92	134
	253	125	175
	254	94	136
	255	95	137

Keyboard/ASCII Control Codes

ASCII Char.	EQUIVALENT FORMS Binary	Octal	Dec	9825A Key Equivalent
NULL	00000000	000	0	*
SOH	00000001	001	1	STOP
STX	00000010	002	2	REWIND
ETX	00000011	003	3	*
EOT	00000100	004	4	*
ENQ	00000101	005	5	*
ACK	00000110	006	6	*
BELL	00000111	007	7	RESULT
BS	00001000	010	8	INSERT
HT	00001001	011	9	DELETE
LF	00001010	012	10	EXEC
V_{TAB}	00001011	013	11	RECALL
FF	00001100	014	12	RUN
CR	00001101	015	13	STORE
SO	00001110	016	14	←
SI	00001111	017	15	←
DLE	00010000	020	16	↓
DC_1	00010001	021	17	↑
DC_2	00010010	022	18	CLEAR
DC_3	00010011	023	19	PRT ALL
DC_4	00010100	024	20	BACK
NAK	00010101	025	21	FWD
SYNC	00010110	026	22	INS/RPL
ETB	00010111	027	23	DELETE
CAN	00011000	030	24	STEP
EM	00011001	031	25	CONT
SUB	00011010	032	26	*
ESC	00011011	033	27	LIST
FS	00011100	034	28	FETCH
GS	00011101	035	29	ERASE
RS	00011110	036	30	RECORD
US	00011111	037	31	LOAD

* = No direct 9825A key equivalent.

9825 and 9820/9821 Compatibility

In general, any program which is used with the HP 9820A/9821A Calculators can be entered into the HP 9825 Calculator with only minor changes, such as changing E (enter exponent) and statement mnemonics to lower case.

The following is a list of subtle differences between the 9820A/9821A and the 9825 Calculators. The list is divided into two sections; those differences which occur when **entering** a program and those which occur when **running** the program.

Entering Programs

- A line label must be followed by a colon. The 9820A/9821A requires a semicolon.
- Parentheses must be used to indicate which relational operator to apply first:

  `if A = B = C;` must be entered as: `if (A = B) = C;` on the 9825 (not the same as `if A = B and B = C`).
- Storing a line with an end statement does not delete higher numbered lines in memory on the 9825.
- `Γ-A` is not allowed on the 9825; use `Γ(-A)`.
- The enter (ent) statement is syntax checked by the 9825 Calculator. The items in an enter statement must be text or variables. Expressions, such as `ent AA→B` are not allowed on the 9825; the equivalent on the 9825 would be `ent A; AA→B`.
- A string of unary operators such as `---X` is not allowed on the 9825.
- File sizes in the mark statement are given in bytes instead of registers. Therefore, `MRK 1,X` in the 9821A becomes `mrk 1,8X` in the 9825.
- Linking programs is done differently on the 9825; `GTO X; LDF Y` becomes `ldf Y,X` on the 9825.
- `A+B→C+X→Y` on the 9820A/9821A must be typed as `(A+B→C)+X→Y` on the 9825.

- `E-6` on the 9820A/9821A must be written `1e-6`.
- The TBL function of the 9820A/9821A Math Block has been replaced as follows:

9820A/9821A Function	9825 Replacement
TBL0	units
TBL1	deg
TBL2	rad
TBL3	grad
TBL4	no replacement
TBL5	csv
TBL6	cfg

Running Programs

- Relational comparisons are made to 12 significant digits on the 9825. The 9820A/9821A rounds to 10 significant digits and then compares. The 9825 equivalent of: `if X = Y;` is: `if drnd(X,10) = drnd(Y,10).`
- Floating point numbers are rounded on the 9825 instead of truncated as on the 9820A/9821A when an integer value is required.

  Some implications of this are shown in these examples for the 9825:

1. r(4.9) refers to r5
2. jmp 2.9 is the same as jmp 3.
3. sfg 5.95 is the same as sfg 6 (and similarly for cfg, flg, and cmf).

- A gto or gsb to a label requires an exact match in the 9825 instead of a match on the last 4 characters as on the 9820A/9821A.
- The 9820A/9821A returned a 0 for 0 ↑ 0. On the 9825, 0 ↑ 0 results in error 73 (default is 1).
- A number, expression, or statement are valid replies on the 9825. However, if `ent A` is the enter statement and a statement such as `10→Z` is entered, flag 13 is set and A retains its previous value. For example, no value is entered by the enter statement and flag 13 is set in the following.

`prt X`	Print statement.
`Z→X`	Assignment statement.

These are valid entries:

A+7π	Expression.
(Z→K)	Imbedded assignment.

- Flag 13 is cleared when a number or expression is supplied during an enter statement.
- On the 9820A/9821A Calculators, if the run program key is pressed without entering a value for: ent R(X+1→X), the value for X would not be incremented and RX would not be modified. On the 9825, the expression, (X+1→X), is executed even if no value is entered.
- The 9825's integer (int) function is defined as the largest whole number less than or equal to the argument. The 9820A/9821A definition is the largest whole number less than or equal to the absolute value of the argument, with the sign of the result being the same as the sign of the argument.
- On the 9825, if an error occurs during the execution of a statement, the entire line is aborted. On the 9820A/9821A the rest of the statements in the line are performed.
- Implied storage to Z is replaced by implied storage to result (res). Z is no longer different from other simple variables. A statement with implied storage cannot be stored - a variable must be given explicitly. A program can access the value of result (res), but the value in res cannot be altered by the program.
- Branching to the line which is numbered one higher than the last line of the program no longer treats that line as if it were an end statement.
- Flags are not cleared by the end statement on the 9825.
- The stop (stp) statement does not destroy subroutine return information.
- On the 9825, the identify file (idf) statement always positions the tape **before** the header of the identified file. Thus, repeated idf statements do not advance the tape. Also an idf statement followed by a mrk statement marks the identified file (any information on the identified file will be lost).

Notes

Appendix B

HPL Syntax

Introduction

The following pages are a compilation of all current 9825 HPL syntax. More information on each operation can be found by referring to the indicated manual and page. The manual titles are abbreviated here:

D	Disk Programming, 09825-90220 (or 09885-90000).
I/O	I/O Control Reference.
M	Matrix Programming.
O&P	Operating & Programming Reference.

The HPL programming language utilizes four basic types of syntax constructions: **statements**, **functions**, **operators** and **commands**. Operators, such as + and mod, are used with numbers and variable names to construct **expressions** (like A+5). Expressions can be included in many statements and executed from the keyboard. Each statement can also be preceded by a line number and stored as a program line (like 10: prt A). Most functions can include expressions, and can be executed from the keyboard. Functions can also be treated as expressions when constructing a statement (like prt sinA). Commands are operator aids that can only be executed from the keyboard; they're not programmable.

