Radio Shack
A Division of Tandy Corporation
ONE TANDY CENTER, FORT WORTH, TEXAS 76102

August 10, 1979

Dear Model II User:

Our TRS-80 Model II Microcomputer is an exciting and powerful small computer. We want to make sure that these manuals supply you with the tools you need to realize the great potential of this new system.

Before releasing the final, complete version of the manuals, we would appreciate hearing from you about your opinions of these manuals ... errors, suggestions -- and even some praise (if you think we deserve it).

Please fill out the attached questionnaire, fold it, and drop it in the mail (no postage necessary). We welcome your help in making our manuals the very best possible.

Thank you.

Sincerely,

Dave Gunzel

Dave Gunzel
Manager
Technical Publications

P.S. When the final, printed manuals are ready, we'll send you a copy ... better because you have helped share in its preparation.

One other thing -- please limit your comments and questions to the manuals. Questions or problems about the computer system should be directed to Customer Service. Call their toll free number, (800) 433-1679.

QUESTIONNAIRE
MODEL II TRS-80

Please rate the manuals in the following areas:

A. Are they UNDERSTANDABLE?
- ☐ Easy to understand
- ☐ Too elementary for my needs
- ☐ Understandable without much difficulty
- ☐ Difficult

Comments: _____

B. Are they well ORGANIZED?
- ☐ Well organized – good reference aids
- ☐ Not organized well – difficult to find information needed

Comments: _____

C. Are they COMPLETE?
- ☐ Complete enough for my use
- ☐ Not enough information
- ☐ Comments: _____

Where would you like us to add more details or illustrations: _____

Other Comments: _____

(space for more comments on back)

Please answer these questions regarding your computer and business background:

A. Are you a:
- ☐ Systems engineer, programmer, data processing manager, etc.
- ☐ Accountant
- ☐ Management consultant
- ☐ Business executive, manager, administrator
- ☐ Computer operator
- ☐ Other_____

B. How are you using the Model II computer?
- ☐ For my company's business applications
- ☐ For my client's business applications
- ☐ For my own personal use
- ☐ To develop programs which will be marketed to:
  - ☐ a. businesses
  - ☐ b. the public

C. Is your computer background . . .
- ☐ Heavy
- ☐ Moderate
- ☐ Light
- ☐ No programming knowledge before reading this manual

Comments: _____

staple or tape

Please fill out questionnaire
on back

FOLD

FIRST
CLASS
PERMIT
#138

no postage necessary

BUSINESS REPLY MAIL

POSTAGE WILL BE PAID BY ADDRESSEE:

D. Gunzel
TRS-80 Model II Manuals
Technical Publications, Dept. 0025
RADIO SHACK
1100 One Tandy Center
Fort Worth, Texas 76102

FOLD

Additional Comments:

Please put me on your mailing list to receive all information about Radio Shack's TRS-80 Microcomputer System – the Microcomputer Newsletter, advance product information, applications hints & tips, user's suggestions, etc.

PRODUCT PURCHASED

32K ☐ DISK EXPANSION UNIT ☐

64K ☐

I have read the Warranty on Hardware and Software ☐

Name _____

Address _____

City _____ State _____ Zip _____

Occupation _____

Intended usage _____

PLACE POSTAGE HERE

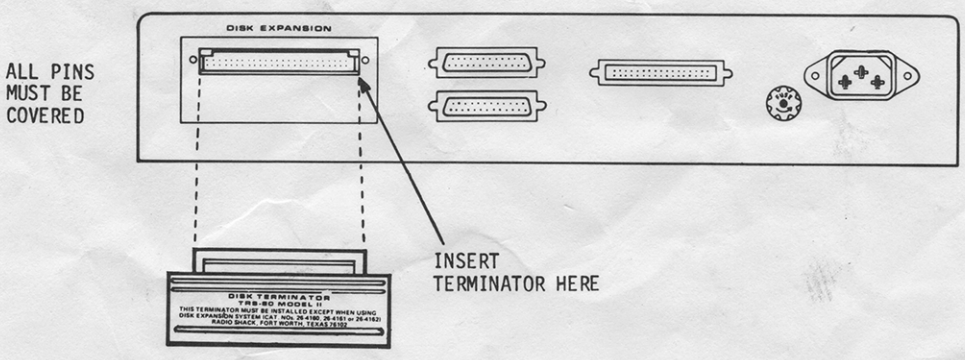
RADIO SHACK
4925 PYLON RD.
FT. WORTH, TX 76106

ATTN DEPT. 0055

IMPORTANT NOTICE

If you are not using a Disk Expansion Unit, the disk expansion terminator must be inserted into the disk expansion connector. The terminator is in the packet with the power cord.

It is very important to carefully center the terminator so that it covers all pins before inserting it. If the terminator is not inserted correctly, the disk will not function properly.



8/23/79

IMPORTANT CORRECTIONS TO MODEL II OWNER'S MANUAL
PRELIMINARY VERSION

I. OPERATION MANUAL
Page 3/1: ALL DRIVES must be empty whenever you turn the Computer on or off.

Page 6/2: To use the Model II WITHOUT a Disk Expansion Unit connected, you must connect the Terminator Plug to the the connector labeled DISK EXPANSION on the back of the Display Console.

II. TRSDOS REFERENCE MANUAL
Page 6: Delete the reference to the Keyboard Code Map--such a map is not included in the preliminary manual.

Page 44: Note that the DEBUG Upload function is not compatible with SETCOM, RS232C or the other serial I/O supervisor calls.

Page 66A: SETCOM command. Before changing any of the parameters for a channel, the channel must be off.

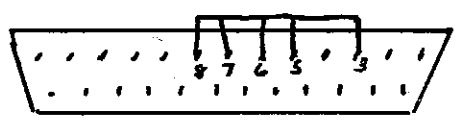
The lowest available baud rate is 110, not 100. However, to get 110 baud, you must still specify 100 in the parameter list. Also make a note of this on page 143.

Page 72: The FORMAT command syntax require braces, not parentheses, around the option list. Change all examples on pages 72 and 73 to include braces instead of parentheses. For example:

```
FORMAT :1 {ID=ACCOUNTS,PW=mouse}
```

Page 144: In the wiring diagram, Model II to Model II, you must connect pins 20 and 6 on each DB-25 male connector. We also recommend that you add a connection from pin 1 to pin 1 on the DB-25 male connectors. This establishes a common CHASSIS GROUND.

In addition, if you are only going to use one of the serial channels, pins 3, 5, 6, 7, and 8 on the other channel must be tied together BEFORE you initialize the channels with SETCOM or RS232C. Prepare a DB-25 male terminator plug for the unused channel:



REAR VIEW OF DB-25 MALE CONNECTOR

III. BASIC REFERENCE MANUAL.
Page 132: To the list of field specifiers, add:

```
^^^^        Causes a number to be printed in leading zero E or
            D format.
```

For example:

```
PRINT USING "#.#####^^^^"; 12.345@1
```

prints

```
 0.12345E+@2
```

Page 208: Change the syntax for INPUT$ to:

```
INPUT$(length, buffer-number)
```

Change the example to:

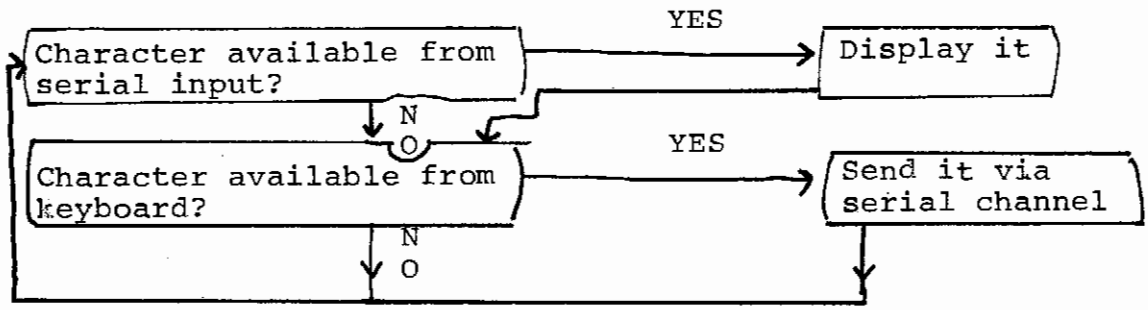
```
A$ = INPUT$(12, 2)
```

Change sample program line 2210 to:

```
2210 T$ = INPUT$(70, 1)
```

The TERM program included with TRSDOS Version 1.1

This program allows you to use serial Channel A for sending and receiving characters in a terminal mode. That is, it will function as a sending/receiving device for ASCII characters. The program alternately checks for a character received from the Channel A and checks for acharacter typed in on the Keyboard. Characters received are automatically sent to the Video Display.



Note that characters you type are not automatically copied to the Video Display. If modems at both ends of the transmission are set to half-duplex, characters you type will be echoed back from the receiving device and sent to the Video Display.

Before using TERM, you must initialize Channel A with SETCOM or RS232C. Select the parameters that suit the modem or equipment you are using. Be sure to put a terminator on Channel B if it is being used (see correction for page 144 of the TRSDOS Reference Manual).

After you have initialized Channel A, type:

```
TERM
```

as a TRSDOS command line. The Computer will go into the terminal mode.

Error Handling

If the data carrrier is not present when TERM tries to send or receive a character, it will display the message:

```
DATA CARRIER IS NOT PRESENT
```

and stop. Press <BREAK> to return to TRSDOS.

TERM doesn't check for any other receive errors (e.g., framing, lost data).

If TERM detects a transmit error (e.g., CTS not available), it will display the message:

```
XMIT ERROR
```

and then look for another Keyboard character. If the transmitter is busy when TERM tries to send a character, TERM will re-send the character.

SOURCE LISTING OF 'TERM'

```
*H 'TERM'INAL PROGRAM
        ORG     2800H
;
;       'TERM' PROGRAM - RECEIVES CHARACTERS INTO CHANNEL A AND TRANSMITS
;                        DATA KEYED IN FROM THE KEYBOARD ONTO CHANNEL A
;
RECV    EQU     $
        LD      A,96            ;SUPERVISOR FUNCTION - RECEIVE CHARACTER, CHANNEL A
        RST     8               ;IF CHARACTER AVAILABLE, IT WILL BE RETTURNED IN REG 'B'
        JR      C,DCDERR        ;DATA CARRIER LOST ERROR
        JR      NZ,KBIN         ;NO CHAR YET
;
;       WE HAVE A CHARACTER NOW FROM THE CHANNEL A RECEIVER
;
        LD      A,8             ;SUPERVISOR FUNCTION - VIDEO CHARACTER OUTPUT
        RST     8               ;OUTPUT THE CHARACTER IN REGISTER 'B'
;
;       NOW TEST IF THERE IS A CHARACTER FROM THE KEYBOARD;  OUTPUT IT
;       ONTO THE CHANNEL A TRANSMITTER IF SO
;
KBIN    EQU     $               ;NOW LOOK TO SEE IF KB INPUT WAITING
        LD      A,4             ;SUPERVISOR FUNCTION - KEYBOARD CHARACTER INPUT
        RST     8               ;IF CHARACTER AVAIL, IT WILL COME BACK IN REG 'B'
        JR      NZ,RECV         ;NO CHARACTER AVAILABLE YET, SEE IF ONE HAS BEEN RECEIVED
;
;       WE HAVE A CHARACTER FROM THE KEYBOARD, NOW TRANSMIT THIS CHARACTER
;
XMIT    LD      A,97            ;SUPERVISOR FUNCTION - TRANSMIT ONTO CHANNEL A
        RST     8               ;DO IT
        JR      C,DCDERR        ;DATA CARRIER HAS BEEN LOST - ERROR!
        JR      Z,RECV          ;CHARACTER WAS TRANSMITTED PROPERLY
        BIT     2,A             ;TRANSMITTER STILL BUSY ERROR BIT
        JR      NZ,XMIT         ;RETRY UNTIL TRANSMITTER IS NOT BUSY

XMITE   EQU     $
        LD      HL,XMITMS       ;ERROR MESSAGE W/ LENGTH BYTE IN FRONT
        LD      B,(HL)          ;GET LENGTH FROM FRONT OF MESSAGE
        INC     HL              ;GET HL => TEXT OF MESSAGE ITSELF
        LD      C,0DH           ;FOLLOW TEXT W/ A CARRIAGE RETURN
        LD      A,9             ;SUPERVISOR FUNCTION - VIDEO LINE ROUTINE
        RST     8               ;OUTPUT MESSAGE TO VIDEO
        JR      RECV            ;GO BACK TO SEE IF RECEIVED CHARACTER AVAILABLE

DCDERR  EQU     $               ;ERROR - DATA CARRIER WAS LOST MESSAGE
        LD      HL,DCDEMS       ;ERROR MESSAGE W/ LENGTH BYTE IN FRONT
        LD      B,(HL)          ;GET LENGTH OF MESSAGE INTO REGISTER 'B'
        INC     HL              ;GET HL => TEXT OF MESSAGE ITSELF
        LD      C,0DH           ;WE WANT A CARRIAGE RETURN TO FOLLOW MESSAGE
        LD      A,9             ;SUPERVISOR FUNCTION - VIDEO LINE ROUTINE
        RST     8               ;OUTPUT TO VIDEO HERE

HALT    EQU     $               ;THIS WILL CAUSE A 'HALT' OF THE PROGRAM
        JR      HALT            ;LOOP BACK - THE 'BREAK' KEY WILL RETURN CONTROL BACK TO TRSDOS

XMITMS  DEFT    'XMIT ERROR'
DCDEMS  DEFT    'DATA CARRIER IS NOT PRESENT'
```

NOTE: "TERM" is intended as a demonstration program only--to help you in writing programs which use the serial i/o capabilities of TRSDOS Model II.

BASCOM/BAS and COMSUB

The BASIC communications program BASCOM/BAS and the communications subroutine COMSUB together perform the same function as the TERM program which is included with TRSDOS. BASCOM/BAS and COMSUB are included to give you a feel for interfacing programs at the assembly-language level with BASIC programs. BASCOM/BAS calls the machine-language COMSUB via the USR function. Like TERM, BASCOM/BAS and COMSUB are included on your system disk and may be examined by employing the TRSDOS command LIST (or the BASIC command LIST in the case of BASCOM/BAS).

The programs allow you to use the keyboard of the Model II to send data in the form of ASCII characters to another computer or device; at the same time, characters transmitted on the other device will be received by the Model II and printed on the Display.

Serial Channel A is used for sending and receiving. (Channel B must be terminated; see correction for page 144 of the TRSDOS Reference Manual.) The programs alternately check Channel A for a character received, and the keyboard for a character typed. Characters typed will be automatically echoed to the Video Display, though this can be defeated by making a two-byte modification to COMSUB; the NOPs at EF9F and EFA0 should be modified to a LD (HL),00.

Before using the two programs, make sure that the RS-232 cable is connected to Channel A and that Channel B is fitted with a terminator device. Then, under TRSDOS, type DO DOCOM. DOCOM is the name of a DO file which (a) executes the SETCOM command (parameters are set to default values); (b) loads COMSUB; (c) loads BASIC with the extension -M:61000, which reserves enough memory for COMSUB. When the BASIC prompt appears on the screen, type RUN "BASCOM/BAS". The program will begin.

Error Handling

When a transmit or receive error is encountered, the word "ERROR" will be printed, followed by an 8-bit error code. If bits 0, 1, and 2 are all off, the error will be a receive error and ARVC--Channel A Receive, page 145 in the TRSDOS manual, should be consulted to pin the error down exactly. If bits 4, 5, 6, and 7 are all off, the error is a transmit error. Consult ATX--Channel A Transmit, page 146 in the TRSDOS manual.

There are two versions of COMSUB, one for 64K and one for 32K. The source listing below is the COMSUB 64K version.

```
;              SUBROUTINE FOR BASIC COMMUNICATIONS PROGRAM
;          THIS ROUTINE MUST BE EXECUTED AT 300 BAUD OR HIGHER

        ORG     0EF80H          ;ON ENTRY DE POINTS TO A 3 BYTE STRING DESCRIPTOR
        INC     DE              ;DE NOW POINTS TO LSB OF STRING ADDDRESS
        LD      A,(DE)          ;LSB OF STRING ADDRESS TO ACCUMULATOR
        LD      L,A             ;LSB OF STRING ADDRESS TO REGISTER L
        INC     DE              ;DE NOW POINTS TO MSB OF STRING ADDRESS
        LD      A,(DE)          ;MSB OF STRING ADDRESS TO ACCUMULATOR
        LD      H,A             ;MSB OF STRING ADDRESS TO REGISTER H
        LD      A,(HL)          ;1 BYTE STRING TO ACCUMULATOR
        CP      0               ;SEE IF CHARACTER IS ZERO
        JR      NZ,XMITER       ;IF NOT ZERO TRANSMIT CHARACTER, ELSE FALL THROUGH TO RECIEVE CHARACTER
        LD      A,96            ;SVC CALL#PORT A RECIEVE
        RST     8               ;
        JR      C,ERROR         ;QUIT ON ERROR IF MODEM CARRIER NOT PRESENT
        RET     NZ              ;RETURN IF NO CHARACTER RECIEVED
        OR      A               ;SET STATUS BITS
        JR      NZ,ERROR        ;QUIT ON ERROR IF ANY STATUS BITS ARE SET
        LD      (HL),B          ;PASS RECIEVED CHARACTER TO STRING LOCATION
        RET

XMITER  LD      B,A             ;CHARACTER TO BE TRANSMITTED TO REGISTER B
        LD      DE,0FFFFH       ;LOOP COUNT IF TRANSMITER BUSY STATUS ENCOUNTERED
XMIT1   LD      A,97            ;SVC CALL#PORT A TRANSMIT
        RST     8               ;
        JR      C,ERROR         ;QUIT ON ERROR IF MODEM CARRIER NOT PRESENT
        NOP                     ;INSERT "LD     (HL),00" HERE WHEN USING MODEM IN HALF-DUPLEX MODE
        NOP                     ;
        RET     Z               ;RETURN IF CHARACTER TRANSMITTED
        BIT     0,A             ;CHECK CLEAR TO SEND STATUS BIT
        JR      NZ,ERROR        ;QUIT ON ERROR IF STATUS BIT SET
        LD      A,D             ;MSB OF LOOP COUNT TO ACCUMULATOR
        OR      E               ;LSB OF LOOP COUNT
        DEC     DE              ;REDUCE LOOP COUNT
        JR      NZ,XMIT1        ;LOOP IF COUNT IS NOT ZERO, ELSE FALL THROUGH TO AN ERROR
ERROR   LD      (HL),00         ;DO NOT DISPLAY CHARACTER IF ERROR ENCOUNTERED
        LD      B,8             ;LOOP COUNT (8 BIT STATUS BYTE)
        LD      HL,BITST3       ;STORAGE AREA
BITEST  BIT     7,A             ;CHECK BIT 7 OF ACCUMULATOR FOR COMMUNICATIONS STATUS
        LD      (HL),'0'        ;LOAD ASC-II ZERO
        JR      Z,BITST1        ;JUMP IF STATUS BIT NOT SET
        INC     (HL)            ;ASC-II ZERO => ASC-II ONE IF STATUS BIT SET
BITST1  RLCA                    ;ROTATE ACCUMULATOR LEFT
        INC     HL              ;MOVE TO NEXT STORAGE POSITION
        DJNZ    BITEST          ;LOOP TO CHECK STATUS OF 8 BITS
        LD      HL,BITST2       ;ERROR MESSAGE TO BE DISPLAYED
        LD      B,14            ;LENGTH OF MESSAGE
        LD      C,'#'           ;CHARACTER TO BE INSERTED AT THE END OF ERROR MESSAGE
        LD      A,9             ;SVC CALL#VIDEO LINE
        RST     8               ;
        RET

BITST2  DEFM    'ERROR#'        ;ERROR MESSAGE
BITST3  DEFS    8               ;STORAGE AREA FOR ERROR STATUS BITS TO BE DISPLAYED
```

An Overview of the Model II Documentation Package

This binder contains the information you need to use the Model II Computer System. It is intended as a practical reference guide to the System. It is NOT a tutorial. Some familiarity with Computers will be very helpful in reading this material and using the Computer.

The binder comes with four manuals; other manuals can be added as you expand your System.

Operation Manual

Explains the connection and operation of the System, including power-up, handling diskettes, the keyboard, maintenance, etc. If you are going to use Radio Shack Applications Software, this Manual will give you all the information you need to get going. It does NOT describe Model II software (Operating System, BASIC, etc.).

Model II Operating System Reference Manual

Describes the Operating System: command format, file specification, operator commands, utilities, system routines available to assembly programmers, memory allocation, keyboard and video display features, etc.

Model II BASIC Reference Manual

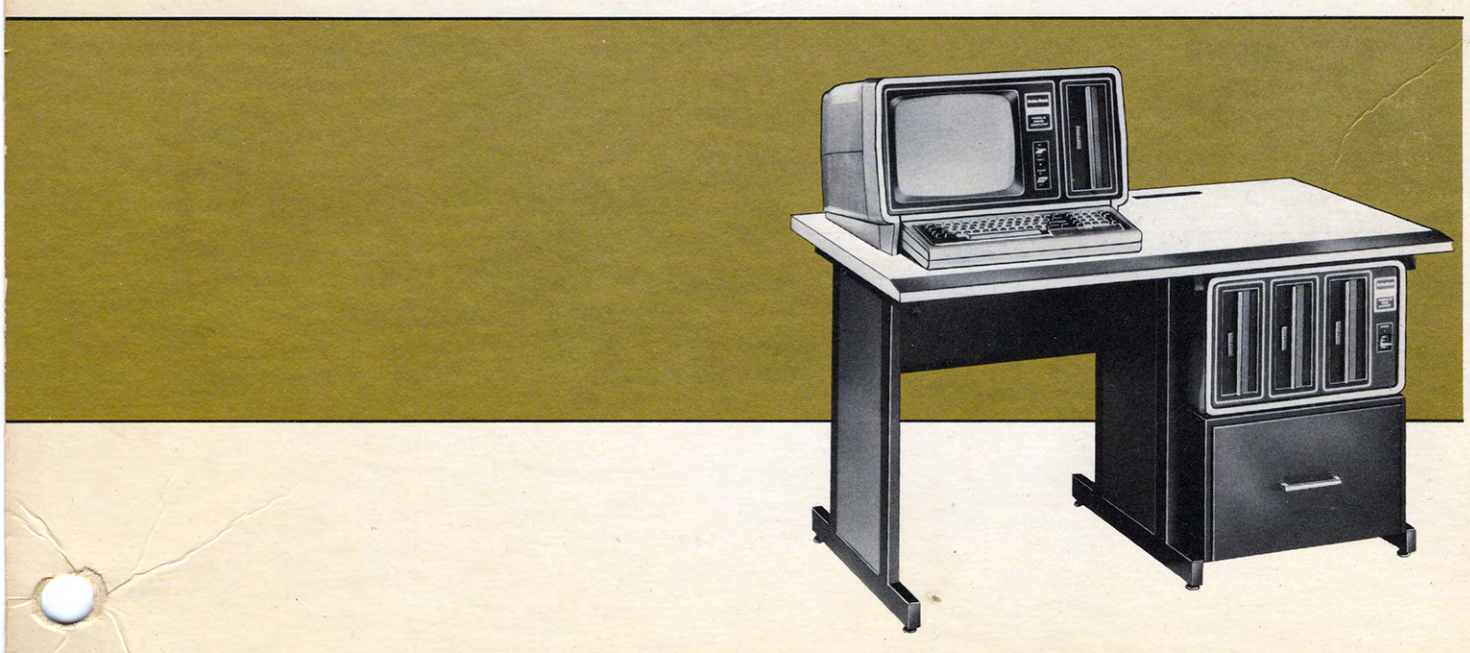
Describes the BASIC programming language used in the Model II. While the manual includes examples of statements and short applications programs, it is not a teaching-book. Radio Shack sells several books which will help you learn to program with BASIC.

OPERATION

Radio Shack®

TRS-80 Model II
Operation Manual

A Guide to Using the Computer:
Connection, Power-Up and Operation



CUSTOM MANUFACTURED IN THE U.S.A. BY RADIO SHACK TC A DIVISION OF TANDY CORPORATION

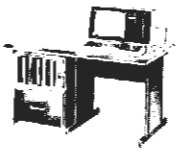
TRS-80 Model II

Operation Manual

Radio Shack®
A DIVISION OF TANDY CORPORATION
One Tandy Center
Fort Worth, Texas 76102

First Edition – 1979

Contents



1 / Brief Description of System

The Radio Shack TRS-80 Model II is a disk-based computer system consisting of two major components:

- a Display Console with built-in disk drive
- a separate Keyboard Enclosure

The Operating System software is loaded from diskette by a built-in "bootstrap" program.

Here is a brief description of the functional elements of the Computer.

Processor

At the heart of the Computer is a Z-80A microprocessor operating at its maximum design speed (4 million machine-cycles per second).

The processor receives power-up and reset instructions from read-only memory (ROM). After the TRSDOS initialization program is loaded from disk, this ROM is electronically switched out of the system and replaced with random access memory (RAM).

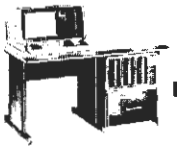
Random Access Memory (RAM)

The basic system includes 32K bytes of random access memory. (1K = 1024.) An additional 32K bytes can be added, for a total of 64K bytes of addressable RAM.

Video Display

The Video Display has its own LSI controller chip, to free the Z-80A processor from display refresh and related tasks.

The Display offers two modes: 80 characters by 24 lines, and 40 characters by 24 lines. The displayable character set includes the full ASCII set (upper and lower case alphabet, numbers, and special symbols), plus 32 graphics characters. Each character can be displayed as white on black or black on white. See **Displayable Characters** in Section 7.



Keyboard

The Model II Keyboard has its own LSI controller to free the Z-80A processor from keyboard scan and related tasks. The Keyboard is in a separate case and is connected to the Display Console via a built-in cable at the bottom front of the Console.

The Model II has the standard typewriter keys (letters, numbers and punctuation symbols); however, each of these keys can output several different codes to the Computer, depending on which mode the Keyboard is in: Unshift, Shift, Caps, or Control. In addition, the Keyboard features a Repeat key and two programmable "function" keys. (See **Keyboard Operation**.)

Floppy Disk Drive

The Model II includes a built-in 8" disk drive. Up to 3 more drives can be added in an external Expansion Unit. (See Section 6, **Add-Ons**.) Because of a special high-density recording technique, each diskette can contain 509,184 bytes of information, which is more than 5 times the capacity of a 5-1/4" diskette. (It would take a 70 wpm typist 24 hours of typing at speed to fill an 8" diskette.)

The "System Drive" (the one that's built-in) must always contain an Operating System diskette. The amount of free space on this drive available for user programs and data depends on the Operating System. (See the **Disk Operating System Manual** for actual diskette space allocation.)

The other optional drives can be devoted exclusively to the storage of user programs and data.

Peripheral Interfaces

There are four interface connections on the back of the Display Console:

- Two serial (RS-232-C) Input/Output (I/O) channels
- A parallel I/O channel, e.g., for connection to TRS-80 standard parallel-interface line printers
- Floppy-disk I/O channel for connection of the Model II Disk Expansion Unit

The Display Console also provides connectors and slots for future expansion. (See Section 6, **Add-Ons**.)

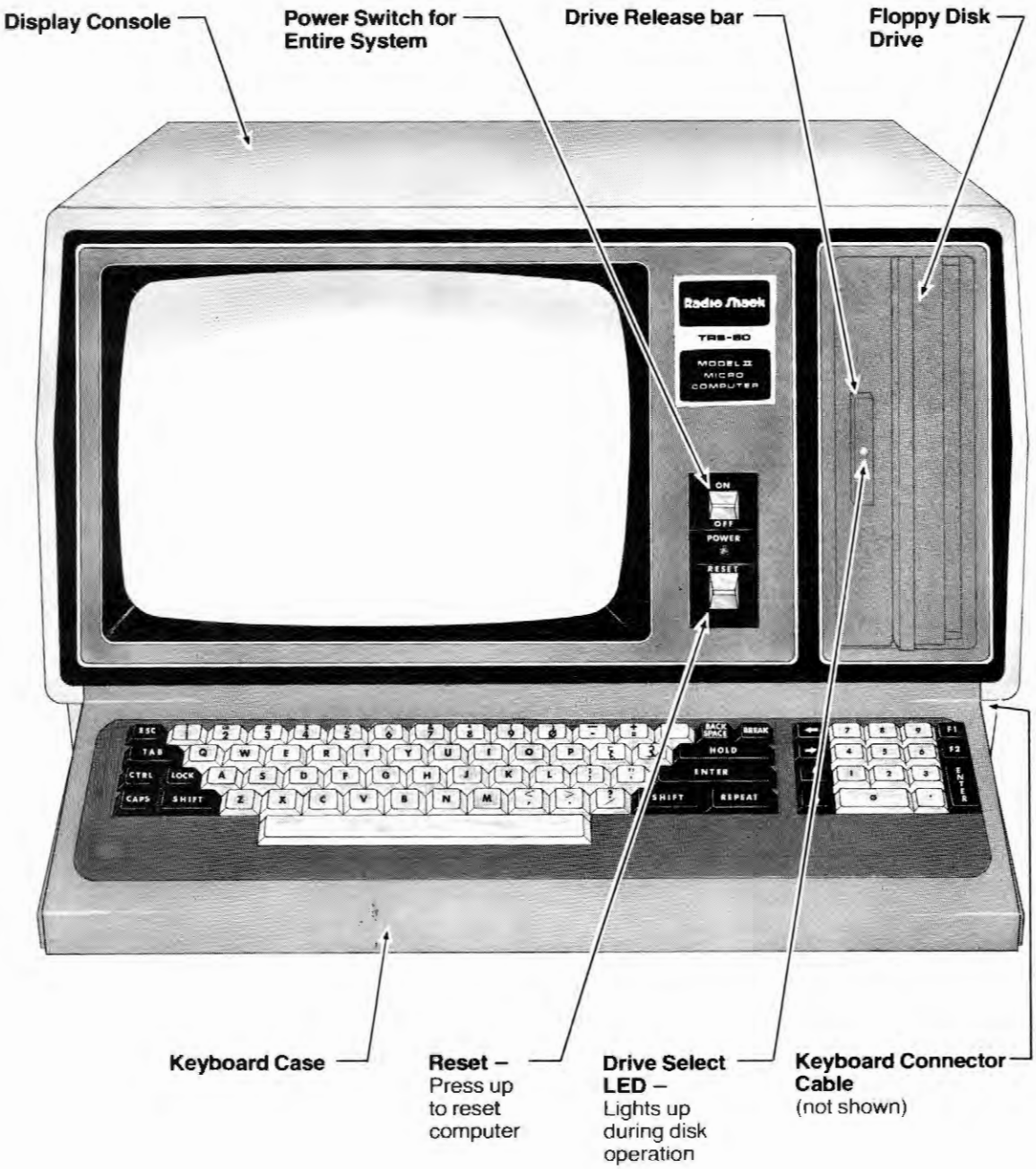


Figure 1: TRS-80 Model II

2 / Installation

Carefully unpack the System. Remove all packing material and save it in case you ever need to transport the System. Be sure you locate all cables, papers, diskettes, etc.

Place the Display Console on the surface where you'll be using the Computer. The Computer should be near a 120 VAC outlet, so that extension cables won't be necessary. (See Notes on AC Power Sources.)

Notice the cable at the bottom right of the Display Console. Plug this into the jack on the right rear of the Keyboard Case. (See **Figure 2.**)

Once connected, the Keyboard Case can be pushed back into the recessed area at the base of the Display Console, or moved to any convenient place within 2 - 2 1/2 feet of the Console.

(For connection of additional peripheral equipment, see Section 6, **Add-Ons**.)

Connect the female plug on the Power Cord to the back of the Display Console. Connect the other end to a source of 120 VAC, 60 Hz. (See **Figure 3.**)

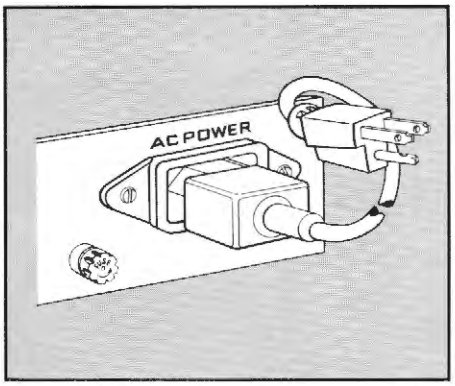


Figure 3. Power Cord connected to Display Console

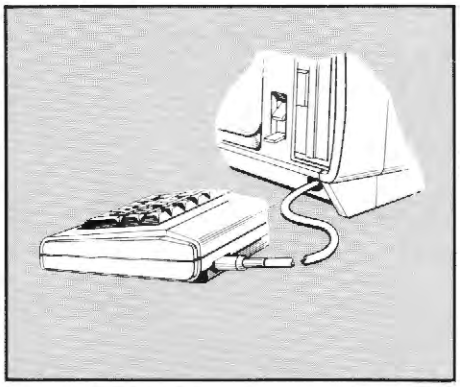
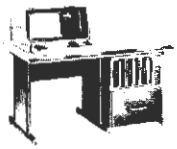


Figure 2. Display Console connected to keyboard.

Note: The power cord has a three-prong safety plug to provide a reliable ground for the system. This ground is very important to the System. If at all possible, plug it directly into a three-prong socket. Otherwise use a 3-to-2 prong adapter and **ground the adapter.**



Notes on AC Power Sources

Computers are sensitive to fluctuations in the power supply at the wall socket, from very short-duration (millionths of a second) voltage spikes, to prolonged drops in current or voltage. This is rarely a problem unless you are operating in the vicinity of heavy electrical machinery. The power supply may also be unstable if some appliance or office machine in the vicinity has a defective switch which arcs when turned on or off.

Your TRS-80 Model II contains a specially designed, built-in AC line filter. It should eliminate all but the most severe interference problems. Should you still experience power-line interference, you should take some or all of the following steps:

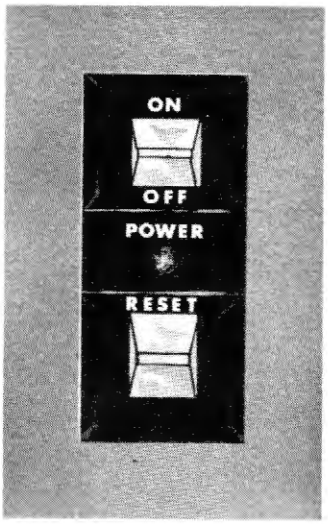
- Install bypass/isolation devices to the noisy appliance
- Fix the defective switch
- Install a separate power line
- Install a special line filter designed for use with computers and other electronic equipment

In severe conditions, all actions may be required.

Power line problems are rare and many times can be prevented by proper choice of installation location. The more complex the system and the more serious the application, the more consideration you should give to providing an ideal power-source for your Computer.

3 / Operation

Turning the Computer On



The drive should be empty (no diskette in place) when you turn on the Computer.

Push the Power Switch up to the ON position. After a few seconds for warmup, the Screen should be filled with a solid white field.

The Computer will now perform a quick check-out of the bootstrap ROM, Z-80A microprocessor, and the first 32K of RAM.

Next, the Computer will prompt you to insert the Operating System Diskette. See **Loading the Operating System.**

Using the RESET Switch

If you should ever lose Keyboard control of the System, or you simply want to re-initialize, press RESET up momentarily and release it. The Computer will repeat the power-up sequence, but the contents of user memory will not be affected.

Note: You do not need to remove the diskette during this Reset sequence.

Notes on Diskettes

Diskettes are precision recording media. Handle them carefully, as described under Section 5, **Care and Maintenance.** Be sure you don't touch the exposed diskette surfaces.

Before inserting the diskette, check the write protect notch. (See illustration.) If you do not want to write to that diskette, it is a good idea to leave it "write-protected". This way, the Operating System will not let you accidentally write to that diskette. To write-proteet a diskette, just leave the write-protect notch UNcovered. (See **Figure 4.**)

If you **do** want to write to the diskette, cover the write protect notch with gummed-foil tape provided with the diskette.

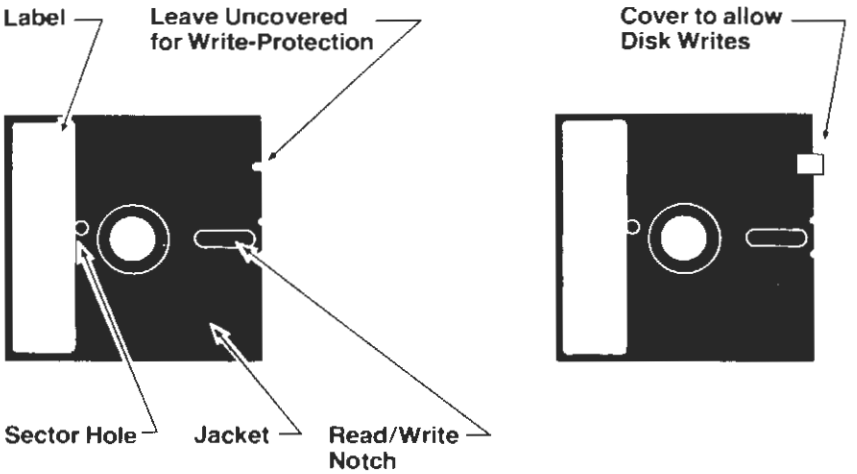


Figure 4. A diskette and a write-protected diskette.

Note: Any alteration of the data on the diskette — even the deletion of data or programs, requires that the diskette **NOT** be write-protected. (Cover the notch with gummed foil tape.)

Inserting a Diskette

1. If the drive door is closed, open it by pressing the release bar until the door springs open. (Refer to **Figure 5.**)
2. Remove the Operating System diskette from its storage envelope. Grasp the label side with the label facing away from the Display and insert it into the drive slot (see photo).
3. Gently push the diskette all the way into the slot. As the diskette reaches the back of the drive slot, you will feel a slight resistance from the seating/eject spring. Continue pressing the diskette in until it locks into place.
4. Close the door by moving it toward the left until it clicks into place. Some pressure may be required.

Removing a diskette

Never remove a diskette while the Drive Select light is on, or while a disk file is Open.

Press the Drive Release Bar. The door will open and the diskette will be partly ejected. Carefully remove it, taking care that the shiny diskette surface doesn't touch the chassis or drive door on the way out.

Note: Once a diskette has been seated in the drive, you must shut the drive door before you can remove the diskette.

Figure 5. Inserting a diskette (Label might extend vertically across the diskette).

Loading the Operating System

When the Computer prompts you to INSERT DISKETTE, carefully insert the Operating System diskette into the drive.

As soon as you close the drive door, the Computer will begin the Operating System bootstrap.

(If nothing happens when you close the drive door, the diskette is probably inserted incorrectly. Remove it and re-insert it correctly.)

The Computer will then execute a dignostic program before starting the Operating System. This lets you verify that the entire system is in working order — before you attempt any data processing.

After Completing the Diagnostic Program, the Computer will load the Operating System. See the **Operating System Manual** for details.

Keyboard Operation

The Keys can be divided into four functional groups: **Alphanumerics, Mode-Select, Numeric Keypad,** and **Control Keys,** as illustrated below:

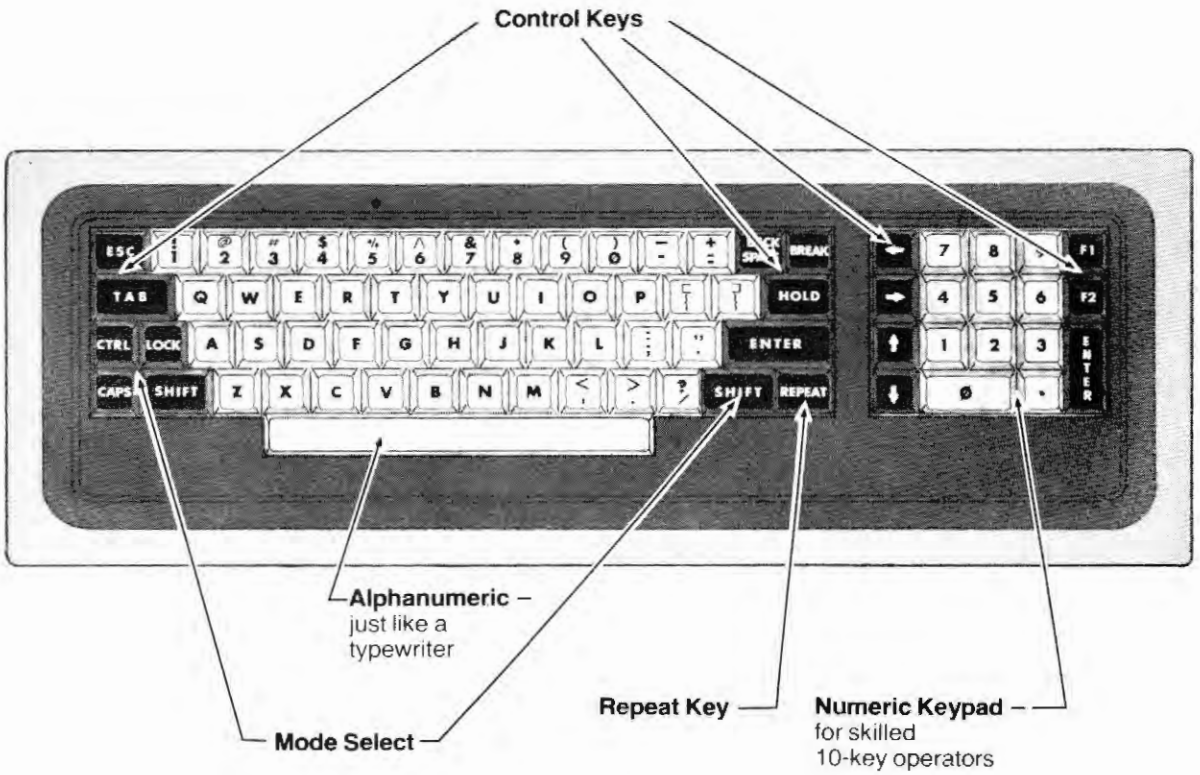
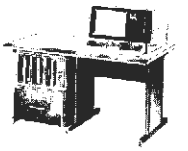


Figure 6. Functional groups of Model II keyboard.

You use the alphanumeric keys just as you would on a normal typewriter. However, each of these keys can send more than one character or code to the Computer, depending on which mode you've selected.



Keyboard Modes

The table below describes the *typical* use of the various modes. This use is determined by the Operating System or by the program currently in execution.

Unshift — Lets you input lower case letters, numbers and unshift punctuation symbols.

SHIFT LOCK — **Shift** — Lets you input capital letters and shift punctuation symbols. Hold down SHIFT while pressing the desired key, or press the LOCK key once so the red light comes on; while that light is on the Keyboard will output only Shifted characters. To return to the Unshift mode, press SHIFT again.

CAPS — **Caps** — Press the CAPS key once and the red light will come on. Typically, in the Caps mode, the alphabet keys A-Z send capital-letter codes only, and all other keys are unaffected. To return to the Unshift mode press CAPS once so the red light goes off.

CTRL — **Control** — Hold down the CTRL key while pressing one of the alphanumerics; this will output the "control" code assigned to that key.

Note: The Shift mode over-rides the Caps mode. So if both LOCK and CAPS lights are on, the Keyboard is in the Shift mode.

Control Keys

There are 12 Control Keys. Each key outputs a single control code — regardless of what mode the keyboard is in. How the Computer interprets these control codes depends on the Operating System, but here's a description of the *typical function* of each Control Key:

ESC — **Escape** – Usually used to exit for a subcommand, ignoring preceding characters in the current line.

TAB — **Tab** – Advances the cursor to the next tab position. The software typically sets Tab positions at 8, 16, 24, 32, etc.

Control Keys (cont.)

BACK SPACE — **Cancels** the last character typed and moves the cursor back one space.

BREAK — **Interrupts** anything in progress in the machine and returns to the command level.

HOLD — **Pauses** execution of the current program. Press HOLD a second time to continue execution.

ENTER — Signifies the **end of the current line**. The Display Cursor will drop to the beginning of the next line. Note that the two **ENTER** keys are identical. The rightmost **ENTER** is for convenient use with the numeric keypad.

SPACE BAR — Enters a **space** (blank) character and moves the cursor one space forward.

← — **Cursor Control — Moves cursor back** one space without cancelling previous character.

→ — **Cursor Control — Moves cursor forward** one space without entering a blank-space character.

↑ — **Cursor Control — In some programs, moves cursor up** one line without erasing previously entered characters.

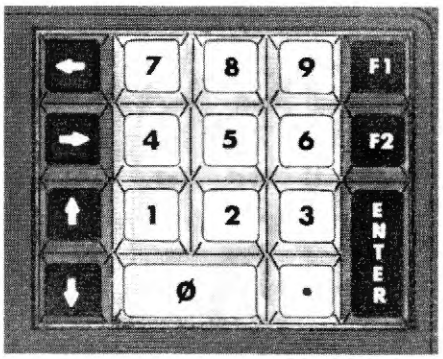
↓ — **Cursor Control — In some programs, moves cursor down** one line without erasing previously entered characters; does not signify end-of-line.

F1
F2 — **Function Keys — Software Programmable.** Outputs a control code which can be used by the Operating System or Applications Software for special functions.

Numeric Keypad

Clustered at the right of the Keyboard is a set of number keys, arrow keys and a second ENTER key. The arrow keys and ENTER keys are described above. The number keys are identical to the number keys on the top row of the main key cluster — except that these number keys output numeric character codes **only.** SHIFT , LOCK , CAPS and CTRL keys do not affect the output from the numeric key cluster.

These keys are convenient for data entry by skilled 10-key operators.

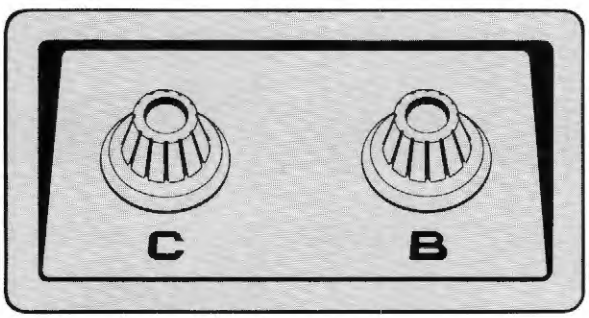


Repeat Key

This special convenience key works in conjunction with any key combination in any mode. Simply hold down REPEAT while you press the desired key(s). While you hold down these keys, the keyboard will output a steady stream of the desired characters.

Video Display Adjustment

Brightness and Contrast controls are located in the recessed area at the bottom left of the Display Console. Adjust as necessary for a comfortable display quality.



4 / Power-Up Diagnostic Messages

Whenever the Computer is turned on or Reset, it executes a built-in diagnostic program to help insure that the system is in good working order. If the Computer detects a hardware fault or other problem, it will display an error message and then stop. This checkout program reduces the chance that you will lose time or data by using a defective system without knowing it.

If one of these error messages is displayed, the first thing you should do is Reset the Computer, and attempt to duplicate the error. If the message re-appears, consult the table below.

Note: This program does not check for multiple faults; as soon as a single fault is found, the Computer displays the appropriate message and stops.

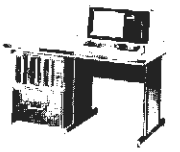
Error Code	What it means — What to do about it
DC	Floppy Disk Controller Error. Defective Diskette — Try another. Defective DC Chip or Drive.
D0	Drive not Ready. Improperly inserted diskette — Re-insert and reset. Defective diskette — Try another. Defective Drive.
SC	CRC Error. Invalid data on diskette or defective diskette — Try another.
TK	Record not found on bootstrap track. Improperly formatted diskette or defective diskette — Re-format or try another.
LD	Lost Data during read. FDC or Drive fault.
RS	Non Radio Shack diskette. Diskette is not Radio Shack Model II Operating System format — Remove, insert proper diskette, and reset.

(Continued on next page)

Error Code	What it means – What to do about it
CK	ROM Checksum Error. Defective ROM.
Z8	Z-80 Fault. Defective CPU.
MF	RAM Fault. Defective RAM in address range 1000H-7FFFH.
PI	PIO Chip Failure.
DM	DMA Chip Failure.
MB	RAM Fault. Defective RAM in address range 0000H-0FFFH
MH	RAM Fault (on 64K systems only). Defective RAM in address range 8000H-FFFFH
SI	SIO Chip Failure.

Before you ask for help . . .

Try the operation several times. Try using other diskettes. Recheck to see that all power and interconnections are right.



5 / Care and Maintenance

Care of Diskettes

In general, handle diskettes carefully, using the same precautions you use with tape cassettes and high-fidelity records. A small indentation, dust particle, or scratch can render all or part of a diskette unreadable — **permanently.**

- Keep the diskette in its storage envelope whenever it is not in one of the drives.
- Do not place a diskette in the drive while you are turning the system on or off.
- Keep diskettes away from magnetic fields (transformers, AC motors, magnets, TVs, radios, etc.). Strong magnetic fields will erase data stored on a diskette.
- Handle diskettes by the jacket only. Do not touch any of the exposed surfaces. **Don't try to wipe or clean the diskette surface**; it scratches easily.
- Keep diskettes out of direct sunlight and away from heat.
- Avoid contamination of diskettes with cigarette ashes, dust or other particles.
- Do not write directly on the diskette jacket with a hard point device such as a ball point pen or lead pencil; use a felt tip pen only.
- Store diskettes in a vertical file folder on a shelf where they are protected from pressure to their sides (just as phono records are stored).
- In very dusty environments, you may need to provide filtered air to the Computer room.

Tips on Labeling Diskettes

Each diskette has a permanent label on its jacket. This label is for "vital statistics" that will never change. For example, to help keep track of diskettes, it's a good idea to assign a unique number to each diskette. Write such a number on the permanent label. You might also put your name on the diskette, and record the date when the diskette was first put into use. Remember, use only a felt tip pen for marking.

This "permanent" label is not a good place to record the contents of the diskette — since that will change, and you don't want to be erasing or scratching out information from this label.

Keep such directory information on the storage box or in a separate record book, using the diskette number as a key to all record-keeping.

Figure 7. Labeled diskette.

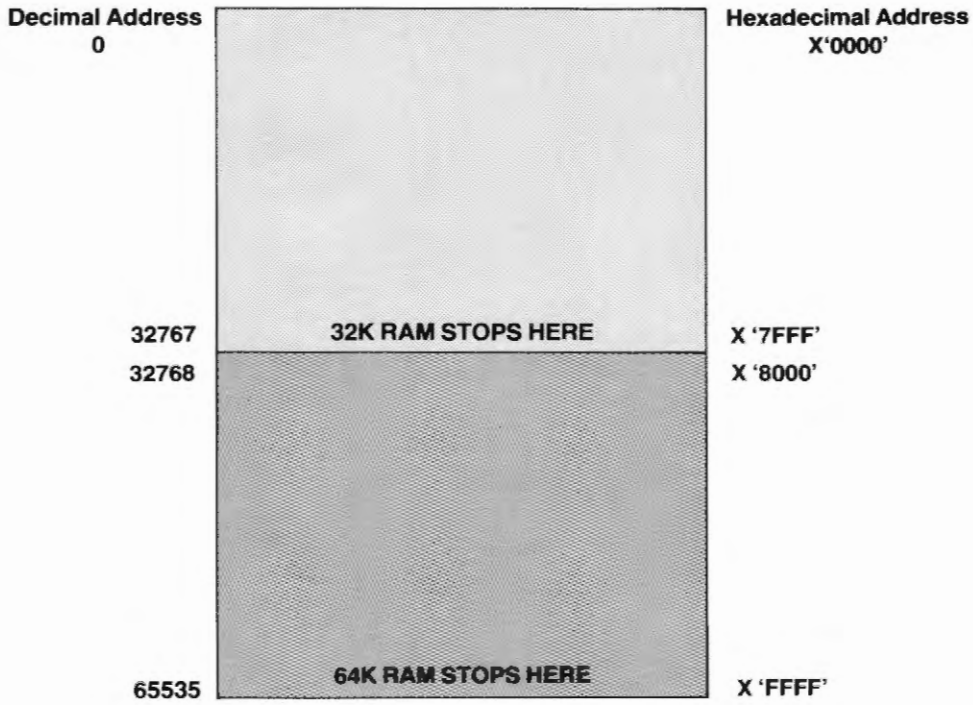
6 / Add-Ons

Inside the Display Console are slots for eight printed circuit boards. Four of these slots are taken up by the boards required by the basic system — Processor, Video Display, Floppy Disk Controller, and Random Access Memory (RAM).

Adding RAM

If your system has 32K of RAM, you can add another 32K by returning the unit to Radio Shack. Another 32K board will be added to the card-cage, leaving three slots still open for future enhancement of your system.

Systems shipped with 64K of RAM have four slots open for future additions, since a single 64K board is used in place of two 32K boards.



RANDOM ACCESS MEMORY CONFIGURATION AFTER SYSTEM IS INITIALIZED

Adding Disk Drives

Each drive you add will increase the on-line storage of your system by 509,104 bytes (roughly equivalent to 300 double-spaced typewritten pages).

Connection of additional Disk Drives is quite simple. The connector is on the back of the Display console, and a connector cable will be supplied with the Disk Expansion Unit.

Note: When the Disk Expansion Unit is **not** connected to the Model II, a specal terminator **must** be connected to the Disk Expansion connector on the back of the Display Console. The Model II comes with this terminator installed.

Further instructions are provided with the Expansion Unit, and can be added at the end of this Operation Manual.

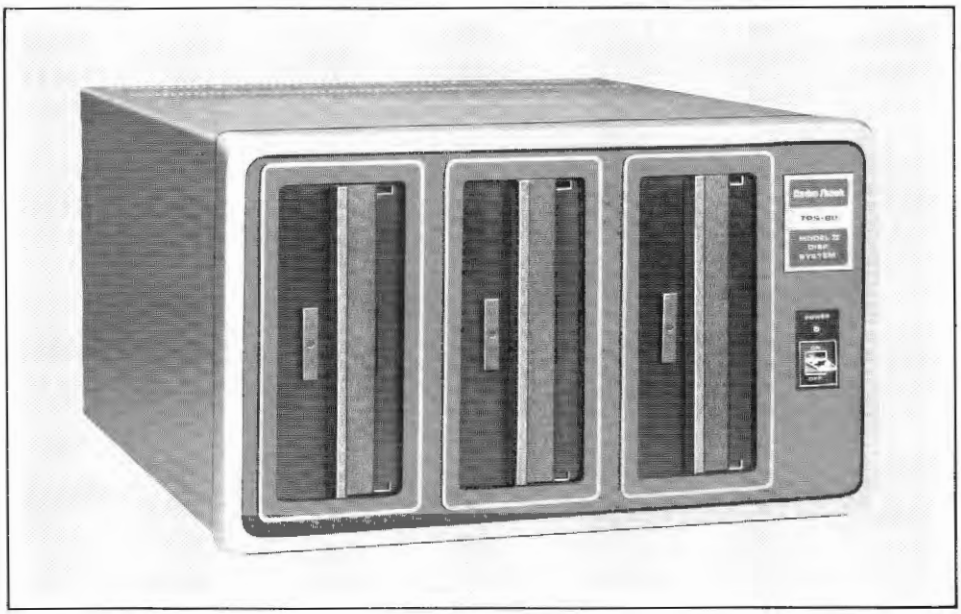


Figure 8. Disk Expansion Unit with three additional Disk Drives.

Connecting Serial Interface Equipment

The Model II provides two serial I/O channels, for connection to equipment like Telephone Interface Modems, Serial Line Printers, etc. Connection instructions will be provided with the serial equipment. You can add such instructions at the end of this Operation Manual. (See **Specifications** for a description of the Serial Interface Signals.)

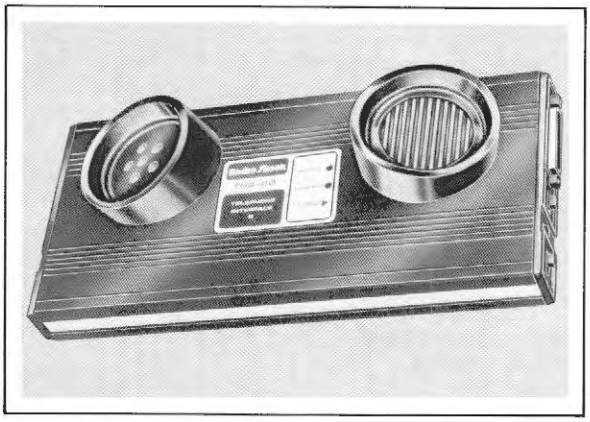


Figure 9. Radio Shack Telephone Interface II Modem for connection of computer system to telephone line.

