NorthStar Computers Inc.
2547 Ninth Street
Berkeley, Ca. 94710

North Star
System Software Manual

Copyright © 1979, North Star Computers, Inc.

SOFT-DOC

PREFACE

This manual describes all the system software that is included with a North Star HORIZON computer or Micro Disk System. Use of the North Star Disk Operating System (DOS), Monitor, and BASIC are described in three of the major sections of this manual. The first major section, GETTING STARTED, describes the initial procedure required to begin using the North Star software.

The table of contents for all the major sections of this manual follows this preface. Two indexes for the BASIC section appear at the very end of the manual. If you receive errata sheets for this manual, be sure to incorporate all the corrections into the manual, or attach the errata sheets to the manual.

This manual applies to North Star system software diskettes stamped "RELEASE 5" or "RELEASE 5.X" where X is a digit indicating the update number. If you are working with earlier releases of North Star software, you should order a copy of the most recent release to take full advantage of all the features described in this manual. This manual covers both single-density and double-density versions of the North Star software. Differences between single- and double- density versions are noted in the text.

Other software available for your North Star system is not described here. For example, North Star Pascal and the North Star Software Exchange diskettes are not described here. Consult a North Star Catalog, Newsletter, or your local computer dealer for up-to-date descriptions of available North Star software.

Every effort has been made to ensure the accuracy of the material presented here. Nevertheless, experience shows that some textual errors always go undetected. If you find any errors, or have some suggestions on how to improve this manual, please contact North Star at the following address:

NORTH STAR COMPUTERS, INC.
ATTN SOFTWARE DOCUMENTATION
2547 NINTH STREET
BERKELEY CA 94710

I. GETTING STARTED

INTRODUCTION
A. DISK DRIVES AND DISKETTES
B. LIST OF SYSTEM SOFTWARE PROGRAMS
C. RAM ALLOCATION
D. PERSONALIZING THE DOS FOR INPUT/OUTPUT
E. SYSTEM START-UP
F. PERSONALIZING A NEW DISKETTE FROM AND OLD DISKETTE
G. INSTALLING THE INPUT/OUTPUT ROUTINES
H. HORIZON PERSONALIZED INPUT/OUTPUT ROUTINES
I. CREATING THE WORKING DISKETTE
J. HARDWARE TESTING

II. THE NORTH STAR DISK OPERATING SYSTEM (DOS)

INTRODUCTION
A. ABOUT FILES
B. COMMANDS
C. SYSTEM START-UP
D. DISK ERRORS
E. DOS LIBRARY ROUTINES
F. ADDITIONAL DOS PERSONALIZATION
G. DOS ENTRY POINTS AND FLAGS
H. UTILITIES
DT (DISK TEST)
CF (COPY FILE)
CD (COPY DISK)
CO (COMPACT)

III. THE NORTH STAR MONITOR

INTRODUCTION
A. COMMAND FORMAT
B. COMMANDS
C. HARDWARE REQUIREMENTS
D. PERSONALIZING THE MONITOR
E. EXAMPLE

IV. THE NORTH STAR BASIC SYSTEM

A. INTRODUCTION

B. BECOMING FAMILIAR WITH BASIC
1. LOADING BASIC.....B-1
2. COMMUNICATING WITH BASIC.....B-2
3. ENTERING A BASIC PROGRAM.....B-6
4. SOME BASIC CONCEPTS.....B-9

C. COMMANDS
1. PROGRAM DEVELOPMENT AND MAINTENANCE
LIST.....C-1
DEL.....C-2
SCR.....C-3
REN.....C-4
AUTO.....C-6

2. PROGRAM MAINTENANCE ON DISK
CAT.....C-7
SAVE.....C-8
NSAVE.....C-9
LOAD.....C-10
APPEND.....C-11

3. EXECUTION CONTROL
RUN.....C-12
CONTROL-C, THE PANIC BUTTON.....C-13
CONT.....C-15

4. MISCELLANEOUS COMMANDS
PSIZE.....C-16
MEMSET.....C-17
LINE (STATEMENT).....C-18
BYE.....C-20

D. USING NUMBERS
E. USING ARRAYS
F. USING STRINGS
G. THREE IMPORTANT STATEMENTS
DIM.....G-1
REM.....G-3
LET.....G-4

H. INPUT AND OUTPUT
1. STATEMENT: PRINT.....H-1
2. FORMATTED PRINTING.....H-3
3. STATEMENT: INPUT.....H-9
STATEMENT: INPUT1.....H-11
4. MULTIPLE I/O DEVICES.....H-12

IV. THE NORTH STAR BASIC SYSTEM (Continued)

I. STORING DATA WITHIN THE PROGRAM TEXT
STATEMENTS:
DATA.....I-1
READ.....I-2
RESTORE.....I-4

J. PROGRAM CONTROL
1. EXECUTION AND CONTROL FLOW.....J-1
2. STATEMENTS:
GOTO.....J-2
IF ... THEN ... ELSE.....J-3
ON ... GOTO.....J-4
STOP.....J-5
END.....J-6

3. THE FOR-NEXT LOOP
DISCUSSION.....J-7
STATEMENTS:
FOR.....J-12
NEXT.....J-13
EXIT.....J-14

4. SUBROUTINES
DISCUSSION.....J-15
STATEMENTS:
GOSUB.....J-17
RETURN.....J-18

K. FUNCTIONS
1. DISCUSSION
BUILT-IN FUNCTIONS.....K-1
USER-FUNCTIONS.....K-8

2. STATEMENTS:
DEF.....K-12
RETURN.....K-13
FNEND.....K-14

L. DATA FILES
1. DISCUSSION.....L-1
2. STATEMENTS:
CREATE.....L-10
DESTROY.....L-11
OPEN.....L-12
CLOSE.....L-13
READ#.....L-14
WRITE#.....L-16

IV. THE NORTH STAR BASIC SYSTEM (Continued)

M. ADVANCED FEATURES
1. TWO ADVANCED STATEMENTS
FILL.....M-1
OUT.....M-3

2. MACHINE LANGUAGE SUBROUTINES.....M-4
3. AUTOMATIC PROGRAM SEQUENCING.....M-6
DISCUSSION.....M-6
STATEMENT: CHAIN.....M-8

4. ERROR TRAPPING AND RECOVERY
DISCUSSION.....M-9
STATEMENT: ERRSET.....M-11

5. THE LINE EDITOR.....M-13

N. COMPATIBILITY WITH OTHER BASICS
1. STRING HANDLING.....N-1
2. INPUT TRANSLATION.....N-2
3. NORTH STAR'S BCD ARITHMETIC.....N-2
4. IF ... THEN EVALUATION.....N-3

O. MISCELLANEOUS TOPICS
1. SPECIAL ENTRY POINTS.....O-1
2. PERSONALIZING BASIC.....O-2
3. NON-STANDARD VERSIONS OF BASIC.....O-12

APPENDICES:

1. SAMPLE PROGRAMS
2. ERROR MESSAGES
3. IMPLEMENTATION NOTES
4. DECIMAL-HEX-BINARY-ASCII CONVERSION TABLE
5. BASIC TOPICS INDEX
6. BASIC KEYWORD INDEX

GETTING STARTED

INTRODUCTION

This part of the manual provides the information and procedures required to make initial use of the North Star system software. This material should be referenced at any of the following times:

A. You are about to use an assembled HORIZON computer or MICRO DISK System for the first time.

B. You have just finished assembling and checking a HORIZON or MICRO DISK System from kit.

C. You are about to use a new release system software diskette for the first time.

The sections that follow provide:

A. Information on the disk drives and how to use them.

B. Itemization of the system software provided with North Star disk systems.

C. Procedures for personalizing the DOS software to make possible input/output communication with your computer's console terminal.

D. Procedures for testing your computer's RAM memory and disk system for correct and reliable operation.

These sections should be read carefully and the specified procedures should be followed in the order given.

DISK DRIVES AND DISKETTES

Your North Star HORIZON or MICRO DISK System equipped computer includes capability for storing large amounts of data and program information on "floppy" diskettes. There may be up to four floppy disk drives connected to your computer through one disk controller board (only three drives for single density controllers). Looking at the front of a disk drive, you will see a small red LED indicator lamp, a slot running through the center of the face, and hinged door at the center, perpendicular to the slot. When the door is closed, no diskette may be inserted into the disk drive. Opening the door permits the withdrawal of the diskette and insertion, through the slot, of another diskette. When the LED light is on, it indicates that that drive is active. The disk system incorporates an automatic shut off feature which will turn off the drive motor(s) when not in use to save wear on both diskettes and disk drives.

DESCRIPTION OF DISKETTES

A diskette is a magnetically coated, thin plastic disk which is permanently sealed within a square protective jacket. The label on the jacket should be in the upper left corner as you are looking at the diskette. There are three holes in the front face of the jacket. The large hole in the middle allows the disk drive spindle to clamp directly onto the diskette in order to spin it around. Data is stored onto and retrieved from diskette much as with a phonograph record, except that, for a diskette, the needle is a magnetic record/playback head. The small, round hole to the lower right of the diskette is called the sector detect hole and is of no importance to this discussion. The large oblong hole at the bottom and the corresponding hole on the flip side expose the diskette's magnetic surface for the record/playback heads. The little square notch in the upper right corner of the diskette is a write protect notch. If you cover this notch with an adhesive tab, then the disk drive will be inhibited from writing over the information stored on the diskette. It will be read-only until the write protect tab is removed at which time both reading and writing of the diskette will be possible.

INSERTION OF DISKETTES

When you insert the diskette into the disk drive, be sure that you are holding the label edge of the diskette, and that it slides all the way in, oblong hole first, with the label facing away from the drive's LED indicator. In a HORIZON, the write protect notch should be at the top. In a horizontally oriented disk drive the notch should be at the left. After the diskette is inserted, make sure the door on the drive is locked into the closed position before you attempt to use the diskette.

CARE OF DISKETTES

Diskettes are delicate and should be handled with great care. Always observe the following rules in the handling and storing of diskettes.

1. Never directly touch the magnetic surfaces of a diskette.

2. Never bend or fold a diskette.

3. Keep a diskette in its protective envelope when not in use.

4. Never expose a diskette to heat, X-ray or other radiation, magnetic fields, moisture, or dust.

LIST OF SYSTEM SOFTWARE PROGRAMS

The following programs are included on a North Star system software diskette:

DOS	The Disk Operating System program.
CO	Utility program for compacting a diskette and optionally converting a diskette to double density.
CD	Copy diskette utility program.
CF	Copy file utility program.
DT	Disk test utility program.
BASIC	The BASIC language system program with software arithmetic.
FPBASIC	The BASIC language system program set up for use with the hardware floating point board.
M2D00	The Monitor program with origin 2D00 (hex). (M2A00 if single density diskette)
M5700	The Monitor program with origin 5700 (hex) and built-in HORIZON input/output routines.
M6700	The Monitor program with origin 6700(hex).
M0000	The Monitor program with origin 0.
MF400	The Monitor program with origin F400 (hex).

RAM ALLOCATION

The following table shows how the 64K byte RAM address space is allocated for the standard version system software and hardware. All addresses are given in hexadecimal notation. The minimum memory configuration requires 16K of RAM in the address range 2000-5FFF(hex).

SINGLE DENSITY	DOUBLE DENSITY	PROGRAMS
2000-29FF	2000-2CFF	DOS
2A00-5FFF *	2D00-5FFF *	BASIC, FPBASIC
2A00-31FF	2D00-34FF	Monitor M2A00, M2D00
2A00-3AFF	2D00-47FF	Utilities CO, CD, CF, DT
5700-5FFF	5700-5FFF	Monitor M5700
6700-6EFF	6700-6EFF	Monitor M6700
E800-EBFF	E800-EBFF	Disk Controller
EFF0-EFFF **	EFF0-EFFF **	Floating Point Board
F400-FBFF	F400-FBFF	Monitor MF400

* The upper limit of BASIC can be set by the user with the MEMSET command. It is initially set to 5FFF.

** Some floating point boards are configured to use DFF0-DFFF.

PERSONALIZING THE DOS FOR INPUT/OUTPUT

Before the North Star system software can be used, input/output routines may have to be installed in the DOS program to allow communication of the DOS with the console terminal of your computer system. This is called "personalizing" the input/output routines of the DOS. Just exactly what steps need to be taken depends on the combination of software and hardware to be used in your system.

A. You have a HORIZON computer and the console terminal is connected to the standard serial interface. In this case, the DOS on the system software diskette supplied with the HORIZON is already personalized and ready to use. Skip to the SYSTEM START-UP section. After the system is successfully started, proceed directly to the CREATING THE WORKING DISKETTE section.

B. You have a HORIZON or other computer and you have a system software diskette which has specific input/output routines installed that match the input/output configuration of your hardware. You are ready to proceed without the need for any additional personalizing of the diskette. Skip to the SYSTEM START-UP section. After the system is successfully started, proceed directly to the CREATING THE WORKING DISKETTE section. Personalized system software diskettes for the more common input/output configuations are available. Consult the North Star Product Catalog and your dealer.

C. You have an unpersonalized system software diskette but also have a different system software diskette which is already personalized for your system. This situation might occur if you have just received a new release of the system software (unpersonalized) and wish to start using it on your already running North Star system. Proceed directly to the PERSONALIZING A NEW DISKETTE FROM AN OLD DISKETTE section.

D. You have an unpersonalized system software diskette and will install the input/output routines yourself. The MICRO DISK System is supplied with such an unpersonalized diskette. This personalization procedure is not possible unless your computer system includes some capability, such as a front panel or ROM monitor, for loading the input/ouput routines into RAM memory. Furthermore, this procedure is not simple. It requires an understanding of the computer's input/output interfaces, hexadecimal numbers, and machine language programming. If all these requirements are met, then proceed directly to the INSTALLING INPUT/OUTPUT ROUTINES section.

E. You have a HORIZON computer but will not use the standard serial interface for connecting your console terminal. In this case follow the procedure described in step D.

Also, if at any time you wish to add input/output devices to the system or modify the existing routines, you must follow the procedure described in step D.

SYSTEM START-UP

Start-up of a HORIZON computer is very simple. First, load a system software diskette into drive #1. Then, turn on the computer power. The HORIZON will automatically start the disk bootstrap program which will turn on the disk drive and load the DOS into RAM from the disk. If the computer hardware is properly configured, then the system should display a DOS command prompt (* or +) and the system will be ready to use. To do a system start-up when the power is already on, depress and release the reset switch.

In a computer system other than a HORIZON which has a North Star MICRO DISK System installed in it, start-up the system as follows. First, with no diskette loaded into any disk drives, turn on the computer and disk drive power. In some computer systems, turning the power on or off while a diskette is loaded into a drive may damage the information stored on the diskette. With the power on, load the system software diskette (already personalized) in drive #1, and then cause the computer to start executing at address E800(hex). A front panel, ROM monitor, or auto-jump feature can be used to start the computer at this address. At this point the DOS software should load as described above.

If after performing the system start-up sequence, you don't get any output on you terminal, it may be because the baud rate setting of the terminal does not match the baud rate setting of the serial interface, or it may be because of some other fault in the hardware configuration (such as improperly addressed RAM boards), or it may be some problem with the input/output personalization routines. All these possibilities should be carefully examined. If typing a key causes that character to be displayed twice, it is probably because the terminal is in half duplex mode rather than full duplex mode. If some computer operations, such as the DOS list command (LI), terminate prematurely, this may be a result of an incorrectly written control-C input/output routine. Other problems may be a result of typing lower case characters for commands instead of upper case.

PERSONALIZING A NEW DISKETTE FROM AN OLD DISKETTE

If it is desired to personalize a new system software diskette using the same personalized input/ouput routines that already exist on an old diskette, then the following procedure can be used to incorporate the old routines into the new software. The new diskette may be unpersonalized or it may have input/output routines that you wish to replace. The following listing gives the DOS and Monitor commands which should be exactly followed to copy successfully the input/output routines. The DOS command prompt will be * instead of + if a single density DOS is used. The computer must include 16K of RAM starting at 2000(hex). It is assumed that the system software has standard origin at 2000(hex).

Using an old diskette, do a system start-up sequence, then:

```
+LF DOS 4000              Load copy of old DOS
```

Next, remove the old diskette and insert a new diskette in drive #1.

```
+LF DOS 5000              Load copy of new DOS
+GO M2D00                 GO M2A00 if new diskette is single density
>MM 4900,100 5800         Move I/O routines from old to new DOS
                            (See below.)
>MM 400D,C 500D           Move jumps from old to new DOS
```

The third command in the above sequence will vary, depending upon the nature of the source and destination diskettes. To transfer old I/O personalization from Release 1, 2, 3, or 4 to Release 5 dual-density DOS, use:

```
>MM 4900,100 5800
```

as above. To transfer I/O personalization between copies of the Release 5 DOS, use:

```
>MM 4800,100 5800
```

To transfer I/O personalization between copies of single-density DOS, Release 4 or earlier, use:

```
>MM 4900,100 5900
```

If the old DOS has additional personalization, copy it now.

A copy of the new DOS with the old input/ouput routines installed now resides at address 5000(hex) in RAM. Proceed to the CREATING THE WORKING DISKETTE section for directions on how to make a diskette which includes this DOS.

INSTALLING THE INPUT/OUTPUT ROUTINES

The DOS is designed to be able to interface to any conceivable terminal input/output configuration. There are four routines required by the DOS: character input (CIN), character output (COUT), control-C detect (CONTC), and terminal initialization (TINIT). In the standard version of the DOS, the input/output routines are located in the 256 byte region from 2900 to 29FF(hex).

CIN

The purpose of CIN is to obtain a single character of input from an input device and to return the value of that character in the accumulator. When CIN is called, the accumulator will contain a device number. This value, in the range 0 to 7, specifies from which of eight possible input devices the single character of input is to be obtained. Device 0 is always assumed to be the console terminal. Devices 1 to 7 may be assigned to any other input devices in the system. CIN may be written so that it ignores the device number in the accumulator if there is only one input device in the system. CIN must do a RET to the calling routine when the input character is ready in the accumulator. The accumulator is the only register which may be modified by the CIN routine. If the input routine is complex enough to require the use of other registers, their values when CIN is called must be saved, and then restored before CIN returns.

COUT

This routine sends a single character of output information to an output device. The character to be output is provided to COUT in the B-register, and the output device number is provided in the accumulator. When COUT has finished sending the output character to the appropriate device, the character itself must be in the accumulator as well as the B-register, and the routine must do a RET back to the calling routine. No registers, other than the accumulator, may be modified by the action of the COUT routine.

CONTC

This routine detects if a control-C has been typed on the console terminal. No information is passed to the routine in any of the registers, and no registers need be saved or restored by CONTC. They are all available for unrestricted use by the routine. If a control-C has been typed, the routine should set the Zero flag. If no character has been typed or if the character typed was not a control-C, then the Zero flag should instead be cleared. As soon as the Zero flag is given its proper value, CONTC must do a RET. CONTC should not wait for a character to be typed. If no character has been typed, it should do a RET immediately after clearing the Zero flag.

TINIT

Many terminals require a special initialization procedure to be followed immediately after they are turned on for use. For example, a video display controller may require that the screen be cleared before the screen is used for the first time after power on. Also, the interface electronics (such as the HORIZON standard serial interface) may require initialization after power-on or reset. The TINIT routine is called once by the DOS right after the bootstrap load and should contain any instructions which implement this one time initialization for all input/output devices used in your system. Since many terminals do not need to be initialized, you may not need to use TINIT. TINIT may freely use all registers, without having to save or restore any. The TINIT routine should do a RET when finished.

STEP BY STEP PROCEDURE

In order to personalize the DOS with input/ouput routines for your hardware configuration, perform the following steps:

1. Write your input/ouput routines carefully following all the rules specified in the above input/output routine descriptions and the DOS ENTRY POINTS AND FLAGS section of the DOS part of this manual. As examples of correct input/output routines, the following section shows the input/ouput routines for the HORIZON.

2. Perform a system start-up sequence using the unpersonalized system software diskette. In an unpersonalized diskette, each of the input/ouput routines is set up to merely do a jump to self instruction. Thus, when you first perform a system start-up sequence, the DOS will end up in a jump to self loop in TINIT, and the unpersonalized DOS will now be loaded into RAM starting at address 2000(hex).

3. Using the computer front panel or ROM monitor, stop the computer and load your input/ouput routines into RAM in the region from 2900 to 29FF(hex).

4. Once the input/output routines have been put into computer memory, you must modify the DOS jump table so that it contains the starting addresses of each of the routines. This jump table occurs from address 200D to 2018(hex). This region is 12 bytes long. Each successive 3 byte section within it consists of an 8080/Z80 JMP instruction (C3 hex) followed by the two byte starting address (low order byte first) of one of the four routines. The following table shows how the region from 200D to 2018(hex) would be modified to recognize CIN, COUT, CONTC, and TINIT if the starting addresses for these routines were 2900, 2920, 2940, and 2960(hex), respectively.

BEFORE		AFTER		
Address	Contents	Address	Contents	
200D	C3 0D 20	200D	C3 20 29	(for COUT)
2010	C3 10 20	2010	C3 00 29	(for CIN)
2013	C3 13 20	2013	C3 60 29	(for TINIT)
2016	C3 16 20	2016	C3 40 29	(for CONTC)

Note that if TINIT is not required, the byte at 2013(hex) should be changed to a RET instruction (C9 hex).

5. If you used a front panel to modify the DOS, then the stack pointer has not been changed. So continue with execution of the new TINIT routine by causing the computer to begin execution at address 2013(hex). If you used a ROM monitor to modify the DOS, then the stack pointer may have been changed but the console terminal has been initialized by the monitor. So continue by causing the computer to begin execution at address 2028(hex), the DOS continue entry point. You should see a DOS command prompt (* or +) on your terminal. If you don't, this means that the input/output routines are faulty or a mistake was made in following the above personalization steps.

6. Copy the personalized DOS at 2000 to 5000(hex) by typing the following commands:

```
+LF DOS 5000
+GO M2D00                    GO 2A00 if single density DOS
>MM 200D.C 500D
>MM 2900,100,5800
```

(If the DOS at 5000H is not Release 5 dual-density, use:

```
>MM 2900,100 5900
```

as the last command in the above sequence, replacing the one listed.)

7. Proceed directly to the CREATING THE WORKING DISKETTE section.

```
2900            *
2900            *I/O ROUTINES FOR STANDARD HORIZON COMPUTER
2900            *     IN RELEASE 4 DOS
2900            *
2900 FE02       CIN CPI 2              CHECK FOR DEVICE 2 POSSIBILITY
2902 CA2229        JZ CIN2             JUMP IF PARALLEL PORT SPECIFIED
2905 FE01          CPI 1               CHECK FOR DEVICE 1 POSSIBILITY
2907 CA1629        JZ CIN1             JUMP IF SECOND SERIAL PORT SPECIFIED
290A            *ASSUME PORT 0 (STANDARD SERIAL PORT) DESIRED
290A DB03       CIN0 IN 3              INPUT FIRST SERIAL PORT STATUS
290C E602          ANI 2               MASK INPUT STATUS BIT
290E CA0A29        JZ CIN0             LOOP IF NO CHARACTER
2911 DB02          IN 2                INPUT THE CHARACTER
2913 E67F          ANI 7FH             MASK OFF PARITY BIT
2915 C9            RET                 RETURN WITH CHARACTR IN A
2916
2916 DB05       CIN1 IN 5
2918 E602          ANI 2
291A CA1629        JZ CIN1
291D DB04          IN 4
291F E67F          ANI 7FH
2921 C9            RET
2922
2922            *SAMPLE PARALLEL INPUT CODE
2922 DB06       CIN2 IN 6              READ MOTHERBOARD STATUS
2924 E602          ANI 2               MASK TO GET THE PI FLAG
2926 CA2229        JZ CIN2             NO INPUT TYPED YET
2929 DB00          IN 0                READ DATA FROM KEYBOARD
292B F5            PUSH PSW            SAVE THE CHARACTER
292C 3E30          MVI A,30H
292E D306          OUT 6               RESET PI FLAG
2930 F1            POP PSW
2931 E67F          ANI 7FH
2933 C9            RET
2934
2934
2934 FE01       COUT CPI 1
2936 CA4929        JZ COUT1            SECOND SERIAL PORT OUTPUT
2939 FE02          CPI 2
293B CA5429        JZ COUT2            PARALLEL OPORT OUTPUT
293E            *ASSUME STANDARD SERIAL PORT OUTPUT
293E DB03       COU0 IN 3              INPUT FIRST SERIAL PORT STATUS
2940 E601          ANI 1               MASK OUTPUT STATUS BIT
2942 CA3E29        JZ COU0             LOOP IF NOT READY TO OUTPUT
2945 78            MOV A,B             MOVE CHARACTER TO A
2946 D302          OUT 2               OUTPUT THE CHARACTER
2948 C9            RET
2949 DB05       COUT1 IN 5
294B E601          ANI 1
294D CA4929        JZ COUT1
2950 78            MOV A,B
2951 D304          OUT 4
2953 C9            RET
```

```
2954             *SAMPLE PARALLEL OUTPUT ROUTINE
2954 DB06        COUT2 IN 6              READ MOTHERBOARD STATUS
2956 E601            ANI 1              MASK TO GET THE PO FLAG
2958 CA5429          JZ COUT2            PRINTER NOT YET READY
295B 78              MOV A,B            GET CHARACTER TO ACC
295C D300            OUT 0              OUTPUT TO PRINTER
295E 3E20            MVI A,20H
2960 D306            OUT 6              RESET PO FLAG
2962 78              MOV A,B        CHARACTER EXPECTED IN ACC ON RETURN
2963 C9              RET
2964
2964 DB03        CONTC IN 3          INPUT SERIAL PORT STATUS
2966 E602            ANI 2           MASK INPUT STATUS BIT
2968 EE02            XRI 2           SET Z-FLAG ONLY IF CHARACTER
296A C0              RNZ             RETURN IF NO CHARACTER TYPED
296B DB02            IN 2            INPUT THE CHARACTER
296D E67F            ANI 7FH         MASK OFF PARITY BIT
296F FE03            CPI 3           SEE IF CHARACTER IS CONTROL-C
2971 37              STC             TELL SOFTWARE A CHAR WAS TYPED (OPTIONAL)
2972 C9              RET             RETURN WITH Z-FLAG PROPERLY SET
2973
2973             *
2973             *TINIT FIRST REWRITES ALL RAM TO SET PARITY CORRECT
2973
2973 210000      TINIT LXI H,0       PREPARE TO CYCLE THROUGH RAM
2976 16E4            MVI D,BADDR/256       SET UP TO SKIP DISK REGION
2978 7C          TINKL MOV A,H       MOVE CURRENT BLOCK NUMBER TO A
2979 BA              CMP D           CHECK IF DISK BLOCK
297A C28329          JNZ TINCP       CONTINUE IF NOT DISK BLOCK
297D C604            ADI 4           ADD 1K TO RAM ADDRESS
297F 67              MOV H,A         PUT UPDATED ADDRESS BACK TO HL
2980 CA9629          JZ TINU         MAKE SURE NOT DONE IF NON-STANDARD
2983 7E          TINCP MOV A,M       READ BYTE FROM RAM
2984 77              MOV M,A         RESTORE IT WITH CORRECT PARITY
2985 2C              INR L           INCREMENT LOW ORDER ADDRESS BYTE
2986 C28329          JNZ TINCP       LOOP IF NOT AT END OF 256 BLOCK
2989 24              INR H           INCREMENT BLOCK NUMBER
298A CA9629          JZ TINU         DONE IF WE ARE BACK TO ZERO
298D 7C              MOV A,H         BLOCK NUMBER TO A
298E E603            ANI 3           MASK LOW ORDER 2 BITS
2990 C28329          JNZ TINCP       CONTINUE IF NOT AT END OF 1K BLOCK
2993 C37829          JMP TINKL       BRANCH TO MAIN LOOP
2996             *
2996             *NOW THAT ALL BYTES HAVE CORRECT PARITY, ENABLE PARITY LOGI
2996             * (IF YOU DON'T HAVE RAM-16-A WITH PARITY, THIS IS A NOP)
2996             *
2996 3E41        TINU MVI A,41H      ENABLE PARITY CODE
2998 D3C0            OUT 0C0H        MEMORY BOARD OUTPUT PORT
299A
299A             *  NOW INITIALIZE MOTHERBOARD AND SET UP BOTH SERIAL PORTS
299A AF              XRA A           ZERO ACC
299B D306            OUT 6           INITIALIZE MOTHERBOARD
299D D306            OUT 6           EXTRA
```

```
299F D306          OUT 6           EXTRA
29A1 D306          OUT 6           EXTRA
29A3 3ECE          MVI A,0CEH      2 STOPS, 16xCLOCK, 8 BITS, NO PARITY
29A5 D303          OUT 3           SEND TO FIRST SERIAL PORT
29A7 3ECE          MVI A,0CEH      SAME CODE AS FIRST PORT
29A9 D305          OUT 5           SECOND PORT
29AB 3E37          MVI A,37H       CMD: RTS, ER, RXE, DTR, TXEN
29AD D303          OUT 3           FIRST PORT
29AF 3E37          MVI A,37H       SAME CODE AS FIRST PORT
29B1 D305          OUT 5           SECOND PORT
29B3             *
29B3             *
29B3 DB02          IN 2            CLEAR STANDARD SERIAL PORT INPUT BUFFEI
29B5 DB04          IN 4            CLEAR SECOND SERIAL PORT INPUT BUFFER
29B7 3E30          MVI A,30H
29B9 D306          OUT 6           RESET PI FLAG (FOR PARALLEL PORT)
29BB C9            RET
29BC             *
```

CREATING THE WORKING DISKETTE

Before using the system software it should be copied to a diskette other than the factory supplied system software diskette. This diskette, called the WORKING DISKETTE, will be the one used on a daily basis. After this procedure is finished, the factory diskette should be retired to a safe place for storage with the write protect tab installed. If the working diskette should ever be accidentally destroyed, the factory diskette can then be used to create a new working diskette.

There are two different procedures for creating the working diskette depending on whether your computer has one or two disk drives. The procedure with two disk drives is much simpler and should be used if at all possible.

TWO DISK DRIVE PROCEDURE

Load the factory diskette with the write protect tab installed in drive #1 and a blank diskette (to become the working diskette) with no write protect tab in drive #2. Then perform the following DOS command:

```
+GO CD 1 2
```

This will copy the complete contents of the factory diskette onto the working diskette. If the factory diskette was already personalized for your system, then you are done. However, if you personalized the DOS input/output routines before coming to this section, then the personalized DOS is at 5000(hex) in RAM and should be copied to the working diskette with the following command:

```
+SF DOS,2 5000
```

Now you are done and the working diskette is ready to use.

SINGLE DISK DRIVE PROCEDURE

Load the factory diskette with write protect tab installed in drive #1 and perform the following DOS commands:

```
+RD 0 4000 8          Read file directory into RAM
+LI                   List file directory on factory diskette
```

Now remove the factory diskette and load the diskette to become the working diskette with the write protect tab removed in drive #1 and perform the following commands:

```
+IN                   Initialize working diskette
+WR 0 4000 8          Write file directory onto diskette
+LI                   List file directory
```

The listed file directory on the working diskette should be

identical to the listed file directory on the factory diskette. If you personalized the DOS input/output routines before coming to this section, then the personalized DOS is at 5000(hex) in RAM and should be copied to the working diskette with the following command:

```
+SF DOS 5000
```

Now, for each file on the factory diskette, perform a sequence like the following which copies the DOS file:

```
Load factory diskette in drive #1
+LF DOS 2D00
Load working diskette in drive #1
+SF DOS 2D00
```

The DOS file should not be copied if a personalized DOS was already copied to the diskette from RAM. After repeating the above sequence once for each file, the factory diskette will be completely copied to the working diskette. You are done and the working diskette is ready to use.