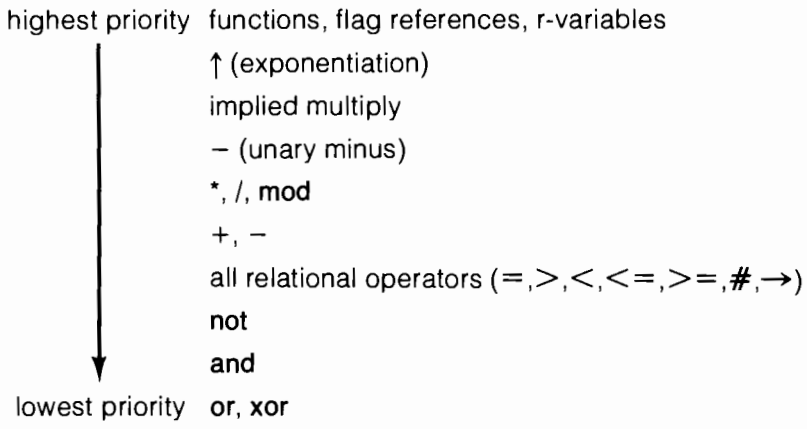
Operators

The available operators are summarized here. For more details see the HPL Programming chapter, page 3-19.

Arithmetic		Relational	
+	Add	→	Assign
−	Subtract, unary −	>	Greater than
*	Multiply	<	Less than
/	Divide	>= or =>	Greater than or equal to
↑	Exponentiate	<= or =<	Less than or equal to
mod	Modulus	# or < > or > <	Not equal to

Logical		String	
and		&	Concatenation
ior	inclusive OR		
xor	exclusive OR		
not			

Math Hierarchy



Operators of the same level in an expression are executed from left to right. Any operations within parentheses, however, are performed first. For more details, see page 3-18 in your Operating and Programming Reference.

Syntax Conventions

These terms and conventions are used in the following listing:

bold type — all key words and characters appearing in bold type must appear exactly as shown. These items are shown in dot matrix in the referenced manuals.

[ ] — elements enclosed in brackets (not key characters or parentheses) are optional.

... — an ellipsis indicates that the preceding parameter or sequence in the syntax can be repeated.

variable name — a numeric or string variable name (like A or R5 or A$). Subscripts are allowed (like A [7]).

array name — an array variable name, with or without subscripts.

string variable — a string variable name (like A$ or B$ [1,4]).

string — either a string variable or text within quotes ("text").

line number — an expression from 1 through 999 referring to a program line.

line label — a unique name assigned to a program line. It's enclosed in quotes, follows the line number, and is followed by a colon. For example: 5: "print": ...

expression — a logical combination of numeric variable names, constants, operators and functions (including user-defined functions) grouped within parentheses as needed. The evaluated expression yields a numeric result.

constant — a fixed number within the computer's range, like 2.23467.

character — a letter, number or symbol.

item list — a series of constants, expressions and/or strings separated by commas, for example: prt 5,A,"was",A+7

subscripts — numbers within brackets which are attached to variable names to designate a particular variable element or boundary. For example: A [10,5] or B$ [1,10]

file number — an expression indicating the tape or disk file.

file name — a string indicating the disk file name.

select code — an expression indicating the device's interface select code setting (an integer from 0 through 16). For example: wrt 6
These select codes are assigned to internal devices:

0 Keyboard.
1 Tape drive.
16 Printer.

device address — a two-digit number appended to the select code, indicating a device's HP-IB address. Device address range is from 01 through 31. For example: wrt 711 outputs to device 11 via the HP-IB interface set to select code 7.

format no. — a number from .1 through .9 appended to the select code to reference a corresponding fmt statement. For example: wrt 7.3 references fmt 3.

return variable — a simple numeric variable name (A or R4) where information is stored after the operation.

flag no. — an expression from 1 through 15 indicating a programmable flag.

A

abs expression
Returns the absolute value of the expression. O&P, 3-22.

acs expression
Returns the principal value of the arccosine of the expression in the current angular units. O&P, 3-25.

add (expression , expression)
Returns the sum of the expressions, added in the current numeric mode, decimal (mdec) or octal (moct). I/O, 3-15.

aprt array variable [, array variable [, ...]]
Prints the specified array's elements on the internal printer. M, 8.

ara array variable$_1$ [{ + − * / } array variable$_2$] → array variable$_3$

Performs the arithmetic operation, element by element, on arrays 1 and 2. The result is stored in array 3. (Example: ara A+B→C). Arithmetic operations can be performed on arrays in place (ara A+B→A), arrays can be copied (ara A→B) and implied multiplication is allowed (ara AB→C). M, 11.

asc expression

Returns the ASCII equivalent of the specified 9825 keycode. O&P, 7-25.

asgn file name , file number [, drive number [, return variable]]

Assigns a number (1 through 10) to an existing disk file name and indicates optional drive number and a return variable (values below). D, 3-5 (or 36).

0	File available and assigned.	5	Memory file.
1	File doesn't exist.	6	Binary program file.
2	Program file.	7	File type not defined.
3	Special function key file.	8	File number out of range.
4	File not defined by 9825.	9	Data file, but logical records not 256 bytes long (98228A ROM only).

asn expression

Returns the principal value of the arcsine of the expression in the current angular units. O&P, 3-26.

expression → variable name$_1$ [→ variable name$_2$ [→ ...]]

Assigns the value of the expression to the variable(s). O&P, 3-19.

atn expression

Returns the principal value of the arctangent of the expression in the current angular units. O&P, 3-26.

avd

Disables automatic tape verification. O&P, 5-24.

ave

Enables automatic tape verification (default setting). O&P, 5-25.

avm

Returns the size (bytes) of unused read/write memory. O&P, 4-27.

axe X coordinate , Y coordinate [, X tic [, Y tic]]

Draws axes through the X,Y point, drawing optional tic marks at X tic and Y tic intervals. 9862 Plotter ROM only. I/O, 7-18.

B

band (expression , expression)

Returns the 16-bit result of ANDing the expressions. I/O, 3-12.

beep

Sounds the computer's beeper. O&P, 3-16.

bit ($expression_1$, $expression_2$)

Returns the binary value of the bit position in expression 2 indicated by expression 1. I/O, 3-15.

boot

Loads 98217A Disk ROM bootstraps from a disk tape to an initialized disk. D, 1-8 (or 67).

bred (buffer name)

Returns the contents of the specified, active, interrupt buffer. O&P, 7-10.

buf "name" [, buffer size or string variable , buffer type]

Sets up and names a data buffer of either type read/write (no type specified) or the specified type (see below). I/O, 6-6.

Buffer Type	Word	Byte
interrupt	0	1
fast read/write	2	3
DMA	4	—

C

cap (string)

Returns an equivalent string of uppercase characters. O&P, 6-24.

cat [select code or buffer name]

Prints a catalog of files on the specified disk or default drive. File types listed below. D, 1-16 (or 20).

B	Binary program file.	M	Memory file.
D	Data file.	O	Other file (not created via 9825).
K	Special function keys file.	P	Program file.

cfg [flag no. [, ...]]