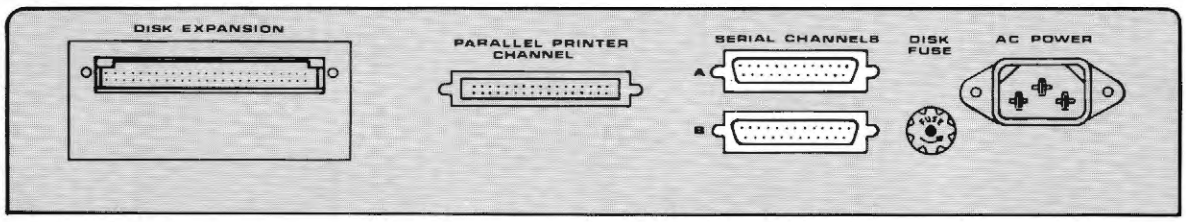


Figure 10. Connect Telephone Interface Modem (or other serial I/O device) to serial channel connection on the back panel of the video screen.

Parallel Interface Equipment

The Model II provides one parallel I/O channel, for connection to Radio Shack Line Printers and other compatible parallel-interface equipment. Connection instructions will be provided with the equipment. You can add such instructions at the end of this Operation Manual. (See **Specifications** for a description of Parallel Interface Signals.)

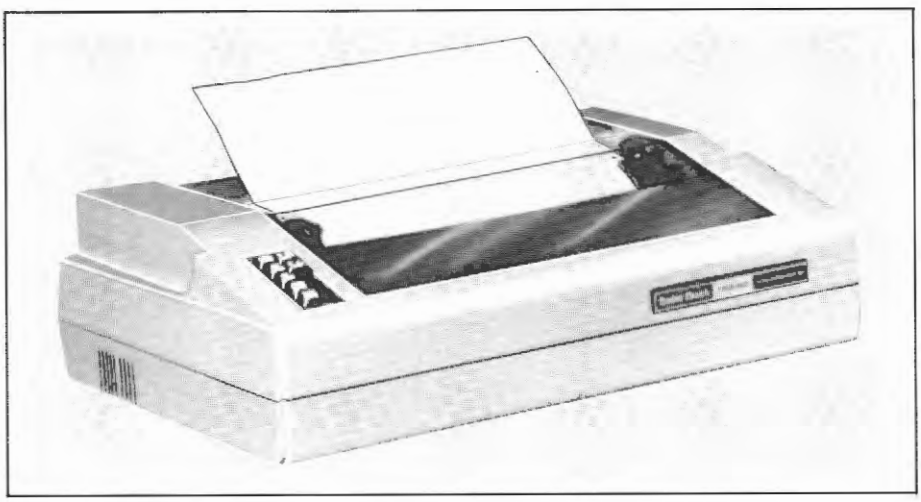


Figure 11. Radio Shack Line Printer III

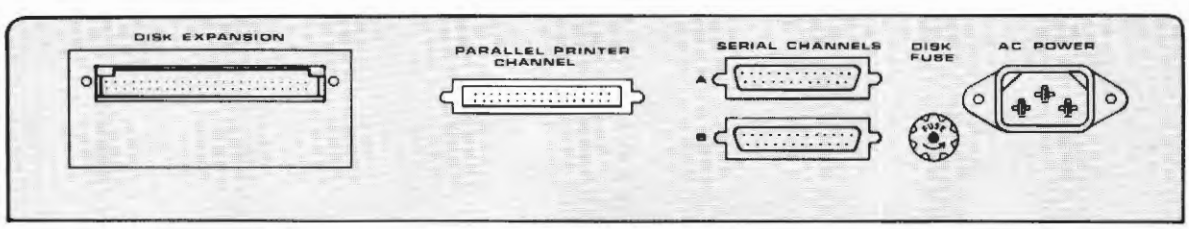


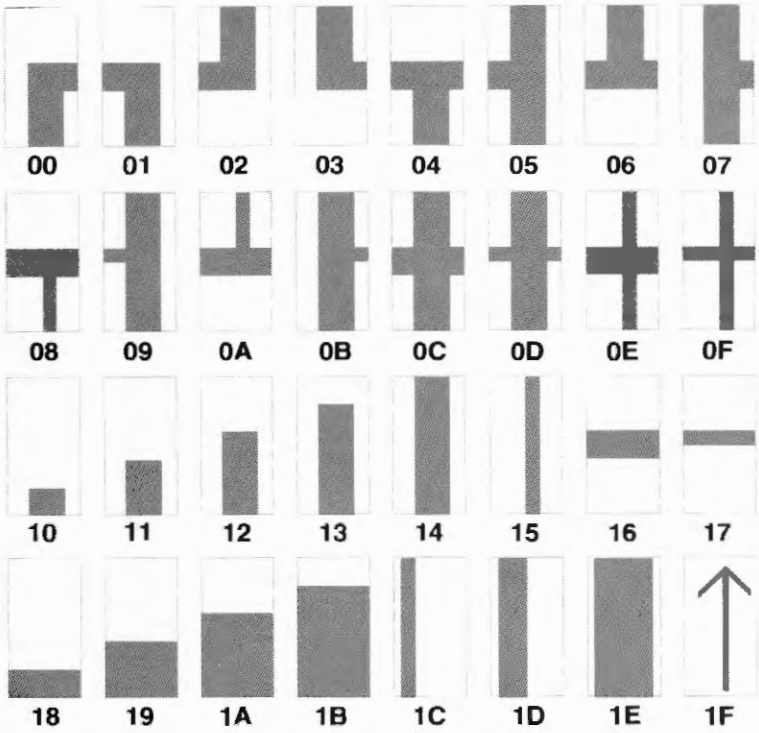
Figure 12. Connect Radio Shack Line Printer (or other compatible parallel interface equipment) to parallel channel connection on back panel of video screen.

7 / Specifications

Display Character Set

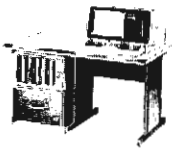
Here are the 32 graphics characters available on the Model II Display, along with their corresponding character codes. For further detail on the use of these codes, see the **Operating System Reference Manual.**

Note: A reverse-character (black on white) is available for each of the display characters, including alphanumerics.



Power Supply

Power Requirements	**105 - 130 VAC, 60 Hz,** **Grounded Outlet** **Maximum current drain: 2.0 Amps** **Typical current drain: 1.5 Amps**



Floppy Disk Drive

Total Storage Capacity (for User Data Capacity, See **Operating System Manual**)	**509,184 bytes per diskette**
Diskette Organization	
Tracks per Diskette	**77 (0-76)**
Sectors per Track	**26 (0-25)**
Bytes per Sector	**256** (except Track 0 = 128)
Data Transfer Rate	**500,000 bits per second**
Required Media	**Radio Shack 8'' Floppy Diskettes,** Catalog Number 26-4905, or 26-04906 (pkg of 10)
Preventive Maintenance Interval	**8000 Power-On Hours** (typical usage) **5000 Power-On Hours** (heavy usage)
Diskette Life*	**3.5 million passes per track**

*In practice, diskette life is usually limited by improper handling. Follow handling recommendations for maximum use.

Serial Interface Signals and Levels

Two channels are available, via the DB-25 connectors on the back of the Display Console. The signals and levels conform to the RS-232-C standard.

Channel A is designed to allow asynchronous or synchronous transmission. Channel B is designed for asynchronous transmission only.

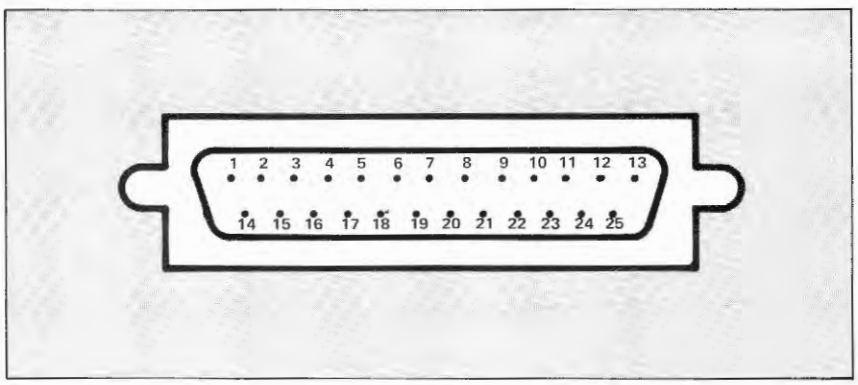
The DB-25 connector pin-outs and signals available are listed below.

CHANNEL A

STANDARD RS-232-C SIGNAL	(PIN #)
I/O TRANSMIT S.E.T.	15
GROUND	1,7
RECEIVED DATA	3
RECEIVER CLOCK	17
TRANSMIT CLOCK	24
DATA SET READY	6
CLEAR-TO-SEND	5
CARRIER DETECT	8
TRANSMIT DATA	2
REQUEST-TO-END	4
DATA TERMINAL READY	20

CHANNEL B

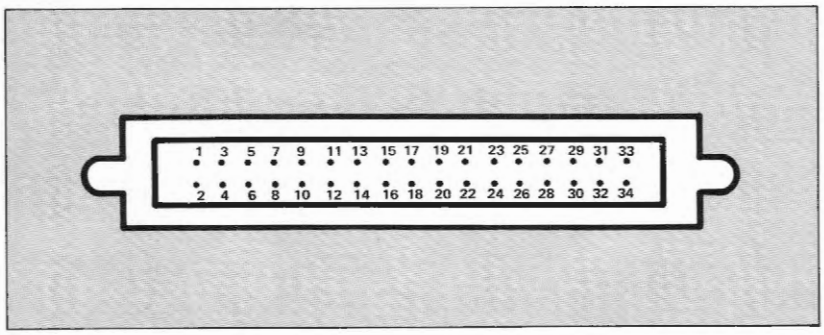
STANDARD RS-232-C SIGNAL	PIN #
GROUND	1,7
RECEIVED DATA	3
RECEIVER XMITTER CLOCK	17
DATA SET READY	6
CLEAR-TO-SEND	5
CARRIER DETECT	8
TRANSMIT DATA	2
REQUEST-TO-SEND	4
DATA TERMINAL READY	20



Parallel Interface Signals and Levels

The Model II includes a parallel interface designed for connection to a line printer via the 34-pin connector on the back panel of the Display Console. Eight data bits are output in parallel, and four data bits are input. All levels are TTL compatible.

The connector pin-outs and signals available are listed on the next page.



SIGNAL	FUNCTION	PIN#
STROBE*	1 µS pulse to clock the data from processor to printer	1
DATA 0	Bit 0 (lsb) of output data byte	3
DATA 1	Bit 1 of output data byte	5
DATA 2	Bit 2 of output data byte	7
DATA 3	Bit 3 of output data byte	9
DATA 4	Bit 4 of output data byte	11
DATA 5	Bit 5 of output data byte	13
DATA 6	Bit 6 of output data byte	15
DATA 7	Bit 7 (msb) of output data byte	17
ACK*	Input to Computer from Printer, low indicates data byte received	19
BUSY	Input to Computer from Printer, high indicates busy	21
PAPER EMPTY	Input to Computer from Printer, high indicates no paper – if Printer doesn't provide this, signal is forced low	23
SELECT	Input to Computer to Printer, high indicates device selected	25
PRIME	Output to Printer to clear buffer and reset printer logic	26
FAULT	Input to Computer from Printer high indicates fault (paper empty, light detect, deselect, etc.)	28
GROUND	Common signal ground	2,4,6,8,10 12,14,16,18, 20,22,24,27, 31,33
NC	Not connected	29,30,32,34

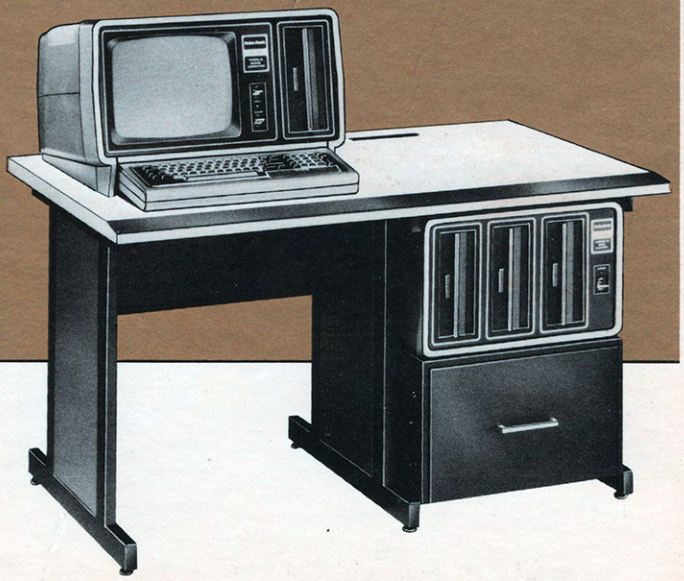
*These signals are active-low.

TRSDOS

Radio Shack®

TRS-80 Model II
Disk Operating System
Reference Manual

A Description of the Operating System:
General Information, Operator Commands,
Technical Information



CUSTOM MANUFACTURED IN THE U.S.A. BY RADIO SHACK TC A DIVISION OF TANDY CORPORATION

R A D I O S H A C K (R)

M O D E L I I T R S D O S

1 / G E N E R A L I N F O R M A T I O N

(M2DOS0 8/10/79)

CONTENTS

1 / General Information

Introduction

Model II TRSDOS ("Triss-Doss") is a powerful and easy-to-use Disk Operating System, providing a full set of library commands and utility programs. In addition, many useful System routines can be called directly by user programs.

Library commands are typed in from the TRSDOS command level to accomplish a variety of operations, including:

- Initialization--setting Printer parameters, date and time, etc.
- File-handling--copying, renaming, deleting, protecting, etc.
- File access--loading into memory, listing to Printer or Display, etc.
- Error identification

See Library Commands for details.

Utility programs provide essential services like:

- Formatting blank diskettes.
- Making backup copies of entire diskettes.

See the Utility programs for details.

System routines are executed via function codes instead of calls to absolute memory addresses. Routines available fall into six categories:

- System control
- Keyboard input
- Video Display input/output
- Line Printer output
- File access
- Computational functions

See the Technical Information section for details.

Notation

For clarity and brevity, we use some special notation and type styles in this book.

CAPITALS and punctuation
Indicate material which must be entered exactly as it appears. (The only punctuation symbols not entered are triple-periods, explained below.) For example, in the line:

DIR {SYS}

every letter and character should be typed exactly as indicated.

lowercase italics
Represent words, letters, characters or values you supply from a set of acceptable values for a particular command. For example, the line:

LIST filespec
indicates that you can supply any valid file specification (defined later) after LIST.

... (triple-periods)
Indicates that preceding items can be repeated. For example:
ATTRIB filespec (option,...)
indicates that several options may be repeated inside the braces.

This special symbol is used occasionally to indicate a blank space character (ASCII code 32).

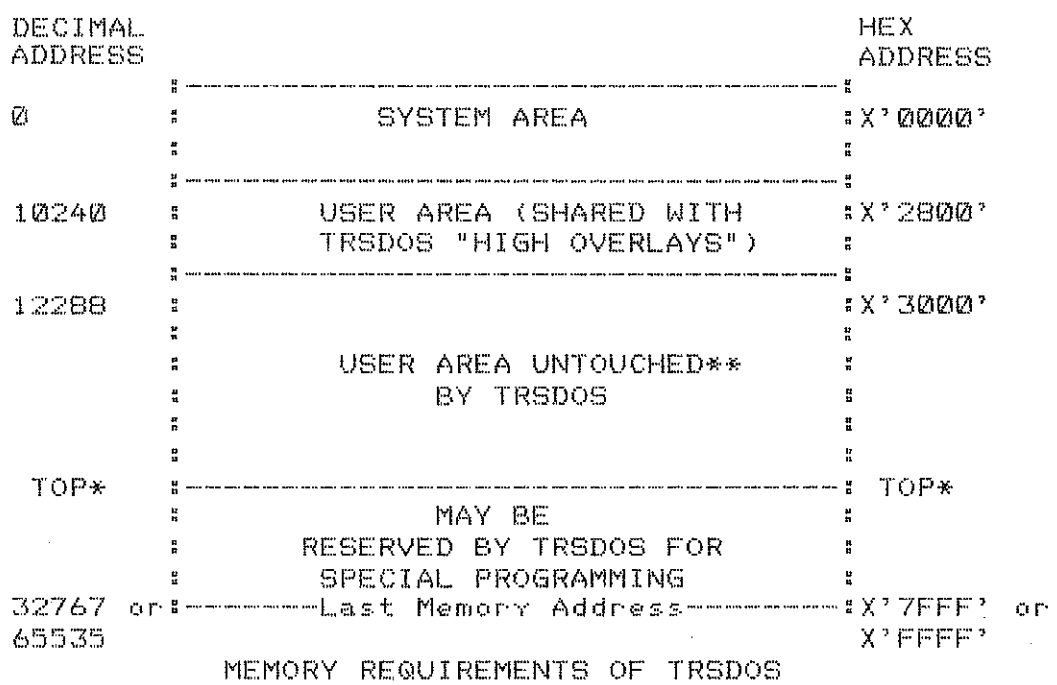
X'NNNN'
Indicates that NNNN is a hexadecimal number. All other numbers in the text of this book are in decimal form, unless otherwise noted. For example:
X'7000'
indicates the hexadecimal value 7000 (decimal 28672).

Memory Requirements

32K Model II Systems are supplied with a 32K version of TRSDOS; 64K Systems, with a 64K version. The two versions are NOT interchangeable--though the only difference is in the location of the Special Programming Area (see illustration).

TRSDOS occupies 6.1 tracks on the System diskette (39,040 bytes). However, only a small portion is actually in memory at any one time. The Supervisor Program, input/output drivers, and other essentials are always in memory. Auxiliary code is loaded as needed into an "overlay area".

Memory addresses 0 through 10239 (X'0'-X'27FF') are reserved for the Operating System. Certain commands, call "high overlays", also use memory addresses up to X'2FFF' (details provided in the Commands section. User programs must be located above X'27FF'; and you may want to locate them above X'2FFF' to allow use of the high overlays without loss of your program.



MEMORY REQUIREMENTS OF TRSDOS

Note: The term "user program" applies to any program which is not a part of TRSDOS. Therefore BASIC is a user program. For memory requirements of BASIC, see the BASIC Reference Manual.

*TOP is a memory protect address set by TRSDOS. If TRSDOS is not protecting high memory, then TOP is the same as "Last Memory Address".

**Single-drive COPY from one diskette to another, BACKUP and FORMAT use ALL user memory.

Loading TRSDOS

See the Operation Manual for instructions on connection, power-up and inserting the System diskette.

Note: A System diskette must be in Drive 0 (the built-in unit) whenever the Computer is in use. Whenever the Computer is turned on or reset, it will automatically load TRSDOS from Drive 0.

After the System starts up, it will prompt you to key in the date. Type in the date in MM/DD/YYYY form and press <ENTER>. For example:

07/07/1979 <ENTER>

for July 7, 1979.
Next the System will prompt you to key in the time. TO SKIP THIS QUESTION, press <ENTER>. The time will start at 00:00:00.

TO SET THE TIME, type in the time in HH.MM.SS 24-hour form. Periods are used instead of colons since they're easier to type in. The seconds are optional. For example:

14.30 <ENTER>

for 2:30 pm.

The System will record the time and date internally and return with the message:

TRSDOS READY

.....

Using the Keyboard

TRSDOS distinguishes between upper and lower case letters. Therefore

dir

is not the same as

DIR

Since TRSDOS commands are always capitalized, you'll probably find it convenient to operate the Keyboard in the Caps mode (press CAPS so the red light comes on). That way, all the alphabet-keys are interpreted as capital letters, regardless of whether the SHIFT key is being pressed.

Certain control keys are useful in the Command Mode:

ESC — Escape--Cancels the current line and lets you start over.

BREAK — Interrupts line entry and starts with a new line.

<- — Backspaces the cursor without erasing any characters. Use this to position the cursor for correcting a portion of a line.

-> — Forward-spaces the cursor without erasing any

characters. Use to position the cursor for correcting a portion of a line.

BACK SPACE — Backspaces the cursor, erasing the last character you typed. Use this to correct entry errors.

ENTER — Signifies end of line. When you press this key, TRSDOS will take your command. Only those characters appearing to the left of the cursor will be used.

HOLD — Pauses execution of a command. Press again to continue. Not functional in all commands.

TAB — Advances the cursor to the next 8-column position. Tab positions are at columns 0, 8, 16, 24, etc.

SPACE — Enters a space (blank) character and moves the cursor one character forward.

REPEAT — For convenience when you want to repeat a single key, hold down REPEAT while pressing the desired key. For example, to backspace halfway back to the beginning a the line, hold down REPEAT and BACKSPACE.

If you type any other control (non-alphanumeric, non-punctuation) key, a +/- symbol will be displayed for that key, but the control key code will be sent to the Computer. Such control keys will either be ignored or cause a parameter error to occur. See the Keyboard Code Map in the Appendix for control codes.

Entering a Command

Whenever the TRSDOS READY prompt is displayed, you can type in a command, up to 80 characters. If the command line is less than 80 characters (as is usually true), you must press <ENTER>to signify end-of-line. TRSDOS will then "take" the command.

For example, type:

```
CLS   <ENTER>
```

and TRSDOS will clear the Display.

Whenever you type in a line, TRSDOS follows this procedure:

First it looks to see if what you've typed is the name of a TRSDOS command. If it is, TRSDOS executes it immediately.

If what you typed is not a TRSDOS command, then TRSDOS will check to see if it's the name of a program file on one of the drives.

When searching for a file, TRSDOS follows the sequence drive 0, drive 1, etc--unless you include an explicit drive specification with the file name (described later on).

If TRSDOS finds a matching user file, it will load and execute the

file. Otherwise, you'll get an error message.

Command Syntax

Command syntax is the general form of a command, like the grammar of an English sentence. The syntax tells you how to put keywords (like DIR, LIST, and CREATE) together with the necessary parameters for each keyword. In this book, we present general syntax inside gray boxes, so they're easy to recognize.

There are three general command formats:

- No-file commands
- One-file commands
- Two-file commands

No-file commands take the form:

```
command {options} comment
  (options) is a list of one or more parameters that
    may be needed by the command. Some commands have
    no options. The braces {} around options can be
    omitted when no comment is added at the end
    of the command line.
  comment is an optional field used to document the
    purpose of the command-line. Comments are useful
    inside automatic keyboard entry files (see BUILD
    and DO commands).
```

For example:

```
TIME 14.30.00
```

is a no-file command, TIME, followed by the parameter option, 14.30.00. No braces are required in this example.

```
TIME {} Get current time.
```

is a no-file command, TIME, followed by a comment. Note that the braces are required to tell TRSDOS that "Get current time" is a comment and not an option list.

One-file commands take the form:

```
command filespec (options) comment
  filespec is a standard TRSDOS file specification
    as described later in this section.
  (options)--See description above.
  comment--See description above.
```

For example:

```
CREATE DATAFILE {NGRANS=40} Need 40 granules
```

is a one-file command, CREATE, followed by a filespec, DATAFILE, an option list, {NGRANS=40}, and a comment, Need 40 granules. In this example, the braces {} are required to tell TRSDOS where the option list ends and the comment begins.

Two-file commands take the form:

command filespec-1 delimiter filespec-2 (options) comment
filespec-1 and -2 are TRSDOS file specifications as described later in this section.
delimiter is one of the following:
blank space or spaces (indicated as ␣)
a comma , surrounded by optional spaces
TO surrounded by optional spaces
(options)--See description above.
comment--See description above.

For example:

```
RENAME PAYROLL1 TO PAYROLL2
```

is a two-file command RENAME, followed by filespec-1 (PAYROLL1), a delimiter, and filespec-2 (PAYROLL2).

File Specification

The only way to store information on disk is to put it in a disk file. Afterwards, that information can be referenced via the file name you gave to the file when you created or renamed it.

A file specification has the general form

filename/ext.password:d(diskname)
- filename consists of a letter followed by up to seven optional numbers or letters.
- /ext is an optional name-extension; ext is a sequence of up to three numbers or letters.
- .password is an optional password; password is a sequence of up to eight numbers or letters.
- :d is an optional drive specification; d is one of the digits 0,1,2,3.
- (diskname) is an optional field of up to 8 letters or numbers. If this field is included, it must be preceded by a drive specification.

Note: There can be no blanks inside a file specification. TRSDOS terminates the file specification at the first blank space.

For example:

FileA/TXT.Manager:3(ACCOUNTS)

references the file named FileA/TXT(ACCOUNTS) with the password Manager, on Drive 3, diskette name ACCOUNTS.

File Names

A file name consists of a name and an optional name-extension. For the name, you can choose any letter, followed by up to seven additional numbers or letters. To use a name extension, start with a diagonal slash / and add up to three numbers or letters.

For example:

MODEL2/TXT	INVNTORY	DATA11/BAS
NAMES/123	August/15	WAREHOUS
TEST	TEST1	TEST/1

are all valid and DISTINCT file names.

Although name-extensions are optional, they are useful for identifying what type of data is in the file. For example, you might want to use the following set of extensions:

/BAS BASIC program
/TXT ASCII text
/MIM Memory image
/REL Relocatable machine-language program
/DVR Input/Output driver

Drive Specification

If you give TRSDOS a file command like:

```
KILL TEST/1
```

the System will search for the file TEST/1, starting at Drive 0 and going to the other drives in sequence 1,2,3 until it finds the file.

Any time TRSDOS has to Open a file (e.g., to List it for you), it will follow the drive lookup sequence 0, 1, 2, 3. When TRSDOS has to write to a file, it will skip over any write-protected diskettes.

It is possible to tell the System exactly which drive you want to use, by means of the drive specification. A drive specification consists of a colon : followed by one of the digits 0,1,2, or 3, corresponding to one of the four drives.

For example:

```
KILL TEST/1:3
```

tells the System to look for file TEST/1 on drive 3 only.

Passwords

You can protect a file from unauthorized access by assigning passwords to the file. That way, a person cannot access a file simply by referring to the file name; he must also use the appropriate password for that file.

TRSDOS allows you to assign two passwords to a file:

- An Update word, which grants the user total access to the information (execute, read, write, rename or delete).
- An Access word, which grants the user limited access to the information (see ATTRIB).

When you create a file, the Update and Access words are both set equal to the password you specify. You can change them later with the PROT or ATTRIB command.

A password consists of a period . followed by 1 to 8 letters or numbers. If you do not assign a password to a file, the System uses a default password of 8 blanks. In this case the file is said to be unprotected; one can gain total access simply by referring to the file name.

For example, suppose you have a file named SECRETS/BAS, and the file has MYNAME as an update and access word. Then this command:

```
KILL SECRETS/BAS.MYNAME
```

will allow the file to be Killed.

Suppose a file is named DOMAIN/BAS and has blank passwords. Then the command:

```
KILL DOMAIN/BAS.GUESS
```

will not be obeyed, since GUESS is the wrong password.

Disk Names

When you reference a file like TESTER/BAS:3, TRSDOS will use whatever diskette is in drive 3. However, if you add a disk name to the file specification, TRSDOS will first check to see that the correct diskette is in the drive. (You assign disk names during the Format or Backup process.)

Note: Only the COPY command looks at the disk name and checks that the correct diskette is inserted. The other commands ignore the disk name in Version 1.1.

A disk name consists of from 1 to 8 letters and numbers inside parentheses (). When you include the disk name in a file specification, you must also include the drive number :d. Otherwise the disk name will be ignored.

For example:

COPY REPORT/TXT:0 TO REPORT/TXT:3(TXTFILES)
tells TRSDOS to copy the file REPORT/TXT on drive 0 to another file named REPORT/TXT on a disk named TXTFILES, using drive 3.

MODEL II TRSDOS

2 / LIBRARY COMMANDS

(M2DOS1 8/10/79)

2 / Library Commands

You can enter a library command whenever the TRSDOS READY prompt is displayed. (Programs can also call library commands. See Technical Information.)

In general, library commands will use memory addresses below X'2800'; however, the following "high memory commands" use addresses up to but not including X'3000':

APPEND COPY CREATE DUMP KILL LIST
BUILD ERROR VERIFY PURGE SETCOM

General rules for entering commands

Don't type any leading blanks in front of the command. For example:

```
TRSDOS READY
    DIR
```

is an error. Omit the spaces before DIR.

There must be at least one space between the command and any option list or comment. For example:

```
DIR(1)
```

is an error. Insert a space between R and (.

There can be any number of spaces between options.

```
DIR    (SYS  ,  PRT)
```

has the same effect as:

```
DIR (SYS,PRT)
```

When the syntax calls for a delimiter (@TO@, comma or space), any other non-alphanumeric non-brace characters will also serve, unless the special punctuation is par of an option keyword, e.g., the = sign in several commands.

```
LIST TEXTFILE (PRT:SLOW)
```

is equivalent to:

```
LIST TEXTFILE (PRT, SLOW)
```

When no ambiguity would result, the braces around the option list can be omitted.

```
CREATE FileA NRECS=100,LRL=64
```

is acceptable, but

```
CREATE FileA NRECS=100,LRL=64     Set up file area
```

is not, since the comment "Set up file area" will be taken as an invalid parameter.

```
DIR    SYS
```

is an error, since SYS is taken as an invalid drive specification. Use

```
DIR   (SYS)
```

instead.

AGAIN
Repeat Last Command

AGAIN

This command tells TRSDOS re-execute the most recently entered command.

Example

AGAIN
TRSDOS will re-execute whatever command was last entered.

Sample Use

AGAIN is useful after TRSDOS has returned an Input/Output error message instead of obeying a command. For example, suppose you type:
KILL OLDFILE:1
and the diskette in drive 1 is write-protected. Then you'll get an error 15. Put a write-enable tab on the diskette and type
AGAIN
Now TRSDOS will re-execute the command.

Suppose you are making multiple backup copies of a file from drive Ø to drive 1. Enter the COPY comand once; for second and third copies, use AGAIN. For example:
COPY DAYSWORK:Ø TO DAYSWORK:1
copies the file to a drive 1 diskette. Now put another diskette into drive 1 and type:
AGAIN
to repeat the copy using the new diskette.

APPEND
Append Files

```
:-------------------------------------------------------------:
:  APPEND file-1 TO file-2                                    :
:       file-1 and file-2 are file specifications.            :
:         The files must have the same type (Fixed or         :
:         Variable), the same Record Length, must both be     :
:         programs or both be data files (P or D in the       :
:         Directory listing).                                 :
:       "TO" is a delimiter. A comma or a space can           :
:         also be used.                                       :
:-------------------------------------------------------------:
```

APPEND copies the contents of file-1 onto the end of file-2. file-1 is unaffected, while file-2 is extended to include file-1. The file types (V or F) and record lengths (for fixed length record files) must match. See CREATE for more information on file types and record lengths.

Examples

```
APPEND Wordfile/2 TO Wordfile/1
A copy of Wordfile/2 is appended to Wordfile/1.

APPEND REGION1/DAT,TOTAL/DAT.guess
A copy of REGION1/DAT is appended to TOTAL/DAT, which is protected
with the password guess.
```

Sample Uses

Suppose you have two data files, PAYROLL/A and PAYROLL/B.

```
      PAYROLL/A                        PAYROLL/B
-------------------------        -------------------------
Atkins, W.R. .........           Lewis, G.E. .........
Baker, J.B. ..........           Miller, L.O. ........
Chambers, C.P. .......           Peterson, B. ........
Dodson, M.W ..........           Rodriguez, F. .......
Kickamon, T.Y ........
```

You can combine the two files with the command:

```
    APPEND PAYROLL/B TO PAYROLL/A
```

PAYROLL/A will now look like this:

```
Atkins, W.R. .........
Baker, J.B. ..........
Chambers, C.P. .......
Dodson, M.W ..........
Kickamon, T.Y. .......
Lewis, G.E. .........
Miller, L.O. ........
Peterson, B. ........
```

Rodriguez, F. .....

PAYROLL/B will be unaffected.

ATTRIB
Change a File's Passwords

ATTRIB file (ACC=password-1, UPD=password-2, PROT=level)
file is a file specification.
ACC=password-1 sets the access word equal to password-1.
If omitted, access word is unchanged.
UPD=password-2 sets the update word equal to password-2.
If omitted, update word is unchanged.
PROT=level specifies the protection level for access. If omitted, level is unchanged.

Level	Degree of access granted by access word
NONE	No access
EXEC	Execute only
READ	Read and execute
WRITE	Read, execute and write
RENAME	Rename, read, execute and write
KILL	Kill, Rename, read, execute and write (gives access word total access)

ATTRIB lets you change the passwords to an existing file. Passwords are initially assigned when the file is created. At that time, the update and access words are set to the same value (either the password you specified or a blank password). See Chapter 1 for details on access and update passwords.

Examples

ATTRIB DATAFILE ACC=JULY14, UPD=MOUSE, PROT=READ
Sets the access password to JULY14 and the update password to MOUSE. Use of the access word will allow only reading and executing the file.

ATTRIB PAYROLL/BAS.SECRET ACC=,
Sets the access word to blanks. The protection level assigned to the access word is left unchanged.

ATTRIB OLD/DAT.Apples UPD=,
Sets the update password to blanks.

ATTRIB PAYROLL/BAS.PW PROT=EXEC
Leaves the access and update words unchanged, but changes the level of access.

ATTRIB DATAFILE/1.PRN PROT=,
Changes the access level to Kill.

Sample Uses

Suppose you have a data file, PAYROLL, and you want an employee to

use the file in preparing paychecks. You want the employee to be able to read the file but not to change it. Then use a command like:

```
ATTRIB PAYROLL ACC=PAYDAY, UPD=Avocado, PROT=READ
```

Now tell the clerk to use the password PAYDAY (which allows read only); while only you know the password, Avocado, which grants total access to the file.

Suppose you want to temporarily stop access to the file. Then use the command:

```
ATTRIB PAYROLL.Avocado  PROT=NONE
```

Now the use of the password PAYDAY grants no access to the file. To restore the previous degree of access, use the command:

```
ATTRIB PAYROLL.Avocado  PROT=READ
```

AUTO
Automatic Command after System Start-Up

AUTO command-line command-line is a TRSDOS command or the name of an executable program file.

This command lets you provide a command to be executed whenever TRSDOS is started (power-up or reset). You can use it to get a desired program running without any operator action required, except typing in the date and time.

When you enter an AUTO command, TRSDOS writes command-line into its start-up procedure. The AUTO command does not check for valid commands; if the command line contains an error, it will be detected the next time the system is started up.

Examples

AUTO DIR (SYS)

Tells TRSDOS to write the command DIR (SYS) at the end of its start-up procedure. Each time the System is reset or powered up, it will automatically execute that command after you enter the date and time.

AUTO BASIC

Tells TRSDOS to load and execute BASIC each time the System is started up.

AUTO FORMS (W=80) For 8-1/2" wide paper

Tells TRSDOS to reset the printer width parameter each time the system is started up.

AUTO PAYROLL/CMD

Tells TRSDOS to load and execute PAYROLL/CMD (must be a machine-language program) after each System start-up.

AUTO DO STARTER

Tells TRSDOS to take automatic command input from the file named STARTER each time the System is started up. See BUILD and DO.

To erase an automatic command

Type:

AUTO

This tells TRSDOS to delete any automatic command and reset the start-up procedure to go directly to the TRSDOS READY mode.

Important Note

You cannot over-ride an automatic command. Therefore be sure a

program is fully debugged before making it an automatic command. Furthermore, programs which are executed via the AUTO function should normally provide a means of exiting to the TRSDOS READY mode. (Unless the BREAK key is blocked by the user program, pressing BREAK will get you back to TRSDOS.)

Sample Use

Suppose you want the TRSDOS to run a certain BASIC program, MENU, each time it is started up. That way, an operator can turn on the Computer and get going without having to enter any TRSDOS commands.

Then use the command:
AUTO BASIC MENU -F:2
to prepare the System to run the BASIC program each time it starts up. (See BASIC Reference Manual for details on loading BASIC.)

BUILD
Create an Automatic Command Input File

```
:-------------------------------------------------------------------------:
: BUILD file                                                              :
:       file is a file specification which cannot include                 :
:         an extension.                                                   :
:-------------------------------------------------------------------------:
```

This command lets you create a an automatic command input file which can be executed via the DO command. The file must contain data that would normally be typed in from the keyboard.

BUILD files are primarily intended for passing command lines to TRSDOS just as if they'd been typed in at the TRSDOS READY level.

BUILDing New Files

When the file you specify does not exist, BUILD creates the file and immediately prompts you to begin inserting lines. Each time you complete a line, press <ENTER>. BUILD will give you another chance to re-do the line or keep it. Press <ESC> to erase and re-do the line; <ENTER> to store it and start the next line.

While typing in a line, you can use <- and -> to position the cursor for corrections. <BACKSPACE> also works as usual. Be sure the cursor is at the end of the desired line before you press <ENTER>.

To end the BUILD file, simply press <ENTER> at the beginning of the line, i.e., when the message:

```
  TYPE IN UP TO 80 CHARS
  ..............................................................
```

Is displayed.

Note: Pressing <BREAK> will also end the file. Only those lines that have been flagged like this:

```
    *** LINE STORED IN FILE ***
```

will be saved.

Editing Existing BUILD-Files

When you specify an existing file in the BUILD command, TRSDOS assumes you want to edit that file. Before starting the edit, it copies the file into a new file with the same name but with the extension /OLD. That way, you will have a backup copy of the file as it was before being edited.

Note: Editing an existing BUILD-file requires that you have write-access to the file. That is, if the access password has a protection level which does not allow writing, then you must supply the update password.

Example:
Suppose the file STARTER already exists, and you type the command:
BUILD STARTER
TRSDOS will first copy STARTER into a new file STARTER/OLD (if STARTER/OLD already exists, previous contents are lost). Then it will let you begin editing the file. As you edit the file, the updated lines will be written into STARTER.

BUILD will display the existing contents of the file, one line at a time. Beneath the line is an option list:
K (keep), D (delete), I (insert), R (replace), Q (quit) ?.

Type the first letter of the desired option and press <ENTER>.

KEEP OPTION: Copies the line as-is into the new file, and displays the next line for editing.

DELETE OPTION: Deletes the line by not copying it into the new file, and displays the next line for editing.

INSERT OPTION: Allows you to insert lines AHEAD of the line being displayed. Using this option is like entering lines into a new file as described above.

After you press <ENTER>, TRSDOS will give you a chance to erase and re-start the insert line, or to store the insert line. Press <ESC> to erase, <ENTER> to store it. You can then insert another line.

To stop inserting, press <ENTER> at the beginning of the line. TRSDOS will then display the next line and the option list.

REPLACE OPTION: Deletes the displayed line and lets you insert replacement lines. Entering replacement lines is like entering lines with the insert option. Press <ENTER> at the beginning of a line to stop inserting. TRSDOS will display the next line and the option list.

QUIT OPTION: Ends the editing session. All remaining lines will be copied into the new file as-is. Before closing the file, TRSDOS will ask if you want to add new lines to the end. (If you simply want to add to a file but make no other changes, type Q at the beginning of the edit session.)

At end of file

Whenever TRSDOS reaches the end of the file, it will ask if you want to add new lines at the end. Type Y <ENTER> to add, N <ENTER> to end the editing session.

Adding lines at the end of a file is just like using the insert line option described above. Type <ENTER> at the beginning of a line to stop adding and close the file.

To recover a BUILD file's previous contents

There are a couple of cases in which you may need to do this. Let's assume you are editing a file named STARTER.

1. After ending the edit session, you realize that you have made an error, and you want to recover the previous version of the file.

2. You accidentally press <BREAK> and end the edit session; only those lines that have been flagged like this:

*** LINE STORED IN FILE ***

will be saved in the new file named STARTER.

The previous file contents are now stored in STARTER/OLD. If you want to re-edit this file, you must Copy it or Rename it to a file name without an extension. For example, you might use these commands:

```
COPY STARTER/OLD TO STARTER {ABS}
```

Now you can edit the previous file's contents. Type:

```
BUILD STARTER
```

To start editing.

CLEAR
Clear User Memory

CLEAR

This command gets you off to a fresh start. It zeroes user memory (loads binary zero into each memory address above X'27FF'). It also returns the System to the state it is in when the first TRSDOS READY message appears: initializes the input/output drivers, un-protects all memory and resets the stack.

Example

CLEAR

CLOCK
Turn on Clock-Display

```
:-------------------------------------------------------------:
: CLOCK {switch}                                              :
:   switch is one of the options, ON or OFF.                  :
:     If switch is not given, ON is assumed.                  :
:-------------------------------------------------------------:
```

This command controls the real-time clock display in the upper right corner of the Video Display. When it is on, the 24-hour time will be displayed and updated once each second, regardless of what program is executing.

TRSDOS starts up with the clock off.

Note: The real-time clock is always running, regardless of whether the clock-display is on or off.

Examples

CLOCK
Turns on the clock-display.

CLOCK OFF
Turns off the clock-display.

CLS
Clear the Screen

CLS

This command clears the Video Display. Use it to erase information that you don't want others to see, for example, file specifications which include passwords.

Example

```
CLS
```

Sample Use

```
CREATE PERSONNL/BAS.secure      NGRANS=200
CLS
```

COPY
Copy a File

COPY file 1 TO file 2 {ABS}
file 1 and 2 are file specifications. For single-drive copies, the file names MUST BE DIFFERENT.
TO is a delimiter. A comma or space can also be used.
ABS is an optional parameter telling TRSDOS to copy file 1 even if file 2 already exists. The previous contents of file 2 will be lost.

This command copies file 1 into the new file defined by file 2. If a disk name is included in either file specification, TRSDOS will ensure that the appropriate diskette is inserted before making the copy. This allows you to copy a file from one diskette to another, using A SINGLE DRIVE if necessary.

When you do not add the ABS ("absolutely") parameter, TRSDOS will NOT overwrite an existing file that matches the specification file 2. Instead it will give you an error message. Use the ABS option to overwrite (destroy) an existing file.

Normally, COPY uses memory below X'3000'; however, when copying from one diskette to another in a SINGLE drive, it will use memory up to the start of protected memory (see "Memory Requirements of TRSDOS").

The disk name must always be preceded by a drive specification; otherwise it will be ignored.

For single-drive copies from one disk to another, both disk names and drive specifications must be provided.

Examples

COPY OLDFILE/BAS TO NEWFILE/BAS
Copies OLDFILE/BAS into a new file named NEWFILE/BAS. TRSDOS will search through all drives for OLDFILE/BAS, and will copy it onto the first diskette which is not write-protected.

COPY NAMEFILE/TXT:0(DEPTC) TO FILEA/TXT:0(DEPTA)
This command specifies a one-drive copy from a diskette named DEPTC to another diskette named DEPTA. TRSDOS will provide the necessary prompting to accomplish the copy. Since it's a one-drive copy, file names must be different.

COPY FILE/A TO FILE/B:1(DOUBLE)
This command copies FILE/A to FILE/B. TRSDOS will search all drives for FILE/A, and will require you to have or insert a diskette named DOUBLE in drive 1.

COPY NEWFILE TO OLDFILE {ABS}

Performs the copy even if OLDFILE already exists, in which case its previous contents are lost.

Sample Use

Whenever a file is updated, use COPY to make a backup file on another diskette. You can also use COPY to restructure a file for faster access. Be sure the destination diskette is already less segmented than the source diskette; otherwise the new file could be more segmented than the old one. (See FREE for information on file segmentation.)

To rename a file on the same diskette, use RENAME, not COPY.

(M2DOS2A 8/6/79)

CREATE
Create a Preallocated File

```
:-------------------------------------------------------------:
: CREATE file {NGRANS=n1, NRECS=n2, LRL=n3, TYPE=letter}      :
:    file is a file specification.                            :
:    NGRANS=n1 indicates how many granules to allocate.       :
:       If NGRANS is ommitted, the number of granules         :
:       allocated is determined by NRECS and LRL.             :
:    NRECS=n2 indicates how many records to allow for.        :
:       If NRECS is omitted, NGRANS determines the size of    :
:       the file. When NRECS is given, LRL must also be       :
:       be given.                                             :
:    LRL=n3 indicates the record length (Fixed-length         :
:       records only). n3 must be in the range <1,256>.       :
:       If LRL is omitted, LRL=256 is used. When LRL is       :
:       given, NRECS must also be given.                      :
:    TYPE=letter specifies the record type: letter equals     :
:       F (Fixed-length records) or V (Variable-length        :
:       records).  If TYPE is omitted, TYPE=F is used.        :
:    Note: {NGRANS} and {NRECS,LRL} are mutually              :
:       exclusive.                                            :
:-------------------------------------------------------------:
```

This command lets you create a file and pre-allocate (set aside) space for its future contents. This is different from the default (normal) TRSDOS procedure, in which space is allocated to a file dynamically, i.e, as necessary WHEN DATA IS WRITTEN INTO THE FILE.

With preallocated files, unused space at the end of file is NOT deallocated (recovered) when the file is Closed. With dynamically allocated files, on the other hand, unused space at the end of the file IS recovered when the file is Closed.

Note: With pre-allocated files, TRSDOS will allocate extra space when you exceed the pre-allocated amount during a write operation.

You may want to use CREATE to prepare a file which will contain a known amount of data. This will usually speed up file write operations, since TRSDOS won't have to do periodic allocations during the write operations. File reading will also be faster, since pre-allocated files are less dispersed on the diskette--requiring less motion of the read/write mechanism to locate the records.

Examples

```
CREATE DATAFILE/BAS  NRECS=300, LRL=256
```
Creates a file named DATAFILE/BAS, and allocates space for 300 256-byte records.

```
CREATE TEXT/1   NGRANS=100, TYPE=V
```

Creates a file named TEXT/1, and allocates 100 granules. The file will contain variable-length records.

CREATE NAMES/TXT.IRIS NRECS=500, LRL=30
Creates a file named NAMES/TXT protected by the password IRIS. The file will be large enough to contain 500 records, each 30 bytes long.

Determining the size of the file

You can allocate space according to number of granules or number of records. (A granule contains 1280 bytes; a record contains from 1 to 256 bytes, depending on LRL.)

The granule is the unit of allocation in TRSDOS; if you ask for 30 granules, that's exactly how much space the file will get.

If, on the other hand, you specify the number of records, TRSDOS will give you the NUMBER OF GRANULES which are required to CONTAIN that many records. For example, if you specify 100 records and a record length of 40, you're asking for a total of 100 * 40 = 4000 bytes. Since TRSDOS allocates spaces in units of granules (1280 bytes), you'll actually get 4 granules--containing 5120 bytes.

Record Length (Fixed-Length Files Only)

A record is the quantity of data TRSDOS processes for you during disk operations. The record length can be any value from 1 to 256.

File Type

TRSDOS allows two types of files: Fixed-Length Record (FLR) files and Variable-Length Record (VLR) files. With FLR files, the record length (from 1 to 256) is set when the file is created, and it cannot be changed. With VLR files, the length of each record is independent of all other records in the file. For example, record 1 might have a length of 70; record 2, 33; record 3, 225; etc. Variable length records consist of a length byte followed by the data, and can contain up to 256 bytes INCLUDING the length byte.

For further explanation of file structure, allocation and types, see Technical Information.

To CREATE a file to be used by BASIC

1. Decide how many records the file will contain. (This is just an estimate. If the file exceeds this number, it will automatically be extended.)
2. If it is a Direct access file, determine the optimum record length (from 1 to 256). If it is a sequential access file, the record length must equal 1.
3. Use a CREATE command like this:
   CREATE file {NRECS=*number*, LRL=*length*}

Sample Use

Suppose you are going to store personnel information on 250 employees, and each data record will look like this:

- Name (Up to 25 letters)
- Social Security Number (11 characters)
- Job Description (Up to 92 characters)

Then your records will need to be 25+11+92=128 bytes long.

You could create an appropriate file with the command:
CREATE PERSONNL/TXT NRECS=250, LRL=128

Once created, this preallocated file will allow faster writing than would a dynamically allocated file, since TRDOS won't have to stop writing periodically to allocate more space (unless you exceed the pre-allocated amount).

DATE
Reset or Get Today's Date

```
:-------------------------------------------------------------:
: DATE {mm/dd/yyyy}                                           :
:    mm is a two-digit month specification                    :
:    dd is a two-digit day-of-month specification             :
:    yyyy is a four-digit year specification                  :
:    If mm/dd/yyyy are given, TRSDOS resets the               :
:       date. If mm/dd/yyyy is omitted, TRSDOS                :
:       displays the current date and time.                   :
:-------------------------------------------------------------:
```

This command lets you reset the date or display the date and time.

The operator sets the date initially when TRSDOS is started up. After that, TRSDOS updates the time and date automatically, using its built-in clock and calendar.
You can enter any four-digit year after 1599.

When you request the date, TRSDOS displays it in this format:

```
    THU JUL 19 1979 200 -- 14.15.31
```

for Thursday, July 19, 1979, the 200th day of the year, 2:15:31 pm.

Note: If the time passes 23.59.59, TRSDOS does not start over at 00.00.00. Instead, it continues with 24.00.00. However, the next time you use the TIME or DATE command, the time will be converted to its correct 24-hour value, and the date will be updated. If you let the clock run past 59.59.59, it will recycle to 00.00.00, and the date will not be updated to include the 60-hour period.

Examples

DATE
Displays the current date and time.

DATE 07/18/1979
Resets the date to July 18, 1979, and displays the new date information.

Sample Use

In addition to resetting and getting the current date, this command can be used to provide complete date information on any date.

For example, the command:

```
   DATE 12/07/1941
```

tells you that December 7, 1941 fell on a Sunday, the 341st day of the year. It also resets the current date.

DEBUG
Start Debugger

DEBUG {switch}
switch is one of the following parameters:
ON turns on the debugger.
OFF turns off the debugger.
If switch is omitted and debugger is off, TRSDOS tells you so.
If switch is omitted and debugger is on, TRSDOS enters the debug monitor.

This command sets up the debug monitor, which allows you to enter, test, and debug machine-language programs. It also includes an Upload function to allow transmission of data from another device to the Model II, via the built-in serial interface (Channel B).

DEBUG loads into the high memory area sometimes reserved by TRSDOS for special programming (see TRSDOS Memory Map). While DEBUG is on, TRSDOS will automatically protect this area from being overlaid by BASIC or other user programs. To use DEBUG from BASIC, you must turn DEBUG on before you start BASIC.

While DEBUG is on, every time you attempt to load and execute a user program, you will enter the debug monitor. In this mode, you can enter any of a special set of single-key commands for studying how your program is working.

DEBUG can only be used on programs in the user area (X'2800' to TOP).

Examples

DEBUG
If DEBUG is off, this command tells you so. If it is on, this command enters the debug monitor.

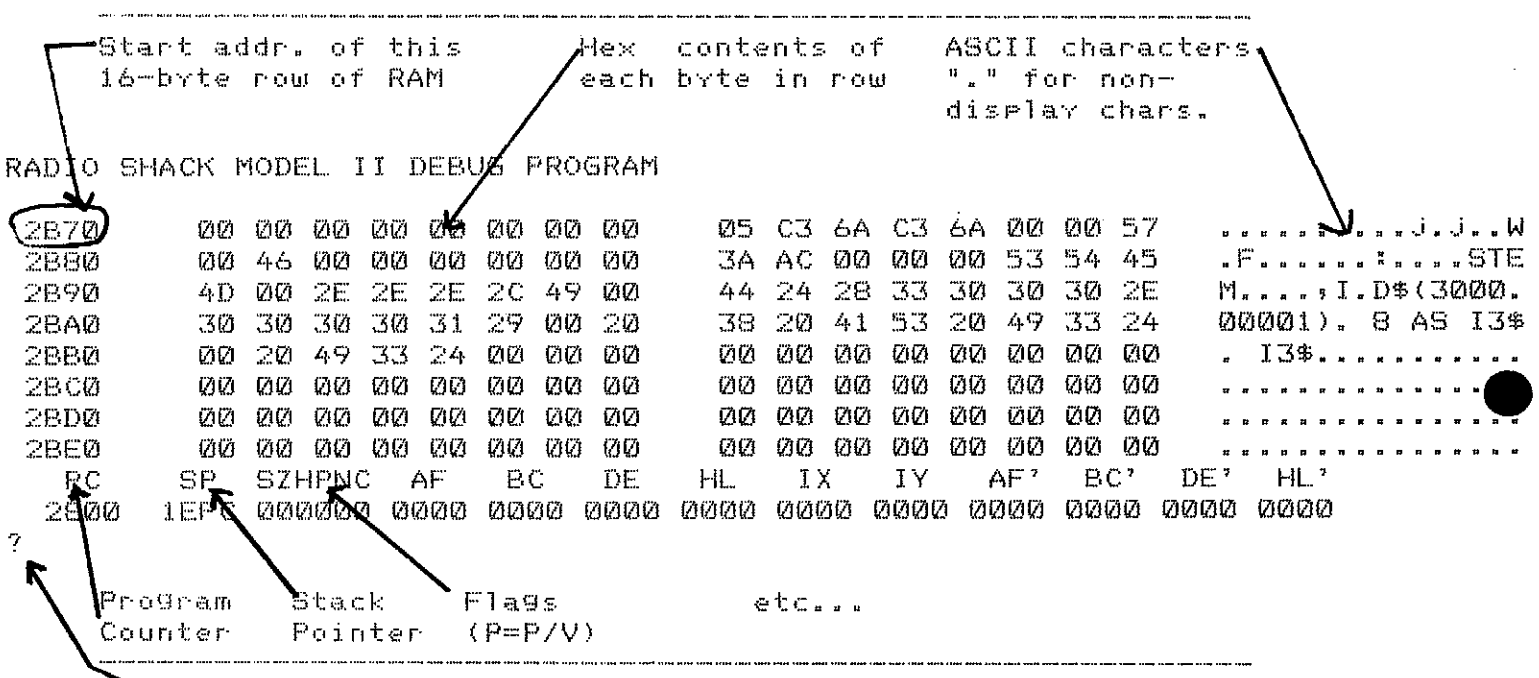
DEBUG OFF
Turns off DEBUG and un-protects high memory.

DEBUG ON
Turns on DEBUG: i.e., loads the debugger into high memory, protects high memory, and sets up a "scroll window"--a block of lines that will be scrolled. The scroll window will consist of the bottom 11 lines on the display. The top 13 lines will be used to contain the debug monitor display.

To enter the debug monitor

Type:
DEBUG ON
DEBUG

While DEBUG is on, you can also enter the debug monitor simply by typing file specification of a user program. TRSDOS will load the program and transfer control to the debugger. The transfer address for the PC register display.



The ? is the command prompt, meaning that you can enter one of the single-key commands. Press <H> (for "help") to display a "menu" or list of debugger commands. To enter one of the commands, press the letter which is capitalized in the command menu. For example, to enter the memory command ("raM"), press <M>.

Most commands will prompt you to enter additional information or subcommands. While entering commands and subcommands, the following keys are useful:

<ESC>	Returns to the ? prompt and cancels the command you're in.
<BACKSPACE>	Backspaces the cursor and erases previous character.
<-	Cursor back without erasing.
->	Cursor forward without erasing.
<F1>	In certain subcommands, homes the cursor.
<TAB>	In certain subcommands, tabs the cursor.

Command Description

Break

Press <B> to set a breakpoint in your program. When execution reaches a breakpoint, control returns to the debug monitor, with the program counter pointing to the breakpoint address. To continue from that point, press <C>. The original instruction will be executed--BUT THE BREAKPOINT WILL NOT BE REMOVED. It will still be there the next time that address is reached.

Note: Place breakpoints at the beginning byte of an opcode--NEVER in the middle of an instruction.

Press <B> to enter the Break command. TRSDOS prompts you to enter the breakpoint number. Up to eight breakpoints are allowed, so type in a number from 1 to 8. Next TRSDOS prompts you to enter the new address for that breakpoint. If the breakpoint has previously been set, TRSDOS displays the old breakpoint address, and the original instruction that goes in that address.

Note: While a breakpoint is in place, a D7 is displayed in the memory display for the breakpoint address.

For example:

```
   ? B #=1 A=2800
```

Puts a breakpoint (#1) at address X'2800'. The memory display for X'2800' will show a D7.

```
RADIO SHACK MODEL II DEBUG PROGRAM
 2800      D7 5B 2E D6 43 F3 43 DE    42 96 44 C9 42 29 5E 95    .↑..C.C.B.D.B)→.
 2810      60 A1 3A 28 4E 17 4F 29    41 7A 4E 8B 4F 91 4F 1E    `.:(N.O)AzN.O.O.
 2820      50 33 50 FD 60 00 61 03    61 6D 64 C0 64 D3 64 E4    P3P.`.a.amd.d.d.
 2830      60 E7 60 EA 60 D6 43 0A    44 36 44 83 44 36 5C 2F    `.`.`.C.D6D.D6↓/
```

To delete a single breakpoint without affecting any others, press <ENTER> instead of providing a new address for the breakpoint. To delete all breakpoints, press <E> for "empty breakpoint table".

Continue

Press <C> to enter this command. It resumes execution of your program at the address pointed to by PC. Use it after the debugger has stopped at a breakpoint. The original instruction at the breakpoint address will be executed, but the breakpoint will remain in place.