REGULAR BACKUP PROCEDURES

It is an inescapable fact that any user of a computer will make frequent mistakes in the instructions given to the computer. Most of these mistakes will be easily corrected. However, a few will cause major loss of information stored on diskettes. For example, to cite an extreme but plausible case, suppose you have spent an entire month typing a data base into your computer and it is stored on a single diskette. You now wish to initialize a new diskette and type an IN command to the DOS. It is not until the command is completed that you realize that you forgot to load the new diskette in the disk drive and that you have just initialized the diskette which held the results of one month's work. This kind of disaster can be avoided by faithfully following these two rules:

1. Always keep a write protect tab on a diskette unless you are about to write on the diskette.

2. Always make a backup copy of any file you have just changed in any significant way.

The copy disk and copy file utility programs make the backup procedure easy.

Important files or diskettes should be stored in a more permanent way. For example, a copy of the personalized working diskette should be retired to safe storage and be recovered only if the normal working diskette is destroyed.

HARDWARE TESTING

It is extremely important that you test the hardware of your computer system thoroughly using the following procedure, before using the computer for any serious work. These procedures should identify any faults or intermittent failures in the computer's RAM and disk system. These procedures should be repeated regularly in order to maintain system integrity and reliability.

RAM TEST PROCEDURE

A failure of the RAM may be the cause of almost any type of problem you may encounter while using your computer. Therefore, frequent testing of the RAM is very important. The RAM is tested with the TM command of the Monitor program. The test repeatedly writes a pattern of data into the region of RAM being tested and then reads the pattern to check that the correct pattern is indeed in the RAM. Since the test modifies the region of RAM it is testing, it is not possible to test the area where the test program itself resides. Therefore, the test procedure must be done in two steps, the first testing the last part of RAM with a Monitor program that resides in the first part of RAM and the second testing the first part of RAM with a Monitor program that resides in the last part of RAM. Start by performing a system start-up sequence and type the following command to start the Monitor:

```
+GO M2D00                    M2A00 if a single density diskette
```

With standard memory addressing, a computer with 16K of RAM will have memory in the range 2000-5FFF(hex), 32K in the range 2000-9FFF(hex), and 48K in the range 2000-DFFF(hex). Test the last part of this region with a command like the following which will test the last 8K of a 16K memory.

```
>TM 4000-5FFF 1
```

The test may run for several minutes with no apparent signs of life on the terminal. You can determine whether or not the test is still running by typing a control-C to stop the test. If the test was still running, the monitor will prompt you for another command with another (>). If nothing happens when you type the control-C, then something is wrong. If the test is allowed to run to completion, it will print the message PASS COMPLETED on the terminal and then start another pass. The program should be allowed to run for several hours to perform a thorough test.

If in the course of its operation the test detects an error in the memory it is testing, it will display on the terminal an error message of the form:

```
xxxx yy READ AS zz
```

The numbers xxxx, yy, and zz are hexadecimal. They represent the address of, the expected contents of, and the actual contents of the byte in RAM where the error was detected. If zz is always FF, then there may not be any RAM board addressed to the area being tested. Another possible cause of errors is an address conflict, for example an attempt to share the same area of memory between a RAM board and a memory mapped device, such as the disk controller or a floating point board, or another RAM board. After the test has run successfully for several hours, perform the following commands to do a similar test on the first part of RAM:

```
>OS                              Return to DOS from the Monitor
+GO M5700                        Load Monitor into last part of RAM
>TM 2000-3FFF 0
```

The M5700 Monitor has its own set of input/output routines since the RAM test will overwrite the input/output routines in the DOS at 2000(hex). Initially, the M5700 routines are personalized for use with a HORIZON. If your machine has some other hardware configuration, then the M5700 input/output routines must be changed to match your DOS routines. See the Monitor section of this manual for details. After the M5700 test has run successfully for several hours, type a control-C to stop the test and type the following command to return to the DOS.

```
>IL
```

DISK TEST PROCEDURE

In order to check for proper operation of the disk controller and disk drive(s), a DOS disk test program (DT) has been provided. This utility will repeatedly write a changing pattern to a specified drive and then attempt to read it back. Refer to the UTILITIES section of the DOS part of this manual for details on operation of the DT utility.

If each drive in your system will pass a disk test for 15 minutes, then your disk subsystem is in good operational order. If an error occurs, this may mean one of several things:

1. The diskette is improperly mounted, has a write protect tab, or has a "bad spot" which will not properly record data. If other diskettes pass the disk test, then the problem is with the diskette.

2. The disk drives are improperly connected to the system. For example, the cable connection has not been made correctly, power is not properly applied to the drives, or the drive configuration has not been done properly.

3. There is a hardware problem in the controller or drive. If your computer memory is operational, and your copy of DOS and DT have not been improperly modified, and the problem is not 1 or 2 above, then there may be a hardware problem in your disk controller or disk drive. In a multiple drive system, you can attempt to isolate the problem by testing both drives to determine if the problem is with an individual drive or not.

North Star

DISK OPERATING SYSTEM

Version 2

INTRODUCTION

The North Star DOS (Disk Operating System) was designed and implemented by staff members of North Star Computers, Inc. for use in conjunction with the North Star MICRO DISK SYSTEM, and HORIZON computer system. The DOS permits a user to issue various "commands" from a terminal for maintaining and using files on diskette. The DOS also provides "library routines" which may be called from user software. These library routines will primarily be of interest to users who will be developing their own system software, as opposed to those users who will primarily use application systems such as BASIC.

Versions of the North Star DOS are available for both single-density and double-density North Star disk systems. The DOS for single-density systems is different from the DOS for double-density systems. When reading this manual, if you have a single-density system, then ingore all references to double-density capabilities.

The DOS occupies 3.25K (D00 hex) bytes of RAM in double-density systems and 2.5K (A00 hex) bytes of RAM in single-density systems, including 256 bytes of RAM for input/output routines. No buffer area outside the DOS is required for any of the DOS commands. The origin of the DOS is 2000(hex) in both standard versions.

The North Star DOS is intended for use only with the North Star MICRO DISK SYSTEM and HORIZON computer, and no license is granted for any other use. Improved copies of the DOS, as they become available, may be obtained for a nominal charge.

Before the DOS can be used with a specific computer configuration, the instructions in the GETTING STARTED section of this manual must be followed.

ABOUT FILES

DISK ADDRESSES

Each diskette consists of 35 concentric TRACKS, and each track is subdivided into 10 SECTORS. A disk sector can hold either 512 bytes of double-density information or 256 bytes of single-density information. For purposes of discussion, a FILE BLOCK is defined to be a unit of information equal to 256 bytes. A sector can therefore contain two file blocks in double-density, or one file block in single-density. Every sector on the disk is identified by a unique DISK ADDRESS - an integer from 0 through 349. For example, sector 3 of track 27 has disk address of 273. Track 0 is the "outermost" track, and track 34 is the "innermost" track.

FILES

The primary DOS function is to permit the creation, deletion and use of files on diskettes. A file is an integral number of file blocks of data and occupies sequential disk sectors. For example, a particular file might occupy disk addresses 17 through 95 on a diskette loaded in drive #2. Note that files must always begin on sector boundaries, and that double-density files must always contain an even number of file blocks.

The first four sectors on each diskette contain a FILE DIRECTORY which specifies a symbolic name, base address, length, type, and data-density information for each file on that diskette. The symbolic name may be up to 8 characters long, and may include any characters except blank and comma. The length of a single-density file may be up to 346 blocks, and a double-density file may extend to 692 blocks. A directory may contain as many as 64 entries in single-density and 128 entries in double-density. No two files in a directory may have the same name, but it is possible for files of the same name to be in directories of diskettes loaded simultaneously on separate drives in a muliple disk drive system.

FILE TYPES

One byte in the file directory entry for each file specifies the "type" of the file. Depending on the specific type, additional bytes in the entry may have special meaning. Only four of the 127 possible file types have been assigned to date:

type 0 - Default type. All new files are assigned type 0 until explicitly changed.

type 1 - Machine language program. This file type identifies a machine language program (object code) that may be executed directly from the DOS with the GO command.

type 2 - BASIC program. This type of file is used to identify a BASIC program that can be LOADed or SAVEd from BASIC.

type 3 - BASIC data file. This type of file is the standard type for data files read and written by BASIC programs.

FILE DIRECTORY STRUCTURE

The file directory occupies disk addresses (sectors) 0 though 3. Each block in the directory holds thirty-two (sixteen in single-density systems) 16-byte entries. The symbolic name of the entry uses the first 8 bytes of an entry. An empty entry is an entry with 8 blanks (20 hex). Following the symbolic name in an entry, the disk address (2 bytes), the file size (two bytes) and the type (1 byte) follow. The last three bytes of an entry are type dependent. In particular, for a type 1 file (GO file), the two bytes following the type byte contain the go-address, and for a type 2 file (BASIC program) the byte following the type byte specifies how many file blocks of the file actually contain valid data.

File directory entry:

bytes 0-7	symbolic name of entry
bytes 8-9	disk address
bytes 10-11	number of blocks in file
byte 12	file type (high bit is 1 if double-density)
bytes 13-15	type-dependent information

COMMANDS

Instructions are issued to the DOS from the terminal by typing COMMANDS. The command format is a 2-letter mnemonic followed by any required arguments. Arguments are separated from the command mnemonic and from each other by a single blank. A command must be terminated by a carriage return before the DOS takes any action. If a typing error occurs during typing of a command, an at-sign(@) or control-N may be typed to permit re-typing of the command. Also, an underline, left-arrow, control-Q, or control-H may be typed to erase the previously typed character.

When a file name is required as a command argument, the disk drive number (in a multiple drive system) may be specified by immediately following the file name with ",1", ",2", ",3", or ",4". Drive #4 four may be specified only in double-density systems. Otherwise, drive #1 is assumed. Some sample file names are:

```
ABC    TEST1234,3    BASIC,1
```

Commands may be typed whenever the prompt character (* for single-density DOS and + for double-density DOS) appears at the left margin of the terminal.

LI <optional device specification> <optional drive number>

This command will list the entire contents of the directory on the diskette loaded in the specified drive. If no drive is specified, then drive #1 is assumed. For each file, its symbolic name, starting disk address, length, data density (single or double), and type will be printed. For type 1 files, the go-address will also be printed. To prematurely terminate a listing, a control-C may be typed. If output to a device other than the console terminal is desired, then the desired output device number may be specified by typing a # character followed by the device number. The device number must correspond to a device that has been interfaced to the system in both hardware and by adding the appropriate personalized input/output routine.

CR <file name> <length> <optional start address> <optional density>

This command will create a new file on the drive indicated by the file name. The length argument specifies the number of 256-byte blocks. If no starting address is given, then the file will start after the "last" (innermost) file currently allocated on the diskette. Otherwise, the supplied starting address will be used. The optional density specification is a single letter, "S" or "D", signifying that the file should be created in single or double density, respectively. If no density choice is specified, double-density is assumed. No density specification may be made with the single-density version of the DOS. The CR command will only create a file

directory entry - no accessing of the file itself will be done.

DE <file name>

This command will delete an existing file directory entry on the indicated drive. No actual accessing of the file blocks will be done. The DE command, in conjunction with the CR command, may be used to change the length of a file on the disk. If this is done, note that the type and type-dependent information will have to be re-entered.

TY <file name> <file type> <optional go-address>

This command is used to change the type of the specified file on the indicated drive. If type 1 is specified, then the third argument must be supplied to specify the "go-address".

GO <file name>

This command is used to load the specified file into RAM from the indicated drive and begin execution. The GO command may be used only with type 1 files. The GO command will read the entire file into RAM beginning at the go-address, and then jump to the go-address. Therefore, the first byte of the file must be the entry point of the program. The GO command sets the HL register pair to a value which points to the remainder of the command line (any characters typed after the file name) as stored in the DOS command buffer in memory. In this way, it is possible to send arguments to a program through the command string. The maximum length of a DOS command line is 20 characters.

The library routines of DOS are all included in the region of DOS preceding address 2A00(hex). For Release 5 dual-density DOS, command and I/O processing are handled by code from 2A00(hex)-2CFF(hex). It is possible to GO to a file with a GO-address in the range 2A00(hex)-2CFF(hex). However, upon return or re-entry to the DOS, the DOS routines in that region will have been overwritten, and no command processing will be possible. Instead, the Release 5 dual-density DOS will print the message:

RE-BOOT

and await an input character from the console terminal. After a system softwae diskette is loaded and a character is typed, the DOS will be re-booted from the disk.

JP <hex RAM address>

This command will cause the computer to jump to the specified

RAM address. It provides a way of executing programs which exist in the address space of the computer. Do not confuse this command with the GO command. However, like the GO command, JP sets the HL register pair to point to the remainder of the command string.

LF <file name> <hex RAM address>
SF <file name> <hex RAM address>

These commands may be used to load or save a disk file to or from RAM. The entire contents of the file will be read to or written from the area starting with the specified RAM address.

RD <disk address> <hex RAM address> <# of blocks> <optional density>
WR <disk address> <hex RAM address> <# of blocks> <optional density>

These commands may be used to read or write a specified drive directly to or from RAM. The WR and RD commands should be used with great care, as typing errors can have catastrophic effects. The disk address may optionally be followed by ",1", ",2", ",3" or ",4" to indicate a particular drive. Otherwise, drive #1 is assumed. Drive #4 may not be specified in single-density systems. The amount of data to transfer is specified as 256-byte file blocks. The optional density specification is a single letter, either "S" for single-density or "D" for double-density. If the density specification is omitted, double-density is assumed. (The single-density DOS, however, will ignore this argument.) Note that a method of copying one diskette to another in a single drive system would involve repeated use of the RD and WR commands.

IN <optional drive number> <optional density>

This command is used to initialize each new diskette to be used in the system. The IN command writes each block on the specified drive with ASCII blank characters (20 hex). The optional density argument, "S" or "D", may be used to specify whether the diskette should be initialized in single or double density format. If this argument is omitted, the diskette will be initialized to double-density. (The single-density version of the DOS will ignore the density specification.) This procedure initializes the directory and also guarantees that no "hard disk error" can result from access to an uninitialized file block. The IN command takes about 15 seconds. Needless to say, one should make sure that the proper diskette is loaded before issuing the IN command. Note that an initialized diskette does not contain a copy of the DOS. The IN command does not require any buffer area outside of the DOS memory area.

DISK SYSTEM START-UP

After power-on, or when it is desired to re-start the disk system, the 8080 or Z80 computer must be forced to begin execution at the PROM bootstrap program starting address (E800 hex in the standard version). The PROM bootstrap program will read a sector from drive #1, disk address 4 into RAM at the DOS starting address (2000 hex in the standard version). After reading in the sector, the bootstrap will branch to the DOS starting address. The program in the first block of the DOS will proceed to read in the remaining sectors of the DOS from disk starting at address 5. Then the DOS will print the prompt character (* or +) and await a command from the terminal.

Once the DOS has been started, it is no longer necessary to leave the diskette in drive #1. The DOS is fully resident in RAM, and makes no disk accesses unless asked to do so. Furthermore, the DOS does not maintain any copies of the diskette file directory in RAM between commands. Thus it is possible, for example, to obtain listings of the file directories of several diskettes by inserting them one at a time and then issuing the LI command. Also, it is possible to copy one diskette to another in a single drive system by repeatedly exchanging diskettes and doing the appropriate sequence of RD and WR commands or LF and SF commands.

DISK ERRORS

Most disk operations are tried 10 times by the DOS before reporting failure. Upon failure, an error message of the following form is printed on the console terminal:

```
DISK ERROR TYPE: x DRIVE: y SECTOR: zzz
```

where x=the error type,
y=the drive number on which the error occurred, and
zzz=the disk address at which the error occurred.

The error types have the following meanings:

1 SYNC BYTE NOT FOUND: Indicates badly written data on the diskette, or a diskette not properly loaded into the drive, or an attempt to read an uninitialized diskette.

2 CRC COMPARE ERROR: Indicates badly written data.

3 VERIFY COMPARE ERROR: Indicates data on disk does not compare with RAM in a verify operation.

4 NO INDEX PULSE: Indicates wrong type of diskette or badly loaded diskette.

5 DENSITY MISMATCH: Indicates single-density data found where double-density data was expected or visa vera.

6 WRITE PROTECT: Indicates a write operation was attempted to a write protected diskette.

If the DOS prints a question-mark(?) in response to a command, this indicates illegal form for the command or an illegal argument value.

DOS LIBRARY ROUTINES

This section describes how user machine language software may interface to the DOS for the accessing of disk files.

The DOS ENTRY POINTS AND FLAGS section shows the entry points for each of the routines to be described here. The exact interfacing requirements are described in that section. The DOS uses the stack pointer existent at call time, and some of the DOS library routines may require as much as 30 bytes of stack storage. Note that the DOS may be re-entered without using the bootstrap PROM. Now follows a discussion of each library routine.

DLOOK

This routine searches for a specified file name in the directory of the indicated disk drive. If the specified name begins with a blank, then an "empty" file directory entry is looked up. On failure to find the requested entry, HL is set to the value of the first free disk address on the indicated drive following the last file on the diskette. The file name must be in the correct syntax.

On success, HL contains a pointer into a buffer in DOS RAM that has a copy of the sought entry. The pointer addresses the first byte following the symbolic name (i.e., byte 8). Also, on return, the ACC specifies the disk drive which was determined from the name passed as argument.

DWRIT

This routine is used to write back to diskette an updated file directory entry which was previously found using DLOOK. No disk activity may occur between the DLOOK and the DWRIT call.

DCOM

This routine may be used to issue an arbitrary disk read or write command. On a read request, DCOM will try 10 times for a successful read before giving up and branching to HDERR. DCOM will fail return if the supplied arguments are out of bounds. However, great care should be used to avoid calling DCOM with incorrect arguments.

DOS

This is an entry point to the DOS command processor. It can be used to return control to a loaded DOS without requiring a PROM bootstrap load.

DOSERR

When a control-C is typed at the console terminal during a diskette directory listing, or when DOS is passed a file name which is syntactically incorrect, DOS branches to the JMP instruction stored at this location. If left unmodified, the DOSERR JMP transfers control back to a DOS error-handling routine. Modifying the address contained in

this JMP instruction will allow a user's application program to retain control under the above-named error conditions.

HDERR

HDERR branches to DOS code that prints an error message and then enters the DOS command processor. DOS branches to HDERR whenever a read attempt fails despite 10 retries. For your software to retain control in the event of a hard disk error, it must modify the address of the HDERR JMP instruction (e.g., LXI H,ADDR; SHLD HDERR+1). The stack is set to the stack pointer value before the call to DCOM. HL is set to the disk address at which the error was discovered. [Note: Software for dealing with hard disk errors is notoriously difficult. It is suggested that due to the expected low frequency of hard disk errors, for most applications the existing HDERR action will be sufficient. Hard disk errors will result primarily from careless use (e.g. forgetting to initialize a diskette, or from removing a diskette while writing is in progress). Hard disk errors can also result from power failure during writing, or from a hardware system failure.]

LIST

This routine will list the file directory of the specified drive. The listing format will be exactly the same as the listing format obtained with the DOS LI command.

OFTEN

This routine is called at least once every 40 milliseconds when DCOM has been called to perform disk operations. In the delivered copy of DOS, this routine simply does a RET. However, OFTEN may be personalized to a routine to poll for input/output requests or to enable and disable interrupts. The OFTEN routine may execute as long as is needed, and disk activity will continue when the OFTEN routine returns. OFTEN must preserve all registers except the accumulator and may only use two bytes of stack space. Note that OFTEN will be called at bootstrap load time, even before the 2900 personalization block is loaded.

Note: Here is a procedure for creating a new file using the above routines: First use DLOOK to search for the desired new name - if DLOOK succeeds then a file of that name aready exists and should not be created. On failure, HL will have the disk address which should be used as the starting address of the new file. Next, use DLOOK to find an empty directory entry by looking up a blank name. If this call to DLOOK fails, then the directory fails. On success, use the pointer in HL to copy the new file name into the directory entry, and copy in the disk address and length and type information. Finally, call DWRIT to copy the new directory entry back to the disk.

ADDITIONAL DOS PERSONALIZATION

The primary type of personalization that can be done to the DOS is the insertion of input/output routines that allow communication of the DOS and other system software with a particular hardware configuration. Input/output routine personalization is described in detail in the GETTING STARTED section of this manual. There are a number of other types of personalization that can be done to the DOS that are described in this section.

READ AFTER WRITE CHECK

If the read after write check option is turned on, then a read and verify operation is performed after every disk write operation which checks that the data written on the disk by the write operation matches what is in RAM. With this option turned on, write operations will be slower, but read operations will be the same speed. It is strongly recommended that the read after write option be turned on unless the application requires great speed of disk access. The read after write option is turned on if the byte value at address 202B(hex) of the standard version of DOS is non-zero and turned off if zero.

PAGE SIZE

The output of some devices, such as CRT's and video displays, can only display a fixed size page of information at one time. If the page size option is enabled, then the file directory listing which is output by the LI command or the LIST library routine will stop after a page of information has been output and will not display the next page until the user indicates he wishes to proceed by typing the return key. The page size option only affects the operation of the console terminal (device #0). If the byte value at address 2033(hex) of the standard version of DOS is zero, then the page size option is turned off and output will be continuous without stopping. If the value is non-zero, then the value is the number of lines on a page and the output will stop after that many less one lines of output have been displayed. The last line on the page will request the user to type return to continue. Initially, the page size option is on and set for a page size of 24 lines.

AUTOMATIC START

If the automatic start option is turned on, then a single command which is stored in the DOS input buffer is automatically executed immediately after a DOS bootstrap operation. This feature, for example, allows for the automatic loading and running of a program such as BASIC upon system start-up. Initially, the automatic start option is turned off. To turn on the option, the byte value at address 2030(hex) of the standard version of DOS should be set to zero. Initially, the value is one. Additionally, the input buffer must be setup to contain the

command which should be automatically executed. The two addresses 2031 and 2032(hex) contain the low order and high order byte, respectively, of the address of the input buffer within the DOS. The input buffer should be loaded with the ASCII values of the successive characters of the desired command. The last character of the loaded command must be a return (0D hex).

EXAMPLE PERSONALIZATION

The following listing shows an example procedure which will modify the version of DOS on a diskette so that the read after write option is turned on, the page size option is turned on and the page size is set to 24 lines, and the automatic start option is turned on and the automatic start command is set to be "GO BASIC".

Load the diskette to be modified in drive #1

```
+LF DOS 4000        Load DOS into RAM
+GO M2D00           Load and run the Monitor
>FM 402B 1          Turn on read after write option
>FM 4033 24T        Set page size to 24 lines
>FM 4030 0          Turn on automatic start option
>DH 4031,2          Determine input buffer address
402D: 31 27         Address is 2731, for example
>FM 4731 "G"        Load input buffer with "GO BASIC"
>FM 4732 "O"
>FM 4733 " "
>FM 4734 "B"
>FM 4735 "A"
>FM 4736 "S"
>FM 4737 "I"
>FM 4738 "C"
>FM 4739 D          Put return code at end of command
>OS                 Return to DOS
+SF DOS 4000        Save modified DOS back on the diskette
```

```
0000            *
0000            *NORTH STAR DISK OPERATING SYSTEM
0000            *
0000                  ORG 2000H          STANDARD VERSION ORIGIN VALUE
2000                  DS 7               THESE CELLS ARE RESERVED
2007            *
2007            *THE OFTEN ROUTINE IS CALLED FREQUENTLY DURING USE OF DCOM
2007            *BC, DE, AND HL MUST BE PRESERVED BY OFTEN.
2007            *ONLY TWO STACK BYTES ARE AVAILABLE.
2007 C9         OFTEN RET          CHANGE TO JMP INSTRUCTION
2008                  DS 2          IF ADDING YOUR OWN OFTEN ROUTINE
200A            *
200A            *THIS NEXT ENTRY IS USED BY THE BOOT PROM TO ENTER THE DOS
200A C30000     START JMP 0        0 IS NOT THE REAL ADDRESS
200D            *
200D            *THIS IS THE CHARACTER OUTPUT ROUTINE
200D            *THE CHARACTER TO BE OUTPUT MUST BE IN THE B REGISTER.
200D            *DEVICE NUMBER MAY BE SUPPLIED IN ACC, IF DESIRED.
200D            *ON RETURN THE CHARACTER MUST ALSO BE IN THE ACC.
200D            *ONLY THE ACC AND FLAGS MAY B MODIFIED
200D C30D20     COUT  JMP COUT     YOUR ROUTINE MUST DO A RET
2010            *
2010            *THIS IS THE CHARACTER INPUT ROUTINE.
2010            *DEVICE NUMBER MAY BE SUPPLIED IN ACC, IF DESIRED.
2010            *THE 7-BIT ASCII CODE MUST BE RETURNED IN THE ACC.
2010            *ONLY THE ACC AND FLAGS MAY BE MODIFIED.
2010 C31020     CIN   JMP CIN      YOUR ROUTINE MUST DO A RET
2013            *
2013            *THIS IS THE TERMINAL INITIALIZATION ROUTINE
2013            *ALL REGISTERS MAY BE USED.
2013            *IF NOT NEEDED, MERELY PATCH IN A RET.
2013 C31320     TINIT JMP TINIT
2016            *
2016            *THIS ROUTINE DETECTS A CONTROL-C
2016            *IF Z IS SET ON RETURN, THAT MEANS A CONTROL-C WAS TYPED.
2016            *OTHERWISE, IF NO CHARACTER WAS TYPED OR A CHARACTER OTHER
2016            * THAN CONTROL-C WAS TYPED, Z MUST NOT BE SET.
2016            *CONTC SHOULD RETURN IMMEDIATELY IF NO CHAR WAS TYPED,
2016            * NOT WAIT FOR A CHARACTER AND THEN RETURN.
2016            *ALL REGISTERS MAY BE USED.
2016 C31620     CONTC JMP CONTC
2019            *
```

```
2019            *DOS LIBRARY ROUTINE ENTRY POINTS, ETC.
2019            *
2019            *THIS ADDRESS IS BRANCHED TO ON HARD DISK ERRORS
2019 C30000     HDERR JMP 0       0 IS NOT THE REAL ADDRESS
201C            *
201C            *THIS IS THE FILE DIRECTORY LOOKUP ROUTINE
201C            *ACC MUST CONTAIN THE DEFAULT UNIT NUMBER (NORMALLY 1)
201C            *HL=POINTER TO LEGAL FILE NAME IN RAM, WITH OPTIONAL DRIVE
201C            * SPECIFICATION FOLLOWED BY EITHER A BLANK OR CARRIAGE RETURN.
201C            *UNIT NUMBER DETERMINED FROM NAME IS ALWAYS RETURNED IN ACC.
201C            *FAILURE IF CARRY SET. ON FAILURE, HL=FIRST FREE DISK ADDRESS
201C            *ON SUCCESS, HL HAS A POINTER TO THE EIGHT BYTE OF A COPY
201C            *OF THE DOS ENTRY IN RAM
201C C30000     DLOOK JMP 0       0 IS NOT THE REAL ADDRESS
201F            *
201F            *THIS ROUTINE WILL WRITE A DIRECTORY ENTRY BACK TO DISK
201F            *NO ARGS ARE NEEDED. MUST FOLLOW DLOOK.
201F C30000     DWRIT JMP 0       0 IS NOT THE REAL ADDRESS
2022            *
2022            *THIS ROUTINE MAY BE USED TO ISSUE A DISK COMMAND
2022            *ACC=NUMBER OF BLOCKS
2022            *B=COMMAND (0=WRITE, 1=READ, 2=VERIFY, -1=SING INIT, -2=DBL INI1
2022            *C=UNIT NUMBER, BIT 7=DOUBLE DENSITY BIT
2022            *DE=STARTING RAM ADDRESS, HL=STARTING DISK ADDRESS
2022            *RETURN WITH CARRY SET MEANS ARGUMENTS WERE ILLEGAL
2022 C30000     DCOM  JMP 0       0 IS NOT THE REAL ADDRESS
2025            *
2025            *THIS ROUTINE MAY BE USED TO LIST A FILE DIRECTORY
2025            *ACC=DISK UNIT, L=OUTPUT DEVICE NUMBER FOR LISTING
2025 C30000     LIST  JMP 0       0 IS NOT THE REAL ADDRESS
2028            *
2028            *THIS ADDRESS IS AN ENTRY POINT TO THE LOADED DOS
2028            *ENTRY HERE WILL RESET THE STACK PTR, AND NOT CALL TINIT
2028 C30000     DOS   JMP 0       0 IS NOT THE REAL ADDRESS
202B            *
202B            *THIS NEXT BYTE IS A FLAG USED BY DOS.
202B            *IF 0, THEN READ-AFTER-WRITE CHECK IS NOT DONE,
202B            *IF 1, THEN READ-AFTER-WRITE CHECK IS DONE.
202B 00         RWCHK DB 0
202C            *
202C            *THIS ADDRESS BRANCHED TO ON CONTROL-C DURING LIST OR
202C            *FILE NAME ERROR DURING DLOOK
202C C30000     DOSERR JMP 0         NOT REALLY 0
202F            *
202F            *THIS BYTE IS SET TO DENSITY AFTER DLOOK CALLS
202F            *00H IF SINGLE DENSITY, 80H IF DOUBLE DENSITY
202F            DEN DS 1
2030            *
2030            *AUTO START FLAG. NORMALLY 1 - SET TO 0 FOR TURNKEY STARTUP
2030 01         AUTOS DB 1
2031            *
2031            *NEXT TWO BYTES IDENTIFY THE LOCATION OF THE DOS INPUT BUFFER
2032 4303          DW 0           NOT REALLY 0
2033            *
2033            *NEXT BYTE SPECIFIES VIDEO TERMINAL LINE COUNT. IF 0, THEN
2033            *NO PAGING OF THE LIST COMMAND WILL BE DONE
2034 18         PAGES DB 24       INITIALIZED FOR 24 LINE TERMINAL
2034            *
```

UTILITIES

There are four operations which may be considered as part of the DOS but are actually implemented as GO files. The operations, and their corresponding GO file names are:

DT - Disk Test.
CF - Copy File.
CD - Copy Disk.
CO - Compact disk and convert to double-density.

Complete descriptions of the utilities follow. Some of the arguments to the utilities can be listed on the command line where "GO" is typed. For example

```
+GO DT 1
```

may be typed to the DOS. This tells the DT utility which drive is to be tested. Any arguments which you do not supply to the utility on the GO command line are explicitly requested by the utility.

The origin in memory of each of the utilities lies just after the end of DOS (2A00H in single-density systems and 2D00H in double-density systems). Each of the utilities requires a 5K buffer area (2.5K in single-density systems). The amount of RAM required by a utility may be computed by adding the buffer size to the size of the utility on diskette. Because the utilities load at the same address as the standard version of BASIC and many other applications programs, you should be careful that no programs or data be overwritten and therefore lost as a result of using a utility.

You may wish to use a utility to operate on a diskette different than the diskette that holds the utility program. In this case, you must change diskettes after the utility has been loaded into RAM. Each of the utilities allows a different diskette to be loaded before actually beginning its operation. Diskettes can be switched any time after the utility makes its first request for input. Do not answer that request until the switch, if any, has been made!

In the following expanded descriptions of the utilities, any references to double-density capability refer only to versions of the utilities for use on double-density systems.

Typical user-computer interaction at the terminal is given as EXAMPLES for each of the utilities. In these examples, note that the DOS prompt given is a plus-sign (+). However, single-density versions of the DOS generate an asterisk (*) as prompt. In examples, the symbol <CR> comes immediately after the user's responses to indicate that a line of user input must always be terminated by striking the RETURN key.

DT - Disk Test.