Clears either all 15 program flags or only the specified flags. O&P, 3-29.

chain file name [, 1st line number [, 2nd line number]]

Loads a program from the specified disk file. Same optional line numbers as get. D, 2-7 (or 25).

char (expression)

Returns the ASCII equivalent character. O&P, 6-20.

cli select code

Sends the abort message to all devices on the HP-IB, I/O, 2-27.

cll 'name' [(variable$_1$ [, variable$_2$ [, ...])]]

Calls the subroutine having the specified label, passing the value of any optional variables as pass-parameters. O&P, 4-10.

cln

Returns the current program line number. O&P, 7-28.

clr select code

Sends the clear message, either the all devices or to only a selected device by including the device address in the select code. I/O, 2-17.

cmd select code , "address parameters" [, "string"]

cmd "device name(s)" or select code [, "string"]

Sends the string of data characters to the specified HP-IB device. I/O, 2-31.

cmf [flag no. [, ...]]

Complements either all 15 program flags or only the specified flags. O&P, 3-29.

cmp (expression)

Returns the 16-bit binary one's complement of the expression. I/O, 3-13.

cont [line number or "line label"]

This command continues program execution, either from the current point or from the specified point. O&P, 2-24.

conv [expression$_1$, expression$_1$ [, expression$_2$, expression$_2$] ...]

Sets up a conversion table (up to 10 sets of expressions) referenced by red and wrt statements. Each expression represents an ASCII character. conv (no parameters) cancels any existing table. I/O, 1-23.

copy [source drive number [, select code] ,] **"to"**
[, destination drive number [, select code]]

Duplicates the contents of the source disk to the destination disk. Disks must be the same type, either single-sided or double-sided. D, 4-7 (or 60).

copy source file name [, drive number [, select code]]
[, destination file number [, drive number [, select code]]
Copies a file to another disk. Omitting an address accesses the default drive. D, 4-7 (or 60).

copy source file number , record number ,
destination file number , record number , no. of records
Copies only the specified number of records, beginning at the specified record numbers. D, 4-10 (or 60).

cos (expression)
Returns the cosine of the expression. O&P, 3-25.

cplt [character-space widths , character-space heights]
Moves the pen the specified distance away from the current point. I/O, 7-41.

csiz [height [, aspect ratio [, paper ratio [, angle of rotation]]]]
Specifies the size, shape and lettering direction for lbl statements. Defaults are: height = 1.5% of paper height; aspect ratio = 1; paper ratio = 1; angle = 0 (left to right lettering). I/O, 7-38.

csv
Clears simple variables A through Z. O&P, 3-39.

ctbl [string variable]
Sets up a conversion table; the value of each string character represents ASCII; the character position represents the foreign code + 1. ctbl with no parameters cancels the table. I/O, 4-6.

D

deg
Sets degrees units for angular calculations. O&P, 3-25.

del line number [, 2nd line number [,*]]
This command deletes either the specified program line or all lines through the optional 2nd line number specified. Including the * changes all remaining references to the deleted lines to the next remaining program line, preventing error 36. O&P, 2-25.

dev "name" , select code
Assigns a name for use in place of the select code in I/O operations. I/O 2-9.

dig X , Y [, return variable]
Reads, computes and stores the current pen position in user units. Return variables: 0 = pen up; 1 = pen down. 9872 Plotter ROM only. I/O, 7-48.

dim variable name [, variable name [, ...]]
Reserves memory for specified variables. Use subscripts to indicate size of each variable. O&P, 3-37.

dirc

Copies the spare 9885 disk directory (default drive) to the main directory. 98217A ROM only. D, 4-16 (or 65).

drive unit no. [, select code]

Sets the default unit (0 through 3) and, optionally, the select code for disk drives. Default is 0,8 for 98217A ROM and 0,707 for 98228A ROM. D, 1-14 (or 17).

drnd (expression , expression)

Returns the value of the first expression, rounded to the number of digits indicated by the second expression. O&P, 3-22.

dsp item list

Displays the items listed. To display quotes use double quotes within the string (e.g., 1: dsp "Display""test""in quotes."). O&P, 3-12.

dto (expression)

Returns the octal equivalent of the decimal value expressed. I/O, 3-12

dtrk

Dumps a bad 9885 track during the disk error recovery routine. 98217A ROM only. D, 4-15 (or 65).

dtype

Returns a code indicating the type of drive, disk and data format at the default disk address. 98228A ROM only. D, 1-15. Return values are:

0 Unable to access default disk controller.
1 Drive door is open or drive not present.
2 Drive door closed, but door was opened since last disk operation. File pointers are cleared.
3 9895 drive, single-sided disk, HP format.
4 9895 drive, double-sided disk, HP format.
5 9895 drive, single-sided disk, unknown format.
6 9895 drive, double-sided disk, unknown format.
7 9895 drive, single-sided disk, IBM 3740 format.
8 9885 drive, single-sided disk.

dump [file name , tape file name] [, expression]

Transfers the contents of the default disk to a tape cartridge. The optional file names indicate to only dump a specified file. The expression can be 1 or 10, indicating the number of disk records to put in each tape file. A positive expression automatically marks the tape. A negative expression suppresses marking the tape. D, 4-12 (or 62).

E

eir select code [, byte]

Enables an interrupt from the specified select code. Specifying byte = 0 disables the interrupt. I/O, 5-6.

end

Halts program execution and sets the program counter to 0. O&P, 3-17.

enp ["prompt",] string variable

Enters and prints data entered from the keyboard. O&P, 3-15.

ent ["prompt",] variable name

Enters data from the keyboard. O&P, 3-13.

eol code [, [,...]] [,– delay in milliseconds]

Specifies up to seven optional ASCII characters for an end-of-line sequence for wrt operations (replaces CR/LFs). The optional delay occurs after the last eol character in the sequence. O&P, 7-12.

eor (expression , expression)

Returns the 16-bit binary result of the exclusive ORing of the expressions. I/O, 3-13.

equ "name$_1$" , "string$_1$" [, "name$_2$", "string$_2$" [, ...]]

Equates the ASCII character string with the name, for use with cmd. I/O, 2-33.

erase [letter or key]

Erases either all programs and variables or the specified areas listed below. O&P, 2-26.

- a Erase entire memory.
- k Erase all special function keys.
- v Erase all variables and flags.
- fn Erase specified key definitions.

ert file number

Erases the current tape track, beginning with the specified file. O&P, 5-15.

exp (expression)

Returns e (2.71828...) raised the expressed power. O&P, 3-24.

F

fdf file number

Positions the tape at the specified file on the current track. O&P, 5-9.

fetch [line number or key]

Displays the specified program line or special function key definition. O&P, 2-27.

files " file name$_1$ " [: unit no.] [, " file name$_2$ " [: unit no.] [, ...]

Assigns names up to 10 disk files. Substituting an * for a file name allows an asgn statement to assign a file name via a string variable. D, 3-3 (or 34).

flg (flag no.)