Decimal Format

Press <D> to enter this command. It displays all addresses in decimal form. However, the contents of all registers and memory addresses are still displayed in hexadecimal. In the decimal display format, you must enter all addresses as five-digit decimal numbers.

Empty Breakpoint Table

Press <E> to empty the breakpoint table. All breakpointed instructions will be restored.

Find Hex String

Press <F> to start this command. It will search in memory for a string up to 20 bytes long. You must enter the search string in hexadecimal format. Press <ENTER> when you have typed in the entire string. The debug monitor will display the first occurrence of the string. It it is not in the search area, the current memory display is unchanged.

For example:

```
? F S=2800 E=4000 D=C30070
```

Searches memory from X'2800' through X'4000' for the three-byte hexadecimal string X'C30070'.

Hex Format

Press <X> to restore the Display to hexadecimal format. In this mode, all addresses must be entered as four-digit hexadecimal numbers.

Jump

Type <J> to enter this command. The debugger will prompt you to type in the address to Jump to. (Jumping to a breakpointed instruction will cause an immediate return to the debug monitor.)

For example:

```
      ? J A=2800
```

starts execution at X'2800'.

Load (Copy memory to memory)

Type <L> to enter this command. It moves a block of memory. The debugger will prompt you to type in the start (S=) and END (E=) addresses of the block to be copied, and the destination address (T=) for the first byte moved.

The move is incremental: the first byte is moved to the first destination address, then the second to the second destination address, etc.

Examples:

```
  ? L S=2800 E=28F0 T=3000
```

Copies addresses from X'2800' to X'28F0' into memory from X'3000' to X'30F0'.

You can use this command to fill memory with a specific value, by putting the desired value in address nnnn, and using a command like this:

```
  ? L S=nnnn E=xxxx T=nnnn+1
```

This will copy the value in nnnn into every location from nnnn+1 to xxxx. For example, if 2800 contains a X'20', then the command:

```
  ? L S=2800 E=3000 T=2801
```

fills memory from 2801 to 3000 with X'20'.

Debug Off

Type <O> to exit the debug monitor and turn off DEBUG. All breakpoints set by the B command will be removed from your program, AND EXECUTION WILL CONTINUE AT THE ADDRESS SHOWN IN PC.

Print Display

Type <P> to send a copy of the Display to the Printer. Printer must have been initialized during TRSDOS startup or by the FORMS command.

Examine and Change Memory

Type <M> to enter this command. The debugger will prompt you to type in the starting address of memory to be examined. As soon as you type in the complete address, the memory display will show the 128-byte area starting with that address. While the A=.... prompt is present, you can scroll through memory 16 bytes at a time by pressing <ENTER>.

TO MODIFY ANY MEMORY IN THE DISPLAY AREA, press <F1> while the A=.... is diplayed. The cursor will move up into the memory dislay area.

While in the memory display area, use the cursor control keys, up arrow, down arrow, <- and ->, and <F1> to position the cursor to the value you want to change. The 128-byte block of memory is displayed in hexadecimal and ASCII format, and you can modify memory by entering hex values or ASCII values, depending on the position of the cursor.

To switch from hexadecimal to ASCII entry or vice versa, press the </> key.

When the cursor is in the hexadecimal area, enter hexadecimal values. The debugger will update the memory display as you type in each nibble (hexadecimal character, half a byte).

When the cursor is in the ASCII area, enter ASCII characters. Press </> to return to hexadecimal entry.

TO CANCEL ALL CHANGES in memory, press <ESC>. TO EFFECT ALL CHANGES, Press <F2>.

Modify Registers

Press <R> to enter this command. The R => prompt appears. Type in a letter indicating which register-pair you want to change:

A	for AF	B	for BC	D	for DE	H	for HL
X	for IX	Y	for IY				
F	for AF'	C	for BC'	E	for DE'	L	for HL'

The cursor will move over to the first byte of the register pair. While in the register modify mode, use the cursor control keys, <- and -> to move over one nibble at a time. Use <TAB> to advance to the next register pair.

TO CANCEL CHANGES in register contents, press <ESC>. TO EFFECT CHANGES made, press <F2>.

System

Press <S> to return to the TRSDOS READY mode. The debugger is still on; when you load and execute a program, you will enter the debugger

again.

Upload

Press <U> to enter the serial input mode, in which the Computer accepts serial input from another device (Model II or other computer, etc.).

Set up the sending device (e.g., another Model II or other computer) to transmit to the Model II via the serial interface, Channel B on back panel. Transmissions must be RS-232C standard, with the following characteristics:

1200 baud
8-bit words
No parity
1 stop bit between words.

The transmitting program must send the data in "Intel (R) Paper Tape Hex Format", described below.

Each byte of data is sent as a pair of hexadecimal ASCII-coded characters:

1) high nibble (most significant four bits), sent as first byte of pair.
2) low nibble (least significant four bits), sent as second byte of pair.

For example, the value X'F7' is sent as two bytes, "F" (X'46') followed by "7" (X'37').

Because only ":" and ASCII coded hexadecimal numbers are sent, data is always in the range <X'30',X'3A'> or <X'41',X'46'>. Values outside this range will terminate reception and produce an error message.

Record Format

Records must be sent as follows:

Character Number	Contents	Comments
1	":"	Sync-character to indicate beginning of record.
2	High nibble of record length (N)	This 2-byte sequence gives the number of byte PAIRS in this record. Zero means 256 byte pairs follow.
3	Low nibble of record length (N)	
4	High nibble of msb of load addr.	This 4-byte sequence gives address where the data is to start loading. Address specified must be in the user area <X'2800',TOP>.
5	Low nibble of msb of load addr.	
6	High nibble of lsb of load addr.	
7	Low nibble of lsb of load addr.	
8	High nibble of EOF (end of file) code	This byte-pair gives the EOF code. Any non-zero value means end of file (no more records follow). A value of zero means more records follow.
9	Low nibble of EOF code	
10	First byte of first data pair	First byte is ASCII code for first hex digit; second byte is ASCII code for second hex digit.
11	Second byte of first data pair	

.
.
.

(continued)

(Record Format, continued)

Character Number	Contents	Comments
8 + (N*2)	First byte of last data pair	Last pair of data characters
9 + (N*2)	Second byte of last data pair	
10 + (N*2)	First byte of data checksum (high nibble)	This pair represents 2's complement of the data (all byte pairs after the ":" up to but not including the checksum). Note that each byte pair is converted back to the original byte of data before it is summed.
11 + (N*2)	Second byte of data checksum (low nibble)	

Note: Only the data bytes (characters 10 through 9+(N*2)) are saved in memory.

Sample record:

Character Number	Sample Data ASCII	Hex Value
1	":"	3A
2	"0"	30
3	"2"	32
4	"2"	32
5	"8"	38
6	"0"	30
7	"0"	30
8	"0"	30
9	"0"	30
10	"3"	33
11	"7"	37
12	"7"	37
13	"0"	30
14	"5"	35
15	"D"	44

This record will contain 2 byte-pairs of data:
"3" "7" representing the value X'37'
"7" "0" representing the value X'70'
and will start loading at X'2800'. The one-byte sum of the original bytes (represented in pairs by characters 2 through 13) is X'A3'. The 2's complement of X'A3' IS X'5D'--which is represented in bytes 14 and 15.

Notes on Using the Upload Function

Because of the baud rate and the absence of complex protocols for error handling, re-transmissions, etc., the Upload function is intended for hard-wired machines, e.g., from a development machine to the Model II in the immediate vicinity.

(M2DOS2B 8/6/79)

DIR
List the Diskette Directory

```
:-------------------------------------------------------------:
: DIR :d {SYS, PRT}                                           :
:     :d is a drive specification. (The colon : before d      :
:        is optional.) If :d is omitted, drive zero           :
:        is used.                                             :
:     SYS tells TRSDOS to list system and user files          :
:        If SYS is omitted, only user files are listed.       :
:     PRT tells TRSDOS to list the directory to the           :
:        Printer. If PRT is omitted, TRSDOS lists the         :
:        directory on the Console Display.                    :
:-------------------------------------------------------------:
```

This command gives you information about a diskette and the files it contains.

To pause the listing, press <HOLD>. To continue, press <HOLD> again. To terminate the listing, press <ESC>.

Examples

DIR
Displays the directory of user files in drive zero.

DIR 1 PRT
Lists to the Printer the directory of user files in drive 1.

DIR {SYS,PRT}
Lists to the Printer the directory of system and user files. The braces are required to prevent TRSDOS from taking SYS as an invalid drive specification.

Sample Directory Listing

```
DISK NAME:TRSDOS                DRIVE:3          08/06/79          00.48.33
 FILE NAME     CREATED      ATTRB  FILE REC  NMBR    NMBR      SPACE      EOF
               MM DD YY            TYPE LEN  RECS    EXTS   ALLOC USED  BYTE
 RAMDOM         7 31 79     D*A7     F    1   167       1       5    1   166
 DEMO           7 31 79     D*X0     F    1   337       1       5    2    80
 TEST           7 31 79     D*X0     F    1   210       1       5    1   209
 TEST/OLD       7 31 79     D*X0     F    1   204       1       5    1   203
 DISPLAY/DBG    7 31 79     P*X0     F  256     1       1       5    1     0
 DBG            7 31 79     P*X0     F  256    17       1      20   17     0
***  309 FREE GRANULES IN    2 EXTENTS ***
```

What the column headings mean

(1) Disk Name--the name assigned to the diskette when it was formatted.

(2) File Name--the name and extension assigned to a file when it was created. The password (if any) is not shown.

(3) Creation Date--when the file was created.

(4) Attributes--a four-character field.
  The first character is either P for Program file or
    D for Data file.
  The second character is either S for System file or
    * for User file.
  The third character gives the password protection status.
    X The file is unprotected (no passwords).
    A The file has an access word but no
       update word.
    U The file has an update word but no access

word.
B The file has both update and access words.
The fourth character specifies the level of access assigned to the access word:
0,1 Kill file and everything listed below.
2 Rename file and everything listed below.
3 Not used
4 Write and everyting listed below.
5 Read and everything listed below.
6 Execute only.
7 None.

(5) File Type--Indicates the record type for the file.
F Fixed-length records fixed-length records
V Variable-length records

(6) Record Length--Assigned when the file was created (applies to fixed-length record files only).

(7) Number of Records--how many logical records have been written. Asterisks signify none have been written or file has variable length records and number written cannot be calculated.

(8) Number of Extents--How many segments (contiguous blocks of up to 32 granules) of disk space are allocated to the file. Asterisks signify none have been allocated.

(9) Space Allocated--How many sectors (256 byte blocks) are allocated to the file. Asterisks signify none have been allocated.

(10) Space Used--How many of these sectors have actually been written. Asterisks signify none have been allocated.

(11) End of File (EOF) Byte--Shows the starting position in a sector of the last record written.

(12) Free Space Remaining--tells how many granules (1280-byte blocks) are free for storing new information. Also tells how the free space is organized; i.e., how many contiguous blocks (extents) make up the free space.

DO
Begin Auto Command Input from Disk File

```
:-------------------------------------------------------------------:
:  DO  file                                                         :
:     file specifies a file created with the BUILD command          :
:-------------------------------------------------------------------:
```

This command reads and executes the lines stored in a special-format file created with the BUILD command. The System executes the commands just as if they had been typed in from the Keyboard, except that they are not echoed to the Video Display (except for PAUSE).

Command lines in a BUILD file may include library commands or file specifications for user programs.

When DO reaches the end of the automatic command input file, it returns control to TRSDOS.

The commands DO and DEBUG cannot be included in an automatic command input file.

Special Notes for Running BASIC Automatically

You can include a command to run BASIC in the DO file. For example, the line:
BASIC PROGRAM
Tells TRSDOS to load and execute BASIC. BASIC in turn will load and run PROGRAM. While the BASIC program is running, the keyboard will operate normally, with one exception: Pressing <BREAK> at any time terminates automatic command input and returns you to TRSDOS READY.

To resume automatic command input, a BASIC program must return to TRSDOS READY via the SYSTEM command. If the BASIC program simply ends and returns to the BASIC command mode, the keyboard will function normally EXCEPT Pressing <BREAK> will automatically return you to TRSDOS READY.

General Notes for Automatic Execution of User Programs

While DO is executing, a user program cannot set up a <BREAK>-key processing program. (See SETBRK in "Technical Informtion".) Inside the user program, the keyboard will function normally. When the program ends and returns to TRSDOS, automatic command input will resume.

Examples

```
        DO STARTER
```

TRSDOS will begin automatic command input from STARTER,after the operator answers the Date and Time prompts.

```
        AUTO DO STARTER
```

Whenever you start TRSDOS, it will begin automatic command input from STARTER.

Sample Use

Suppose you want to set up the following TRSDOS functions automatically on start-up:

```
        FORMS W=80
        CLOCK ON
        VERIFY OFF
```

THEN use BUILD to create such a file. If you called it BEGIN, then use the command:

```
        AUTO DO BEGIN
```

to perform the commands each time TRSDOS starts up.

DUMP
Store a Program into a Disk File

```
:-------------------------------------------------------------------:
: DUMP file {START=address-1, END=address-2, TRA=                   :
:           address-3, RELO=address-4, RORT=letter}                 :
:   file is a file specification.                                   :
:   START=address-1 specifies the start address of                  :
:     the memory block.                                             :
:   END=address-2 specifies the end address of the                  :
:     memory block.                                                 :
:   TRA=address-3 specifies the transfer address, where             :
:     execution starts when the program is loaded. If               :
:     omitted, address-4 is used.                                   :
:   RELO=address-4 specifies the start address for loading          :
:     the program back into memory. If omitted, address-1           :
:     is used.                                                      :
:   RORT=letter specifies whether whether the program is            :
:     directly executable from TRSDOS. RORT stands for              :
:     "Return OR Transfer". If RORT=R, then TRSDOS can              :
:     load but not execute file. If RORT=T, then TRSDOS             :
:     can load and execute file from the TRSDOS READY               :
:     mode. If RORT is omitted, RORT=T is used.                     :
:   Note: Addresses must be in hexadecimal form, without            :
:   the X'  ' notation.                                             :
:-------------------------------------------------------------------:
```

This command copies a machine-language program from memory into a program file. You can then load and execute the program at any time by entering the file name in the TRSDOS READY mode.

You can enter machine language programs directly into memory, via the DEBUG command.

Examples

DUMP LISTER/CMD START=7000, END=7100, TRA=7004
Creates a program file named LISTER/CMD containing the program in memory locations X'7000' to X'7100'. When loaded, LISTER/CMD will occupy the same addresses, and TRSDOS will protect memory beginning at X'7000'. The program is executable from the TRSDOS READY level.

DUMP PROG2/CMD START=6000, END=6F00, TRA=3010, RELO=3000
Creates a program file named PROG2/CMD containing the program in addresses X'6000' to X'6F00'. When loaded, PROG2/CMD will reside from X'3000' to X'3F00. Execution will start at X'3010'. The program is executable from TRSDOS READY.

DUMP ROUTINE/1 {START=6800,END=7000,RORT=R}
Creates a program file which cannot be executed from the TRSDOS READY level. Typically, this would be a routine to be called by another program.

(M2DOS3 8/6/79)

ERROR
Display Error Message

```
:-------------------------------------------------------------------:
: ERROR  number                                                     :
:   number is a decimal number for a TRSDOS error code             :
:-------------------------------------------------------------------:
```

This command displays a descriptive error message. When TRSDOS gives you a reverse (black-on-white) message like:
* * ERROR 47 * *
You type back
ERROR 47
to see the full error message.

Example

ERROR 3
Gives you the message
PARAMETER ERROR ON CALL

FORMS
Set Printer Parameters

```
:-------------------------------------------------------------:
: FORMS {P = page size, L = lines, W = width, C = control}    :
: FORMS {T}                                                   :
:    P = page size tells TRSDOS the total number of lines     :
:          per page. If omitted, 66 is used.                  :
:    L = lines tells TRSDOS the maximum number of lines       :
:          to print before an automatic form feed. If         :
:          omitted, 60 is used. lines cannot be greater       :
:          than page size.                                    :
:    W = width tells TRSDOS the maximum number of             :
:          characters per line. If omitted, 132 is used.      :
:    C = control tells TRSDOS to initialize the Printer       :
:          by sending it the specified code.                  :
:          The code can be hexadecimal value in the           :
:          range <0,FF>. Do not use the X' ' notation.        :
:    FORMS T is a special version of the command, telling     :
:          TRSDOS to advance printer to top of form.          :
:          Printer must have been previously initialized      :
:          and must be ready. When T is given in the          :
:          option list, any other keywords in the option      :
:          list are ignored.                                  :
:-------------------------------------------------------------:
```

This command lets you set up the TRSDOS Printer software to suit the Printer you have attached. If the Printer was on-line when you started TRSDOS, and the default parameters P = 66, L = 60, W = 132, and C = 0 are appropriate, then you do not need to use this command.

In addition to setting parameters, FORMS verifies that the Printer is on-line, and it lets you adjust paper to the top of form.

Examples

FORMS
Resets all parameters to their default values.

FORMS L=56
Resets the maximum number of printed lines per page to 56, leaving 10 lines blank on each page.

FORMS C=14
Sends the initialization code X'14' to the Printer.

FORMS T
Advances Printer to top of form. Useful when you have done some printing and want to start next printing at top of form.

Setting the Parameters

Page Size. Multiply your form length in inches by the number of printed lines per inch to get the appropriate value. Most Printers print 6 lines per inch. Therefore standard 11-inch forms have a page size of 66 lines. That's why the default is PAGE=66.

Lines per page. This number determines the number of blank lines on each page. If you set lines equal to page size, then TRSDOS will print every line on the page. If you set lines equal to page size minus 6, then TRSDOS will leave 6 blank lines on each page. Lines per page cannot exceed page size.

Width. This number sets the maximum number of characters per line. If a print line exceeds this width, TRSDOS will automatically break the line at the maximum length and continue it at the beginning of the next Print line.

Control Codes. Some Printers require an initialization code (for example, to set up for double-size characters). The code you specify is sent to the Printer during execution of the FORMS command.

FREE
Display Disk Allocation Map

FREE :d {PRT} :d is a drive specification. (The colon : before d is optional.) If :d is omitted, drive 0 is used. PRT tells TRSDOS to send the map to the Printer If PRT is omitted, TRSDOS sends the map to the Console Display.

This command gives you a map of granule allocation on a diskette. (A granule, 1280 bytes, is the unit of space allocation.) This information is useful when you want to optimize file access time.

When a diskette has been used extensively (file updates, files killed, extended, etc.), files often become segmented (dispersed or fragmented). This slows the access time, since the disk read/write mechanism must move back and forth across the diskette to read or write to a file.

FREE helps you determine just how segmented a diskette is. If you decide that you'd like to re-organize a particular file to allow faster access, you can then COPY it onto a relatively "clean" diskette.

Example

FREE
Displays a free space map of the diskette in drive 0.

FREE {PRT}
List the free space map for drive 0 to the Printer. The braces are required so TRSDOS won't take PRT as an invalid drive specification.

FREE :2 PRT
Lists the drive 2 map to the Printer.

A Typical FREE Display

Four special symbols are used in the FREE map:

.	Unused Granule
D	Directory Information
X	Allocated Granule
F	Track is Flawed (Unusable)

Here's a typical display:

```
                              F R E E   S P A C E   M A P
TRK #     TRSDOS ------------------------------------------------- DRIVE:0
01-04:    X X X X X  |  X X X X X  |  X X X X X  |  X X X X X
05-08:    X X X X X  |  X X X X X  |  X X X . .  |  . . . . .
09-12:    . . . . .  |  . . . . .  |  . . . . .  |  . . . . .
13-16:    . . . . .  |  . . . . .  |  . . . . .  |  . . . . .
17-20:    . . . . .  |  . . . . .  |  . . . . .  |  . . . . .
21-24:    . . . . .  |  . . . . .  |  . . . . .  |  . . . . .
25-28:    . . . . .  |  . . . . .  |  . . . . .  |  . . . . .
29-32:    . . . . .  |  . . . . .  |  . . . . .  |  . . . . .
33-36:    . . . . .  |  . . . . .  |  . . . . .  |  . . . . .
37-40:    . . . . .  |  . . . . .  |  . . . . .  |  . . . . .
41-44:    . . . . .  |  . . . . .  |  X X X X X  |  D D D D D
45-48:    X X X X X  |  X X X X X  |  X X X X X  |  X X X X X
49-52:    . . . . .  |  . . . . .  |  . . . . .  |  . . . . .
53-56:    . . . . .  |  . . . . .  |  . . . . .  |  . . . . .
57-60:    . . . . .  |  . . . . .  |  . . . . .  |  . . . . .
61-64:    . . . . .  |  . . . . .  |  . . . . .  |  . . . . .
65-68:    . . . . .  |  . . . . .  |  . . . . .  |  . . . . .
69-72:    . . . . .  |  . . . . .  |  . . . . .  |  . . . . .
73-76:    . . . . .  |  . . . . .  |  . . . . .  |  . . . . .
```

I
Swap Diskettes

I

Immediately after you change diskettes in any drive, enter this command so TRSDOS will be able to perform important "bookkeeping" tasks.

Example

I
tells the System you have changed one of the diskettes.

KILL
Delete a File

```
:-------------------------------------------------------------------------:
: KILL file                                                               :
:      file is a file specification                                       :
:-------------------------------------------------------------------------:
```

This command deletes a file from the directory and frees the space allocated to that file. If no drive is specified, TRSDOS will search for the file, starting with drive 0. Before deleting the file, TRSDOS will display the file name and the drive that contains the file. Type Y <ENTER> to Kill the file, N <ENTER> to cancel the command.

DO NOT KILL AN OPEN FILE.

Examples

KILL TESTPROG/BAS
Deletes the named file from the first drive that contains it.

KILL JOBFILE/IDY.foggy
Deletes the named file from the first drive that contains it. The file is protected with the password foggy.

KILL FORM/123:3
Deletes FORM/123 from drive 3.

Sample Uses

When updating a file, it is a good practice to input from the old file and output updated information to a new file. That way, if the update is wrong, you still have the old file as a backup. When you have verified that the update file is correct, you can Kill the old file.

KILL is also useful in conjunction with pre-allocated files. Suppose you have finished writing to a pre-allocated file, and one or more granules are unused in the pre-allocated file. Then you can copy the pre-allocated file to a dynamically allocated file, and afterwards Kill the pre-allocated file. This is the only way to reduce the size of a pre-allocated file.

LIB
Display Library Commands

LIB

This command lists to the Display all the Library Commands.

Example

LIB

LIST
List Contents of a File

```
: LIST file (PRT, SLOW, R=record-number, A)                    :
:   file is a file specification                               :
:   PRT tells TRSDOS to list to the Printer. If PRT is         :
:     omitted, the Console Display is used.                    :
:   SLOW tells  TRSDOS to pause briefly after each             :
:       record. If omitted, the listing is continuous.         :
:   R=record-number tells TRSDOS the starting record for       :
:       the listing. record-number must be in the range        :
:       <1;65535>. If omitted, record 1 is used.               :
:   A  tells TRSDOS to list ASCII characters only (no          :
:       hexadecimal values). If omitted, ASCII and             :
:       equivalent hexadecimal values are listed.              :
```

This routine lists the contents of a file. The listing shows both the hexadecimal contents and the ASCII characters corresponding to each value. For values outside the range <X'20',X'7E'>, a period is displayed.

To stop the listing, press HOLD. Press HOLD again to continue. Press <ESC> or <BREAK> to terminate the listing.

Examples

LIST DATA/BAS
Lists the contents of DATA/BAS.

LIST TEXTFILE/1 SLOW
Lists the contents of TEXTFILE/1, pausing after each record.

LIST TEXTFILE/1 R=100, A
The listing starts with the 100th record in TEXTFILE/1. Only ASCII characters are displayed.

LIST PROGRAM/CMD PRT
Lists the file PROGRAM/CMD to the Printer.

Listing Format

LIST numbers each record as it is listed, and prints a heading showing the relative position of each byte in the record. Here's a sample listing after the command:
LIST VARFILE PRT

```
VARFILE                          TYPE=V          WED JUL 04 1979 185 -- 17.46.08          PAGE  1

          BYTE 1...5...10...15...20...25...30...35...40...45...50...55...60...65...70...75...80...85...90...95..100

R= 1        1 ↑!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
LRL= 91       3333333333333333333333333333333333333333333333333333444444444444444444444444444444333333333333333
              B!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
              ----------

R= 2        1 .!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
LRL= 143      833333333333333333333333333333333333333333333344444444444444444444444444444433333333333333333333333
              F!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

            101 456555666667777788888999999AAAAABBBBBCCCCCDD
                333333333333333333333333344444444444444444
                456555666667777788888999991111122222333334

R= 3        1 .!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
LRL= 155      933333333333333333333333333333333333333333333344444444444444444444444444444433333333333333333333333
              B!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

            101 456555666667777788888999999AAAAABBBBBCCCCCDDDDDEEEEEFFFF
                333333333333333333333333333444444444444444444444444444
                456555666667777788888999991111122222333334444455555666
                ----------

R= 4        1 9!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
LRL= 57       3333333333333333333333333333333333333333333333344444444444
              9!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
              ----------

R= 5        1 .!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
LRL= 148      933333333333333333333333333333333333333333333344444444444444444444444444444433333333333333333333333
              41!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

            101 456555666667777788888999999AAAAABBBBBCCCCCDDDDDEE
                333333333333333333333333344444444444444444444
                456555666667777788888999991111122222333334444455
                ----------

R= 6        1 .!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
LRL= 144      933333333333333333333333333333333333333333333344444444444444444444444444444433333333333333333333333
              01!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

            101 456555666667777788888999999AAAAABBBBBCCCCCDDD
                333333333333333333333333344444444444444444
                456555666667777788888999991111122222333334
                ----------

R= 7        1 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
LRL= 124      733333333333333333333333333333333333333333333344444444444444444444444444444433333333333333333333333
              C!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

            101 456555666667777788888999
                333333333333333333333333
                456555666667777788888999
                ----------

R= 8        1 .!"#$%&'()*+,-./0123456789
LRL= 26       133333333333333333333333
              A!"#$%&'()*+,-./0123456789
              ----------

          BYTE 1...5...10...15...20...25...30...35...40...45...50...55...60...65...70...75...80...85...90...95..100
```

Here's a sample listing after the command:
LIST TEST PRT,A

```
PROG/TXT                              TYPE=F              MON AUG 06 1979 218 -- 01.16.36          PAGE  1

          BYTE 1...5...10...15...20...25...30...35...40...45...50...55...60...65...70...75...80...85...90...95..100

R= 1        1 1140 IF TIME$ = 1:00 PRINT "Time is 10:15 A.M.--time to pick up the mail."      :END.1150 PRINT "THI
LRL= 1
          101 S IS A TEST".1160 READ A,B,C.1170 DATA 3.14159265358979623, 333.00000303000322211, 3.3D9.
```

LOAD
Load a Program File

```
:-------------------------------------------------------------------:
: LOAD file                                                         :
:      file is a file specification for a file created              :
:        by the DUMP command.                                       :
:-------------------------------------------------------------------:
```

This command loads into memory a machine-language program file. After the file is loaded, TRSDOS returns to the TRSDOS READY mode.

You cannot use this command to load a BASIC program or any file created by BASIC. See the BASIC Reference Manual for instructions on loading BASIC programs.

Example

LOAD PAYROLL/pt1
Loads the file PAYROLL/pt1.

Sample Use

Often several program modules must be loaded into memory for use by a master program. For example, suppose PAYROLL/pt1 and PAYROLL/pt2 are modules, and MENU is the master program. Then you could use the commands:
LOAD PAYROLL/pt1
LOAD PAYROLL/pt2
to get the modules into memory, and then type:
MENU
to load and execute MENU.

If PAYROLL/pt1 and PAYROLL/pt2 were DUMPed with RORT=R, then you can load by typing the file name without the LOAD command, i.e.,
PAYROLL/pt1
PAYROLL/pt2
After each is loaded, TRSDOS READY returns.

PAUSE
Pause Execution for Operator Action

```
:-------------------------------------------------------------:
:  PAUSE   prompting message                                  :
:     prompting message is an optional message to be          :
:       displayed during the pause.                           :
:-------------------------------------------------------------:
```

This command is intended for use inside a DO file. It causes TRSDOS to print a message and then wait for the operator to press <ENTER>.

Example

PAUSE Insert Disk #21
Prints PAUSE followed by the message and prompts the operator to press <ENTER> to continue.

PAUSE
Prints PAUSE and prompts the operator to press <ENTER> to continue.

See BUILD and DO for sample uses.

PURGE
Delete Files

PURGE :d (file-class)
:d is drive specification. The colon : is optional. If :d is omitted, drive 0 is used.
file-class is one and only one of the following:
SYS — System files (program and data)
PROG — User program files
DATA — User data files
ALL — All files, user and system

This command allows quick deletion of files from a particular diskette. To use PURGE, you must know the diskette's master password. (TRSDOS System diskettes are supplied with the password PASSWORD.)

All System files are required for TRSDOS to function. Do not eliminate System files if you want to use the diskette in drive 0.

When the command is entered, TRSDOS will ask for the diskette's password. Type in up to 8 characters, and press <ENTER> if you typed fewer than 8 characters. The System will then display user file names one at a time, prompting you to Kill or leave each file.

Example

PURGE :1
TRSDOS will let you purge files from drive 1.

PURGE
TRSDOS will let you purge files from drive 0.

PROT
Use Diskette's Master Password

```
:-------------------------------------------------------------:
: PROT  :d  (OLD=password,options)                            :
:   :d is a drive specification. The colon : is optional.     :
:    options include any of the following:                    :
:      OLD=password specifies the diskette's current          :
:          password. This is required.                        :
:      options include the following:                         :
:      PW        Tells TRSDOS to change the master password.:
:                If omitted, the master password is left      :
:                unchanged.                                   :
:      NEW=password   Required after PW, gives TRSDOS the     :
:                new password (up to 8 characters)            :
:      LOCK      Tells TRSDOS to protect all user files       :
:                with the latest password. Update and         :
:                access words will both be set to the         :
:                master password.                             :
:      UNLOCK    Tells TRSDOS to remove passwords from all    :
:                user files.                                  :
:    If LOCK and UNLOCK are omitted, user file protection     :
:    is left unchanged. If one is used, the other must be     :
:    omitted.                                                 :
:-------------------------------------------------------------:
```

PROT changes file protection on a large scale. If you know the diskette's master password, you can change it. You can also protect or un-protect all user files.

A diskette's master password is initially assigned during the format or backup process. The TRSDOS diskette is supplied with the master password PASSWORD.

Examples

PROT :1 (OLD=PASSWORD, PW, NEW=H2O)
Tells TRSDOS to change the master password of the drive 1 diskette from PASSWORD to H2O.

PROT :0 (OLD=H2O, UNLOCK)
Tells TRSDOS to remove passwords from every user file on the drive 0 diskette (must have the password H2O).

PROT :0 (OLD=H2O, PW, NEW=ELEPHANT, LOCK)
Tells TRSDOS to change the master password from H2O to ELEPHANT and assign the new one to every user file.

RENAME
Rename a File

```
:-------------------------------------------------------------------:
: Rename file-1 TO file-2                                           :
:    file-1 and file-2 are file specifications                      :
:      If file-2 includes a drive specification or                  :
:      password, it will be ignored, since the file will            :
:      remain on the same drive and will retain its former          :
:      password, if any.                                            :
:    ØTOØ is a delimiter. A comma or space may also be              :
:      used.                                                        :
:-------------------------------------------------------------------:
```

This command renames a file. Only the name/extension is changed; the data in the file and its physical location on the diskette are unaffected.

RENAME cannot be used to change a file's password. Use ATTRIB to do that.

Examples

RENAME Miss/BAS TO Ms/BAS
TRSDOS will search for Miss/BAS starting with drive Ø, and will rename it to Ms/BAS.

RENAME REPORT/AUG:3, REPORT/SEP
Renames REPORT/AUG on drive 3 to REPORT/SEP.

RENAME MASTER.12345678 TO MASTER/A
Searches for MASTER and renames it to MASTER/A. The password 12345678 must grant at least RENAME access (see Passwords in chapter 1). The renamed file has the same password.

SETCOM
Set Up RS-232C Communications

SETCOM { A=(baud rate, word length, parity, stop bits),
B=(baud rate, word length, parity, stop bits) }

A=(options) tells TRSDOS to initialize channel A.
To turn channel A off, use A = OFF instead of
A = (options)
If A = (options) is omitted, status of channel A
is unchanged.
B=(options) tells TRSDOS to initialize channel B.
To turn channel B off, use B = OFF instead of
B = (options)
If B = (options) is omitted, status of channel B
is unchanged.
The options tell TRSDOS what RS-232C parameters
to use. The following parameters are available:

baud rate	100, 150, 300, 600, 1200, 2400, 4800 If not specified, 300 is used.
word length	5, 6, 7, 8 If not specified, 7 is used.
parity	E for even, O for odd, N for none If not specified, even is used.
stop bits	1, 2 If not specified, 1 is used.

Every option but the last must be followed by a comma.
The options are positional, e.g., the third item in an
option list must always specify parity. To use a default
value, omit the option. If you want to list subsequent
options, you must include a comma for each default.

This command initializes RS-232C communications via channels A and B on the back panel. Before executing it, you should connect the communications device (modem, etc.) to the Model II.

See the Model II Operation Manual for a description of RS-232C signals used in channels A and B. For hard-wired connection from one Model II to another, see the wiring diagram in Technical Information, RS232C supervisor call.

SETCOM uses the Special Programming Area above TOP (see Memory Requirements). To use the serial I/O channels from BASIC you must execute SETCOM BEFORE starting BASIC.

Once you initialize a channel, you can begin sending and receiving data, using four system routines that are set up during initialization:

ARCV	Channel A receive, function code 96
ATX	Channel A transmit, function code 97

BRCV Channel B receive, function code 98
BTX Channel B transmit, function code 99

These system routines are only available when the respective channel has been initialized. See Technical Information for details.

Examples

SETCOM A=( )
Sets up channel A for serial communications, using all the default parameters. System function calls 96 and 97 are available for serial I/O. The status of channel B is unchanged.

SETCOM B=(4800, 8, , 2), A=OFF
Sets up channel B:

baud rate	4800
word length	8 bits
parity	Even (default)
stop bits	2

and turns off channel A.

SETCOM A=(2400, 8, O), B=(, , , 2)
Sets up channels A and B:

	Channel A	Channel B
baud rate	2400	300 (default)
word length	8	7 (default)
parity	Odd	Even (default)
stop bits	1 (default)	2

TIME
Reset or Get the Time

TIME hh.mm.ss
hh is a two-digit hour specification.
mm is a two-digit minute specification.
ss is a two-digit second specification.
.ss is optional; if omitted, .00 is used.
If hh.mm.ss is given, TRSDOS resets the time.
If hh.mm.ss is not given, TRSDOS displays the current time and date.

This command lets you reset the time or display the date and time.

The operator can set the time initially when TRSDOS is started up. After that, TRSDOS updates the time and date automatically, using its built-in clock and calendar.

When you request the time, TRSDOS displays it in this format:

THU JUL 19 1979 200 -- 14.15.31
for Thursday, July 19, 1979, the 200th day of the year, 2:15:31 PM.

Note: If the time passes 23.59.59, TRSDOS does not start over at 00.00.00. Instead, it continues with 24.00.00. However, the next time you use the TIME or DATE command, the time will be converted to its correct 24-hour value, and the date will be updated. If the clock is allowed to run past 59.59.59, it will re-cycle to zero, and the date will not be updated to include the 60-hour period.

Examples

TIME
Displays the current date and time.

TIME 13.20.00
Resets the time to 1:20:00 PM.

TIME . 18.24
Resets the time to 6:24:00 PM.

Note: Periods are used instead of the customary colons since periods are easier to type in--you don't have to press SHIFT.

VERIFY
Automatic Read After Write

```
:-------------------------------------------------------------------:
: VERIFY {switch}                                                   :
:    switch is one of the following:                                :
:       ON     Turn on the verify function.                        :
:       OFF    Turn off the verify function.                       :
:    If switch is omitted, the current status is displayed          :
:       and is left unchanged.                                      :
:-------------------------------------------------------------------:
```

This command controls the verify function. When it is on, TRSDOS will read after each write operation, to verify that the data is readable. If the data is not readable after retries, TRSDOS will return an error message, so you'll know that the operation was not successful.

Note: TRSDOS always verifies directory writes. User writes (writing data into a file) are only verified when VERIFY is ON.

TRSDOS starts up with VERIFY ON. For most applications, you should leave it ON.

Examples

VERIFY ON
Turns on the verify function.

VERIFY OFF
Turns off the verify function.

VERIFY
Displays the status of the verify switch.

MODEL II TRSDOS

3 / UTILITY PROGRAMS

(M2DOS4 8/6/79)

3 / Utility Programs

TRSDOS includes two utility programs, BACKUP and FORMAT.

Before any disk can be used to store information, it must be formatted. Use FORMAT to prepare a new (blank) diskette, or the "start fresh" with a previously formatted diskette.

Use BACKUP to copy all the information on a diskette onto another diskette. This gives you safe copies of important data and programs. BE SURE TO MAKE A BACKUP COPY OF YOUR SYSTEM DISK--before you begin using the System.

Both utilities can be used in any Model II System--single or multiple drive. Both use all available memory, but do not overlay the resident System. Initialization values (time, date, printer parameters, etc.) are intact upon return from either utility.

BACKUP
Duplicate a Diskette

BACKUP

This program duplicates the data from a diskette onto another formatted diskette, by copying all allocated granules.

To execute BACKUP, type
BACKUP

BACKUP provides all necessary prompting for input. The destination diskette must be formatted; if it contains data, BACKUP will warn you and ask if you want to write over the data.

Prompting Messages

SOURCE DRIVE?
Type in the number of the drive which will contain the source diskette (diskette to be duplicated). This can be any drive 0 through 3.

SOURCE DISK PASSWORD?
Type in the password, up to 8 numbers or letters, and press ENTER if you typed fewer than 8.

DESTINATION DRIVE?
Type in the number of the drive which will contain the destination diskette. This can be any drive 0 through 3.

DO YOU WANT TO CHANGE DISK INFORMATION?
Type Y if you want to change the master password, diskette name, or date. BACKUP will prompt you to enter the new information (If you don't want to change any of these, type N.):

To change the password, type in up to 8 numbers and letters, and press ENTER if you typed fewer than 8.

To change the diskette name, type in up to 8 numbers and letters and press ENTER if you typed fewer than 8.

To change the date, obey the prompts.

cESTINATION DISK READY? Insert the destination diskette into the destination drive and type Y.

Note: For single-drive backups, you will need to swap source and destination diskettes when prompted by the messages:
SOURCE DISK READY? and DESTINATION DISK READY?

If there is a flaw on one of the destination tracks, BACKUP will terminate and return to TRSDOS without completing the duplication. The flawed diskette should be re-formatted and used as a data diskette (multi-drive users only).

BACKUP also does a consistency check of the directory track, comparing individual file entries with a separate table of space allocation. If any inconsistencies are found, BACKUP lists the affected files and prompts you to choose one of the following options:

<1> Copy only those files whose allocation information is consistent.
<2> Copy all files, regardless of inconsistencies.
<3> Abort the backup procedure.

Files with inconsistencies may or may not be usable.

FORMAT
Organize a Diskette

```
: FORMAT :d {ID=disk name, PW=password, option}   <- Lower case Required
:    :d specifies the drive to be used. The colon : is
:       optional. d can be specify any drive 0 through 3.
:    ID=disk name tells TRSDOS what to assign.
:       disk name can be up to 8 numbers and letters with
:         no embedded blank spaces.
:    PW=password tells TRSDOS what password to assign.
:       password can be up to 8 numbers or letters with
:         no embedded blank spaces.
:    option tells TRSDOS how much verifying to do:
:       FULL     Thorough check for flaws.
:       QUICK    Quick check for flaws.
:       NONE     No checking for flaws.
```

This program organizes a diskette into tracks and sectors. The diskette may be either blank (new or bulk-erased) or previously formatted. If it contains data, FORMAT will warn you and ask if you want to continue. If you continue, the data will be lost. In general, it's a good practice to bulk erase a diskette before formatting it.

Format also does a specified amount of checking for areas on the disk which cannot store data due to flaws in the recording surface. If it finds a flawed area, TRSDOS "locks out" the affected track and will never try to write to that track.

Examples

FORMAT :1 (ID=ACCOUNTS,PW=mouse)
Drive 1 will be used, and the disk will be given the name ACCOUNTS and master password mouse. TRSDOS will do the full test for flaws.

FORMAT :0 (ID=TESTDISK,PW=PASSWORD,QUICK)
Drive 0 will be used, and the disk will be given the name TESTDISK and master password PASSWORD. FORMAT will do a quick for flaws.

FORMAT :3 (ID=Jack, PW=12345678, NONE)
Drive 3 will be used, disk name Jack, password 12345678, with no checking for flaws.

When to FORMAT

To prepare a new diskette.
Before you can use a new diskette, you must format it. After formatting, record the disk name, date of creation and password in a safe place. This will help you estimate how long a diskette has been in use, and prevent your forgetting the master password. (For uses

this application, always use the FULL verify option.

To erase all data from a diskette.
To "start over" with a diskette, you can format it. All data will be lost. For this application, the QUICK verify option is probably adequate--unless you have had problems disk input/output errors with the diskette. (See below.)

To lock out flawed areas.
After long use, flaws may develop on a diskette. Reformat the diskette to lock out these tracks while leaving the good tracks available for data storage. Use the FULL verify option for this application.

MODEL II TRSDOS

4 / TECHNICAL INFORMATION

(M2DOS5 8/10/79)

5 / Technical Information

CONTENTS

0. Introduction

This chapter gives a practical description of TRSDOS on a technical level. You do not need it to use the Operator Commands, nor do you need it to run BASIC applications programs on the Computer. You DO need it to write assembly programs which use System routines. You may also find the information incidentally useful in programming with BASIC.

1. Diskette Organization

Model II uses single-sided, double-density diskettes. Each diskette contains 77 tracks, numbered 0-76.

Each track contains 26 sectors, numbered 1-26. Each sector contains 256 bytes, except for track 0 sectors, which contain 128 bytes. The total capacity of a diskette is:

(76 * 26 * 256) + (1 * 26 * 128) = 509,184 bytes.

Disk Space Available to User

Sector 26 of each track is reserved for system use, giving the user 25 sectors per track. On System diskettes, 65 tracks are available for the user; on non-System diskettes, 75 tracks are available.

Details:

Track 0 is reserved by the System. It is not accessible. Another track (usually track 44) is reserved by the System for the diskette directory. On Operating System diskettes, ten additional tracks are used for System files.

Unit of Allocation

The only unit of disk space allocation is the "granule". A TRSDOS granule is defined as 5 sectors. Therefore the smallest non-empty file consists of 5 sectors, i.e., one granule.

NON-SYSTEM DISKETTE	TRACKS	GRANULES	SECTORS	BYTES
1	75	375	1875	480,000
-	1	5	25	6400
-	-	1	5	1280
-	-	-	1	256

SPACE AVAILABLE TO USER

2. Disk Files

2.1 Methods of File Allocation

Model II provides two ways to allocate disk space for files: Dynamic Allocation and Pre-Allocation.

Dynamic Allocaton

With Dynamic Allocation, the System allocates granules only at the time of write. For example, when a file is first Opened for output, no space is allocated. The first space allocation is done at the first write. Additional space is added as required by subsequent writes.

With dynamically allocated files, unused granules are de-allocated (recovered) when the file is Closed.

Pre-Allocation

With pre-allocation, the file is allocated a specified number of granules when it is created. Pre-allocated files can only be created by the operator command CREATE.

TRSDOS will dynamically extend (enlarge) a pre-allocated file as needed for subsequent write operations. However, TRSDOS will not de-allocate unused granules when a pre-allocated file is Closed. The way to reduce the size of a pre-allocated file is to Copy it to a smaller pre-allocated file or to a dynamically allocated file and Kill the old file.

2.2 Record Length

The Model II transfers data to and from diskettes one sector at a time, i.e., in 256-byte blocks. These are the System's "physical" records.

User records or "logical" records are the buffers of data you wish to transfer to or from a file. These can be from 1 to 256 bytes long.

TRSDOS will automatically "block" your logical records into physical records which will be transferred to disk, and "deblock" the physical records into logical records which are used by your program. Therefore your ONLY concern during file access is with logical records. You never need to worry about physical records, sectors, tracks, etc. This is to your benefit, since physical record lengths and features may change in later TRSDOS versions, while the concept of logical records will not.

From this point on, the term "record" refers to a "logical record".

Spanning

If the record length is not an even divisor of 256, the records will automatically be spanned across sectors.

For example, if the record length is 200, Sectors 1 and 2 will look like this:

```
:-----------SECTOR 1-------:--------SECTOR 2-----------:
:                           :                          :
:<-record 1--> <--------record  2--------> <--------record 3-----
:< 200 bytes > < 56 bytes> < 144 bytes > < 112 bytes:>:
:                           :                          :
:---------------------------:--------------------------:
```

Sector 3 (not shown) contains the last 82 bytes of record 3.

Fixed-Length and Variable Length Records

Model II files can have either fixed-length or variable-length records. Files with fixed-length records will be referred to as FLRs; files with variable length records, VLRs.

Record length in an FLR file is set when the file is Opened for the first time. This length can be any value from 1 to 256 bytes. Once set, the record length in an FLR cannot be changed, unless the file is being over-written with new data.

Record length in a VLR file is specified in a one-byte length-field at the beginning of each record. The record-lengths in a VLR file can vary. For example, the first record in a file might have a

length of 32, the second, 17; the third, 250; etc.

The record-length byte indicates the entire length of the record, INCLUDING the length-byte. This can be any value from 0 to 255. A value of 1 can be used, but it has no meaning.

Examples:
A length-byte value of zero indicates that the record contains 255 bytes of data:

```
:------:--------------------: End
:  0   : 255 bytes of data  : of
:------:--------------------: Record
 LENGTH       DATA
  BYTE
```

A length-byte value of 2 indicates that the record contains 1 byte of data:

```
:------:--------------------: End
:  2   : one byte of data   : of
:------:--------------------: Record
 LENGTH       DATA
  BYTE
```

A length-byte value of 16 indicates that the record contains 15 bytes of data:

```
:------:--------------------: End
: 16   : 15 bytes of data   : of
:------:--------------------: Record
 LENGTH       DATA
  BYTE
```

2.3 Record Processing Capabilities

Model II TRSDOS allows both Direct and Sequential file access. Direct access--sometimes called "random access", but "direct" is more descriptive--allows you to process any record you specify.

> NOTE
> A file can contain up to 65535 records. Records are numbered from 0 (beginning of file) to 65534. A record number of 65535 indicates the end of file (EOF). These limits will be changed in a later release of TRSDOS.

Sequential access allows you to process records in sequence: Record N, N+1, N+2,.... With sequential access, you do not specify a record number; instead, the Operating System accesses the next record after the current one.

For files with Fixed Length Records (FLRs), you can position the current record pointer to the beginning of the file, end of file, or to any record in the file. In short, you can use Direct and/or Sequential Access with FLRs at any time during processing.

For files with variable length records (VLRs), you can only position the current record pointer to the beginning of the file or to the end of file. You cannot position to any other record in the file, since the position of interior VLRs cannot be calculated. If short, you can only use Sequential access with VLRs.

The Direct access routines are Direct-Read and Direct-Write; the Sequential access routines are Read-Next and Write-Next. Direct access routines always access the record you specify. Sequential access routines always access the record FOLLOWING the last record processed. (When the file is first opened, sequential processing starts with record 0.)

Examples

Assume you have a Fixed Length Record file currently Open. Here are some typical sequences you can accomplish via the file processing routines.

1.Read and/or write records in the file--in any order

This is done using Direct-Write and Direct-Read routines. You could read record 5, write at end of file, read record 3, write record 3, etc.

2. Sequential Read (or Write) beginning anywhere in the file.

First you would do a Direct-Read to the record where you want to start reading or writing. After that, you would do sequential reads or writes until done.

3. Sequential Write starting at end of file.

First do a Direct-Write to the end of file. Then do sequential writes until done.

4. Determine the number of records in a file.

First do a direct-read to end of file, then use the LOCATE routine to get the current record number, which now equals (number of records) + 1.

Examples with Variable Length Records

1. Sequential Write starting at end of file.

First do a Direct-Write to the end of file. Then do sequential writes until done.

2. Start reading or writing at first record

Open the file and start reading or writing sequentially until done.

Note: Writing to a VLR file AUTOMATICALLY resets the end-of-file to the last record you write. This means you cannot update a VLR file directly; you must read in the file and output the updated information to a new VLR file.

3. How to Use the Supervisor Calls

Supervisor Calls (SVC's) are Operating System routines available to any user program. The routines alter certain System functions and conditions; provide file access; perform I/O to the Keyboard, Video Display, and Printer; and perform various computations

All the SVCs leave memory above X'2FFF' untouched. Only those Z-80 registers used to pass parameters from the SVC are altered. All others are unaffected. However, all the Prime registers are used by the System; they are not restored.

Each SVC is assigned a Function Code. These codes run from 0 through 127. Only the first 96 are defined by the System; codes 96-127 are available for user definition.

To specify a given Supervisor Call, your program refers to the SVC's Function Code.

3.1 Calling Procedure

All SVCs are accomplished via the RST 8 instruction.

1. Load the Function Code for the desired SVC into the A register. Also load any other registers which are needed by the SVC, as detailed in Section 4.

2. Execute a RST 8 instruction

3. Upon return from the SVC, the Z flag will be set if the function was successful. If the Z flag is not set, there was an error. The A register contains the appropriate error code (except after certain computational SVC's, which use the A-register to return other information).

Examples

```
Time-Delay
        LD    BC,TIMCNT           ; LENGTH OF DELAY
        LD    A,6                 ; FUNCTION CODE 6 = DELAY-SVC
        RST   8                   ; JUMP TO SVC
;     DELAY OVER-PROGRAM CONTINUES HERE

Output a line to the Video Display
        LD    HL,MSG              ; POINT TO THE MESSAGE
        LD    B,10                ; B=CHARACTER COUNT
        LD    C,0DH               ; C=CTRL CHAR. TO ADD AT END
        LD    A,9                 ; CODE 9 = DISPLAY LINE-SVC
        RST   8                   ; JUMP TO SVC
        JR    NZ,GOTERR           ; JUMP IF I/O ERROR
;     IF NO ERROR THEN PROGRAM CONTINUES HERE
MSG     DEFM 'TEN BYTES'


Get a character from the Keyboard
GETCHAR LD    A,4                 ; CODE 4 = GET CHARACTER-SVC
        RST   8                   ; JUMP TO SVC
        JR    NZ,GETCHAR          ; DO AGAIN IF NO CHARACTER
;    CHARACTER IS IN REGISTER B
```

3.2 Error Codes and Messages

Register A usually contains a return code after any function call, with the Z flag set when no error occurred. Exceptions are certain computational routines, which use the A and F to pass back data and status information.

List of Error Codes and Messages

0	NO ERROR FOUND
1	BAD FUNCTION CODE ON SVC CALL OR NO FUNCTION EXISTS
2	CHARACTER NOT AVAILABLE
3	PARAMETER ERROR ON CALL
4	CRC ERROR DURING DISK I/O OPERATION
5	DISK SECTOR NOT FOUND
6	ATTEMPT TO OPEN A FILE WHICH HAS NOT BEEN CLOSED
7	DRIVE DOOR WAS OPENED WHILE FILE OPEN FOR WRITE
8	DISK DRIVE NOT READY
9	INVALID DATA PROVIDED BY CALLER
10	MAXIMUM OF 16 FILES MAY BE OPEN AT ONCE
11	FILE ALREADY IN DIRECTORY
12	NO DRIVE AVAILABLE FOR AN OPEN
13	WRITE ATTEMPT TO A READ ONLY FILE
14	WRITE FAULT ON DISK I/O
15	DISK IS WRITE PROTECTED
16	DCB IS MODIFIED AND IS UNUSABLE
17	DIRECTORY READ ERROR
18	DIRECTORY WRITE ERROR
19	IMPROPER FILE NAME (filespec)
20	FAD READ ERROR
21	FAD WRITE ERROR
22	FID READ ERROR
23	FID WRITE ERROR
24	FILE NOT FOUND
25	FILE ACCESS DENIED DUE TO PASSWORD PROTECTION
26	DIRECTORY SPACE FULL
27	DISK SPACE FULL
28	ATTEMPT TO READ PAST EOF
29	READ ATTEMPT OUTSIDE OF FILE LIMITS
30	NO MORE EXTENTS AVAILABLE (16 MAXIMUM)
31	PROGRAM NOT FOUND
32	UNKNOWN DRIVE NUMBER (filespec)
33	DISK SPACE ALLOCATION CANNOT BE MADE DUE TO FRAGMENTATION OF SPACE
34	ATTEMPT TO USE A NON PROGRAM FILE AS A PROGRAM
35	MEMORY FAULT DURING PROGRAM LOAD
36	PARAMETER FOR OPEN IS INCORRECT
37	OPEN ATTEMPT FOR A FILE ALREADY OPEN
38	I/O ATTEMPT TO AN UNOPEN FILE
39	ILLEGAL I/O ATTEMPT
40	SEEK ERROR
41	DATA LOST DURING DISK I/O (HARDWARE FAULT)
42	PRINTER NOT READY
43	PRINTER OUT OF PAPER
44	PRINTER FAULT (MAY BE TURNED OFF)
45	PRINTER NOT AVAILABLE
46	NOT APPLICABLE TO VLR TYPE FILES
47	REQUIRED COMMAND PARAMETER NOT FOUND
48	INCORRECT COMMAND PARAMETER
49	* * UNKNOWN ERROR CODE * *
50	* * UNKNOWN ERROR CODE * *

(M2DOS6 8/6/79)

4. Supervisor Calls

In this section we will use the following notation:

Notation	Meaning
RP = data	The register pair RP contains the data.
n1 < R < n2	The register R contains a value greater than n1 and less than n2
(RP) = data	The register pair RP contains the address of ("points to") the data.
NZ => Error	If Z Flag is not set, an error occurred.

4.1 System Control

Supervisor calls described in this section:

Function Code	Name	Purpose
0	INITIO	Initializes all I/O drivers
2	SETUSR	Sets up a user-defined SVC
3	SETBRK	Sets up <BREAK> key processing program
15	DISKID	Reads a diskette ID
25	TIMER	Set timer to interrupt a program
36	JP2DOS	Returns to TRSDOS (TRSDOS READY)
37	DOSCMD	Sends TRSDOS a command and then returns to TRSDOS READY
38	RETCMD	Sends TRSDOS a command and return to caller
39	ERROR	Displays "ERROR number"
52	ERRMSG	Returns Error Message to Buffer

INITIO ("Initialize I/O")--Function Code 0

This routine initializes all input/output drivers. It calls all of the other initialization routines. There are no parameters.

NOTE

This routine has been done already by the System. Users should never call it, except in extreme error conditions.

Entry Conditions

A = 0

Exit Conditions

NZ => Error
A = Error Code

SETUSR ("Set User") Function Code 2

This routine sets or removes a user vector. This gives you the ability to add SVC functions. Function codes 96-127 are available for user definition.

Once added, such a function can then be called via the RST 8 instruction, just like the System's SVC routines.

Your routine must reside above X'27FF', and should end with a RETurn instruction.

To change a previously defined function, you must first remove the old vector.

Entry Conditions

(HL) = Entry address of your routine (when C not 0)
B = Function code to be used, 95 < code < 128
C = Set/Reset code. If C=0, remove the vector. Otherwise, add the vector
A = 2

Exit Conditions

(HL) = Removed vector address (when C=0 on entry)

SETBRK ("Set <BREAK>")--Function Code 3

This routine lets you enable the <BREAK> key by defining a <BREAK>-key processing program. Whenever <BREAK> is pressed, your processing program takes over. On entry to the <BREAK> processing program, the return address of the interrupted routine is on the top of the stack and can be returned to with a RETurn instruction. All of the registers are intact upon entry to the routine.

The routine also lets you disable the <BREAK> key, by removing the address of the processing program. While <BREAK> is enabled, you cannot change processing programs; you must disable it first.

The <BREAK> key processing program must reside above X'27FF'.

See Handling Programmed Interrupts for programming information.

Entry Conditions

(HL) = Address of <BREAK> key processing program. When <BREAK> is pressed, control transfers to this address. If HL = 0, then address of previous processing program is removed.
A = 3

Exit Conditions

NZ => Error
A = Error Code
(HL) = Address of deleted <BREAK> key processing program if HL = 0 on entry

DISKID--Function Code 15

This routine reads the Diskette ID from any or all of drives 0 through 3. (The Diskette ID is assigned by the FORMAT and BACKUP utilities.) This routine is useful when the program needs to ensure that the Operator has inserted the proper diskette.