The Disk Test utility tests the specified drive and the diskette loaded in that drive. The following cycle is continuously repeated:

a) The entire diskette is written with data, starting at sector 0. An incrementing pattern is used. If the read after write check is enabled (see DOS section ADDITIONAL DOS PERSONALIZATION), then each track is verified immediately after it is written.

b) The data on the entire diskette is verified, starting at sector 0. If any sector cannot be read or contains data different than what was written, an error messgae is printed on the console terminal and the test stops.

c) If no errors have been detected by this point, the message

   PASS COMPLETED.

   is printed on the console terminal.

To terminate a disk test, type control-C. A diskette used for a disk test does not emerge from the test containing the information which was previously on it. Also, a diskette which was used for a disk test must be initialized before it is subsequently used for data storage.

EXAMPLES

```
+GO DT<CR>
DRIVE NUMBER: 2<CR>
SINGLE(S) OR DOUBLE(D) DENSITY TEST? D<CR>
LOAD DISKETTE AND PRESS RETURN TO BEGIN TEST.<CR>
PASS COMPLETE.
PASS COMPLETE.
PASS COMPLETE.
CONTROL-C STOP        User types control-C here.
+


+GO DT 2 D<CR>
LOAD DISKETTE AND PRESS RETURN TO BEGIN TEST.<CR>
DISK ERROR TYPE 3 DRIVE 2 SECTOR 352
+
```

CF - Copy File.

The Copy File utility copies the contents and type information from a source file to a destination file. The destination file may be a file which already exists, but if it does not, it is created automatically. If the destination file already exists it must be at least as large as the source file (in 256-byte file blocks). Whether the destination file exists or not, CF asks if the destination file should be written in double or single density. The source and destination files may be on different diskettes loaded on different drives, or they may be on the same diskette.

If any sectors in a source file are recorded in a density different than the density specified in the directory entry, the CF utility treats those sectors as sectors full of blanks at the specified density. No change is made to the source file, however.

Note that versions of the CF utility delivered for single-density systems only provide single-density operation.

EXAMPLES

```
+GO CF<CR>
FROM FILE: TEST<CR>
TO FILE: PROGRAM,2<CR>
EXISTING FILE. SINGLE(S) OR DOUBLE(D) DENSITY? D<CR>
COPY COMPLETED.
+


+GO CF ABC ABC1<CR>
NEW FILE. SINGLE(S) OR DOUBLE(D) DENSITY? S<CR>
COPY COMPLETED.
+
```

CD - Copy Disk.

The Copy Disk utility copies the entire contents of a diskette loaded on one specified drive to a diskette loaded in another specified drive. The source diskette may contain single-density information, double-density information, or a combination of the two. After the copy is completed, the destination diskette will contain all the same information as the source diskette, and each sector will be recorded in the same density as the source. If any information on the source diskette is impossible to read, the copy terminates. The copy operation can be retried after the bad sector has been rewritten.

EXAMPLES

```
+GO CD<CR>
COPY FROM DRIVE: 1<CR>
TO DRIVE: 2<CR>
LOAD DISKETTES AND PRESS RETURN TO BEGIN COPY.<CR>
COPY COMPLETED.
+
```

```
+GO CD 2 3<CR>
LOAD DISKETTES AND PRESS RETURN TO BEGIN COPY.<CR>
COPY COMPLETED.
+
```

CO - Compact.

The Compact utility is used to "compact" the file space on a diskette. Any unused disk space between existing files is eliminated by moving the files toward track Ø. Thus, the CO utility can be used to reclaim disk space after files have been deleted or shortened, or in case files were created in such a way as to leave gaps of disk space between them.

The CO utility also provides a second, optional function which converts a diskette to double-density format. That is, as a result of running CO, the diskette file directory will be recorded in double-density, and all files that were previously single-density files will become double-density files. Each pair of single-density file blocks (256-bytes per block) is stored in one double-density sector (512 bytes).

Before actually beginning to move files on the diskette, CO checks the file directory for any "overlapping" files. Overlapping files are any files which include at least one sector in common. Overlapping files can only be created when the optional <disk address> argument is used with the DOS CR command, or by applications programs which create such files. If any overlapping files are discovered by the CO check, the file names are printed on the console terminal and the user is given the opportunity to abort the compaction. If overlapping files exist, the compaction may yield unpredictable results. (NOTE: The special case of a file with disk address beginning at Ø is ignored by this check, and by the compaction process.)

A compaction can take from 1 to 3Ø seconds.

EXAMPLES

```
+GO CO<CR>
LOAD DISKETTE AND SPECIFY DRIVE #: 1<CR>
CONVERT TO DOUBLE DENSITY? Y<CR>
COMPACTION COMPLETED.
+


+GO CO 3<CR>
CONVERT TO DOUBLE DENSITY? Y<CR>
THE FOLLOWING FILES HAVE CONFLICTS
DATA1
TEST123
PROCEED WITH COMPACT IN PRESENCE OF CONFLICTS? N<CR>
+
```

North Star Monitor

Version 2

by Thos Sumner

INTRODUCTION

The North Star Monitor is a program which provides the user with certain maintenance and debugging functions which would normally be provided in a limited way on systems which include a control panel. The Monitor is intended to be used in conjunction with the North Star Disk Operating System (DOS). No license is extended for use of the Monitor in systems without a North Star disk controller board.

Commands to the Monitor are entered via the console using a format consistent with the DOS commands. The console is defined to be the terminal with which the DOS normally communicates - communication is done using the DOS I/O routines. When the Monitor is in COMMAND MODE, i.e., is ready to accept a command, it will print a > at the beginning of a line on the console. Command editing facilities compatible with the North Star BASIC editing features are included in the Monitor.

The following list summarizes the commands available:

- CM - Compare memory block contents
- FM - Fill memory block
- MM - Move memory block contents
- SM - Search memory block
- TM - Test memory block
- DH - Display memory hexadecimal
- DA - Display memory with ASCII interpretation
- DS - Display memory and substitute values
- JP - Jump to program
- OS - Return control to the DOS
- IL - Perform initial load from bootstrap PROM
- OD - Assign output device number for the Monitor
- ID - Assign input device number for the Monitor

A detailed description of each command appears in a later section below. All printed output from the Monitor is formatted to fit into sixty-four character lines.

INTRODUCTION

THE NORTH STAR BASIC SYSTEM

Version 6

by Jim Merritt

ABOUT NORTH STAR BASIC

North Star BASIC was created by Dr. Charles A. Grant and Dr. Mark Greenberg of North Star Computers, Inc. This manual describes version 6, an extended disk BASIC intended for use with the North Star HORIZON computer or MICRO DISK SYSTEM. Version 6 includes many features especially designed to facilitate scientific, business, and industrial applications programming. Of special note are North Star BASIC's facilities for programmed error handling, automatic program sequencing (CHAINing), formatted output, sophisticated string handling, and machine language subroutine interface. Both single line and multiple line user-function definitions are supported, as well as multiple-dimension numeric arrays, and complete disk file handling capabilities. Data files may be accessed sequentially, randomly, or on a byte by byte basis. North Star BASIC combines all these "extras" with the usual features found in any reasonable implementation of BASIC, to yield a unique development tool which promotes the writing of powerful BASIC programs. Special design features ease the task of "converting" programs written for other BASIC systems so that they will run under North Star BASIC. BASIC is also supplied in a version which uses the North Star Hardware Floating Point Board (FPB-A). The two versions, Floating Point and Non-Floating Point, are identical in features and operation but the FPB version executes arithmetic operations faster.

The North Star Version 6 BASIC software is intended for use only with the North Star HORIZON computer or MICRO DISK SYSTEM, and no license is granted for any other use. Improved copies of Version 6, as they become available, may be obtained for a nominal charge.

HOW THIS MANUAL IS ARRANGED

This manual attempts to meet the needs of both the novice programmer, with little or no BASIC background, and the experienced BASIC programmer, who needs only know the particular characteristics of North Star BASIC.

For the expert, individual STATEMENTS, COMMANDS, and other specific language features are covered in their own brief

exposition sections. Each exposition consists of the following:

SYNTAX GUIDE: This includes one or more brief models which define the form of the STATEMENT or COMMAND within North Star's BASIC syntax.

ACTION: This tells what happens when the STATEMENT or COMMAND is used.

EXAMPLES (or EXAMPLE PROGRAMS): These show the STATEMENT or COMMAND in typical use. When the feature may take a variety of forms, an attempt has been made to provide several representative examples. Frequently, the feature is illustrated in the context of a sample program or program segment.

REMARKS: Whenever necessary, this section is included to provide further information about the feature's use.

ERROR MESSAGES: Improper formation or usage of a language feature will result in a BASIC error condition which will lead to both the termination of the program or COMMAND being processed, as well as an ERROR MESSAGE sent to you. Wherever applicable, the common ERROR MESSAGES associated with improper use of a given feature, as well as their probable causes, are given in the ERROR MESSAGES section for that feature. Note that common error messages which apply generally to all STATEMENTS and COMMANDS are described in APPENDIX 2.

SEE ALSO: Here you will find cross references to relevant manual sections, study of which may help you more fully understand a given feature.

The manual includes several appendices in the back, two of which provide thorough indexing of all topics and features in the manual. Other appendices contain charts, tables, and detailed information useful to the practicing programmer.

For the beginner, there are many DISCUSSION sections, which explain the underlying concepts and capabilities of North Star BASIC. Programming methodology and strategy are also examined in these sections. This is not to say that the DISCUSSION sections should be ignored by experienced programmers. On the contrary, experts will find much useful information in these sections.

DISCUSSION and exposition sections have been interspersed throughout the manual. Furthermore, an attempt has been made to organize the manual so that elementary material is presented first, while more advanced features and concepts

are treated later. This has been done to facilitate the beginner's likely "cover to cover" approach to manual reading. While the manual is not intended as a course in BASIC programming, a thorough front to back study of it will yield much knowledge of programming in general, and programming in North Star BASIC in particular. Those who are absolute beginners in the field are referred to the introductory computer and programming texts at local libraries, book stores, and computer retail stores. If you desire instruction on the fundamentals of programming and computers, choose one such book and use it as a primer to this manual.

Finally, for all users, APPENDIX 1 contains many sample programs which illustrate the typical integration of North Star STATEMENTs and other features and capabilities into finished software.

BECOMING FAMILIAR WITH BASIC

DISCUSSION: LOADING BASIC

The procedure for loading and executing the North Star Disk Operating System (DOS), as well as for informing the software about your terminal type and memory ability, is described in the DOS section of the NORTH STAR SYSTEM SOFTWARE MANUAL.

Once the North Star DOS is loaded into RAM memory and is operating on your computer, initiating BASIC is very simple.

First, make sure that a diskette with BASIC on it is correctly seated in your #1 drive (your only drive, if you have just one!). Then, simply type

GO BASIC

in response to the DOS prompt. Don't forget to strike the RETURN key after typing GO BASIC! North Star BASIC will respond after about 2 seconds of disk-drive activity by typing READY. You are now "in" BASIC, and are ready to proceed.

DISCUSSION: COMMUNICATING WITH BASIC

This section assumes that you have gone through the steps necessary to start a session with BASIC (see DISCUSSION: LOADING BASIC), and have received a READY message, indicating that BASIC is waiting to perform directly (if it can) whatever instruction you give. In order to make most efficient use of your sessions with BASIC, you need to know several things about communicating with the system.

You will type to the system using its primary input/output (I/O) device, called the CONSOLE TERMINAL. This device will include either a printing mechanism or a video screen, as well as a keyboard, similar to that found on a typical electric typewriter. On a computer keyboard, however, there are a few symbols and extra keys which may be new to you. Note the position of "extra" keys, especially the ones marked "CONTROL" (or "CNTL", or something similar), SHIFT, RETURN (or CARRIAGE RETURN) and also at-sign (@) and underline (_), respectively. Finally, locate the "UPPER CASE" or "ALPHA LOCK" key. (If your keyboard does not have lower case capability, you need not worry about this last key.)

BASIC USES UPPER CASE

BASIC requires that instruction words given to it be typed in upper case (capital) letters. For terminals that generate lower case letters, it is necessary to force the terminal to give upper case whenever a letter key is struck. (This is so you won't have to hold down the SHIFT key every time you want to type a capital letter!) Find the mechanism which disables the generation of lower case letters from your terminal (sometimes called "UPPER CASE" or "ALPHA LOCK"), and use it. (Throughout this discussion, please refer to your terminal's own operating manual in order to learn how to find and use any special mechanisms or special keys mentioned here.)

TYPING TO BASIC

Try typing some nonsense to the system:

```
THX 1138
```

Be sure to strike the RETURN key after you finish typing a line to BASIC. This is the signal for BASIC to accept and process what you've typed. If you fail to strike the RETURN key, BASIC will patiently wait

forever for you to type more! (Striking RETURN is the same thing as saying "over" to a partner in a two-way radio conversation; it assures that each party gets the full transmission from the other, and that each waits for the proper time to speak.) Your memory may be jogged once or twice later in this manual about the necessity of that RETURN signal, however, for the most part, it will be assumed that you'll remember to end each line to BASIC by striking the RETURN key. (The notation <CR> may also be used to indicate the striking of the RETURN key, especially in examples.)

BASIC should respond to your gibberish with the message:

```
SYNTAX ERROR
```

In general, "SYNTAX ERROR" is BASIC's way of saying "I don't understand you". It usually means that you typed the right thing incorrectly, or (as in this case) the wrong thing altogether. This is an example of an ERROR MESSAGE. Such messages are sent to you in order to alert you to any difficulties which BASIC encounters as it attempts to carry out your instructions. The error message should provide a clue as to the nature of the problem, and imply the possible steps you might use to correct it. (Correcting computer problems is called "debugging". A problem itself is known as a "bug".)

Let's type something which BASIC will understand:

```
PRINT 3/2 <CR>
```

(Remember that the <CR> means to strike the RETURN key!)

You should get the answer 1.5 on the terminal.

What happens if you make a mistake in your typing? If you catch your error before striking RETURN, you can do one of two things to correct the mistake:

1) You can erase characters, one by one, until you have erased the erroneous one(s), then retype the rest of the line from that point. In standard versions of BASIC as shipped from the factory, you should strike the underline (_) key to erase the last character typed. (This character sometimes appears as a left arrow on older terminals.) An underline will appear on the terminal to help you

keep count of how many characters you have erased in this fashion. You can strike the underline key as many times as it takes you to "back up" to and erase the mistake. For example, if you typed

```
PRONT
```

you would strike the underline key 3 times. First the T would be erased, then the N, and finally the erroneous O. You would see

```
PRONT___
```

on the terminal as evidence of this. Now type

```
INT
```

to finish the word. On the terminal, it will look like

```
PRONT___INT
```

but when you strike the RETURN key, BASIC will know that this is what you really mean:

```
PRINT
```

If you have a CRT (video) terminal, you may wish to use the backspace key to "back over" the characters you erase, then retype over them. North Star BASIC permits this.

2) Using the one-character erase provided by the underline key is fine when the error is only one or, at most, a few characters back, but what happens when you type in a very long line and discover a mistake in the first part of it? To cancel a whole line before the <CR> has been struck, just type an at-sign (@). The terminal will automatically move to the next line, where you may begin typing afresh.

```
PRINR "AN EARLY ERROR@  {this line cancelled}
PRINT "ALL OK"
```

North Star BASIC provides more sophisticated ways to correct your typing errors, in the form of a LINE EDITOR. After learning a little more about BASIC and programming, see DISCUSSION: THE LINE EDITOR for further details.

CONTROL CHARACTERS

The purpose of the "CONTROL" or "CNTL" key is similar to that of the SHIFT key. However, whereas SHIFT causes upper case letters and punctuation to be generated when it is held down during typing, the CONTROL key, when held down during typing, causes generation of a new, largely invisible set of characters which are unique to computer terminals. These are the CONTROL CHARACTERS. For the most part, there is a control character "alphabet" from A to Z. You will find that many characters are useful in North Star BASIC, especially control-C, the "PANIC BUTTON", whose purpose and function is described in its own DISCUSSION section. Try using control-C now. Hold down the CONTROL key and then type C at the same time (then let up on both, of course.) You should get the message:

```
STOP
```

The reason why this happens is explained elsewhere. Note that many control characters (such as control-P) are ignored by BASIC. BASIC rings the terminal's bell (or beeps its beeper) when it ignores a character. Only certain control characters are significant to BASIC; other sections (especially that concerning the LINE EDITOR) specify which ones.

For those whose video terminals do not include an explicit "backspace" key, as mentioned earlier in this discussion, note that control-H is a substitute for "backspace".

DISCUSSION: ENTERING A BASIC PROGRAM

The rules for entering a new BASIC program at the console terminal are described in this section. An annotated example of a program-entry session follows the description of the rules.

A PROGRAM is a sequence of legal BASIC statements. One or more statements may be entered at a time on a PROGRAM LINE. This program line must be preceded by a LINE NUMBER, an integer in the range 0 to 65535.

When entering program lines, you signal the computer to accept a newly-entered line by strikng the RETURN key.

1) If the line number of the newly-entered line doesn't match a line number in any existing program line, then the line is simply ADDED to the program.

2) If the line number of the newly-entered line duplicates that of an existing line, the new line REPLACES the old line of that number.

3) If the RETURN key is struck immediately after typing only a line number, and the line number corresponds to that of an existing program line, that line is DELETED from the program.

4) Typing in the command SCR (with no line number) results in the immediate erasure of the entire current program.

5) To store a program onto diskette for the very first time, the NSAVE command is used. Updating the program afterwards is accomplished with the SAVE command.

The RUN command causes BASIC to begin executing the current program. This session also includes use of the LIST and CAT commands, which print a LISTing of the current program, and a CATalog of the names of programs and other files on a diskette, respectively, to the terminal. Note that commands are typed in without line numbers, and are executed immediately.

Further details on the PRINT statement, and the CAT, LIST, NSAVE, SAVE, SCR, and RUN commands are available in later sections of this manual.

The content of the following dialogue between person

and computer (in the left-hand column) has been chosen to illustrate the simple rules for entering a BASIC program and saving it on diskette.

You are invited to take the programmer's role in the "script". The lines which you type into the system always have the symbol <CR> at the end. This is to remind you to strike the RETURN key after typing each line of input. All the other lines in the "script" are BASIC's responses to you. To help you better understand what is going on with each new line in the dialogue, comments are provided in the right-hand column. These are NOT part of the program, and you should not attempt to type anything from that column to BASIC.

{DIALOGUE}	{Comments}
READY	BASIC is READY to work with you.
SCR<CR>	Erases any previous program.
READY	BASIC's response to your SCR.
LIST<CR>	You want to see the current program. There is none.
READY	BASIC ends its program LISTing.
10 PRINT 6/4<CR>	You enter 2 program lines (note
20 PRINT "WELCOME TO BASIC"<CR>	required line numbers).
5 PRINT "FIRST PROGRAM"<CR>	This line is out of sequence.
LIST<CR>	Check to see what you've done.
5 PRINT "FIRST PROGRAM"	Program LISTS out. Note that
10 PRINT 6/4	BASIC has put program lines in
20 PRINT "WELCOME TO BASIC"	proper sequence.
READY	
RUN<CR>	Now, RUN the program and see results!
FIRST PROGRAM	Note that quotes aren't printed.
1.5	Note that you get RESULT of 6/4.
WELCOME TO BASIC	Again, no quotes.
READY	
10<CR>	Typing the line number erases line.
LIST<CR>	
5 PRINT "FIRST PROGRAM"	Line 10 is now gone.
20 PRINT "WELCOME TO BASIC"	
READY	
30 PRINT 2+2<CR>	A new line, 30, is added.
LIST<CR>	
5 PRINT "FIRST PROGRAM"	
20 PRINT "WELCOME TO BASIC"	
30 PRINT 2+2	New line 30 is put in its proper place.
READY	

```
RUN<CR>                                 See how the modified program RUNs.

FIRST PROGRAM
WELCOME TO BASIC
 4                                      Again, you get result; here, of 2+2
READY
CAT<CR>                                 Get listing of programs on diskette.
DOS          4   10 D    0             Depending on which diskette you use,
BASIC        9   50 D    1 2D00           you may get a different CATalog.
READY
NSAVE FIRST<CR>                         Save the program into new file "FIRST".
READY
CAT<CR>                                 Now, check to see that it's there.
DOS          4   10 D    0
BASIC        9   50 D    1 2D00
FIRST       34    4 D    2             It has been added to the diskette.
READY
30 PRINT 2-2<CR>                        Replace line 30.  Current program
LIST<CR>                                   is now DIFFERENT from one on disk.

5 PRINT "FIRST PROGRAM"                 LISTing verifies that an addition
20 PRINT "WELCOME TO BASIC"                has been made.  (Though we do it
30 PRINT 2-2                               for explanation, you need not
READY                                      LIST after every change.)
NSAVE FIRST<CR>                         Attempt to update the disk file.
ARG ERROR                               NSAVE is WRONG command to update.
READY                                   Use it only once for each program file.
SAVE FIRST<CR>                          From then on, use SAVE.
READY                                   Note that, with save, update is OK.
SCR<CR>                                 Erase the current program.
READY
LIST<CR>                                Verify that it is gone.

READY
LOAD FIRST<CR>                          Now, get it back from diskette!
READY                                   When writing programs, SAVE often to
LIST<CR>                                   be sure that disk file holds
                                           most current version!
5 PRINT "FIRST PROGRAM"                 Note that the program in "FIRST"
20 PRINT "WELCOME TO BASIC"                has been retrieved.  If it had not
30 PRINT 2-2                               been on a disk file, it would have
READY                                      had to have been retyped after SCR.
RUN<CR>

FIRST PROGRAM                           Verify that it RUNs.
WELCOME TO BASIC
 0
READY                                   And now, move onward!
```

DISCUSSION: SOME BASIC CONCEPTS

The North Star BASIC system has two modes of operation:

> DIRECT MODE, in which lines typed to the system are executed without delay;
>
> PROGRAM MODE in which the system executes instructions which have been stored previously in the form of a PROGRAM.

Prior to learning how to work with BASIC in these modes, you must understand certain concepts and terminology, which are explained in this section.

A COMMAND is a special type of instruction which may be executed only in direct mode, never as part of a program. Commands generally provide services which are not meaningful or useful while a program is RUNning.

For example, the command LIST generates a listing of the program currently in the BASIC program/data area of memory. (This is called the CURRENT PROGRAM.) It is a rare application which requires a program to list itself, and so the LIST function is a command. Each command is described in detail in its own section of the manual.

NOTE: The following paragraph uses advanced terminology which is defined elsewhere in the manual.

String and numeric arguments to commands may only be literals. The use of other types of expressions as arguments is not allowed. Moreover, disk file names in commands are not quoted. These restrictions on argument representation are the biggest difference between commands and direct statements, which will be discussed later.

A STATEMENT is a BASIC instruction which may be used as part of a PROGRAM. Typical among statements are PRINT, which causes information to be output to a terminal, and REM, which "does nothing", but provides a way for the programmer to insert REMarks about the workings of the program into the program itself.

Statements begin with a KEYWORD from which the statement derives its name. (PRINT is both a keyword and a statement name.) The keyword may be followed by ARGUMENTs and other keywords. An argument is a

piece of information on which the statement operates, or which is used to modify the operation of the statement. For example, the string literal "HI" is the argument of the following statement:

```
PRINT "HI"
```

A BASIC program is structured as a sequence of LINEs, each containing one or more statements. A line starts with a LINE NUMBER, which is an INTEGER (that is, a whole number) in the range 0 to 65535. A statement follows the line number, and the combination is called a PROGRAM LINE. A typical line is

```
70 PRINT "THIS IS ONE STATEMENT."
```

More than one statement may exist on a program line, as long as individual statements on that line are separated by a backslash (\) character. Here is an example of a multiple-statement program line with three statements:

```
835 A = 0 \ B = 0 \ REM INITIALIZE A AND B
```

Many statements may be executed in direct mode in order to get immediate results. This is accomplished by typing a statement without preceding it with a line number. Such a statement is called a DIRECT STATEMENT, and is executed as soon as it has been completely typed (indicated by striking the RETURN key). PRINT is such a statement. If, for example, you type

```
PRINT 3+3
```

into BASIC, you will immediately get back 6 on the terminal. This ability to use PRINT in direct mode, and therefore immediately generate the results of arithmetic expressions is sometimes called "calculator mode". As long as you put the keyword PRINT in front of numeric expressions, you may use your computer as a powerful desk-type calculator. This will have no effect on the current program.

Direct statements should not be confused with commands. A direct statement differs from a command in that it may also be executed as part of a program, by being included on a program line, whereas a command may only be executed in direct mode. Each command and statement has its own rules as to what constitutes its proper form and when it can be used

correctly.

The following statements may be executed as direct statements:

DIM	IF...THEN...ELSE	OPEN
IF...THEN	FILL	CLOSE
PRINT	OUT	READ#
LET	CREATE	WRITE#
RESTORE	DESTROY	CHAIN

COMMAND:

```
LIST
LIST <line number interval>
LIST <device number expression>
LIST <device number expression>, <line number interval>
```

ACTION: Prints the text of the current program. The optional device expression is formed by following a cross-hatch (#) with a single digit from 0 to 7, corresponding to an active output device. If no device is given, device #0 (the console terminal) is assumed. If the line number interval is specified, only the program lines numbered within that interval will be LISTed. The interval is formed as follows:

<single line number> -- only the specified line number will be LISTed.
<single line number>, -- all lines from the specified line number to the end of the program will be LISTed.
<line number>, <line number> -- all program lines from the first specified line number to the second will be LISTed.

If no interval is given, the entire program will be LISTed.

EXAMPLES:

```
LIST
LIST 1000
LIST 30,
LIST 100,200
LIST #1
LIST #3,30,700
```

REMARKS: The 2nd line number in the interval (if given) should be greater than or equal to the first.

For the convenience of users with CRT screens, the program listing may automatically be "paged". Refer to DISCUSSION: PERSONALIZING BASIC for details.

ERROR MESSAGES:

LINE NUMBER ERROR
One or both of the lines specified in the line number interval do not exist within the current program.

OUT OF BOUNDS ERROR
One or both of the line numbers specified in the line number interval do not lie in the range 0 to 65535.

COMMAND: DEL <line number>, <line number>

ACTION: All program lines within the given interval are DELeted from the current program. The second line number must be strictly greater than the first.

EXAMPLES:
```
DEL 10,20
DEL 1000,1075
```

REMARKS: DEL is used to DELete whole blocks of program lines at one time. If it is desired to remove only one line, just type the appropriate line number, followed immediately by striking the RETURN key.

All variables are cleared as a result of DEL (or any other command which modifies the current program).

Unless the DELeted lines have been SAVEd as part of a program on diskette, they will be permanently lost and will have to be re-entered manually if needed later.

ERROR MESSAGES:

ARG ERROR
The second line number in the interval is not greater than the first.

LINE NUMBER ERROR
One or both of the lines specified in the line number interval do not exist within the current program.

OUT OF BOUNDS ERROR
One or both of the line numbers specified in the line number interval are less than 0 or greater than 65535.

SEE ALSO: DISCUSSION: COMMUNICATING WITH BASIC
COMMAND: SCR

COMMAND: SCR

ACTION: SCR erases (SCRatches) the current program and any existing variables from the user workspace.

EXAMPLE: SCR

REMARKS: SCR is used to clear the workspace prior to entering a new program.

Only the current program is affected. Any copies of the program existing on diskette remain unaltered.

Unless a copy of the program exists on diskette or some other storage medium, the only way it can be retrieved after SCR is to retype it by hand. Therefore, it is important to make copies of the program on diskette before using SCR, if that program, or parts of it, will be used later.

ERROR MESSAGES: None.

SEE ALSO: COMMAND: SAVE
COMMAND: NSAVE
COMMAND: LOAD
COMMAND: DESTROY
COMMAND: DEL

COMMAND: REN
REN <line number>
REN <line number>, <increment value>

ACTION: The entire current program is RENumbered. The first line in the program is given the line number specified in the REN command (10 if no line number is specified). If a line number is given, then an optional increment value may be added to the command. All line numbers will automatically be separated by the given increment value (10, if no increment is specified). The increment value, if used, must be an integer, from 1 to 32767.

EXAMPLES:
```
REN
REN 1000
REN 1000,100
```

After the command

```
REN 100
```

program A will be changed to program B:

program A

```
1 REM READS AND PRINTS DATA
2 REM IN LINE 1000
3 READ Z
10 IF Z<0 THEN 2000
70 PRINT Z \ GOTO 3
1000 DATA 1,2,3,-1
3000 REM LINE 2000 HASN'T YET BEEN WRITTEN
```

program B

```
100 REM READS AND PRINTS DATA
110 REM IN LINE 1000
120 READ Z
130 IF Z<0 THEN 2000
140 PRINT Z \ GOTO 120
150 DATA 1,2,3,-1
160 REM LINE 2000 HASN'T YET BEEN WRITTEN
```

REMARKS: RENumbering is usually done to produce a uniform increment value between statement numbers so that inserting new statements becomes more convenient.

It is not possible to specify an increment value without giving a line number as well, but a line number may be specified without an accompanying increment value, in which case the increment is

assumed to be 1Ø.

Note that, while program references to line numbers (such as those found in GOTO, GOSUB, RESTORE, and similar statements) are modified to reflect the program's new line number structure, references to line numbers in REM statements remain unchanged.

If a GOTO, GOSUB, RESTORE, or similar statement in the original program references a non-existent line number, that reference will remain unaltered after a RENumbering operation.

ERROR MESSAGES: (If any of the following errors occurs, no RENumbering is performed.)

OUT OF BOUNDS ERROR
This error is produced in any of the following situations:

1) The line number specified in the command is greater than 65535;

2) The increment value is greater than 32767, or less than 1;

3) The combination of starting line number and increment value would result in a program where some line numbers would necessarily be greater than 65535.

ARG ERROR
The line number or the increment value specified is not a positive integer, or the two values are not separated by a comma.

SEE ALSO: COMMAND: AUTO

COMMAND: AUTO
AUTO <initial line number>
AUTO <initial line number>, <increment value>

ACTION: Initiates automatic line numbering mode, in which BASIC will automatically generate new line numbers for successive lines of program text. The specified line number will be the first line number used in auto-mode. Each successive automatically-supplied line number will be incremented from the last by the specified increment value. The increment value must be an integer in the range of 1 to 65535. An increment value may not be supplied unless an initial line number is also provided. When an initial line number or increment value is not given, it is assumed to be 10.

EXAMPLES:
```
AUTO
AUTO 400
AUTO 1000,100
```

REMARKS: In automatic line numbering mode, a new line number will be printed at the start of every line.

Auto-mode will persist until one of the following occurs:

a) a carriage return is typed immediately after the line number;

b) a line without a line number is typed (by using the North Star BASIC line-editing capabilities to delete the line number from the beginning of the line);

c) the next automatically-generated line number would be greater than 65535.

Note that if the "automatic" line numbers overlap existing lines in the current program, the existing lines will be REPLACED by the new ones.

ERROR MESSAGES: OUT OF BOUNDS ERROR
Either the initial line number, the increment value, or both are greater than 65535 or less than 0.

ARG ERROR
Either the initial line number, the increment value, or both are negative, or non-integers.

SEE ALSO: COMMAND: REN

COMMAND: CAT
CAT <drive number>
CAT <output device expression>
CAT <output device expression>, <drive number>

ACTION: A catalog listing of the files on the diskette loaded in the specified disk drive is printed on the specified output device. The output device expression must consist of a cross-hatch (#) followed by a single digit from 0 to 7. The drive number must be a single digit from 1 to 4. If the output device expression is omitted, the catalog listing is sent to the console terminal (device #0). If the drive number isn't specified, it is assumed to be drive #1.

EXAMPLES:
```
CAT        {drive #1's catalog to console}
CAT 3      {drive #3's catalog to console}
CAT #1     {drive #1's catalog to device #1}
CAT #2,3   {drive #3's catalog to device #2}
```

REMARKS: The listing produced is identical to that obtained through use of the DOS LI command.