Returns flag status: 1 = set; 0 = clear. O&P, 3-30.

flt expression

Sets floating point notation; from 0 through 11 places allowed. O&P, 3-10.

fmt [format no. ,] [spec$_1$ [, spec$_2$ ...]]

Sets up a list of format specs for red and wrt operations. Format number can be from 0 through 9. Format specs are listed below. Omitting specs cancels specified format. Omitting format no. sets format 0. A repeat factor can precede each spec. I/O, 1-8.

b	Single-character binary output.	**x**	Blank space.
cw	String character data.	**z**	Suppresses auto CR/LF.
ew.d	Exponential format.	/	Outputs CR/LF.
fw.d	Fixed-point.	"text"	Outputs text.
fzw.d	Fixed point with leading zeros.		

w = field width.

d = number of digits to right of decimal point.

for simple variable = initial value **to** value [**by** step value]

Defines start of a for-next loop. O&P, 4-3.

frc (expression)

Returns the fractional part of the expression. O&P, 3-22.

fti (expression)

Rounds and changes the expression to integer precision. The result can be stored in a two-character field. O&P, 4-26.

fts (expression)

Changes the expression to split precision for storage in a four-character field. O&P, 4-20.

fxd expression

Sets the fixed-point format; from 0 through 11 places are allowed. O&P, 3-9.

G

get file name [, 1st line no. [, 2nd line no.]]

Loads the program from the specified disk file. The lines are stored, beginning either at line 0 or at the optional 1st line number. The optional 2nd line number indicates where program execution should begin. D, 2-4 (or 23).

getb file name

Loads the specified disk binary program file. D, 2-11 (or 64).

getk file name

Loads the special function keys disk file. D, 2-9 (or 29).

getm file name

Loads the specified disk memory file. D, 2-10 (or 57).

grad

Sets the grads units for angular calculations. O&P, 3-25.

gsb line number or line label

Branches program execution to the specified subroutine. O&P, 3-34.

gsb + or − no. of lines

Branches to the subroutine beginning the number of lines relative to the current line. O&P, 3-34.

gto line number or line label

Sends program execution to the specified line. O&P, 3-31.

gto + or − no. of lines

Sends execution to specified line relative to the current line. O&P, 3-31.

I

idf file number [, file type [, current size [, absolute-size or [, track]]]]

Returns info on the current tape file. See tlist for file types. O&P, 5-7.

idn array name [, array name [, ...]]

Creates identity (square) matrices. All elements are 0 except major diagonal elements which are 1. M, 22.

if $expression_1 = expression_2$

If the equation is true, the rest of the line is executed. If false, execution immediately branches to the next line. Any relational operator can be used (<, #, >=, etc.). When both expressions are strings, the characters are compared using ASCII values. O&P, 3-36.

ina array variable [: number or simple variable]
[, array variable [: number or simple variable] ...]

Initializes each element of the array to the specified value (number or variable). Omitting the value initializes each element to 0. M, 8.

init

Runs the 9885 disk initialization routine and loads bootstraps. 98217A ROM only. D, 4-2 (or 90).

init drive number , select code [, interleave factor]

Initializes disks in either 9885 or 9895 drive. 98228A ROM only. The interleave can be an integer from 1 thru 29. D, 4-3.

int (expression)

Returns the integer value of the expression. O&P, 3-22.

inv array variable$_1$ → array variable$_2$ [, simple variable]

Stores the inverse matrix of array 1 in array 2. If the simple variable is specified, the determinant of array 1 is returned. M, 24.

iof select code

Returns interface flag state: 0 if peripheral busy; 1 if ready. I/O, 4-12.

ior (expression , expression)

Returns the 16-bit result of the inclusive OR operation on the expression. I/O, 3-13.

ios select code

Returns interface status: 0 if in error condition; 1 if operational. I/O, 4-12.

iplt X increment , Y increment [, expression]

Moves the pen the number of X and Y units from its current position. The expression is for pen control; see plt. I/O, 7-29.

iret

Ends an interrupt service routine and returns to main program. I/O, 5-7.

itf (string variable)

Returns a full-precision number from the packed, integer-precision number (a two-character string). O&P, 7-26.

J

jmp expression

Jumps program execution the relative number of lines forward (+ expression) or back (− expression). jmp 0 returns execution to the beginning of the current line. O&P, 3-33.

K

key

Returns the earliest, unprocessed keycode in the keyboard buffer. 0 indicates no keycodes in the buffer. O&P, 7-8.

kill file name

Purges the specified disk file from the default disk. D, 1-18 (or 27).

killall

Purges all disk user files. 98217A ROM only. D, 1-18 (or 67).

killall drive number , select code

Purges all user files from the specified disk. 98228A ROM only. D, 1-18.

kret

Returns execution to the main program after the key buffer is emptied. O&P, 7-9.

L

lbl expression or "string" [, expression or "string" [, ...]]

Prints characters on the plotter. I/O, 7-36.

lcl select code

Sends the local message to all HP-IB devices or, if the select code includes a device address, sends a clear lockout/local message. I/O, 2-20.

ldb file number

Loads a binary program from the specified tape file. O&P, 5-23.

ldf [file number [, line number$_1$ [, line number$_2$]]]

Loads the specified tape file into the appropriate area of memory. The optional line numbers indicate where to start loading (line number 1) and continuing (line number 2) a program. Omitting the file number loads file 0. O&P, 5-18.

ldf [file number [, data list]]

Loads data from the specified tape file into the listed variables. O&P, 5-21.

ldk [file number]

Loads the special function key file into memory. Omitting the file number loads tape file 0. O&P, 5-22.

ldp [file number [, line number$_1$ [, line number$_2$]]

Loads a program from either file 0 (file number omitted) or the specified file. The optional line numbers indicate where to start loading (line number 1) and were to start running (line number 2). O&P, 5-18.

len (string variable)

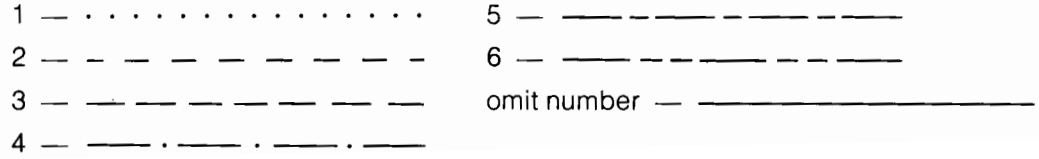
Returns the character length of the string. O&P, 6-14.

lim [X lower left , X upper right , Y lower left , Y upper right]

Restricts plotter pen movement to the stated bounds in user units. If bounds are omitted, movement is limited to the mechanical limits. 9872 Plotter ROM only. I/O, 7-34.

line [pattern number [, pattern length]]

Specifies the type of line plotted with plt, iplt, xax and yax. 9872 patterns are listed below. Pattern length is percentage of the total line length; default is 4%, 9872 Plotter ROM only. I/O, 7-32.