Entry Conditions

```
  B  = Drive Select Code. If B = 0, read from drive 0, etc.
       B must be one of the following: 0, 1, 2, 3, or 255. If
       B = 255, then routine reads from all four drives.
(HL) = Buffer to hold the diskette ID('s).
       If B = 0, 1, 2 or 3, then buffer must be 8-bytes long.
       If B = 255, then buffer must be 32 bytes long. Drive 0
       ID will be placed in first 8 bytes, then drive 1, etc.
  A  = 15
```

Exit Conditions

The Diskette ID('s) are placed in the buffers pointed to by register-pair HL. If a drive is not ready, blanks are placed into the buffer.

```
 NZ => Error
  A  = Error Code
```

TIMER
Function Code 25

This routine lets you start a timer to interrupt a program when time runs out. Unlike the DELAY routine, TIMER runs concurrent with your program. One application would be to give an operator a specified number of seconds for keyboard input, and to interrupt the keyboard input routine if no input was made within the time limit.

When setting the timer, you tell it how many seconds to count down. TRSDOS will then continue executing your program, until the timer counts down to zero or you reset the timer.

This is a "one-shot" timer. When it counts to zero and causes an interrupt, it automatically shuts off.

See Programming with TRSDOS for information on interrupts.

Entry Conditions

(HL) = Routine to handle interrupt when timer
counts to zero.
BC = Number of seconds to count down.
A = 25

If HL and BC both equal zero, then timer is turned off.
If HL = 0 and BC is not equal to zero, then time count is reset to the value in BC, and timing continues.

JP2DOS ("Jump to DOS")--Function 36

This program simply returns control to the command level (TRSDOS READY). All Open files are Closed automatically.

Entry Conditions

A = 36

DOSCMD ("DOS Command")--Function Code 37

This routine sends TRSDOS a command. After the command is executed, control returns to TRSDOS (TRSDOS READY). All Open files are closed automatically.

Entry Conditions

(HL) = TRSDOS command string
B = Length of command string
A = 37

RETCMD ("Return after Command")--Function Code 38

This routine sends TRSDOS an operator command. After completion of the command, control returns to your program. All Open files are Closed automatically.

NOTE

Take care that TRSDOS doesn't overlay your program while loading the command file you specified. Most TRSDOS library commands use memory below X'27FF'; a few go up to but not including X'2FFF'. Single-drive, single-disk copies use all user memory. See Library Commands for details.

Entry Conditions

(HL) = TRSDOS command string
B = Length of command string
A = 38

Exit Conditions

NZ => Error
A = Error Code

ERROR--Function Code 39

This routine displays the message "ERROR" followed by the specified error code. The message appears at the current cursor position.

Entry Conditions

B = Error Code
A = 39

Exit Conditions

NZ = Error
A = Error Code

ERRMSG ("Error Message")--Function Code 52

This routine returns an 80-byte descriptive error message to the specified buffer area. (See list of error messages, Section 3.2.)

Entry Conditions

```
 B   =  Error Code corresponding to message
(HL) = 80-byte buffer area in user area ( above X'27FF')
 A   =  52
```

Exit Conditions

```
NZ = Error
 A = Error Code
```

4.2 Keyboard

Supervisor calls described in this section*:

Function Code	Name	Purpose
1	KBINIT	Clears stored keystrokes.
4	KBCHAR	Gets a character from keyboard.
5	KBLINE	Gets a line from keyboard.
12	VIDKEY	Display message and get line from KB.

*VIDKEY is described in Section 4.3.

KBINIT ("Keyboard Initialize")--Function Code 1

This routine initializes the keyboard input driver. This call should be made before you start keyboard input. It clears all previous keystrokes.

Entry Conditions

A = 1

Exit Conditions

NZ => Error
A = Error Code

KBCHAR ("Keyboard Character")--Function Code 4

This routine gets one character from the keyboard. The routine returns immediately either with or without a character in register B.

The <BREAK> key is masked from the user--it will never be returned, since it is intercepted by the System. If the <BREAK> key is enabled, control passes to the processing program (see SETBRK) whenever <BREAK> is pressed. Otherwise, <BREAK> is ignored.

Entry Conditions

A = 4

Exit Conditions

B = Character found, if any. Only codes within the range <0,127> can be returned. If no character is returned, B is unchanged.
NZ => No character present
A = Error Code

KBLINE ("Keyboard Line")--Function Code 5

This routine inputs a line from the Keyboard into a buffer, and echoes the line to the Display, starting at the current cursor position. As each character is received and displayed, the cursor advances to the next position (Scroll Mode--see section 4.3.)

On entry to this routine, the input buffer is filled with periods, and these periods appear on the display, indicating the length of the input field for the operator's convenience.

The line ends when a carriage return is typed or when the input buffer is filled. A carriage return is always sent to the Display upon termination of line input, but is stored only if the Operator actually pressed <ENTER>.

Entry Conditions

(HL) = Start of input buffer
B = Maximum number of characters to receive;
0 < B
A = 5

Exit Conditions

B = Actual number of characters input, including carriage return.
C = 0 if input buffer was filled without carriage return.
If line ended with a carriage return, then C = X'0D'.

Received Control Codes, code < 32

Control codes not listed below are placed in the buffer and represented on the display with +/- symbols

KEY	HEX CODE	FUNCTION
<-	08	Backspaces the cursor to allow editing of line. Does not erase characters.
->	09	Advances the cursor to allow editing of line. Does not erase characters.
<ENTER>	0D	Terminates line. Clears trailing periods on display but not in buffer.
<CTRL-W>	17	Fills remainder of input buffer with blanks, blanks remainder of Display line.
<CTRL-X>	18	Fills remainder of input buffer with blanks, blanks to end of Display.
<ESC>	1B	Reinitializes input function by filling input buffer with periods and restoring cursor to original position.

(M2DOS7 8/6/79)

4.3 Video Display

Supervisor Calls described in this section:

Function Code	Name	Purpose
7	VDINIT	Initializes Display
8	VDCHAR	Sends a character, Scroll Mode
9	VDLINE	Sends a line, Scroll Mode
10	VDGRAF	Sends characters, Graphics Mode
11	VDREAD	Reads characters, Graphics Mode
12	VIDKEY	Displays message and gets line from KB
26	CURSOR	Turns cursor on or off
27	SCROLL	Sets number of lines at top of display which are not scrolled

The Display has two modes of operation--Scroll and Graphics. Cursor motion and allowable input characters are different in the two modes.

Scroll Mode

In the Scroll Mode, the Display can be thought of as a sequence of 1920 display positions, as illustrated below:

Line 0	0, 1, 2, 3, ... 78,79
Line 1	80,81,82,83,. 159
	.
	.
	.
	.
	.
	.
Line 22	1760,1761, ...1838,1839
Line 23	1840,1841, ...1919,1919

DISPLAY POSITIONS, SCROLL MODE

NOTE
The Display has two character sizes: 80 characters per line and 40 characters per line. The illustration above shows the 80 character per line mode.

In the scroll mode, each time an acceptable display character is received, it is displayed at the current cursor position, and the cursor advances to the next higher numbered position.

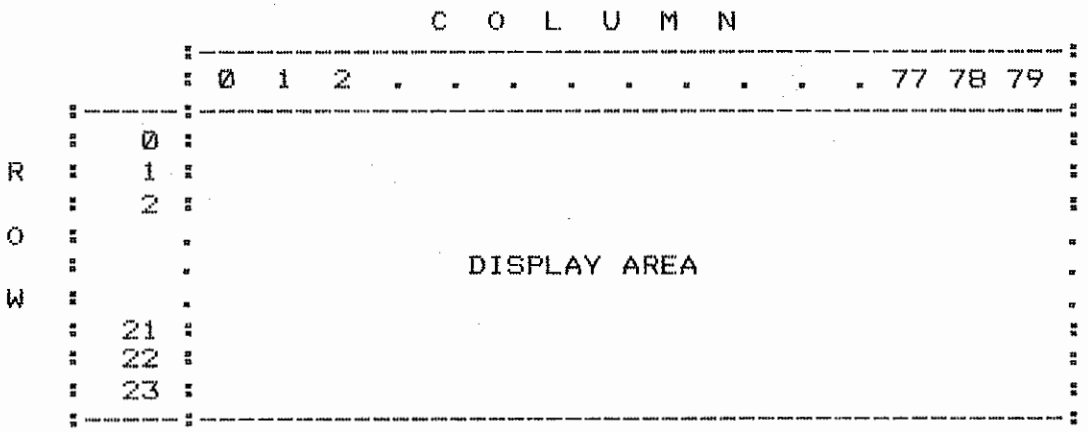
When the cursor is on the bottom line and a line-feed or carriage return is received, or when the bottom line is filled, the entire Display is "scrolled":

Line 0 is deleted
Lines 1-23 are moved up one line
Line 23 is blanked
The cursor is set to the beginning of line 23.

Note: From 0 to 23 lines at the display can be protected from scrolling via the SCROLL function call.

Graphics Mode

In the Graphics Mode, the Display can be thought of as an 80 by 24 matrix, as illustrated below:



DISPLAY POSITIONS, GRAPHICS MODE

NOTE
The Display has two character sizes:
80 characters per line and 40 characters per line. The illustration above shows the 80 character per line mode.

Each time an acceptable display character is received, it is diplayed at the current cursor position (which is set on entry to the Graphics Mode routines). Before displaying the next character, the cursor position is advanced, as follows:

If the cursor is to the left of Column 79, it advances to the next column position on the same row.

If the cursor is at Column 79, it wraps around to Column 0 on the same row.

In short, no scrolling is done in the Graphics Mode.

Cursor motion works the same way in all directions. For example, if the cursor is at Row 23, Column 40, and the v (down arrow) character is received, the cursor wraps around to Row 0 in the same column.

VDINIT ("Video Initialization")--Function Code 7

Call this initialization routine once before starting any I/O to the Display. It blanks the screen and resets the cursor to the top left corner, (position 1 in the Scroll Mode illustration).

Entry Conditions

B = Characters size switch. If B = 0 then sets to 40 characters/line size. Otherwise, sets to 80 characters/line size.
C = Normal/Reverse switch. If C = 0 then sets Reverse mode, black on white background. Otherwise sets Normal mode, white on black background.
A = 7

Exit Conditions

NZ => Error
A = Error Code

VDCHAR ("Video Character")--Function Code 8

This routine outputs a character to the current cursor position. It is a Scroll Mode routine, as described above.

Received Control Codes, code < X'20'

Control Codes not listed below are ignored.

KEY	HEX CODE	FUNCTION
F1	01	Cursor on.
F2	02	Cursor off.
BKSP	08	Moves cursor back one position and blanks the character at that position.
TAB	09	Advances cursor to next tab position. Tab positions are at 8-byte boundaries, 8,16,24,32,...
CTRL-J	0A	Line feed--cursor moves down to next row, same column position.
ENTER	0D	Moves cursor down to beginning of next line.
CTRL-W	17	Erases to end of line, cursor doesn't move.
CTRL-X	18	Erases to end of screen, cursor doesn't move.
CTRL-Y	19	Sets Normal Display mode (white on black). Remains Normal till reset by programmer.
CTRL-Z	1A	Sets Reverse Display mode (black on white). Remains Reverse till reset by programmer.
ESC	1B	Erases screen and homes cursor (position 0).
<-	1C	Moves cursor back one position.
->	1D	Moves cursor forward one position.

Entry Conditions

B = ASCII code for character to be output to the Display; character codes MUST be in the range <0,127>.
A = 8

Exit Conditions

NZ => Error
A = Error Code

VDLINE ("Video Line")--Function Code 9

This routine writes a buffer of data to the Display, starting at the current cursor position. It is a Scroll Mode routine.

The buffer should contain ASCII codes in the range <0,127>.

Received Control Codes, code < X'20'

Same as for VDCHAR.

Entry Conditions

(HL) = Beginning of buffer containing characters to be sent to the Display
B = Number of characters to be sent
C = End of line character. This character will be sent to the display after the buffer text.
A = 9

Exit Conditions

NZ => Error
A = Error Code

In case of an error:
B = Number of characters NOT displayed, including the one causing the error
C = Character causing the error

Upon return, the cursor is always set to the position following the last character displayed.

VDGRAF ("Video Graphics")--Function Code 10

This function displays a buffer of characters, starting at a specified row and column. It is a Graphics Mode routine (the cursor "wraps" the Display).

Displayable Characters
This routine lets you display the 32 graphics characters (and their reverse images). The codes are numbered from 0 through X'1F', and are pictured in the Operator's Manual. Codes X'20' through X'7F' are displayed as standard ASCII characters.

In addition, several special control codes are available:

HEX CODE	FUNCTION
0F9	Sets Normal (White on Black) mode. Cursor does not advance.
0FA	Sets Reverse (Black on White) mode. Cursor does not advance. Reverts to Normal on return from each graphics write.
0FB	Homes cursor (Row 0, Column 0).
0FC	<- Moves cursor back one space. Col.=Col-1. When column equals 79, cursor "wraps" to Col. 0 on preceding row.
0FD	-> Moves cursor forward one space. Col.=Col.+1. When column equals 0, cursor wraps to Col. 79 on next row down.
0FE	! (up a_row) Moves cursor up one row. Row=Row-1. "Wraps" to Row 23 when Row=0.
0FF	v (down arrow) Moves cursor down one row. Row=Row+1. "Wraps" up to Row 0 when Row=23.

At exit, the cursor is always set to the Graphics position immediately after the last character displayed. If the Buffer length was zero, the cursor is set to position specified in BC registers.

Entry Conditions

B = Row on screen to start displaying the buffer, B < 24. If B > 23, then B mod 24 is used as row position.
C = Column on screen to start displaying the buffer. In 80 charactcer/line mode, C < 80. For C > 79, C mod 80 is used as column position. In 40 character/line mode, C < 40. For C > 39, C mod 40 is used as column position.
D = Length of buffer, in range <0,255>
(HL) = Beginning of text buffer. The buffer should contain codes below X'80' or the special control codes above X'F8'. Any value outside these ranges will cause an error.
A = 10

Exit Conditions

NZ => Error (Invalid character sent)
A = Error Code

VDREAD ("Video Read")--Function Code 11

This routine reads characters from the Video Display into a specified buffer. It is a Graphics Mode routine; when it reads past the last column, it wraps back to column 1 on the next row. When it reads past column 79 on row 23, it wraps back to row 0, column 0.

Reverse (black on white) mode characters are read in as ASCII codes just like their Normal counterparts; reverse mode is indicated when the most significant bit (bit 7) is set.

This routine can also be used just to locate the cursor (see below).

Entry Conditions

```
 B   =  Row on screen where read starts, B < 24.
        If B > 23, then B mod 24 is used as row position.
 C   =  Column on screen where read starts.
        In 80 charactcer/line mode, C < 80.
        For C > 79, C mod 80 is used as column position.
        In 40 character/line mode, C < 40.
        For C > 39, C mod 40 is used as column position.
 D   =  Length of buffer, in range <0,255>. If D = 0, then
        B and C are ignored. Current cursor position will be
        returned as row, column in BC register pair.
(HL) = Beginning of text buffer
 A   =  11
```

Exit Conditions

```
 BC = Current cursor position, B = row, C = column. CURSOR
      postion at exit is the same as at entry--VDREAD does
      not change it.
 NZ = Error
  A = Error Code
```

VIDKEY--Function Code 12

This routine sends a prompting message to the Display and then waits for a line from the Keyboard. It is a Scroll Mode routine, combining the functions of VDLINE and KBLINE.

The routine writes the specified text buffer to the Display, starting at the current cursor position. The text buffer must contain codes < X'80'. Refer to VDLINE for a list of Received Control Codes and other details.

After the Video write, the cursor will be positioned immediately after the last character displayed. (To move it to another position, control codes can be placed at the end of the text buffer.)

Next, the routine gets a line from the Keyboard.

NOTE

Before starting the line input, all previously stored keystrokes are cleared.

Refer to KBLINE for a list of Received Control Codes and other details.

Entry Conditions

(HL) = Beginning of text buffer containing display message.
B = Number of characters to be displayed, B in the range <0,255>
C = Length of Keybord input field, C in the range <0,255>
(DE) = Beginning of text buffer where Keyboard input will be stored
A = 12

Exit Conditions

NZ => Error (Illegal value in display buffer)
A = Error Code

If Z is set (no error), then registers B and C contain:
B = Number of characters input from the Keyboard, including carrriage return, if any
C = Keyboard line termination. If C = 0, then input buffer was filled. Otherwise C = control character that terminated the line (carriage return)

If Z is not set (error), the registers B and C contain:
B = Number of characters not displayed, including the one causing the error
C = Character causing the error

CURSOR--Function Code 26

This routine turns the cursor display on or off. TRSDOS keeps track of the current cursor position whether it is on or off.

Entry Conditions

B = Function Switch. If B = Ø then cursor will be turned off. If B <> Ø then cursor will be turned on.
A = 26

SCROLL--Function Code 27

This routine lets you protect a portion of the Display from scrolling. From 0 to 23 lines at the TOP of the display can be protected; when scrolling occurs, only lines below the protected area will be changed.

Entry Conditions

B = Number of lines to be protected, in range <0,23>
A = 27

4.4 Line Printer

Supervisor Calls described in this section:

Function Code	Name	Purpose
17	PRINIT	Initializes the Line Printer Driver.
18	PRCHAR	Send a character to the Printer.
19	PRLINE	Send a line to the Printer.

PRINIT ("Printer Initialization")--Function Code 17

This routine initializes the Line Printer driver. It is automatically called when the system is initialized. You don't need to call it unless:

- You want to change some of the parameters.
- The Printer was not available (ready) when the System was initialized.

When initialized by the System, the following parameters are set:

Page Length (lines to a page)	: 66
Printed Lines per page	: 60
Automatic Form Feed	: Yes
Line length (Characters/Line)	: 132

NOTE

Linefeeds are done by the System during initialization. You may want to reset the paper before using the Printer for the first time.

Entry Conditions

B = Page Length (66 is standard)
C = Printed Lines per page (60 is standard).
If C = 0, no automatic form feed is done.
Otherwise, automatic form feed is done after C lines have been printed.
D = Maximum number of characters in a line (132 is standard)
A = 17

Exit Conditions

NZ => Error
A = Error Code

PRCHAR ("Print Character")--Function Code 18

This routine sends one character to the Printer.

NOTE
Most Printers do not print until their buffer is filled or a carriage return is received.

Entry Conditions

B = ASCII code for character to send
A = 18

Exit Conditions

NZ => Error
A = Error Code

PRLINE ("Print Line")--Function Code 19

This routine sends a line to the Printer. The line can include control characters as well as printable data. A tab character embedded in the buffer will cause the Printer to skip over to the next 8-byte boundary.

Entry Conditions

(HL) = Start of text buffer containing data and controls to send to Printer
B = Length of buffer (number of characters to send)
C = Control character (any character) to send after last character in buffer
A = 19

Exit Conditions

NZ => Error
A = Error Code

(M2DOS8 8/6/79)

4.5 File Access

Supervisor calls described in this section:

Function Code	Name	Function
33	LOCATE	Returns the current record number.
34	READNX	Gets next record (Sequential access)
35	DIRRD	Reads specified record (Direct Access).
40	OPEN	Sets up access to new or existing file.
41	KILL	Deletes the file from the directory.
42	CLOSE	Terminates access to an Open file.
43	WRITNX	Writes next record (Sequential Access).
44	POSNWR	Writes specified record (Direct Access).

OPEN - Function Code 40

This one call handles both the creation and Opening of files.

A given file can only be Open under one Data Control Block at a time. Because of the versatile file processing routines, this one DCB is sufficient to handle the various I/O applications.

Entry Conditions for OPEN

(DE) = 60-byte Data Control Block (see below).
(HL) = 11-byte Parameter List (see below).
A = 40.

Exit Conditions

NZ => Error.
A = Error Code.

Before calling OPEN, you must reserve space for the Data Control Block, Parameter List, Buffer Area and Record Area, as described below.

Data Control Block (60 bytes)

The Data Control Block (DCB) is used by the System for file access bookkeeping. You will also use it to pass the file specification for the file you want to Open, as follows:

Before calling OPEN, place the file specification at the beginning of the DCB, followed by a carriage return. See Chapterr 1, "File Specification" for details.

For example ($ signifies a carriage return):

CONTENTS OF FIRST BYTES OF DCB BEFORE OPEN

F I L E N A M E / E X T . P A S S W O R D : D (DISKNAME)$

While a file is Open, the filespec is replaced with information used by the System for bookkeeping. When the file is Closed, the original filespec (except for the password) will be put back into the DCB.

IMPORTANT NOTE

Do not ever modify any portion of the DCB while the file is Open. If you do, the results will be unpredictable.

Parameter List (11 bytes)

This list contains information TRSDOS needs to create or access the file:

CONTENTS OF PARAMETER LIST

BUFADR	RECADR	EODAD	R/W	RL	F/V	0/1/2	00

BUFADR (BUFFER ADDRESS). This two-byte field must point to the beginning of the Buffer Area.

The Buffer Area is the space TRSDOS will use to process all file accesses. If spanned records are possible, you must reserve 512 bytes. If no spanning is possible, reserve only 256 bytes.

With Fixed Length Record files, spanning is only required when the record length is not an even divisor of 256. For example, if the record length is 64, then each physical record contains four records exactly, and no spanning is required. In this case, reserve only 256 bytes for processing.

However, if the record length is 24 (not an even divisor of 256), then some records will have to be spanned. In this case, you will need to reserve 512 bytes.

With Variable Length Record files, you must always reserve 512 bytes for processing. This is because spanning may be required, depending on the lengths of the individual records in the file.

RECADR (RECORD ADDRESS). This two-byte field must point to the beginning of your Record Area.

For disk reads, this is where TRSDOS will place the record. For disk writes, this is where you put the record to be written.

Exception: For FLR files with a record length of 256, this address is not used. Your record will be in the buffer area pointed to by BUFADR.

For Fixed Length Record files with record length not equal to 256, this buffer should be the same size as the record length. For Variable Length Files, this area should be long enough to contain the longest record in the file (including the length-byte). If you are not sure what the longest record will be, reserve 256 bytes.

EODAD (END OF DATA ADDRESS). This two byte field can be used to give TRSDOS a transfer address to use in case the end of file is reached during an attempted read. If EODAD = 0 and end of file is reached,

the SVC will return with the end of file error code in register A.

R/W (READ OR WRITE). Put an ASCII "R" here to allow read-access only; put an ASCII "W" here to allow read and/or write access.

RL (RECORD LENGTH). This one-byte field specifies the record length to be used. Zero indicates a record length of 256. For Variable Length record files; this field is ignored. If the file already exists, and the Creation Code is 0, the System will supply the correct RL value, regardless of what you put there.

V/F (Variable or Fixed Length). This one byte field contains either an ASCII "V" for Variable or an ASCII "F" for Fixed. Once a file has been created, this attribute cannot be changed. If the file already exists, and the Creation Code is 0, the System will supply the correct F/V value, regardless of what you put in the parameter list.

0/1/2 (CREATION CODE). This one byte field contains a binary number 0, 1, or 2.

CODE	MEANING
0	Open the file only if it already exists. Do not create a new file in directory. Record Length and end of file are NOT reset.
1	Create a new file only; do not Open an existing file. Record Length and end of file are set at Open time.
2	Open existing file; if file not found, create it. Record Length and end of file are reset.

00 (END-OF-LIST MARKER). Always put a binary zero at the end of the Parameter List.

READNX--Function Code 34

This routine reads the next record after the current record. (Current record is the last record accessed.) If the file has just been Opened, READNX will read the first record.

Entry Conditions

(DE) = Data Control Block for currently Open file.
(HL) = Reserved for use in later versions of TRSDOS.
A = 34.

Exit Conditions

NZ = Error
A = Error Code.

Upon return, your record is in the Record Area pointed to by RECADR in the parameter list, or, if RL=256 and record type is Fixed, your record is in the area pointed to by BUFADR.

WRITNX--Function Code 43

This routine writes the next record after the last record accessed, that is, it writes sequentially. If WRITNX is the first access after the file is Opened, the first record will be written.

NOTE

When you write to a Variable-Length Record file, the end of file is reset to the last record written, regardless of its previous position.

Entry Conditions

(DE) = Data Control Block for currently Open file.
(HL) = Reserved for use in later versions of TRSDOS.
A = 43.

Before calling WRITNX, put your record in the record area pointed to by RECADR in the Parameter list, or, if LRL=256 and record type is Fixed, put record in area pointed to by BUFADR.

Exit Conditions

NZ = Error.
A => Error Code.

DIRRD--Function Code 35

This routine reads the specified record, allowing direct access.

NOTE

With VLR files, you can only use it to read the first record or to read the end of file.

Entry Conditions

(DE) = Data Control Block for currently Open file.
BC = Desired record number.
BC = 0 means position to beginning of file.
BC = X'FFFF' means position to end of file.
(HL) = Reserved for use in later versions of TRSDOS.
A = 35

Note: In a future release of TRSDOS, (BC) = address of a four-byte value specifying a record number.

Exit Conditions

NZ => Error.
A = Error Code.

Upon return, the record will be in the Record Area pointed to by RECADR in the parameter list, or, if RL=256 and record type is Fixed, your record is in the area pointed to by BUFADR.

DIRWR--Function Code 44

This routine writes the specified record. It writes the record into the specified record position of the file.

NOTE
For VLR files, you can only position to the beginning or end of file. When you write to a VLR file, the end of file is reset to last record written.

Entry Conditions

(DE) = Data Control Block for currently Open file.
BC = Record number you want to write.
BC = 0 means write first record in file.
BC = X'FFFF' means write record at end of file.
(HL) = Reserved for use in later versions of TRSDOS.
A = 44

Before calling DIRWR, put the record into the Record Area pointed to by RECADR in the Parameter List, or, if RL=256 and record type is Fixed, your record is in the area pointed to by BUFADR.

Note: In a future release of TRSDOS, (BC) = address of a four-byte value specifying a record number.

Exit Conditions

NZ => Error.
A = Error Code.

LOCATE--Function Call 33

This function returns the number of the current record; i.e., the number of the last record accessed. You can use this call only with Fixed Length Record files.

Entry Conditions

(DE) = Data Control Block for currently Open file.
(HL) = Reserved for use in later versions of TRSDOS.
A = 33

Note: In a future release of TRSDOS; (BC) = address of a four-byte value specifying a record number.

Exit Conditions

BC = Current Record Number
NZ => Error
A = Error Code

CLOSE--Function Code 42

This routine terminates access to the file. If there are records in the Buffer Area not yet written, they will be written at this time.

Entry Conditions

(DE) = Data Control Block for currently Open file.
A = 42

Exit Conditions

NZ => Error
A = Error Code

Upon return, the filespec (except for the password) will be put back into the DCB.

KILL--Function Code 41

This routine deletes the specified file from the directory. A file must be Closed before it can be killed.

Entry Conditions

(DE) = Data Control Block, containing standard TRSDOS filespec (see illustration in description of OPEN).
A = 41

Exit Conditions

NZ => Error.
A = Error Code.

M2DOS9 8/6/79

4.6 Computational

Supervisor calls described in this section:

Function Code	Name	Function
6	DELAY	Provides a delay-loop
20	RANDOM	Provides a random number, range <0,254>
21	BINDEC	Converts binary to ASCII-coded decimal, and vice versa
22	STCMP	Compares two text strings
23	MPYDIV	Performs 8 bit * 16 bit multiplication and 16 bit / 8 bit division
24	BINHEX	Converts binary to ASCII-coded hexadecimal, and vice-versa
45	DATE	Sets or returns the time and date.
46	PARSER	Finds the alphanumeric parameter field in a text string
49	STSCAN	Looks for a specified string inside a text buffer

DELAY--Function Code 6

This routine provides a delay routine, returning control to the calling program after the specified time has elapsed.

Entry Conditions

BC = Delay Multiplier. If BC = 0, then delay time will be 426 milliseconds. If BC > 0, then delay time will be:
6.5 * (BC - 1) + 22 microseconds

A = 6

RANDOM--Function Code 20

This routine returns a random one-byte value. To extend the cycle of repetition, the instantaneous time/date are used in generating the number.

You pass the routine a limit value; the value returned is in the range <0,limit-1>. For example, if the limit is 255, then the value returned will be in the range <0,254>.

Entry Conditions

B = Limit value.
A = 20

Exit Conditions

C = Random number.
For B > 1, number returned is in range <0,B-1>.
For B = 0 or 1, number returned = 0

BINDEC--Function Code 21

This routine converts a two-byte binary number to ASCII-coded decimal, and vice versa. Decimal range is <0,65535>.

Entry Conditions

B = Function Switch.
If B = 0, then convert binary to ASCII decimal.
If B not 0 then convert ASCII decimal to binary.

Contents of other registers when B = 0 (bin->dec):
DE = Two-byte binary number to convert
(HL) = 5 byte area to contain ASCII coded decimal value upon return. The field will contain decimal digits (X'30'-X'39') leading zeroes on the left as necessary to fill the field, for example, the number 21 would be: 00021.

Contents of other registers when B is not 0 (dec->bin):
(HL) = 5-byte area containing ASCII decimal value to be converted to binary.
A = 21

Exit Conditions

(HL) = Decimal value
DE = Binary value

BINHEX--Function Code 24

This routine converts a two-byte binary number to ASCII-coded hexadecimal, and vice versa. Hexadecimal range is <0,FFFF>.

Entry Conditions

B = Function Switch.
If B = 0, then convert binary to ASCII hexadecimal.
If B is not 0, convert ASCII hexadecimal to binary.
A = 24

Contents of other registers when B = 0 (bin->hex):
DE = Two-byte binary number to convert
(HL) = 4 byte area to contain ASCII coded hexadecimal value upon return. The field will contain hexadecimal digits with leading zeroes on the left as necessary to fill the field, for example, the number X'FF' would be: 00FF.

Contents of other registers when B is not 0 (hex->bin):
(HL) = 4-byte area containing ASCII hexadecimal value to be converted as described above.

Exit Conditions

(HL) = Hexadecimal value
DE = Binary value

MPYDIV ("Multiply Divide")--Function Code 23

This routine does multiplication and division with one 2-byte value and one 1-byte value.

Entry Conditions

B = Function switch.
If B = 0 then multiply.
If B is not 0 then divide.
A = 23

For multiplication:
HL = Multiplicand
C = Multiplier

For division:
HL = Dividend
C = Divisor

Exit Conditions

HL = Result (product HL * C or quotient HL/C)
A = Overflow byte (multiplication only)
C = Remainder (division only)

Status bits affected by division:
Carry flag set if dividing by zero. Divide not attempted.
Z flag set only if the quotient is zero.

Status bits affected by multiplication:
Carry Flag set if overflow.
Z flag set only if result is zero.

STCMP--Function Code 22

This routine compares two strings to determine their collating sequence.

Entry Conditions

(DE) = First string.
(HL) = Second string.
BC = Number of characters to compare.
A = 22

Exit Conditions

Status bits indicate results, as follows:

Z Flag set indicates strings are identical.
NZ indicates strings not identical

Carry Flag set indicates first string (pointed to by DE) precedes second string (HL) in collating sequence.

Other register contents:
A = First non-matching character in first string.

When strings are not equal, you can get further information from the prime registers, as follows:
(HL') = Address of first non-matching character in second string.
(DE') = Address of first non-matching character in first string.
BC' = Number of characters remaining, including the non-matching character.

DATE--Function Code 45

This routine sets or returns the real-time (time and date). The data can be returned as a 26-byte ASCII string containing 8 fields, as illustrated below (numbers refer to byte lengths of each field).

NAME OF DAY	MON.	DAY OF MON.	YR	DAY OF YEAR	TIME	MON #	DAY OF WEEK
3	3	2	4	3	8	2	1

CONTENTS OF TIME/DATE STRING
(Length of field is shown under each field)

Example Time/Date string:
SATAPR28197911813.20.42045
Represents the data "Saturday, April 28, 1979, 118th day of the year, 13.20.42 hours, 4th month of the year, 5th day of the week. (Periods are used instead of colons, since they're more easily entered from the keyboard.)

NOTES
DAY OF WEEK Field: Monday is Day 0.
The date calculations are based on the Julian Calendar.

Entry Conditions

B = Function Switch.

If B = 0 (Get time/date):
(HL) = 26 byte buffer where date/time will be stored.

If B = 1 (Set date):
(HL) = 10 byte buffer containing date in this form:
MM/DD/YYYY

If B = 2 (Set time):
(HL) = 8 byte buffer containing time in this form:
HH.MM.SS

PARSER--Function Code 46

This general-purpose routine "parses" (analyzes) a text buffer into fields and subfields. PARSER is useful for analyzing TRSDOS command lines (including keyword commands, file specifications, keyword options and parameters. It can also be used as a fundamental routine for a compiler or text editor. This versatility derives from the generality of the routine.

By necessity, the description of PARSER is rather long and detailed. In actual use, the routine is as convenient as it is powerful. For example, PARSER is designed to allow repetitive calls for processing a text buffer; on exit from the routine, parameters for the next call are all readily available in appropriate registers.

The routine has pre-defined sets of field-characters and separators; you can use these or re-define them to suit your application.

In general, a field is any string of alphanumeric characters (A-Z,a-z,0-9) with no embedded blanks. Fields are delimited by separators and terminators, defined below. For example, the line:

```
  BAUD=300, PARITY=EVEN, WORD=7
```

contains 6 fields: BAUD, 300, PARITY, EVEN, and 7.

However, a field can also be delimited by paired quote marks:

```
   "field" or 'field'
```

When the quote marks are used, ANY characters, not just alphanumerics, are taken as part of the field. The quote marks are not included in the field. For example, the line:

```
   'DATE (07/11/79)'
```

will be interpreted as one field containing everything inside the quotes.

When a quote mark is used to mark the start of a field, the same type of quote mark must be used to mark the end of the field. This allows you to include quotes in a field, for example:

```
   "X'FF00'"
```

will be parsed as one field containing EVERYTHING inside the double quotes " ", including the single quote marks ' '.

A separator is any non-alphanumeric character. PARSE will always stop when a separator is encountered, EXCEPT when the separator is a blank (X'20'). Leading and trailing blanks are ignored. After trailing blanks, PARSE stops at the beginning of the next field, or on the first non-blank separator.

You can also define terminators, which will stop the parse regardless of whether a field has been found. Unless you specifically define these, PARSE will only stop on non-blank separators.

Separators and terminators have the same effect on a parse; the only difference is in how they affect the F (Flag) register on exit.

To re-define the field, separator, and terminator sets

If you need to change the field and separator sets, or define terminators, you can provide three change-lists via a List Address Block, explained later.

Entry Conditions

(HL) = Start address of text buffer
(DE) = Address of List Address Block.
DE = 0 indicates no lists are to be used.
C = Maximum length of parse
A = 46

Exit Conditions

(HL) = Field-position pointer:
(HL) = First byte of field, if a delimited field was found.
(HL) = Terminator or non-blank separator if no field was found.
(HL) = Last byte of buffer if parse reached maximum length.
B = Actual length of field, excluding leading and trailing blanks.
A = Character preceding the field just found. If B = 0 A = X'FF'
C = Number of bytes remaining to parse after terminator or separator. Note that trailing blanks have been parsed.
D = Separator or terminator at end of field.
If D = X'FF' then parse stopped without finding a non-blank separator or terminator.
E = Displacement pointer for next parse call.
Add E to HL to get:
a) Beginning address of next field, or
b) Address of byte following the last byte parsed.
Note that if parse reached maximum length, then E + HL = Address following end of text buffer.
If parse did not reach maximum length, and E = 1, then E + HL = Address following separator or terminator.

Status bits (F register) affected when parse did NOT reach reach maximum length:
Z flag:
Z (set) if parse ended with a separator.
NZ (not set) if parse ended with a terminator.
C flag:
C (set) if there were trailing blanks between end of field and next non-blank separator or terminator.
NC (reset--not set) if there were no trailing blanks.

List Address Block

The List Address Block is six bytes long and contains two-byte addresses (lsb,msb) for three change-lists:

- List 1: Characters to be used as terminators
- List 2: Additions to the set of field characters
- List 3: Deletions from the set of field characters; i.e., alphanumerics to be interpreted as separators.

Each list has the following form:

1st byte	2nd byte	3rd byte	----------	n+1 byte
n, number of char's in list	first character	second character	----------	nth character

Notes:

1. There are three ways to indicate a null list:
   a) Set the character-count byte (n) equal to zero.
   b) Set the pointer in the List Address Block to zero.
   c) Set DE=0 if you aren't going provide any lists.
2. Characters are stored in lists in ASCII form.
3. If a character appears in more than one list, it will have the characteristics of the first list that contains it.

Here's is a typical List Address Block with its associated lists. Assume that on entry to PARSER, (DE) = X'8000'.

Address	Length	Hexadecimal Contents
X'8000'	2	X'9000' (start of list 1)
X'8002'	2	X'9010' (start of list 2)
X'8004'	2	X'9020' (start of list 3)
X'9000'	1	X'04' (4 char's in list 1)
X'9001'	1	X'0D' (carriage return)
X'9002'	1	X'7B' (left brace {)
X'9003'	1	X'7D' (right brace })
X'9004'	1	X'29' (left paren))
X'9010'	1	X'03' (3 char's in list 2)
X'9011'	1	X'3F' ("?")
X'9012'	1	X'40' ("@")
X'9013'	1	X'23' ("#")
X'9020'	1	X'00' (null list)

Sample Programming

The following code shows typical repetitive uses of PARSER to break up a parameter list.

```
;-----------------PREPARE FOR PARSING LOOP-----------------------------------
        LD      C,MAXLEN        ; C = Maximum length to parse
        LD      E, 0            ; For initial call to NXTFLD
        LD      HL,BUFFER       ; (HL) = string to parse
;-----------------PARSE LOOP-------------------------------------------------
PARSE   CALL    NXTFLD          ; Routine to call PARSER
        CALL    HANDLR          ; Routine to handle new field
        JR      NZ,NXTRTN       ; Go to next routine if
                                ;  parsed ended on terminator
        LD      A,C             ; Else get new max length
        OR      A               ; Is it zero?
        JR      NZ,PARSE        ; If not, then continue
        LD      A,0FFH          ;
        CP      D               ; If D=0FFH then no separator
                                ;  at end of buffer.
        JR      Z,ERR           ; So go to error routine
        JR      NXTRTN          ; Else, then do next routine.
;-----------------FIELD-HANDLING ROUTINE-------------------------------------
HANDLR  PUSH    AF              ; Must save status registers
                                ; and any other registers
                                ; will be changed.
                                ;
; Processing code goes here...
        POP     AF              ; Restore AF (and other
                                ; registers saved at entry)
        RET
;-----------------CALL TO PARSER---------------------------------------------
NXTFLD  LD      D,0             ; Zero msb of DE
        ADD     HL,DE           ; (HL) = where to start parse
        LD      DE,LAB          ; (DE) = List address block
                                ; If DE=0 then no lists used.
        LD      A,46            ; Function Code
        RST     8
        RET
;-----------------PROGRAM CONTINUES HERE-------------------------------------
NXTRTN  EQU     $
```

STSCAN ("String Scan")--Function Code 49

This is a general purpose string scan. It searches through a specified text buffer for the specified string. This string can consist of any values 0-255 (it is not strictly alphanumeric).

Entry Conditions

(HL) = Text area to be searched.
(DE) = Compare string.
B = Length of compare string.
A = 49

Exit Conditions

NZ => String not found.
Z => String found.
(HL) = Start position of matching string in search string.

(m2dos10 8/9/79)

4.7 Serial Communications

Supervisor Calls described in this section:

Code	Name	Function
55	RS232C	Set or turn off channel A or B for serial input/output.
96	ARCV	Channel A receive
97	ATX	Channel A transmit
98	BRCV	Channel B receive
99	BTX	Channel B receive

These routines allow you to use the Model II's RS-232C interface, channels A and B on the back panel. See the Model II Operation Manual for a description of signals available.

RS232C--Initialize RS-232C Channel
Function Code 55

This routine sets up or disables either channel A or B. Before using it, the appropriate channel should be connected to the modem or other equipment.

This routine sets the standard RS-232C parameters, and defines a pair of supervisor calls for I/O to the specified channel. When you initialize Channel A, SVC's 96 and 97 are defined; when you initialize Channel B, SVC's 98 and 99 are defined. See ARCV, ARTX, BRCV, and BTX.

Before re-initializing a channel, ALWAYS turn it off.

Entry Conditions

(HL) = Parameter list described below.
B = Function switch
If B is not equal to zero then turn on channel and define I/O SVC's for that channel.
If B is equal to zero then turn off channel and delete I/O SVC's for that channel. In this case only the first byte in the parameter list ("A" or "B") is used.
A = 55

Parameter List

This six-byte list includes the necessary RS-232C parameters:

CHANNEL	BAUD RATE	WORD LENGTH	PARITY	STOP BITS	END LIST MARKER

CHANNEL is an ASCII "A" for channel A, or "B" for channel B.

BAUD RATE is a binary value from 1 to 7:

1	for 100 baud
2	for 150 baud
3	for 300 baud
4	for 600 baud
5	for 1200 baud
6	for 2400 baud
7	for 4800 baud

WORD LENGTH is a binary value from 5 to 8:

5	for 5-bit words
6	for 6-bit words
7	for 7-bit words
8	for 8-bit words

PARITY is an ASCII "E" for even, "O" for odd, or "N" for none.

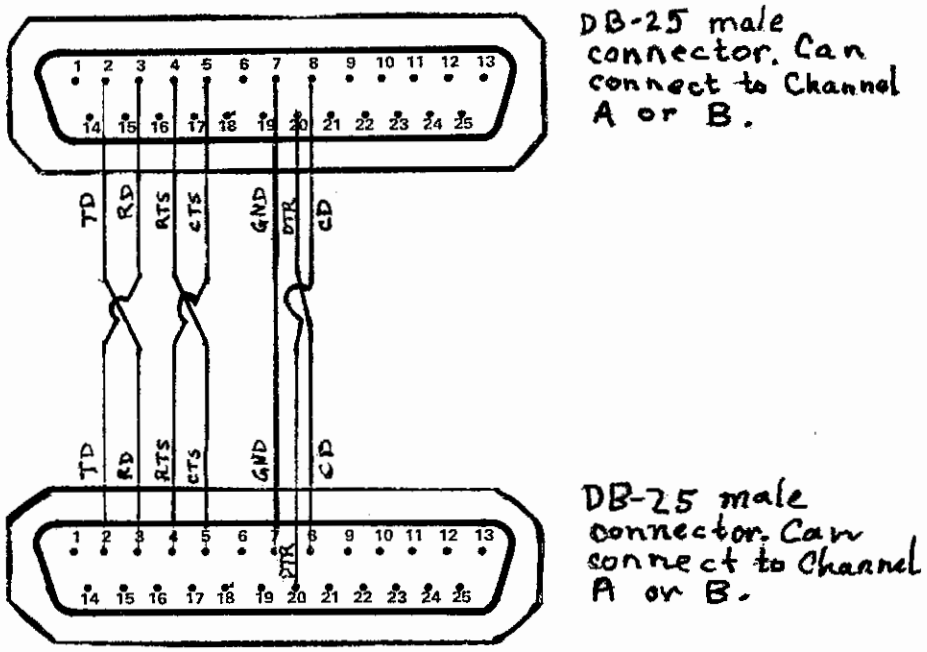
STOP BITS is a binary 1 for 1 stop bit, or 2 for 2 stop bits.

END LIST MARKER is a binary Ø.

Exit Conditions

NZ = Error
A = Error return code

For hard-wiring between two Model II's without a modem, use the wiring arrangement described below:



Connection Diagram, Model II (Channel A or B) to Model II (Channel A or B). Use stranded wire, 24-gauge, to connect two DB-25 connectors as illustrated. If wire length exceeds 50 feet, twist lines 7 (GND), 2 (TD) and 3 (RD). Refer to the Model II Operation Manual for a description of signals available.

ARCV--Channel A Receive
Function Code 96

This routine inputs a character from the serial Channel A. In practice, it is analogous to keyboard character input (see KBCHAR).

TRSDOS sets up ARCV and ATX when you initialize channel A (see RS232C). If you call this routine without previously initializing channel A, you will get an error return code of 1 (no function exists).

Entry Conditions

A = 96

Exit Conditions

B = Character found, if any.
NZ = No character found.
Carry Flag Set = Modem carrier not present when SVC was entered
A = Communictions status:

Bit	Meaning when set
0	Not used.
1	Not used.
2	Not used.
3	Modem carrier was lost.
4	Parity error occurred on character found in register B.
5	Data lost--more than one character received between SVC's. B contains last character rec'd.
6	Framing error occurred on last character rec'd.
7	A "break sequence" (extended null character) was received.

ATX--Channel A Transmit
Function Code 97

This routine sends a character to the serial Channel A. In practice, it is analogous to video character output (see VDCHAR).

TRSDOS sets up ARCV and ATX when you initialize channel A (see RS232C). If you call this routine without previously initializing channel A, you will get an error return code of 1 (no function exists).

Entry Conditions

B = ASCII code for character to be sent
A = 97

Exit Conditions

NZ = No character sent
Carry Flag Set = Modem carrier not present when SVC was entered.
A = Communictions status:

Bit	Meaning when set
0	Clear to Send (CTS) was not detected.
1	Not used.
2	Transmitter is busy.
3	Modem carrier was lost.
4	Not used.
5	Not used.
6	Not used.
7	Not used.

BRCV--Channel B Receive
Function Code 98

This routine inputs a character from the serial Channel B. In practice, it is analogous to keyboard character input (see KBCHAR).

TRSDOS sets up BRCV and BTX when you initialize channel B (see RS232C). If you call this routine without previously initializing channel B, you will get an error return code of 1 (no function exists).

Entry Conditions

A = 98

Exit Conditions

B = Character found, if any.
NZ = No character found.
Carry Flag Set = Modem carrier not present when SVC was entered.
A = Communictions status:

Bit	Meaning when set
0	Not used.
1	Not used.
2	Not used.
3	Modem carrier was lost.
4	Parity error occurred on character found in register B.
5	Data lost--more than one character received between SVC's. B contains last character rec'd.
6	Framing error occurred on last character rec'd.
7	A "break sequence" (extended null character) was received.

BTX--Channel B Transmit
Function Code 99

This routine sends a character to the serial Channel B. In practice, it is analogous to video character output (see VDCHAR).

TRSDOS sets up BRCV and BTX when you initialize channel B (see RS232C). If you call this routine without previously initializing channel B, you will get an error return code of 1 (no function exists).

Entry Conditions

B = ASCII code for character to be sent
A = 99

Exit Conditions

NZ = No character sent
Carry Flag Set = Modem carrier not present when SVC was entered.
A = Communictions status:

Bit	Meaning when set
0	Clear to Send (CTS) was not detected.
1	Not used.
2	Transmitter is busy.
3	Modem carrier was lost.
4	Not used.
5	Not used.
6	Not used.
7	Not used.

5. Programming with TRSDOS

This section tells you how execute your own machine-language programs under TRSDOS. In includes two sections:

- Program Entry Conditions--how control is transferred to your program after it is loaded from disk.
- Handling Programmed Interrupts--how to write an interrupt service routine for BREAK-key processing and TIMER interrupts.

To create and use a program:

1. Enter the program into memory, either with DEBUG, an assembler, or via the serial interface channel from another device.
2. Use the DUMP command to save the program as an executable disk file, setting load and transfer addresses.
3. To run the program, input the file name to the TRSDOS command interpreter (TRSDOS READY mode).

5.1 Program Entry Conditions

Upon entry to your program, TRSDOS sets up the following registers:

BC = First byte following your program, i.e., the first free byte for use by your program.

DE = Highest memory address not protected by TRSDOS, i.e., the end of memory which can be used by your program.

HL = Buffer containing the last command entered to the TRSDOS command interpreter. The first byte of the buffer contains the length of the command line, not including the carriage return. The text of the command follows this length byte.

length of text: n	1st byte of command	2nd byte of command	.....	nth byte of command	X'ØD'

5.2 Handling Programmed Interrupts

TRSDOS allows two user-programmed interrupts as described under SETBRK and TIMER. When either kind of interrupt is received (BREAK key is pressed or TIMER counts to zero), control transfer to your interrupt handling routine.

Note: System routines called by your program are also subject to interrupts. Interrupt handlers can also be interrupted.

Upon entry to your interrupt processing routine, TRSDOS sets up the registers as follows:

(SP) = The address of the next instruction to be executed

when the interrupt was received.
Other registers:
Contents are the same as they were when the interrupt was received.

Before doing any processing, you should save all registers. When finished processing, restore all registers and execute a return to continue with the interrupted program.

It is good practice to keep interrupt handling routines short; ideally, the routine simply flags the main program that an interrupt has occurred and returns. The main program can then respond to the interrupt flag when convenient.

Always end your interrupt handler with the RET instruction and with all registers intact.

TRSDOS is serially reusable but not always re-entrant. More specifically, your interrupt routine should not make use of the supervisor calls, since under some conditions this will produce unpredictable results.

MODEL II TRSDOS

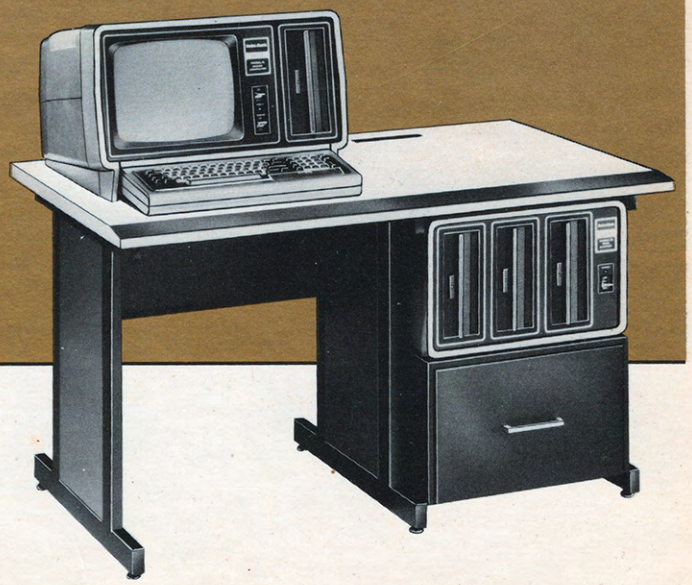
5 / INDEX

INDEX

Radio Shack®

TRS-80 Model II BASIC Reference Manual

A Description of the Model II BASIC Programming Language: Definitions, Syntax, Examples and Sample Programs



BASIC

CUSTOM MANUFACTURED IN THE U.S.A. BY RADIO SHACK TC A DIVISION OF TANDY CORPORATION

RADIO SHACK (R)

MODEL II BASIC

LANGUAGE REFERENCE MANUAL

C O N T E N T S

(M2BASIC1 8/9/79)

1 / Using Model II BASIC

General Information

Model II BASIC is an easy-to-use, extended version of the BASIC programming language. It is designed to run under the TRS-80 Disk Operating System (TRSDOS), and is included on the System diskette.

Model II BASIC executes your programs directly. It does not produce a low-level, machine-language translation. In technical terms, it is an interpreter, not a compiler. This makes it especially powerful for interactive use during program development and debugging.

Model II BASIC offers all the standard features of the language, plus several important additions, including:

- Program line renumbering
- Line editor for easy program corrections and changes
- Ability to execute a TRSDOS command and return to BASIC with program and variables intact
- Direct and sequential access to data in disk files
- Special functions to allow BASIC programs to call machine-language subroutines
- Recovery from operator errors--the System won't stop if you attempt output to a device (such as a Printer or Disk Drive) which is not ready.

About This Reference Manual

This manual describes the keywords, data types, and other features which are available in Model II BASIC. You'll find plenty of examples and sample programs to help you try out the language. There is also a Glossary in the Appendix.

The manual is organized this way:

CHAPTER 1. Using Model II BASIC
- A. General Information
- B. Notation
- C. Memory Requirements
- D. Loading BASIC
- E. Modes of Operation
- F. Using the Keyboard
- G. Using the Video Display

CHAPTER 2. BASIC Concepts
- A. Data
  - 1. Data Storage Types
    - a. Numbers (Integer, Single and Double Precision)
    - b. Strings
  - 2. Constants
    - a. Numbers and Strings
    - b. Types of Constants
  - 3. Variables
    - a. Names
    - b. Type Declaration
      - i. Default types
      - ii. Tags (!,#,%,$)
    - c. Arrays
  - 4. Data Conversion
- B. Operations
  - 1. Statements
  - 2. Expressions
    - a. Operators
      - i. Arithmetic
      - ii. Logical, Relational, and Boolean
      - iii. String
    - b. Evaluation of Expressions
      - i. Parentheses
      - ii. Order of Operations
      - iii. Type Conversions
    - c. Functions

For More Information

If you are a newcomer to BASIC, you'll probably need a good programming manual to use along with this book. Here are a few we recommend:

BASIC AND THE PERSONAL COMPUTER, Thomas Dwyer and Margot Critchfield; Addison-Wesley Publishing Company, 1978.

BASIC FOR HOME COMPUTERS: A SELF-TEACHING GUIDE, Bob Albrecht, LeRoy Finkel, and Jerald R. Brown; Wiley & Sons, 1978.

BASIC FROM THE GROUND UP, David E. Simon; Hayden Book Company, 1978.

ILLUSTRATING BASIC, Donald Alcock; Cambridge University Press, 1977.

Notation

For clarity and brevity, we use some special notation and type styles in this manual.

CAPITALS and punctuation
Indicate material which must be entered exactly as it appears. (The only punctuation symbols not entered are triple-periods, explained below.) For example, in the line:

```
PRINT "THE TIME IS " TIME$
```

every letter and character should be typed exactly as indicated.

lowercase italics
Represent words, letters, characters or values you supply from a set of acceptable values for a particular command. For example, the line:

```
LIST line-range
```

indicates that you can supply any valid *line-range* specification (defined later) after LIST.

... (ellipsis)
Indicates that preceding items can be repeated. For example:

```
INPUT variable, ...
```

Indicates that several variables may be repeated after INPUT.

␣
This special symbol is used occasionally to indicate a blank space character (ASCII code 32). For example:

```
INPUT "WHAT IS YOUR NAME?␣";N$
```

The ␣ indicates that there is a single blank space after the question mark.

<aaaa,bbbb>
Indicates a numeric range with lower limit aaaa and upper limit bbbb. Both limits are included in the range. For example:

```
<-32768,32767>
```

represents the range of numbers from -32768 to 32767 inclusive. The context will specify whether integers or real numbers are intended.

X'NNNN'
Indicates that NNNN is a hexadecimal number. Numbers used in this manual are in decimal form, unless otherwise noted. For example:

```
X'700A'
```

is a hexadecimal representation of the decimal number 28682.

O'NNNNNN'

Indicates that NNNNN is an octal number. Numbers used in this manual are in decimal form; unless otherwise noted. For example:

```
Ø'17707'
```

is an octal representation of the decimal number 8135.

<keyname>
Indicates one of the keys; usually a special control key like <ENTER>. For example:

```
PRINT "THE TIME IS "  TIME$  <ENTER>
```

indicates you should press <ENTER> after typing in the text.

<CTRL-keyname>
Indicates a control character. To output the character, hold down <CTRL> and press the specified key. For example:

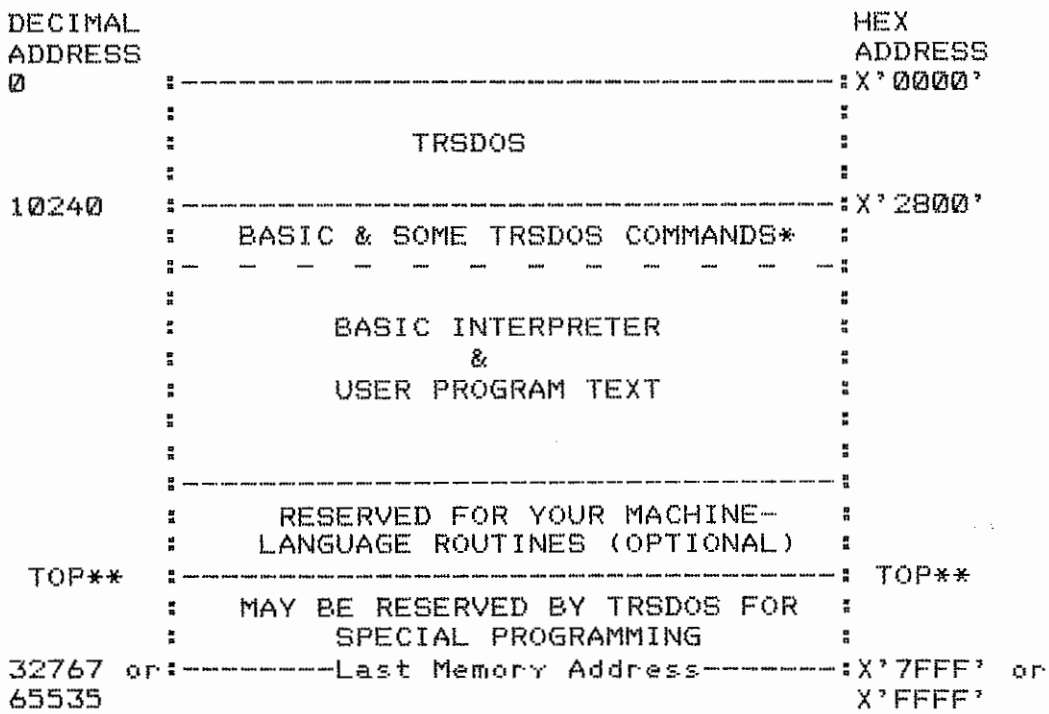
```
<CTRL-R>
```

Indicates that you should hold down <CTRL> and press <R>

Memory Requirements

BASIC occupies 14 granules (17920 bytes) on the System disk. It loads into memory starting at the beginning of user memory, 10240. The amount of memory required by BASIC depends on how many concurrent data file you specify when you load BASIC. During loading, you can also to reserve a portion of high memory for storage of machine-language subroutines.