Like the program LISTing, the CATalog may be "paged", but this is a function of the DOS rather than of BASIC. See the DOS section of this manual for details.

The user should be sure that the output device expressions and/or drive numbers (when specified) refer to existing devices and drives, respectively.

ERROR MESSAGES: HARD DISK ERROR
This error occurs under the following circumstances:

1) The specified drive is not installed in the system.
2) The power to the specified drive is not on.
3) The diskette is not properly seated within the specified drive (drive door is open, etc.).
4) There is no directory on the diskette in the specified drive.
5) The directory on diskette has been destroyed.

FILE ERROR
The drive number specified is greater than 4.

SEE ALSO: DOS section of this manual.

COMMAND: SAVE <file name>

ACTION: The current program is permanently SAVEd into an existing BASIC program (type 2) file on diskette.

EXAMPLES:
```
SAVE PROG      {PROG is on diskette in drive #1}
SAVE TEST7,2   {TEST7 is on diskette in drive #2}
```

REMARKS: The specified file must be of sufficient size to hold the program for the SAVE to be successful.

It is possible to SAVE the null program onto a program file. (This can be accomplished by using the SCRatch command immediately prior to SAVE.) This effectively "erases" any program which was previously stored in that file.

SAVE doesn't change the current program/data space in any way, so it is possible to use the CONT command after SAVE should one be performed during a program interruption caused by control-C or the STOP statement.

ERROR MESSAGES:

OUT OF BOUNDS ERROR
The current program is too big to fit in the specified file.

FILE ERROR
The specified file name is improper. It
a) is too long;
b) contains illegal characters (i.e. comma or blank);
c) specifies an illegal drive number.

The FILE ERROR also occurs when the diskette in the specified drive is write protected.

ARG ERROR
The specified file does not exist.

TYPE ERROR
The specified file is not a BASIC program (type 2) file.

HARD DISK ERROR
Refer to COMMAND: CAT.

SEE ALSO:
COMMAND: NSAVE
COMMAND: LOAD
DOS section of this manual.

COMMAND: NSAVE <file name>
NSAVE <file name> <file size>

ACTION: The specified BASIC program file is created on diskette to the desired size in file blocks, and the current program is SAVEd into it. If no file size is specified, three file blocks are added to the actual size of the current program, and the resulting number is taken as the file size. The density of the file created is set to be the same density as that of the file directory on the diskette.

EXAMPLES:
```
NSAVE PROGRAM
NSAVE GREEN,2 25
NSAVE MPG,3
NSAVE BIGPROG 50
```

REMARKS: NSAVE is merely a special form of the SAVE command, and is used to SAVE a program for which a diskette file does not yet exist.

A FILE BLOCK is 256 bytes of information. Note that in double-density format, two file blocks are stored on each disk sector. In single-density format, a file block and a disk sector are equivalent in size.

When doing an NSAVE, the size specified in creating the file should allow for the eventual expansion of the program. When a program becomes too large to be SAVEd in a file, then a longer file will have to be used.

ERROR MESSAGES: An attempted NSAVE may result in any of the errors possible when using SAVE. The following are unique to NSAVE:

ARG ERROR
In addition to its causes under SAVE, an ARG ERROR may also occur during an NSAVE if the specified file already exists on diskette.

FILE ERROR
In addition to its causes under SAVE, a FILE ERROR may occur during an NSAVE if there is not room enough on the diskette for the new program file.

SEE ALSO: COMMAND: SAVE
COMMAND: PSIZE

COMMAND: LOAD <file name>

ACTION: The BASIC program contained in the specified file is LOADed into the program/data area and becomes the current program.

EXAMPLES:
```
LOAD PROG3        {load from drive #1}
LOAD TEST8,2      {load from drive #2}
```

REMARKS: The specified file must be of type 2.

The successful LOAD command performs a SCRatch of the program/data area before LOADing the program.

ERROR MESSAGES:

TOO LARGE OR NO PROGRAM ERROR
Either the program in the specified file is too big to fit in the program/data area, or the file does not contain a valid BASIC program. In either case, a SCRatch of the program/data area occurs. (See COMMAND: MEMSET and DISCUSSION: PERSONALIZING BASIC for information on how to increase the size of BASIC's program/data area in order to avoid this error.)

HARD DISK ERROR
Refer to COMMAND: CAT. Depending on the point during the LOAD operation at which such an error occurs, a memory SCRatch may be performed.

If an attempted LOAD results in any of the following errors, no change in the program/data area occurs. Specifically, all variables will retain their values, the current program will remain, and, if the abortive LOAD occurs during a program whose execution has been interrupted by control-C or the execution of a STOP statement, the CONT command may still be used to resume program execution.

FILE ERROR
See COMMAND: SAVE

ARG ERROR
See COMMAND: SAVE

TYPE ERROR
See COMMAND: SAVE

SEE ALSO: COMMAND: SAVE
COMMAND: SCR

COMMAND: APPEND <file name>

ACTION: APPENDs the BASIC program in the specified diskette file to the end of the current program. (The lowest line number in the specified program must be greater than the largest line number in the current program in order for an APPEND to be successful.)

EXAMPLES:
```
APPEND MYPROG
APPEND TESTER,2
```

REMARKS: If there is no current program, APPEND acts like LOAD.

A successful APPEND will always clear all variables in the program/data area.

ERROR MESSAGES:

LINE NUMBER ERROR
The lowest number in the program to APPEND is less than or equal to the highest number in the current program.

TOO LARGE OR NO PROGRAM ERROR
Either there is not a valid BASIC program in the specified file, or the program which would result from the APPEND operation is too large to fit into available memory. In the latter case, the current program remains unmodified.

Please refer to COMMAND: LOAD and COMMAND: SAVE for details on the following errors which may also occur during an attempted APPEND:

HARD DISK ERROR
FILE ERROR
ARG ERROR
TYPE ERROR

COMMAND: RUN <line number>

ACTION: RUN initiates execution of the current program. If the optional line number is included, execution begins at that program line; otherwise, if no line number is specified, execution begins at the first line in the program.

EXAMPLES:
```
RUN
RUN 100
```

REMARKS: Any variables which were assigned values before RUN is used are cleared prior to starting the program. This means that all numeric variables are reset to 0; existing strings and arrays are destroyed, and will be initialized to spaces and zeroes, respectively, if and when created during the execution of the current program. Note that any variables set in direct mode before the RUN will also be cleared as a result of the RUN command.

ERROR MESSAGES:

NO PROGRAM ERROR
RUN was used before entering or LOADing a program.

LINE NUMBER ERROR
The optional line number included as part of the RUN command is not in the current program.

ARG ERROR
The optional argument is not a legal line number.

SEE ALSO: COMMAND: CONT
STATEMENT: CHAIN

DISCUSSION: CONTROL-C, THE PANIC BUTTON

Occasionally, you may desire to interrupt a program's execution at some random point while it is RUNning. This may be because you wish to repair a program error, or because you do not want program execution to continue to completion.

Your "PANIC BUTTON" is "control-C". This "stop everything" signal is sent to the computer whenever you hold down the "control" key then press the "C" key at the same time on your console terminal.

If a program is RUNning, the currently executing statement will finish, and the message

STOP IN LINE XXXXX

will be printed on the terminal, where XXXXX will actually be the line number where execution stopped.

If you are LISTing a program when control-C is pushed, the line being listed will be completed, and the message

STOP

will be sent to the console terminal.

Whenever you use control-C, you will be returned to BASIC's direct mode, where you are free to examine the program and variables.

Perhaps you may someday "PANIC"-out of a long-running program because you fear that it is caught in an "endless" loop. However, upon examination of the program and its variables, you discover that the program is operating correctly, but just takes a long time to finish. In this and similar instances, you may use the CONT command to resume execution at the point where the program was interrupted by control-C.

(You may not use CONT if, during the interruption, you modify any part of the program text.)

BASIC may be instructed to ignore the control-C signal. This is accomplished by changing certain internal data in the BASIC interpreter itself, a procedure described in DISCUSSION: PERSONALIZING BASIC. Because it involves modification to BASIC and also makes it impossible to stop an improperly-written "runaway" program without somehow stopping

the computer altogether, you should leave control-C enabled until your program is fully debugged.

SEE ALSO: COMMAND: CONT
STATEMENT: STOP
DISCUSSION: SOME BASIC CONCEPTS
DISCUSSION: PERSONALIZING BASIC

COMMAND: CONT

ACTION: CONT causes execution of a previously RUNning BASIC program to CONTinue after the execution of a STOP statement or after a control-C interruption. Normally, execution will continue at the program statement immediately following the last statement executed. (See REMARKS, below, for exceptions to this rule.)

EXAMPLE PROGRAM:

```
10 PRINT "THIS LINE PRINTED AFTER RUN"
20 STOP
30 PRINT "THIS LINE PRINTED AFTER CONT"
```

REMARKS: CONT may not be used if the previously running program has stopped because of an error or the execution of an END statement. Also, CONT may not be used if any modification has been made to any line of the current program since the interruption occurred.

It is possible to use direct statements during the interruption caused by STOP or control-C, for example, to examine or change variable values. After doing so, you may use CONT to CONTinue with the program.

If the stop was caused by control-C interruption during the execution of an INPUT statement, then execution will continue at the beginning of that INPUT statement.

ERROR MESSAGES: CONTINUE ERROR
This error occurs because of one of the following four reasons:

1) The program has stopped because it executed an END statement.

2) It has stopped because of a program error.

3) The program has been changed between the time it stopped and the time you typed CONT.

4) The current program has not yet been RUN.

SEE ALSO: DISCUSSION: CONTROL-C, THE PANIC BUTTON
STATEMENT: STOP

COMMAND: PSIZE

ACTION: The size of the current program in file blocks is printed on the console terminal.

EXAMPLE: PSIZE

REMARKS: The PSIZE command may be used to determine how many file blocks on diskette will be required to store the current program. This figure is helpful in creating new program files, and in using the NSAVE command.

The approximate number of bytes in the BASIC program may be calculated by multiplying the number obtained through PSIZE by 256 (the number of bytes in a file block).

ERROR MESSAGES: None.

SEE ALSO: COMMAND: NSAVE

COMMAND: MEMSET <memory address>

ACTION: The upper bound of the program/data memory region available to BASIC is changed to the specified address, which must be an integer constant in the range of 0 to 65535.

EXAMPLES:
```
MEMSET 24575  {last memory cell is 5FFFH}
MEMSET 32767  {last cell is 7FFFH}
MEMSET 40959  {last cell is 9FFFH}
```

REMARKS: Note that the address specified in a MEMSET command is expressed as a decimal (base 10) number. Addresses in microcomputer memory are commonly given in HEXADECIMAL (base 16) notation. If the desired upper memory bound is known only in hexadecimal, it will be necessary to convert the number into decimal before using MEMSET. (See APPENDIX 1: SAMPLE PROGRAMS for a routine which performs this conversion.)

All variables in the program/data area are cleared after MEMSET, but any current program remains intact.

MEMSET also modifies the copy of BASIC in RAM so that, if any copies of it are made, they will assume the new memory configuration when executed.

ERROR MESSAGES: ARG ERROR
The memory address specified as upper bound does not contain usable memory.

OUT OF BOUNDS ERROR

1) The address is larger than 65535.

2) If there is a current program, the specified upper bound would lead to a program/data area too small to hold it.

3) If there is no current program, the specified upper bound implies elimination of the program/data area altogether.

SEE ALSO: DISCUSSION: PERSONALIZING BASIC

STATEMENT: LINE <numeric expression>
LINE #<device expression>, <numeric expr.>

ACTION: The line length for the specified I/O device is changed to the value of the numeric expression, which must be an integer from 10 to 132. The device expression must be numeric, and evaluate to an integer from 0 to 7. If no device expression is specified, the desired device is assumed to be #0 (the console terminal).

EXAMPLES:
```
100 LINE 132
70 LINE L(X)+40
250 LINE #3,B
900 LINE #D(Q), 64
```

REMARKS: A fixed-length input/output line is a necessity because BASIC must keep track of the current PRINT position on the terminal or screen in order for the TAB function to work correctly. Use of the LINE statement allows the user or programmer to adjust this line length to the requirements of a particular terminal device. For example, many video-display boards provide for 32 or 64-character lines, while integrated terminals usually have 80 character-positions to a line. Printer units have line lengths ranging from 40 to 132 characters.

Different line lengths may be in effect for different terminals at any one time.

If a line of output information is longer than the current line length for the given device, the line will be "split" at the line length boundary and the rest of the output will be continued on the next line. (A carriage return is automatically generated by BASIC to advance the rest of the output to the next line.)

If an attempt is made to INPUT more characters than are allowed on one line, a "LENGTH ERROR" occurs.

LINE may be used as a direct statement.

Line lengths set by a LINE statement remain in effect until the session with BASIC is terminated. A line length of 132, for example, will remain in effect even after the program which set it has ENDed.

When BASIC "comes up", the initial length of device #0 (the console terminal) is 80 characters. The initial value for each of the seven other possible

system I/O devices is also 80. These initial values may be changed using procedures which are covered in DISCUSSION: PERSONALIZING BASIC.

ERROR MESSAGES: OUT OF BOUNDS ERROR
The device number or line length specified in the LINE statement is out of range.

SEE ALSO: DISCUSSION: FUNCTIONS (built-in: TAB)
STATEMENT: INPUT
STATEMENT: INPUT1
DISCUSSION: PERSONALIZING BASIC

COMMAND: BYE

ACTION: The current session with BASIC is terminated, and control returns to the DOS.

EXAMPLE: BYE

REMARKS: The BYE command does not affect BASIC's program/data area in any way -- the current program and any data associated with it remain intact. It is possible to return to BASIC and resume work with the current program later, provided that the memory containing BASIC and its program/data area is not disturbed in the meantime.

ERROR MESSAGES: None.

SEE ALSO: DISCUSSION: SPECIAL ENTRY POINTS

DISCUSSION: USING NUMBERS

This section describes numbers and how to use them in conjunction with the standard version of North Star BASIC. Those with non-standard versions of BASIC should read the section called DISCUSSION: NON-STANDARD VERSIONS OF BASIC which provides extra information applicable to their individual situations.

CONSTANTS

Numbers are represented within BASIC programs much as they are written in everyday usage. Here are some numbers as they might be written in a typical BASIC program:

Ø	347	-33.333	.ØØ176	1.ØØ3
.1	-8	123.4567	-.3	Ø.2

Numbers such as these are called NUMERIC CONSTANTS.

Constants may also be written in SCIENTIFIC NOTATION (also called EXPONENTIAL FORMAT or E-FORMAT). This is a way to represent very small or very large numbers without having to deal with leading or trailing zeroes which can make a number seem uncomfortably long. Here are the same numbers as in the examples above, but written in scientific notation:

ØE+ØØ	3.47E+Ø2	-3.3333E+Ø1	1.76E-Ø3	1.ØØ3E+ØØ
1E-Ø1	8E+ØØ	1.234567E+Ø2	-3E-Ø1	2E-Ø1

A number in scientific notation has a MANTISSA part and an EXPONENT part. These are separated by the letter E, which may be read as "times 10 to the power of". Thus, 1.76E-Ø3 would be read as "1.76 times 10 to the power of -3".

PRECISION

Numbers in the standard version of North Star BASIC are stored with 8-digit precision. Other precisions are available -- see DISCUSSION: NON-STANDARD VERSIONS OF BASIC for details. North Star BASIC uses the most accurate form of microcomputer arithmetic available: Binary-Coded-Decimal (BCD) -- see DISCUSSION: COMPATIBILITY WITH OTHER BASICS. All arithmetic operations are rounded to 8 digits in the standard version of North Star BASIC -- e.g., the sum of .12345678 and .Ø11111111 would be rounded to

.13456789, since .134567891 requires 9 digits.

EXAMPLE: FRACTIONS. What is the decimal representation of 2/3? An endless string of 6's after the decimal point is the only correct answer. However, when doing decimal arithmetic, both people and computers round off the long fraction to a reasonably accurate (but not completely accurate) number. BASIC, for example, will round 2/3 to .66666667. Notice that the total number of digits is now 8. It is impossible to get a more accurate representation of 2/3 in standard North Star BASIC. The fraction 1/2, on the other hand, needs only a single digit (.5) to represent it exactly!

EXAMPLE: MIXED DECIMAL FRACTIONS WITH LARGE WHOLE PARTS. Eight-digit precision also means that the number 1234.56789 must be rounded before it can be handled by the machine. North Star BASIC will round this to 1234.5679. Notice that the least-important, rightmost digit is rounded. This is BASIC's standard rounding procedure, and insures that the rounded number remains as close to the original value as possible.

Business users should note that the largest dollars-and-cents figure which may be exactly represented by 8 digits (without rounding cents to dimes or dollars) is $999,999.99 For applications where dollars-and-cents amounts larger than this must be handled, you should obtain a special version of BASIC (with greater precision).

EXAMPLE: A VERY LARGE NUMBER. The number 987654321 will be rounded to 987654320, and, henceforth will normally be PRINTed in scientific notation by BASIC as 9.8765432E+08. As you can see, the "eight-digit-rule" is followed in this conversion, even though scientific notation is invoked in order to correctly represent the number. The last (9th) digit is "dropped", but scientific notation representation insures that a 0 will be "remembered" for the ninth digit in order to maintain proper place values for the remaining digits. Notice that, because of this effect, BASIC considers 987654320, 987654321, and 987654322 to be equal to one-another because they differ only in their (ignored) ninth digits.

EXAMPLE: A VERY SMALL NUMBER The number .00000000123 will not be rounded by North Star BASIC, but .000000001234567 89 will be rounded. To see why, think of the two numbers as expressed in scientific

notation. The first becomes 1.23E-09. The mantissa (which is the only component of an E-format number that is affected by precision) is only 3 digits long -- well within the 8 allowed. The second number converts to 1.23456789E-09, with a 9-digit mantissa which is too many digits. The number will be rounded to 1.2345679E-09. (Note that scientific notation is a more compact way to write these very small numbers.) Finally, if you added 1 to either number, it would be rounded to become exactly 1. Check the E-format versions for the clear reason. This time, you'll come up with 1.00000000123E+00 and 1.00000000012345679E+00. Both mantissas exceed 8 digits in length. Rounding them to 8 digits leaves only the number 1 for each.

RANGE

A number may be positive, negative, or zero. Positive and negative numbers in standard (8-digit) precision North Star BASIC can range in magnitude from 1E-64 to 9.9999999E+62.

If you type a numeric constant into BASIC which is too large for BASIC to handle, a SYNTAX ERROR will occur. If a number which is too small is typed in, it will be rounded down to zero.

VARIABLES

In BASIC, as in most other programming languages, a NUMERIC VARIABLE is considered to be a place (in computer memory) where a numeric VALUE may be held. It is, in effect, a "storage place" which may be occupied by any one numeric value at any time. If a new number is put in a variable, that number totally replaces the previous value which the variable held.

All numeric variables are given initial values of zero until given different values in explicit LET statements.

Variables are given NAMES, and a variable name is used to refer to the variable and/or its contents when writing programs.

Numeric variable names in North Star BASIC consist of a single capital letter, or a single capital letter followed by a single digit from 0 to 9. Here are some legal North Star BASIC variable names:

A B7 C3 Z Q N8 P0

Because these variables may contain only one value, they are called "simple" variables.

OPERATORS

Operators are used in BASIC as they are in regular arithmetic -- to combine two numeric values (operands) or to modify one operand in certain predefined ways. Three classes of operators, arithmetic, relational, and boolean are used with numbers. Each class will be examined separately:

ARITHMETIC OPERATORS

These operators correspond to those used in common mathematic expressions:

OPERATOR	FUNCTION	EXAMPLE
↑ (or ^)	exponentiation	9↑2=81
*	multiplication	5*1.5=7.5
/	division	3/2=1.5
-	subtraction	3.2-2=1.2
+	addition	7.9+2.1=10
-	negation	-3, -27

RELATIONAL OPERATORS

The relational operators are used to compare pairs of numeric values. The numeric result of a relational comparison is either 1 (which stands for "true") or 0 ("false"). Usually, relational comparisons are employed as conditions for IF...THEN statements (See STATEMENT: IF). For example, at a certain point in a program, it might be desired to assign the value of 10 to the variable T if the value of X is greater than 10. The comparison (X>10) would be used as

```
IF X>10 THEN T=10
```

The IF statement will assign 10 to T based on the truth or falsehood of the relational comparison at the time the statement is executed. The following chart presents the relational operators available in North Star BASIC:

OPERATOR	RELATION	EXAMPLES
>	greater than	(6>1)=1 (true) (2>3)=0 (false)
<	less than	(0<0)=0 (false) (1<3)=1 (true)
<=	less than or equal to	(5<=5)=1 (3<=5)=1 (6<=5)=0
>=	greater than or equal to	(8>=7)=1 (7>=7)=1 (6>=7)=0
=	equal to	(9=9)=1 (9=7)=0
<>	not equal to	(4<>5)=1 (2<>2)=0

BOOLEAN OPERATORS

The boolean operators (AND, OR and NOT) may be used to combine or otherwise modify relational (true/false) expressions so as to provide for complex logical evaluation. Furthermore, any numeric values may be the objects of a boolean operation: all non-zero values will be treated as "true" (1), while 0 will be treated as "false". The result of a boolean operation is either "true" (1) or "false" (0). The table below summarizes the effects of the boolean operators. <A1> and <A2> stand for operands.

OPERATOR	EXPLANATION	EXAMPLES
<A1> AND <A2>	If both <A1> and <A2> are true (non-zero), the AND operation is "true" (1), else it is "false" (0).	(3>5 AND 2<3)=0 (3>2 AND 0<=0)=1 (2=3 AND 0>-1)=0
<A1> OR <A2>	If at least one argument is true, then the OR operation is true. If both are false, the OR is false.	(3>5 OR 2<3)=1 (3>2 OR 0<=0)=1 (2=3 OR 0<-1)=0

NOT <A1>	Negates the boolean value of the argument. If <A1> is non-zero (true), the NOT operation is false. If <A1>=0 then NOT <A1> is true.	NOT 7=0 NOT 0=1 NOT (3>5)=1 NOT (3<5)=0

EXPRESSIONS

Any valid combination of numeric constants, numeric variable names, operators, function calls, and array-element names is a NUMERIC EXPRESSION. (See DISCUSSION: FUNCTIONS and DISCUSSION: USING ARRAYS for complete details concerning "function calls" and "array-element names". These are two advanced features of North Star BASIC which are not covered in this introductory section.) A single constant, 3.14, or variable name, A, is an expression all by itself. In contrast, long constructs such as

```
(NOT(3+(SQRT(X*Y)/M3-47)/8)↑3
```

are also numeric expressions.

EXAMPLES OF LEGAL NUMERIC EXPRESSIONS

```
3.14
43+A
((X+2)↑(Q-R))*SQRT(Z)
```

EXAMPLES OF ILLEGAL NUMERIC EXPRESSIONS

```
438,000.33   (REASON: CONSTANTS CANNOT CONTAIN COMMAS)
7**Y  (REASON: TWO OPERATORS IN A ROW ARE NOT ALLOWED)
((3*ABS(A))+4  (REASON: IMPROPER PARENTHESES NESTING)
```

ORDER OF EVALUATION OF OPERATORS

Is 7+3*2 equal to 20 or 13? This depends on whether the addition or multiplication is performed first. For purposes of determining the order of evaluation of operators, each operator is said to have a certain PRECEDENCE. The rule for the order of evaluation is as follows: Higher precedence operators are evaluated first, and operators of equal precedence are evaluated left-to-right. OPERATORS ENCLOSED IN PARENTHESES ARE EVALUATED BEFORE OPERATORS NOT ENCLOSED IN PARENTHESES. When there are parentheses within other parentheses, operators within the innermost parentheses are evaluated first. The operators are listed below in order of decreasing

precedence -- that is, operators which are higher in the list have higher precedence than those toward the bottom of the list. Operators on the same line have equal precedence.

```
NOT, unary minus (-, negates a number)
↑  (exponentiation)
*,/  (multiplication and division)
+,-  (addition and subtraction)
=,<,>,<>,<=,>=  (relationals)
AND
OR
```

Thus, 7+3*2 is equal to 13, but (7+3)*2 is 20. Also, 3*8/2 is 12, -5+4 is -1 (the "-" is a unary minus here), and (1=2 OR 3=1) is 0.

DISCUSSION: USING ARRAYS

INDEXING AND SUBSCRIPTING

An ARRAY is an ordered collection of numeric variables. The entire array, as a whole, has a single variable name, and all the variables (called ELEMENTS) in the array share that name, much as the members of a typical family share the same surname. An individual element in an array is identified by its unique INDEX NUMBER, which denotes its position in the ordering of the array elements. For the convenience of both those who prefer counting from zero and those who prefer counting from one, an extra element, the "zero element", is included in each array. For example, a "50-element array", having a maximum index number of 50, actually has 51 elements, indexed 0, 1, 2, ... , 49, 50.

To represent a given array element in a numeric expression, you must follow the name of the array with a subscript -- the index number of the desired element enclosed in parentheses. For example, the zero-element of array A would be written as A(0), the eighth element as A(8), etc.

The index in a subscript may take the form of any numeric expression -- it need not merely be a constant. Therefore, if the simple variable I contains the value of 4, then A(I) will represent the same element as A(4). Care should be taken, however, to make sure that any expression used as an array index will not evaluate to a negative number or a number greater than the maximum index of the given array. If either of these things happens, an OUT OF BOUNDS ERROR will occur. If the index evaulates to a non-integer, BASIC will TRUNCATE the value to an integer. (Truncation involves throwing away the fractional part of a number and keeping only the whole part. The number 3.6 would be truncated to the whole number 3. Note that this is not the same as rounding.)

Note that the simple variable A and an array A may co-exist in the same program without in any way affecting each other. Arrays and simple variables with the same names are separate, distinct entities. BASIC does not confuse the two, since a simple variable name will never be followed by a subscript, while the name of an array must ALWAYS be followed by one.

MULTIPLE-DIMENSION ARRAYS

Arrays which require only one index may be thought of as single "rows" of variables. BASIC also permits the definition of arrays which use more than one index in their subscripts. The addition of each new index to an array is said to add another "dimension" to the array, and an array with n indices is called an "n-dimensional" array. When using more than one index to reference a single element, the indices must be separated by commas. Remember that each index is allowed to be a numeric expression.

To access the third element in the fifth row of a two-dimensional array M, for example, you write M(5,3). Assuming M has a maximum row number of X and a greatest column index of Y, the following statements will list the contents of each element in the array in an appropriate tabular format:

```
10 FOR I=0 TO X
20   FOR J=0 TO Y
25     REM Print next element w/no <CR>
30     PRINT TAB(I*15),M(I,J),
35     REM Each column of numbers
36     REM is 15 spaces wide.
40   NEXT
45   REM Print <CR> before starting next row.
50   PRINT
60 NEXT
```

Space for arrays is reserved by the programmer using the DIM statement. A DIM statement specifies how many dimensions an array will have, and what the maximum index will be in each dimension.

```
10 DIM X(1000), Y(2,3), Z(10,10,10)
```

The above defines an array X consisting of elements indexed from 0 to 1000 (1001 elements altogether), a two-dimensional array Y with maximum row index of 2 and maximum column index of 3, and a three-dimensional array Z with dimensions of 10, 10, and 10. In keeping with the "zero-element" convenience feature mentioned above, each array dimesnion includes a zero-element, so that array Z above actually contains 11 elements, instead of 10, in each dimension, indexed from 0 to 10.

When more than one dimension is specified, the maximum indices must be separated by commas. Commas must also separate array declarations when more than

one occurs in a single DIM statement.

The maximum index for any dimension in an array declaration may also be given in the form of a numeric expression. If the variable Q contains the value 10, then the following DIM statement will result in the creation of the same arrays as the previously given one:

```
10 DIM X(Q*Q*Q), Y(Q/5,3), Z(Q,10,SQRT(Q*Q))
```

An array may have any number of dimensions, but arrays with many dimensions tend to take up huge amounts of memory space. Consider that an array F, declared as F(10,10,10,10), will result in the reservation of 14,641 variable spaces in memory! (This corresponds to 11*11*11*11, not 10*10*10*10 -- remember the 0-element in each dimension!) Each element of the array takes up several bytes, and chances are this particular array would be too large to fit in the memory of your computer.

Whenever there is not enough memory available in the program/data area to hold an array, a MEMORY FULL ERROR occurs.

DEFAULT DIMENSIONS

All arrays of more than one dimension and most one-dimension arrays must be declared in DIM statements before being used. However, it is not necessary to declare a one-dimensional array of maximum index 10 or less. Any array which is used without first being declared in a DIM statement is automatically created by BASIC to be one-dimensional, and of maximum index 10. If you desire a specific maximum index greater or smaller than 10, however, you must use a DIM statement to create the array. An attempt to reference an element in a multi-dimensional array before the array has been dimensioned in a DIM statement will fail, causing an OUT OF BOUNDS ERROR. When dimensioned, an array is automaticaly initialized so that all of its elements contain the value 0.

ARRAYS MAY NOT BE RE-DIMENSIONED

No matter how created, either by an explicit declaration in a DIM statement or automatically, by BASIC, no array may be re-dimensioned in another DIM statement later during program execution. Specifically, this means that the size of arrays may

not grow or shrink during the RUN of a program. Any attempt to "re-dimension" an existing array will result in a DIMENSION ERROR.

ARRAY REFERENCES IN NUMERIC EXPRESSIONS

As mentioned in the chapter on USING NUMBERS, array elements may be used in numeric expression, since they are perfectly legal variable names. Here are some examples of array elements used in expressions:

```
10 X=SQRT(Q(3,5)+ABS(B))
60 PRINT M(F(A,B),L(A,B))
90 N(A)= N(A+1)/2
```

SEE ALSO: DISCUSSION: USING NUMBERS
STATEMENT: DIM

DISCUSSION: USING STRINGS

A STRING is a sequence of letters and/or other characters. For example, the following are strings:

```
HELLO     NG;34*    ABC123
THE DATE IS 7/7/78
```

STRING CONSTANTS

Strings enclosed in quotation marks are called STRING CONSTANTS. Note that the quotation marks themselves are not part of the string, but serve only to mark its boundaries for convenient recognition by both human beings and machines. The following are examples of BASIC string constants:

```
"HELLO"     "NG;34*"     "ABC123"
"THE DATE IS 7/7/78"
```

THE NULL STRING

The string represented by two consecutive quotes ("") contains no characters, and is called the NULL STRING.

STRING VARIABLES

Just as numbers may be held in numeric variables, so can strings be held in STRING VARIABLES. String variables are named similarly to numeric variables, and differ only in that a dollar-sign ($) is added to the name to denote the type of the variable as string. Thus, a legal string variable name consists of a single capital letter (A-Z) followed by a dollar-sign, or a capital letter and a single digit (Ø-9), followed by a dollar sign.

Examples of legal string variable names:

```
A$    Q7$    Z3$    R$
```

DIMENSIONING STRING VARIABLES

Before they can be used to hold string values in a program, string variables must be DIMENSIONED. DIMensioning a string causes BASIC to reserve memory space to hold the value of a string. To dimension a string, the string name must be included in a DIM statement, along with its MAXIMUM LENGTH in characters, before it is used to store a string value in a program. (For the proper method of doing this,

see STATEMENT: DIM.) If you use a string variable without having first declared it in a DIM statement, BASIC will automatically dimension it to a maximum length of 10 characters. Once created, strings may not be re-dimensioned in a program.

A string variable may contain any string whose length is less than or equal to the dimension of the string. The CURRENT LENGTH of the variable is the length, in characters, of the value it contains. Thus, if A$ is dimensioned to a maximum length of 26 characters, it may hold the entire alphabet (current length = 26 characters), the string "CAT" (current length = 3), or even the null string (current length = 0).