list [# select code] [line number [, line number]]

Lists the entire program on the internal printer (no parameters) or lists the program to the specified select code. The line numbers indicate starting and ending lines for the listing. O&P, 3-39 and I/O, 1-23.

list f1 or **listk**

Lists the special function key definition (list f1 or all definitions (list k). O&P, 3-39.

lkd

Disables live keyboard mode. O&P, 2-32.

lke

Enables live keyboard mode. O&P, 2-32.

llo select code

Sends the local lockout message to all HP-IB devices. I/O, 2-19.

ln (expression)

Returns the natural log ($\log_e$) of the expression. O&P, 3-24.

load [disk file name , tape file number]

Loads files previously dumped to a tape back onto the disk. Omitting all parameters loads the entire dump back onto the disk. Including parameters loads only selected data files back onto the disk. D, 4-13 (or 63).

log (expression)

Returns the common log ($\log_{10}$) of the expression. O&P, 3-24.

ltr X coordinate , Y coordinate [, HWD]

Moves the 9862 plotter pen to the specified point and specifies dimensions for lettering. H and W can be from 1 through 9. D is lettering direction and can be from 1 through 4. 9862 Plotter ROM only. I/O, 7-47.

ltrk

Returns corrected data to a reinitialized track during disk error-recovery routine. 98217A ROM only. D, 4-15 (or 65).

M

mat array variable$_1$* array variable$_2$ → array variable$_3$

Array multiplication (arrays must have correct dimensions). M, 19.

max (expression [, expression [, ...]])

Returns the largest value in the list. O&P, 3-22.

mdec

Sets the decimal mode (default) for binary operations. I/O, 3-11.

min (expression [, expression [, ...]] **)**

Returns the smallest value in the list. O&P, 3-22.

moct

Sets the octal mode for binary operations. I/O, 3-11.

mrk number of files , file size [, return variable]

Marks the number of files, beginning at the tape's current position. The last file number marked is returned in the optional return variable. O&P, 5-10.

N

nal

Returns the last program line number plus one; used with store to store strings. O&P, 7-24.

next simple variable

Terminates for-next loop and tests for loop completion. O&P, 4-3.

nor [line number [, line number]]

Clears the master program flag, either while executing all lines (omit all parameters) or only for the specified line numbers. O&P, 3-44.

num ("character" or substring **)**

Returns the ASCII-decimal value of the character. O&P, 6-21.

O

ofs X coordinate , Y coordinate

Offsets the origin (0,0) to point X,Y. I/O, 7-27.

on end file number , line specifier

Enables a branch to the specified line or label when a disk EOF or EOR mark is encountered during read and write operations. D, 3-19 (or 50).

on err "line label"

Enables an error-trapping routine. The program branches to the label and the erl, ern and rom functions are assigned values when an error occurs. I/O, 4-4.

on key ["line label" [, flag no.]]

Enables a keyboard interrupt routine. The program branches to the label and optionally sets the flag when the keyboard buffer overflows. Omitting all parameters disables the keyboard interrupt. O&P, 7-6.

oni select code , "label"

References an interrupt service routine associated with the peripheral's select code. I/O, 5-5.

open file name , number of records

Creates a disk data file of the specified size (256-byte records). D, 3-2 (or 33).

otd (expression)

Returns the decimal equivalent of the octal value expressed. I/O, 3-12.

P

par (expression)

Sets the parity type (listed below) used for I/O checking. I/O, 4-9.

0	Parity disabled.	2	Even parity.
1	Parity = 1.	3	Odd parity.

pclr

Sets default plotter values except scale units, select code, P1, P2, pen location and pen#. 9872 Plotter ROM only. I/O, 7-10.

pct select code

Passes active control to the specified HP-IB device. I/O, 2-26.

pen

Raises the plotter pen. I/O, 7-22.

pen# [expression]

Selects the plotter pen (1 through 4). 9872 Plotter ROM only. I/O, 7-22.

% string [:]

The % free-text prefix allows storing text without syntax checking. Free text is terminated with a semicolon or end of line. O&P, 7-25.

plt X coordinate , Y coordinate [, expression]

Move plotter pen to specified X,Y point. Optional expression controls pen (see below). I/O, 7-22.

even	lowers pen.	positive	action before plotting.
odd	raise pen.	negative	action after plotting.

pol select code

Conducts a parallel poll on the HP-IB. I/O, 2-25.

polc select code , byte

Sets parallel poll bits on the specified HP-IB device. I/O, 2-26.

polu select code

Clears parallel poll bits on the specified device. I/O, 2-26.

pos ($string_1$, $string_2$)

Returns the character position of the second string within the first. O&P, 6-16.

prnd (expression , expression)

Returns the first expression rounded to the power of ten indicated by the second expression. O&P, 3-22.

prt expression or string [, expression or string [, ...]]

Prints the list of items on the internal printer. To print quotes use double quotes (e.g., 3: prt "print""text""in quotes."). O&P, 3-12.

psc select code

Sets the select code for all plotter ROM operations. psc 0 causes either the program to ignore all plotter operations (9872 ROM) or the computer to suppress output to plotter (9862 ROM). I/O, 7-5.

ptyp

Sets a plotter lettering mode. Press STOP key to terminate mode. I/O, 7-45.

R

rad

Sets radians units for angular calculations. O&P, 3-25.

√ (expression)

Returns the square root of the expression. O&P, 3-22.

rcf [file number [, line number [, line number]] [, **"SE"** or **"DB"**]]

Records either all program lines onto the specified tape file (no line numbers) or only the specified block of lines. Including SE prevents the program from being listed or displayed when reloaded. Including DB records all trace and stop flags with the program for debugging. O&P, 5-16.

rcf file number , variable list

Records the listed variables onto the tape file. O&P, 5-16.

rck file number

Records the special function key definitions on the tape file. O&P, 5-22.

rcm file number

Records the entire computer memory on the specified tape file. O&P, 5-22.

rdb (select code)

Returns one 16-bit binary character code from the specified device. I/O, 3-4.

rdi (register number)

Returns a status byte from the interface specified by wti 0. I/O, 4-12.

rdm array variable [, array variable [, ...]]