Here's a memory allocation map:



* Certain TRSDOS commands use memory in the range <X'2800,X'2FFF'>. See "Library Commands" in the TRSDOS Reference Manual for a list. All TRSDOS commands except for these can be called from BASIC via the BASIC command, SYSTEM.

** TOP is a memory protect address set by TRSDOS. If TRSDOS is not protecting high memory, then TOP is the same as Last Memory Address.

Loading BASIC

See the Operation Manual for instructions on connection, power-up and inserting the System diskette.

Note: A System diskette must be in Drive 0 (the built-in unit) whenever the Computer is in use. Whenever the Computer is turned On or Reset, it will automatically load TRSDOS from Drive 0.

After the System starts up, it will prompt you to key in the date. Type in the date in MM/DD/YYYY form and press <ENTER>. For example:

07/25/1979 <ENTER>

for July 25, 1979.

Next the System will prompt you to key in the time. TO SKIP THIS QUESTION, press <ENTER>. The time will start at 00:00:00.

TO SET THE TIME, type in the time in HH.MM.SS 24-hour form. Periods are used instead of colons, since they're easier to type in. The seconds .SS are optional. For example:

14.30 <ENTER>

for 2:30 PM.

The System will record the date and time internally and return with the message:

```
TRSDOS READY
.....................................................
```

You can now load and execute BASIC. The simplest way to do this is to type:

```
BASIC    <ENTER>
```

BASIC will load (takes several seconds) and display a start-up heading like this:

```
Radio Shack TRS-80 Model II BASIC Vers  1.1
xxxxx Bytes free, 0 files
Ready
>
```

XXXXX bytes free tells you how much memory is available for storage and execution of BASIC programs. 0 files tells you that no data files can be Opened from BASIC. If you want to Open data files, you need to specify how many when you load BASIC (see next paragraphs).

Options for Loading BASIC

There are several other ways to start up BASIC, as summarized in this block:

```
BASIC program  -F:files -M:address
        program is a TRSDOS file specification for a
           BASIC program. After start-up, BASIC will run
           it. If program is omitted, BASIC will start-up
           in the command mode.
        -F:files tells BASIC the maximum number of files
           that may be Open at once. files is a number
           from 0 to 15. If -F:files is omitted, maximum
           is set to 0.
        -M:address tells BASIC not to use memory
           above address. address is a decimal number.
           If -M:address is omitted, BASIC uses all
           memory up to TOP.
```

The options allow you to specify any or all of the following:

- A program to run after BASIC is started.
- Maximum number of data files that may be Open at once. The larger the number of files, the less area available for storing and executing your programs. (Each file you specify takes 834 bytes of memory.) So use the smallest value that will suit your needs.
- Highest address to be used by BASIC during program execution. Omit this unless you are going to call machine-language subroutines.

Examples

```
TRSDOS READY
BASIC          <ENTER>
```

Tells BASIC not to run a program, but to enter the command mode; to allow for zero concurrent files; and to use all memory available from TRSDOS.

```
TRSDOS READY
BASIC -F:1
```

Just like the preceding example, except that only one file can be Open at any given time.

```
TRSDOS READY
BASIC -M:32000
```

BASIC won't allow you to Open any files, and 32000 is the highest address it will use during program execution.

```
TRSDOS READY
BASIC PAYROLL -F:3
```

BASIC will start up, load and run the BASIC program PAYROLL; three data files can be Opened, and BASIC can use all memory available from TRSDOS.

Modes of Operation

BASIC has three modes of operation:

- Command mode--for typing in program lines and immediate lines
- Execute mode--for execution of programs and immediate lines
- Edit mode--for editing program and immediate lines

Command Mode

Whenever you enter the command mode, BASIC displays a header and a special prompt:

```
Ready    (header)
>        (prompt followed by blinking block
          "cursor")
```

While you are in the command mode, BASIC will display the prompt at the beginning of the current logical line (the line you are typing in).

A logical line is a string of up to 255 characters and is always terminated with a carriage return (stored when you press <ENTER>). A physical line, on the other hand, is one line on the Display. A physical line contains a maximum of 80 characters.

For example, if you type 100 R's and then press <ENTER>, you will have two physical lines, but only one logical line.

The blinking block is called a cursor. It tells you where the next character you type will be displayed.

In the command mode, BASIC does not take your input until you complete the logical line by pressing <ENTER>. This is called "line input", as opposed to "character input".

Interpretation of an Input Line

BASIC always ignores leading spaces in the line--it jumps ahead to the first non-space character. If this character IS NOT a digit, BASIC treats the line as an immediate line. If it IS a digit, BASIC treats the line as a program line.

For example:

```
   Ready
   PRINT "THE TIME IS " TIME$   <ENTER>
```

BASIC takes this as an immediate line.

If you type:

```
   Ready
   10 PRINT "THE TIME IS " TIME$  <ENTER>
```

BASIC takes this as a program line.

Immediate Line

An immediate line consists of one or more statements separated by colons. The line is executed as soon as you press <ENTER>. For example:

```
Ready
CLS: PRINT "THE SQUARE ROOT OF 2 IS" SQR(2)
```

is an immediate line. When you press <ENTER>, BASIC executes it.

Program Line

A program line consists of a line number in the range <0;65535>, followed by one or more statements separated by colons. When you press <ENTER>, the line is stored in the program text area of memory, along with any other lines you have entered this way. The program is not executed until you type RUN or another execute command. For example:

```
100 CLS: PRINT "THE SQUARE ROOT OF 2 IS" SQR(2)
```

Is a program line. When you press <ENTER>, BASIC stores it in the program text area. To execute it, type:

```
RUN  <ENTER>
```

Special Keys in the Command Mode

<?> When used in an immediate line, the question mark can stand for the commonly used keyword PRINT. For example, the immediate line:

```
? "HELLO."
```

is the same as the immediate line:

```
PRINT "HELLO."
```

Note: "L?" does NOT mean "LPRINT".

<.> The period can stand for "current program line", i.e., the last program line entered or edited. The period can be used in most places where a line number would normally appear. For example, the immediate line:

```
LIST.
```

tells BASIC to list the current program line.

<'> The single-quote tells BASIC to ignore the rest of the logical line. It is an abbreviation for the BASIC keyword REM. When used in a multi-statement line, it does not have to be preceded by a colon. For example, when you type in the line:

```
PRINT 1+1 ' 2+2
```

BASIC will Print the sum 1+1 but not 2+2.

Execute Mode

Whenever BASIC is executing statements (immediate lines or programs) it is in the execute mode. In this mode, the contents of the Video Display are under program control.

Special Keys in Execute Mode

<HOLD>	Pauses execution. Press again to continue.
<BREAK>	Terminates execution and returns you to the command mode.

Edit Mode

BASIC includes a line editor for correcting command or program lines. You can also use it to correct keyboard input to an INPUT statement.

To edit an immediate line, press <F1> BEFORE you have pressed ENTER. To edit a program line, type in the command:

```
EDIT line number
```

where line number specifies the desired line.

When the editor is working on a program line, it displays the number of the line being edited. When the editor is working on an immediate line or a line being input TO an INPUT statement, it displays a ! symbol in the first column on the line.

In the edit mode, Keyboard input is character-oriented, rather than line-oriented. That is, BASIC takes a specified number of characters as soon as they are typed in--without waiting for you to press <ENTER>.

See "Using the Line Editor" for details.

Using the Keyboard

BASIC has two ways of inputting data from the keyboard:

- Line Input--BASIC does not take the input until you press <ENTER>.
- Character Input: BASIC takes a specified number of characters without waiting for you to press <ENTER>.

In the command mode, BASIC uses line input. In the edit mode, it uses character input. Both types of input are available in the execute mode. See INPUT, INPUT$, LINE INPUT, INKEY$.

Keyboard Line Input

When you type number, letter, and punctuation keys, BASIC inputs them into the current line. Certain other keys and key combinations have special meanings to BASIC. Control keys not mentioned below are ignored during line input.

<BACKSPACE> Backspaces the cursor, erasing the preceding character in the line. Use this to correct typing errors. <CTRL-H> is the same code.

<SPACEBAR> Enters a blank space character and advances the cursor.

<F1> Puts you in the Edit Mode. The current line will be edited. See "Using the Line Editor." <CTRL-A> is the same code.

<BREAK> Interrupts line entry and starts over with a new line. <CTRL-C> is the same code. <BREAK> is echoed to the Display as <carat> C.

<TAB> Advances the cursor to the next 8-character boundary. Tab positions are at 0,8,16,24,... Use this for indenting program lines. <CTRL-I> is the same code.

<CTRL-J> Line feed--starts a new physical line without ending the current logical line.

(Keyboard Line Input, continued)

<CTRL-O> Toggles (switches the state of) the Display function, i.e., turns it on or off.
If the Display is on, <CTRL-O> turns it off. Subsequent characters typed will not not be echoed to Display, but will be input into the current line. Any programmed output to the Display will also be ignored.

If the Display function is off, <CTRL-O> turns it on. Subsequent characters typed will be echoed to the Display.

Whenever BASIC enters the command mode, it turns on the Display function.

<CTRL-O> is echoed as <carat> O.

<CTRL-R> Retypes the currrent logical line.

<CTRL-U> Restarts the current logical line (though the old line remains on the Display). The key is echoed to the display as <carat> U.

<ENTER> Ends the current logical line. BASIC will take the line.

<REPEAT> For convenience when you want to repeat a single key, hold down <REPEAT> while pressing the desired key. For example, to backspace halfway across the Display, hold down <REPEAT> and <BACKSPACE>.

Keyboard Character Input

In this mode, key input is not echoed to the display. Any key you press is accepted as input, except for <BREAK>, which interrupts the input and return you to the command mode.

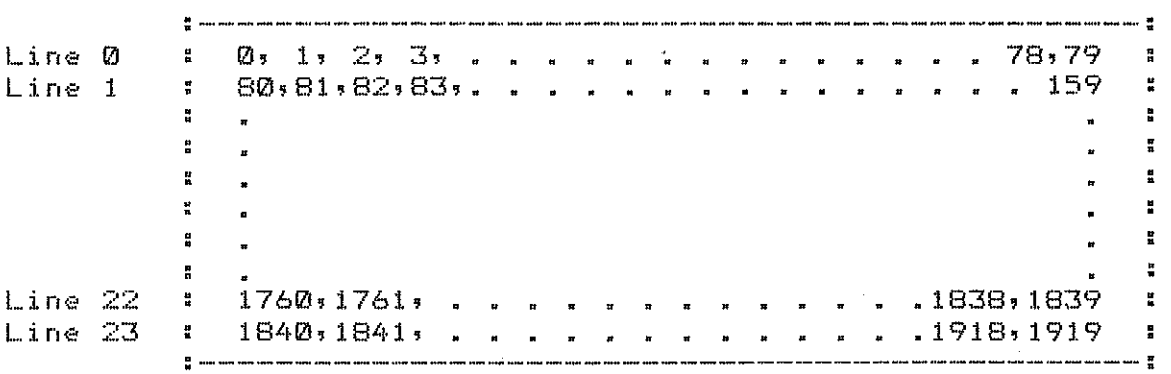
Using the Video Display

Model II BASIC gives you easy access to the Video Display's full character set, including all standard ASCII symbols and 32 special graphics codes. Every character can also be displayed in reverse (black on white).

The Display has two modes of operation--Scroll and Graphics. Cursor motion and position-labeling are different in the two modes.

Scroll Mode

In the Scroll Mode, the Display can be thought of as a sequence of 1920 display positions, as illustrated below:



DISPLAY POSITIONS, SCROLL MODE

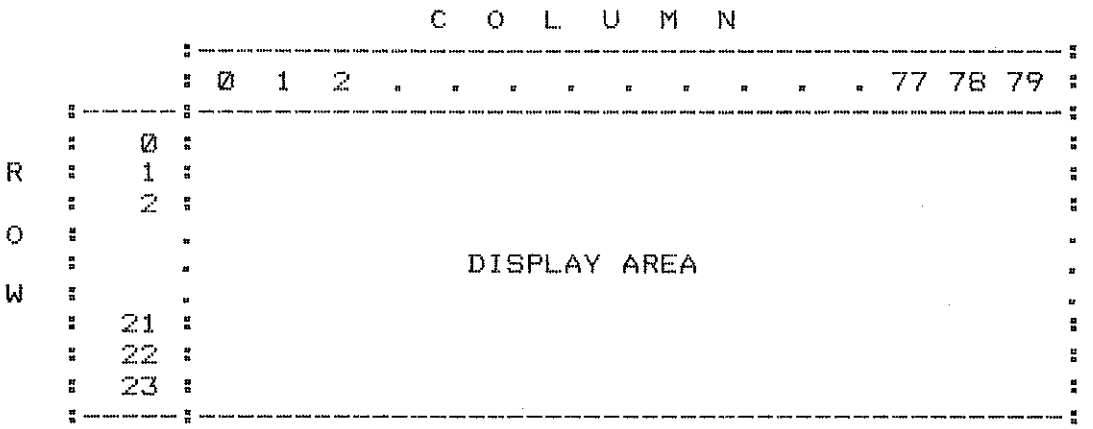
In scroll mode output, each time an acceptable display character is received, it is displayed at the current cursor position, and the cursor advances to the next higher numbered position.

When the cursor is on the bottom line and a line-feed or carriage return is received, or when the bottom line is filled, the entire Display is "scrolled":

- Line 0 is deleted
- Lines 1-23 are moved up one line
- Line 23 is blanked
- The cursor is set to the beginning of line 23.

Graphics Mode

In the Graphics Mode, the Display can be thought of as an 80 by 24 matrix, as illustrated below:



DISPLAY POSITIONS, GRAPHICS MODE

In Graphics mode output, the cursor "wraps" the display whenever it moves beyond the row or column boundaries. That is:

Current position	Direction	New position
column 79	forward	column 0, same row
column 0	back	column 79, same row
row 23	down	row 0, same column
row 0	up	row 23, same column

Video Display Output

All output to the Display is done via PRINT statements. To send actual codes to the Display, use the CHR$ function.

For example:

```
PRINT CHR$(26)
```

Sends code 26 to the Display, which sets the reverse mode.

The tables below summarize the Model II BASIC Display codes.

Display Control Codes

Code		Display Function
Decimal	Hexadecimal	
1	01	Turns on cursor
2	02	Turns off cursor
8	08	Backspaces cursor and erases
9	09	Tabs cursor to next 8-character boundary
10	0A	Line feed. Moves cursor down one row without changing column position.
13	0D	Moves cursor to start of next line
23	17	Erases to end of line, cursor doesn't move
24	18	Erases to end of screen, cursor doesn't move
25	19	Sets normal (white on black) display mode
26	1A	Sets reverse (black on white) display mode
27	1B	Erases screen and homes cursor (position 0)
28	1C	Scroll mode cursor motion: Moves cursor back one position without erasing.
29	1D	Scroll mode cursor motion: Moves cursor forward one position; if old position = 1919, display is scrolled up one line and new position = 1840.
252	FC	Graphics mode cursor motion: Moves cursor back one column; column=column-1. If column=0, new column=79. row is unchanged.

(Display control codes, continued)

Code		Display Function
Decimal	Hexadecimal	
253	FD	Graphics mode cursor motion: Moves cursor forward one column; column=column+1. If column =79, new column=0
254	FE	Graphics mode cursor motion: Moves cursor up one row; row= row-1; if row=0, new row =23
255	FF	Graphics mode cursor motion: Moves cursor down one row; row= row+1; if row=23, new row =0

Graphics Characters are codes 128-159. To see them, run the program:

```
10 FOR I=128 TO 159
20   PRINT I; CHR$(I),
30 NEXT
```

Standard ASCII characters (upper and lowercase letters, numbers and punctuation) are codes 32 to 127. To see them, run the program:

```
40 FOR I=32 TO 127
50   PRINT I; CHR$(I),
60 NEXT
```

(M2BASIC2 8/9/79)

2 / BASIC Concepts

This section contains the background information you'll need to write programs in Model II BASIC. It describes the types of data (information) BASIC can handle, and the operations BASIC can perform on the data.

Program

A program consists of one or more numbered logical lines, each line consisting of one or more BASIC statements. BASIC allows line numbers from 0 to 65529 inclusive. The program lines can include up to 255 total characters including the line number, and may be broken into two or more physical lines.

For example, here is a program:

```
line      BASIC       colon between     BASIC
number    statement   statements        statement

   100    CLS:  PRINT CHR$(26) "THIS IS REVERSE MODE"
   110    FOR I=1 TO 10000: NEXT I    'DELAY LOOP
   120    PRINT CHR$(25);:
   130    CLS: PRINT "THIS IS NORMAL MODE"
```

When BASIC executes a program, it handles the statements one at a time, starting at the first and proceding to the last. Some statements allow you to change this sequence. (See Program Sequence Statements.)

Statement

A statement is a complete instruction to BASIC, telling the Computer to perform some operations. If the operations involve data, the statement may include that, too. For example,

```
PRINT "THE SQUARE ROOT OF 2 IS" SQR(2)
```

is a complete statement. The number 2 is the data, and the operations are:

- Displaying the message in quotes
- Computing the square root of 2
- Displaying the resultant value

Data

BASIC can handle two kinds of data:

- Numbers, representing quantities and subject to standard mathematical operations
- Strings, representing sequences of characters and subject to special non-mathematical string operations

Each kind of data has its own memory storage requirement and its own range of values.

Numeric Data

BASIC allows three types of numbers: integer, single-precision and double-precision. You can declare the type of a number, or let BASIC assign a type. Each type serves a specific purpose in terms of precision, speed in arithmetic operations, and range of possible values.

Integer Type

To be stored as an integer type, a number must be whole and in the range <-32768,32767>. An integer value requires two bytes of memory for storage. Arithmetic operations are fastest when both operands are integers.

For example:

1 32000 -2 500 -12345

can all be stored as integers.

Single-Precision Type

Single-precision numbers can include up to 7 significant digits, and can represent normalized values* with exponents up to +/-38, i.e., numbers in the range:

<-1 x 10^38, -1 x 10^-38> <1 x 10^-38, 1 x 10^38>

A single-precision value requires 4 bytes of memory for storage. BASIC assumes a number is single precision if you do not specify the level of precision.

*In this reference manual, normalized value is one in which exactly one digit appears to the left of the decimal point. For example, 12.3 expressed in normalized form is 1.22 x 10.

For example:

10.001 -200034 1.774E6 6.024E-23 123.4567

can all be stored as single precision values.

NOTE

When used in a decimal number, the symbol E stands for "single-precision times 10 to the power of..."

Therefore 6.024E-23 represents the single-precision value 6.024 × 10^-23

Double-Precision Type

Double-precision numbers can include up to 17 significant digits, and can represent values in the same range as that for single-precision numbers. A double-precision value requires 8 bytes of memory for storage. Arithmetic operations involving at least one double-precision number are slower than the same operations when all operands are single-precision or integer.

For example:
1010234578 -8.7777651010 3.14159265358979 8.00100708D12
can all be stored as double-precision values.

NOTE

When used in a decimal number, the symbol D stands for "double-precision times 10 to the power of..." Therefore 8.00100708 D12 represents the value 8.00100708 × 10^12

String Data

Strings (sequences of characters) are useful for storing non-numeric information such as names, addresses, text, etc. Any ASCII character can be stored in a string. For example, the data:

Jack Brown, Age 38

can be stored as a string of 18 characters. Each character (and blank) in the string is stored as an ASCII code, requiring one byte of storage. The above string would be stored internally as:

Hex Code-	4A	61	63	6B	20	42	72	6F	77	6E	2C	20	41	67	65	20	33	38
	J	a	c	k		B	r	o	w	n	,		A	g	e		3	8

A string can be up to 255 characters long. Strings with length zero are called "null" or "empty".

Data Constants

All data is input to a program in the form of constants--values which are not subject to change. For example, the statement:

```
PRINT  "1 PLUS 1 EQUALS"  2
```

contains one string constant,

```
1 PLUS 1 EQUALS
```

and one numeric constant,

```
2
```

In this example, the constants serve as "input" to the PRINT statement--telling it what values to print on the Display.

Typing of Constants

When BASIC encounters a data constant in a statement, it must determine the type of the constant (string, integer, single-precision or double-precision). Here are the rules it uses:

I. If the value is enclosed in double-quotes, it is a string. For example, in the statements:

```
A$ = "Yes"
B$ = "3331 Waverly Way"
PRINT "1234567890"
```

the values in quotes are automatically categorized as strings. (A$ and B$ are variables, as explained later in this section.)

II. If the value is not in quotes, it is a number.* For example, in the statements:

```
A = 123001
B = 1
PRINT 12345, -7.32145 E6
```

all the data is numeric.

*There are exceptions to this rule. See DATA, INPUT, LINE INPUT, INKEY$, and INPUT$.

III. Whole numbers in the range <-32768,32767> are integers. For example, the statements:

```
A = 12350
B = -12
PRINT 10012, -21000
```

contain integer constants only.

IV. Numbers which are not integer type and which contain seven or fewer digits are single-precision. For example, the statements:

```
A = 1234567
B = -1.23
PRINT 11000.25
```

all the numbers are single-precision.

V. If the number contains more than eight digits, it is double-precision. For example, in the statements:

```
A = 123456789O1234567
B = -10000000000
PRINT 2.777000321
```

all the numbers are double-precision.

Type Declaration Characters

You can override BASIC's normal typing criteria by adding the following "tags" to the end of the numeric constant:

! Makes the number single-precision. For example, in the statement:

```
A = 12.345678901234!
```

the constant is classified as single-precision, and rounded to seven digits: 12.34568

Makes the number double-precision. For example, in statement:

```
PRINT 3# / 2
```

the first constant is classified as double-precision before the division takes place.

(Addition and the other operations are described later in this section.)

Hexadecimal and Octal Constants

Model II BASIC allows two additional types of constants, hexadecimal and octal numbers.

Hexadecimal numbers are quantities represented in base 16 notation, composed of the numerals 0-9 and the letters A-F. Hexadecimal constants must be in the range <0,FFFF>. They are stored as two-byte integers, corresponding to decimal integers as follows:

Hexadecimal Range	Equivalent Decimal Range
<0,7FFF>	<0,32767>
<8000,FFFF>	<-32768,-1>

Any number preceded by the symbol &H is interpreted as a hexadecimal constant. For example:

```
&HA010   &HFF   &HD1    &H10    &H0D
```

are all hexadecimal constants.

Octal numbers are quantities represented in base 8 notation, composed of the numerals 0-7. Octal constants must be in the range <0,177777>. They are stored as two-byte integers, corresponding to decimal integers as follows:

Octal Range	Equivalent Decimal Range
<0,77777>	<0,32767>
<100000,177777>	<-32768,-1>

Any number preceded by the symbol &O or & is interpreted as an octal constant. For example:

```
&70     &044    &10077  &123407
```

are all octal constants.

Variables

A variable is a place in memory--a sort of box or pigeonhole--where data can be stored. Unlike a constant, a variable's value can change. This allows you to write programs dealing with changing quantities.

Variable Names

In BASIC, variables are represented by names. Variable names must begin with a letter, A through Z. This letter may be followed by a digit, 0 through 9, or another letter.

For example:

A AA A2 B7 MJ

are all valid and distinct variable names.

Variable names may be longer than two characters. However, only the first two characters are significant in BASIC.

For example:

SU SUM SUPERNUMERARY

are all treated as the SAME variable by BASIC.

Reserved Words

Certain combinations of letters are reserved as BASIC keywords, and cannot be used in variable names. For example:

OR LAND NAME LENGTH MIFFED

cannot be used as variable names, because they contain the reserved words OR, AND, NAME, LEN, and IF, respectively.

See the Appendix for a list of reserved words.

Types of Variables

As with constants, there are four types of variables. The first three are numeric: integer, single-precision and double-precision; the fourth is string.

Depending on its type, one variable can contain values from only one of these groups.

The first letter of the variable name determines what the type is. Initially, all letters A through Z have the single-precision attribute. This means that all variables are single-precision (that is, they can only hold single-precision values).

For example:

A B X1 CY TRS H4

are all single-precision variables initially.

However, you can assign different attributes to any of the letters, by means of DEFINT (define-integer), DEFDBL (define double-precision), and DEFSTR (define-string) statements.

For example:

```
DEFSTR L
```

makes all variables which start with L into string variables. After the above statement, the variables:

```
L       L1      LL      LØ
```

can all hold string values, and only string values.

Type Declaration Tags

You can always override the type of a variable name by adding a type declaration tag at the end. There are four type declaration tags:

%	Integer
!	Single-precision
#	Double-precision
$	String

For example:

```
I%      FT%     NUM%    COUNTER%
```

are all integer variables, REGARDLESS of what attributes have been assigned to the letters I, F, N and C.

```
T!      RY!     QUAN!   PERCENT!
```

are all single-precision variables, REGARDLESS of what attributes have been assigned to the letters T, R, Q and P.

```
X#      RR#     PREV#   LASTNUM#
```

are all double-precision variables, REGARDLESS of what attributes have been assigned to the letters X, R, P and L.

```
Q$      CA$     WRD$    ENTRY$
```

are all string variables, REGARDLESS of what attributes have been assigned to the letters Q, C, W and E.

Note that any given variable name can represent four different variables. For example:

```
A5#     A5!     A5%     A5$
```

are all valid and DISTINCT variable names.

One further implication of type declaration: Any variable name used without a tag is equivalent to the same variable name used with one of the four tags. For example, after the statement:

```
DEFSTR C
```

the variable referenced by the name C1 is identical to the variable referenced by the name C1$.

Array Variables

BASIC allows subscripted variables or arrays. An array name references a list of values, or elements, instead of a single

element.

The array can have one or more dimensions. Each dimension is specified by a subscript. Array subscripts ALWAYS start with zero. Therefore the statement:

```
DIM A(12, 10)
```

creates an array A with 143 elements arranged in 13 rows of 11 columns.

```
A(5, 7)
```

refers to the element at row 5, column 7 in array A.

See the DIM statement description for more information.

Data Conversion

Often it is necessary to convert a value from one type to another type. BASIC will perform many conversions automatically; other conversions require that you use special conversion functions.

For example, suppose you want to add two numbers:

```
1 + 1.2345678901234567
```

The first number is an integer constant; the second, a double-precision constant. Because of different storage formats for the two types, the operation is physically impossible until one of the numbers is converted to match the other's type.

According to rules described later, BASIC converts the 1 to double precision. Then the two double-precision numbers can be added to produce a double-precision result.

What concerns us here is not the addition, or the rule for deciding which number is converted. Here we are only interested in the conversion itself.

Illegal Conversions

BASIC cannot automatically convert numeric values to string, or vice versa. For example, the statements:

```
A$ = 1234
A# = "1234"
```

are illegal. (Use STR$ and VAL to accomplish such conversions.)

Legal Conversions

BASIC can convert any numeric type into any other numeric type. For example:

```
A# = A%      Integer to double-precision
A! = A#      Double-precision to single-precision
A! = A%      Integer to single-precision
```

Rules for Conversion

Single or double-precision to integer type:

BASIC returns the largest integer that is not greater than the original value.

Note: The original value must be greater than or equal to -32768, and less than 32768.

Examples:

```
    A% = 32767.9
```

Assigns A% the value 32767.

```
    A% = 2.5D3
```

Assigns A% the value 2500.

```
    A% = -123.45678901234567
```

Assigns A% the value -124.

```
    A% = -32768.1
```

Produces an Overflow Error (out of integer range).

Integer to single- or double-precision:

No error is introduced. The converted value looks like the original value with 7 or 17 zeros to the right of the decimal place.

```
        A# = 32767
```

Stores 32767.00000000000 in A#.

```
        A! = -1234
```

Stores -1234.000 in A!.

Double- to single-precision:

This involves converting a number with up to 17 significant digits into a number with no more than 7. BASIC chops off (truncates) the 10 least significant digits, and performs 4/5 rounding on the least significant digit of the converted number.

That is, if the most significant digit (MSD) of the chopped-off portion is less than 5, then the least significant digit (LSD) of the remaining portion is left unchanged. But if the MSD of the chopped off portion is greater than 4, BASIC adds 1 to the LSD of the remaining portion.

Examples:

```
        A! = 1.2345678901234567
```

Stores 1.234568 in A!

Note: The statement:

```
        PRINT A!
```

Will display the value 1.23457, because only six digits are

displayed. The full seven digits are stored in memory.

```
        A! = 1.333333333333333
 Stores 1.333333 in A!.

        A! = 10000095
```

Stores 1000010 in A!, though only the first 6 digits can be displayed via the PRINT statement.

Single- to double-precision:

To make this conversion, BASIC simply adds trailing zeros to the single-precision number. If the original value has an exact binary representation in single-precision format, no error will be introduced. For example:

```
        A# = 1.5
```

Stores 1.5000000000000 in A#, since 1.5 does have an exact binary representation.

However, for numbers which have no exact binary representation, an error is introduced when zeroes are added. For example:

```
        A# = 1.3
```

Stores ↑1.2999999523162841 in A#.

Because most fractional numbers do not have an exact binary representation, you should keep such conversions out of your programs. For example, whenever you assign a constant value to a double-precision variable, you can force the constant to be double-precision:

```
        A# = 1.3#       or   A# = 1.3D
```

Both store 1.3 in A#.

Here is a special technique for converting single-precision to double-precison, without introducing error into the double-precision value. It is useful when the single-precision value is stored in a variable.

Take the single-precision variable, convert it to a string with STR$, then convert the resultant string back into a number with VAL. That is, use

```
         VAL(STR$(single-precision variable)).
```

For example, compare the following program:

```
        10 A! = 1.3       'Single-Precision
        20 A# = A!        'Single->Double
        30 PRINT A#
```

prints a value of:

```
         1.2999999523162841
```

Compare with this program:

```
        40 A! = 1.3       'Single-Precision
        50 A# = VAL(STR$(A!))    'Special conversion technique
        60 PRINT A#
```

which prints a value of:

```
         1.3
```

The conversion in line 50 causes the value in A! to be stored accurately in double-precision variable A#.

(M2BASIC3 8/9/79)

Operations

An operation instructs the Computer to do something.

There are two levels of operations:
- .Statements, which are complete instructions
- .Expressions, which serve as parameters and data for statements

Statements

Statements tell the Computer to perform some action. Statements are complete in themselves. Once the statement has been written, no other information needs to be added to the statement for it to be executed.

For example, the statement:

```
DEFINT N-R
```

Is complete as it stands.

A statement is made up of a keyword* followed by whatever parameters or data are needed. The data is usually represented by an expression (defined below).

* A keyword is any sequence of characters which has a predefined meaning for BASIC. "PRINT", "INPUT", and "SQR" are all examples of keywords.

For example:

```
PRINT "MODEL II"
```

Tells BASIC to display the message inside quotes. PRINT is the keyword; "MODEL II", the data.

```
LIST 100-130
```

Tells BASIC to list the resident program lines in the range 100-130. LIST is the keyword; 100-130, the parameter.

```
A1 = 5 * A / 3
```

Tells BASIC to give A1 the value of the expression on the right of the equals sign.

Expressions

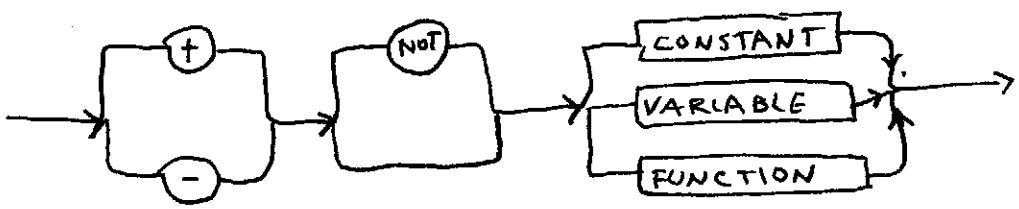
The concept of an expression is important in this manual, since it is used in most of the syntax descriptions. Throughout these descriptions, you will encounter the terms "numeric expression", "string expression", "logical expression", etc. Understanding the concept will allow you to grasp the full potential of BASIC's operations.

Expressions are composed of
- .Constants
- .Variables
- .Operators
- .Functions

A simple expression consists of a single term: a constant, variable, or function preceded by an optional + or - sign or the logical operator NOT.

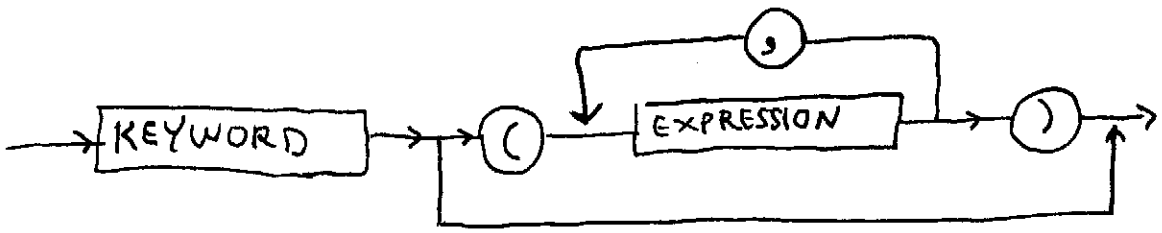
Note: For simplicity, expression and term definitions do not necessarily conform to standard computer usage.

For example:
```
    i    A1    -33.565    1.2345-E5    Z#    SQR(3)    NOT Ø
```
Here's a picture defining a TERM (items in square boxes are defined elsewhere):



A function consists of a keyword usually followed by an argument list in parentheses. Each of the arguments can be an expression. For example:
```
    SQR(2.5+A)    TAN(Y)    CINT(X#)    MID$(A$,3,N)
```

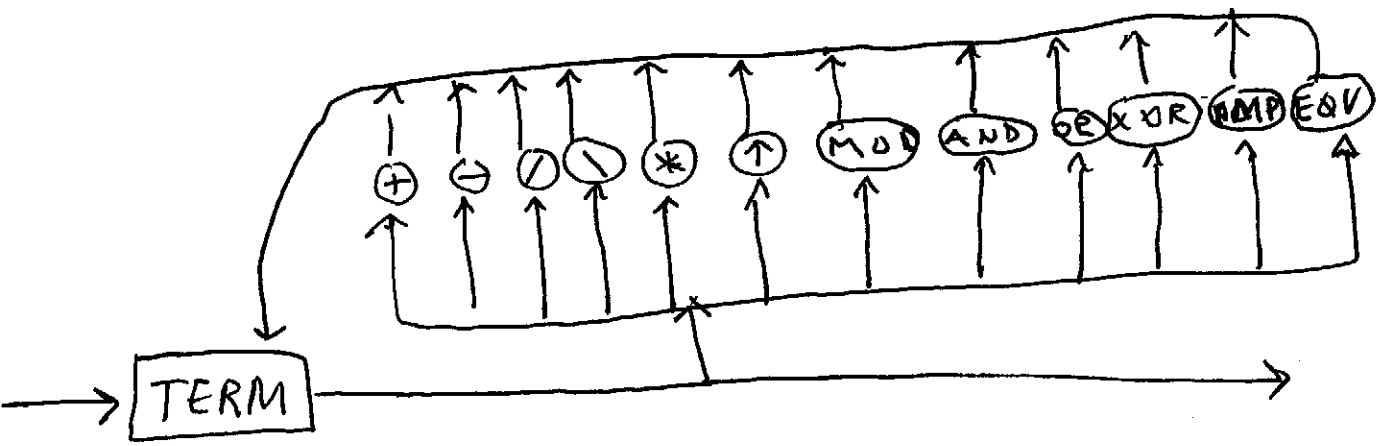
Here's a picture defining a FUNCTION.



In general, an expression consists of one term or two or more terms combined by operators (defined below). For example:

A + 1 A / B C * D + E + 3.5

Here's a picture defining a COMPLEX EXPRESSION:



Operators

An operator is a single symbol or word which signifies some action to be taken on one or two specified values referred to as operands.

For example:

```
5^2
```

The operator ^ connects or relates its two operands, the numbers 5 and 2, and indicates exponentiation, 5 to the power of 2.

Operators fall into three categories:

- .Numeric
- .Logical
- .String

In the descriptions below, we use the terms integer operation, single-precision operation, and double-precision operation. The importance is that integer operations involve two-byte operands; single-precision, four-byte operands; and double-precision, eight-byte operands. The more bytes involved in an operation, the slower the operation.

Numeric Operators

There are nine different numeric operators. Two of them, sign + and sign -, are unary, that is, they have only one operand. A sign operator has no effect on the precision of its operand.

For example, in the statement:

```
PRINT -77, +77
```

the sign operators - and + produce the values negative 77 and positive 77, respectively.

Note: When no sign operator appears in front of a numeric term, + is assumed.

The other numeric operators are all binary, that is, they all take two operands. These operators are

+	Addition
-	Subtraction
*	Multiplication
/	Division
\	Integer division (keyboard character <CTRL 9>
^	Exponentiation (keyboard character <SHIFT 6>
MOD	Modulus arithmetic

Addition

The + operator is the symbol for addition. The addition is done with the precision of the more precise operand (the less precise operand is converted).

For example, when one operand is integer type and the other is single precision, the integer is converted to single-precision and four-byte addition is done. When one operand is single-precision and the other is double-precision, the single-precision number is converted to double-precision and eight-byte addition is done.

Examples:

```
PRINT 2 + 3
```

Integer addition.

```
PRINT 3.1 + 3
```

Single-precision addition.

```
PRINT 1.2345678901234567 + 1
```

Double-precision addition.

Subtraction

The - operator is the symbol for subtraction. As with addition, the operation is done with the precision of the more precise operand (the less precise operand is converted).

Examples:

```
    PRINT 33 - 11
```

Integer subtraction.

```
    PRINT 35 - 11.1
```

Single-precision subtraction.

```
    PRINT 12.3456789Ø1234567 - 11
```

Double-precision subtraction.

Multiplication

The * operator is the symbol for multiplication. Once again, the operation is done with the precision of the more precise operand (the less precise operand is converted).

Examples:

```
    PRINT 33 * 11
```

Integer multiplication.

```
    PRINT 33.1 * 11
```

Single-precision multiplication.

```
   PRINT 1.2345678901234567 * 11
```

Double-precision multiplication.

Division

The / symbol is used to indicate ordinary division. Both operands are converted to single or double-precision, depending on their original precision:

- If either operand is double-precision, then both are converted to double-precision and eight-byte division is performed.
- If neither operand is double-precision, then both are converted to single-precision and four-byte division is performed.

Examples:

```
    PRINT 3/4
```

Single-precision division.

```
    PRINT 3.Ø/4
```

Single-Precision division.

```
    PRINT 3 / 1.2345678901234567
```

Double-precision division.

Integer Division

The integer division operator \ converts its operands into integer type, then performs integer division, in which the remainder after

division is ignored, leaving an integer result. (If either operand is outside the range <-32768,32767>, an error will occur.)

Note: To enter the \ operator, press <CTRL-9.>

For example:

```
PRINT 7 \ 3
```

prints the value 2, since 7 divided by 3 equals 2 remainder 1.

Exponentiation

The symbol ^ (read: carat) denotes exponentiation. It converts both its operands to single-precision, and returns a single precision result.

Note: To enter the ^ operator, press <SHIFT-6>.

For example:

```
PRINT 6 ^ .3
```

prints 6 to the .3 power.

Modulus Arithmetic

The MOD ("modulo") operator allows you to do modulus arithmetic, i.e., arithmetic in which every number is converted to its equivalent in a cyclical counting scheme. For example, a 24-hour clock indicates the hour in modulo 24: although the hour keeps incrementing, it is always expressed as a number from 0 to 23.

MOD requires two operands, for example:

```
A MOD B
```

B is the modulus (the counting base) and A is the number to be converted.

(Expressed in mathematical terms, A MOD B returns the REMAINDER after whole-number division of A by B. In this sense, it is the converse of \, which returns the WHOLE NUMBER QUOTIENT and ignores the remainder.)

MOD converts both operands to integer type before performing the operation. If either operand is outside the range <-32768,32767> an error will occur.

Examples:

```
PRINT 155 MOD 15
```

Prints 5, since 155/15 gives a whole number quotient of 10 with remainder 5.

```
PRINT 79 MOD 12
```

Prints 7, since 78/12 equals 6 with remainder 7.

```
10 INPUT "TYPE IN AN ANGLE IN DEGREES"; A%
20 PRINT A% " = " A% \ 90 " * 90 + " A% MOD 90
```

Input a positive angle greater than 90. Line 20 expresses the angle

as a multiple of 90 degrees plus a remainder.

The table below summarizes the precision of operations for all numeric operators. (I = integer; S = single-precision; D = double-precision.)

Important: For effects of conversions on accuracy, see "Data Conversion".

Operator	Operand(s)	Value Returned
+, -, *	II	I
	IS SS	S
	ID SD DD	D
/	II IS SS	S
	ID SD DD	D
\	All possible combinations	I
^	All possible combinations	S
MOD	All possible combinations	I
+ (sign) and - (sign)	I	I
	S	S
	D	D

Logical Operators

Logical operators deal with True/False conditions, comparisons, and tests. They allow you to build elaborate decision-making structures into programs, to perform bit manipulations, to sort data, etc.

All logical operators convert their operands to two-byte integers. If an operand is outside of the range <-32768,32767> an error will occur.

The logical operators include the three relational operators:

< > =

and six Boolean word-operators:

AND OR XOR NOT IMP EQV

Note: An expression involving a logical operator is called a logical expresion.

Relational Operators

Relational operators compare two operands for numerical precedence. Here is a table of the relational operators and their various combinations:

<	Less than
>	Greater than
=	Equal to
>< or <>	Not equal to
=< or <=	Less than or equal to
=> or >=	Greater than or equal to

Relational operators can return only two possible values: True or False. Actually, BASIC returns the number -1 to indicate True, and 0 to indicate False. But the quantity (-1 or 0) is rarely used as a number. More often, it is used as a decision-making operator, as in the line:

```
IF A = B THEN GOTO 1000  ELSE  END
```

The logical expression A = B returns negative one (-1) when A equals B, and zero when A does not equal B. But you don't care about the numbers -1 and 0. What matters to you is that if the expression is True, GOTO 1000 is executed; otherwise BASIC ENDs the program.

Here's an example where the result of a logical expression IS used as a quantity:

```
MAX =  -(A < B) * B  -  (B <= A) * A
```

For any two integer type values A and B, MAX contains the larger of the two.

Note: All relational operators can also be used to compare strings for precedence. The result of such a comparison is still either a True (logical -1) or False (logical 0). See String Operators.

Boolean Operators

In this section, we will explain how Boolean operators are implemented in Model II BASIC. However, we will not try to explain Boolean algebra, decimal-to-binary conversions, binary arithmetic, and similar subjects. If you need to learn something about these topics, Radio Shack's Understanding Digital Electronics (Catalog Number 62-2010) and TRS-80 Assembly-Language Programming (Catalog Number 62-2006) are the books to start with.

Model II BASIC includes six Boolean operators:

AND	OR	XOR
EQV	IMP	NOT

All the Boolean operators relate two operands except for NOT, which acts on a single operand.

These operators can be used to set up decision structures. For this application, both operands are usually relational expressions and the operator is one of the following:
AND, OR, XOR, NOT.

AND
If both expressions are True, then AND returns a logical True. Otherwise it returns a logical False.

OR
If either of the expressions is True, or both are True, this operand returns a logical True. Otherwise it returns a logical False.

XOR ("Exclusive-OR")
Only when one of the expressions is True does OR return a logical True. Otherwise it returns a logical False.

NOT
NOT is a unary operator (acts on one operand). When the expression is True, NOT returns a logical False. When it is False, NOT returns a logical True.

Example

```
  (IF A <= 90 AND A >= 0) THEN PRINT "Value is okay."
```

Only if A is in the range <0,90> will BASIC print the "okay" message.

Bit Manipulation

For this application, both operands are usually numeric expressions. BASIC does a bit-by-bit comparison of the two operands, according to predefined rules for the specific operator.

Note: The operands are converted to integer type, stored internally as 16-bit, two's complement numbers. To understand the results of bit-by-bit comparisons, you need to keep this in mind.

The following table summarizes the action of Boolean operators in bit manipulation.

BOOLEAN OPERATIONS

Operator	Meaning of Operation	First Operand	Second Operand	Result
AND	When both bits are 1, the result will be 1. Otherwise, the result will be Ø.	1	1	1
		1	Ø	Ø
		Ø	1	Ø
		Ø	Ø	Ø
OR	Result will be 1 unless both bits are Ø.	1	1	1
		1	Ø	1
		Ø	1	1
		Ø	Ø	Ø
XOR (exclusive or)	Result will be 1 unless both bits are the same.	1	1	Ø
		1	Ø	1
		Ø	1	1
		Ø	Ø	Ø
EQV (equivalence)	Result will be 1 unless both bits are different.	1	1	1
		1	Ø	Ø
		Ø	1	Ø
		Ø	Ø	1
IMP (implication)	Result will be 1 unless first bit is 1 and second bit is Ø.	1	1	1
		1	Ø	Ø
		Ø	1	1
		Ø	Ø	1
NOT	Result is opposite of bit.	1	----	Ø
		Ø	----	1

As an example of bit manipulation, suppose you want to change lowercase characters to uppercase and uppercase characters to lowercase. You could do this by checking the ASCII code of each character and adding or subtracting decimal 32 (hexadecimal 20) depending on whether the character was uppercase or lower. But this routine could be done more simply, using only the operator XOR.

The ASCII codes for uppercase characters are decimal 65-90 (hexadecimal 41-5A); for lowercase, decimal 97-122 (hexadecimal 61-7A). Looking at these ranges in binary, you can see that all capital letters have a 0 in bit position 5, while all lowercase letters have a 1 in bit position 5.

Note: Position 7 is the most significant bit; position 0 is least significant, as illustrated below:

```
most significant                          least significant
            bit                                   bit

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

So, to convert from lower to uppercase and vice versa, you just toggle (reverse the state of) bit 5. Decimal 32 has the following binary representation:

```
        00100000
```

Notice that bit 5 is a 1; all others are zeroes. When you XOR decimal 32 with any number, you will effectively toggle bit 5. For letters, this will switch cases, upper to lower and vice versa.

For instance, since 72 is the ASCII code for "H":

```
   PRINT  CHR$(72 XOR 32)
```

prints a lowercase "h".

You can check this by consulting XOR in the table above and XOR-ing the two numbers by hand.

String Operators

There are seven string operators in Model II BASIC. These operators allow you to compare strings and to concatenate them (i.e., string them together).

The comparison operators for strings are the same as those for numbers, although their meanings are slightly different. Instead of comparing numerical magnitudes, the operators compare sorting precedence (i.e., alphabetical sequence).

<	Precedes
>	Follows
=	Has the same precedence
<>	Does not have the same precedence
<=	Precedes or has the same precedence
>=	Follows or has the same precedence

Comparison is made character by character on the basis of ASCII codes. When a non-matching character is found, the string containing the character with a lower ASCII code is taken as the smaller ("precedent") of the two strings. See the Appendix for an ASCII code table.

Examples:

```
"A" < "B"
```

The ASCII code for A is decimal 65; for B it's 66.

```
"COOL" > "CODE"
```

ASCII for O is 79; for D it's 68.

If, while comparison is proceeding, the end of one string is reached before any non-matching characters are found, the SHORTER string is considered to be the smaller. For example:

```
"TRAIL" < "TRAILER"
```

Leading and trailing blanks are significant. For example:

```
" A" < "A "
```

ASCII for " " (space) is 32; for A it's 65.

```
"Z-80" < "Z-80A"
```

The string on the left is four characters long; the string on the right is five.

Here are some examples of how you might use the string comparison operators in a program:

```
IF A$ <> B$ THEN END
```

If string A$ is not the same as B$, the program ends.

```
IF A$ < B$ THEN PRINT A$
```

If A$ alphabetically *precedes* B$, A$ is printed.

```
IF NME$ = "CARRUTHERS" OR CITY$ = "BUFFALO"
   THEN PRINT NME$, CITY$
```

If the value of NME$ is CARRUTHERS, then CARRUTHERS plus the current value of CITY$ will be printed, OR if the value of CITY$ is BUFFALO, then BUFFALO will be printed plus the current value of NME$.

The concatenation operator is represented by the symbol +. This operator takes two strings as its operands and returns a single string as its result by adding the string on the right of the + sign to the string on the left. If the new string is greater than 255 characters, a String Too Long error will occur.

For example:

```
PRINT "CATS " + "LOVE " + "MICE"
```

which returns

```
CATS LOVE MICE
```

Evaluation of Expressions

When an expression involves multiple operations, BASIC performs the operations according to a well-defined hierarchy, so that results are always predictable.

Parentheses

When a complex expression includes parentheses, BASIC always evaluates the expression inside the parentheses before evaluating the rest of the expression. For example, the expression:

```
        8 - ( 3 - 2 )
```

is evaluated like this:

```
        3 - 2 = 1
              8 - 1 = 7
```

With nested parentheses, BASIC starts evaluation at the innermost level and works outward. For example:

```
        4 * ( 2 - (3 - 4) )
```

is evaluated like this:

```
        3-4 = -1
                2 - -1 = 3
                         4 * 3 = 12
```

Order of Operations

When evaluating a sequence of operations on the same level of parenthesis, BASIC uses the following hierarchy to determine what operation to do first. Operators are shown below in decreasing order of precedence. Operators listed in the same entry in the table have the same precedence and are executed in order FROM LEFT TO RIGHT.

```
^          Exponentiation
+,-        Unary sign operands (NOT addition and subtraction)
*, /
\           Integer division
MOD
+, -       Addition and subtraction
<, >, =, <=, >=, <>
NOT
AND
OR
XOR
EQV
IMP
```

For example, in the line

```
        X * X + 5 ^ 2.8
```

BASIC will find the value of 5 to the 2.8 power. Next it will multiply X * X, and finally add this value to the value of 5 to the 2.8. If you want BASIC to perform the indicated operations in a different order, you must add parentheses, e.g.

```
        X * (X + 5 ^ 2.8)
```

or

```
X * (X + 5) ^ 2.8
```

Here's another example:

```
IF X = 0 OR Y > 0 AND Z = 1 THEN GOTO 255
```

The relational operators = and > have the highest precedence, so they will be performed first. Since they both have the same precedence, they will be performed one after another, left to right. Then the Boolean operations will be performed. AND has a higher precedence than OR, so the AND operation will be performed before the OR. Therefore, the line above means that if X = 0, or if Y > 0 and Z = 1, control switches to line 255.

If the line above looks confusing because you can't remember which operator is precedent over which, then you can use parentheses to make the sequence obvious:

```
IF X = 0  OR ((Y > 0) AND (Z = 1)) THEN GOTO 255
```

Type Conversions

During evaluation of an expression, BASIC often has to perform type conversions. Unless you're careful in forming expressions, these conversions can produce invalid results. For example, in the expression:

```
A# * C!
```

C! must be converted to double-precision before the multiplication can take place. This will usually introduce an error into the result.

Before evaluating the expression:

```
A + B^1.2345678
```

BASIC must convert 1.2345678 to to single-precision. You cannot expect double-precision from a single-precision operator or function.

See "Data Conversion" for details on the effects of type conversion on accuracy, and for special conversion techniques.

Functions

A function is a built-in subroutine. The functions supplied in Model II BASIC save you from having to write equivalent BASIC routines, and they operate faster than a BASIC routine would.

A function consists of a keyword followed by required input values, referred to as arguments or parameters. The arguments are always enclosed in parentheses and separated by commas. Some functions have no arguments; others require up to three. The quantity output or returned by a function is called the value of the function.

Examples:

```
SQR(A)
```

Tells BASIC to compute the square root of the quantity A. SQR is the keyword, and A is the argument.

```
MID$(A$, 3, 2)
```

Tells BASIC to return a substring of the string A$, starting with the third character, with length 2. MID$ is the keyword, and A$, 3 and 2 are its arguments or parameters.

Since functions are syntactically equivalent to expressions, they cannot stand alone in a BASIC program. They must be used in statements.

For example:

```
A = SQR (B)
```

Assigns A the value of square root of B.

```
PRINT MID$(A$, 3, 2)
```

Prints the substring of A$ starting at the third character and two characters long.

```
PRINT LOG(SQR(2))
```

Prints the natural logarithm of the square root of 2.

In this manual, functions are classified as numeric when they return a number, and string when they return a string.
Wherever the syntax calls for a numeric expression, you can use a numeric function; for a string expression, you can use a string function.

There is another special class of functions which return information about the allocation of memory and the location of various quantities in memory. For example:

```
MEM
```

Returns the number of bytes of memory available for storing program text, numeric and array variables.

Chapter 3

BASIC Keywords

PAGE NUMBERING RESUMES ON THE NEXT

PAGE WITH NUMBER 57.

There are no pages numbered 50,51,52,53,54,55,56.

A. Statements

Command Statements

Command statements tell BASIC to enter another operation mode or to perform various System functions (like loading a program from disk). Although they can be included inside a program, their primary use is outside of a program.

For example, the command statement

```
NEW
```

Erases the entire program currently in memory and zeroes all variables.

AUTO
Number Lines Automatically

AUTO *startline, increment*
startline is a line number specifying the first line number to be used. If *startline* is omitted, 10 is used. A period (".") can be substituted for *startline*. In this case, the current line number is used.
increment is a number specifying the increment to be used between lines. If *increment* is omitted, 10 is used.

AUTO turns on an automatic line numbering function. After you enter this command, BASIC will supply the *startline*. All you have to do is type in the text of the line and press **ENTER**. Then AUTO will display the next line number, using *increment* or a default increment of 10.

To turn off the AUTO function, press **BREAK** at any time. The current line will be cancelled.

Whenever AUTO provides a line number that is already in use, it will display an asterisk immediately after the line number. Press **BREAK** if you do not want to change that line.

Examples

```
AUTO
```

starts automatic numbering with line 10, using increments of 10 between line numbers.

```
AUTO 100
```

starts numbering with 100, using increments of 10 between line numbers.

```
AUTO 1000, 100
```

starts numbering with 1000, using increments of 100 between line numbers.

```
AUTO ,5
```

starts numbering with 0, using increments of 5 between line numbers.

```
AUTO .
```

starts numbering with the current line number, using increments of 10 between line numbers.

DELETE
Erase Program Lines from Memory

DELETE *startline-endline*

startline is a line number specifying the lower limit for the deletion. If *startline* is omitted, then the first line in the program is used as *startline*.

endline is a line number specifying the last line in your program that you want to delete. Endline must reference an existing program line.

A period (".") can be substituted for either *startline* or *endline*. The period signifies the current line number.

DELETE removes from memory the specified range of program lines.

Examples

```
DELETE 70
```

Erases line 70 from memory. If there is no line 70, an error will occur.

```
DELETE 50-110
```

Erases lines 50 through 110 inclusive.

```
DELETE -40
```

Erases all program lines up to and including line 40.

```
DELETE -.
```

Erases all program lines up to and including the line that has just been entered or edited.

```
DELETE.
```

Erases the program line that has just been entered or edited.

EDIT
Edit Program Line

EDIT *line number*

EDIT allows the specified line to be revised without affecting any other lines. The EDIT command has a powerful set of subcommands which are discussed in detail in the section on Program Editing and Debugging.

```
EDIT 100
```

Edits line 100

```
EDIT.
```

Edits the current line.

KILL
Delete File from Disk

KILL *filespec*

KILL deletes the specified file from the diskette directory.

If no drive specification is included in the filespec, BASIC will search for the first drive that contains the filespec, and attempt to delete that file.

Do not KILL an open file. CLOSE it first.

Example

```
KILL "FILE/BAS"
```

deletes this file from the first drive which contains it.