Immediately after being dimensioned, a string is initialized to contain all blanks. Thus, if A$ is dimensioned to be 26 characters long, it initially contains a string of 26 blanks.

SUBSTRINGS

The programmer can access parts of a string -- smaller segments consisting of one or more consecutive characters from within the string. Such a segment is called a SUBSTRING.

Substrings of string variables are represented by SUBSTRING NOTATION -- adding a SUBSTRING INTERVAL, in parentheses, to the variable name. For example, assume that A$ holds the string value "ABCDE". (Unless otherwise stated, this will be the permanent value of A$ throughout the discussion.) To represent its substring "CD", you would write A$(3,4), which specifies a substring consisting of the 3rd through the 4th characters of A$. A$(3,3) would yield the value of "C", and A$(2,5) would represent "BCDE".

Either or both of the numeric values in a substring interval may be represented by any numeric expression, as long as each expression evaluates to a value greater than or equal to 1 and less than or equal to the current number of characters in the string. Whenever any of the numeric values in a substring interval are non-integer, BASIC ignores the fractional parts. Thus, 5.6 is taken as 5, and 1.23 is taken as 1. If A=3 and B=4 then A$(A,B) would be the same as A$(3,4), or "CD". If B is more than 5, or A is less than 1, A$(A,B) would not be allowed, causing an OUT OF BOUNDS ERROR. This error will also occur if the value of the first expression is greater than the value of the second. Therefore, a backwards

substring such as A$(4,2) is illegal.

THE OPEN-ENDED SUBSTRING

A special form of substring notation is used to reference a substring consisting of all the characters from a given starting position in the string through its end. OPEN-ENDED SUBSTRING NOTATION uses only one numeric expression, which specifies the starting position within the string, and which must be greater than or equal to 1 and less than or equal to the length of the original string. For example, A$(3) stands for "CDE". Note that the value of A$ as a whole is the same as the value of the open-ended substring A$(1). A$(5) and A$(5,5) are the same as well, since the 5th character is the last character in A$. Use of open-ended substring notation eliminates the need, in certain situations, to know the current length of the original string.

STRING OPERATIONS: CONCATENATION

One operation may be performed on strings: CONCATENATION, symbolized by the "plus" operator (+). This is not to be confused with numeric addition. Instead, concatenation is the joining of two strings, front to back, rather like coupling railroad cars together. For example, "CAR"+"LOAD" represents the same value as "CARLOAD". Any string value may be concatenated with any other string value to yield a third value which consists of the two linked together. A$(2,3)+A$(2) yields the value "BCBCDE". (Remember that A$ has held "ABCDE" throughout this discussion.) Concatenation operations can be "chained", such as in

```
A$(1,1)+A$(3,3)+A$(3,3)+A$(5)+A$(4)+" MEANS YIELD"
```

which gives the value "ACCEDE MEANS YIELD".

STRING FUNCTIONS

BASIC includes certain built-in FUNCTIONS which return useful string values. It is also possible to define single-line and multiple-line user-functions which return string values. See DISCUSSION: FUNCTIONS for more detailed information.

STRING EXPRESSIONS

A STRING EXPRESSION is a string variable, substring, string function, or a quoted string literal. The

concatenation of two string values is also a string expression. Long, involved compound expressions may be formed by combining one or more of the elements mentioned above. For example:

```
A$
F$+",2"
A$(1,X)+CHR$(97)+A$+"GO FOR BROKE"+FNS$(25)
```

The built-in string functions (e.g. CHR$) and the user-defined string functions (e.g. FNS$) will be discussed later.

STRING COMPARISONS

String values may be compared using the comparison operators = , > , < , <= , >= , and <> . BASIC compares string values using the following rules:

1) Two string values are equal only if they have the same number of characters, and have matching characters in each character position.

2) Strings are compared character by character, left to right, until a difference occurs or one of the strings ends.

3) If a difference exists, and the ASCII value of the first different character in the first string is less than that of the corresponding character in the second string, then the first string is "less than" the second string. If the character in the first string is greater than its counterpart in the second string, then the first string is "greater than" the second.

4) If one of the strings ends before a difference is found, the shorter string is considered to be "less than" the larger one.

5) As a consequence of rule #4, the null string is always less than a non-null string.

When using strings composed solely of alphabetic characters of the same case (either upper or lower, but not both), this scheme corresponds to comparsion by "dictionary order", where an "entry" is considered to be "less than" another if it comes before the other in the dictionary, and "greater" than the other if it comes after. Thus "bird" is less than (comes before) "tree", and "zero" is greater than (comes after) "aardvark". The difference between string

comparisons in BASIC and regular word-comparsion by alphabetic order lies solely in the fact that the ASCII character set, used to define "alphabetic" order in BASIC, has 128 "letters" as opposed to our usual 26. To give you a better idea of this expanded "alphabetic order", here are some samples of string comparisons. Use the five rules above and the table of ASCII codes in APPENDIX 4: DECIMAL-HEX-BINARY-ASCII CONVERSION TABLE to check the following examples:

```
"Z" > "COCOA"        "120" < "75"
"123" < "124"        "AB " > "AB"
"123" < "ABC"        "AB1" > "AB01"
"ABC" < "abc"        "," > "!"
"ABC" > "AB"
```

NOTE: The logical operators AND, OR and NOT may not be used to combine the effects of two or more string comparisons in an IF statement. These three operators may be used in numeric comparisons only.

ASSIGNMENT TO STRINGS AND SUBSTRINGS

Any legal string expression may be assigned to a string variable or any part of a variable (by the use of substring notation), as in the following examples:

```
A$="CAT"
Q7$(1,3)="DOG"
```

(In the second example, note that the first three characters of Q7$ will become "DOG". Any characters in Q7$ past the third will not be changed.)

If a string value is assigned to a string variable which has been dimensioned to be too small to hold the entire value, its rightmost characters are discarded until the resulting truncated value will fit in the variable. Similarly, if an assigned value is too big to fit in a substring interval, it is truncated to the proper length. As an illustration, try RUNning the following program:

```
10 REM Demonstration of automatic
20 REM string truncation in assignment.
100 DIM L$(13)
110 L$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
120 PRINT L$
130 L$(2,3)="12345"
140 PRINT L$
```

The output of the program looks like this:

```
ABCDEFGHIJKLM
A12DEFGHIJKLM
```

The value shown on the first line of output is a result of the assignment statement in program line 110. Although the attempt was made to assign the entire alphabet to L$, only the first 13 characters fit, due to the dimension declared for L$ in line 100. The rest of the alphabet was discarded.

The second output line shows the value produced by the assignment in line 130. The assignment asks that a five character string value be squeezed into a two-character interval, which is not possible. As a result, BASIC assigned only the first two characters of "12345", or "12" to the substring, ignoring the rest.

When assigning to a substring interval, if the value assigned is smaller in length than the substring interval, any remaining characters in that interval are not modified, as in the following example program:

```
10 REM More substring assignment.
20 DIM L$(13)
30 L$="ABCDEFGHIJKLM"
40 PRINT L$
50 L$(5,9)="12345"
60 PRINT L$
70 L$(5,9)="abc"
80 PRINT L$
```

Here are the three output lines produced by the program:

```
ABCDEFGHIJKLM
ABCD12345JKLM
ABCDabc45JKLM
```

In the assignment of line 50, "12345" exactly fit the substring L$(5,9). However, in line 70, "abc" was two characters short, so only the first three characters of the substring, characters 5 through 7, were modified.

It is also possible to use the open-ended substring form to specify a substring interval into which a value is to be assigned. For example, L$(5) is taken to specify the same interval as L$(5,LEN(L$)).

(LEN(L$) stands for the current length of L$.) In the substring assignment example above, exactly the same results would have been obtained if the substring interval expressions in the string assignment statements had been replaced by open-ended substring expressions.

Assignment of the null string to any substring specified by regular or open-ended substring notation causes no change in the string.

MAXIMUM LENGTH VS. CURRENT LENGTH

The maximum length of a string variable is the maximum number of characters which it can hold. M$, dimensioned to 50, can hold up to 50 characters at once, but no more. On the other hand, a string's CURRENT length (as determined by the LEN function) is the number of characters which the variable actually does contain at any one time. Thus, if M$ contains "CAT", its current length is 3, despite the fact that its maximum length is 50. As long as M$="CAT", BASIC statements and string expressions may not access any character positions in M$ beyond the third. While M$="CAT", the character positions beyond the third simply do not exist, and a reference such as M$(3,5) is illegal. But, if M$ is changed to "STICK", then its current length becomes 5, and M$(3,5) is allowed. However, it is always incorrect to reference a character position beyond the maximum length of the string. In this example of M$, the substring reference M$(40,60) will always be illegal, since M$ can never grow larger than 50 characters in length, and therefore, the character positions from 51 to 60 will never exist.

CHARACTER SET IN BASIC

Up to now, "character" has been used in its intuitive sense, as a digit, letter or punctuation character which may be typed in by a user or printed on a terminal. In fact, the BASIC character set includes "invisible" control characters and the many "undefined characters" which may be represented as byte (8-bit) values. Altogether, BASIC's character set includes 256 values. The first 128 of them (0 to 127) correspond to the 128 characters of the international ASCII standard. The remaining 128 characters (128 to 255) are generally undefined on most terminals, but are available to the North Star BASIC programmer as a convenience. The built-in string function CHR$ may be used to represent any

character which cannot be typed or printed. Note that CHR$(34) may be used to represent a quote-mark.

```
10 A$="HI THERE"
20 PRINT A$
30 A$=CHR$(34)+A$+CHR$(34)
40 REM Above puts quote-marks in A$
50 PRINT A$
```

When RUN, the above program produces these results:

```
HI THERE
"HI THERE"
```

SEE ALSO: DISCUSSION: FUNCTIONS
DISCUSSION: USING NUMBERS (EXPRESSIONS)
STATEMENT: DIM
APPENDIX 4: DECIMAL-HEX-BINARY-ASCII CONVERSION TABLE

STATEMENT: DIM <list of array or string size declarations>

ACTION: Reserves program/data area memory space for strings and arrays as specified in the declarations.

EXAMPLES:
```
10 DIM A$(30),Q(100),Z(5,2)
60 DIM X7(X,Y), X8(X,X,X)
70 DIM C$(100*3)
```

REMARKS: A DIM statement automatically initializes the variables declared in it. After a DIM statement is executed, the length of any string declared in it is equal to the declared size and all character positions are filled with spaces. (For example, after executing line 10 above, A$ will be a 30-character string filled with spaces.) All elements of any array declared in a DIM statement will be initialized to zero.

When declaring strings, the single numeric expression enclosed in parentheses specifies the maximum number of characters which the string variable may hold. A declaration for a single array may contain several numeric expressions within the parentheses, each denoting the maximum index value in each "dimension" of the array. Thus, after execution of the DIM statements in lines 10 and 60 above, Q will be a one-dimensional array with a maximum index of 100, Z will be a two-dimensional array with 5 rows and 2 columns, and X8 will be a three-dimensional array with a maximum index of X in any of its three dimensions.

If a string or array is referenced in any statement without having been declared in a prior DIM statement, it is automatically created, initialized, and dimensioned by BASIC, strings to a maximum length of 10, and arrays to one dimension and maximum index of 10.

Whether "dimensioned" explicitly through a DIM statement or implicitly through first reference to a previously non-existent variable, a string or array may not be "re-dimensioned" (declared in a DIM statement executed later in time during the same RUN of a program). Any attempt to do so will lead to a DIMENSION ERROR. (For the same reason, a DIM statement itself may not be repeated during the execution of a program.)

ERROR MESSAGES: MEMORY FULL ERROR
Not enough program/data area memory is available to

hold one or more of the variables declared in the DIM statement responsible for the error. See APPENDIX 3: IMPLEMENTATION NOTES for details of memory allocation.

DIMENSION ERROR
An attempt was made to re-dimension a string or an array which already exists.

SEE ALSO: DISCUSSION: USING STRINGS
DISUCSSION: USING ARRAYS

STATEMENT: REM <optional line of any text>

ACTION: None. REM statements are ignored by BASIC.

EXAMPLES:
```
10 REM THE REM STATEMENT IS USED TO
20 REM INSERT COMMENTS IN A PROGRAM.
30 REM FOR EXAMPLE --
35 REM
40 N=G-W \ REM NET GETS GROSS LESS WITHHOLDING
45 REM
70 REM Lower case letters are ok in REMs.
```

REMARKS: As can be seen from example line 40, a REM may be included on the same line as other BASIC statements, however, it must always be the last statement on a line. The reason for this is, all text after the REM reserved word on a line is treated as a comment and is ignored by BASIC. Therefore, any statements which appear after a REM on the same line will not be executed.

As with other North Star BASIC statements, the characters ":", ";", "[", and "]" are translated to "\", ",", "(", and ")", respectively, within REM text.

ERROR MESSAGES: None.

STATEMENT: LET <numeric variable> = <numeric expression>
LET <string/substring variable> = <string expression>
<numeric variable> = <numeric expression>
<string/substring variable> = <string expression>

ACTION: The value of the expression on the right hand side of the equal-sign is assigned to the variable named on the left side. The reserved-word LET is optional, and may be omitted.

EXAMPLES:
```
10 X=X+1
50 LET A(X)=6
35 LET Q=SQRT(X)+Y
20 B$="HELLO THERE"
61 M$(2,11)=FNN$("415-549-0858")
150 LET Z$=STR$(Q)+Z$(1,2)+"BOX"
```

REMARKS: BASIC permits only one assignment per LET statement. However, several assignments may be made on one line, as in:

```
10 A=0 \ B=0 \ C=0
```

Note, in line 10 above, the apparent mathematical impossibility of X=X+1. However, as an assignment, this makes sense -- the right-hand expression is evaluated with the current value of X, and the result obtained then becomes X's new current value. X=X+N has the effect of increasing the value of X by N. (It is sometimes easier to understand assignment if one resists reading LET statements as "Q equals Q+1", for example, and says instead, "Q gets Q+1", or "Z becomes M+173".)

Only single variable names are legal on the left side of an assignment (LET) statement. Also, it is impossible to assign entire arrays with a single LET statement. Each individual element of an array must be assigned separately.

ERROR MESSAGES: TYPE ERROR
The type of the expression on the right side is not the same as the type of the variable on the left side. It is illegal to assign a string value to a numeric variable, or a numeric value to a string variable.

SEE ALSO: DISCUSSION: USING NUMBERS
DISCUSSION: USING STRINGS
DISCUSSION: USING ARRAYS

STATEMENT: PRINT
PRINT <list of string and/or numeric expressions>
PRINT "<device expression>
PRINT #<dev. exp.>, <string/numeric expr. list>

ACTION: The data indicated in the OUTPUT DATA LIST is printed on the specified output device. After the entire list is printed, the print-head or cursor of the terminal is moved to the start of the next line. If there is no output list, only a blank line is printed. If no device is specified, output is printed on device #0, the console terminal. The device expression consists of a numeric expression which evaluates to an integer from 0 to 7, corresponding to a connected output device. A piece of "data information" in the output list consists of any string or numeric expression. PRINT formatting expressions may also be included in the output list. See DISCUSSION: FORMATTED PRINTING for complete details. Elements in the output data list must be separated by commas. Elements in the same list will be printed on the same output line. Information which cannot fit on one output line will be continued on the next.

If a comma follows the output list, the print-head or cursor will not be moved to the next line, so subsequent output will appear on the same line.

EXAMPLES:
```
PRINT
PRINT "THE ANSWER IS: ",
PRINT A,B,C,A7
PRINT #D
PRINT #Q,A,B,"HELLO",C(3),Q$
```

Here is a sample program, designed to demonstrate the action of the PRINT statement as described above. Try it:

```
10 A=3
20 B=4
30 PRINT "A EQUALS",A,
40 PRINT "   B EQUALS",B
50 END
```

When this program is RUN, the following should appear on your terminal:

```
A EQUALS 3    B EQUALS 4
```

REMARKS: The exclamation point (!) may be used as an abbreviation for the keyword PRINT. Thus, the

statement

```
PRINT "STRING"
```

is the same as

```
!"STRING"
```

This is especially convenient when using the PRINT statement in direct mode.

Note that the comma (as separator in the PRINT output list) performs the same function as the semi-colon in many other versions of BASIC. To obtain output "tabbing", use the TAB function, as described in DISCUSSION: FUNCTIONS (built-in, TAB).

SEE ALSO: DISCUSSION: FORMATTED PRINTING
DISCUSSION: MULTIPLE I/O DEVICES
STATEMENT: LINE

DISCUSSION: FORMATTED PRINTING

NOTE: Read DISCUSSION: USING NUMBERS and STATEMENT: PRINT before beginning this section!

REGULAR AND E-FORMAT NUMBER PRINTING

Normally, BASIC will "choose" between "regular" form and exponential/scientific form for the most appropriate method to PRINT a numeric value. BASIC chooses the methods which will result in the most concise printed figure. Note that a space before each "regular" number is automatically printed.

```
 3.1415
 .7319
-8.Ø3
-.Ø4
```

When a numeric value is too large or too small to PRINT in regular form, BASIC will automatically use E-FORMAT. E-format consists of a space, a minus sign if the number is negative, the first digit of the mantissa, a decimal point (if there are any digits left in the mantissa), any other mantissa digits, an "E" (to denote the beginning of the exponent), a plus or minus sign to denote the sign of the exponent, and tne two digits of the exponent itself (-- the first digit may be Ø). Here are some numbers in E-format:

```
 1.4Ø73749E+14
-2E-Ø9
-5.4128376E+13
```

When BASIC chooses the format of printed values, the PRINT statement is in "FREE FORMAT" -- i.e., BASIC is free to PRINT tne values using the most concise format. Sometimes, however you may want certain values to be printed only in E-format, or only with two decimal places, or only as integers (with no decimal points). In other words, you may want to determine the format under which these numbers will be printed, as opposed to letting the computer choose. To do this, BASIC permits you to include numeric "format specifications" within the output lists of PRINT statements. These format specifications always begin with a per-cent sign (%).

WHAT IS A FORMATTED NUMBER?

A programmer-formatted (as opposed to a free-formatted) number always takes up exactly a given

number of spaces on the printed line. This is called the FIELD WIDTH. The field width is defined by the programmer in the format specification, and must reserve enough character positions in the printed line to hold all the characters in the number as printed. A field width of 6, for example, is too small to accommodate the number 1234.56, because 7 character positions are actually required -- six for the digits, and one for the decimal point! Also remember to leave room for plus or minus signs if they might occur in the number, as well as the letter "E", if E-format is being used to display a number in scientific notation. If the specified field isn't wide enough to PRINT a given number, then a FORMAT ERROR will occur when an attempt is made to PRINT the number using that format.

The next few examples will make use of "I-FORMAT" to illustrate some general points about BASIC's formatting mechanism. Only numbers with integer values may be printed using I-format. The I-format specification consists of the per-cent sign (%), a number, and the capital letter "I", as in the following:

```
%3I
```

The number given specifies the number of column positions on the printed line which will be reserved to hold the number. The %3I format specification, for example, requires that any number printed according to it must be an integer, and must fit in three character positions. Therefore, 0, positive numbers from 1 to 999, and negative numbers from -1 to -99 may be printed under this format. Remember that the negative-sign counts as taking a character position.

When printing a programmer formatted number, BASIC does not automatically insert leading spaces to keep the number from "bumping up against" previously printed information on the same line, as it does in free-format. The statement

```
PRINT "OOPS",%3I,349
```

results in

```
OOPS349
```

on the terminal. In order to separate your formatted output from other output, you may elect to PRINT

explicit spaces before (and after) the number, use the TAB function, or specify a field width large enough to provide at least one blank space between the number and previous information on the line.

RIGHT JUSTIFICATION

All programmer formatted numbers are automatically right justified within their PRINT fields. That is, the number is printed so that, in a field which is n character positions wide, the last character in the printed number occurs in the n-th (rightmost) character position of the field, and spaces fill to the left. The following numbers are right justified:

```
  349
 1234
  7.3
 8.42
-2118.37
 1.61
```

Note that, when right justified numbers having the same number of digits after the decimal point are printed one above the other, the decimal points will "line up". (Note that decimal-point numbers cannot be printed using I-format, but are included in this example because BASIC's decimal-point format, to be discussed soon, also right justifies.)

The statement

```
PRINT "HERE IS A GAP:",%10I,2
```

produces the output

```
HERE IS A GAP:         2
```

because the field, specified as 10 positions in width, is more than large enough for the 1-digit number 2.

DECIMAL PLACES

In the case of floating-point and E-format numbers, you may also decide how many decimal places are to be displayed when a formatted number is printed. For example, the floating point format %7F2 will put numbers from -999.99 to 9999.99 in "dollars-and-cents" form, with only two digits to the right of the decimal point:

```
-302.63
  51.00
 987.12
1234.56
```

(The field is 7 positions wide.)

Note that, if the number is an integer, zeroes are used to fill the decimal positions. Suppression of those "trailing zeroes" will be discussed later.

If a number to be printed has more decimal places than the format specification indicates, the value printed is the number rounded to the indicated number of digits.

Here are the allowable formats:

(in the following, n and m stand for integer constants)

model name/effect

nFm F-format:
Each subsequent numeric value in the PRINT list will be printed in an n-character field, right justified, with m digits to the right of the decimal point.

nI I-format:
Each subsequent numeric value in the PRINT list will be printed in an n-character field, right justified, provided they are integers (have no fractional part). If a value to be printed under this format is non-integer, a FORMAT ERROR will occur.

nEm E-format:
Subsequent numeric values in the PRINT list will be printed in scientific notation in an n-character field, right justified, with m digits to the right of the mantissa decimal point.

A format specification which consists only of a percent-sign specifies a return to free format.

All numeric values in a PRINT-list are printed using the new format specification until a subsequent format specification appears in the list, or until the end of the data/format list itself. Note that the printing of numbers in subsequent PRINT

statements will not usually be affected by format specifications in previously-executed PRINTs. In particular, for the two lines:

```
10 PRINT %3I,A,B,C
20 PRINT D
```

All values in line 10 will be printed according to the %3I format, but D (in line 20) will be printed using free format. The format specification in line 10 can affect only values which line 10 itself prints.

DEFAULT FORMAT VS. CURRENT FORMAT

BASIC keeps track of two format specifications: the CURRENT FORMAT and the DEFAULT FORMAT. Each numeric value in a PRINT output list is printed using the current format. At the beginning of each PRINT statement, the value of the current format is made equivalent to that of the default format. Thereafter, the current format is changed each time a format specification occurs in the PRINT output list. The default format is set initially to free-format, and may be changed by using the cross-hatch (#) format character in a format specification as described below.

OTHER FORMAT CHARACTERS

Certain other FORMAT CHARACTERS may be used to modify the effects of a format-specification. Several of these characters may be combined in one format specification, if you wish. All format CHARACTERS in a format specification must come after the % and before the format specification itself. Here are the characters:

Z Trailing zeroes after the decimal point are suppressed; spaces will be printed instead.

The format specification after this character will become the default format. Also, number-to-string conversion is done using the default format (see DISCUSSION: FUNCTIONS, built-in, STR$). Note that %# will force free format to be the default format. This is useful in cases where you have made another format the default, and would like to return to free-format.

C Commas will be placed to the left of the decimal point as needed to group each sequence of three

digits -- e.g. 1,234,567. (Note that the "C" option is not effective with E-format specifications.)

$ A dollar sign will be placed to the left of the value when it is printed.

Caution! When using C or $ with a format specification, you must be sure that the field width specifies enough character positions to contain the longest number you intend to PRINT in that format, plus any dollar sign, plus any maximum amount of commas which may be inserted by the machine. For instance, the statement

```
PRINT %C9F2, D
```

will yield the output

```
$3,478.92
```

when D=3478.92, but will result in a FORMAT ERROR if D=107843. The number should be printed as $107,843.00, but this requires the field width to be at least 11.

EXAMPLES:

FORMAT	VALUE	OUTPUT
%8F2	19.355	19.36
%$6F2	45.12	$45.12
%C9I	1000000	1,000,000
%C8I	1000000	FORMAT ERROR
%10E3	472	+4.720E+02
%Z10E3	472	+4.72E+02
%$C11F2	201758.88	$201,758.88

STATEMENT: INPUT <list of variables>
INPUT <string constant>, <var. list>
INPUT #<device expression>, <var. list>
INPUT #<dev. expr.>, <string const.>, <var. list>

ACTION: User input of string or numeric constant data is "requested" and accepted from the terminal named by the device expression. If there is no device expression, the console, device #0, is assumed. The device expression must be a numeric expression which evaluates to an integer from 0 to 7. The data provided by the user is assigned to the variables named in the INPUT statement's variable list. If no string constant is specified, input is "prompted" by a question-mark (sent to the terminal before input-data is accepted). If a string constant is given, however, this string is sent to the terminal as prompt, instead. The user strikes the RETURN key when finished providing data-input.

EXAMPLES:
```
10 INPUT A,B,Q$
70 INPUT "YOUR NAME: ",N$
35 INPUT #3,X,Y
30 INPUT #X,"COMMAND: ",C$(5,9)
19 INPUT "",X \ REM No prompt is given at all.
```

REMARKS: INPUT may not be used in direct mode.

INPUT will "wait" forever for user-response, until the RETURN key is struck.

String constants entered by the user in response to INPUT should not be quoted. (If quotes are typed, they will become part of the string.)

If an INPUT statement requires several consecutive numeric data-items to be given by the user, it is possible to put them all on one line, as long as they are separated from one-another by commas. For example, a proper response to an INPUT statement which asks for three numbers is:

123, 456, 789 <CR>

However, since carriage-returns must terminate the INPUT of a string, the "comma-method" is not suitable for inputting several consecutive strings. To INPUT more than one string value on one line of the terminal, successive INPUTl statements must be used. (See STATEMENT: INPUTl.)

To illustrate proper user-response to an INPUT statement, assume that example line 10 is executed. A question-mark (?) will appear on the terminal.

```
?
```

This indicates that the computer is waiting for INPUT, and the knowledgable user might type in the following:

```
2, 3, WEASEL<CR>
```

(<CR>, of course, signifies striking the RETURN key.) After RETURN is struck, A will be set to 2, B to 3, and Q$ to the string value "WEASEL".

A single carriage-return (representing no input) is acceptable when the next item in the variable list is a string. In this case, the string will be set null. However, valid numeric input must be supplied for numeric items in a variable list -- an INPUT ERROR will occur if this isn't done.

Note that the line editor may be used to modify the user's input line before <CR> is struck.

When too few data items are typed before RETURN is struck, BASIC will type a double-question-mark (??) as auxilliary prompt, and await further INPUT for the given variable list. It will repeat this step as long as necessary until all variables named in the variable list have been assigned values typed in from the terminal.

Note that the INPUT statements and the built-in INP function are not the same.

ERROR MESSAGES:

LENGTH ERROR
The line of data-input is too long.

INPUT ERROR -- PLEASE RETYPE
A numeric value was required by the INPUT statement, but a non-numeric value was supplied by the user. The user is automatically given a chance to rectify the mistake by retyping all data elements required by the INPUT statement.

SEE ALSO:

DISCUSSION: USING NUMBERS
DISCUSSION: CONTROL-C, THE PANIC BUTTON
DISCUSSION: FUNCTIONS (built-in, INP)
STATEMENT: INPUT1

STATEMENT:
```
INPUTl <list of variables>
INPUTl <string constant>, <var. list>
INPUTl #<device expression>, <var. list>
INPUTl #<dev. expr.>, <string const.>, <var. list>
```

ACTION: Exactly the same as STATEMENT: INPUT, except that when the user strikes the RETURN key to terminate an input line, no carriage-return is echoed to the terminal. Subsequent input or output will occur on the same line.

EXAMPLES:
```
50 INPUTl Z,W,B7,A(3)
25 INPUTl #D(Q), "GUESS? ",G
```

REMARKS: See STATEMENT: INPUT

ERROR MESSAGES: See STATEMENT: INPUT

DISCUSSION: MULTIPLE I/O DEVICES

A computer system may include several input/output (I/O) devices, such as a video terminal, printer, graphics display, etc. North Star BASIC provides a convenient means for BASIC programs to make use of up to 8 separate I/O devices. A unique integer numoer from 0 to 7 is assigned to each one. Device #0 must correspond to your main communication link to your computer -- also known as the console terminal. It is generally a teletype-style or a CRT (video) terminal. When your copy of DOS has been personalized to handle multiple I/O devices, your BASIC programs will be able to access the many I/O devices through the PRINT and INPUT statements. (See DOS section of this manual, chapter on "PERSONALIZATION" for details.)

A PRINT, INPUT, INPUTl or LINE statement accommodates an optional DEVICE EXPRESSION, which consists of a cross-hatch (#), followed by a numeric expression which evaluates to an integer number from 0 to 7. This expression indicates the device desired for input or output. If used in any of these statements, the device expression must be the first thing after the statement's keyword. Here are some examples:

```
PRINT #1, "TEST"
PRINT #Q,X,B,7
PRINT #D+3, "CRAZY",Q
PRINT #D7(X)

INPUT #B, L3
INPUT #7, "COMMAND: ",C$

LINE #1, 132
LINE #D, L
```

If the device expression is omitted, it is assumed to be 0 (the console).

As a final example, assume that device #0 is the console terminal, device 1 is a remote printer, and device 2 is a remote CRT. The following program causes a different message to be printed on each of the three devices:

```
10 REM Multiple I/O demonstration.
20 PRINT "THIS MESSAGE GOES TO THE CONSOLE."
30 PRINT #0,"THIS ONE DOES, TOO."
40 PRINT #1,"THIS WILL GO TO THE REMOTE PRINTER"
50 PRINT #2,"THIS SHOWS UP ON THE REMOTE CRT"
```

The PRINT/INPUT device expression, characterized by a cross-hatch, should not be confused with the PRINT statement's format specification, which begins with a per-cent sign (%).

SEE ALSO: STATEMENT: PRINT
STATEMENT: INPUT
STATEMENT: INPUT1
DOS section of this manual, chapter on PERSONALIZATION

STORING DATA WITHIN THE PROGRAM TEXT

STATEMENT: DATA <list of constants>

ACTION: The string and numeric constant values included in the list are stored as data and may be accessed, in order, by the BASIC program of which they are a part. If a list contains more than one constant, each constant must be separated from the next by a comma.

EXAMPLES:
```
1000 DATA "STRING DATA", "NUMBER IS NEXT",2
20 DATA 15
115 DATA 2, 7, 25, "HI", 0
```

REMARKS: The DATA statement provides a way to store information within the text of a BASIC program. This data may be accessed by a RUNning program when a READ statement is executed.

DATA statements may be placed anywhere in the program, and are ignored by BASIC except when an attempt is made to access the information they contain. In other words, DATA statements are non-executable.

ERROR MESSAGES: SYNTAX ERROR
An improperly-formed constant was placed in a DATA statement (i.e, a string without the opening or closing quote mark) and this results in a SYNTAX ERROR when a READ statement attempts to access this constant.

SEE ALSO: STATEMENT: READ
STATEMENT: RESTORE

STATEMENT: READ <list of variables>

ACTION: For each variable in the variable list, the next sequentially-available DATA element from the program's DATA statements is assigned to that variable.

EXAMPLE PROGRAM:

```
5 REM Example of READ
10 READ A,B
20 READ C(3),Q$
30 PRINT A,B, C(3), Q$
40 READ X
50 PRINT X
60 DATA 1,2,3," HI",4
```

Running this program yields the output:

```
1  2  3 HI
4
```

REMARKS: The variable and the corresponding constant in a DATA statement must be of the same type (i.e., a numeric constant may only be READ into a numeric variable, and a string-constant into a string variable).

A special internal "pointer" allows BASIC to keep track of the "current" data element. When a program is RUN, this pointer is initially set to to the first data element in the program's first DATA statement, or to "END OF DATA" if there are no DATA statements in the program.