Redimensions the array(s) to the specified dimensions. M, 16.

rds (select code)

Returns the current status word from the specified interface. I/O, 3-5.

red select code [. format no.] , variable list

Reads and stores data from the specified device. I/O, 1-5.

rem select code

Sends the remote message to either all HP-IB devices or only one device when its address is included in the select code. I/O, 2-18.

renm old file name , new file name

Renames a disk file on the default disk. D, 1-17 (or 28).

repk

Repacks files on the default disk. D, 4-5 (or 58).

res

Returns the result of the last keyboard operation not stored in a variable. O&P, 2-20.

resave file name [, 1st line number [, last line number]]

Stores a program (or only the specified lines) in an existing disk file. D, 2-9 (or 28).

ret

Ends a subroutine and returns program execution to the main program (line after gsb). O&P, 3-34.

rew

Rewinds the tape. O&P, 5-6.

rkbd select code [, expression]

Enables a remote keyboard to control the computer. The expression indicates the keycode interpretation: 0 = ASCII (default) or 1 = 9825 keycodes. O&P, 7-24.

rnd (expression)

Returns a pseudo-random number from 0 to (less than) 1. A negative expression is used as a new seed. O&P, 3-22.

rot ($expression_1$, $expression_2$)

Returns the result of binary rotation of the 16-bit equivalent of expression 1, rotated the number of bits indicated by expression 2. I/O, 3-13.

rprt file number , record number [, data list] [,"**end**" or "**ens**"]

Prints the list of data items on the disk file, starting at the specified record. Including "end" prints an EOF mark after the data. Including "ens" suppresses the automatic EOR mark printed after data. D, 3-12 (or 43).

rqs select code , byte

Requests service from the HP-IB system controller and sends the serial status byte upon response to a serial poll. I/O, 2-21.

rread file number , record number [, variable list]

Reads data from the disk file, starting at the specified record. Omitting the variable list just repositions the file pointer. D, 3-15 (or 46).

rss (select code)

Returns the 98036 Interface status register byte. O&P, 7-16.

run [line number or "label"]

Begins program execution, either at line 0 or at the specified line. O&P, 2-9.

S

save file name [, 1st line number [, last line number]]

Creates a program file and stores either the entire program (no line numbers) or only the specified lines. D, 2-2 (or 18).

savek file name

Creates a key file and stores all special function key definitions. D, 2-9 (or 29).

savem file name

Creates a memory file and stores the computer's read/write memory. D, 2-10 (or 57).

scl Xp1 , Xp2 , Yp1 , Yp2

Locates the origin and specifies user units for plotting operations. I/O, 7-7.

sfg [flag no. [, flag no. [, ...]]]

Sets either all 15 program flags to 1 or only the specified flags. O&P, 3-28.

sgn (expression)

Returns sign of expression: 0 = zero; 1 = positive; −1 = negative. O&P, 3-22.

shf (expression$_1$, expression$_2$)

Returns the result of right-shifting the 16-bit binary equivalent of expression 1, the number of places indicated by expression 2. A negative expression 2 shifts the byte to the left. I/O, 3-14.

sin (expression)

Returns the sine of the expression. O&P, 3-2.

smpy number or simple variable [*] array variable$_1$ → array variable$_2$

Multiplies each element of array 1 by the scalar number. The * can be omitted. M, 13.

spc [expression]

Outputs the expressed number of line feeds on the internal printer. O&P, 3-16.

sprt file number , data list [, **"end"** or **"ens"**]

Prints the list of data items on the disk file. Including "end" prints an EOF mark after the data. Including "ens" suppresses the automatic EOR mark printed after data. D, 3-7 (or 38).

sread file number , variable list

Reads data from the specified file into the listed variables. D, 3-10 (or 41).

stf (string variable)

Unpacks and returns a split-precision number from its four-character string. O&P, 4-20.

store string name or "string" [, line number]

Stores program lines from an executing program. O&P, 7-21.

stp [line number$_1$ [, line number$_2$]]

Stops program execution either immediately or, optionally, at the specified line (line 1). Specifying both line numbers indicates a block of lines to stop at. O&P, 3-17.

str (expression)

Returns the ASCII character equivalent to the expression. O&P, 6-19.

T

tan (expression)

Returns the tangent of the expression. O&P, 3-25.

tfr source name , destination name [, expression [, last character]]

Transfers data between an I/O buffer and a peripheral device. Optional expression indicates the total number of bytes to transfer. Optional last character expression is the decimal value of the character to terminate the transfer. I/O, 6-8.

time (expression)

Causes an I/O operation to wait for a device to become ready for the specified number of milliseconds. I/O, 4-4.

tinit

Reinitializes a bad 9885 track during disk error recovery. 98217A ROM only. D, 4-15 (or 65).

tlist

Catalogs tape files on the internal printer (file types below). O&P, 5-9.

0	Null file.	4	Memory file.
1	Binary program.	5	Special function key file.
2	Numeric data file.	6	Program file.
3	String or string/data.		

tn ↑ (expression)

Returns 10 raised to the specified power. O&P, 3-24.

trc [1st line number [, last line number]]

Sets the master flag and, optionally, trace flags for specified program lines. O&P, 3-44.

trg select code

Sends the trigger message to the specified HP-IB device. I/O, 2-17.

trk expression

Specifies the tape track (0 or 1) for successive operations. O&P, 5-6.

trn array name → array name

Transposes rows and columns between arrays. M, 23.

type ([−] expression)

Returns a value indicating the next data-item type in a disk file. A positive expression causes any encountered EORs to be skipped (like with sread). A negative expression causes any EORs to be identified (like with rread). D, 3-20 (or 51). Return values are:

0	Unidentified type.	Indicates string overlapping record boundaries:	
1	Full-precision number.	2.1	Start of string.
2	String (within record).	2.2	Middle of string.
3	EOF mark.	2.3	End of string.
4	EOR mark.		

U

units

Returns the currently-set angular units. O&P, 3-25.

V

val (string)

Returns the numeric value of the string. O&P, 6-17.

vfy [return variable]

Verifies the contents of a tape file with the original in memory. Return variable: 0 = no error: 1 = error. O&P, 5-25.

vfyb

Checks 98217A bootstraps on disk with those on the disk system cartridge. 98217A ROM only. D, 4-14 (or 67).

voff

Disables data-verification with disk print and copy. D, 4-6 (or 58).

von

Enables the disk data verification (default). D, 4-6 (or 59).

W

wait expression

The program waits for the specified time in milliseconds (from 1 to 32767). O&P, 3-16.

wrt select code [. format no.] [, item list]

Outputs the items to the specified device. I/O, 1-3.

wsc select code , expression

Outputs a control word (expression) to the specified interface. O&P, 7-14.

wsm select code , expression [, expression]

Outputs a mode word and, optionally a control word (second expression) to the specified 98036 Interface. O&P, 7-15.

wtb select code , expression [, expression [, ...]]

Outputs the byte representing each number or character to the specified device. I/O, 3-3.

wtc select code , expression

Outputs a control byte to the specified interface. I/O, 3-9.

wti 0, select code

Specifies an interface for successive wti or rdi operations. I/O, 4-11.

wti $expression_1$, $expression_2$

Outputs a control byte (expression 2) to a specified interface register (expression 1). I/O, 4-11.

X

xax Yoffset [, tic interval [, start [, end [, no. of tics/label]]]]

Draws an X axis with optional tic marks and labels. 9872 Plotter ROM only. I/O, 7-11.

xref

Prints a cross reference of program variables and line numbers, using the current program in memory. O&P, 4-32.

Y

yax Xoffset [, tic interval [, start [, end [, no. of tics/label]]]]

Draws a Y axis with optional tic marks and labels. 9872 Plotter ROM only. I/O, 7-11.