```
KILL "DATA:2"
```

deletes this file from drive #2.

LIST
Display Program Lines

LIST *startline-endline*
startline is a line number specifying the lower limit for the listing. If *startline* is omitted, then the first line in the program is used.
endline is a line number specifying the upper limit for the listing. If *endline* is omitted, the last line in the program is used.

A period (".") can be substituted for either *startline* or *endline*. The period signifies the current line number.

LIST instructs the Computer to display the specified range of program lines currently in memory. The arguments are optional.

Examples

```
LIST
```

Displays the entire program. To stop the automatic scrolling, press HOLD. This will freeze the display. Press any key to continue the listing.

```
LIST 50
```

Displays line 50.

```
LIST 50-85
```

Displays lines in the range 50-85.

```
LIST 227-
```

Displays line 227 and all higher-number lines.

```
LIST.-
```

Displays the program line that has just been entered or edited, and all higher-numbered lines.

```
LIST-227
```

Displays all lines up to and including 227.

```
LIST-.
```

Displays all lines up to and including the line that has just been entered or edited.

```
LIST.
```

Displays the line that has just been entered or edited.

LLIST
Print Program Lines

LLIST *startline-endline*
startline is a line number specifying the lower limit for the listing. If *startline* is omitted, then the first line in the program is used as *startline.*
endline is a line number specifying the upper limit for the listing. If *endline* is omitted, the last line in the program is used as *endline.*
A period (".") can be substituted for either *startline* or *endline.* The period signifies the current line number.

LLIST works like LIST, but its output is to the Printer rather than the Display. LLIST instructs the Computer to print the specified range of program lines currently in memory. The arguments are optional.

Examples

```
LLIST
```

Lists the entire program to the printer. To stop this process, press HOLD. This will cause a temporary halt in the Computer's output to the Printer. Press any key to continue printing.

```
LLIST 780
```

Prints line 780.

```
LLIST 68-90
```

Prints lines in the range 68-90.

```
LLIST 50-
```

Prints lines 50 and all higher-numbered lines.

```
LLIST.-
```

Prints the program line that has just been entered or edited plus all higher-numbered lines.

```
LLIST-50
```

Prints all lines up to and including 50.

```
LLIST-.
```

Prints all lines up to and including the line that has just been entered or edited.

```
LLIST.
```

Prints the line that has just been entered or edited.

LOAD
Load Basic Program File

LOAD "*filespec*" [,R]
R (optional) tells BASIC to RUN the program after it is loaded.

This command loads a BASIC program file into RAM. If the R option is used, BASIC will proceed to RUN the program automatically. Otherwise, BASIC will return to the command mode.

LOAD wipes out any resident BASIC program, clears all variables, and closes all open files unless the R option is used, in which case open files will not be closed.

LOAD with the R option is equivalent to the command RUN *filespec*, R. Either of these commands can be used inside programs to allow program chaining (one program calling another).

If you attempt to LOAD a non-BASIC file, a Direct Statement in File or Load Format error will occur.

Example

```
LOAD "PROG1/BAS:2"
```

This loads PROG1/BAS from drive 2 BASIC then returns to the command mode.

```
LOAD "PROG1/BAS"
```

Since no drive specification is included in this command, BASIC will begin searching for this program file in drive 0 and load the first one it finds with the name PROG1/BAS.

MERGE
Merge Disk Program with Resident Program

MERGE *filespec/txt*
filespec/txt is a BASIC file in ASCII format, e.g., a program saved with the A option.

The MERGE statement takes a BASIC program from disk and merges it with the resident BASIC program in RAM.

Program lines in the disk program are inserted into the resident program in sequential order. For example, if three of the lines from the disk program are numbered 75, 85, and 90, and three of the lines from the resident program are numbered 70, 80, and 100, when MERGE is used on the two programs, this portion of the new program will be numbered, 70, 75, 80, 85, 90, 100.

If line numbers in the disk program coincide with line numbers in the resident program, the resident lines will be replaced by those from the disk program. For example, if three of the lines from the disk program are numbered 5, 10, and 20, and three of the lines from the resident program are numbered 10, 20, and 30, when MERGE is used on the two programs, this portion of the new program will be numbered 5, 10, 20, 30. Lines 10 and 20 of the new program will be identical to lines 10 and 20 on the disk program.

MERGE closes all files and clears all variables. Upon completion, BASIC returns to the command mode.

Example

Let's say we have a BASIC program on disk, PROG2/TXT, which we want to merge with the program we've been working on in RAM. Then

```
MERGE "PROG2/TXT"
```

will do the job.

Sample Uses

MERGE provides a convenient means of putting modular programs together. For example, an often-used set of BASIC subroutines can be tacked onto a variety of programs with this command.

Suppose the following program is in RAM:

```
80 REM      MAIN PROGRAM
90 GOSUB 1000
100 REM      PROGRAM LINE
110 REM      PROGRAM LINE
120 REM      PROGRAM LINE
130 END
```

And suppose the following subroutine, SUB/TXT, is stored on disk in ASCII format:

```
1000 REM      BEGINNING OF SUBROUTINE
1010 REM      SUBROUTINE LINE
1020 REM      SUBROUTINE LINE
1030 REM      SUBROUTINE LINE
1040 RETURN
```

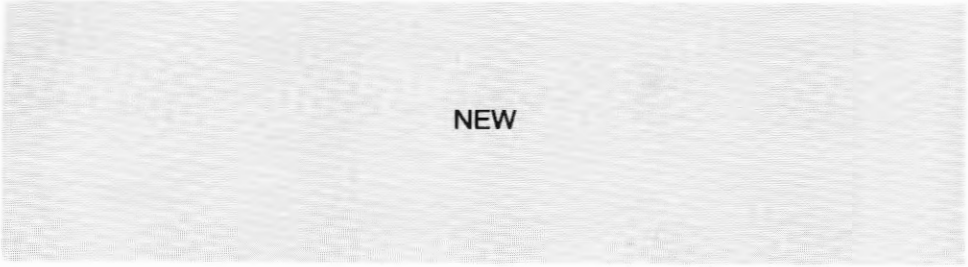
We can MERGE the subroutine with the main program using the statement

```
MERGE "SUB/TXT"
```

and the new program in RAM would be:

```
80 REM      MAIN PROGRAM
90 GOSUB 1000
100 REM      PROGRAM LINE
110 REM      PROGRAM LINE
120 REM      PROGRAM LINE
130 END
1000 REM      BEGINNING OF SUBROUTINE
1010 REM      SUBROUTINE LINE
1020 REM      SUBROUTINE LINE
1030 REM      SUBROUTINE LINE
1040 RETURN
```

NEW
Erase Program from Memory



NEW erases all program lines, sets numeric variables to zero and string variables to null.

Example

```
NEW
```

RENUM
Renumber Program

RENUM *newline, startline, increment*

newline specifies the new line number of the first line to be renumbered. If *newline* is omitted, the line number 10 is used.

startline specifies the line number in the original program where you want to start renumbering. If *startline* is omitted, the entire program will be renumbered.

increment specifies the increment to be used between each successive renumbered line. If *increment* is omitted, 10 is used.

RENUM changes all ine numbers in the specified range, as well as all linc number references appearing after GOTO, GOSUB, THEN, ON . . . GOTO, ON . . . GOSUB, ON ERROR GOTO, and ERL [relational operator] – throughout the program.

All the RENUM arguments are optional.

RENUM will add trailing blanks to line number references which contain fewer than 5 digits. These blanks will not accumulate during subsequent renumbering operations on the same program.

Examples

```
RENUM
```

Renumbers the entire resident program, incrementing by 10's. The new number of the first line will be 10.

```
RENUM 6000,5000,100
```

Renumbers all lines numbered from 5000 up. The first renumbered line will become 6000, and an increment of 100 will be used between subsequent lines.

```
RENUM 10000,1000
```

Renumbers line 1000 and all higher-numbered lines. The first renumbered line will become line 10000. An increment of 10 will be used between subsequent line numbers.

```
RENUM 100,,100
```

Renumbers the entire program, starting with a new line number of 100, and incrementing by 100's. Notice that the commas must be retained even though the middle argument is gone.

```
RENUM,,5
```

Renumbers the entire program, starting with a new line number of 10, and incrementing by 5's.

Error Conditions

1. RENUM cannot be used to change the order of program lines. For example, if the original program has lines numbered 10, 20 and 30, then the command

   RENUM 15,30

   is illegal, since the result would be to move the third line of the program ahead of the second. In this case, an FC (illegal function call) error will result, and the original program will be left unchanged.

2. RENUM will not create new line numbers greater than 65529. Instead, an FC error will result, and the original program will be left unchanged.

3. If an undefined line number is used inside your original program, RENUM will print a warning message, UNDEFINED LINE xxxx in yyyy, where xxxx is the original line number reference and yyyy is the original number of the line containing xxxx.

   Note that RENUM will renumber the program in spite of this warning message. It will replace the number xxxx with 5 blanks, and will renumber yyyy, according to the parameters in your RENUM command.

   For example, if your original program includes the line

   110 GOTO 1000

   but does NOT include a line 1000, then RENUM will print a warning,

   UNDEFINED 1000 in 110

   and renumber the program. The text of original line 110 will be changed to

   GOTO <five blanks here>

RUN
Execute Program

RUN *startline*

startline is a line number specifying where you want program execution to start. If *startline* is omitted, the first line in the program is used. A period (".") can be used in place of *startline*. The execution will start at the current line number.

RUN *filespec*,R

filespec is the *filespec* for a BASIC program stored on disk. If, R is added, BASIC leaves open files open. Otherwise, all files are closed.

RUN followed by a line-number, period, or nothing at all simply executes the program in memory.

RUN followed by a filespec loads a program from disk and then runs it. Any resident BASIC program will be replaced by the new program.

RUN atuomatically CLEARS all variables.

Examples

```
RUN
```

Execution starts at lowest line number.

```
RUN 100
```

Execution starts at line 100.

```
RUN "DISKDUMP/BAS"
```

When you type the above line and press ENTER, the BASIC sector-dump program will be loaded and executed.

Sample Uses

Suppose you have two programs in memory. One of them begins at line 100 and ends at line 180; the other begins at 200 and ends at 350. Furthermore, the first program has been appropriately terminated (i.e., 180 END). You want to run the second program, stop, observe its output, and then run the first. Type:

```
RUN 200
```

and the second program will execute. When you want to begin execution of the first program, simply type:

```
RUN
```

Sample Program

Suppose you save the following program on disk with the name "PROG1/BAS":

```
200 PRINT "PROG1 EXECUTING ... "
210 RUN "PROG2/BAS"
```

And save this program on disk with the name "PROG2/BAS":

```
220 PRINT "PROG2 EXECUTINE ... "
230 RUN "PROG1/BAS"
```

Now type:

```
RUN "PROG1/BAS
```

ENTER

and you'll see a simple example of program chaining. Hold down the BREAK key to interrupt the program chain.

SAVE
Save Program

SAVE *filespec*, A
A causes the file to be stored in ASCII rather than compressed format.

The SAVE command lets you save your BASIC programs on disk. If the filespec you use as the argument of SAVE already exists, its contents will be lost as the file is re-created.

You can save a program in compressed or ASCII format. Using compressed format takes up less disk space and is faster during SAVEs and LOADs. BASIC programs are stored in RAM using compressed format.

Using the ASCII option makes it possible to do certain things that can't be done with compressed-format BASIC files. Some examples:

- A disk file must be in ASCII form before the MERGE command can be used.
- A disk file must be in ASCII form before TRSDOS commands LIST and PRINT can be used.
- Programs which read in other programs as data typically require that the data programs be stored in ASCII.

For compressed-format programs, a useful convention is to use the extention /BAS. For ASCII-format programs, use /TXT.

Example

```
SAVE "FILE1/BAS.JOHNQDOE:3"
```

saves the resident BASIC program in compressed format. The file name is FILE1; the extension is /BAS; the password is JOHNQDOE. The file is placed on drive 3.

```
SAVE "MATHPAK/TXT", A
```

saves the resident program in ASCII form, using the name MATHPAK/TXT, on the first non-write-protected drive.

SYSTEM
Return to TRSDOS

SYSTEM "command"
command is a string expression specifying a TRSDOS command.

command MUST NOT specify any of the TRSDOS "high memory commands" listed in the TRSDOS Reference Manual, Library Commands section. Furthermore, to call DEBUG from BASIC, you MUST turn DEBGUG on before starting BASIC.

SYSTEM is used to return to TRSDOS, the disk operating system. The argument *command* causes the System to execute the specified TRSDOS command and immediately return back to BASIC. Your program and variables will be unaffected.

If *command* is omitted, SYSTEM returns you to the TRSDOS READY mode.

Examples

```
SYSTEM
```

Returns you to TRSDOS. Your resident BASIC programs will be lost.

```
SYSTEM "DIR"
```

Causes the TRSDOS command, "DIR" (print Directory) to be run, and then returns to BASIC. Your resident BASIC program will remain intact.

Sample Program

```
350 PRINT "THIS IS A PROGRAM FILE "
360 PRINT "BEFORE SAVING IT, I WANT TO SEE WHAT
    FILENAMES HAVE BEEN USED"
370 FOR N=1 TO 1000: NEXT
380 SYSTEM "DIR"
390 PRINT "NOW I CAN CHOOSE A FILENAME WHICH HASN'T
    BEEN USED"
400 END
```

Line 380 causes the system to execute the TRSDOS command DIR which displays a file directory. After displaying the directory, the System immediately returns to BASIC and runs the next line in the program. Line 370 simply sets a two-second pause before displaying the directory.

Program Statements

Program statements allow you to define variable types, initialize and allocate memory, perform input and output, and control the sequence in which statements are executed.

Most program statements can be used in immediate lines as well as in programs. For example:

```
PRINT 23 * 11
```

is an immediate line. As soon as you end the line by pressing ENTER , BASIC executes it. But the line:

```
100 PRINT 23 * 11
```

is a program line. When you press **ENTER**, BASIC does not execute the line but stores in memory to be executed when you type RUN.

Definition and Intialization

The statements in this category perform one or more of three functions.

They change default values set initially by BASIC. For instance, upon initialization, BASIC sets variable V to single-precision. But the statement

```
DEFDBL V
```

resets V to double-precision.

They reserve and allocate memory space. The statement

```
DIM A(12, 12)
```

sets off enough memory to hold a 169 (13 X 13) element array.

They reset and initialize BASIC's pointers. The statement

```
RESTORE
```

causes BASIC's data pointer to be reset to the first data item.

CLEAR
Clear Variables and Allocate String Space

CLEAR *string space*
string space is a numeric expression; if *stringspace* is omitted, string space allocation is unchanged.

When used without an argument, CLEAR sets all numeric variables to zero, and all string variables to null. When used with an argument, this command performs a second function in addition to the one just described: it causes the Computer to set aside for string storage the specified number of bytes. When BASIC is initialized 100 bytes are automatically set aside for strings.

The amount of string storage CLEARed must equal or exceed the greatest number of characters stored in string variables during execution; otherwise an Out of String Space error will occur. By setting string storage to the exact amount needed, your program can make more efficient use of memory. A program which uses no string variables could include a CLEAR 0 statement, for example.

Examples

```
CLEAR
```

All variables are cleared but string space is unchanged.

```
CLEAR 75
```

All variables are cleared and 75 bytes of memory are reserved for string storage.

Sample Program

```
60 CLEAR 100
70 PRINT FRE(A$)
80 CLEAR 0
90 PRINT FRE(A$)
100 CLEAR 100
```

Since CLEAR initializes all variables, you must use it near the beginning of your program, before any variables have been defined.

DATA
Store Program-Data

DATA *item-list*
item list is a list of string and/or numeric constants, separated by commas.

The DATA statement lets you store data inside your program to be accessed by READ statements. The data items will be read sequentially, starting with the first item in the first DATA statement, and ending with the last item in the last DATA statement. Expressions are not allowed in a DATA list. If your string values include leading blanks, colons, or commas, you must enclose these values in quotes.

DATA statements may appear anywhere it is convenient in a program. Generaly, they are placed consecutively, but this is not required. It is important that the data types in a DATA statement match up with the variable types in the corresponding READ statement.

Examples

```
DATA NEW YORK, CHICAGO, LOS ANGELES, PHILADELPHIA, DETROIT
```

This line contains five string data items. Note that quote marks aren't needed, since the strings contain no delimiters or leading blanks.

```
DATA 2.72, 3.14159, 0.0174533, 57.29578
```

This line contains four numeric data items.

```
DATA "SMITH, T.H.", 38, "THORN, J.R.", 41
```

The quote marks are required around the first and third items.

Sample Program

```
170 CLS: PRINT: READ HEADING$: PRINT HEADING$: PRINT STRING
$(42, "-")
180 ON ERROR GOTO 500
190 READ C$: READ DOB: READ N$
200 PRINT C$, DOB, N$: GOTO 190
210 DATA COMPOSER       DATE OF BIRTH       NATIONALITY
220 DATA BOCCHERINI,    1743,               ITALIAN
230 DATA GLUCK,         1714,               GERMAN
240 DATA HAYDN,         1732,               AUSTRIAN
250 DATA MOZART,        1756,               AUSTRIAN
500 IF ERR = 4 THEN END
510 ON ERROR GOTO 0
```

This program prints a list of some major composers of the late 18th Century. Notice we use an ON ERROR GOTO statement to allow the inclusion of data lists of unknown length. For a different means of achieving the same end, see the sample program for READ.

DEFDBL
Define Variables as Double-Precision

DEFDBL *letter list*
letter list is a sequence of individual letters or letter-ranges; the elements in the list must be separated by commas.
a letter-range is of the form:
letter1 — letter2

DEFDBL causes variables beginning with any letter specified in *letter list* to be classified as double-precision, unless a type declaration character is added to the variable name. Double-precision values include 17 digits of precision, though only 16 are printed out.

DEFDBL is ordinarily used at the beginning of a program. Otherwise, it might suddenly change the meaning of a variable that lacks a type declaration character.

Examples

```
DEFDBL K
```

causes any variable beginning with the letter K to be double-precision.

```
DEFDBL Q, S-Z, A-E
```

causes any variable beginning with the letters Q, S through Z, or A through E to be double-precision.

Sample Program

```
570 DEFDBL X
580 A = 3.1415926535897932
590 X = 3.1415926535897932
600 PRINT "PI IN SINGLE PRECISION IS" A
610 PRINT "PI IN DOUBLE PRECISION IS" X
```

DEF FN
Define Function

DEF FN *function name* (*argument-1 . . .*) = *formula*
function name is any valid variable name.
argument-1 and subsequent arguments are used in defining what the function does.
formula is an expresson usually involving the argument(s) passed on the left side of the equals sign.

The DEF FN statement lets you create your own function. That is, you only have to call the new function by name, and the associated operations will automatically be performed. Once a function has been defined with the DEF FN statement, you can call it simply by inserting FN in front of *function name*. You can use it exactly as you might use one of the built-in functions, like SIN, ABS and STRING$.

The type of variable used for *function name* determines the type of value the function will return. For example, if *function name* is single precision, then that function will return a single-precision value, regardless of the precision of the arguments.

The particular variables you use as arguments in the DEF FN statement (*argument-1*, . . .) are not assigned to the function. When you call the function later, any variable name of the same type can be used.

Furthermore, using a variable as an argument in a DEF FN statement has no effect on the value of that variable. So you can use that particular variable in another part of your program without worrying about interference from DEF FN.

The function must be defined with at least one argument. In other words, there must be at least one variable in the position of *argument-1* above, even if this variable is not actually used to pass a value to the function.

Examples

```
DEF FNR(A,B) = A + INT((B - (A - 1)) * RND(0))
```

This statement defines function FNR which returns a random number between integers A and B. The values for A and B are passed when the function is "called", i.e., used in a statement like:

```
Y = FNR(R1, R2)
```

If R1 and R2 have been assigned the values 2 and 8, this line would asign a random number between 2 and 8 to Y.

```
DEF FNL$(X) = STRING$(X, "-")
```

Defines function FNL$ which returns a string of hyphens, X characters long. The value for X is passed when the function is called:

```
PRINT FNL$(30)
```

This line prints a string of 30 hyphens.

Here's an example showing DEF FN used for a complex computation — in double precision.

```
DEF FNX#(A#, B#) = (A# -B#) * (A# - B#)
```

Defines function FNX# which returns the double-precision value of the square of the difference between A# and B#. The values for A# and B# are passed when the function is called:

```
S# = FNX#(A#, B#)
```

We assume that values for A# and B# were assigned elsewhere in the program.

Sample Program

```
710 DEF FNV(T) = (1087 + SQR(273 + T))/16.52
720 INPUT "AIR TEMPERATURE IN DEGREES CELSIUS"; T
730 PRINT "THE SPEED OF SOUND IN AIR OF" T "DEGREES
    CELSIUS IS" FNV(T) "FEET PER SECOND."
```

DEFINT
Define Variables as Integers

DEFINT *letter list*
letter list is a sequence of individual letters or letter-ranges; the elements in the list must be separated by commas.
a letter-range is of the form:
letter1 — letter2

DEFINT causes variables beginning with any letter specified in *letter list* to be classified as integer, unless a type declaration character is added to the variable name. Integer values must be in the range (–32768,32767). They are stored internally in two-byte, two's complement form.

DEFINT may be placed anywhere in a program, but it may change the meaning of variable references without type declaration characters. Therefore, it is normally placed at the beginning of a program.

Examples

```
DEFINT A,I,N
```

After the above line, all variables beginning with A, I, or N will be treated as integers. For example, A1, AA, I3, and NUMBER will be integer variables. However, A1#, AA#, I3#, and NUMBER# would still be double-precision variables, because type-declaration characters always override DEF statements.

```
DEFINT I-N
```

causes any variable beginning with the letters I through N to be treated as an integer variable.

Sample Program

```
880 DEFINT W
890 Z = 1.99999: W = 1.99999
900 PRINT "THE VALUE OF SINGLE-PRECISION Z IS" Z
910 PRINT "BUT THE VALUE OF INTEGER W IS" W
```

DEFSNG
Define Variables as Single-Precision

DEFSNG *letter list*
letter list is a sequence of individual letters or letter-ranges; the elements in the list must be separated by commas.
a letter-range is of the form:
letter1 — letter2

DEFSNG causes variables beginning with any letter specified in *letter list* to be classsified as single-precision, unless a type declaration character is added to the variable name. Double-precision values include 7 digits of precision, though only 6 are printed out.

Example

```
DEFSNG I, W-Z
```

causes any variables beginning with the letters I or W through Z to be treated as single-precision. However, I% would still be an integer variable, and I# a double-precision variable, because of their type declaration characters.

Sample Program

```
960 CLS: DEFINT P: PI = 3.14159
970 PRINT "ALL P'S ARE INTEGERS: WE CAN ONLY MAKE PI =" PI
980 INPUT "WANT TO MAKE P'S SINGLE-PRECISION WITH DEFSNG (Y/
N)"; A$
990 IF A$ = "N" THEN END
1000 CLS: DEFSNG P: PI = 3.14159
1010 PRINT "NOW ALL P'S ARE SINGLE-PRECISION: WE CAN MAKE PI
 =" PI
```

DEFSTR
Define Variables as Strings

DEFSTR *letter list*

letter list is a sequence of individual letters or letter-ranges; the elements in the list must be separated by commas.

a letter-range is of the form:
letter1 — letter2

DEFSTR causes variables beginning with any letter specified in letter list to be classified as strings, unless a type declaration character is added to the variable name.

Example

```
DEFSTR C, L-Z
```

causes any variables beginning with the letters C or L through Z to be string variables, unless a type declaration character is added. After this line is executed, L1 = "WASHINGTON" will be valid.

Sample Program

```
70 S = 555: PRINT "S =" S
80 DEFSTR S
90 S = "SALTON SEA": PRINT "S = " S
```

DEFUSR
Define Point of Entry for USR Routine

DEFUSR*n* = *address*
n equals one of the digits 0,1,...,9; if *n* is omitted, 0 is assumed.
address specifies the entry address to a machine-language routine. Address must be in the range [—32768, 32767].

DEFUSR lets you define the entry points for up to 10 machine-language routines.

Examples

```
DEFUSR3 = &H7D00
```

assigns the entry point 7D00 hex, 32000 decimal, to the USR3 call. When your program calls USR3, control will branch to your subroutine beginning at hex 7D00.

```
DEFUSR = (BASE + 16)
```

assigns start address (BASE + 16) to the USR0 routine.

DIM
Set Up Array

DIM *array1* (*dimension list*) *array2*(*dimension list*)
array1, array2, . . . are variables which name the array(s).
dimension lists are of the form:
subscript1, subscript2,...
each *subscript* is a numeric expression specifying the highest-numbered element in that dimension of the array.
Note: The lowest element in a dimension is always zero.

This statement sets up one or more arrays for structured data processing. Each array has one or more dimensions.

Arrays may be of any type: string, integer, single-precision or double-precision, depending on the type of variable name used to name the array.

When the array is created, BASIC reserves space in memory for each element of the array. (For string arrays, BASIC reserves space for pointers to the string elements, not for the elements themselves.) All elements in a newly created array are set to zero (numeric arrays) or the null string (string arrays).

Arrays can be created implicitly, without explicit DIM statements. Simply refer to the desired array in a BASIC statement, e.g.,

```
A(5) = 300
```

If this is the first reference to array A( ), then BASIC will create the array and assign element A(5) the value of 300. Each dimension of an implicitly defined array is defined to be 11 elements deep, subscripts 0-10.

When an array has been defined, it cannot be re-dimensioned. You must clear the array first (with ERASE, CLEAR or NEW or other variable-clearing operation).

Examples

```
DIM AR(100)
```

Sets up a one-dimensional array AR( ), containing 101 elements: A(O), A(1), A(2), . . ., A(98), A(99), and A(100). The type of the array depends on the type of the name AR. Unless previously changed by a DEFINT, DEFDBL or DEFSTR statement, AR is a single-precision variable.

Note: The array AR( ) is completely independent of the variable AR.

```
DIM L1%(8,25)
```

Sets up a two-dimensional array L1%(,), containing 9 x 26 integer elements, LI%(0,0), L1%(1,0), L1%(2,0), . . . , L1%(8,0), L1%(0,1), LI%(1,1), . . ., L1%(8,1), . . ., L1%(0,25),L1%(1,25), . . ., L1%(8,25).

Two-dimensional arrays like AR(,) can be thought of as a table in which the first subscript specifies a row position, and the second subscript specifies a column position:

0,0	0,1	0,2	0,3	. . .	0,23	0,24	0,25
1,0	1,1	1,2	1,3	. . .	1,23	1,24	1,25
.							
.							
.							
7,0	7,1	7,2	7,3	. . .	7,23	7,24	7,25
8,0	8,1	8,2	8,3	. . .	8,23	8,24	8,25

```
DIM B1(2,5,8), CR(2,5,8), LY$(50,2)
```

Sets up three arrays:

B1 (, ,) and CR (, ,) are three-dimensional, each containing 3*6*9 elements.

LY (,) is two-dimensional, containing 51*3 string elements.

Sample Program

```
170 CLEAR 4000: CLS
180 INPUT "HOW MANY MEMBERS IN THE CLUB"; M
190 DIM L$(M,4)
200 FOR I = 1 TO M
210     PRINT "NAME OF MEMBER #" I;: LINE INPUT"? "; L$(I,1)
220     INPUT "AGE"; L$(I,2)
230     INPUT "PHONE"; L$(I,3)
240     LINE INPUT "ADDRESS? "; L$(I,4)
250 NEXT I
260 PRINT
270 PRINT "THE LIST IS STORED AS FOLLOWS:"
280 PRINT "NAME", "AGE", "PHONE", "ADDRESS"
290 PRINT STRING$(80, "-")
300 FOR I = 1 TO M
310     FOR J = 1 TO 4
320         PRINT L$(I,J),
330     NEXT J
340     PRINT
350 NEXT I
```

ERASE
Delete Array

ERASE *array1, array2,...*
array1, array2 are variable names for currently defined arrays.

The ERASE statement eliminates arrays from a program and allows their space in memory to be used for other purposes. ERASE will only operate on arrays. It can't be used to delete single elements of an array.

If one of the arguments of ERASE is a variable name which is not used in the program, an Illegal Function Call will occur.

Arrays deleted in an ERASE statement may be re-dimensioned.

Example

```
ERASE C,F, TABLE
```

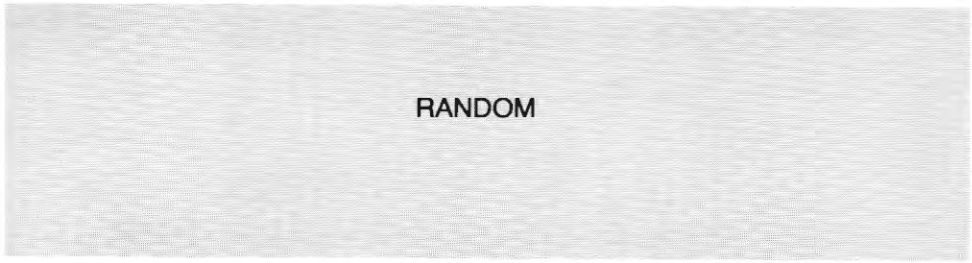
Erases the three specified arrays.

Sample Program

```
400 DIM A(5,5):X=0
410       FOR I=0 TO 5
420             FOR J=0 TO 5
430             X = X + 1
440             A(I,J) = X
445             PRINT A(I,J);
450             NEXT J
460       NEXT I
470 ERASE A
480 DIM A(100)
```

The array that is set up in line 400 is destroyed by the ERASE A statement in line 470. The memory space which is thereby released is now available for further use. The array may be re-dimensioned, as we've chosen to do in line 480.

RANDOM
Reseed Random Number Generator



RANDOM reseeds the random number generator. If your program uses the RND function, the same sequence of pseudorandom numbers will be generated every time the Computer is turned on and the program loaded. Therefore, you may want to put RANDOM at the beginning of the program. This will ensure that you get an unpredictable sequence of pseudorandom numbers each time you load the program.

Random needs to execute just once.

Sample Program

```
600 CLS: RANDOM
610 INPUT "PICK A NUMBER BETWEEN 1 AND 5"; A
620 B = RND(5)
630 IF A = B THEN 650
640 PRINT "YOU LOSE, THE ANSWER IS" B "-- TRY AGAIN."
645 GOTO 610
650 PRINT "YOU PICKED THE RIGHT NUMBER -- YOU WIN!": GOTO 610
```

REM
Comment Line (Remarks)

REM

REM instructs the Computer to ignore the rest of the program line. This allows you to insert remarks into your program for documentation. Then, when you (or someone else) look at a listing of your program, it will be easier to figure out.

If REM is used in a multi-statement proram line, it must be the last statement.

An apostrophe (') may be used as an abbreviation for :REM.

Examples

```
' THIS IS A REMARK
```

Sample Program

```
780 REM  CUSTOMER LOADING PROGRAM
790 REM  THESE LINES ARE INSTRUCTIONS TO THE OPERATOR
800 PRINT "LOADING CUSTOMER FILE"
810 PRINT "THE SCREEN WILL SHOW YOU A SAMPLE ENTRY"
820 REM  THE NEXT LINE SETS A PAUSE BEFORE CLEARING THE SCREEN
830 FOR N=1 TO 1500: NEXT
840 CLS
850 REM  THE NEXT LINES SET THE SAMPLE DISPLAY
```

The above program shows some of the graphic possibilities of REM statements. Any alphanumeric character may be included in a REM statement, and the maximum length is the same as that of other statements: 255 characters total.

RESTORE
Reset Data Pointer

RESTORE causes the next READ statement to be executed to start over with the first item in the first DATA statement. This lets your program re-use the same DATA lines.

Sample Program

```
160 READ X$
170 RESTORE
180 READ Y$
190 PRINT X$, Y$
200 DATA THIS IS THE FIRST ITEM, AND THIS IS THE SECOND
```

When this program is run,

THIS IS THE FIRST ITEM THIS IS THE FIRST ITEM

will be printed on the Display. Because of the RESTORE statement in line 170, the second READ statement starts over with the first DATA item.

Assignment

An assignment statement puts a certain value into a variable or field or trades the value of one variable with another.

```
LSET COLOR$ = "VERMILION"
```

This statement assigns the value VERMILION to the field COLOR$.

```
SWAP A%, B%
```

A% and B% exchange values with one another.

LET
Assign Value to Variable

LET *variable* = *expression*

LET may be used when assigning values to variables. Model II BASIC doesn't require assignment statements to begin with LET, but you might want to use it to ensure compatibility with those versions of BASIC that do require it.

Examples

```
LET A$ = "A ROSE IS A ROSE"
LET B1 = 1.23
LET X = X - Z1
```

In each case, the variable on the left side of the equals sign is assigned the value of the constant or expression on the right side.

Sample Program

```
550 P = 1001: PRINT "P =" P
560 LET P = 2001: PRINT "NOW P = " P
```

LSET and RSET
Place Data in a Direct Access Buffer Field

LSET *name* = *data* and RSET *name* = *data*
name is a field name
data is the data to be placed in the buffer field named by name

These two statements let you place string data into fields previously set up by a FIELD statement.

Examples

Suppose NM$ and AD$ have been defined as field names for a direct access file buffer. NM$ has a length of 18 characters; AD$ has a length of 25 characters. The statements

```
LSET NM$ = "JIM CRICKET,JR."
LSET AD$ = "2000 EAST PECAN ST."
```

put the data in the buffer as follows:

JIM␣CRICKET,JR.␣␣␣	2000␣EAST␣PECAN␣ST.␣␣␣␣␣␣

Notice that filler blanks were placed to the right of the data strings in both cases. If we had used RSET statements instead of LSET, the filler spaces would have been placed to the left. This is the only difference between LSET and RSET.

If a string item is too large to fit in the specified buffer field, it is always truncated on the right. That is, the extra characters on the right are ignored.

MID$=
Replace Portion of String

MID$=(*oldstring, position, length*) = *replacement-string*
oldstring is the variable-name of the string you want to change
position is the numeric expression specifying the position of the first character to be changed
length is a numeric expression specifying the number of characters to be replaced
replacement-string is a string expression to replace the specified portion of *oldstring*

Note: If *replacement-string* is shorter than *length*, then the entire *replacement-string* will be used.

This statement lets you replace any part of a string with a specified new string, giving you a powerful string editing capability.

Note that the *length* of the resultant string is always the same as the original string.

Examples:

A$ = "LINCOLN" in the examples below:

```
MID$(A$, 3, 4) = "12345": PRINT A$
```

which returns LI1234N.

```
MID$(A$, 1, 2) = "": PRINT A$
```

which returns LINCOLN.

```
MID$(A$, 5) = "12345": PRINT A$
```

returns LINC123.

```
MID$(A$, 5) = "01": PRINT A$
```

returns LINCO1N.

```
MID$(A$, 1, 3) = "***": PRINT A$
```

returns ***COLN.

Sample Program

```
770 CLS: PRINT: PRINT
780 LINE INPUT "TYPE IN A MONTH AND DAY MM/DD. "; S$
790 P = INSTR(S$, "/")
800 IF P = 0 THEN 780
810 MID$(S$, P, 1) = CHR$(45)
820 PRINT S$ " IS EASIER TO READ, ISN'T IT?"
```

This program uses INSTR to search for the slash ("/"). When it finds it (if it finds it), it uses MID$= to substitute a "–" (CHR$(45)) for it.

READ
Get Value from DATA Statement

READ *variable*

READ instructs the Computer to read a value from a DATA statement and assign that value to the specified variable. The first time a READ is executed, the first value in the first DATA statement will be used; the second time, the second value in the DATA statement will be read. When all the items in the first DATA statement have been read, the next READ will use the first value in the second DATA statement; etc. (An Out-of-Data error occurs if there are more attempts to READ than there are DATA items.)

Examples

```
READ T
```

reads a numeric value from a DATA statement.

```
READ S$
```

reads a string value from a DATA statement.

Sample Program

This program illustrates a common application for READ and DATA statements.

```
40 PRINT "NAME","AGE"
50 READ N$
60 IF N$="END" THEN PRINT "END OF LIST": END
70 READ AGE
80 IF AGE<18 THEN PRINT N$,AGE
90 GOTO 50
100 DATA "SMITH, JOHN", 30, "ANDERSON, T.M.", 20
110 DATA "JONES, BILL", 15, "DOE, SALLY", 21
120 DATA "COLLINS, W.P.", 17, END
```

RSET
Place Data in a Direct Access Buffer Field

SEE LSET for syntax and description.

SWAP
Exchange Values of Variables

SWAP *variable1, variable 2*

The SWAP statement allows the values of two variables to be exchanged. Either or both of the variables may be elements of arrays. If one or both of the variables are non-array variables which have not had values assigned to them, an Illegal Function Call error will result. Both variables must be of the same type or a Type Mismatch error will result.

Example

```
SWAP F1#, F2#
```

The contents of F2# are put into F1#, and the contents of F1# are put into F2#.

Sample Program

```
250 INPUT "TYPE IN A VALUE FOR F$ "; F$
260 INPUT "TYPE IN A VALUE FOR L$ "; L$
270 SWAP F$, L$
280 PRINT "AFTER SWAP, F$ = " F$ " AND L$ = " L$
```

c. Program Sequence

Control in a BASIC program normally proceeds from one line to the next higher-numbered line to the next higher-numbered line, until the end of the program is reached. The program sequence statements can be used to alter this step-by-step process. With the help of these statements, you can alter the transfer of control in your BASIC program to produce jumps to other parts of the program, iterative loops, and other useful control structures.

For example, the statement

```
IF NOT X > 5 AND NOT Y > 8 THEN 100
```

transfers control to line 100 if X is not greater than 5, and, at the same time, Y is not greater than 8.

```
FOR I = 1 TO 10000: NEXT I
```

Program control will pass back and forth between the FOR statement and the NEXT statement ten thousand times before moving on to the next line, causing a delay of approximately eleven seconds.

END
Terminate Program

END

END terminates execution of a program whenever it is reached in a program line. Some versions of BASIC require END as the last statement in a program. In Model II BASIC it is optional. END is primarily used in Model II BASIC to force execution to terminate at some point other than the logical end of the program.

Sample Program

```
40 INPUT S1,S2
50 GOSUB 100
55 PRINT H
60 END
100 H=SQR(S1*S1 + S2*S2)
110 RETURN
```

The END statement in line 60 prevents program control from "crashing" into the subroutine. Now line 100 can only be accessed by a branching statement such as 50 GOSUB 100.

FOR/NEXT
Establish Program Loop

FOR *variable* = *initial value* TO *final value* STEP *increment*
NEXT *variable*

variable is any valid variable name; *variable* is optional after NEXT
initial value, *final value*, and *increment* are numeric constants, variables, or expressions.
STEP *increment* is optional; if STEP *increment* is omitted, a value of 1 is assumed.

FOR . . . TO . . . STEP/NEXT opens an iterative (repetitive) loop so that a sequence of program statements may be executed over and over a specified number of times.

The first time the FOR statement is executed, *variable* is set to *initial value*. Execution proceeds until a NEXT is encountered. At this point, *variable* is incremented by the amount specified in step *increment*. (If *increment* has a negative value, then *variable* is actually decremented.) STEP *increment* is often omitted, in which case an increment of 1 is used.

Then *variable* is compared with *final value*. If *variable* is greater than *final value*, the loop is completed and execution continues with the statement following NEXT. (If *increment* is a negative number, the loop ends when *variable* is **less** than *final value*.) If *variable* has not yet exceeded *final value*, control passes to the statement following the FOR statement.

Sample Programs

```
830 FOR I = 10 TO 1 STEP -1
840 PRINT I;
850 NEXT I
```

When this program is run, the following output is produced:

```
10  9  8  7  6  5  4  3  2  1
```

FOR NEXT loops may be "nested":

```
880 FOR I = 1 TO 3
890 PRINT "OUTER LOOP"
900       FOR J = 1 TO 2
910       PRINT "      INNER LOOP"
920       NEXT J
930 NEXT I
```

NEXT can be used to close nested loops, by listing the counter-variables. For example, delete line 920 and change 930 to:

```
NEXT J, I
```

GOSUB
Go to Specified Subroutine

GOSUB *line number.*

GOSUB transfers program control to the subroutine beginning at the specified line number. When the Computer encounters a RETURN statement in the subroutine, it then returns control to the statement which follows GOSUB. GOSUB is similar to GOTO in that it may be preceded by a test statement.

Example

```
GOSUB 1000
```

When this line is executed, control will automatically branch to the subroutine at 1000.

Sample Program

```
260 GOSUB 280
270 PRINT "BACK FROM SUBROUTINE": END
280 PRINT "EXECUTING THE SUBROUTINE"
290 RETURN
```

Control is transferred from line 260 to the subroutine beginning at line 280. Line 290 instructs the Computer to return to the statement immediately following GOSUB.

GOTO
Go To Specified Line Number

GOTO *line number*

GOTO transfers program control to the specified line number. Used alone, GOTO *line number* results in an unconditional (automatic) branch. However, test statements may precede the GOTO to effect a conditional branch.

You can use GOTO in the command mode as an alternative to RUN. GOTO *line number* causes execution to begin at the specified line number, *without an automatic CLEAR*. This lets you pass values assigned in the command mode to variables in the execute mode.

Example

```
GOTO 100
```

When this line is executed, control will automatically be transferred to line 100.

Sample Program

```
160 GOTO 200
170 PRINT "AND ARAMIS -- AND D'ARTAGNAN MAKES FOUR.": END
180 PRINT "PORTHOS, ";
190 GOTO 170
200 PRINT "ATHOS, ";
210 GOTO 180
```

IF . . . THEN . . . ELSE
Test Conditional Expression

IF *test* THEN *statement* or *line number* ELSE *statement* or *line number*
ELSE *statement* or *line number* is optional.

The IF . . . THEN . . . ELSE statement instructs the Computer to test the following logical or relational expression. If the expression is true, control will proceed to the THEN clause immediately following the expression. If the expression is false, control will jump to the matching ELSE statement (if one is included) or down to the next program line.

Examples

```
IF X > 127 THEN PRINT "OUT OF RANGE"    END
```

If X is greater than 127, control will pass to PRINT and then to END. If X is not greater than 127, control will jump down to the next line in the program, skipping the PRINT and END statements.

```
IF X > 0 AND Y <> 0 THEN Y = X + 180
```

If both expressions are true, then Y will be assigned the value X + 180. Otherwise control will pass directly to the next program line, skipping the THEN clause.

```
IF A < B PRINT "A < B" ELSE PRINT "A >= A"
```

If A is less than B, the Computer prints the fact and then proceeds down to the next program line, skipping the ELSE statement. If A is not less than B, the Computer jumps directly to the ELSE statement and prints the specified message. **Then** control passes to the next statement in the program.

```
IF A$ = "YES" THEN 210 ELSE IF A$ = "NO" THEN 400 ELSE 370
```

If A$ is YES then the program branches to line 210 . If not, the program skips over to the first ELSE, which introduces a new test. If A$ is NO then the program branches to line 400 . If A$ is any value besides NO or YES, the program skips to the second ELSE and the program branches to line 370.

```
IF A > .001 THEN B = 1/A: A = A/5: ELSE 1510
```

If the value of A is indeed greater than .001, then the next two statements will be executed, assigning new values to B and A. Then the program will drop down to the next line, skipping the ELSE statement. But if A is less than or equal to .001, then the program jumps directly over to ELSE, which then instructs it to branch to 1510 . Note that GOTO is not required after ELSE.

Sample Program

IF THEN ELSE statements may be nested. However, you must take care to match up the IFs and ELSEs.

```
1040 INPUT "ENTER TWO NUMBERS"; A, B
1050 IF A <= B THEN IF A < B THEN PRINT A; ELSE
PRINT "NEITHER "; ELSE PRINT B;
1060 PRINT "IS SMALLER THAN THE OTHER."
```

For any pair of numbers that you enter, this program will pick out and print the smaller of the two.

ON . . . GOSUB
Test and Branch to Subroutine

ON *expression* GOSUB *line number, line number . . .*
expression is a number between 0 and 255.

ON . . . GOSUB is a multi-way branching statement like ON GOTO, except that control passes to a subroutine rather than just being shifted to another part of the program. For further information, see ON GOTO.

Example

```
ON Y GOSUB 1000, 2000, 3000
```

When program execution reaches the line above, if Y = 1, the subroutine beginning at 1000 will be called. If Y = 2, the subroutine at 2000 will be called. If Y = 3, the subroutine at 3000 will be called.

Sample Program

```
430 INPUT "CHOOSE 1, 2, OR 3" ; I
440 ON I GOSUB 500, 600, 700
450 END
500 PRINT "SUBROUTINE #1": RETURN
600 PRINT "SUBROUTINE #2": RETURN
700 PRINT "SUBROUTINE #3": RETURN
```

ON . . . GOTO
Test and Branch to Different Program Line

ON *test-value* GOTO *line number, line number, . . .*
test value is a numeric expression between 0 and 255.

ON . . . GOTO is a multi-way branching statement that is controlled by a test value.

When ON . . . GOTO is executed, test-value is evaluated and the integer portion is obtained. We'll refer to this integer portion as J. The Computer counts over to the Jth line number in the list of line numbers after GOTO, and branches to this line number. If there is no Jth line number, then control passes to the next statement in the program.

Notice that if *test value* is less than zero, an error will occur. There may be any number of line numbers after GOTO.

Examples

```
ON MI GOTO 150, 160, 170, 150, 180
```

says "Evaluate MI.
If integer portion of MI equals 1 then go to line 150;
If it equals 2, then go to 160;
If it equals 3, then go to 170;
If it equals 4, then go to 150;
If it equals 5, then go to 180;

If the integer portion of MI doesn't equal any of the numbers 1 through 5, advance to the next statement in the program."

Sample Program

```
750 INPUT "TYPE IN ANY NUMBER" ; X
760 ON SGN(X) + 2 GOTO 770, 780, 790
770 PRINT "NEGATIVE": END
780 PRINT "ZERO" :END
790 PRINT "POSITIVE" :END
```

SGN(X) returns −1 for X less than zero; 0 for X equal to zero; and +1 for X greater than 0. By adding 2, the expression takes on the values 1, 2, and 3, depending on whether X is negative, zero, or positive. Control then branches to the appropriate line number.

RETURN
Return Control to Calling Program

RETURN ends a subroutine by returning control to the statement immediately following the most-recently executed GOSUB. If RETURN is encountered without execution of a matching GOSUB, an error will occur.

Sample Program

```
330 PRINT "THIS PROGRAM FINDS THE AREA OF A CIRCLE"
340 INPUT "TYPE IN A VALUE FOR THE RADIUS" ; R
350 GOSUB 370
360 PRINT "AREA IS"; A: END
370 A = 3.14 * R * R
380 RETURN
```

d. Input/Output

The input/output statements transfer data between the CPU and peripheral devices.

For example:

```
LPRINT "This is a test"
```

sends the sentence "This is a test" to the Line Printer. The statement

```
GET 1
```

reads the current record from disk and places it in direct file buffer #1.

For further information on file-access programming, see Chapter 4.

Keyboard

INPUT
Input Data to Program

INPUT "*message*"; *variable 1, variable 2, . . .*

When BASIC encounters the INPUT statement in a program it stops execution of the program until you enter certain values from the keyboard. The INPUT statement may specify a list of string or numeric variables, indicating string or numeric values to be input. For instance, INPUT X$, X1, Z$, Z1 calls for you to input a string literal, a number, another string literal, and another number, *in that order.*

When the statement is encountered, the Computer will display a ?. You may then enter the values all at once or one at a time. To enter values all at once, separate them by commas. (If your string literal includes leading blanks, colons, or commas, you must enclose the string in quotes.)

If you ENTER the values one at a time, the Computer will display a ??, indicating that more data is expected. Continue entering data until all the variables have been set, at which time the Computer will advance to the next statement in your program.

Be sure to enter the correct type of value according to what is called for by the INPUT statement. For example, you can't input a string-value into a numeric variable. If you try, the Computer will display a ?REDO FROM START and give you another chance to enter the correct type of data value, starting with the *first* value to be called for by the INPUT list.

If you ENTER more data elements than the INPUT statement specifies, the Computer will display the message ?EXTRA IGNORED and continue with normal execution of your program.

You can include a "prompting message" in your INPUT statement. This will make it easier to input the data correctly. The prompting message must immediately follow INPUT. It must be enclosed in quotes, and it must be followed by a semicolon.

You can enter any valid constant. 2, 105, 1, 3#, etc. are all valid constants.

Examples

```
INPUT Y%
```

If this line were part of your program, when this line is reached, you must type any integer number and press ENTER before the program will continue.

```
INPUT SENTENCE$
```

Here you would have to type in a string when this line is reached. The string wouldn't have to be enclosed in quotation marks unless it contained a comma, a colon, or a leading blank.

```
INPUT "ENTER YOUR NAME AND AGE (NAME, AGE)"; N$, A
```

This line would print a message on the screen which would help the person at the keyboard to enter the right sort of data.

Sample Program

```
50 INPUT "HOW MUCH DO YOU WEIGH"; X
60 PRINT "ON MARS YOU WOULD WEIGH ABOUT" CINT(X * .38) "POUNDS."
```

LINE INPUT
Input a Line from Keyboard

LINE INPUT["*prompt*"] ;*variable*
prompt is a prompting message
var$ is the name that will be assigned to the line you type in

LINE INPUT (or LINEINPUT – the space is optional) is similar to INPUT, except:

- The Computer will not display a question mark when waiting for your operator's input
- Each LINE INPUT statement can assign a value to just one variable
- Commas and quotes your operator can use as part of the string input
- Leading blanks are not ignored – they become part of *var$*
- The only way to terminate the string input is to press **ENTER**

LINE INPUT is a convenient way to input string data without having to worry about accidental entry of delimiters (commas, quotation marks, colons, etc.). The **ENTER** key serves as the only delimiter. If you want anyone to be able to input information into your program without special instructions, use the LINE INPUT statement.

Some situations require that you input commas, quotes and leading blanks as part of the date. LINE INPUT serves well in such cases.

Examples:

```
LINE INPUT A$
```

Input A$ without displaying any prompt.

```
LINE INPUT "LAST NAME, FIRST NAME? ";N$
```

Displays a prompt message and inputs data. Commas will not terminate the input string, as they would in an input statement.

Sample Program

```
200 REM   CUSTOMER SURVEY
205 CLEAR 1000
207 PRINT
210 LINE INPUT "TYPE IN YOUR NAME "; A$
220 LINE INPUT "DO YOU LIKE YOUR COMPUTER? "; B$
230 LINE INPUT "WHY? "; C$
235 PRINT
240 PRINT A$ : PRINT
250 IF B$= "NO" THEN 270
260 PRINT "I LIKE MY COMPUTER BECAUSE "; C$ :END
270 PRINT "I DO NOT LIKE MY COMPUTER BECAUSE "; C$
```

Notice that when line 210 was executed, a question mark was not displayed after the statement, "Type in your name". Also, notice on line 230 you can answer the question "Why" with a statement full of delimeters (",;'etc.).

Video Display

CLS
Clear Screen

CLS

CLS clears the screen. It fills the display with blanks and moves the cursor to the upper-left corner. Alphanumeric characters are wiped out as well as graphics blocks. CLS can be very useful if you should want to present an attractive Display output.

Sample Program

```
540 CLS
550 FOR I = 1 TO 24
560 PRINT STRING$(79,33)
570 NEXT I
580 GOTO 540
```

PRINT, PRINT@, PRINT TAB, PRINT USING
Output to Display

PRINT@ *position, item list*
  @ *position* is a number between 0 and 1919, or
  @ *position* is two numbers,(row, column), *row* between 0 and 23 and *column* between 0 and 79. if @ *position* is omitted, the current cursor position is used.
  item list is a list composed of any of the following items:
    TAB (*number*)
      number is a numeric expression between 0 and 255
    constants
    variables
    expressions,
  where any of these items may be separated by the optional delimiters "," and";".

PRINT@ *position,* USING *format; item list*
  format is one or more of the field specifiers #, *, $, %, !, " " (space), or any alphanumeric character.
  item list is a list composed of constants and variables, which must be separated by the delimiters "," or ";".

PRINT prints an item or a list of items on the Display. The items to be printed may be separated by commas or semicolons. If commas are used, the cursor automatically advances to the next tab position before printing the next item. If semicolons are used, spaces are not inserted between the items printed on the Display. There are ten tab positions to a line, one at each 8-byte column boundary.

Positive numbers are printed with a leading blank, instead of a plus sign. All numbers are printed with a trailing blank. No blanks are inserted before or after strings; you can insert them with the help of quotation marks.

A semicolon at the end of a line overrides the cursor-return so that the next PRINT begins where the last one left off. If no trailing punctuation is used with PRINT, the cursor drops down to the beginning of the next line.

Examples

```
        PRINT "I" ;, "VOTED" ; "FOR" ; "THAT" ; "RASCA
"DEWEY"
```

"I" is printed at tab position 0. "VOTEDFORTHATRASCAL" printed at 28; "DEWEY" at 56.

```
   PRINT A$ B$ C$
```

This line is fully equivalent to

```
   PRINT A$;B$;C$
```

Sample Program

```
   70 N = A
   80 A$ = "EDSELS": B$ = " AND MY WIFE OWNS B."
   90 PRINT "I OWN" N A$ " ";
   100 PRINT B$
```

When run, this program gives

I OWN 6 EDSELS AND MY WIFE OWNS 8,"

Notice that " " prints a space.

PRINT @ *n*

PRINT@ specifies exactly where printing is to begin. There must be no spaces between PRINT and @. The location specified must be a number between 0 and 1919.

Whenever you cause something to PRINT@ on the bottom line of the display, there is an automatic line feed; everything on the Display moves up one line. To suppress this automatic line feed, use a trailing semicolon at the end of the statement.

```
    PRINT @ (11,39),"*"
```

Prints an asterisk in the middle of the Display.

```
    PRINT @ 0, "*"
```

Prints an asterisk at the top left corner of the Display.

Examples

```
PRINT@ 550, "LOCATION 550"
```

Run this to find out where position 550 is.

```
PRINT@ 1000, X
```

Let's say the value of X in the above example is 7. "7" will be printed at location 1001, not 1000. Recall that a positive number will be printed with a leading blank to indicate its sign rather than a plus sign. So a space is printed at 1000 and the number itself is printed at 1001.

Sample Program

```
150 LINE INPUT "TYPE SOMETHING IN.  YOU'LL GET AN ECHO."; L$
155 CLS
160 PRINT@ 500, L$
170 PRINT@ 1000, L$
180 PRINT@ 1500, L$
```

PRINT TAB (*n*)

PRINT TAB moves the cursor to the specified position on the current line (or on succeeding lines if you specify TAB positions greater than 80). TAB may be used more than once in a print list.

Since numerical expressions may be used to specify a TAB position, TAB can be very useful in creating tables, graphs of mathematical functions, etc.

TAB can't be used to move the cursor to the left. If the cursor is to the right of the specified position, the TAB statement will simply be ignored.

Example

```
PRINT TAB(5) "TABBED 5"; TAB(25) "TABBED 25"
```

Notice that no punctuation is needed after the TAB modifiers.

Sample Program

```
220 CLS
230 PRINT TAB(2) "CATALOG NO."; TAB(16) "DESCRIPTION OF ITEM";
240 PRINT TAB(39) "QUANTITY"; TAB(51) "PRICE PER ITEM";
245 PRINT TAB(69) "TOTAL PRICE"
```

PRINT USING *format*

The PRINT USING statement allows you to specify a format for printing string and numeric values. It can be used in applications such as printing report headings, accounting reports, checks, or wherever a specific print format is required.

The PRINT USING statement ordinarily takes this form: PRINT USING *format, item list.* PRINT USING takes the value *item list,* inserts it into the expression *format* as directed by the field specifiers of *format,* and prints the resulting expression. *Format* may be expressed as a variable as well as a constant.

Examples of Field Specifiers

The following field specifiers may be used as part of *format*:

This sign specifies the position of each digit located in the numeric value. The number of # signs you use establishes the numeric field. If the numeric field is greater than the number of digits in the numeric value, then the unused field positions to the left of the number will be displayed as spaces and those to the right of the decimal point will be displayed as zeros.

. The decimal point can be placed anywhere in the numeric field established by the # sign. Rounding-off will take place when digits to the right of the decimal point are suppressed.

, The comma — when placed in any position between the first digit and the decimal point — will display a comma to the left of every third digit as required. The comma establishes an additional position in the field.

In all the examples below, the first line represents a program line as you might type it in; the second line is the value returned after the first line has been run.

```
PRINT USING "######"; 66.2
66
PRINT USING "##.#"; 58.76
58.8
PRINT USING "##########,"; 123456789
123,456,789
```

****** Two asterisks placed at the beginning of the field will cause all unused positions to the left of the decimal to be filled with asterisks. The two asterisks will establish two more positions in the field.

```
PRINT USING "**######"; 44.0
**** 44
```

$$ Two dollar signs placed at the beginning of the field will act as a floating dollar sign. That is, the dollar sign will occupy the first position preceding the number.