When a data value is READ into a variable, the data pointer moves to the next element in the DATA statement. If there is no more data in the statement, the pointer is moved to the first element in the next DATA statement which occurs in the program. This process continues until there are no more DATA statements, at which time the pointer is set to "END OF DATA". After this happens, should a READ be attempted, it will result in a program error. Unless a RESTORE statement is executed, each data item may be READ once and only once, in the order in which it appears in the program text.

ERROR MESSAGES: READ ERROR
Either an attempt was made to read data once the "END OF DATA" condition occurred (without the execution of an intervening RESTORE), or the value was not of the

same type as the variable to which it was to be assigned.

SEE ALSO: STATEMENT: DATA
STATEMENT: RESTORE

STATEMENT: RESTORE
RESTORE <line number>

ACTION: The "pointer" to the next data item to be READ is moved to the first item in the first DATA statement in the program text. If a line number is specified, the pointer is moved to the first data item in the DATA statement at (or the first DATA statement occurring after) the given line.

EXAMPLE PROGRAM:

```
5 REM Example of RESTORE
10 READ A \ PRINT A
20 RESTORE
30 READ A \ PRINT A
40 RESTORE 70
50 READ A \ PRINT A
60 DATA 1,2,3,4
70 DATA 5,6,7,8
```

Running the above program produces the output:

```
1
1
5
```

REMARKS: RESTORE provides a means by which the same information in DATA statements may be READ more than once by a program. RESTORE makes it possible to "recycle" data (as shown in lines 10 to 30 in the example program), or "skip around" the data (as in lines 40 and 50).

The RUN command causes an automatic RESTORE (to the first DATA statement).

ERROR MESSAGES: Same as STATEMENT: GOTO

SEE ALSO: STATEMENT: READ
STATEMENT: DATA

DISUCSSION: EXECUTION AND CONTROL FLOW

The action specified by each statement in a BASIC program is performed when that statement is "executed". In BASIC, statements are usually executed in a sequential fashion, one after the other. BASIC scans a program and executes its statements as you would read the program listing: from lines with lower numbers to lines with greater numbers, and, if there is more than one statement on a line, from the leftmost statement to the rightmost statement on that line.

The order of statement execution (also called CONTROL FLOW) may be altered through the use of several special BASIC statements: GOTO, IF...THEN, FOR, NEXT, EXIT, GOSUB, RETURN, and ON...GOTO. Each of these CONTROL STATEMENTs is described in greater detail in its own section of this manual.

A control statement forces BASIC to treat the line number it specifies or the program location it implies as the location of the next statement to execute. Unless another control statement is encountered, BASIC will return to sequential execution at the new location.

In BASIC programs, the natural flow of control is often diverted, in order to achieve savings in program execution time and storage requirements. For example, repetition of program lines, a powerful space-saver, may be accomplished by using IF...THEN and GOTO statements. A common repetitive "looping" technique uses the statements FOR and NEXT (and, occasionally, the EXIT statement as well). Often, the program must make a choice on which of several alternative instruction blocks is to be executed next, based on a given condition. IF...THEN statements are used to evaluate the conditions and route control to the appropriate parts of the program. In certain situations, the ON...GOTO statement may be used in this capacity. Finally, GOSUB and RETURN are used to implement subroutines, which allow a programmer to substitute single GOSUB statements for entire large program segments, provided the segments (subroutines) are defined elsewhere in the program text.

STATEMENT: GOTO <line number>

ACTION: A GOTO statement causes an immediate "jump" to the specified line, instead of proceeding with the normal sequence of statement execution. Regular sequential execution resumes at the specified line.

EXAMPLE PROGRAM:

```
10 PRINT "THIS PRINTS FIRST"
20 GOTO 40
30 PRINT "THIS NEVER PRINTS"
35 PRINT "THIS PRINTS THIRD"
37 END
40 PRINT "THIS PRINTS SECOND"
50 GOTO 35
```

REMARKS: There may be no blank between GO and TO. GOTO is a single BASIC keyword.

Note that a <line number> must be a numeric integer constant. It may not be a variable or complex expression.

ERROR MESSAGES:

LINE NUMBER ERROR
The specified line does not exist within the BASIC program.

OUT OF BOUNDS ERROR
The line number specified in the GOTO statement is larger than 65535. (NOTE: This error occurs as soon as the erroneous line is typed!)

SEE ALSO: DISCUSSION: EXECUTION AND CONTROL FLOW
STATEMENT: EXIT
STATEMENT: ON ... GOTO

STATEMENT: IF <logical expression> THEN <statement>
IF <log. expr.> THEN <statement> ELSE <statement>

ACTION: When the logical expression is true, the statement after the word THEN is executed. When the condition is false, the statement after ELSE (if it is used) is executed. If no ELSE is specified, and the condition is false, the IF statement is ignored and execution continues with the next statement in sequential order. A single line number may be placed after THEN or ELSE, and is equivalent to (and shorthand for) a GOTO statement referencing that line number.

EXAMPLES:
```
10 IF X=5 THEN 1000
100 IF A$="CLYDE" THEN PRINT "HI" ELSE PRINT "BAD PW"
75 IF Q(7)<>3 AND W THEN GOSUB 110 ELSE LET X=15
230 IF A$="HI" THEN IF B$="THERE" THEN PRINT "YES? ",
999 IF Z THEN END
```

REMARKS: Only the THEN or ELSE part of an IF statement (never both) will be executed for each time the IF statement itself is executed.

The statement after THEN or ELSE may itself be an IF statement. Such multiple IFs are said to be NESTED. There is, of course, a rather small practical limit as to how deeply IFs may be nested, since the whole statement must fit on one line.

ERROR MESSAGES: IF statements do not usually cause error messages in and of themselves. Errors which occur during the execution of an IF statement may usually be attributed to the type of statement used in either its THEN or ELSE clause, or the mis-formation of the logical expression. Check the section on the appropriate type of statement or feature to track down the cause of each individual error.

SEE ALSO: DISCUSSION: USING NUMBERS (relational and boolean operators)
STATEMENT: GOTO

STATEMENT: ON <numeric expression> GOTO <list of line numbers>

ACTION: The numeric expression is used to choose a single number from the list of line numbers. Then, as with GOTO, execution is immediately transferred to the line with the chosen number.

EXAMPLES:
```
10 ON C GOTO 100, 200, 300, 400
105 ON X-10 GOTO 10, 20, 30, 40, 50, 60, 70
```

REMARKS: The numeric expression must evaluate to a quantity greater or equal to 1. There may be as many line numbers in an ON...GOTO statement as will fit on a program line.

The first line number in the list will be chosen if the expression evaluates to 1, the second if it reduces to 2, the twentieth if it equals 20, and so on. For example, in statement 10 above, if the value of C is 3, then the result will be the same as GOTO 300. An ON...GOTO statement with N line numbers in its list will work for integer values from 1 to N.

ERROR MESSAGES:

SYNTAX ERROR
This can happen with ON...GOTO because the numeric expression, when truncated, evaluated to an integer less than 1 or greater than the number of line numbers in the list.

TYPE ERROR
The expression specified was not a numeric expression.

LINE NUMBER ERROR
See STATEMENT: GOTO

OUT OF BOUNDS ERROR
See STATEMENT: GOTO

SEE ALSO: STATEMENT: GOTO

STATEMENT: STOP

ACTION: This statement causes program execution to stop. A message is sent to the console terminal, indicating the point in the program where the stop occurs.

EXAMPLE: 20 STOP

REMARKS: STOP is generally used during program development to provide temporary breakpoints at known spots during the execution of the program. Execution of a STOP returns the computer to DIRECT MODE, at which time LET and PRINT may be used as direct statements in order to change and examine, respectively, the values of variables within the program.

If CONT is used to resume program execution after STOP, any variables modified in direct mode during the interruption will retain the new values as the program resumes.

Program text may also be listed during the breakpoint provided by STOP, but, if you intend to continue with the program using the CONT command, you must be careful not to change any of the program text (edit, insert, or delete program lines) during the interim. If you do, CONT will not work, and you will be forced to RUN the program all over again.

ERROR MESSAGES: None.

SEE ALSO: STATEMENT: END
COMMAND: CONT
DISCUSSION: CONTROL C, THE PANIC BUTTON
DISCUSSION: SOME BASIC CONCEPTS

STATEMENT: END

ACTION: Terminates program execution.

EXAMPLE PROGRAM:
```
10 REM PRINT "2+2=",P
20 END
```

REMARKS: END is similar to STOP, except that you can't CONTinue after an END, nor is any message sent to the console terminal. END causes the end of program execution and a return to DIRECT MODE. It is useful when you want to terminate program execution at some point in the midst of the program.

If normal sequential execution extends past the last statement (the end of the listing) before an END is executed, END will be assumed as the "last" statement. Therefore, you are not required to use END as the last statement in the program.

ERROR MESSAGES: None.

SEE ALSO: STATEMENT: STOP

DISCUSSION: THE FOR-NEXT LOOP

BODY OF THE LOOP

BASIC includes facilities for the FOR-NEXT loop (namely the statements FOR and NEXT) in order to provide for repetition of any arbitrary block of BASIC statements. The block to be repeated (also called the BODY of the loop), symbolized here as {BODY}, is "sandwiched" between a FOR statement and a NEXT statement.

EXAMPLE #1

```
10 FOR I=1 TO 10
     {BODY}
99 NEXT
100 REM More program statements.
```

In EXAMPLE #1, the statements represented by {BODY} will be repeated 10 times unless specific action is taken within the body to terminate repetition prior to the completion of the 10th cycle (for example, see the paragraphs on EXIT, below).

THE CONTROL VARIABLE AND THE LIMIT VALUE

In line 10, I, a numeric variable, is called the CONTROL VARIABLE of the loop. By using I as a counter, BASIC will be able to know when to quit repeating {BODY}. In the example, the first time {BODY} is executed, I will be set to 1 (the INITIAL value, as specified in the FOR statement). After that, whenever execution proceeds through {BODY} and reaches the NEXT statement in line 99, I will be increased by 1. At such times, BASIC will compare I against 10 (the LIMIT value set in line 10). If I is less than or equal to the limit value, execution returns once more to the start of {BODY}, and the cycle begins again. On the other hand, if I is greater than the limit value, then repetition ceases, and execution continues beyond the NEXT statement (in the case of EXAMPLE #1, at line 100).

THE OPTIONAL STEP VALUE

In the example, I was increased by 1 after every repetition of the body. It is often useful for the value of the control variable to be increased by a different amount than 1 each time, or perhaps it should even be decreased! This is accomplished by adding a STEP clause to the FOR statement.

EXAMPLE #2

```
10 FOR J=1 TO 10 STEP 2
     {BODY}
99 NEXT
```

EXAMPLE #3

```
10 FOR K(3)=5 TO 1 STEP -1
     {BODY}
99 NEXT
```

EXAMPLE #2 will repeat {BODY} five times, with successive values of J being 1, 3, 5, 7, and 9. J is increased by 2 after each iteration.

In EXAMPLE #3, {BODY} is also repeated 5 times, but the value of K(3) will decrease by 1 for each iteration.

If the STEP clause is not used in a FOR, then the step value is always assumed to be 1.

Note that, when the step value is positive, the initial value must be less than or equal to the limit value. When the step value is negative, the initial value must be greater than or equal to the limit. If these rules are not followed, {BODY} will never be executed, as in the next example:

EXAMPLE #4

```
10 FOR Q=5 TO 1
20   PRINT "THIS LINE WILL NEVER BE EXECUTED"
99 NEXT
100 PRINT "PAST THE LOOP"
```

RUNning the above program yields only the message

```
PAST THE LOOP
```

on your terminal. In this case, line 20 is the body, but even before it can be executed, BASIC sees that the value of Q is greater than 1, and that, with an implied step of 1, Q will never acquire the limit value of 1, so it does not execute the body at all, and jumps down to line 100 to continue execution.

The initial, limit, and step value expressions in a FOR statement need not be integer in nature. Thus, it is possible to have a loop such as

EXAMPLE #5

```
10 FOR I=.1 TO 10.5 STEP .1
     {BODY}
99 NEXT
100 REM Above loop will repeat 105 times.
```

Because I is a regular BASIC variable, its value may be compared with others or changed outright during repetition, using the IF and LET statements, respectively. Changing the value of the control variable, however, should be done with great care, and is an advanced technique not recommended for the beginning programmer. It is not possible to change the initial, limit, or step values of the loop during iteration. They are permanently set for the given loop when its FOR statement is first executed. (It is suggested that the control variable not be used in the LIMIT or STEP expressions.)

FOR-LOOP NESTING

FOR loops may be executed while other FOR loops are already in progress. This is called NESTING of FOR loops.

EXAMPLE #6

```
10 FOR I=0 TO 9
20   FOR J=0 TO 9
30     PRINT I,J
40   NEXT
50 NEXT
```

In EXAMPLE #6, the loop controlled by J is the body of the loop controlled by I. The statements from 20 to 40 will be repeated 10 times (as I goes from 0 to 9), but these statements in themselves comprise a loop which will also repeat 10 times. The net effect is that, for every change in J, line 30 will have been executed once, but for every change in I it will have been repeated 10 times. As a result of EXAMPLE #6, line 30, the body of the inner loop, will be repeated 10 times 10, or 100, times. The following is a sample of the output generated:

{see next page}

```
 0 1
 0 2
 0 3
  .
  . etc.
  .
 9 7
 9 8
 9 9
```

FOR loops may be nested to any arbitrary depth. However, there must always be a NEXT to match each FOR. Also, a different variable must be used to control each nested loop.

THE OPTIONAL CONTROL VARIABLE IN NEXT

The control variable of a loop may optionally be specified in the NEXT statement which ends that loop.

EXAMPLE #7

```
10 FOR I=1 TO 10
20   FOR J=1 TO 10
30     PRINT I,J
40   NEXT I
50 NEXT J
```

Inclusion of the control variable in the NEXT statement is useful in clarifying the program text (determining which NEXT goes with which FOR). If the optional control variable is specified in the NEXT statement, BASIC will perform a syntax check during program execution and will cause a program error if the control variable specified in the NEXT is not the same as that specified in the matching FOR.

USING EXIT

A FOR loop may be terminated before all the specified repetitions have been performed if an EXIT statement is executed. EXIT is used to transfer program control to a line outside the loop -- that is, before the loop's FOR statement or after its NEXT. EXIT is like a GOTO, in that it causes a transfer of control to the specified line number, but it also tells BASIC to end the current FOR loop -- no more repetitions will be necessary. BASIC uses memory storage to remember information about the FOR loop while it is repeating. EXIT tells BASIC to release the memory used by the current loop. If it is not used to jump out of a FOR loop, then subsequent loops may not

execute correctly.

EXAMPLE #8

```
10 REM Assume a 10-element array A.
20 REM The following searches A from
30 REM element 1 to 10 for the first
40 REM nonzero element.  The index of
50 REM this element will be N.  If all
60 REM elements are 0, N will also be 0.
70 REM A FOR-loop is used for the scan,
75 REM and EXIT stops scan if nonzero found.
76 REM
80 FOR N=1 TO 10
90   IF A(N)<>0 THEN EXIT 130
110 NEXT
120 N=0 \ REM By this point, A is all zeroes.
130 REM By this point, N contains
140 REM correct index or zero.
```

EXITING FROM NESTED LOOPS

Several nested loops may all be terminated prematurely at once using EXIT, but a separate EXIT statement must be used for each embedded loop. For example, if execution is proceeding at line 70 in the inner loop of a two-deep nest (similar to EXAMPLE #6), and it is desired to go to line 600, outside the outermost loop, the following example represents an efficient method of doing so using the EXIT statement:

EXAMPLE #9

```
70 EXIT 71
71 EXIT 600
```

SEE ALSO: STATEMENT: FOR
STATEMENT: NEXT
STATEMENT: EXIT

STATEMENT: FOR <control variable> = <initial value> TO <limit value>
FOR <control variable> = <initial value>
TO <limit value> STEP <step value>

ACTION: Begins a FOR-NEXT loop.

EXAMPLES:
```
15 FOR J=1 TO 10 \ REM Will cause 10 iterations.
25 FOR Q(7)=3 TO 1 \ REM No looping will occur.
40 FOR A=B*7 TO SQRT(X)
50 FOR X=.1 TO 1.3 STEP .1
90 FOR J=3 TO 1 STEP -1
70 FOR I=10+J TO 100+J STEP D(X)
```

REMARKS: For a complete description of the FOR-NEXT loop, see DISCUSSION: THE FOR-NEXT LOOP.

The INITIAL, LIMIT, and optional STEP values may be any numeric expressions.

If the initial value is greater than the limit value and step is positive, or if initial value is less than the limit and step is negative, the body of the loop will not be executed.

ERROR MESSAGES: MISSING NEXT ERROR
BASIC could not find a NEXT statement to associate with the FOR.

SEE ALSO: DISCUSSION: THE FOR-NEXT LOOP
STATEMENT: NEXT
STATEMENT: EXIT

STATEMENT: NEXT
NEXT <numeric variable>

ACTION: Terminates execution of the loop which starts with the matching FOR statement. For a complete description of FOR-NEXT loops, see DISCUSSION: THE FOR-NEXT LOOP.

If the optional numeric variable name is specified as part of the NEXT statement, a check is made to match that variable name against the control variable specified in the corresponding FOR statement.

EXAMPLES:
```
NEXT
NEXT Q
NEXT A(1)
```

REMARKS: It should be noted that the "check variable" in the NEXT statement, while optional in North Star BASIC, is required in almost every other dialect of the BASIC language. The use of NEXT without the check variable can speed program execution.

Upon normal completion of a FOR-NEXT loop, the control variable will contain tne first value that exceeds the limit. To illustrate, here is an example program:

```
10 FOR K=1 TO 5 STEP 2
20 NEXT K
30 PRINT K
```

When RUN, the above generates the following output:

```
7
```

Note that NEXT should not be used as the THEN or ELSE part of an IF statement.

ERROR MESSAGES: CONTROL STACK ERROR
An attempt was made to execute a NEXT statement with no FOR loop in effect. Also, this error occurs when the variable specified in the NEXT statement doesn't match the control variable specified in the previous FOR statement. This usually means that loops are improperly nested.

SEE ALSO: STATEMENT: FOR
DISCUSSION: THE FOR-NEXT LOOP

STATEMENT: EXIT <line number>

ACTION: Terminates execution of the currently-running FOR-NEXT loop and transfers execution to the specified line.

EXAMPLE:
```
20 EXIT 95
```

REMARKS: EXIT is a special form of GOTO, and is used for roughly the same purpose as GOTO -- to transfer program execution from one point to another. The only difference is that EXIT should be used only to "jump" from some point within an active FOR-NEXT loop to a point outside the loop. When "jumping" from point to point within a FOR loop, or when no loop is active, GOTO should be used.

Each use of an EXIT statement terminates only the current FOR-NEXT loop. See DISCUSSION: THE FOR-NEXT LOOP for the correct method of EXITing from nested loops.

ERROR MESSAGES: CONTROL STACK ERROR
EXIT was used when no FOR-loop was being executed.

LINE NUMBER ERROR
See STATEMENT: GOTO

OUT OF BOUNDS ERROR
See STATEMENT: GOTO

SEE ALSO: DISCUSSION: THE FOR-NEXT LOOP
STATEMENT: NEXT
STATEMENT: FOR
STATEMENT: GOTO

DISCUSSION: SUBROUTINES

When writing programs, you will often find that you need to repeat what amounts to essentially the same sequence of statements at various separate locations in the program text. For example, your program may require the user to answer "yes" or "no" to certain questions. After writing the program, you find that sequences similar to that below occur several times in the text:

```
10 REM Get yes or no answer.
15 REM Keeps trying till Y or N ans given.
20 INPUT "PLEASE ANSWER YES OR NO: ",A$
30 IF A$="" THEN 20\REM No ans given.
40 A$=A$(1,1)
50 IF A$="Y" THEN 70\REM OK ans
60 IF A$<>"N" THEN 20\REM Not = Y either.
70 REM At this point, ans was Y or N.
```

It is certainly troublesome for you (and a waste of program space besides) to type the same sequence of statements over and over again. If you required several such answers at one point in the program, of course, you could use a loop to repeat the statements as often as necessary. However, the problem is different when you must perform the same actions in different parts of the program.

A very nice solution to this problem involves writing just one copy of the segment at one point in the program, then somehow telling BASIC to "re-execute" that part whenever necessary. That is, at those points in the program where you need to get a yes or no answer, BASIC would "jump over" to the part of the program which gets the answer, then "return" to the original point to continue on with whatever should happen after the answer has been obtained.

In this situation, the "answer" segment would be called a SUBROUTINE. This subroutine would be "invoked" (or "called") from other parts of the program to perform its single, important task.

North Star BASIC makes available two special statements which provide subroutine capability. (Both are described in detail in their own sections.) The first is GOSUB, which is used to call a subroutine. The GOSUB keyword is followed by a line number, which tells BASIC where the subroutine begins in the program text. BASIC reacts to a GOSUB by transferring execution to the specified line number,

while "remembering" the point where the subroutine was called. (The action of the GOTO is similar, but no calling location is remembered, which makes GOTO unsuitable for subroutine calling.) When the subroutine is finished, BASIC uses the "remembered" location to return to the point in the program immediately after the subroutine was called. BASIC knows when a subroutine is finished only when it executes a RETURN statement. RETURN merely says to BASIC, "go back to the calling point now". It is not necessary to make RETURN the last physical statement in a subroutine, though it turns out that, in practice, this usually happens.

The "answer" program segment above may be turned into a legal BASIC subroutine merely by replacing the last REM statement with RETURN, and translating the appropriate line numbers:

```
1000 REM Subroutine example.
1010 REM Get yes or no answer in A$
1015 REM Keeps trying till Y or N ans given.
1020 INPUT "PLEASE ANSWER YES OR NO: ",A$
1030 IF A$="" THEN 1020\REM No ans given.
1040 A$=A$(1,1)\REM Examine 1st char only.
1050 IF A$="Y" THEN 1070\REM OK ans.
1060 IF A$<>"N" THEN 1020\REM Not = Y either.
1070 RETURN
```

The subroutine may now be called at any point in the program where it is desired to retrieve a yes or no answer. Here is an example, showing how the subroutine at line 1000 would be called:

```
40 PRINT "Are you over 6 feet tall?"
50 GOSUB 1000 \ REM Collect answer in A$
60 REM More program statements.
```

STATEMENT: GOSUB <line number>

ACTION: The location of the statement immediately after the GOSUB statement is "remembered" by BASIC, and program execution jumps to the specified line. GOSUB is used to execute a sequence of statements, called a SUBROUTINE, elsewhere in the program. Execution will resume at the "remembered" location if a RETURN statement is executed as part of the subroutine.

EXAMPLE PROGRAM:

```
10 REM Illustration of subroutines.
20 PRINT "READY TO CALL SUBROUTINE"
30 GOSUB 1000
40 PRINT "WE ARE BACK!"
50 END
```

This example assumes that there also exists a subroutine beginning at line 1000 which sends the message "NOW IN THE SUBROUTINE" to the terminal. If so, RUNning the program produces the following results:

```
READY TO CALL SUBROUTINE
NOW IN THE SUBROUTINE
WE ARE BACK
```

REMARKS: A subroutine may be called while another is in progress. The only limit on this "subroutine nesting" is the amount of memory available during program execution. ("Remembering" the location of the "return" point takes memory space.)

ERROR MESSAGES:

LINE NUMBER ERROR
See STATEMENT: GOTO

OUT OF BOUNDS ERROR
See STATEMENT: GOTO

SEE ALSO: STATEMENT: RETURN
STATEMENT: GOTO
DISCUSSION: SUBROUTINES

STATEMENT: RETURN

ACTION: To conclude a subroutine, RETURN is used to cause program execution to resume immediately after the GOSUB statement which called the subroutine.

EXAMPLE: 1099 RETURN

REMARKS: There are two versions of the RETURN statement in North Star BASIC. This version is for use with subroutines only. Another is used with user-functions. See Chapter K, FUNCTIONS, for details on that version of RETURN.

ERROR MESSAGES: CONTROL STACK ERROR
The RETURN statement was executed when no GOSUB was currently active.

SEE ALSO: STATEMENT: GOSUB
DISCUSSION: SUBROUTINES

DISCUSSION: FUNCTIONS

BUILT-IN FUNCTIONS

When you want to compute the cosine or the square root of a number within your program, how can you do this? Of course, it's always possible to write a subroutine in BASIC to compute the cosine or square root of an arbitrary number, but doing so consumes your time, is likely to slow down your program if the particular computation is needed often, and certainly enlarges the program.

BASIC includes built-in FUNCTIONS, two of which handle cosine and square root calculations, respectively. The other available built-in functions compute many different values, both numeric and string, which programmers often need, and whose availability makes the task of writing efficient programs easier.

When writing a program, if you need the cosine of Ø, write COS(Ø). If you want the square root of 9, use SQRT(9). The function can be used in a program wherever the actual number can. COS(Ø) stands for (and can be used in place of) the number 1. Writing SQRT(9) is the same as writing 3.

ARGUMENTS

The value in parentheses in a function call is called an ARGUMENT to the function. The function will use the value(s) of the specified argument(s) to generate the function value. SQRT(4), for example, uses the numeric value 4 to generate its square root, 2.

All functions in North Star BASIC require at least one argument, and some may require more. If a function requires more than one argument, it will expect them to be separated by commas to form an ARGUMENT LIST within the parentheses.

Expressions can be used as arguments. COS(2*7) represents the same number as COS(14). If the variable A contains the number 14, then COS(A) also is the same as COS(14).

Functions can be used in expressions. Thus, the statement

```
A=?*SQRT(1ØØ)
```

would put the value of 20 in A.

Because expressions can be arguments, and functions can be expressions, functions can be used as arguments. COS(SQRT(100)/10-1) is the same as COS(0).

You must supply functions with the exact number and types of arguments they require, in exactly the order required, or else when the program runs and the erroneous function call is found, BASIC will halt execution and complain of a SYNTAX ERROR. Such an error will occur, for example, if you attempt to use SQRT("HI") in a program or direct statement. The SQRT function wants a numeric argument, and "HI" is a string (see DISCUSSION: USING STRINGS). COS(2,3) causes a SYNTAX ERROR becuase the COS function wants only one numeric argument.

The following pages contain a list and description of all the functions built-into North Star BASIC. Each function description includes the name of the function, the order of expected arguments, as well as the type (numeric or string) and purpose of each. A short paragraph describes the value represented by the function as well as how the arguments relate to that value.

FUNCTIONS USEFUL IN MATHEMATIC OPERATIONS

ABS(<numeric expression>)
Returns the absolute value of the numeric expression. ABS(3)=3, ABS(-3)=3, and ABS(0)=0.

SGN(<numeric expression>)
Returns 1, 0, or -1, indicating whether the <numeric expression> is positive, zero-valued, or negative, respectively. SGN(10)=1, SGN(0)=0, and SGN(-3.2)=-1.

INT(<numeric expression>)
Returns the greatest integer value less than or equal to the value of the argument. INT(3)=3, INT(3.9)=3, and INT(-3.5)=-4.

LOG(<numeric expression>)
Returns an approximation to the natural logarithm of the value of the <numeric expression>. If LOG is called with an argument value less than or equal to zero a program error will occur. LOG(1)=0, LOG(7)=1.9459101, and LOG(.1)=-2.3025851

EXP(<numeric expression>)
Returns an approximation to the value of e raised to the power of the numeric expression. EXP(0)=1, EXP(2)=7.3890562, EXP(-2.3025851)=.1, and EXP(1)=2.7182817

SQRT(<numeric expression>)
Returns an approximation to the positive square root of the numeric expression. A program error will occur if this function is called with a negative argument. SQRT(0)=0, SQRT(10)=3.1622776, and SQRT(.3)=.54772256

SIN(<numeric expression>)
This function computes an approximation to the trigonometric sine of the value of the numeric expression. The expression must specify an angle in radians. (Note that 2 * pi radians = 360 degrees.) SIN(0)=0, SIN(3.1415926/2)=1.

COS(<numeric expression>)
COS computes an approximation to the trigonometric cosine of the value of the numeric expression, which must specify an angle in radians. COS(0)=1, COS(3.1415926/2)=0.

ATN(<numeric expression>)
The ATN function computes an approximation to the trigonometric arctangent function. The angle value returned is expressed in radians. ATN(5)=1.3734007, ATN(1.7)=1.0390722.

FUNCTIONS USEFUL IN STRING OPERATIONS

LEN(<string name>)
Returns the current length of the string held in the string variable named as the argument. If A$="CAT" then LEN(A$) will be equal to 3. If A$ holds the null string, then LEN(A$) will return 0.

CHR$(<numeric expression>)
The CHR$ function returns a one-character string as its value. The argument value (in decimal) specifies the ASCII character code for the character to be returned in the string. Note that the argument to CHR$ can be any integer in the range of 0 to 255. CHR$(65)="A", CHR$(97)="a", CHR$(32)=" " (space), and so on.

ASC(<string constant, string variable, or substring reference>)
Returns a numeric value -- the numeric ASCII code of the first character contained in the argument. The

argument must not be the null string. ASC("B")=66, ASC("CLUNK")=67. (Note that CHR$ and ASC are inverse functions.)

VAL(<string expression>)
Converts the value of the string expression to a number and returns that number as its value. If the expression doesn't evaluate to a legal numeric constant, then a program error occurs. Leading blanks are ignored. VAL("123")=(the number) 123. VAL("000000")=0. VAL("abcde"), VAL(" "), and VAL("") will cause errors. Note that if any non-numeric characters follow the numeric constant which is at the beginning of the string expression, they will be ignored. For example, VAL("123XYZ")=123, but VAL("XYZ123") causes an error.

STR$(<numeric expression>)
This is the inverse function of VAL -- it converts the numeric value of its argument into a string representation of that number, and returns that string as the function value. The format of the string depends upon the default format as specified in a PRINT statement (i.e., free-format if no previous PRINT statement has specified a default format). See STATEMENT: PRINT and DISCUSSION: PRINT FORMATTING for further details.

FUNCTIONS USEFUL FOR SPECIALIZED INPUT

INCHAR$(<numeric expression>)
This function will await the typing of a single character at the input device specified by number in the numeric expression. The character will be returned as a single character string. Control characters as well as printing characters will be returned. Control-C will be returned only if control-C program-interruption has been disabled. (See DISCUSSION: CONTROL-C, THE PANIC BUTTON.) The character will not be echoed by BASIC (printed on the terminal when its key is pressed). Assuming device 0 is the system console and device 1 is a remote terminal, then INCHAR$(0) will return a single character typed at the console, and INCHAR$(1) will return one character typed at the remote location. The following short program will fetch an individual character from the console terminal and will echo it on that terminal's screen, printer, etc:

```
10 T$=INCHAP$(0) \ REM Get the character ...
20 PRINT T$, \ REM      and echo it.
```

INP(<numeric expression>)
This function performs an 8080 or Z80 IN instruction from the input port specified by the argument value. The numeric value returned by the function is the contents of the accumulator (in the range of 0 to 255) after the IN instruction. Note that INP will not wait for valid data, as do INCHAR$, INPUT, and INPUT1, but instead fetches whatever byte value exists at the input port, whether or not that value represents useful data.

FUNCTIONS USED IN MANIPULATING DISK FILES

TYP(<numeric expression>)
This function returns as its value a number which indicates the type (numeric = 2, string = 1, end-of-file = 0) of the next data item in the open disk file with open file number given by the value of the function's argument. See DISCUSSION: DATA FILES for details.

FILE(<string expression>)
Returns a number corresponding to the type of the file specified by the <string expression>, which must evaluate to a legal disk file name as defined in DISCUSSION: DATA FILES. If the argument is not a legal file name, or is not the name of a disk file on a currently loaded diskette, then the value -1 is returned. Assuming that "ABC" is the name of a BASIC program file on a disk in drive 2, then FILE("ABC,2") will return the value 2. FILE("DOS") will return a 0 if the diskette in drive 1 is a system diskette.