Appendix C

Subject Index

This index references subjects in these 9825 manuals:

Title	Part No.	Abbreviation
9825 Operating & Programming Reference	09825-90200	O&P
9825 I/O Control Reference	09825-90210	I/O
9825 Disk Programming Manual	09825-90220	D
Matrix Programming Manual	09825-90022	M

The index does not list subjects in the Interfacing Concepts guide or manuals supplied with computer peripherals or interfaces. Page references for the old 9825/9885 Disk Programming Manual, 09885-90000, are listed in parentheses.

Subject Index

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

S

t

u

v

w

x

y

z

Notes

Appendix D
Table of Contents

Notes

Appendix D

Error Codes

An error in a program sets the program line counter to line 0. Press the continue key to continue the program from line 0. Execute the continue command with a line number to continue at any desired line (such as: cont 50).

00	System error.
01	Unexpected peripheral interrupt.
02*	Unterminated text.
03*	Mnemonic is unknown. Mnemonic not found because disk may be down.
04	System is secured.
05	Operation not allowed; line cannot be stored or executed with line number.
06*	Syntax error in number.
07*	Syntax error in input line.
08	Internal representation of the line is too long (gives cursor sometimes).
09	gto, gsb, or end statement not allowed in present context. Attempt to execute a next statement either from keyboard while for/next loop using same variable is executed in program or from program while for/next loop using same variable is executed from keyboard. Attempt to call function or subroutine from keyboard.
10*	gto or gsb statement requires an integer.
11	Integer out of range or integer required; must be from −32768 thru +32767.
12*	Line cannot be stored; can only be executed.

* Press the RECALL key to position the cursor at the location of the error.

13 ent statement not allowed in present context.

14 Program structure destroyed.

15 Printer out of paper or printer failure.

16 String Variables ROM not present for the string comparison. Argument in relational comparison not allowed.

17 Parameter out of range.

18 Incorrect parameter.

19 Bad line number.

20 Missing ROM or binary program. The second number indicates the missing ROM. In the program mode, the line number is given instead of the ROM number. Displayed number and missing item:

1 Binary Program	10 Matrix ROM
4 Systems Programming ROM	11 Plotter ROM
6 Strings ROM	12 General I/O ROM
8 Extended I/O ROM	17 Disk ROM
9 Advanced Programming ROM	

21 Line is too long to store.

22 Improper dimension specification.

23 Simple variable already allocated.

24 Array already dimensioned.

25 Dimensions of array disagree with number of subscripts.

26 Subscript of array element out of bounds.
P-number reference is negative.

27 Undefined array.

28 ret statement has no matching gsb statement.

29 Cannot execute line because a ROM or binary program is missing.

30 Special function key not defined.

31 Non-existent program line.

32 Improper data type.
Non-numeric value in for statement or in fts or fti function.

33	Data types do not match in an assignment statement.
34	Display overflow due to pressing a special function key.
35	Improper flag reference (no such flag).
36	Attempt to delete destination of a gto or gsb statement.
37	Display buffer overflow caused by dsp statement.
38	Insufficient memory for subroutine return pointer. Memory overflow during function or subroutine call.
39	Insufficient memory for variable allocation or binary program. Dimensioned string cannot exceed 32,766 elements.
40	Insufficient memory for operation. Memory overflow while using for statement or while allocating local p-numbers.
41	No cartridge in tape transport.
42	Tape cartridge is write protected. (Slide record tab to right for recording.)
43	Unexpected Beginning-Of-Tape (BOT) or End-Of-Tape (EOT) marker encountered. Tape transport failure.
44	Verify has failed.
45	Attempted execution of idf statement without parameters or mrk statement when tape position is unknown.
46	Read error in file body.
47	Read error in file head.
48	End-Of-Tape (EOT) encountered before all files were marked.
49	File too small.
50	Idf statement for a program file must be last statement in the line. get or chain statement should be the last statement in a line.
51 or 52	Memory configuration error for attempted ldm statement. For example, a ROM present when memory was recorded is now not present (see error 20), or attempting to load a memory file recorded on a 9825 into a 9825B.

Memory files are not compatible between the 9825A and 9825B. Only the program portion can be recovered by loading the memory file into the original machine and doing a rcf. This program file can then be loaded into any 9825 with the ldf statement.

53 Negative parameter in cartridge statement.

54 Binary program to be loaded is larger than present binary program and variables have been allocated.

55 Illegal or missing parameter in a cartridge statement.

56 Data list is contiguous in memory for a cartridge statement.

57 Improper file type.

58 Invalid parameter in rcf statement; "SE" or "DB" expected.

59 Attempt to record a program or special function keys which do not exist.

60 Attempt to load an empty file or the null file (type = 0).

61 The line referenced in an ldf or ldp statement does not exist. If the line containing the ldf or ldp statement has been overlaid by the load operation, the line number in the display may be incorrect.

62 Specified memory space is smaller than cartridge file size.

63 Cartridge load operation would overlay subroutine return address in program; load not executed.

Disk load operation would overlay gsb return address; load not executed.

64 Attempt to execute ldk, ldf (program file), or ldp during live keyboard statement.

get, chain or getk not allowed from live keyboard mode or during an ent statement.

65 File not found.
File specified in the previous fdf statement does not exist.

Default values associated with errors 66 thru 77 when flag 14 is set are explained in the programming chapter of the operating and programming manual.

66 Division by zero.
A mod B, with B equal to zero.

67	Square root of negative number.
68	Tan (n * π/2 radians). Tan (n * 90 degrees). Tan (n * 100 grads). where n is an odd integer.
69	ln or log of a negative number.
70	ln or log of zero.
71	asn or acs of number less than −1 or greater than +1.
72	Negative base to non-integer power.
73	Zero to the zero power (0↑0).
74	Storage range overflow.
75	Storage range underflow.
76	Calculation range overflow.
77	Calculation range underflow.

A0	Relational operator in for statement not allowed. No closing apostrophe.
A1	A for statement has no matching next statement.
A2	A next statement encountered without a previous for statement.
A3	Non-numeric parameter passed as a p-number.
A4	No return parameter for a function call.
A5	No functions or subroutines running. Improper p-number.
A6	Attempt to allocate local p-numbers from the keyboard.
A7	Wrong number of parameters in fts, stf, fti, or itf function. stf or itf parameter must be a string (not a numeric). stf or itf parameter contains too few characters.
A8	Overflow or underflow in fts function. Overflow in fti function.
A9	String Variables ROM missing for stf or itf functions.

Errors B0 thru B8 may result during the binary disk initialization and disk error recovery routines.

B0	Wrong syntax, argument out of range or variable not properly dimensioned.
B1	More than six defective tracks on the disk.
B2	Verify error. Boots on the disk not identical to boots on the cartridge.
B3	dtrk or tinit not allowed because error information lost or error not d5, d6, d7 or d9.
B4	Attempt to access record for error correction which isn't part of data file.
B5	Improper string length (inconsistent with length given in header).
B6	Not enough space in computer buffer for data item. Item can't be placed in this part of buffer.
B7	Missing Disk or String ROM.
B8	Track still bad after tinit.