```
PRINT USING "$$###.##"; 118.6735, 462.9983, 34.2500
$118.67$463.00 $34.25
```

****$** If these three signs are used at the beginning of the field, then the vacant positions to the left of the number will be filled by the * sign and the $ sign will again position itself in the first position preceding the number.

```
PRINT USING "**$#.##"; 8.333
**$8.33
```

+ When a + sign is placed at the beginning or end of the field, it will be printed as specified as a + for positive numbers of as a − for negative numbers

```
PRINT USING "+*********"; 75200
**+75200
PRINT USING "+####"; -216
−216
```

− When a − sign is placed at the end of tbe field, it will cause a negative sign to appear after all negative numbers. A space will appear after positive numbers.

```
PRINT USING "####.#-"; -8124.420
8124.4−
```

! This causes the Computer to use the first string character of the current value.

```
PRINT USING "!"; "TANZANIA"
T
```

% spaces % To specify a string field of more than one character, *%spaces%* is used. The length of the field will be the number of spaces between the percent signs plus 2.

One space between the backslashes:

```
PRINT USING "\ \"; "TANZANIA"
TAN
```

Four spaces between the backslashes:

```
PRINT USING "\    \"; "TANZANIA"; "ETHIOPIA"
TANZANETHIOP
```

Any other character that you can include in *format* will be displayed as a string literal.

```
PRINT USING "$$.## BUCKS"; 8.625
$8.63 BUCKS
```

If *item list* is a numeric value, the % sign is automatically printed if the field is not large enough to contain the number of digits found in the numeric value. The entire number to the left of the decimal will be displayed preceded by this sign.

```
PRINT USING "%"
PRINT USING "###.#"; 100000
%100000.0
```

Sample Program

```
420 CLS: A$ = "**$##,######.## DOLLARS"
430 INPUT "WHAT IS YOUR FIRST NAME"; F$
440 INPUT "WHAT IS YOUR MIDDLE NAME"; M$
450 INPUT "WHAT IS YOUR LAST NAME"; L$
460 INPUT "ENTER AMOUNT PAYABLE"; P
470 CLS: PRINT "PAY TO THE ORDER OF ";
480 PRINT USING "!! !! "; F$; "."; M$; ".";
490 PRINT L$
500 PRINT: PRINT USING A$; P
```

In line 480, each ! picks up the first character of one of the following string (F$, ".", M$, and "." again). Notice the two spaces in "!! !! ". These two spaces insert the appropriate spaces after the initials of the name (see below). Also notice the use of the variables A$ for *format* and P for *item list* in line 500. Any serious use of the PRINT USING statement would probably require the use of variables at least for *item list* rather than constants. (We've used constants in our examples for the sake of better illustration.)

When the program above is run, the output should look something like this:

```
WHAT IS YOUR FIRST NAME? JOHN
WHAT IS YOUR MIDDLE NAME? PAUL
WHAT IS YOUR LAST NAME? JONES
ENTER AMOUNT PAYABLE? 12345.6
PAY TO THE ORDER OF J.P. JONES
******** $12,345.60 DOLLARS
```

Line Printer

LPRINT, LPRINT TAB, LPRINT USING
Output to Printer

LPRINT item list
item list is a list composed of any of the following items:
TAB (*number*)
number is a numeric expression between 0 and 1920
constants
variables
expressions
where any of these items may be separated by commas or semi-colons

LPRINT USING *format; item list*
format is one or more of the field specifiers #, *, $, %, !, or other characters.
item list is a list composed of constants and variables, which must be separated by commas or semi-colons.

LPRINT, LPRINT TAB, and LPRINT USING allow you to output to the Line Printer.

Examples

```
LPRINT (A * 2)/2
```

Sends the value of the expresion (A * 2)/2 to the Line Printer.

```
LPRINT TAB(50) "TABBED 50"
```

Moves the Line Printer carriage to tab position 50 and prints TABBED 50.

```
LPRINT USING "#####.#"; 2.17
```

Sends the formatted value2.2 to the Line Printer.

For more examples and a more detailed explanation of how to use these statements, see PRINT.

Sample Program

```
330 INPUT X
340 IF X<0 THEN 330
350 LPRINT "SQUARE ROOT IS" SQR(X)
360 END
```

Disk

For programming information, see Chapter 4, "File Access Techniques."

CLOSE
Close Access to File

CLOSE *buffer-number, buffer-number . . .*
buffer-number = 1,2,3, . . . 15
If *buffer-number* is omitted, all open files will be closed.

This command terminates access to a file through the specified buffer or buffers. If buffer-number has not been assigned in a previous OPEN statement, then

```
CLOSE buffer-number
```

has no effect.

Do not remove a diskette which contains an open file. Close the file first. This is because the last records may not have been written to disk yet. Closing the file will write the data, if it hasn't already been written.

The following actions and conditions cause all files to be closed:

```
NEW <ENTER>
RUN <ENTER>
MERGE filespec <ENTER>
Editing a file
Adding or deleting program lines
Execution of the CLEAR statement
Disk errors
```

Examples

```
CLOSE 1,2,8
```

Terminates the file assignments to buffers 1, 2, and 8. These buffers can now be assigned to other files with OPEN statements.

```
CLOSE FIRST% + COUNT%
```

Terminates the file assignement to the buffer specified by the sum FIRST% + COUNT%.

FIELD
Organize a Direct File-Buffer Into Fields

FIELD *buffer-number,* length AS *name, buffer-number,* length AS *name . . .*
buffer-number specifies a direct-access file buffer (1,2,3, . . . 15)
name defines a variable name for the first field

The FIELD statement is used to organize a direct file buffer so data can be passed from BASIC to disk and disk to BASIC. Before fielding a buffer, you must use an OPEN statement to assign that buffer to a particular disk file. (The direct access mode, i.e., OPEN "D", . . . must be used.)

You may use the FIELD statement any number of times to "re-organize" a file buffer. FIELDing a buffer does not clear the contents of the buffer; only the means of accessing the buffer (the field names) are changed. Furthermore, two or more field names can reference the same area of the buffer.

Examples

```
FIELD 1, 128 AS A$, 128 AS B$
```

This statement tells BASIC to assign two 128-byte buffers to the string variables A$ and B$. If you now print A$ or B$, you will see the contents of the buffer. Of course, this value would be meaningless unless you've previously used GET to read a 256-byte record from disk.

Note: All data – both strings and numbers – must be placed into the buffer in string form. There are three pairs of functions (MKI$/CVI, MKS$/CVS, and MKD$/CVD) for converting numbers to strings and strings to numbers. See section B of this chapter.

```
FIELD 3, 16 AS NM$, 25 AS AD$, 10 AS CY$, 2 AS ST$,
      7 AS ZP$
```

The first 16 bytes of buffer 3 are assigned the buffer name NM$; the next 25 bytes, AD$; the next 10, CY$; the next 2, ST$; the next 7, ZP$.

GET
Directly Access a Record from Disk

GET *buffer-number, record number*
buffer-number specifies a direct access file buffer (1,2,3, . . . 15)
record number specifies which record to GET in the file; if omitted, the current record will be read

This statement gets a data record from a disk file and places it in the specified buffer. Before using GET, you must open the file and assign a buffer to it.

When BASIC encounters the GET statement, it reads the record number from the file and places it into the buffer. If you omit record number, it will read the current record.

The current record is the record whose number is one greater than that of the last record accessed. The first time you access a file via a particular buffer, the current record is set to 1.

Examples

```
GET 1
```

Gets record 1 into buffer 1.

```
GET 1, 25
```

Gets record 25 onto buffer 1.

INPUT#
Sequential Read from Disk

INPUT# *buffer-number, name, name . . .*
buffer-number specifies a sequential input file buffer (1,2,3, . . . 15)
name is the variable name to contain the data from the file

This statement inputs data from a disk file.

With INPUT#, data is input sequentially. That is, when the file is opened, a pointer is set to the beginning of the file. The pointer advances each time data is input. To start reading from the beginning of the file again, you must close the file buffer and re-open it.

INPUT# doesn't care how the data was placed on the disk – whether a single PRINT# statement put it there, or whether it required ten different PRINT# statements. What matters to INPUT# is the position of the terminating characters and the EOF marker.

When inputting data into a variable, BASIC ignores leading blanks. When the first non-blank character is encountered, BASIC assumes it has encountered the beginning of the data item.

The data item ends when a terminating character is encountered or when a terminating condition occurs. The terminating characters vary, depending on whether BASIC is inputting to a numeric or string variable.

Examples

```
INPUT#1, A,B
```

Sequentially inputs two numeric data items from disk and places them in A and B. File-buffer #1 is used.

```
INPUT#4, A$, B$, C$
```

Sequentially input three string data items from disk and places them in A$, B$, and C$. File-buffer #4 is used.

LINE INPUT#
Read Line of Text from Disk

LINE INPUT# *buffer-number, name*
buffer-number specifies a sequential input file buffer (1,2,3, . . . 15)
name is the variable name to contain the string data

Similar to LINE INPUT from the keyboard, LINE INPUT# reads a "line" of string data into name. LINE INPUT# is useful when you want to read an ASCII-format BASIC program file as data, or when you want to read in data without following the usual restrictions regarding leading characters and terminators.

LINE INPUT# reads everything from the first character up to

- a carriage return character which is not preceded by a line feed character
- the end-of-file
- the 255th data character (the 255th character is included in the string)

Other characters encountered – quotes, commas, leading blanks, <LF><EN> pairs – are included in the string.

Example

If the data on disk looks like

```
10 CLEAR 500 <X'0D'>
20 OPEN "I", 1, "PROG" <'0A'>
```

then the statement

```
LINE INPUT#1,A$
```

could be used repetitively to read each program line, one line at a time.

OPEN
Assign a Buffer to a File and Set Mode

OPEN *mode, buffer-number, filespec, record-length*
mode is a string expression or constant of which only the first character is significant; this character specifies the mode in which the file is to be opened: I for sequential input, O for sequential output, D for direct-access input-output.
buffer-number specifies a buffer to be assigned the file specified by filespec
filespec defines a TRSDOS file specification
record-length = 0,1,2 . . . 255. If record-length is omitted or if a value of 0 is used, the record length will be 256.

This statement makes it possible to access a file. Mode determines what kind of access you'll have via the specified buffer. Buffer-number determines which buffer will be assigned to the file. Filespec names the file to be accessed. If filespec does not exist, then TRSDOS may or may not create it, depending on the access mode.

When a file is open, it is referenced by the buffer-number which was assigned to it. GET buffer-number, PUT buffer-number, PRINT# buffer-number, INPUT# buffer-number, all reference the file which was opened via buffer-number. The mode must be correct.

Once a buffer has been assigned to a file with the OPEN statement, that buffer can't be used in another OPEN statement. You have to CLOSE it first.

Examples

```
OPEN "O", 1, "CLIENTS/TXT"
```

Opens the file "CLIENTS/TXT" for sequential output. Buffer 1 will be used. If the file does not exist, it will be created. If it already exists, then its previous contents are lost.

```
OPEN "D", 2, "DATA/BAS.SPECIAL"
```

Opens the file DATA/BAS with password SPECIAL in the direct access mode. Buffer number 2 is used. If DATA/BAS does not exist, it will be created on the first non write-protected drive.

```
OPEN "D", 5, "TEST/BAS", 64
```

Opens the file TEXT/BAS for direct access. Buffer number 5 is used. The record length is 64. If this record length does not match the record-length assigned to TEXT/BAS when the file was originally opened, an error will occur.

PRINT#
Sequential Write to Disk File

PRINT# *buffer-number, expression; . . .*
buffer-number specifies a sequential output file buffer (1,2,3, . . . 15)
expression is the expression to be evaluated and written to disk
; is a delimiter placed between every two expressions to be printed to disk. A comma (",") can also be used. The delimiter is not used if there is only one expression to be written

This statement writes data sequentially to the specified file. When you first open a file for sequential output, a pointer is set to the beginning of the file. Thus the first PRINT# places data at the beginning of the file. At the end of each PRINT# operation the pointer advances, so values are written in sequence.

A PRINT# statement creates a disk image similar to what a PRINT to the Display creates on the screen. Remember this, and you'll be able to set up your PRINT# list correctly for access by one or more INPUT statements.

PRINT# does not compress the data before writing it to disk. It writes an ASCII-coded image of the data.

Examples

If A = 123.45

```
PRINT#1,A
```

will write a nine-byte character sequence onto disk:

```
' 123.45'   <EN>
```

where " ' " indicates a blank.

The punctuation in the PRINT list is very important. Unquoted commas and semicolons have the same effect as they do in regular PRINT to Display statements. For example, if A = 2300 and B = 1.303, then

```
PRINT#1, A;B
```

places the data on disk as

```
' 2300'' '' '' '' '' '' '1.303'   <EN>
```

The comma between A and B in the PRINT# list causes 10 extra spaces in the disk file. Generally you wouldn't want to use up disk space this way, so you should use semicolons instead of commas.

Files can be written in a carefully controlled format using PRINT# USING. Or you can use this option to control how many characters of a value are written to disk.

For example, suppose A$ = "LUDWIG", B$ = "VAN", and C$ = "BEETHOVEN". Then the statement

```
PRINT#1,USING"!.!.\   \";A$;B$;C$
```

would write the data in nickname form:

```
L.V.BEET   <EN>
```

(In this case, we didn't want to add any explicit delimiters.) See 2.C.ii., PRINT, for more information on the USING option.

PUT
Write a Direct Access Record to Disk

PUT *buffer-number, record number*
buffer-number specifies a direct access file buffer (1,2,3, . . . 15)
record number specifies the record number of the file. If record number is omitted, the current record number is used

This statement moves data from the buffer of a file into a specified place in the file. Before putting data into a file, you must

- OPEN a file, which assigns a buffer and defines the access mode (which must be D)
- FIELD the buffer, so you can
- place data into the buffer with LSET and RSET statements.

The first time you access a file via a particular buffer, the current record is set equal to 1. (The current record is the record whose number is one greater than the last record accessed.)

If the record number you PUT is higher than the end-of-file record number, then record number become the new end-of-file record number.

Examples

```
PUT 1
```

Puts record 1 into buffer 1.

```
PUT 1,25
```

Puts record 25 into buffer 1.

Debug Statements

The debug statements allow you to isolate logical errors in your programs, and to trap input/output errors so that your program can continue execution in spite of the error.

For example:

```
100 STOP
```

interrupts program execution at the specified line. The Computer will automatically return to the command mode to allow you to test the contents of variables via immediate statements like:

```
PRINT X,Y,Z
```

CONT
Resume Execution of Program

CONT

When program execution has been stopped (by the BREAK key or by a STOP statement in the program), type CONT and **ENTER** to continue execution at the point where the stop or break occurred. During such a break or stop in execution, you may examine variable values (using PRINT) or change these values. Then type CONT and **ENTER** and execution will continue with the current variable values. CONT, when used with STOP and the BREAK key, is primarily a debugging tool.

NOTE: You cannot use CONT after EDITing your program lines or otherwise changing your program. CONT is also invalid after execution has ended normally.

See also STOP.

ERL
Get Line Number of Error

ERL returns the line number in which an error has occurred. This function is primarily used inside an error-handling routine. If no error has occurred when ERL is called, line number 0 is returned. Otherwise, ERL returns the line number in which the error occurred. If the error occurred in the command mode, 65535 (the largest number representable in two bytes) is returned.

Examples

```
PRINT ERL
```

Prints the line number of the error.

```
E = ERL
```

Stores the error's line number for future use.

Sample Program

```
1999 REM ERL PROGRAM
2000 CLEAR 100: ON ERROR GOTO 2125
2010 INPUT"WHAT NAME ARE YOU LOOKING FOR";Q$
2020 READ T$
2030 IF Q$=T$ THEN PRINT T$" IS IN THE LIST": GOTO 2060
2040 GOTO 2020
2050 PRINT Q$" IS NOT IN THE LIST.": RESTORE: GOTO 2010
2060 INPUT"WHAT IS YOUR FAVORITE FRUIT";F$
2070 READ T$
2080 IF F$=T$ THEN PRINT"YOU'RE IN LUCK--WE HAVE SOME.":
     RESTORE: GOTO 2010
2090 GOTO 2070
2100 PRINT"SORRY--WE DON'T HAVE ANY "F$"S": RESTORE: GOTO 2060
2110 DATA TOM, DICK, HARRY, JAMES, ROBERT, SUE, SALLY,
     CERELLE, MARY
2120 DATA WATERMELON, PEACH, PEAR, ORANGE, APPLE, CHERRY,
     TOMATO, AVOCADO
2125 IF ERR<>4 THEN ON ERROR GOTO 0
2130 IF ERL=2020 THEN RESUME 2050
2140 IF ERL=2070 THEN RESUME 2100
2150 ON ERROR GOTO  0
```

ERR
Get Error Code

```
ERR
```

ERR is similar to ERL, except that ERR returns the code of the error rather than the line in which the error occurred. ERR is normally used inside an error-handling routine accessed by ON ERROR GOTO. See the section on error codes in the Appendix.

Examples

```
IF ERR = 7 THEN 1000 ELSE 2000
```

If the error is an Out of Memory error (code 7) the program branches to line 1000; if it is any other error, control will instead go to line 2000.

Sample Program

```
2160 ON ERROR GOTO 2220
2170 READ A
2180 PRINT A
2190 GOTO 2170
2200 PRINT "DATA HAS BEEN READ IN"
2210 END
2220 IF ERR = 4 THEN RESUME 2200
2230 ON ERROR GOTO 0
2232 DATA  4, 2, 0, 5, 9, 2, 2, 31, 7, 13, 1
```

This program "traps" the Out of Data error, since 4 is the code for that error.

ERROR
Simulate Error

ERROR *code*
code is a numeric expression in the range [0,255]

ERROR lets you simulate a specified error during program execution. The major use of this statement is for testing an ON ERROR GOTO routine. When the ERROR *code* statement is encountered, the Computer will proceed exactly as if that error had occurred. Refer to the Appendix for a listing of error codes and their meanings.

Example

```
ERROR 1
```

When the program reaches this line, a Next Without For error (code 1) will "occur", and the Computer will print a message to this effect.

Sample Program

```
2240 INPUT N
2250 ERROR N
```

When you input one of the error code numbers, that error will be simulated in line 2250.

ON ERROR GOTO
Set Up Error-trapping Routine

ON ERROR GOTO *line number*

When the Computer encounters any kind of error in your program, it normally breaks out of execution and prints an error message. With ON ERROR GOTO, you can set up an error-trapping routine which will allow your program to "recover" from an error and continue, without any break in execution. Normally you have a particular type of error in mind when you use the ON ERROR GOTO statement.

For example, suppose your program performs some division operations and you have not ruled out the possibility of division by zero. You might want to write a routine to handle a division-by-zero error, and then use ON ERROR GOTO to branch to that routine when such an error occurs.

The ON ERROR GOTO must be executed before the error occurs or it will have no effect.

The ON ERROR GOTO statement can be disabled by executing the statement, ON ERROR GOTO 0.

If you use this inside an error-trapping routine, BASIC will handle the current error normally.

The error handling routine must be terminated by a RESUME statement. See **RESUME.**

Examples

```
ON ERROR GOTO 1500
```

If an error occurs in your program anywhere after this line, control will suddenly shift to line 1500.

Sample Program

For the use of ON ERROR GOTO in a program, see the sample programs for ERL and ERR.

RESUME, RESUME NEXT
Terminate Error-Trapping Routine

RESUME *line number*
line number is optional.

RESUME NEXT

RESUME terminates an error-handling routine by specifying where normal execution is to resume. Place a RESUME statement at the end of an error-trapping routine. That way later errors can also be trapped.

RESUME without an argument and RESUME 0 both cause the Computer to return to the statement in which the error occurred.

RESUME followed by a line number causes the Computer to branch to the specified line number.

RESUME NEXT causes the Computer to branch to the statement following the point at which the error occurred.

Examples

```
RESUME
```

If an error occurs, when program execution reaches the line above, control will be transferred to the line in which the error occurred.

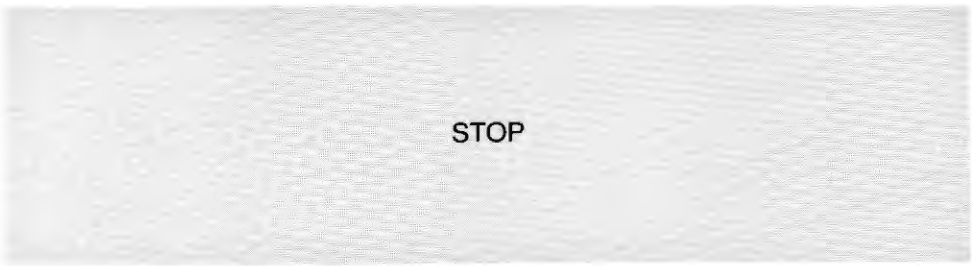
```
RESUME 10
```

If an error occurs, control will be transferred to line 10 after the problem has been fixed.

Sample Program

For the use of RESUME in a program, see the sample programs for ERL and ERR.

STOP
Interrupt Execution of Program



STOP interrupts the execution of your program and prints the words BREAK IN followed by the number of the line that contains the STOP. STOP is primarily a debugging aid. During the break in execution, you can examine variables or change their values.

The CONT command is used to resume execution at the point where it was halted. But if the program itself is altered during the break, CONT can't be used.

Sample Program

```
2260 X = RND(10)
2270 STOP
2280 GOTO 2260
```

A random number between 1 and 10 will be assigned to X and program execution will halt at line 2270. You can now examine the value of X with PRINT X or ? X. Type CONT to start the cycle again.

TRON, TROFF
Turn Trace Function On, Off

TRON

TROFF

TRON turns on a trace function that lets you follow program flow for debugging and for analysis of the execution of the program. Each time the program advances to a new program line, that line number will be displayed inside a pair of brackets.

Sample Program

```
2290 TRON
2300 X = X * 3.14159
2310 TROFF
```

might be helpful in assuring you that line 2300 is actually being executed, since each time it is executed [2300] will be printed on the Display. (We assume the program doesn't jump directly to line 2300 without passing through line 2290, which would execute the assignment statement without turning the trace on.)

After a program is debugged, the TRON and TROFF statements can be removed.

Functions

Computational Functions

Computational functions are internal to BASIC, i.e., they are not concerned with input/output. They perform some action on their argument or arguments and return a result.

There are numeric and string computational functions.

```
ASC
```

is a numeric computational function because it returns a numeric result.

```
OCT$
```

is a string computational function because it returns a string result.

Numeric

ABS
Compute Absolute Value

ABS (*number*)
number is any numeric expression

ABS returns the absolute value of the argument, i.e., the magnitude of the number without respect to its sign. ABS(x)=x for x greater than or equal to zero, and ABS(x)=−x for x less than zero.

Examples

```
X = ABS(Y)
```

The absolute value of Y is assigned to X.

```
IF ABS(X) < 1E-6 THEN PRINT "TOO SMALL"
```

TOO SMALL is printed only if the absolute value of X is less than the indicated number.

Sample Program

```
100 INPUT "WHAT'S THE TEMPERATURE OUTSIDE (DEGREES F)"; TEMP
110 IF TEMP < 0 THEN PRINT "THAT'S" ABS(TEMP)
    "BELOW ZERO!  BRR!": END
120 IF TEMP = 0 THEN PRINT "ZERO DEGREES!  MITE COLD!": END
130 PRINT TEMP "DEGREES ABOVE ZERO?  BALMY!": END
```

ASC
Get ASCII Code

ASC (*string*)
string is a string expression. If *string* is null, an Illegal Function Call will occur.

ASC returns the ASCII code of the first character of the string. The value is returned as a decimal number.

Examples

```
PRINT ASC("A")
PRINT ASC("AB")
```

Both lines will print 65, the ASCII code for "A".

```
PRINT ASC(RIGHT$(T$, 1))
```

Prints the ASCII code of the last character of T$.

Sample Programming

Refer to the ASCII code table in the Appendix. Note that the ASCII code for a lower-case letter is equal to that letter's upper case code plus 32. So ASC can be used to convert lower-case to upper case, simply by subtracting 32 from ASC(x). For instance:

```
140 INPUT "LETTER (a-z)"; X$
150 IF X$ >= "a" AND X$ <= "z" THEN X$ = CHR$(ASC(X$) -32)
160 PRINT X$
```

ASC can be used to make sure that a program is receiving the proper input. Suppose you've written a program that requires the user to input numerals 0-9. To make sure that only those characters are input, and exclude all other characters, you can insert the following routine.

```
170 INPUT "ENTER A NUMBER (0-9)"; N$
180 IF ASC(N$) < 48 OR ASC(N$) > 57 THEN 170
185 IF LEN(N$) > 1 THEN 170
```

ATN
Compute Arctangent

ATN (*number*)
number is a numeric expression

ATN returns the angle whose tangent is number. The angle will be in radians; to convert to degrees, multiply ATN(X) by 57.29578.

Examples

```
X = ATN(Y/3)
```

Assigns the value of the arctangent of Y/3 to X.

```
PRINT ATN(1.0023) * 57.2
```

Prints the indicated value.

```
R = N * ATN(-20 * F2/F1)
```

Assigns the indicated value to R.

Sample Program

```
190 INPUT "TANGENT"; T
200 PRINT "ANGLE IS" ATN(T) * 57.29578 "RADIANS"
```

CDBL
Convert to Double-Precision

CDBL (*number*)
where *number* is any numeric expression.

Returns a double-precision representation of the argument. The value returned will contain 17 digits, but only the digits contained in the argument will be significant.

CDBL may be useful when you want to force an operation to be done in double-precision, even though the operands are single precision or even integers. For example, CDBL (I%)/J% will return a fraction with 17 digits of precision.

Examples

```
Y# = CDBL(N * 3) + M
```

The operations on the right are forced double-precision.

Sample Program

```
210 FOR I = 1 TO 25
220 PRINT 1/CDBL(I),
230 NEXT I
```

Prints the elements of the harmonic series 1, 1/2, 1/3, . . . 1/25 in double-precision.

CINT
Converts to Integer Representation

CINT (*number*)
number is a numeric expression such that −32768 <= *number*<32768.

CINT returns the largest integer not greater than the argument. For example, CINT(1.5) returns 1; CINT(−1.5) returns −2. The result is a two-byte integer.

Examples

```
PRINT CINT(15.0075)
```

Prints the indicated value.

```
K = CINT(X#) + CINT(Y#)
```

The addition will involve only integer arithmetic, which is much faster than double-precision.

Sample Program

```
240 INPUT "ENTER A POSITIVE DECIMAL NUMBER
(LIKE DDDD.DDDD)"; N
250 PRINT "INTEGER PORTION IS"; CINT(N)
```

COS
Compute Cosine

COS (*number*)
number is a numeric expression.

COS returns the cosine of the angle *number*. The angle must be given in radians. When *number* is in degrees, use COS(number * .01745329).

Examples

```
Y = COS(X)
```

Assigns the value of COS(X) to Y.

```
Y = COS(X * .01745329)
```

If X is an angle in degrees, the above line will give its cosine.

```
PRINT COS(5.8) - COS(85 * .42)
```

Prints the difference of the two cosines.

```
G2 = G1 * ((COS(A)) ^ 15)
```

Computes the indicated cosine and stores it in G2.

Sample Program

```
260 INPUT "ANGLE IN RADIANS"; A
270 PRINT "COSINE IS" COS(A)
```

CSNG
Convert to Single-Precision

CSNG (*number*)
number is a numeric expression.

CSNG returns a single-precision representation of the argument. When the argument is a double-precision value, it is returned as six significant digits with "4/5" rounding in the least significant digit. For instance, CSNG(.6666666666666667) returns .666667; CSNG(.3333333333333333) returns .333333.

Examples

```
FC = CSNG(TM#)
```

Assigns the value CSNG (TM#) to FC.

```
PRINT CSNG(.1453885509)
```

Prints a single-precision value.

```
R = CSNG(A#/B#)
```

Performs the indicated computation and stores it in R.

Sample Program

```
280 PI#=3.1415926535897 9
290 B# = 18.000000795
300 PRINT CSNG(PI# * B#)
```

This program prints a single-precision value after the double-precision multiplication.

EXP
Compute Natural Antilog

EXP (*number*)
number is a numeric expression.

Returns the natural exponential of *number*, that is, e^{number}. This is the inverse of the LOG function; therefore, X = EXP(LOG(X)).

Examples

```
H = EXP(A)
```

Assigns the value of EXP(A) to H.

```
PRINT EXP(-2)
```

Prints the value .135335.

```
E = (G1 + G2 - .07) * EXP(.055 * (G1 + G2))
```

Performs the required calculation and stores it in E.

Sample Program

```
310 INPUT "NUMBER"; N
320 PRINT "E RAISED TO THE N POWER IS" EXP(N)
```

FIX
Return Truncated Value

FIX (*number*)
number is a numeric expression.

FIX returns a truncated representation of the argument. All digits to the right of the decimal point are simply chopped off, so the resultant value is an integer. For non-negative X, FIX(X) = INT(X). For negative values of X, FIX(X) = INT(X) + 1.

Examples

```
Y = FIX(X)
```

The truncated number is put in Y.

```
PRINT FIX(2.2)
```

Prints the value 2.

```
PRINT FIX(-2.2)
```

Prints the value -2.

Sample Program

```
330 INPUT "NUMBER"; A#
340 Y# = ABS(A# - FIX(A#))
350 PRINT "FRACTIONAL PORTION IS" Y#
```

This program splits any number into its integer and fractional parts.

INSTR
Search for Specified String

INSTR (*position, string1, string2*)
position specifies the position in *string1* where the search is to begin. *Position* is optional; if it is not supplied, search automatically begins at the first character in *string1*. (Position 1 is the first character in *string1*.)
string1 is the string to be searched.
string2 is the substring you want to search for.

This function lets you search through a string to see if it contains another string. If it does, INSTR returns the starting position of the substring in the target string; otherwise, zero is returned. Note that the entire substring must be contained in the search string, or zero is returned. Also, note that INSTR only finds the first occurrence of a substring, starting at the position you specify.

Examples

In these examples, A$ = "LINCOLN":

```
INSTR(A$, "INC")
```

returns a value of 2.

```
INSTR (A$, "12")
```

returns 0.

```
INSTR(A$, "LINCOLNABRAHAM")
```

returns 0. For a slightly different use of INSTR, look at

```
INSTR (3, "1232123", "12")
```

which returns 5.

Sample Program

The program below uses INSTR to search through the addresses contained in the program's DATA lines. It counts the number of addresses with a specified county zip code (761--) and returns that number. The zip code is preceded by an asterisk to distinguish it from the other numeric data found in the address.

```
360 RESTORE
370 COUNTER = 0
380 ON ERROR GOTO 410
390 READ ADDRESS$
400 IF INSTR(ADDRESS$, "*761") <> 0 THEN COUNTER = COUNTER + 1
    ELSE 390
405 GOTO 390
410 PRINT "NUMBER OF TARRANT COUNTY, TX ADDRESSES IS" COUNTER:
    END
420 DATA "5950 GORHAM DRIVE, BURLESON, TX *76148"
430 DATA "71 FIRSTFIELD ROAD, GAITHERSBURG, MD *20760"
440 DATA "1000 TWO TANDY CENTER, FORTH WORTH, TX *76102"
450 DATA "16633 SOUTH CENTRAL EXPRESSWAY, RICHARDSON, TX *75080"
```

INT
Convert to Integer Value

INT(*number*)
number is any numeric expression.

INT returns an integer representation of the argument, using the largest whole number that is not greater than the argument. The result has the same precision as the argument. The argument is not limited to the range −32768 to 32767.

Examples

```
A = INT(X)
```

Gets the integer value of X and stores it in A.

```
PRINT INT(2.5)
```

Prints the value 2.

```
PRINT INT(-2.5)
```

Prints −3.

Sample Program:

```
460 INPUT X#
470 IF X# < 0 THEN GOTO 460
480 A = INT((X# * 100) + .5)/100
490 PRINT A
```

If you type in a positive number with a fraction like 25.733720, this program will round it off to two decimal places and print it.

LEN
Get Length of String

LEN (*string*)
string is a non-null string expression.

LEN returns the character length of the specified string.

Examples

```
X = LEN(SENTENCE$)
```

Gets the length of SENTENCE$ and stores it in X.

```
PRINT LEN("CAMBRIDGE") + LEN("BERKELEY")
```

Prints the value 17.

Sample Program

```
500 A$ = ""
510 B$ = "TOM"
520 PRINT A$, B$, B$ + B$
530 PRINT LEN(A$), LEN(B$), LEN(B$ + B$)
```

When this short program is run, the following will be printed on the display:

```
                TOM             TOMTOM
 0               3               6
```

LOG
Compute Natural Logarithm

LOG (*number*)
number is a numeric expression.

LOG returns the natural logarithm of the argument. This is the inverse of the EXP function, so X = LOG(EXP(X)). To find the logarithm of a number to another base *B*, use the formula LOG B(X) = LOG E(X)/LOG E(B). For example, LOG(32767)/LOG(2) returns the logarithm to base 2 of 32767.

Examples

```
B = LOG(A)
```

Computes the value of LOG(A) and stores it in B.

```
PRINT LOG(3.14159)
```

Prints the value 1.14473.

```
Z = 10 * LOG(P2/P1)
```

Performs the indicated calculation and assigns it to Z.

This program demonstrates the use of LOG. It utilizes a formula taken from space communications research.

Sample Program

```
540 INPUT "DISTANCE SIGNAL MUST TRAVEL (MILES)"; D
550 INPUT "SIGNAL FREQUENCY (GIGAHERTZ)"; F
560 L = 96.58 + (20 * LOG(F)) + (20 * LOG(D))
570 PRINT "SIGNAL STRENGTH LOSS IN FREE SPACE IS" L "DECIBELS."
```

RND
Generate Pseudorandom Number

RND (*number*)
number is a numeric expression such that
0 .=*number* .32768.

RND produces a pseudorandom number using the current "seed" number. The seed is generated internally and is not accessible to the user. RND may be used to produce random numbers between 0 and 1, or random integers greater than 0, depending on the argument.

RND(0) returns a single-precision value between 0 and 1. RND(X), where X is an interger between 1 and 32768, returns an integer between 1 and X. For example, RND(55) returns a pseudorandom integer between 1 and 55. RND(55.5) returns a number in the same range, because RND uses the integer value of the argument.

Examples

```
A = RND(2)
```

A is given a value of 1 or 2.

```
A = RND(Z)
```

Returns a random integer between 1 and Z and assigns it to A.

```
PRINT RND(0)
```

Prints a decimal fraction between 0 and 1.

Sample Program

```
580 FOR I = 1 TO 100
590 PRINT RND(10);
600 NEXT I
```

This prints 100 pseudorandom numbers between 1 and 10.

SGN
Get Sign

SGN (*number*)
number is a numeric expression.

This is the "sign" function. It returns −1 if its argument is a negative number, 0 if its argument is zero, and 1 if its argument is a positive number.

Examples

```
Y = SGN(A * B)
```

The function determines what the sign of the expression A * B is, and passes the appropriate number (−1, 0, 1) to Y.

```
PRINT SGN(N)
```

Prints the appropriate number on the Display.

Sample Program

```
610 INPUT "ENTER A NUMBER"; X
620 ON SGN(X) + 2 GOTO 630, 640, 650
630 PRINT "NEGATIVE": END
640 PRINT "ZERO": END
650 PRINT "POSITIVE": END
```

SIN
Compute Sine

SIN (*number*)
number is a numeric expression.

SIN returns the sine of the argument, which must be in radians. To obtain the sine of X when X is in degrees, use SIN(X * .0174533).

Examples

```
W = SIN(MX)
```

Assigns the value of SIN(MX) to W.

```
PRINT SIN(7.96)
```

Prints the value .994385.

```
E = (A * A) * (SIN(D)/2)
```

Performs the indicated calculation and stores it in E.

Sample Program

```
660 INPUT "ANGLE IN DEGREES"; A
670 PRINT "SINE IS " SIN(A * .0174533)
```

SQR
Compute Square Root

SQR (*number*)
number is a non-negative numeric expression.

SQR returns the square root of its argument.

Examples

```
Y = SQR(A + B)
```

Performs the required calculation and stores it in Y.

```
PRINT SQR(155.7)
```

Prints the value 12.478.

Sample Program

```
680 INPUT "TOTAL RESISTANCE (OHMS)"; R
690 INPUT "TOTAL REACTANCE (OHMS)"; X
700 Z = SQR((R * R) + (X * X))
710 PRINT "TOTAL IMPEDANCE (OHMS) IS" Z
```

This program computes the total impedance for series circuits.

TAN
Compute Tangent

TAN (*number*)
number is a numeric expression.

TAN returns the tangent of the argument. The argument must be in radians. To obtain the tangent of X when X is in degrees, use TAN(X * .01745329).

Examples

```
L = TAN(M)
```

Assigns the value of TAN(M) to L.

```
PRINT TAN(7.96)
```

Prints the value −9.39702.

```
Z = (TAN(L2 - L1))/2
```

Performs the indicated calculation and stores the result in Z.

Sample Program

```
720 INPUT "ANGLE IN DEGREES"; ANGLE
730 T = TAN(ANGLE * .01745329)
740 PRINT "TAN IS" T
```

VAL
Evaluate String

VAL (*string*)
string is a string expression.

VAL is the inverse of the STR$ function; it returns the number represented by the characters in a string argument. This number may be integer, single precision, or double precision depending on the range of values and the rules used for typing all constants.

For example, if A$ = "12" and B$ = "34" then VAL(A$ + "." + B$) returns the value 12.34 and VAL(A$ + "E" + B$) returns the value 12E34, that is, 12 * 10↑34.

VAL terminates its evaluation on the first character which has no meaning in a numeric term – e.g., Z, ?, etc. The current value at termination is used.

If the string is non-numeric or null, VAL returns a zero.

Examples

```
PRINT VAL("100 DOLLARS")
```

prints 100.

```
PRINT VAL("1234E5")
```

prints 1.234E+08.

```
B = VAL("3" + "*" + "2")
```

The value 3 is assigned to B.

Sample Program

REM WHAT SIDE OF THE STREET?

```
750 REM      WHAT SIDE OF THE STREET?
760 REM      NORTH IS EVEN; SOUTH IS ODD
770 LINE INPUT "ENTER THE ADDRESS (NUMBER AND STREET) "; AD$
780 C = INT(VAL(AD$)/2) * 2
790 IF C = VAL(AD$) THEN PRINT "NORTH SIDE": GOTO 770
800 PRINT "SOUTH SIDE": GOTO 770
```

String

CHR$
Get Character for ASCII or Control Code

CHR$ (*number*)
number is a numeric expression,
number = 0,1,2, . . .,255

CHR$ is the inverse of the ASC function. It returns a one-character string; this character has the ASCII, control, or graphics code number specified by the argument of the function.

Examples:

```
P$ = CHR$(T)
```

The function CHR$ converts the number T into its ASCII character equivalent and puts the character into P$.

```
PRINT CHR$(35)
Prints a # on the Display.
```

```
PRINT CHR$(26)
```

Puts the Display into its black-on-white mode (use CHR$ (25) to return to normal).

```
A$ = A$ + CHR$(I)
```

The character whose ASCII code is I is added to the end of A$.

Sample Program

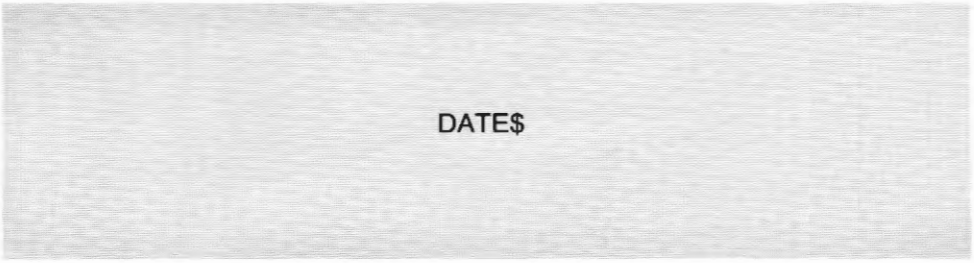
Using CHR$, you can assign quotation marks to strings, even though they are ordinarily used as string-delimiters. Since the ASCII code for quotations is 34, A$ = CHR$(34) assigns the value " to 34.

```
700 A$ = CHR$(34)
710 PRINT "HE SAID, "; A$; "HELLO."; A$
```

When this is RUN, the following line will be printed on the Display:

```
HE SAID, "HELLO."
```

DATE$
Get Today's Date



This function lets you display today's date and use it in a program.

The operator sets the date initially when TRSDOS is started up. When you request the date, BASIC will display it in this fashion:

```
SATAPR281979118 45
```

which means Saturday, April 28, 1979, 118th day of the year, 4th month of the year, 5th day of the week (Monday is the 0th day of the week).

Example

```
PRINT DATE$
```

which returns

```
WEDJUL251979206 72
```

Sample Program

```
        1090 PRINT "Inventory Check:"
        1100 IF DATE$ = "THUJAN311980031 13" THEN PRINT "Today is
the last day of January 1980.  Time to perform monthly
inventory.": END
        1110 A$ = LEFT$(DATE$, 8): B$ =  RIGHT$(A$, 2)
        1120 B = VAL(B$)
        1130 PRINT 31 - B " days until inventory time."
```

HEX$
Compute Hexadecimal Value

HEX$ (*number*)
number is a numeric expression,
−32768 ≤ *number* 32768

HEX$ returns a string which represents the hexadecimal value of the argument. The value returned is like any other string – it cannot be used in a number expression. That is, you cannot add hex strings. You **can** concatenate them, though.

Examples:

```
PRINT HEX$(30); HEX$(50); HEX$(90)
```

prints the following strings:

```
1E        32        5A
```

```
Y$ = HEX$(X/16)
```

Y$ is the hexadecimal string representing the integer quotient X/16.

Sample Program

```
720 INPUT "DECIMAL VALUE"; DEC
730 PRINT "HEXADECIMAL VALUE IS " HEX$(DEC)
```

LEFT$
Get Left Portion of String

LEFT$ (*string, number*)
string is a string expresson, *string* null string
number is a numeric expression, LEN(*string*) = number

LEFT$ returns the first *number* characters of *string*. If LEN(*string*) = *number,* the entire string is returned.

Examples:

```
PRINT LEFT$("BATTLESHIPS", 6)
```

Prints the left six characters of BATTLESHIPS, namely, BATTLE.

```
PRINT LEFT$("BIG FIERCE DOG", 20)
```

Since BIG FIERCE DOG is less than 20 characters long, the whole phrase is printed.

```
PHRASE$ = LEFT$(M$, 12)
```

Puts the first 12 characters of M$ into PHRASE$.

```
PRINT LEFT$("ALPHA" + "BETA" + "GAMMA", 8)
```

Prints ALPHABET.

Sample Program

```
740 A$ = "TIMOTHY"
750 B$ = LEFT$(A$, 3)
760 PRINT B$; "--THAT'S SHORT FOR "; A$
```

When this is run, the following will be printed:

```
TIM--THAT'S SHORT FOR TIMOTHY
```

MID$
Get Substring

MID$ (string, position, length)
string is a string expression
position is the position where the substring begins in string
length is the number of characters in the substring (this parameter is optional)

MID$ returns a substring of string. The substring begins at position in string and is length characters longs.

If length is omitted, the entire string beginning at position will be returned.

Examples

If A$ = "WEATHERFORD" then

```
PRINT MID$(A$, 3, 2)
```

prints AT.

```
F$ = MID$(A$, 3)
```

puts ATHERFORD into F$.

Sample Program

```
200 INPUT "AREA CODE AND NUMBER (NNN-NNN-NNNN)"; PH$
210 EX$ = MID$(PH$, 5, 3)
220 PRINT "NUMBER IS IN THE " EX$ " EXCHANGE."
```

The first three digits of a local phone number are sometimes called the exchange of the number. This program looks at a complete phone number (area code, exchange, last four digits) and picks out the exchange of that number.

OCT$
Compute Octal Value

OCT$(*number*)
number is a numeric expression.

OCT$ returns a string which represents the octal value of the argument. The value returned is like any other string – it cannot be used in a numeric expression

Examples:

```
PRINT OCT$(30), OCT$(50), OCT$(90)
```

prints the following strings:

```
 36          62          132
```

```
Y$ = OCT$(X/84)
```

Y$ is a string representation of the integer quotient X/84 to base 8.

Sample Program

```
830 INPUT "DECIMAL VALUE"; DEC
840 PRINT "OCTAL VALUE IS " OCT$(DEC)
```

RIGHT$
Get Right Portion of String

RIGHT$ (*string, number*)
string is a string expression, *string* not equal to null string
number is a numeric express.

RIGHT$ returns the last *number* characters of *string*. If LEN(string) is less than or equal to *number*, the entire string is returned.

Examples:

```
PRINT RIGHT$("WATERMELON", 5)
```

Prints the five right characters of WATERMELON, namely, MELON.

```
PRINT RIGHT$("MILKY WAY", 25)
```

Since MILKY WAY is less than 25 characters long, the whole phrase is printed.

```
ZIP$ = RIGHT$(ADDRESS$, 5)
```

Puts the last five characters of ADDRESS$ into ZIP$.

```
PRINT RIGHT$(STRING$(50, CHR$(33)), 1)
```

Prints a single "!".

Sample Program

```
850 RESTORE: ON ERROR GOTO 880
860 READ COMPANY$
870 PRINT RIGHT$(COMPANY$, 2);: GOTO 860
880 END
890 DATA "BECHMAN LUMBER COMPANY, SEATTLE, WA"
900 DATA "ED NORTON SEWER SERVICE, BROOKLYN, NY"
910 DATA "HAMMOND MANUFACTURING COMPANY, HAMMOND, IN"
```

This program prints the name of the state in which each company is located.

SPACE$
Return String of Spaces

SPACE$ (*length*)
length is a numeric expression,
0 = *length* 256.

SPACE$ returns a string of spaces. The number of spaces is determined by the argument.

Examples:

```
PRINT "DESCRIPTION" SPACE$(4) "TYPE" SPACE$(9) "QUANTITY"
```

Prints DESCRIPTION followed by four spaces followed by TYPE followed by nine spaces followed by QUANTITY.

```
A$ = SPACE$(14)
```

Puts a string of fourteen spaces into A$.

```
SP$ = SPACE$(N)
```

Puts a string of N spaces into SP$.

Sample Program

```
920 PRINT "Here"
930 PRINT SPACE$(13) "is"
940 PRINT SPACE$(26) "an"
950 PRINT SPACE$(39) "example"
960 PRINT SPACE$(52) "of"
970 PRINT SPACE$(65) "SPACE$"
```

STR$
Convert to String Representation

STR$(*number*)
number is a numeric expression.

STR$ converts its argument to a string. For example, if X = 58.5, then STR$(X) equals the string " 58.5". Notice that a leading blank is inserted before 58.5 to allow for the sign of X. While arithmetic operations may be performed on X, only string functions and operations may be performed on the string " 58.5".

Examples:

```
S$ = STR$(X)
```

Converts the number X into a string and stores it in S$.

```
T$ = STR$(A * 18)
```

Converts the number A * 18 into a string and stores it in T$.

Sample Program

```
 980 CLEAR 200
 990 CLS: LINE INPUT "TO APPLY FOR VOTER REGISTRATION,
     TYPE YOUR FULL NAME. "; NM$
1000 INPUT "AND WHAT'S YOUR AGE"; AGE
1010 IF AGE < 18 THEN PRINT "SORRY, WE CAN'T REGISTER YOU.
     YOU MUST BE 18.":
     END
1020 REG$ = NM$ + STR$(AGE) + STR$(RND(10000)): CLS
1030 PRINT "NAME - AGE - VOTER REGISTRATION NUMBER":
     PRINT REG$
```

In the above program, the variable AGE must first be tested as a numeric value and later stored as part of a string. STR$ is used to accomplish the latter function, not only for AGE but for RND as well.

STRING$
Return String of Characters

STRING$ (*length, character*)
length is a numeric expression,
number = 0,1,2, . . .,255 0 = *length* 256
character is a string expression or an ASCII code.

STRING$ returns a string of characters. How many characters are returned depends on STRING$'s first argument; what characters they are depends on its second argument. For example, STRING$(30, 65) returns a string of thirty "A"s. STRING$(30,20) returns a string of 30 blanks, since 20 is the code for a blank character.

STRING$ is useful for creating graphs, tables, and so on.

Examples:

```
B$ = STRING$(25, "X")
```

Puts a string of 25 "X"s into B$.

```
PRINT STRING$(50, 10)
```

10 is ASCII code for a line feed, so the line above will print 50 blank lines on the Display.

Sample Program

```
1040 CLEAR 300
1050 INPUT "TYPE IN THREE NUMBERS BETWEEN 33 AND 159 (N1, N2
, N3)"; N1, N2, N3
1060 CLS: FOR I = 1 TO 4: PRINT STRING$(20, N1): NEXT I
1070 FOR J = 1 TO 2: PRINT STRING$(40, N2): NEXT J
1080 PRINT STRING$(80, N3)
```

TIME$
Get the Time

TIME$

This function lets you use the time in a program.

The operator sets the time initially when TRSDOS is started up. When you request the time, TIME$ will supply it using this format:

```
14.47.18
```

which means 14 hours, 47 minutes, and 18 seconds (24-hour clock) or 2:47:18 P.M.

Example

```
A$ = TIME$
```

When this line is reached in your program, the current time is stored in A$.

Sample Program

```
1140 IF LEFT$(TIME$, 5) = "10.15" THEN PRINT "Time is 10:15
A.M.--time to pick up the mail.": END
1150 GOTO 1140
```

Input/Output Functions

The input/output functions are concerned with the transfer of data from the CPU to peripheral devices, and from peripheral devices to the CPU. They also return information which indicates a peripheral's state of readiness.

Model II input/output functions are dependent on TRSDOS input/output drivers.

Keyboard

INKEY$
Get Keyboard Character

INKEY$

Returns a one-character string from the keyboard without the necessity of having to press ENTER If no key is pressed, a null string (length zero) is returned. Characters typed to INKEY$ are not echoed to the Display.

INKEY$ is invariably put inside some sort of loop. Otherwise program execution would pass through the line containing INKEY$ before a key could be pressed.

Example

```
A$ = INKEY$
```

When put into a loop, the above program fragment will get a key from the keyboard and store it in A$. If the line above is used by itself, when control reaches it and no key is being pressed, a null string (" ") will be stored in A$.

Sample Program

```
1200 CLS
1210 PRINT@ 540, INKEY$;
1220 GOTO 1210
```

When you run this program, the screen will remain blank (except for the cursor) until you strike a key. The last key that you strike will remain on the Display until you press another one. Whenever you fail to hit a key while this program is executing, a null string, i.e., nothing, is printed at 540.

INPUT$
Input a Character String

INPUT$ (*length*)
length is a numeric expression.

This function allows a program to input a specified number of keyboard characters. As soon as the last required character is typed, execution continues. (You don't have to press ENTER to signify end-of-line.) The characters you type will not be displayed on the screen.

Any character you type will be accepted (except BREAK).

Examples

```
A$=INPUT$(5)
```

A string of 5 characters must be input before BASIC will proceed to the next line of the program.

Sample Program

This program shows how you might use INPUT$ to have an operator input a password to access a protected file. By using INPUT$, the operator can type in the password without anyone seeing it on the Video Display. (To see the full file specification, Run the program, then type:

PRINT F$.

```
110 LINE INPUT "TYPE IN THE FILENAME/EXT"; F$
120 PRINT "TYPE IN THE PASSWORD -- MUST TYPE 8 CHARACTERS: ";
130 P$ = INPUT$(8)
140 F$ = F$ + "." + P$
```

Video Display

POS
Returns Cursor Position

POS (*dummy*)
dummy is any numeric expression.

POS returns a number from 1 to 80 indicating the current cursor position on the Display.

Examples

```
PRINT TAB(40) POS(0)
```

The PRINT TAB statement moves the cursor to position 40. Since the cursor is at 40, POS(0) returns the value 40, and 40 is printed on the Display. (However, since a blank is inserted before the "4" to accommodate the sign, the "4" is actually at position 41.) The "0" in POS(0) is the dummy argument.

Sample Program

```
1230 PRINT "THESE" TAB(POS(0) + 5) "WORDS";
1232 PRINT TAB(POS(0) + 5) "ARE" TAB(POS(0) + 5) "EVENLY";
1240 PRINT TAB(POS(0) + 5) "SPACED"
```

When this program is RUN, you should get this output:

```
THESE     WORDS     ARE     EVENLY     SPACED
```

ROW
Get Row Position of Cursor

ROW (*dummy*)
dummy is a dummy argument.

The ROW function finds the row at which the cursor is currently located, and returns that row-number. The 24 rows are numbered 0-23.

Examples

```
X = ROW(Y)
```

The row-number of the cursor's position at the time this line is encountered is assigned to X.

```
PRINT ROW(0)
```

The row-number is printed on the Display.

Sample Program

When a key is typed, the program below will print it, find its Display row-number and column-number, print this information, find its ASCII code, and print this information too.

```
100 CLS
110 R=0:C=0
120 PRINT@(21,32),"ROW","COLUMN"
130 X$=INPUT$(1)
140 PRINT @(R,C),X$;
150 C=POS(0):R=ROW(0)
160 PRINT @ (22,32),R,C;
163 PRINT @ (23,32),STRING$(20,32);
165 PRINT @(23,32),"ASCII CODE IS "HEX$(ASC(X$));
170 PRINT@(R,C),"";
180 GOTO 130
```

SPC
Print Line of Blanks

SPC (*number*)
number is a numeric expression,
number = 0,1,2, . . ., 255

SPC prints a line of blanks. The number of blanks is determined by the argument of SPC.

Examples

```
PRINT SPC(25); "Hello"
```

Sample Program

```
1250 PRINT SPC(75) "Here"
1260 PRINT SPC(60) "is"
1270 PRINT SPC(45) "an"
1280 PRINT SPC(30) "example"
1290 PRINT SPC(15) "of"
1300 PRINT "SPC"
```

Disk

See Chapter 4 for sample programming using these functions.

CVD, CVI, CVS
Restore String Data to Numeric

CVD (*string*)
string is a string expression which defines an eight-character string; *string* is typically the name of a buffer-field containing a numeric string. If LEN(*string*) <8, an Illegal Function Call occurs; if LEN(*string*)>8, only the first eight characters are used.

CVI (*string*)
string is a string expression which defines a two-character string; *string* is typically the name of a buffer-field containing a numeric string. If LEN(*string*) <2, an Illegal Function Call occurs; if LEN(*string*)>2, only the first two characters are used.

CVS (*string*)
string is a string expression which defines a four-character string; *string* is typically the name of a buffer-field containing a numeric string. If LEN(*string*) <4, an Illegal Function Call occurs; if LEN(*string*)>4, only the first four characters are used.

These functions let you restore data to numeric form after it is read from disk. Typically the data has been read by a GET statement, and is stored in a direct access file buffer. CVD, CVI, and CVS are the inverses of MKD$, MKI$, and MKS$, respectively.

Examples

Suppose the name GROSSPAY$ references an eight-byte field in a direct access file buffer, and after GETting a record, GROSSPAY$ contains an MKD$ representation of the number 13123.38. Then the statement

```
A# = CVD(GROSSPAY$)
```

assigns the numeric value 13123.38 to the double-precision variable A#.

Sample program

```
1420 OPEN "D", 1, "TEST/DAT"
1430 FIELD 1, 2 AS I1$, 4 AS I2$, 8 AS I3$
1440 GET 1
1450 PRINT CVI(I1$), CVS(I2$), CVD(I3$)
1460 CLOSE
```

This program opens a file named "TEST/DAT" which is assumed to have been previously created. (For the program which creates the file, see the section on MKD$, MKI$, and MKS$.) CVI, CVS, and CVD are used to convert string data back to numeric form.

EOF
End-of-file detector

EOF (*buffer number*)

This function checks to see whether all characters up to the end-of-file marker have been accessed, so you can avoid INPUT PAST END errors during sequential input.

Assuming *number* specifies an open file, then EOF(*number*) returns 0 (false) when the EOF record has not yet been read, and −1 (true) when it has been read.

Examples

```
IF EOF(FILE) THEN CLOSE FILE
```

This line determines whether the end-of-file has been reached. If it has, the specified buffer (file) is closed.

Sample program

The following sequence of lines reads numeric data from DATA/TXT into the array A( ). When the last data character in the file is read, the EOF test in line 30 "passes", so the program branches out of the disk access loop, preventing an INPUT PAST END error from occurring. Also note that the variable I contains the number of elements input into array A( ).