MISCELLANEOUS FUNCTIONS

RND(<numeric expression>)
This function returns a pseudo-random numeric value between 0 and 1. The number generated is dependent upon the previous number generated by the function. The very first number in the sequence is called the "seed", or starting value. If the value of the argument is negative, BASIC selects a random seed (based upon the status of the disk system), and computes the value of the function from it. (The "randomizing" effects of using RND with a negative argument are enhanced if user-input is requested between the last disk access and the "negative" call to RND.) If the argument evaluates to 0, the previously computed value is used to generate another pseudo-random value in the sequence. If the argument reduces to a value between 0 and 1, this number is

used as the new seed, the sequence is restarted, and the first value generated from the new seed is returned as the value of the function. The following program will set a random seed and then print 10 pseudo-random values:

```
10 J=RND(-1)
20 FOR J=1 TO 10
30   PRINT RND(0)
40 NEXT
```

EXAM(<numeric expression>)
The EXAM function returns the contents of the computer memory byte addressed by the value of the <numeric expression>. The argument should evaluate to an integer from 0 to 65535. The value returned will be numeric, an integer from 0 to 255.

FREE(<numeric expression>)
Returns the current total number of bytes remaining in the BASIC memory for additional user-program or data. Free storage, as this memory area is called, is also used for internal "bookkeeping" storage and storage of temporary values used by BASIC, such as string values during concatenation. The argument value, as long as it is numeric, is ignored, and most programmers use 0.

TAB(<numeric expression>)
This function can only be used in a PRINT statement. Use of the TAB function will cause the cursor or print-head of the output device specified in the PRINT statement to advance to the character position specified as argument to TAB. BASIC accomplishes this by printing the appropriate number of spaces. The first character position on a line is the 0th position, all others being numbered sequentially from 0. If the cursor or print-head is past the specified position, then it will not move at all.

CALL(<numeric expression>)
CALL(<numeric expression>, <numeric expression>)
CALL permits BASIC programs to use machine-language subroutines. The value returned is an integer from 0 to 65535, which represents the value in the HL register-pair when the machine-language subroutine returns control to BASIC. The first argument to CALL is a numeric value from 0 to 65535 which represents the decimal value of the memory address where the machine-language subroutine begins. The optional second argument, also an integer value from 0 to 65535, will be passed to the machine-language routine

in the DE register pair. For more information on CALL and the use of machine-language subroutines in general, see DISCUSSION: MACHINE LANGUAGE SUBROUTINES.

USER-FUNCTIONS

Functions may be written in North Star BASIC as part of a BASIC program. They are accessible (just as built-in functions are) to any part of the program. These USER-FUNCTIONS can return either string or numeric values, and can accept as many string and/or numeric arguments as are necessary to compute the function value.

FUNCTION NAMES

User-functions take names of the following form: the two letters FN followed immediately by a regular string or numeric variable name, as in FNX, FNQ7, FNA$, FNZ3$, etc. The type of the variable-name part of the function name determines the type of the value that the function returns. FNX, therefore, is a numeric user-function, while FNA$ returns a string value. Note that user-function names are separate and distinct from variable names. In particular, the values returned by FNA$ (for example), will not affect the value stored in variable A$, nor will assignment to A$ change the value that FNA$ returns.

SINGLE-LINE FUNCTIONS

A user-function can be defined by a single line, or may require many lines to define. For example, the following is a one-line user-function:

```
10 DEF FNR(V,P)=INT((V*10↑P)+.5)/(10↑P)
```

FNR, as defined in the DEF statement above, will return as its value V rounded-up to the P-th decimal place. For example, FNR(3.1415,2) makes V stand for 3.1415, and P for 2. The value returned will be 3.14.

PASSING VALUES TO USER-FUNCTIONS

A DEF statement must include a list of string and/or numeric variable names, called PARAMETERS to the function. This parameter list is enclosed in parentheses following the function name. For example, in the following DEF statement, X$, Y, and Z are parameters to function FNW:

```
50 DEF FNW(X$,Y,Z)=LEN(X$)+Y+Z
```

A FUNCTION CALL must include a list of string and/or numeric expressions. This expression list is

enclosed in parentheses following the function name. When a function is called, the values of the expressions in the expression list are assigned, one-by-one, left-to-right, to the corresponding variables in the parameter list of the called function. After this assignment process, the variables named in the parameter list will contain the corresponding values from the expression list and can be used in the body of the function in computing the function value.

The number of expressions in the function call's expression list must match the type of the corresponding parameter in the parameter list. If the types or number of parameters in the function definition do not match the types or number of expressions in the function call, an ARGUMENT MISMATCH ERROR or a SYNTAX ERROR will occur.

NUMERIC PARAMETERS

At function call time, before each numeric variable in the parameter list is assigned its value from the expression list, the value of the variable is saved by BASIC. When function execution is completed, the saved values of the numeric variables from the parameter list are restored as the values of those variables. Thus, the values of the numeric variables from the parameter list after the function call is completed remain the same as before the function was called. This means that the numeric parameters of a function may be thought of as separate variables when used during function execution.

Try the following:

```
10 DEF FNX(B)=B*3
20 B=2 \ PRINT B
30 PRINT FNX(3)
40 PRINT B
```

B prints-out as 2 before as well as after FNX is called, even though B=3 during the evaluation of FNX because of the B-value of 3 supplied in parentheses in the function call.

STRING PARAMETERS

Unlike those of numeric parameters, the values of string parameters of a function are not saved at function call time. Thus, after function execution is completed, those variables will retain the most recent values they acquired during function

execution. Note that the assignment of string expressions to string parameters at function call time follows the same rules as assignment to string variables in LET statements. In particular, if the string parameter has not been DIMensioned as a string variable before the function call, it will automatically be DIMensioned to maximum length of 10.

To contrast the treatment of string and numeric parameters at function call time, try this program:

```
10 DEF FNQ(X,X$)=ASC(X$)+X
20 X=7 \ X$="FIRST"
30 PRINT X$,X
40 PRINT FNQ(1,"NEXT")
50 PRINT X$,X
```

Note that, although the value of the numeric variable X is saved while the name of X is used for an argument to FNQ, the same is not true for X$. After the function is evaluated, X$ still retains the value it was assigned during its use as FNQ argument.

MULTI-LINE USER-FUNCTIONS

The second type of user-function, the multiple-line function, permits a value to be computed and RETURNed by a set of one or more BASIC statements, as opposed to the single expression of the single-line function. The operation and purpose of multi-line functions therefore closely parallels that of subroutines. However, multi-line functions permit the easy passing of arguments, and the return of a single, computed result value.

The definition of a multi-line function employs the DEF statement, but without the "value equation" necessary to single-line function definitions. The DEF statement which begins a multi-line function contains only the keyword DEF, the name of the function, and the list of its parameters:

```
10 DEF FNM(X,M)
```

The statements which compute the function value follow this line. When the value has been computed, a special version of the RETURN statement causes function execution to cease, and specifies the value to be RETURNed as the function value. Finally, to signal the physical end of the function definition itself, the FNEND statement is used. As an example, add to the definition of FNM (started in line 10,

above) so that it becomes a function which RETURNs the value of X modulo M, that is, the remainder generated when X is divided by M:

```
10 DEF FNM(X,M)
20 IF M<=0 OR M<>INT(M) THEN 40
30 RETURN ABS(X)-(INT(ABS(X)/M)*M)
40 PRINT "ERROR IN MODULO" \ RETURN -1
50 FNEND
```

In general, multi-line functions (as opposed to single-line ones) are needed when the algorithm which computes the function value is too complex to fit on one line as a single expression.

SOME FINAL NOTES

Functions cannot be defined within other functions. One definition must finish before another can begin. In particular, a "FUNCTION DEF ERROR" will occur if you forget to include the FNEND which must conclude every multi-line function definition, then, later in the program text, attempt to define another function.

All user-functions must have at least one (1) parameter. It is not necessary to use the parameter in computation, but it must be a part of the definition, nevertheless.

It is not possible to pass entire numeric arrays as arguments to user-functions, but individual elements of arrays, like simple variables, are allowed. Thus, FNQ(A(3),"GAIL") is a proper call of the function given as example above.

User-functions cannot be called in direct mode. If you use a statement in direct mode which includes an expression with a call to a user-function in it, you will get an "ILLEGAL DIRECT ERROR".

SEE ALSO: STATEMENT: DEF
STATEMENT: RETURN (CHAPTER K)
STATEMENT: FNEND

STATEMENT: DEF <function name>(<parameter list>)=<expression>
DEF <function name>(<parameter list>)

ACTION: The first form defines a single-line user-function, numeric or string. When evaluated, the single-line function returns the value of the expression on the right side of the equal sign. The type of the expression must match the type of the function name, string or numeric.

The second form begins the definition of a multi-line user-function. The function value in this case is determined by the expression in the RETURN statement used in the body of the function definition itself. The type of the expression in any RETURN statement in the function body must be of the same type as the function name.

A user-function name consists of the letters FN followed by a string or numeric variable name (such as FNA$, FNQ7, etc.).

EXAMPLES: (single-line)
```
70 DEF FNH(X,Y)=SQR((X↑2)+(Y↑2)) \ REM Hypotenuse
45 DEF FNU$(L$)=CHR$(ASC(L$)-32) \ REM Low to upp case
```

(multi-line)
```
110 DEF FNQ(A,B,C)
589 DEF FNA7$(A$,A,M)
```

REMARKS: The addition of the FN prefix distinguishes function names from variable names. FNA and variable A are not the same, nor even necessarily related.

If a DEF statement is encountered during program execution, then execution will skip forward to the first statement after the definition. Function definitions may be located anywhere in the program text. Function definition occurs before program execution begins.

ERROR MESSAGES: FUNCTION DEF ERROR
An (apparently) single-line function was defined improperly, or an attempt was made to define a function within the definition of a multi-line function.

SEE ALSO: DISCUSSION: FUNCTIONS (user-functions)
STATEMENT: RETURN (CHAPTER K)
STATEMENT: FNEND

STATEMENT: RETURN <string or numeric expression>

ACTION: The evaluation of the multiple-line user-function currently in progress terminates. The function value becomes the value of the expression in the RETURN statement.

EXAMPLES:
```
10 RETURN F$+",2"
20 RETURN A
65 RETURN X+3
99 RETURN "CONSTANT"
```

REMARKS: Do not confuse this form of the RETURN statement with that which is used for subroutines. Improper utilization of this form to conclude a subroutine, or of the subroutine form to terminate a multi-line user-function will result in a SYNTAX ERROR.

The value RETURNed by a multi-line function must be of the same type as the function name. String functions may not RETURN numeric values, and numeric functions may not RETURN string values.

ERROR MESSAGES: SYNTAX ERROR
The RETURN expression doesn't match the function type.

SEE ALSO: DISCUSSION: FUNCTIONS (user-functions)
STATEMENT: FNEND
STATEMENT: DEF
STATEMENT: RETURN (CHAPTER J)

STATEMENT: FNEND

ACTION: FNEND marks the end of the segment of program text which constitutes a multiple-line user-function definition.

EXAMPLE FUNCTION:
```
10 DEF FNF(X) \ REM Compute factorial.
15 X=INT(ABS(X)) \ REM Eliminate bad arguments.
20 IF X=0 OR X=1 THEN RETURN 1 ELSE RETURN FNF(X-1)*X
30 FNEND
```

REMARKS: The FNEND statement should not be confused with the RETURN statement used to end multi-line user-function execution.

The FNEND statement may not appear on the same program line as a DEF statement.

ERROR MESSAGES:

CONTROL STACK ERROR
The FNEND statement is not supposed to be executed. This error results when an FNEND statement is executed.

FUNCTION DEF ERROR
The FNEND statement is on the same line as a DEF statement, or an FNEND statement exists which cannot be matched with a corresponding DEF statement.

SEE ALSO: DISUCSSION: FUNCTIONS (user-functions)
STATEMENT: DEF
STATEMENT: RETURN (CHAPTER K)

DISCUSSION: DATA FILES

Data is stored on diskette in FILES. A file is a section of storage space on the diskette which is reserved for data storage use by giving it a FILE NAME and three other attributes: a LENGTH (or SIZE), a TYPE, and an INFORMATION DENSITY. You can list this information for each file on diskette by using the CAT command. Each CATalog listing is of the following format:

```
NAME    LOC    SIZE    TYPE    DENSITY    TDI
```

For example, the listing

```
PROG1    73    20    2    D
```

denotes a file named "PROG1", starting at sector 73 on the diskette, with a size of 20 256-byte disk blocks, and of type 2. The "D" at the end of the CATalog listing signifies that the information stored in the "PROG1" file is stored in double-density format. If a file is stored in single-density, an "S" will appear in this position instead. (The "type dependent information", or TDI, is not shown in this example, is rarely used, and will not occur in any of our examples.) All this file information is stored in a special place on the diskette (the first four sectors, Ø to 3) called the DIRECTORY.

FILE NAMES

The NAME of a file consists of a series of not more than 8 printable characters. (The "printable characters" include the upper and lower case alphabets, the digits Ø to 9, and the various punctuation symbols.) Any characters may be used in any order, with the exception of the space and the comma. The space may not be used anywhere within a file name. The comma may only be used in a specific situation, which will be discussed in a moment. The name of a file must be unique on a diskette -- that is, two or more files may not share the same file name on the same diskette. For example, only one file on a diskette may have the name FILE1. However, it should be noted that the upper and lower case sets of letters are considered to be separate and distinct with respect to the names of files, so FILE1 and file1 are not the same file name, and may be used to name different files on the same diskette. A DRIVE NUMBER SUFFIX may be added to the name of a file to indicate that the desired file is located on a

diskette in a specific drive, which resolves any possible confusion between files of the same name on different diskettes. The drive-suffix is formed by following the name of the file with a comma, and then a single digit, corresponding to the selected drive. If, for example, the file "PROG" is on the diskette in drive #2, the proper way to write its name is "PROG,2". File "POP" in drive #3 would be called "POP,3". If no suffix is given, then the system assumes that the file is on the diskette in drive #1. The file names "SYNONYM" and "SYNONYM,2" refer to separate files on different diskettes.

A FILE NAME is an unambiguous reference to a specific file, and so specifies not only the file's name on diskette, but also the drive in which it is located. Thus, a FILE NAME consists of an actual name of no more than 8 printable characters plus an optional drive-suffix (which is assumed to reference drive #1 if omitted). A file name is a string value. Statements which require file names as arguments will accept any string expression, as long as it evaluates to a legal file name.

FILE SIZES (LENGTHS)

The size of a file is specified in FILE BLOCKs. A file block is 256 bytes of information. In the directory CATalog listing, the size of a file is given in file blocks.

In a double-density system, each file must have an even number of file blocks, because file space on diskette is allocated in terms of SECTORS. Two file blocks will fit in one sector of a double-density system. In single-density systems, a disk block is the same as a sector.

Each file in North Star BASIC occupies a contiguous section of disk storage. A file may be any number of file blocks in length, provided that there is sufficient contiguous storage space for it on the diskette.

FILE TYPES

Every file has a type, which can be used to classify a file according to how it is used. For example, the North Star convention is that a type 2 file always holds a program written in BASIC. A file of type 3 is used to store data used by BASIC programs. A type 1 file should contain an executable machine language

program, such as the BASIC interpreter itself. These, however, are only 3 of the 128 possible type designations (from Ø to 127). You are free to use the others as you wish, to signify special types of file contents which are meaningful for you. For example, you could write a special business program and arbitrarily declare that all data files relating to it would be of type 7. Facilities within North Star BASIC allow you to determine a file's type when accessing or creating it.

CREATING FILES

A file must be created, and space reserved for it, before it may be used to store data. The CREATE statement may be used to create a file of any type or length, on a diskette in any disk drive. The density of the file created is set to be the same as the density of the file directory on the diskette. Once created, the file's size in file blocks is fixed. The amount of information in that file can never exceed the allocated space.

OPENING FILES

Before you can access a data file, you must associate its file name with a FILE NUMBER using the OPEN statement. From that point on, use the designated file number when referring to the file. For example, suppose "ACCT" is OPENed as file #2. Then, all BASIC statements in your program which are intended to access "ACCT" should refer to file #2, instead of the actual file name.

CLOSING FILES

When you are finished using a file, the CLOSE statement will free the file number associated with the file so that another file may be OPENed with that number.

Closing a file also causes any information which is part of the file but which is temporarily stored in RAM memory to be written to the file on diskette.

If your program requires manual "swapping" of several diskettes in and out of one drive, it is essential that all files on a given diskette be CLOSEd before it is dismounted from the drive. This is to ensure that all the latest changes in the files' contents are actually transferred to the diskette. More importantly, it ensures that no subsequent WRITE

activity intended for these files will occur on the wrong diskette.

TYPES OF DATA ELEMENTS IN FILES

Three types of data may be stored in BASIC data files: NUMBERS, STRINGS, and separate BYTES. Each type of item takes up a certain amount of space on the file when it is stored. Numbers always take up a fixed amount of space. This space is sufficient to hold any numeric value. Strings can take up variable amounts of space, depending upon the current length of the string when it is written to a file. Separate byte values require only one byte of disk storage space to store. Each element of byte information contains a binary integer value from Ø to 255.

BASIC writes strings and numbers to data files using a certain well-defined formats. Consequently, it is easy for BASIC to "recognize" string and numeric data when a file is READ. Bytes, however, cannot be so identified. The programmer must always know when byte data will be encountered during file reading and writing. If such knowledge is not available to a file READing program, it may be impossible for that program to make sense of a file's contents.

DATA ACCESS

READ# AND WRITE#

The two statements which permit input from a file and output to a file are READ# and WRITE#. READ# inputs data from a file and assigns it to variables as specified by the programmer. WRITE# overwrites any previously existing information at a given point in the file with new information, also as specified by the programmer. (See STATEMENT: READ# and STATEMENT: WRITE# for specific details.) READ# and WRITE# may be used to access string, numeric, or byte-valued information in SEQUENTIAL or RANDOM fashion. The rest of this DISCUSSION examines these data-access methods.

SEQUENTIAL ACCESS

The simplest files consist of sequences of data values (all string, all numeric, all byte, or combinations of these). This means that the first data value is located at the start of the file, and succeeding values follow immediately afterward, one after another. BASIC automatically places a special

end-of-file mark (called an ENDMARK) after the last value in a sequential file. This facilitates later READing of the file, because the ENDMARK may act as a signal to the program to quit READing, lest a program error occur when an attempt is made to READ (or READ past) the ENDMARK.

A check for the ENDMARK can be made with the built-in TYP function. TYP, when supplied with the number of an open file as argument, returns the numeric code for the type of the next element to be READ from that file:

TYPE	NEXT VALUE
0	ENDMARK
1	string
2	number

Therefore, if the value of TYP(1) is 0, then the end of file #1 has been reached, and no more READing from that file should be attempted. The TYP function also permits a program to know whether to READ a string or numeric value next, since the types for those data elements are also returned. This is important, because a program which tries to READ a numeric value into a string variable, or a string value into a numeric variable will generate a TYPE ERROR. With this in mind, here is a program which READs an existing sequential data file whose contents include an unknown sequence of intermixed string and numeric values, then PRINTs the contents to the console terminal:

```
10 REM Report contents of sequential
20 REM data file of unknown structure.
25 REM Assume no string bigger than 500 chars.
30 DIM S$(500),F$(10)
40 REM S$ will hold string values read,
50 REM F$ will hold file name, and
60 REM N will hold numbers read.
70 INPUT "TYPE NAME OF FILE TO READ: ",F$
80 OPEN #1,F$
90 IF TYP(1)=0 THEN 240
100 REM Above is ENDMARK check.
110 IF TYP(1)=2 THEN 190
120 REM Above checks if number is next --
130 REM if not, string is next.
140 REM READ/PRINT string.
150 READ #1,S$
160 PRINT S$
170 REM Go back for more data.
```

```
180 GOTO 90
190 REM READ/PRINT number.
200 READ #1,N
210 PRINT N
220 REM Get more data.
230 GOTO 90
240 REM No more data.
250 PRINT "** END OF FILE **"
260 CLOSE #1
270 END
```

The following sample program WRITEs the numbers 1 to 10 to existing data file "DAT7", then READs them back and PRINTs them on the terminal. Note that, after writing, the file is CLOSEd and re-OPENed in order to begin READing at the start, since the last-executed WRITE statement leaves BASIC "looking" at the ENDMARK.

```
10 REM WRITE 10 numbers to file
20 REM and READ them back again.
30 REM First, WRITE 'em!
40 OPEN #1, "DAT7"
50 FOR I=1 TO 10
60   WRITE #1,I
70 NEXT
80 CLOSE #1
90 REM Now, READ and PRINT.
100 OPEN #1,"DAT7"
110 IF TYP(1)=0 THEN 170
120 REM Above checks for ENDMARK.
130 READ #1,I
140 PRINT I
150 REM Now back for next number.
160 GOTO 110
170 REM Quit.
180 PRINT "** END OF FILE **"
190 CLOSE #1
200 END
```

APPENDING TO SEQUENTIAL FILES.

To add new data to the end of an existing sequential file, it is necessary to READ to the ENDMARK before beginning to WRITE. If the sequential file "DAT7" already contains the numbers 1 to 10, then the following program will add the numbers 11 to 20 to its end.

```
10 REM Add 11-20 to DAT7 file.
20 OPEN #1,"DAT7"
30 REM Now READ to ENDMARK.
```

```
40 IF TYP(1)=0 THEN 70
50 READ #1,N
60 GOTO 40
70 REM Now add the numbers.
80 FOR I=11 TO 20
90    WRITE #1,I
100 NEXT
110 REM Quit.
120 PRINT "DONE"
130 CLOSE #1
140 END
```

SEQUENTIAL BYTE ACCESS

Files may also be accessed at the byte-by-byte level simply by using the ampersand character (&) to prefix variables into which values will be READ, or to prefix expressions to be written:

```
10 REM READ a byte value, then WRITE one.
20 REM Assumes file #1 is OPEN.
30 READ #1, &X
40 REM Byte goes into X.
50 WRITE #1,&65
60 REM Byte value 65 goes to file #1.
```

Only numeric expressions and variables may be given the &-prefix. Byte values are integers in the range 0-255, and naturally, since BASIC automatically converts from decimal to binary and back, each consumes only one byte of file storage space. You should be sure that any value you intend to WRITE as a byte to a file lies in the legal byte range.

Note than an ENDMARK will always be written after the last data item in a WRITE statement, whether or not that last item is a byte-value. To disable writing of the ENDMARK, use the NOENDMARK option in your WRITE statements.

RANDOM DATA ACCESS

BASIC keeps track of where it is supposed to READ and WRITE next in an open file by maintaining a FILE POINTER for it. This pointer specifies the number of bytes from the start of the file to the current READ/WRITE position. This number is called a RANDOM FILE ADDRESS. When a file is OPENed, its file pointer is set to 0, meaning that the first data access will happen at the start of the file. You can change the value of the pointer, and so access file data beginning at any point in a file. This is

called "random access" and is one of the quickest means of storing and retrieving data in files because it is not necessary to READ all the data items in a file in order to get to the one you want. By changing the file pointer to reference the location of the data-item you seek, you can READ or WRITE it immediately.

A RANDOM ADDRESS EXPRESSION is added to a READ# or WRITE# statement in order to access data randomly. The random address expression is a numeric expression following a percent sign (for example: %R*5). The expression must evaluate to an integer from 0 to the value

```
SIZE*256-1
```

where SIZE represents the size of the file in disk-blocks. If an address-expression is ever negative or greater than the limit given by the above formula, a program error will occur.

In order to use random access, you must be able to determine the necessary random address of the particular piece of data you want. The easiest way to do this is to require that all items in the file be of the same type or size. For example, a file intended for random access might consist of all numbers, or all 10-character strings. Alternately, a random access file might contain 100 records of 62 bytes each. Each record might consist of 4 numbers in a row, plus a string of length 40.

How was the figure of 62 bytes for the record size computed? In order to find out how much disk storage space a group of items will require, you must add-up all the actual sizes of each of the elements. Refer to APPENDIX 3: IMPLEMENTATION NOTES, for information on computing the storage-sizes for strings and numbers.

Knowing exactly how long each element or record is, you can treat the entire file as a huge array of items or records, computing the random address of the Xth item in the file with the following expression:

```
(X-1)*R
```

where R is the size of an individual record or item, given in bytes. Add a per-cent sign in front of this expression, and you have a legal random address expression! To illustrate, given a file of strings,

the storage length of each being 42 bytes, then the first string would occur at address 0, which is (1-1)*42. The 50th string occurs at random address (50-1)*42 = 49*42 = 2058.

Random access records may easily be updated in place, although you must still use NOENDMARK to avoid the writing of an ENDMARK after rewriting the record. (The extra ENDMARK could contaminate the data in the next record!)

Here is a program which accesses any element of a random access file of 1000 strings, each of which is 250 characters long:

```
10 REM Random string access.
20 OPEN #1,"RANDSTR"
30 DIM R$(250)
40 R=250+2
50 REM R is size of one item -- see
55 REM implementation notes for details.
60 INPUT "WHICH STRING (1-1000, 0 TO QUIT)? ",I
70 IF I=0 THEN 130
80 IF I<1 OR I>1000 THEN 60
85 REM Check for out of range item number.
90 READ #1 %(I-1)*R,R$
100 PRINT "STRING #",I,": ",R$
110 PRINT
120 GOTO 60
130 PRINT "QUIT"
140 CLOSE #1
150 END
```

Byte values may also be accessed randomly using these same techniques, provided that the ampersand is employed to specify byte access.

SEE ALSO: STATEMENT: OPEN
STATEMENT: CLOSE
STATEMENT: READ#
STATEMENT: WRITE#
STATEMENT: CREATE
STATEMENT: DESTROY
DISCUSSION: FUNCTIONS (built-in: TYP, FILE)

STATEMENT: CREATE <file name>, <file size>
CREATE <file name>, <file size>, <file type>

ACTION: A new file of the specified name, size and type is created on diskette. The file size and (if present) the file type must be numeric expressions which evaluate to non-negative integer quantities. The file size refers to the number of 256-byte blocks the file will contain and can be no more than the number of free file blocks remaining at the end of the diskette. The file type must be no greater than 127. If no type is specified, type 3 (BASIC data file) is assumed. The file name may be any string expression whose value constitutes a legal file name (see DISCUSSION: DATA FILES). The density of the file created is set to be the same as the density of the file directory on the specified diskette.

EXAMPLES:
```
CREATE "SAMPLE",25
CREATE "DATA,2",100,10
CREATE F$+D$,S,T
```

REMARKS: CREATE merely reserves disk space in the directory under the given file name -- no information of any kind is actually written into a file when it is CREATEd.

ERROR MESSAGES:

FILE ERROR
Either the file name is illegal, or there is not enough room on the diskette to hold a file of the indicated size.

OUT OF BOUNDS ERROR
The file type specified is not in the range 0 to 127, or the specified file size is out of legal size range.

HARD DISK ERROR
See COMMAND: SAVE

SEE ALSO: DISCUSSION: DATA FILES
STATEMENT: DESTROY
DISCUSSION: FUNCTIONS (built-in: FILE)

STATEMENT: DESTROY <file name>

ACTION: The file specified by the the file name is removed from its diskette. The "file name" may be any string expression whose value is a legal file name. (See DISCUSSION: DATA FILES.)

EXAMPLES:
```
DESTROY "VICTIM"
DESTROY F$+",2"
DESTROY "TEMP"+D$(1,1)
```

REMARKS: The DESTROY statement is equivalent to the DE command in the DOS.

ERROR MESSAGES:

FILE ERROR
The file name is illegal, or the named file does not exist.

HARD DISK ERROR
See COMMAND: SAVE

SEE ALSO: DISCUSSION: DATA FILES
STATEMENT: CREATE

STATEMENT:
```
OPEN #<file number expression>, <file name>
OPEN #<file no. expr.>, <file name>, <size variable>
OPEN #<file no. expr.> %<type expression>, <file name>
OPEN #<file number expression> %<type expression>,
     <file name>, <size variable>
```

ACTION: The diskette file with the given name is assigned the specified file number. Until the file is CLOSEd, it may be referenced by using the file number. The file number expression must evaluate to an integer from 0 to 7. If the optional type expression is omitted, the named file must be of type 3 (BASIC data file) for the OPEN to be successful. The OPEN will succeed if and only if the file is of the given type. The type expression must evaluate to an integer from 0 to 127. The file name may be any string expression and must evaluate to a legal file name as specified in DISCUSSION: DATA FILES. If the optional size variable is used, the size of the successfully OPENed file, given in 256-byte disk blocks, will be assigned to the specified numeric variable.

EXAMPLES:
```
OPEN #1,"DATA"
OPEN #7%4,"CUSTLIST"+D$
OPEN #F%T,F$,S
```

REMARKS: An active file-number must be "freed" by a CLOSE statement before it may be re-used in a BASIC program (used again in an OPEN statement).

A RUN, END, SCR, LOAD or CHAIN will close all open files.

ERROR MESSAGES:

TYPE ERROR
The named file is not of the type specified in the OPEN statement (type 3, if no type is explicitly specified).

FILE ERROR
This is caused by three conditions:
1) The file number is already assigned to a file.
2) The file name has been formed incorrectly.
3) The named file does not exist on the diskette in the specified drive.

OUT OF BOUNDS ERROR
The file number or type value is out of range.

SEE ALSO: DISCUSSION: DATA FILES
STATEMENT: CLOSE

STATEMENT: CLOSE #<file number expression>

ACTION: Prevents further access to the file with the specified file number. Also guarantees that RAM buffer space for the file is written to the file on diskette if necessary.

EXAMPLES:
```
CLOSE #1
CLOSE #A*2
CLOSE #B7(3)
```

REMARKS: Files should be CLOSEd as soon as there is no longer any need to READ from or WRITE to them. This insures that any changes made to the files will be permanent, because the buffer is written out, if necessary, when a CLOSE occurs.

The "buffer-flushing" action of the CLOSE statement, where accumulated data is actually written to the diskette file, will also occur under the following circumstances:

a) The file pointer is changed to address a byte location in another file block.

b) An END or CHAIN statement is executed.

c) A STOP statement is executed or a control-C interruption occurs.

d) The program halts because of a program error.

Only the execution of CLOSE, END, or CHAIN statements, however, will disassociate the diskette file from its file number. During an interruption caused by STOP, control-C, or a program error, any files OPENed within the program remain OPEN, and may be accessed in direct mode.

ERROR MESSAGES: FILE ERROR
The file number expression did not evaluate to an integer from 0 to 7, or the diskette is write-protected.

SEE ALSO: STATEMENT: OPEN
DISCUSSION: DATA FILES

STATEMENT: READ #<file number expression>, <variable list>
READ #<file no. expr.> %<random address>, <var. list>

ACTION: For each variable in the list, the next sequential data value from the specified diskette file is obtained, and assigned to the variable. READing of values may commence at a specified point in the file (x-many BYTE positions from the start) if the random address is used. The address specification consists of a per-cent sign (%) followed by a numeric expression which evaluates to an integer between 0 and the last legal byte address within the file. The file number is a numeric expression of integer value from 0 to 7. Any numeric variable in the list may be prefixed with an ampersand (&) which instructs BASIC to READ the next byte of data and assign its decimal value (interpreted as an integer from 0 to 255) to the variable.

EXAMPLES:
```
READ #2, A,B,C
READ #3,Q,&B7,A$
READ #F%L,&X,&Y,&Z
READ #0%FNL(I)+3,R8,Z$,R9
```

REMARKS: BASIC maintains a "pointer" into each open file. When the file is OPENed, the pointer is set to the beginning of the file, this pointing to the first byte of the first value in the file. Each time a value is assigned to a variable, the file "pointer" moves past that value, and points to the first byte of the next value in the file.