C0	Missing General I/O or Extended I/O ROM.
C1	Incorrect number of parameters.
C2	Improper parameter specified.
C3	Wrong parameter type.
C4	Illegal buffer type for bred statement.
C5	Key buffer overflow.
C6	Too large or wrong sign of parameter.
C7	Improper execution of store statement.
C8	Illegal use of kret.
C9	Missing 98036A Interface card.

D0 Improper argument.

D1 Argument out of range.

D2 Improper file size; must be an integer from 1 thru 32767. No lines to store for save or savek.

D3 Invalid file name.

D4 File not found.

D5 Duplicate file name or attempt to copy non-data file to existing file.

D6 Wrong file type.

D7 Directory overflow.

D8 Insufficient storage space on disk.

D9 Verify error. Disk controller detected no read errors, but the data read back doesn't compare with the original. Reprint data. If the problem persists, service the drive, interface or the computer.

DISK IS DOWN (98217A ROM)

UNABLE TO ACCESS DISC CONTROLLER (98228A ROM)

Computer cannot access the disk controller. If control is not restored (e.g., power on) press RESET or STOP to cancel operation.

d0 Firmware/driver out of synchronization.
Too many defective tracks within it (press RESET).

d1 All drives in system not powered on.

d2 Door opened while disk being accessed or during dump, load or copy.

d3 Disk not in drive or no such drive number.
Door open on 9895 drive.

d4 Write not allowed to protected disk.

d5 Record header error (use error recovery routine.)

d6 Track not found (use error recovery routine.)

d7 Data checkword error. (use error recovery routine.)

d8 Hardware failure (Press RESET).

d9 Verify error. Data is readable under normal margins but not under reduced margins. Reprint data. If problem persists, back up disk (new media) or service drive.

E0	General I/O ROM missing. HP-IB error under interrupt.
E1	Wrong number of parameters.
E2	Improper buffer device or equate table usage. Multiple-listeners error. Buffer busy.
E3	Wrong parameter type.
E4	Timeout error.
E5	Buffer underflow or overflow.
E6	Parameter value out of range.
E7	Parity failure.
E8	Improper use of iret statement. Attempt to DMA with HP-IB. Buffer or select code is busy.
E9	Illegal HP-IB operation.

F0	File overflow when read or print executed.
F1	Bootstraps not found (98217A ROM) or wrong memory configuration for 98228A Disk ROM (9825T required).
F2	String read but wrong data type encountered.
F3	Attempt to read data item but type doesn't match.
F4	Availability table overflow (repack).
F5	Attempt on end branch from other than running program.
F6	Unassigned data file pointer.
F7	Disk is down; line cannot be reconstructed.
F8	Disk is down and (STOP) pressed.
F9	System error (save files individually and reinitialize).

G1	Incorrect format numbers.
G2	Referenced format statement has an error.
G3	Incorrect I/O parameters.
G4	Incorrect select code.
G5	Incorrect read parameter.
G6	Improper conv statement parameters.
G7	Unacceptable input data.
G8	Peripheral device down.
G9	Interface hardware problem.

M1*	Syntax error.
M2	Improper dimensions. Array dimensions incompatible with each other or incompatible with the stated operation.
M3	Improper redimension specification. New number of dimensions must equal original number; new size cannot exceed original size.
M4*	Operation not allowed. An array which appears to the left of ' cannot also appear on the right.
M5	Matrix cannot be inverted. Computed determinant = 0.

* Press the RECALL key to position the cursor at the location of the error.

9862A Plotter ROM Error Codes

P1 Wrong state.
Statements executed out of order.

P2 Wrong number of parameters.

P3 Wrong type of parameters. Parameters for an lbl statement must be expressions, text, or string variables.

P4 Scale out of range. Maximum value is less than or equal to the minimum value.

P5 Integer out of range. Pen control parameter is out of the range —32768 thru +32767 or the select code is not 0 or in the range 2 thru 15.

P6 Character size out of range. Width or height in letter statement is zero or there is an integer overflow in csize calculations or results.

P7 Not used.

P8 Axes origin off-scale. X, Y specified for axis statement doesn't fall on plotter surface.

PLT DOWN Check interface connection and select code setting; be sure LINE and CHART HOLD are on.

9872A Plotter ROM (HP-GL) Error Codes

P1 Attempt to store into constant. Occurs when one or more parameters in a dig statement are constants rather than variables.

P2 Wrong number of parameters. Occurs on instructions with numeric-only parameter lists (scl, ofs, plt, iptl, cplt, xax, yax, lim, dig, csiz, line, pen#, and psc). In certain unusual cases where a parameter list contains user-level function calls, an instruction having an incorrect number of parameters may be executed.

P3	Wrong type of parameter or illegal parameter value.
P4	No HP-IB device number specified. Occurs when psc parameter is from 0 thru 14 and an HP-IB card is at the corresponding select code.
P5	Pen control value not from −32768 thru 32767. Hardware transmission error occurs between plotter and computer.
P6	No HP-IB card at specified select code.
P7	axe or ltr statement encountered; 9872 ROM cannot execute them.
P8	Computer (STOP) key cancelled operation. Occurs when the plotter fails to respond for three seconds after the (STOP) key has been pressed.
p0	Transmission error. The calculator has received an illegal ASCII input from the plotter.
p1	Instruction not recognized. The plotter has received an illegal character sequence.
p2	Wrong number of parameters. Too many or too few parameters have been sent with an instruction.
p3	Bad parameter. The parameters sent to the plotter with an instruction are out of range for that instruction.
p4	Illegal character. The character specified as a parameter is not in the allowable set for that instruction.
p5	Unknown character set. A character set out of the range 0 thru 4 has been designated as either the standard or alternate character set.
p6	Position overflow. An attempt to draw a character or perform a cplot that is located outside of the plotters numeric limit of −32768 thru +32767.

Errors generated by write (wrt) and read (red) statements will typically be displayed in the next executed plotter ROM statement. This can be avoided by using an output error command (wrt select code, "OE";) followed by a read statement (red select code, variable) to check for errors after read or write statements that address the plotter.

S0	Invalid set of strings in data list of ldf statement.
S1	Improper argument for string function or string variable.
S2	More parameters than expected for string function or string variable.
S3	Accessing or assigning to non-contiguous string, num function of null string.
S4	Trying to find the value of non-numeric string or null string. Exponent too large. Exponent format invalid (e.g., 1e+ +).
S5	Invalid destination type for string assignment.
S6	Parameter is zero, or negative, exceeded dimensioned size. Invalid sequence of parameters for string variable.
S7	String not yet allocated.
S8	String previously allocated.
S9	Maximum string length exceeded; additional string length must be specified in dim statement.
SPARE DIR.	Printed when the spare disk directory (backup track) automatically replaces the main directory.