```
1470 DIM A(100)       'ASSUMING THIS IS A SAFE VALUE
1480 OPEN "I", 1, "DATA/TXT"
1490 I% = 0
1500 IF EOF(1) THEN GOTO 1540
1510 INPUT#1, A(I%)
1520 I% = I% + 1
1530 GOTO 1500
1540 REM   PROG. CONT. HERE AFTER DISK INPUT
```

INPUT$
Input Specified Number of Bytes from Disk

INPUT$(length, buffer-number)
buffer-number is a sequential input file buffer (1,2,3, . . . 15)
length is the number of bytes to be input

This function is analogous to KEYBOARD INPUT$ except that it inputs data from disk rather than the keyboard.

You can use disk INPUT$ to get a certain specified number of bytes (sequential access only). INPUT$, in contrast to INPUT#, allows you to get any number of data bytes (up to 255) from disk.

Example

```
A$ = INPUT$(2, 12)
```

Inputs 12 bytes from disk into A$. File-buffer 2 is used.

Sample Program

```
2200 OPEN "I", 1, "TEST/DAT"
2210 T$ = INPUT$(1, 70)
2220 CLOSE
```

If a file TEST/DAT has been created previously, this program will open it, retrieve 70 bytes from it, store the date in T$, and close the file.

LOC
Get Current Record Number

LOC (*file number*)
file number is a numeric expression specifying the buffer for a currently-open file

LOC is used to determine the current record number, i.e., the number of the last record processed since the file was opened. It returns the record number that will be used if a GET or PUT is executed with the record number omitted.

LOC is also valid for sequential files, and gives the number of 1-byte records processed since the OPEN statement was executed.

Example

```
PRINT LOC(1)
```

Sample Program

```
   1310 A$ = "WILLIAM WILSON"
   1320 GET 1
   1330 IF N$ = A$ THEN PRINT "FOUND IN RECORD" LOC(1):
CLOSE: END
   1340 GOTO 1320
```

This is a portion of a program. Elsewhere the file has been opened and fielded. N$ is a field variable. If N$ matches A$ the record number in which it was found is printed.

LOF
Get End-of-File Record Number

LOF (*number*)
number specifies a random access buffer,
number = 1,2,. . .,15

This function tells you the number of the last, i.e., highest-numbered, record in a file. It is useful for both sequential and direct access.

Examples

```
Y = LOF(5)
```

Puts the record number into variable Y.

Sample Programs

During direct access to a pre-existing file, you often need a way to know when you've read the last valid record. LOF provides a way.

```
1540 OPEN "R", 1, "UNKNOWN/TXT"
1550 FIELD 1, 255 AS A$
1560 FOR I% = 1 TO LOF(1)   'LOF(1) = HIGHEST REC-
1570 GET 1,I%               'ORD NUM. TO BE ACCESSED
1580 PRINT A$
1590 NEXT I%
```

If you attempt to GET record numbers beyond the end-of-file record, BASIC simply fills the buffer with zeroes, and no error is generated.

When you want to add to the end of a file, LOF tells you where to start adding:

```
1600 I% = LOF(1) + 1      'HIGHEST EXISTING RECORD
1610 PUT 1,I%             'ADD NEXT RECORD
```

MKD$, MKI$, MKS$
Convert Numeric to String

MKD$(*number*), MKI$(*number*), MKS$(*number*)
number is a numeric expression.

These three functions are the inverses of CVD, CVI, and CVS. They change a number to a string. Actually, the byte values which make up the number are not changed; only one byte, the internal data-type specifier, is changed, so that numeric data can be placed in a string variable.

MKD$ returns an eight-byte string; MKI$ returns a two-byte string; and MKS$ returns a four-byte string.

Examples

```
LSET AVG$ = MKS$(0.123)
```

Sample Program

```
1350 OPEN "D", 1, "TEST/DAT"
1360 FIELD 1, 2 AS I1$, 4 AS I2$, 8 AS I3$
1370 LSET I1$ = MKI$(3000)
1380 LSET I2$ = MKS$(3000.1)
1390 LSET I3$ = MKD$(3000.00001)
1400 PUT 1
1410 CLOSE
```

Special Functions

With the special functions you can perform memory-related tasks like finding or changing the amount of total memory or string space, and discovering the absolute memory address of the value of a variable.

For example:

```
S$ = FRE(A$)
```

will find the number of bytes of string storage space left, and put this value in S$.

Other special functions, such as VARPTR and USRn, let you interface your BASIC program with machine-language programs.

FRE
Get Amount of Memory/String Space

FRE (*dummy*)
dummy is a numeric dummy argument, or a string dummy argument.

FRE returns two different values depending on its argument. If the argument is a number or numeric variable, FRE will return the total amount of memory available. If the argument is a string or string variable, FRE will return the total amount of string storage space that is available.

Examples

```
PRINT FRE(44)
```

Prints the amount of memory left.

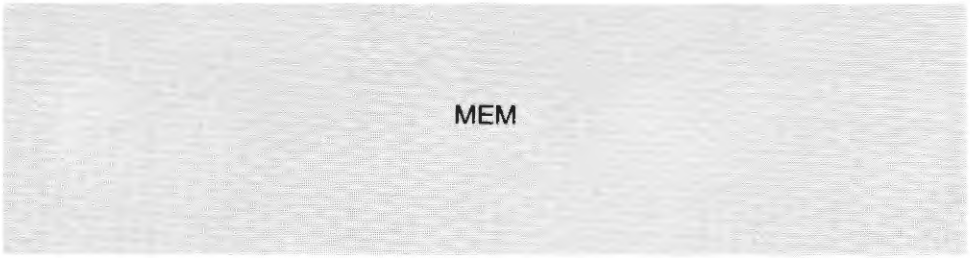
```
PRINT FRE("44")
```

Prints the amount of string space left.

Sample Program

```
1580 PRINT FRE("Z")
1590 PHRASE$ = "THE MODEL II TRS-80" +
     " IS BASED ON A Z-80 PROCESSOR."
1600 PRINT FRE("Z")
```

MEM
Get Amount of Memory



MEM performs the same function as FRE when FRE is followed by a numeric dummy argument. MEM returns the number of unused and unprotected bytes in memory. This function may be used in the immediate mode to see how much space a resident program occupies, or it may be used inside a program to avert out of memory errors by allocating less string space and dimensioning smaller array sizes. MEM requires no argument.

Example

```
PRINT MEM
```

Enter this command (in the immediate mode; no line number is needed). The number returned indicates the amount of leftover memory, i.e., memory not being used to store programs, variables, strings, stack, or not reserved for object files.

Sample Program

```
1610 IF MEM < 80 THEN 1630
1620 DIM A(15)
1630 REM      PROGRAM CONTINUES HERE
```

If fewer than 80 bytes of memory are left, control switches to another part of the program. Otherwise, an array of 15 elements is created.

VARPTR
Gets Absolute Memory Address

VARPTR (*variable name*) or (*file number*)

VARPTR returns an absolute memory address which will help you locate a value in memory. When used with a variable name, it locates the contents of that variable. When used with a file number, it returns the address of the file's data buffer. If the variable you specify has not been assigned a name, or the file has not been opened, an Illegal Function Call will occur.

VARPTR is used primarily to pass a value to a machine laguage subroutine via USRn. Since VARPTR returns an address which indicates where the value of a variable is stored, this address can be passed to a machine language subroutine as the argument of USR; the subroutine can then extract the contents of the variable with the help of the address that was supplied to it.

If VARPTR(*integer variable*) returns address K:
Address K contains the least significant byte (LSB) of 2-byte *integer*.
Address K+1 contains the most significant byte (MSB) of *integer*.

If VARPTR(*single precision variable*) returns address K:

(K)*	=	LSB of value
(K+1)	=	Next most sig. byte (Next MSB)
(K+2)	=	MSB with hidden (implied) leading one. Most significant bit is the sign of the number
(K+3)	=	exponent of value excess 128 (128 is added to the exponent).

If VARPTR(*double precision variable*) returns K:

(K)	=	LSB of value
(K+1)	=	Next MSB
(K+...)	=	Next MSB
(K+6)	=	MSB with hidden (implied) leading one. Most significant bit is the sign of the number.
(K+7)	=	exponent of value excess 128 (128 is added to the exponent).

For single and double precision values, the number is stored in normalized exponential form, so that a decimal is assumed before the MSB. 128 is added to the exponent. Furthermore, the high bit of MSB is used as a sign bit. It is set to 0 if the number is positive or to 1 if the number is negative. See examples below.

If VARPTR(*string variable*) returns K:
K = length of string
(K+1) = LSB of string value starting address
(K+2) = MSB of string value starting address
* (K) signifies "contents of address K"

The address will probably be in high RAM where string storage space has been set aside. But, if your string variable is a constant (a string literal), then it will point to the area of memory where the program line with the constant is stored, in the program buffer area. Thus, program statements like A$="HELLO" do not use string storage space.

For all of the above variables, addresses (K-1) and (K-2) will store the TRS-80 Character Code for the variable name. Address (K-3) will contain a descriptor code that tells the Computer what the variable type is. Integer is 02; single precision is 04; double precision is 08; and string is 03.

VARPTR(*array variable*) will return the address for the first byte of that element in the array. The element will consist of 2 bytes if it is an integer array; 3 bytes if it is a string array; 4 bytes if it is a single precision array; and 8 bytes if it is a double precision array.

The first element in the array is preceded by:
1. A sequence of two bytes per dimension, each two-byte pair indicating the "depth" of each respective dimension.
2. A single byte indicating the total number of dimensions in the array.
3. A two-byte pair indicating the total number of elements in the array.
4. A two-byte pair containing the ASCII-coded array name.
5. A one-byte type-descriptor (02 = Integer, 03 = String, 04 = Single-Precision, 08 – Double-Precision).

Item 1 immediately precedes the first element, Item 2 precedes Item 1, and so on.

The elements of the array are stored sequentially with the first dimension-subscripts varying "fastest", then the second, etc.

Examples

A! = 2 will be stored as follows:
2 = 10 Binary, normalized as .1E2 = .1 x 10^2
So exponent of A is 128+2 = 130 (called excess 128)
MSB of A is 10000000; however, the high bit is changed to zero since the value is positive (called hidden or implied leading one).

So A! is stored as

Exponent (K+3)	MSB (K+2)	Next MSB (K+1)	LSB (K)
130	0	0	0

A!= – .5 will be stored as

Exponent (K+3)	MSB (K+2)	Next MSB (K+1)	LSB (K)
128	128	0	0

A!=7 will be stored as

Exponent (K+3)	MSB (K+2)	Next MSB (K+1)	LSB (K)
131	96	0	0

A!=–7:

Exponent (K+3)	MSB (K+2)	Next MSB (K+1)	LSB (K)
131	224	0	0

Zero is simply stored as a zero-exponent. The other bytes are insignificant.

Example

```
Y = USR1(VARPTR(X))
```

If X is an integer value, VARPTR(X) finds the address of the least significant byte of X. This address is passed to the subroutine, which in turn passes its result to Y.

USRn
Call User's External Subroutine

USR*n* (*number*)
n specifies one of ten available USR calls, *n* = 0,1,2, . . .,9.
If *n* is omitted, zero is assumed.
number is a numeric expression, −32768 < = *number*<32768, which is passed as an integer argument to the routine.

These functions (USR0 through USR9) let you call as many as 10 machine-language subroutines and then continue execution of your BASIC program. These subroutines must have been previously defined with DEFUSR*n* statements.

"Machine language" is the low-level language used internally by your Computer. It consists of Z-80 microprocessor instructions. Machine-language subroutines are useful for special applications (things you can't do in BASIC) and for doing things very fast (like white-out the Display).

Writing such routines requires familiarity with assembly-language programming and with the Z-80 instruction set. For more information on this subject, see the Radio Shack book, *TRS-80 Assembly-Language Programming,* by William Barden, Jr.

When a USR call is encountered in a statement, control goes to the address defined in the DEFUSR*n* statement. This address specifies the entry point to your machine-language routine.

Examples

```
X = USR5(Y)
```

When this statement is executed, BASIC calls the machine-language routine USR5, previously defined in a DEFUSR5 = *address* statement.

```
Passing arguments from BASIC to the subroutine:
```

```
Upon entry to a USRn subroutine, the following register contents are
set up (for notation, see page 86 of the TRSDOS Reference Manual).

 A = Type of argument in USRn reference
     A = 8 if argument is double-precision.
     A = 4 if argument is single-precision.
     A = 2 if argument is integer.
     A = 3 is argument is string.
 HL  = When the argument is a number, this register
       points to the argument storage area (ASA)
       Described later.
```

Returning values from the subroutine to BASIC

When the USRn argument is a variable, you can modify its value by changing the ASA or string contents, as pointed to by HL or DE. For example, the statement:

```
X = USRF1(A%)
```

transfers control to the USR1 subroutine, with HL pointing to the two-byte ASA for integer variable A%. Suppose you modify the contents of this storage area. When you do a RET instruction to return to BASIC, A% will have a new value, and X will be assigned this new value.

In general, USRn(argument) will return the same type of value as argument. However, you can use BASIC's MAKINT routine to return an integer value. The address of the MAKINT routine is stored at <X'2805',X'2806'>.

For example, you might include the following code at the end of your program to return a value to BASIC.

```
MAKINT  EQU     2805H
        LD      HL,VAL          ; VAL is the value to
                                ; be returned.
        PUSH    HL              ; Save value in stack
        LD      HL,(MAKINT)     ; Restore VAL into HL
        EX      (SP),HL         ; and put MAKINT
                                ; into stack.
        RET                     ; Do MAKINT & return
                                ; to BASIC program.
```

DE = When the argument is a string, this register points points to a string descriptor, as follows:
The first byte gives the length of the string.
The next two bytes give the address where the string is stored: least significant byte (LSB) followed by most significant byte (MSB).

Description of Argument Storage Area (ASA)
For numeric values only.

For double-precision numbers:

ASA — Exponent in 128-excess form. E.g., a value of 200 indicates a 0 exponent; a value of 128 indicates a -62 exponent. A value of 0 always indicates the number is zero.

ASA-1 — Highest 7 bits of the mantissa with hidden (implied leading one. Bit 7 is the sign of the number (0 positive, 1 negative). E.g., a value of X'84' indicates the number is negative and the MSB of the mantissa is X'84'. A value of X'04' indicates the number is positive and the MSB of the mantissa is X'84'.

ASA-2 through ASA-6 — Successive 8-bit blocks of the mantissa.

ASA-7 — Lowest 8 bits of the mantissa.

For single-precision numbers:

ASA through ASA-3 same as for double-precision numbers.

For integer numbers:

ASA — LSB of the number.

ASA+1 — MSB of the number. Together, the two bytes represent the number in signed, two's complement form.

To convert the argument to integer type:

Your routine can call BASIC's FRCINT routine to put the argument into HL in 16-bit, signed two's complement form.
The address of FRCINT is stored in <X'2803',X'2804'>.

For example, you can put the following code at the beginning of your subroutine:

```
        FRCINT  EQU     2803H           ; Converts USR argument
                                        ;   to integer in HL
                LD      HL,CTNU         ; (HL) = continuation
                                        ;   address.
                PUSH    HL              ;  Save it for return
                                        ;    from FRCINT
                LD      HL,(FRCINT)     ; (HL) = force integer
                                        ;     routine
                JP      (HL)            ; Do FRCINT routine
        CTNU    ; Programs continues here with argument
                ; in HL in two's complement integer form.
```

(M2BASIC4 8/10/79)

1 / File Access Techniques

Model II BASIC provides two means of file access:

- Sequential--in which you start reading or writing data at the beginning of a file; subsequent reaads or or writes are done at following positions in the file.
- Direct--in which you start reading or writing at any record you specify. (Direct access also called random access, but "direct" is more descriptive.)

Sequential access is stream-oriented; that is, the number of characters read or written can vary, and is usually determined by delimiters in the data. Direct access is record-oriented; that is, data is always read or written in fixed-length blocks called records.

Note: When you start BASIC from TRSDOS, you select the maximum number of files you will want to have Open simultaneously. For example, the TRSDOS command line:

```
TRSDOS READY
BASIC -F:3
```

starts BASIC with a maximum of three concurrent data files, i.e., data files Open simultaneously.

To do any input/output to a disk file, you must first Open the file. When you Open the file, you specify what kind of access you want:

- "O" for sequential output
- "I" for sequential input
- "D" for direct input/output ("R" can also be used)

You also assign a file buffer for BASIC to use during file accesses. This number can be from 1 to 15, but must not exceed the number of concurrent files you requested when you started BASIC from TRSDOS. For example, if you started BASIC with 3 files, you can use buffer numbers 1, 2, and 3. Once you assign a buffer number to a file, you cannot assign that number to another file until you Close the first file.

Examples:

```
OPEN "O", 1, "TEST"
```

Creates a sequential output file named TEST on the first available drive; if TEST already exists, its previous contents are lost. Buffer 1 will be used for this file.

```
OPEN "I", 2, "TEST"
```

Opens TEST for sequential input, using buffer 2.

```
OPEN"D",1,"TEST"
```

Opens TEST for direct access, using buffer 1. If TEST does not exist, it will be created on the first available drive. Since record length is not specified, 256-byte records will be used.

```
    OPEN"D", 1, "TEST", 40
```
Same as preceding example, but 40-byte records will be used.

Sequential Access

This is the simplest way to store data in and retrieve it from a file. It is ideal for storing free-form data without wasting space between data items. You read the data back in the same order in which it was written.

There are several important points to keep in mind.

1. You must start writing at the beginning of the file. If the data you are seeking is somewhere inside, you have to read your way up to it.

2. Each time you Open a file for sequential output, the file's previous contents are lost.

3. To update (change) a sequential file, read in the file and write out the updated data to a NEW output file.

4. Data written sequentially usually includes delimiters (markers) to signify where each data item begins and ends. To read a file sequentially, you must know ahead of time the format of the data. For example, does the file consist of lines of text terminated with carriage return? does it consist of numbers separated by blank spaces? does it consist of alternting text and numeric information? etc.

5. Sequential files are always written as ASCII coded text, one byte for each character of data. For example, the number:

```
       b1.2345b
```

requires 8 bytes of disk storage, including the leading and trailing blanks that are supplied. The text string:

```
        Johnson, Robert
```

requires 15 bytes of disk storage.

6. Sequential files are always written with a record length of one. This matters if you want to Close the file and re-Open it for Direct access; in such a case, you must specify a record length of 1.

Sequential Output – An Example

Suppose we want to store a table of English-to-metric conversion constants:

English unit	Metric equivalent
1 inch	2.54001 centimeters
1 mile	1.60935 kilometers
1 acre	4046.86 sq. meters
1 cubic inch	0.01638716 liter
1 U.S. gallon	3.785 liters
1 liquid quart	0.9463 liter
1 lb (avoir)	0.45359 kilogram

First we decide what the data image is going to be. Let's say we want it to look like this:

english unit–>*metric unit*, *factor* <EN>

For example, the stored data would start out:

IN–>CM,♭2.54001♭ <EN>

The following program will create such a data file.

Note: <EN> represents a carriage return, hex OD.

```
10 OPEN"O",1,"METRIC/TXT"
20 FORI%=1 TO 7
30 READ UNIT$,FACTR
40 PRINT#1,UNIT$;",";FACTR
50 NEXT
60 CLOSE
70 DATA IN->CM,2.54001,MI->KM,1.60935,ACRE->SQ.M,4046.86
80 DATA CU.IN->LTR,1.638716E-2,GAL->LTR,3.785
90 DATA LIQ.QT->LTR,0.9463,LB->KG,0.45359
```

Line 10 creates a disk file named METRIC/TXT, and assigns buffer 1 for sequential output to that file. The extension /TXT is used because sequential output always stores the data as ASCII-coded text.

Note: If METRIC/TXT already exists, line 10 will cause all its data to be lost. Here's why: Whenever a file is opened for sequential output, the EOF marker is set to the beginning of the file. In effect, TRSDOS "forgets" that anything has ever been written beyond this point.

Line 40 prints the current contents of UNIT$ and FACTR to the file buffer. The disk-write won't actually take place until the buffer is filled or you close the file, whichever happens first. Since the string items do not contain delimiters, it is not necessary to print explicit quotes around them. The explicit comma is sufficient.

Line 60 closes the file. The EOF marker points to the end of the last data item, i.e., 0.45359, so that later, during input, DISK BASIC will know when it has read all the data.

Sequential Input – An Example

The following program reads the data from METRIC/TXT into two "parallel" arrays, then asks you to enter a conversion problem.

```
5 CLEAR 500
10 DIM UNITS$(9),FACTR(9) 'ALLOWS FOR UP TO 10 DATA PAIRS
20 OPEN"I",1,"METRIC/TXT"
25 I%=0
30 IF EOF(1) THEN 70
40 INPUT#1,UNIT$(I%),FACTR(I%)
50 I%=I%+1
60 GOTO 30
70 REM...THE CONVERSION FACTORS HAVE BEEN READ IN
100 CLS: PRINT TAB(5)"*** ENGLISH TO METRIC CONVERSIONS ***"
110 FOR ITEM%=0TOI%-1
120 PRINT USING"(## )          %        %";ITEM%,UNIT$(ITEM%)
130 NEXT
140  PRINT@704,"WHICH CONVERSION ";
150 INPUT CHOICE%
155 PRINT@768,"ENTER ENGLISH QUANTITY";
160 INPUT V
170 PRINT"THE METRIC EQUIVALENT IS"V*FACTR(CHOICE%)
180 INPUT"PRESS ENTER TO CONTINUE";X
190 PRINT@704,CHR$(31); 'CLEAR TO END OF FRAME
200 GOTO 140
```

Line 20 opens the file for sequential input. The read pointer is automatically set to the beginning of the file.

Line 30 checks to see that the end-of-file record hasn't been read. If it has, control branches from the disk input loop to the part of the program that uses the newly acquired data.

Line 40 reads a value into the string array UNIT$( ), and a number into the single-precision array FACTR( ). Note that this INPUT list parallels the PRINT# list that created the data file (see the section "Sequential Output: An example"). This parallelism is not required, however. We could just as successfully have used:

```
40 INPUT#1,UNIT$(I%): INPUT#1,FACTR(I%)
```

How to update a file

Suppose you want to add more entries into the English-Metric conversion file. You can't simply re-open the file for sequential output and PRINT# the extra data – that would immediately set the end-of-file marker to the beginning of the file, effectively destroying the file's previous contents. Do this instead:

1) Open the file for sequential input
2) Open another new data file for sequential output
3) Input a block of data and update the data as necessary
4) Output the data to the new file
5) Repeat steps 3 and 4 until all data has been read, updated, and output to the new file; then go to step 6
6) Close both files

Sequential LINE INPUT - An Example

Using the line-oriented input, you can write programs that edit other BASIC program files : renumber them, change LPRINTs to PRINTs, etc. – as long as these "target" programs are stored in ASCII format.

The following program counts the number of lines in any disk file with the extension "/TXT".

```
10 CLEAR 300
20 INPUT "WHAT IS THE NAME OF THE PROGRAM"; PROG$
30 IF INSTR(PROG$,"/TXT")=0 THEN 110 'REQUIRE /TXT EXTENSION
40 OPEN"I",1,PROG$
50 I%=0
60 IF EOF(1)THEN 90
70 I%=I%+1: LINE INPUT#1,TEMP$
80 GOTO60
90 PRINT"THE PROGRAM IS"I%"LINES LONG."
100 CLOSE: GOTO20
110 PRINT "FILESPEC MUST INCLUDE THE EXTENSION '/TXT'"
120 GOTO20
```

For BASIC programs stored in ASCII, each program line ends with an < EN > character not preceded by an < LF > line feed.
So the LINE INPUT in line 70 automatically reads one entire line at a time, into the variable TEMP$. Variable I% actually does the counting.

DIRECT ACCESS TECHNIQUES

Direct access offers several advantages over sequential access:

- Instead of having to start reading at the beginning of a file, you can read any record you specify.
- To update a file, you don't have to read in the entire file, update the data, and write it out again. You can rewrite or add to any record you choose, without having to go through any of the other records.
- Direct access is more efficient – data takes up less space and is read and written faster.
- Opening a file for direct access allows you to write to and read from the file via the same buffer.
- Direct access provides many powerful statements and functions to structure your data. Once you have set up the structure, random input/output becomes quite simple.

The last advantage listed above is also the "hard part" of direct access. It takes a little extra thought.

For the purposes of direct access, you can think of a disk file as a set of boxes – like a wall of post-office boxes. Just like the post office receptacles, the file boxes are numbered.

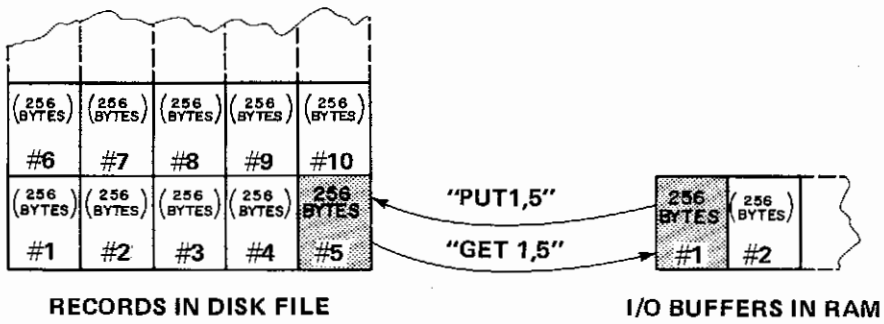
The number of boxes in a file will vary, but it's always a multiple of 5.

The smallest non-empty file contains 5 boxes, numbered 1 through 5. When the file needs more space to hold more data, TRSDOS provides it in increments of 5.

Each record may contain between 1 and 256 bytes. The length of the records is set when you create a file, in the OPEN statement.

You can place data in any record, or read the contents of any record, with statements like:

```
PUT 1,5     write buffer-1 contents to record 5
GET 1,5     read the contents of record 5 into buffer-1
```



The buffer is a waiting area for the data. Before writing data to a file, you must place it in the buffer assigned to the file. After reading data from a file, you must retrieve it from the buffer.

As you can see from the sample PUT and GET statements above, data is passed to and from the disk in 256-byte chunks.

You can place several values into the buffer before PUTting its contents into the disk file, to avoid wasting disk space.

This is accomplished by 1) dividing the buffer up into fields and naming them, then 2) placing the string or numeric data into the fields.

For example, suppose we want to store a glossary on disk. Each record will consist of a word followed by its definition. We start with:

```
100 OPEN"R",1,"GLOSSARY/BAS"
110 FIELD 1,16 AS WD$,240 AS MEANING$
```

Line 100 opens a file named GLOSSARY/BAS (creates it if it doesn't already exist); and gives buffer 1 direct access to the file.

Line 110 defines two fields onto buffer 1:
WD$ consists of the first 16 bytes of the buffer;
MEANING$ consists of the last 240 bytes.

WD$ and MEANING$ are now **field-names.**

What makes field names different? Most string variables point to an area in memory called the string space. This is where the value of the string is stored.

Field names, on the other hand, point to the buffer area assigned in the FIELD statement. So, for example, the statement:

```
10 PRINT WD$ ": " MEANING$
```

displays the contents of the two buffer fields defined above.

These values are meaningless unless we first place data in the buffer. LSET, RSET and GET can all be used to accomplish this function. We'll start with LSET and RSET, which are used in preparation for disk output.

Our first entry is the word "left-justify" followed by its definition.

```
100 OPEN"R",1,"GLOSSARY/BAS"
110 FIELD 1,16 AS WD$,240 AS MEANING$
120 LSET WD$="LEFT-JUSTIFY"
130 LSET MEANING$="TO PLACE A VALUE IN A FIELD FROM LEFT
TO RIGHT; IF THE DATA DOESN'T FILL THE FIELD, BLANKS ARE ADDED
ON THE RIGHT; IF THE DATA IS TOO LONG, THE EXTRA CHARACTERS ON
THE RIGHT ARE IGNORED. LSET IS A LEFT-JUSTIFY FUNCTION."
```

Line 120 left-justifies the value in quotes into the first field in buffer 1. Line 130 does the same thing to its quoted string. When typing in line 130, you should insert line-feed <CTRL J> characters to force line breaks as above. This makes it easier to print out the data after reading it back in to a string variable.

Note: RSET would place filler-blanks to the **left** of the item. Truncation would still be on the right.

Now that the data is in the buffer, we can write it to disk with a simple PUT statement:

```
140 PUT 1,1
150 CLOSE
```

This writes the first record into the file GLOSSARY/BAS.

To read and print the first record in GLOSSARY/BAS, use the following sequence:

```
160 OPEN"R",1,"GLOSSARY/BAS"
170 FIELD 1, 16 AS WD$,240 AS MEANING$
180 GET 1,1
190 PRINT WD$ ":  " MEANING$
200 CLOSE
```

Lines 160 and 170 are required only because we closed the file in line 150. If we hadn't closed it, we could go directly to line 180.

DIRECT ACCESS: A GENERAL PROCEDURE

The above example shows the necessary sequences to read and write using `direct` access. But it does not demonstrate the primary advantages of this form of access – in particular, it doesn't show how to update existing files by going directly to the desired record.

The program below, GLOSSACC/BAS, develops the glossary example to show some of the techniques of `direct` access for file maintenance. But before looking at the program, study this general procedure for creating and maintaining files via `direct` access.

Step Number	See GLOSSACC/BAS, Line Number
1. OPEN the file	110
2. FIELD the buffer	120
3. GET the record to be updated	140
4. Display current contents of the record (use CVD,CVI,CVS before displaying numeric data)	145-170
5. LSET and RSET new values into the fields (use MKD$,MKI$,MKS$ with numeric data before setting it into the buffer)	210-230
6. PUT the updated record	240
7. To update another record, continue at step 3. Otherwise, go to step 8.	250-260
8. Close the file	270

```
10 REM..GLOSSACC/BAS..
100 CLS: CLEAR 300
110 OPEN"R",1,"GLOSSARY/BAS"
120 FIELD 1,16 AS WD$,238 AS MEANING$,2 AS NX$
130 INPUT"WHAT RECORD DO YOU WANT TO ACCESS";R%
140 GET 1,R%
145 NX%=CVI(NX$)   'SAVE LINK TO NEXT ALPHABETICAL ENTRY
150 PRINT"WORD:  "WD$
160 PRINT"DEF'N:": PRINTMEANING$
170 PRINT"NEXT ALPHABETICAL ENTRY: RECORD#"NX%: PRINT
180 W$="": INPUT"TYPE NEW WORD<EN> OR <EN> IF OK";W$
190 D$="":PRINT"TYPE NEW DEF'N<EN> OR <EN> IF OK?":LINEINPUTD$
200 INPUT"TYPE NEW SEQUENCE NUMBER OR <EN> IF OK";NX%
210 IF W$<>""THEN LSET WD$=W$
220 IF D$<>""THEN LSET MEANING$=D$
230 LSET NX$=MKI$(NX%)
240 PUT 1,R%
245 R%=NX%  'USE NEXT ALPHA. LINK AS DEFAULT FOR NEXT RECORD
250 CLS: INPUT"    TYPE<EN> TO READ NEXT ALPHA. ENTRY,
    OR RECORD # <EN> FOR SPECIFIC ENTRY,
    OR 0 <EN> TO QUIT";R%
260 IF 0<R% THEN  140
270 CLOSE
280 END
```

Notice we've added a field, NX$, to the record (line 120). NX$ will contain the number of the record which comes next in alphabetical sequence. This enables us to proceed alphabetically through the glossary, provided we know which record contains the entry which should come first.

For example, suppose the glossary contains:

record#	word (WD$)	defn.	pointer to next alpha. entry (NX$)
1	LEFT-JUSTIFY	...	3
2	BYTE	...	4
3	RIGHT-JUSTIFY	...	0
4	HEXADECIMAL	...	1

When we read record 2 (BYTE), it tells us that record 4 (HEXADECIMAL) is next, which then tells us record 1 (LEFT-JUSTIFY) is next, etc. The last entry, record 3 (RIGHT-JUSTIFY), points us to zero, which we take to mean "THE END".

Since NX$ will contain an integer, we have to first convert that number to a two-byte string representation, using MKI$ (line 230 above).

The following program displays the glossary in alphabetical sequence:

```
300 REM. . . GLOSSOUT/BAS. . .
310 CLS: CLEAR 300
320 OPEN"R", 1, "GLOSSARY/BAS"
330 FIELD 1, 16 AS WD$, 238 AS MEANING$, 2 AS NX$
340 INPUT"WHICH RECORD IS FIRST ALPHABETICALLY"; N%
350 GET 1, N%
360 PRINT: PRINTWD$
370 PRINTMEANING$
380 N%=CVI(NX$)
390 INPUT"PRESS ENTER TO CONTINUE"; X
400 IF N%<>0 THEN 350
410 CLOSE
420 END
```

Overlapping Fields

Suppose you want to access a field in two ways – in total and in part. Then you can assign two field names to the same area of the buffer.

For example, if the first two digits of a six-digit stock-number specify a category, you might use the following field structure:

```
FIELD 1, 6 AS STOCK$, .....
FIELD 1, 2 AS CTG$, .....
```

Now STOCK$ will reference the entire stock-number field, while CTG$ will reference only the first two digits of the number.

Chapter 5

Using the Line Editor

Using the Line Editor

The Line Editor is a powerful set of subcommands which simplifies programming by making it easy to make corrections. In inputting long application programs, the Editor is a fast and efficient way to debug the program and get it running. There are two ways to activate the Editor:

F1

If you type in a long program line or an input line and realize you have made a mistake, you can also activate the Editor by hitting the F1 key before you press ENTER . This will activate the Editor and all of its subcommands listed above.

EDIT *line number*

This command starts the Editor when you want to edit program lines which have already been entered. You must specify which line you wish to edit, in one of two ways:

This command puts you in the Edit Mode. You must specify which line you wish to edit, in one of two ways:

EDIT *line-number* ENTER	Lets you edit the specified line. If line number is not in use, an FC error occurs
or	
EDIT.	Lets you edit the current program line – last line entered or altered or in which an error has occurred.

For example, type in and ENTER the following line:

```
100 FOR I = 1 TO 10 STEP .5 : PRINT I, I^2, I^3 : NEXT
```

This line will be used in exercising all the Edit subcommands described below.

Now type EDIT 100 and hit ENTER. The Computer will display:

```
100
```

This starts the Editor. You may begin editing line 100.

NOTE: EDITing a program line automatically clears all variable values and eliminates pending FOR/NEXT and GOSUB operations. If BASIC encounters a syntax error during program execution, it will automatically put you in the EDIT mode. Before EDITing the line, you may want to examine current variable values. In this case, you must type Q as your first EDIT command. This will return you to the command mode, where you may examine variable values. Any other EDIT command (typing E, pressing ENTER, etc.) will clear out all variables.

ENTER key

Hitting ENTER while in the Edit Mode causes the Computer to record all the changes you've made (if any) in the current line, and returns you to the Command Mode.

*n*Space-bar

In the Edit Mode, hitting the Space-Bar moves the cursor over one space to the right and displays any character stored in the preceding position. For example, using line 100 entered above, put the Computer in the Edit Mode so the Display shows:

```
100
```

Now hit the Space-Bar. The cursor will move over one space, and the first character of the program line will be displayed. If this character was a blank, then a blank will be displayed. Hit the Space-Bar until you reach the first non-blank character:

```
100 F
```

is displayed. To move over more than one space at a time, hit the desired number of spaces first, and then hit the Space-Bar. For example, type 5 and hit Space-Bar, and the display will show something like this (may vary depending on how many blanks you inserted in the line):

```
100 FOR I=
```

Now type 8 and hit the Space-Bar. The cursor will move over 8 spaces to the right, and 8 more characters will be displayed.

*n*BACKSPACE

Moves the cursor to the left by *n* spaces. If no number *n* is specified, the cursor moves back one space. When the cursor moves to the left, all characters in its "path" are erased from the display, but **they are not deleted from the program line.** Using this in conjunction with D or K or C can give misleading Video Displays of your program lines. So, be careful using it! For example, assuming you've used *n*Space-Bar so that the Display shows:

```
100 FOR I=1 TO 10   !
```

type 8 and hit the BACKSPACE key. The Display will show something like this:

```
100 FOR I=
```

(will vary depending on number of blanks in your line 100)

ESC

Hitting the ESC key effects an escape from any of the Insert subcommands listed below: X, I and H. After escaping from an Insert subcommand, you'll still be in the Edit Mode, and the cursor will remain in its current position. (Hitting **ENTER** is another way to exit these Insert subcommands).

L (List Line)

When the Computer is in the Edit Mode, and is not currently executing one of the subcommands below, hitting L causes the remainder of the program line to be displayed. The cursor drops down to the next line of the Display, reprints the current line number, and moves to the first position of the line. For example, when the Display shows

```
100
```

hit L (without hitting ENTER key) and line 100 will be displayed:

```
100 FOR I=1 TO 10 STEP .5 : PRINT I, I^2, I^3 : NEXT
100
```

This lets you look at the line in its current form while you're doing the editing.

X (Extend Line)

Causes the rest of the current line to be displayed, moves cursor to end of line, and puts Computer in the Insert subcommand mode so you can add material to the end of the line. For example, using line 100, when the Display shows

```
100
```

hit X (without hitting ENTER) and the entire line will be displayed; notice that the cursor now follows the last character on the line:

```
100 FOR I=1 TO 10 STEP .5 : PRINT I, I^2, I^3 :NEXT
```

We can now add another statement to the line, or delete material from the line by using the BACKSPACE key. For example, type : PRINT"DONE" at the end of the line. Now hit ENTER. If you now type LIST 100, the Display should show something like this:

```
100 FOR I=1 TO 10 STEP .5 : PRINT I, I^2, I^3 : NEXT : PRINT"DONE"
```

Note: If you want to continue editing the line, type the ESC key to get out of the "X" command mode.

I (Insert)

Allows you to insert material beginning at the current cursor position on the line. (Hitting BACKSPACE will actually delete material from the line in this mode.) For example, type and ENTER the EDIT 100 command, then use the Space Bar to move over to the decimal point in line 100. The Display will show:

```
100 FOR I=1 TO 10 STEP .
```

suppose you want to change the increment from .5 to .25. Hit the I key (don't hit ENTER) and the Computer will now let you insert material at the current position. Now hit 2 so the Display shows:

```
100 FOR I=1 TO 10 STEP .2
```

You've made the necessary change, so hit ESC to escape from the Insert Subcommand. Now hit L key to display remainder of line and move cursor back to the beginning of the line:

```
100 FOR I=1 TO 10 STEP .25 : PRINT I, I^2, I^3 : NEXT : PRINT "DONE"
100
```

You can also exit the Insert subcommand and save all changes by hitting ENTER. This will return you to Command mode.

A (Cancel and Restart)

Moves the cursor back to the beginning of the program line and cancels editing changes already made. For example, if you have added, deleted, or changed something in a line, and you wish to go back to the beginning of the line and cancel the changes already made: first hit ESC (to escape from any subcommand you may be executing); then hit A. (The cursor will drop down to the next line, display the line number and move to the first program character.

E (Save Changes and Exit)

Causes Computer to end editing and save all changes made. You must be in Edit Mode, not executing any subcommand, when you hit E to end editing.

Q (Cancel and Exit)

Tells Computer to end editing and cancel all changes made in the current editing session. If you've decided not the change the line, type Q to cancel changes and leave Edit Mode.

H (Hack and Insert)

Tells Computer to delete remainder of line and lets you insert material at the current cursor position. Hitting BACKSPACE will actually delete a character from the line in this mode. For example, using line 100 listed above, enter the Edit Mode and space over to the last statement, PRINT"DONE". Suppose you wish to delete this statement and insert and END statement. Display will show:

```
100 FOR I=1 TO 10 STEP .25 : PRINT I, I^2, I^3 : NEXT :
```

Now type H and then type END. Hit **ENTER** key. List the line:

```
100 FOR I=1 TO 10 STEP .25 : PRINT I, I^2, I^3 : NEXT : END
```

should be displayed.

Note: To continue editing the line, type the ESC key to get you out of the "H" subcommand.

*n*D (Delete)

Tells Computer to delete the specified number *n* characters to the right of the cursor. The deleted characters will be enclosed in exclamation marks to show you which characters were affected. For example, using line 100, space over to the PRINT command statement:

```
100 FOR I=1 TO 10 STEP .25 :
```

Now type 19D. This tells the Computer to delete 19 characters to the right of the cursor. The display should show something like this:

```
100 FOR I=1 TO 10 STEP .25 : /PRINT I, I^2, I^3 :/
```

When you list the complete line, you'll see that the PRINT statement has been deleted.

*n*C (Change)

Tells the Computer to let you change the specified number of characters beginning at the current cursor position. If you type C without a preceding number, the Computer assumes you want to change one character. When you have entered *n* number of characters, the Computer returns you to the Edit Mode (so you're not in the *n*C Subcommand). For example, using line 100, suppose you want to change the final value of the FOR-NEXT loop, from "10" to "15". In the Edit Mode, space over to just before the "0" in "10".

```
100 FOR I=1 TO 1
```

Now type C. Computer will assume you want to change just one character. Type 5, then hit L. When you list the line, you'll see that the change has been made.

```
100 FOR I=1 TO 15 STEP .25 : NEXT : END
```

would be the current line if you've followed the editing sequence in this chapter.

The BACKSPACE does not work as a backspace under the C command in the Editor. Instead, it replaces the character you want to change with a backspace. So it should not be used. If you make a mistake while typing in a change, Edit the line again to correct it, instead of using the BACKSPACE key.

*n*Sc (Search)

Tells the Computer to search for the *n*th occurrence of the character *c*, and move the cursor to that position. If you don't specify a value for *n*, the Computer will search for the first occurrence of the specified character. If character *c* is not found, cursor goes to the end of the line. Note: The Computer only searches through characters to the right of the cursor.

For example, using the current form of line 100, type EDIT 100 (**ENTER**) and then hit 2S:. This tells the Computer to search for the second occurrence of the colon character. Display should show:

```
100 FOR I=1 TO 15 STEP .25 : NEXT
```

You may now execute one of the subcommands beginning at the current cursor position. For example, suppose you want to add the counter variable after the NEXT statement. Type I to enter the Insert subcommand, then type the variable name, I. That's all you want to insert, so hit ESC to escape from the Insert subcommand. The next time you list the line, it should appear as:

```
100 FOR I=1 TO 15 STEP .25 : NEXT I: END
```

*n*K*c* (Search and "Kill")

Tells the Computer to delete all characters up to the *n*th occurrence of character *c*, and move the cursor to that position. For example, using the current version of line 100, suppose we want to delete the entire line up to the END statement. Type EDIT 100 (**ENTER**), and then type **2K:**. This tells the Computer to delete all characters up to the 2nd occurrence of the colon. Display should show:

```
100 FOR I=1 TO 15 STEP .25 : NEXT I/
```

The second colon still needs to be deleted, so type **D**. The Display will now show:

```
100 /FOR I=1 TO 15 STEP .25 : NEXT I//:/
```

Now hit **ENTER** and type LIST 100 (**ENTER**).

Line 100 should look something like this:

```
100 END
```

APPENDIX

A reserved word with a dollar-sign ("$") after it may be used as a numeric variable name if the dollar-sign is dropped. For instance, CHR and CHR# are valid variable names. However, DEF statements may not be used to assign values to this type of variable.

ABS
AND
ASC
ATN
AUTO
CDBL
CHR$
CINT
CLEAR
CLOCK
CLOSE
CLS
CONT
COS
CSNG
CVD
CVI
CVS
DATA
DATE$
DEF
DEFDBL
DEFFN
DEFINT
DEFSNG
DEFSTR
DEFUSR
DELETE
DIM
EDIT
ELSE
END
EOF
EQV
ERASE
ERL
ERR
ERROR
EXP
FIELD
FILES
FIX
FN
FOR
FORMAT
FRE
FREE
GET
GOSUB
GOTO
HEX$
IF
IMP
INKEY$
INPUT
INPUT$
INSTR
INT
KILL
LEFT$
LEN
LET
LINE
LINEINPUT
LIST
LLIST
LOAD
LOC
LOF
LOG
LPOS
LSET
MEM
MERGE
MID$
MKD$
MKI$
MKS$
MOD
NAME
NEW
NEXT
NOT
OCT$
ON
OPEN
OR
POINT
POS
POSN
PRINT
PUT
RANDOM
READ
REM
RENAME
RENUM
RESTORE
RESUME
RETURN
RIGHT$
RND
ROW
RSET
RUN
SAVE
SGN
SIN
SPACE$
SPC
SQR
STEP
STOP
STR$
STRING$
SWAP
SYSTEM
TAB
TAN
THEN
TIME$
TO
TROFF
TRON
USING
USR
VAL
VARPTR
VERIFY
WIDTH
XOR

MODEL II BASIC ERROR MESSAGES

CODE	ABBREVIATION	MEANING
1	NF	NEXT without FOR
2	SN	Syntax error
3	RG	Return without GOSUB
4	OD	Out of data
5	FC	Illegal function call
6	OV	Overflow
7	OM	Out of memory
8	UL	Undefined line
9	BS	Subscript out of range
10	DD	Redimensioned array
11	/0	Division by zero
12	ID	Illegal direct
13	TM	Type mismatch
14	OS	Out of string space
15	LS	String too long
16	ST	String formula too complex
17	CN	Can't continue
18	UF	Undefined user function
19	NR	No RESUME
20	RW	RESUME without error
21	UE	Unprintable error
22	MO	Missing operand
23	BO	Line buffer overflow
		DISK ERRORS
50	FO	Field overflow
51	IE	Internal error
52	BN	Bad file number
53	FF	File not found
54	BM	Bad file mode
55	AO	File already open
57	FE	Disk I/O error
58	UE	File already exists
61	RN	Disk full
62	NM	Input past end
63	MM	Bad record number
64	UE	Bad file name
66	FL	Direct statement in file
67	UE	Too many files

Glossary

access

The method in which information is read from or written to disk; see `direct` access and sequential access.

address

A location in memory, usually specified as a two-byte hexadecimal number. The address range<0 to FFFF>is represented in decimal as<0 to 32767> <-32768,...,-1>

alphabetic

Referring strictly to the letters A-Z.

alphanumeric

Referring to the set of letters A-Z and the numerals 0-9.

argument

The string or numeric quantity which is supplied to a function and is then operated on to derive a result; this result is referred to as the **value** of the function.

array

An organized set of elements which can be referenced in total or individually, using the array name and one or more subscripts. In BASIC, any variable name can be used to name an array; and arrays can have one or more dimensions. AR( ) signifies a one-dimensional array named AR; AR(,) signifies a two-dimensional array named AR; etc.

ASCII

American Standard Code for Information Interchange. This method of coding is used to store textual data. Numeric data is typically stored in a more compressed format.

ASCII format disk file

Disk files in which each byte corresponds to one character of the original data. For example, a BASIC program stored in ASCII format "looks like" the program listing, except that each character is ASCII-coded. Compare to compressed-format file.

backup disk

An exact copy of the original: a "safe copy". You should keep backups of your original TRSDOS diskette and all important data diskettes.

BASIC

Beginners' All-purpose Symbolic Instruction Code, the programming language which is stored in ROM in the TRS-80. Radio Shack supports LEVEL I BASIC, LEVEL II BASIC, and DISK BASIC. LEVEL II is a subset of DISK BASIC.

binary

Having two possible states, e.g., the binary digits 0 and 1. The binary (base 2) numbering system uses sequences of zeroes and ones to represent quantities. This is analagous to the Computer's internal representation of date, using electrical values for 0 and 1.

bit

Binary digit; the smallest unit of memory in the Computer, capable of representing the values 0 and 1.

break

To interrupt execution of a program. In BASIC the statement

```
STOP
```

causes a break in execution, as does pressing the BREAK key.

buffer

An area in RAM where data is accumulated for further processing. For example, to pass data from BASIC to a disk file, and vice-versa, the data must go through a file-buffer.

buffer field

A portion of the buffer which you define as the storage area for a buffer-field variable. Dividing a buffer into fields allows you to pass multiple values to and from disk storage.

byte

The smallest addressable unit of memory in the Computer, consisting of 8 consecutive bits, and capable of representing 256 different values, e.g., decimal values from 0 to 255.

compressed-format

A method of storing information in less space than a standard ASCII representation would require. An integer always requires two bytes; a single-precision number, four; a double-precision number, 8 — regardless of how many characters are required to represent the numbers as text. String values cannot be stored in compressed format.

BASIC programs in RAM and non-ASCII disk files are stored in compressed-format, with all BASIC keywords stored as special one-byte codes.

close

Terminate access to a disk file. Before re-accessing the file, you must re-open it.

data

Information that is passed to our output from a program; under LEVEL II and DISK BASIC, there are four types of data:

- integer numbers
- single-precision floating point numbers
- double-precision floating point numbers
- character-string sequences, or just "strings"

debug

To isolate and remove logical or syntax errors from a program.

decimal

Capable of assuming one of ten states, e.g., the decimal digits 0,1,...,9. Decimal (base 10) numbering is the everyday system, using sequences of decimal digits. Decimal numbers are stored in binary code in the Computer.

default

An action or value which is supplied by the Computer when you do not specify an action or value to be used.

delimiter

A character which marks the beginning or end of a data item, and is not a part of the data. For example, the double-quote symbol is a string delimiter to BASIC.

destination

The device or address which receives the data during a data transfer operation. For example, during a BACKUP operation, the destination disk is the one onto which the source-disk is being copied.

device

A physical part of the computer system used for data I/O, e.g., keyboard, display, line printer, cassette, disk drive, voice synthesizer.

directory

A listing of the files which are contained on a disk.

```
direct access
Direct access lets you read or write directly to a
file.  Contrast with sequential access.
```

diskette or **disk**

A magnetic recording medium for mass data storage.

drive specification or **drivespec**

An optional field in a TRSDOS file specification and in some TRSDOS commands, consisting of a colon followed by one of the digits 0 through 3. The drivespec is used to specify which drive is to be used for a disk read or write.

When the drivespec is omitted from a command involving a read operation, TRSDOS will search through all the disks for the desired file, starting with drive 0.

When the drivespec is omitted from a command involving a write operation, TRSDOS will generally search through all non write-protected drives for the desired file.

drive number

An integer value from 0 to 3, specifying one of the Disk drives.

dummy variable

A variable name which is used in an expression to meet syntactic requirements, but whose value is insignificant to the programmer.

edit

To change existing information.

entry point

The address of a machine-language program or routine where execution is to begin. This is not necessarily the same as the starting address. Entry point is also referred to as the **transfer address.**

field

A user-defined subdivision of a `direct` access file-buffer, created and named with the FIELD statement.

field name

A string variable which has been assigned to a field in a `direct` access file-buffer via the FIELD statement.

file

An organized collection of related data. Under TRSDOS, a file is the largest block of information which can be addressed with a single command. BASIC programs and data sets are stored on disk in distinct files.

file extension

An optional field in a file specification, consisting of a / followed by up to `three alphanumeric characters; the` extension can be used to identify the file type, e.g., /BAS, /TXT, /CIM, for BASIC, text, and core image, respectively.

filename

A required field in a file specification, consisting of one alphabetic followed by up to 7 alphanumeric characters. Filenames are assigned when a file is created or renamed.

file specification or **filespec**

A sequence of characters which specifies a particular disk file under TRSDOS, consisting of a mandatory filename, followed by an optional extension, password, and drivespec, and optional disk.

format

To organize a new or magnetically erased diskette into tracks and sectors, via the TRSDOS FORMAT utility.

granule

The smallest unit of allocatable space on a disk, consisting of 5 sectors.

hexadecimal or **hex**

Capable of existing in one of 16 possible states. For example, the hexadecimal digits are 0,1,2,. . ,,9,A,B,C,D,E,F. Hexadecimal (base-16) numbers are sequences of hexadecimal digits. Address and byte values are frequently given in hexadecimal form. Under DISK BASIC, hexadecimal constants can be entered by prefixing the constant with &H.

increment

The value which is added to a counter each time one cycle of a repetitive procedure is completed.

input

To transfer data from outside the Computer (from a disk file, keyboard, etc.) into RAM.

kilobyte or **K**

1024 bytes of memory. Thus a 12 K ROM includes 12*1024=12288 bytes.

logical expression

An expression which is evaluated as either True (=-1) or FALSE (=0).

logical record

A block of data which contains from 1 to 256 bytes, and can be addressed as a unit.

machine language

The Z-80 instruction set, usually specified in hexadecimal code. All higher-level languages must be translated into machine-language in order to be executed by the Computer.

null string

A string which has a length of zero; For example, the assignment

```
A$ = " "
```

makes A$ a null-string.

object code

Machine language derived from "source code", typically, from Assembly Language.

octal

Capable of existing in one of 8 states, for example, the octal digits are 0,1,...,7. Octal (base-8) numbers are sequences of octal digits. Address and byte values are frequently given in octal form. Under Model II BASIC, an octal constant can be entered by prefixing the octal number with the symbol &O.

open

To prepare a file for access by assigning a sequential input, sequential output, or random I/O buffer to it.

output

To transfer data from inside a Computer's memory to some external area, e.g., a disk file or a line printer.

parameter

Optional information supplied with a command to specify how the command is to operate. TRSDOS parameters are placed inside parentheses.

password

An optional field in a filespec consisting of up to 8 alphanumeric characters. If a file is created without a password, 8 blanks become the default password. To access a file, you must specify the password in the filespec.

Using the TRSDOS ATTRIB command, you can assign both update and access passwords; the access password will grant only a limited degree of access, while the update password grants total access to the file. See **filespec.**

prompt

A character or message provided by the Computer to indicate that it's ready to accept keyboard input.

protected file

A disk file which has a non-blank password, and therefore can only be accessed by reference to that password.

protection level

The degree of access granted by using the access password: kill, rename, write, read, or execute.

random access memory or **RAM**

Semiconductor memory which can be addressed directly and either read from or written to. See "Memory Requirements".

routine

A sequence of instructions to carry out a certain function; typically, a routine may be called from multiple points in a program. For example: keyboard scan routine.

sector

One-tenth of a track on a diskette, containing 256 bytes of storage; a TRSDOS "physical record".

sequential access

Reading from a disk file or writing to it "from start to finish", without being able to directly access a particular record in the file.

statement

A complete instruction in BASIC.

string

Any sequence of characters which must be examined verbatim for meaning: in other words, the string does not correspond to a quantity. For example, the *number* 1234 represents the same quantity as 1000+234, but the *string* "1234" does not. (String addition is actually concatenation, or stringing-together, so that: "1234" equals "1" + "2" + "3" + "4").

syntax

The "grammatical" requirements for a command or statement. Syntax generally refers to punctuation and ordering of elements within a statement. See "Notation Conventions", **General Information**, for a description of syntax abbreviations used in this manual.

transfer address

See **entry point.**

TRSDOS

TRS-80 Disk Operating System, pronounced "triss-doss". TRSDOS is supplied on disk and is then loaded into RAM.

user RAM or **user memory**

See **direct access memory.**

utility

A program or routine which serves a limited, specific purpose. There are two extended TRSDOS utilities, FORMAT and BACKUP, and two non-TRSDOS utilities, DISKDUMP/BAS and TAPEDISK.

write-protect

To physically protect a disk from being written to by leaving the write-protect notch uncovered.

INDEX

IMPORTANT NOTICE

ALL RADIO SHACK COMPUTER PROGRAMS ARE DISTRIBUTED ON AN "AS IS" BASIS WITHOUT WARRANTY

Radio Shack shall have no liability or responsibility to customer or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by computer equipment or programs sold by Radio Shack, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer or computer programs.

NOTE: Good data processing procedure dictates that the user test the program, run and test sample sets of data, and run the system in parallel with the system previously in use for a period of time adequate to insure that results of operation of the computer or program are satisfactory.

LIMITED WARRANTY

For a period of 90 days from the date of delivery, Radio Shack warrants to the original purchaser that the computer hardware described herein shall be free from defects in material and workmanship under normal use and service. This warranty is only applicable to purchases from Radio Shack company-owned retail outlets and through duly authorized franchisees and dealers. The warranty shall be void if this unit's case or cabinet is opened or if the unit is altered or modified. During this period, if a defect should occur, the product must be returned to a Radio Shack store or dealer for repair, and proof of purchase must be presented. Purchaser's sole and exclusive remedy in the event of defect is expressly limited to the correction of the defect by adjustment, repair or replacement at Radio Shack's election and sole expense, except there shall be no obligation to replace or repair items which by their nature are expendable. No representation or other affirmation of fact, including, but not limited to, statements regarding capacity, suitability for use, or performance of the equipment, shall be or be deemed to be a warranty or representation by Radio Shack, for any purpose, nor give rise to any liability or obligation of Radio Shack whatsoever.

EXCEPT AS SPECIFICALLY PROVIDED IN THIS AGREEMENT, THERE ARE NO OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE AND IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS OR BENEFITS, INDIRECT, SPECIAL, CONSEQUENTIAL OR OTHER SIMILAR DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR OTHERWISE.

RADIO SHACK A DIVISION OF TANDY CORPORATION

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA	BELGIUM	U. K.
280-316 VICTORIA ROAD RYDALMERE, N.S.W. 2116	PARC INDUSTRIEL DE NANINNE 5140 NANINNE	BILSTON ROAD WEDNESBURY WEST MIDLANDS WS10 7JN

PRINTED IN U.S.A.
8749124
8749122
8749123