Use of the optional random address expression resets the file pointer to the specified byte address in the file, before READing begins.

ERROR MESSAGES: TYPE ERROR
The types of the variable and the value to be assigned to it do not match. For example, this will occur if an attempt is made to READ a string value into a numeric variable. A TYPE ERROR also occurs when an attempt is made to READ more data than is included in the file (READing the ENDMARK). This error will also occur if use of random-accessing results in the file pointer being set to, for example, the "middle" of a string or numeric value in the file.

OUT OF BOUNDS ERROR
Either or both of the following conditions has occurred:

1) The random access address is less than 0 or greater than (the file size in blocks)*256-1.

2) The file number is less than 0 or greater than 7.

SEE ALSO: DISCUSSION: DATA FILES
STATEMENT: WRITE#

STATEMENT: WRITE #<file number>, <expression list>
WRITE #<file no.> %<random address>, <expr. list>

ACTION: Each value in the expression list is written to the diskette file to which the file number refers. If there is more than one value in the expression list, the values are written sequentially (one-after-another) in the order listed. After all the values in a WRITE statement's expression list have been written to the specified file, an ENDMARK is written after the last item. Note that after any WRITE operation which WRITEs an ENDMARK, the file pointer will point to the ENDMARK just written. In this way, new data placed at the end of the file will overwrite old ENDMARKs, and the result is that there is always only one ENDMARK in a file after proper sequential access. The programmer may opt to suppress the writing of the ENDMARK by using the reserved-word NOENDMARK as the last item in the WRITE statement. Writing may begin at any arbitrary point in the file if the random address, an offset (calculated in bytes) from the start of the file, is included. Both the file number and the random address may be any valid numeric expressions, so long as the file number evaluates to an integer from 0 to 7 (corresponding to an opened file), and the random address is an integer between 0 and the last byte address in the file. Any numeric expression in the expression list may be prefixed with an ampersand (&) character. This signals BASIC to convert the value to a single byte and WRITE it to the file. (Any value so prefixed must evaluate to an integer from 0 to 255.)

EXAMPLES:
```
90 WRITE #1,A,B,C$
75 WRITE #F, "HI THERE",Q,X7(B),NOENDMARK
80 WRITE #0%P,R$
33 WRITE #X, &B1,&B2,&1
20 WRITE #3%Z(M), &E, NOENDMARK
30 WRITE #2%(R-1)*S,X$,Y$,Z$
```

REMARKS: Even when & is used to cause writing of individual bytes, an ENDMARK is still written after the values in the expression list. Thus

```
WRITE #1,&B
```

will result in the writing of two bytes, the byte-value of B and the ENDMARK. When the intention is to write only a single byte using a single WRITE statement, the NOENDMARK option should be exercised.

ERROR MESSAGES:

FILE ERROR
The diskette containing the specified file is write-protected.

OUT OF BOUNDS ERROR
Either or both of the following conditions has occurred:

1) The random access address is less than 0 or greater than the file's highest permissible random address.

2) The file number is not within the range of 0 to 7.

SEE ALSO:

DISCUSSION: DATA FILES
STATEMENT: READ#
STATEMENT: OPEN
STATEMENT: CLOSE
APPENDIX 3: IMPLEMENTATION NOTES

STATEMENT: FILL <memory address>, <byte value>

ACTION: The byte value is placed in the RAM memory cell with the specified address. A byte value is a numeric expression which evaluates to an integer from 0 to 255. The memory address must be a numeric expression equal to an integer from 0 to 65535.

EXAMPLES:
```
FILL M+S,0
FILL (2*16↑3)+(13*16↑2)+(1*16↑1)+(3*16↑0),16
FILL FNC("2D13"),16
FILL 65535,B
FILL 100,31
```

REMARKS: The FILL statement allows the user to change specific bytes in RAM memory, and so is useful in the following applications (as well as many others):

1) Personalizing BASIC.

2) Loading user-defined machine language routines in free memory.

3) Putting parameters to machine-language user-functions in free memory.

4) Manipulating video-display memory for custom graphics applications.

Note that both the memory address and the byte value must be in decimal (base 10) form, and BASIC will convert them to binary when FILL is executed. North Star BASIC does not accept hexadecimal (base 16) numbers. If you wish to use "hex" when specifying addresses of byte values, you should make use of a hex-to-decimal conversion function. Refer to DISCUSSION: FUNCTIONS for an explanation of user-functions, as well as APPENDIX 1: SAMPLE PROGRAMS for a user-function written at North Star to perform the conversion.

If either the byte- or address- values reduce to non-integers, the fractional portion is eliminated (TRUNCATED) and the remaining whole portion is used.

If, after truncation, the byte value is greater than 255, only it's remainder, when divided by 256 (in other words, the value modulo 256) is used. (For example, 257 modulo 256 =1 -- FILL X, 257 would put a 1-byte in the address represented by X.) No similar provision is made for the memory address, however.

CAUTION: FILL may reference an address at which no memory cell exists or even an address within DOS, BASIC, or the program/data area. Thus, FILL gives the programmer power to make some very bad mistakes.

ERROR MESSAGES: OUT OF BOUNDS ERROR

1) The byte value or the memory address (or both) is less than zero.

2) The memory address is greater than 65535.

SEE ALSO:
DISCUSSION: PERSONALIZING BASIC
DISCUSSION: FUNCTIONS (built-in: EXAM; user-functions)
DISCUSSION: MACHINE LANGUAGE SUBROUTINES
APPENDIX 1: SAMPLE PROGRAMS

STATEMENT: OUT <port number>, <byte value>

ACTION: The byte value is sent to the indicated 8080 or Z-80 output port. Both port number and byte values must be numeric expressions which evaluate to integers from 0 to 255.

EXAMPLES:
```
OUT 2,65
OUT P,B
OUT P7+1,ASC("0")
```

REMARKS: Both the port number and the byte value must be decimal (base 10) numbers. (Refer to STATEMENT: FILL for further elaboration on this.)

Frequently it is necessary to determine whether or not a given output port is ready to receive data, by examining a special input port (called a STATUS PORT) for evidence of a ready signal. (The built-in function INP may be used to facilitate this.) In such circumstances, a program should wait until the ready signal is given before executing an OUT statement. This process of waiting and OUTing is called "handshaking". If OUT is used before the signal is received, the byte value may be lost before arriving at its proper output destination. The OUT statement does not provide its own handshaking -- it is the programmer's responsibility to determine whether or not handshaking logic is necessary when communicating with a particular output port, and to implement it with the appropriate statements if so.

The PRINT and OUT statements do very different things and should not be confused with each other.

ERROR MESSAGES: OUT OF BOUNDS ERROR
One or both of the values specified lies outside the range of 0 to 255.

SEE ALSO: DISCUSSION: FUNCTIONS (built-in: INP)
STATEMENT: FILL

DISCUSSION: MACHINE LANGUAGE SUBROUTINES (CALL)

North Star BASIC provides a method through which you may "link" your BASIC programs to machine language subroutines which you have written to perform certain tasks.

A machine language routine must lie outside of the computer memory area reserved for the DOS, BASIC, and BASIC's program/data area. (You may restrict this area, and thus leave room for machine language routines in high memory, through use of the MEMSET command, for example.)

Machine language routines are accessed through the built-in BASIC function named CALL. CALL takes at least one argument, the numeric address in computer memory (an integer from 0 to 65535) where your machine language routine begins. An optional second argument, also a numeric expression in the above range, can be communicated to your routine in the D & E register pair. The value will be truncated to an integer if it has a fractional part. Negative arguments are not allowed. All registers may be used by your machine language routines -- BASIC will have already preserved any operating information which it will need later.

When your routine is finished, it should execute a RET (return) instruction, which will allow BASIC to resume control and continue with the execution of the BASIC program. If the machine language routine uses the stack, then it should use its own stack area. The stack area and stack pointer used by the BASIC interpreter should not be modified by the machine language routine. The number returned as CALL's function-value will be the decimal representation of the contents of the H & L register pair whenever the machine language routine terminates. Thus, it is possible to communicate a single numeric value to your routine from BASIC, and collect a single value from the routine when it returns.

Here are the models for proper formation of the CALL function-call:

```
CALL(<address expression>)
CALL(<address expression>, <argument expression>)
```

For an example of CALL in use, let's suppose there exists a machine language routine at address 60000, and that it will require the optional argument value.

The following line effects a transfer to that routine, passing the value of variable A as argument in the D & E registers as a positive, 16-bit binary integer:

```
10 Q=CALL(60000,A)
```

If, in this instance, the binary value of 578 is in the H & L register-pair when the machine language routine returns, then the variable Q will be set to 578 when BASIC resumes control.

Note that CALL looks like, and acts as a numeric function. CALL may be a part of any numeric expression in BASIC, and may be used anywhere any other numeric function might be used. Note that the following:

```
50 CALL(M,A)
```

is in error -- CALL is not a statement.

Below are some more examples of CALL in use. In one-argument instances of CALL, no specific argument value is sent to the machine language routine in the D & E register-pair, however, the CALL function always returns a value: whatever is in the H & L pair upon return to BASIC.

```
200 PRINT CALL(A(3)),A$
570 X=CALL(R+1024,G)
400 Q(CALL(43025,Y))=M
25 DEF FNM(G,D)=CALL(50000,G*256+D)
1030 F=CALL(S,ASC(S$))
```

Using machine language routines correctly is difficult and should only be attempted by expreienced programmers, and only then if no other alternative is available.

SEE ALSO: STATEMENT: FILL
STATEMENT: PRINT
DISCUSSION: MULTIPLE I/O DEVICES
DISCUSSION: FUNCTIONS

DISCUSSION: CHAINING (AUTOMATIC PROGRAM SEQUENCING)

Through use of the CHAIN statement (discussed in detail under STATEMENT: CHAIN), one program may cause another to be automatically LOADed and RUN, eliminating the need for the user to initiate and supervise such activities from the keyboard. Thus, a sequence of programs may operate virtually unattended for long periods (unless, of course, one or more of the programs requires interactive data-input or various diskettes need to be swapped in and out of the drives). There are two situations when CHAINing is most effectively used:

1) You desire to use several separate programs as a complete software "system" where each program can automatically transfer to another program whenever necessary.

2) A program may be too large to fit into the available program/data area, but can be broken up into separate, self-contained modules which CHAIN between themselves to accomplish the desired task.

COMMUNICATION BETWEEN CHAINED PROGRAMS

All variables are cleared by a successful CHAIN operation, so variables which are shared by one or more modules must be "restored" at the start of each module.

It is frequently necessary for a CHAINed program to accept information from the module which precedes it or pass data to the program to which it will CHAIN. Several methods may be used to accomplish program-to-program communication. The two most commonly-used ones are described below.

A data file may be shared between two programs, and thus provide for communication between them. This file might be a common data-base (of invoices, customer names, calendar items, switchboard messages, etc.), in which case each separate module would infer the action it should take by examining the current state of the file. Programs may use files to communicate in a more direct fashion if actual variables are shared between them: program A would WRITE the values of those variables into a file in a certain order, and then would CHAIN to program B, which would READ them back in the same order.

The second method for inter-program communication

involves storing the appropriate data in otherwise unused RAM memory, outside the program/data area, where it will survive the SCRatch which is implicit in a CHAIN. There are a good many techniques for utilizing RAM memory in this way -- most involve the use of the EXAM function and the FILL statement.

"TESTING THE WATER" FOR A SAFE CHAIN

If the file specified in a CHAIN statement does not exist, is not of type 2, or does not hold a valid BASIC program, the CHAIN operation will fail. It is not easily possible to check an alleged "program" stored on diskette to be certain that it is in perfect condition, but the built-in FILE function may be used to determine if a given program file exists and is of type 2 before an attempt is made to CHAIN to it. Use of the ERRSET statement may also help in such situations.

SEE ALSO:
STATEMENT: CHAIN
DISCUSSION: DATA FILES
STATEMENT: READ#
STATEMENT: WRITE#
STATEMENT: FILL
DISCUSSION: FUNCTIONS (built in: EXAM, FILE)
STATEMENT: ERRSET
DISCUSSION: ERROR TRAPPING AND RECOVERY

STATEMENT: CHAIN <program file name>

ACTION: The BASIC program contained in the specified file is automatically LOADed into the program/data area from diskette (replacing any current program), then automatically begins RUNning at the lowest numbered program line. The program file name must be a string expression which evaluates to a legal BASIC program (type 2) file name as described in DISCUSSION: DATA FILES.

EXAMPLES:
```
10 CHAIN "PROG,2"
100 CHAIN P$+D$
73 CHAIN "PROG"+N$(X,X)+",2"
```

REMARKS: CHAIN makes possible the automatic sequencing of 2 or more programs, freeing the operator from the task of having to LOAD and RUN each new program as the previous one ENDs. A CHAIN statement in program A, for example, may automatically initiate program B; a CHAIN in B may lead to C, and so on.

After a successful CHAIN, any previous program and data are cleared. All files currently open in the calling program are automatically CLOSEd. Communication between CHAINed programs may be facilitated by the use of common data files, or by use of EXAM and FILL.

Because CHAIN is a direct statement, it may be used instead of the LOAD-RUN sequence for manual program initiation. However, remember that the file name in a CHAIN statement is a string expression, and that string constants must always be enclosed by double quotes (e.g.: CHAIN "PROG" is legal, but CHAIN PROG is not).

ERROR MESSAGES: Same as COMMAND: LOAD

SEE ALSO: DISCUSSION: CHAINING
COMMAND: LOAD
COMMAND: RUN

DISCUSSION: ERROR TRAPPING AND RECOVERY

Normally, when a program error occurs while a BASIC program is RUNning, BASIC automatically terminates the execution of the program and issues an error message. This is to aid the programmer in finding and correcting the error. For many possible end-user applications, a BASIC program should operate in the presence of errors rather than terminate execution and print an error message. The program should detect the error condition, and then take corrective action without requiring the user to debug and re-execute the program. Certain kinds of errors resulting from incorrect input, improper diskette handling, or inconsistent data might be too difficult or time-consuming to anticipate and detect using regular BASIC statements.

To make convenient ERROR-RECOVERY UNDER PROGRAM CONTROL possible, North Star BASIC includes the special ERRSET statement. With this statement, the programmer specifies a line number which references the first statement of an ERROR-RECOVERY ROUTINE, which exists somewhere in the program. Once an ERRSET has specified the desired error-recovery routine, any program error which occurs during program execution will cause an immediate "GOTO" to that routine. (This is called TRAPPING THE ERROR.) The BASIC statements in the error-recovery routine determine the action to take under error conditions. A good routine will also include statements which attempt to correct the error condition. For example, if a user was told to insert a diskette into a drive, and then the computer detects a hard disk error when it attempts to open a file on the diskette, either the diskette has been inserted incorrectly, or the data on it is invalid. A good error-recovery routine might give the user a chance to re-insert the diskette.

The programmer must also specify two variable names in the ERRSET statement along with the line number of the start of the error-recovery routine, for example:

```
10 ERRSET 1000,L,E
```

When an error is trapped, the line number of the statement where the error occurred is assigned as the value of the first variable, and a numeric code, corresponding to the type of the error, is assigned to the second variable. By examining the value of these two variables, the program can determine not

only what caused the error-condition, but where in the program it occurred, and with this knowledge, decide what to do about the error. North Star BASIC program errors and their codes are listed in APPENDIX 2.

Note that if the error-handling routine in a program is written to make any decisions based on the number of the line in which the error occurs, it may be very unwise to RENumber the program.

When an error-trap occurs, any subroutines, user-functions, and FOR-NEXT loops which were active at the "trap" are still active. Thus, it is possible to execute a GOTO statement back to the point where the error occurred, or to the statement immediately after that point, and continue the execution of the program after the error-condition has been handled.

Error-trapping is disabled automatically after each "trap". After error recovery is complete, another ERRSET statement can be executed to resume error-trapping mode.

When the program no longer requires the use of BASIC's error-trapping feature, error-trapping can be disabled explicitly by executing the ERRSET statement with no arguments -- for example:

```
100 ERRSET
```

Unless the control-C program-interruption feature is disabled (as mentioned in DISCUSSION: CONTROL-C, and DISCUSSION: PERSONALIZING BASIC) a trappable "program error" will occur every time control-C is pressed while the program is RUNning in error-trapping mode. If you do not wish for control-C to be treated as an "error", then the control-C feature must be disabled.

SEE ALSO: STATEMENT: ERRSET
APPENDIX 1: SAMPLE PROGRAMS
APPENDIX 2: ERROR MESSAGES

STATEMENT: ERRSET <line number>, <numeric variable>, <numeric var.>
ERRSET

ACTION: Following the execution of an ERRSET statement which specifies a line number and two variables, the occurrence of a program error or a control-C (unless disabled) will cause an automatic GOTO to the specified line number. The line number where the error occurred is assigned to the first variable, and a numeric error code corresponding to the type of error is assigned to the second. This process is an ERROR-TRAP. After a trap, further traps are disabled until a subsequent ERRSET is executed. Execution of an ERRSET statement with no line number or variable specifications disables error-trapping.

EXAMPLES:
```
10 ERRSET 1000, L, E
20 ERRSET 570,E(0),E(1)
30 ERRSET
```

REMARKS: The use of ERRSET makes possible programs which always retain control even under error conditions. This is useful when writing software intended for use by persons who are unfamiliar with the North Star System or computers in general. Programs written for such users may effectively "take care of themselves".

After a trap has occurred or trapping has otherwise been disabled, another ERRSET statement must be executed to resume trapping mode.

When trapping is disabled, a program error causes immediate termination of the program, followed by an error message printed to the console.

ERRSET may not be used in direct mode -- error trapping does not function in direct mode. A program with error trapping enabled will retain that mode after a STOP interruption, but trapping will not resume until program execution CONTinues.

Not all errors are trappable with ERRSET. Refer to APPENDIX 2. Those errors without error codes are not trapped. Note that it is possible to trap the action of the control-C panic button as an "error". In trapping mode, control-C will always cause a trappable "error" unless the panic-button feature has been disabled (a process described in DISCUSSION: PERSONALIZING BASIC).

The subroutine, function, and FOR-NEXT calling histories of a program remain intact after an error-

trap occurs, providing the programmer with a chance to recover from the error, if possible.

ERROR MESSAGES: Same as STATEMENT: GOTO.

SEE ALSO: DISCUSSION: ERROR TRAPPING
APPENDIX 2: ERROR MESSAGES
DISCUSSION: PERSONALIZING BASIC
DISCUSSION: CONTROL-C, THE PANIC BUTTON

DISCUSSION: THE LINE EDITOR

INTRODUCTION TO THE EDITOR

Anyone who has used the North Star BASIC system for any length of time is already aware of the "delete-character" function performed by the underline, RUB/DEL, and backspace keys, as well as the "cancel-line" function of the at-sign (@) key. These are two features of the larger LINE EDITOR, which allows you to modify, quickly and efficiently, lines of information which you type into North Star BASIC. Mostly, people use the line editor to change or correct program text, a line at a time. However, the editor may also be used on commands and responses to INPUT or INPUT1 statements. Because the program-development aspect of the editor is by far the most important to the average BASIC user, this purpose will be emphasized here.

The character-delete and line-cancel functions of the editor permit instantaneous correction of typing errors as they are made during the entry of a line. The editor also allows the correction and modification of program lines which have already been typed into the system. For example, after SCRatching the program/data area, type the following PRINT statement into BASIC:

```
10 PRINT "TOTAL RECEIPTS TO DATE: ",T1
```

As soon as you strike the RETURN, and this line becomes part of your current program, pretend that you have made a mistake: the variable to be printed should actually be T2, not T1. In BASICs without a line editor facility, you would be forced to retype the entire line in order to correct the one erroneous character. However, North Star BASIC always "remembers" the last line you type to it. This, for discussion purposes, will be called the OLD LINE. As a rule of thumb, whenever you strike the RETURN to terminate a line of input to BASIC, that line immediately becomes the old line. (There is one exception to this rule, which will be discussed in a moment.) Utilizing the higher functions of the line editor, you can convert the old line into a correct NEW LINE which will then replace its predecessor in the program. For now, to prove to yourself that BASIC indeed "remembers" the old line, type control-G. Notice that the line you just typed reappears. The cursor or print-head on your terminal will sit just at the end of the line. By striking control-G

before typing anything else, you have instructed the line editor to take the old line from the beginning to the end, and treat it as a new line of input, copying the line to the terminal as it does so. In effect, by using just one control-character, you have "retyped" the old line. If you now strike RETURN, the new line will replace line 10 -- but since the new line is identical to the old, no net improvement will result: T1 should still be changed to T2. However, suppose you strike the underline key. Now, the last character in the new line (the 1 that should be a 2) is erased, and you may type the correct one. If you strike RETURN at this point, the correct line will replace its faulty predecessor. To correct the reasonably long line 10, all that was required was to strike four keys: control-G, underline, the "2" key, and RETURN.

(When one is used to such a procedure, it is much faster and less tedious than retyping the whole line, although, for this introductory example, you probably spent more time being careful, reading directions, and observing results, than you would if you had just retyped the whole thing to start with. Practice with the editor -- your speed will improve tremendously. Even after just an hour or so of experience with the editor, you will note a gratifying increase in your efficiency when entering and modifying BASIC programs.)

Now, try another example. Realize that, as soon as you strike the RETURN key to end the new line, it became the old line, and you may now use the editor on it. Type

```
20
```

(and don't strike RETURN!). Now strike control-G. You should see the following on your terminal.

```
20 PRINT "TOTAL RECEIPTS TO DATE: ",T2
```

If you strike RETURN, a new line 20 will be added to your current program. Its contents will be identical to the contents of line 10. What you have done is create a completely new line by combining newly-typed information with part of the old line. When you typed the line number 20, you were typing over the first two characters of the old line. When you pressed control-G, the line editor knew to copy only the remaining part of the old line to the new line. The first two characters of the old line were

discarded in favor of your new information. Suppose that there had been no third character in the old line -- that it was only, say, one or two characters long itself. Then, there would have been nothing for the control-G function to copy to the new line. In this case, as in others where the editor cannot comply with your wishes, it rings the bell (or beeps the beeper) on your terminal.

THE EDIT COMMAND

So far, all that has been shown is only how the most recently typed line may be modified or used to create a new line. What if, after typing line 20 in the example above, you want to go back and modify line 10 again? This time, line 20 would be the old line -- not line 10. The editor would still want to work with line 20. To surmount this problem, you can force BASIC to treat line 10 as the old line, by using the EDIT command as follows:

```
EDIT 10
```

This forces the line editor to replace the "natural" (most recently typed) old line with the program line you specify. In this example, line 10 would become the old line. (Note that, if you type in other commands besides EDIT, the command line itself becomes the old line. The EDIT command, however, is the one exception to the "rule of thumb" mentioned earlier. When you strike RETURN after typing the EDIT command, the command line is discarded, and the program line specified becomes the old line instead.)

Notice that there is no obvious response to the EDIT command -- the cursor or print-head simply moves to the start of the next line. However, if you strike control-G, you will see that line 10 has indeed become the old line, since it is immediately printed on the terminal. Using the EDIT command, you can force any program line to be the old line, and thus you can modify any part of your program, or create totally new lines by taking information from a "forced" old line, and combining it, under a new line number, with newly-typed information. The following discusses all the special functions of the line editor, as well as some theory behind the editor's operation.

LINE EDITOR SPECIFICS AND FUNCTIONS

Assume that you have just strike RETURN to enter the

above line 10 into your program. Line 10 is now the old line. BASIC is waiting for you to type (or use editor commands to help form) a new line. At this stage, the old line is stored in BASIC's memory, and two "pointers" are kept: one to the current character position in the old line (the OL pointer), and the other to the current character position in the new line being typed (the NL pointer). Before you start typing the new line, both these pointers are set at the starts of their respective lines. (It is obvious that the new line pointer is set to the start of the new line, since you haven't typed anything new yet!) Most of the editor functions are most completely explained with reference to these dual pointers.

Typing a normal character (not a control-character editing command) in the absence of any other editing function will result in both pointers being advanced one position. The typed character is added to the new line, and the old line pointer now points to the next character in the old line. In the sequence above, for example, when you typed 20 to start the new program line, the NL pointer ended up pointing just beyond the 0 in 20, while the OL pointer was skipped past the 10 in the old line, and pointed at the space just beyond the line number.

```
BEFORE:
     (old line)  10 PRINT etc....
                 ↑  OL pointer

     (new line)
                 ↑  NL pointer -- next char typed goes here

AFTER:
     (old line)  10 PRINT etc....
                   ↑  OL pointer

     (new line)  20
                   ↑ NL pointer -- next char goes here
```

Here are the editing functions, along with the control-character commands which invoke them:

Control-G: COPY REST OF OLD LINE TO END OF NEW LINE
Copy all the characters from the OL pointer character position through the end of the old line over to the new line, starting at the NL pointer character position. If the OL pointer already points past the end of the old line, no characters will be copied, and the bell will ring.

Control-A: COPY ONE CHARACTER FROM OLD LINE

The character in the old line pointed to by the OL pointer is copied to the new line at the character position designated by the NL pointer. As a result, both pointers will be advanced by one position. If there is no character to copy, the bell rings. Repeated use of the control-A command will eventually give the same result as one control-G command.

Control-Q: BACK-UP ONE CHARACTER

This erases the last character of the new line, and decrements both the OL and NL pointers by one. If either pointer is already pointing to the beginning of its line, the bell is rung. An underline is printed on the terminal to denote the erasure of a single character. Typing the underline, DEL/RUB, or backspace (control-H) keys will also give the same result as control-Q.

Control-Z: ERASE ONE CHARACTER FROM OLD LINE

This command advances the OL pointer by one position, without copying anything to the new line or advancing the NL pointer. This effectively "erases" the skipped character from the old line so that it cannot be copied to the new line. A per cent sign (%) is printed to the terminal to indicate the action of this command. If the OL pointer is already at the end of the old line, then the command is rejected and the bell is rung.

Control-D: COPY UP TO SPECIFIED CHARACTER

A second character (called the SEARCH CHARACTER) must be typed before this command is executed. The result is that the contents of the old line from the current OL pointer position will be copied to the new line (starting at the NL pointer position) up to (but not including) the first old-line occurrence of the search character. If the search character cannot be found in the old line, no characters are copied to the new line, and the bell is rung. For example, try typing

```
10 PRINT "HERE IS A TEST LINE"
```

to BASIC, striking RETURN afterwards so that it becomes the old line. Now, strike control-D and then capital-S. Notice that neither the control character nor the letter S appear on the terminal, but the following is seen instead:

```
10 PRINT "HERE I
```

The old line has been copied to the new line up to (but not including) the first instance of capital-S in the old line. (To copy over the rest of the line, of course, use control-G.)

Control-Y: SWITCH SPECIAL INSERT MODE ON AND OFF.
If insert mode is on, control-Y will turn it off, and if it is off, the same command will turn it on. Insert mode starts out by being "off" at the beginning of every new line. When insert mode is off, typing normal (non-control) characters advances the OL as well as the NL pointer (so that the new material may "type over the old line"). When insert mode is on, however, typing normal characters will not advance the OL pointer (although the NL pointer is necessarily advanced). The result of all this is that insert mode may be used to insert some new material in the middle of the old line. (An example will be given in a moment.) When insert mode goes on, a left angle-bracket (<) appears on the terminal. When it goes off, a right angle-bracket (>) is printed. (Note that these characters do not become part of the new line itself -- they are printed on the terminal only to signal to you the current status of insert mode.) While normal typing will not advance the OL pointer during insert mode, editing commands which are supposed to change the value of the OL pointer will continue to do so. For example, typing control-G during insert mode will still copy the rest of the old line over to the new line and advance the OL pointer to the end of the old line. To get the feel of insert mode, and the on-off action of control-Y, set up an old line by typing the following:

```
10 PRINT "TEST LINE"
```

Now, use the control-D command twice, to "speed" you to a point just after the quote-mark at the beginning of the string literal. (To accomplish this, strike four keys: control-D, T, control-D again, and T again.) Here is what you should see on the terminal:

```
10 PRINT "
```

Now, strike control-Y, which gives you this:

```
10 PRINT "<
```

Type the words

```
HERE IS A
```

and then a space. Then strike control-Y again. The terminal should now look like:

```
10 PRINT "<HERE IS A >
```

By going into insert mode temporarily, you avoided typing over and so obliterating any part of the old line. So, if you now strike control-G, everything which came after the first quote in the old line will be copied to the new line:

```
10 PRINT "<HERE IS A >TEST LINE"
```

If you strike RETURN at this point, the new line 10 will replace the old, and the net effect will be that the new material will have been inserted between the first quote-mark and the subsequent T of the old. To see this net effect, strike control-G again and follow it with a RETURN.

Control-N: CANCEL AND RE-EDIT NEW LINE

This command cancels the partially-completed new line and permits another new-line to be entered. The canceled new line becomes the old line for subsequent editing. An at-sign (@) is printed and advancement to the next terminal line occurs when this command is typed. The at-sign itself may be typed instead of control-N to achieve the same results. After the cancel is executed, both OL and NL pointers are reset to the start of their respective lines.

SEE ALSO: DISCUSSION: COMMUNICATING WITH BASIC

DISCUSSION: COMPATIBILITY WITH OTHER BASICS

This section provides some information which may be useful to you if you are attempting to convert programs into North Star BASIC from other versions of BASIC.

STRING HANDLING

The operations and functions used to access strings and substrings often differ widely between different versions of the BASIC language. DISCUSSION: USING STRINGS details the system implemented in North Star BASIC, where substring access is achieved through string-name subscripting. However, some BASIC systems use the so-called "MID-LEFT-RIGHT" convention, where access to substrings is made possible by the three built-in string functions MID$, LEFT$, and RIGHT$. Programs which use this method of substring access will have to be modified to reflect North Star string conventions. In general:

OTHER BASICS		NORTH STAR BASIC
LEFT$(X$,L)	is the same as	X$(1,L)
RIGHT$(X$,R)	is the same as	X$(LEN(X$)-R+1)
MID$(X$,L,N)	is the same as	X$(L,L+N-1)

STRING TABLES

Some versions of BASIC implement arrays of strings with the syntax which is used for substring referencing in North Star BASIC. An array of strings may be achieved in North Star BASIC by partitioning a string variable into fixed-length substrings. For example, an array of N strings, each of maximum length L would be DIMensioned as:

```
10 DIM A$(N*L)
```

and the Jth string element (where J extends from 0 to N-1) would be accessed using:

```
A$(J*L+1,(J+1)*L)
```

STRING DECLARATIONS

In North Star BASIC, all strings longer than 10 characters must be explicitly declared in a program's DIMension statements. Strings may be dimensioned to any length desired, to the limit of available computer memory. Some other BASICs do not require

that string variables be dimensioned before use, but may set a small upper limit on the maximum length of strings which may be used in a program.

INPUT TRANSLATION

Certain characters, when they are typed into North Star BASIC, are automatically translated into other characters. This is done to help minimize the effort of converting programs written for other BASIC systems into North Star BASIC. This conversion is not performed upon text within quoted strings. The following chart summarizes the translation process.

[	becomes	(
]	becomes	)
: (colon)	becomes	\ (backslash)
; (semi-colon)	becomes	, (comma)

Thus, the line input as

```
10 PRINT A$[3,4]; : LET A$[3,4]="HI"
```

becomes

```
10 PRINT A$(3,4), \ LET A$(3,4)="HI"
```

NORTH STAR'S BCD ARITHMETIC

North Star BASIC uses the BCD (binary coded decimal) system for implementing floating-point arithmetic (as opposed to binary integer arithmetic in some BASICs, and straight binary floating methods in others.)

Within the limits of its precision (8-digits in the standard version), North Star BASIC's BCD method is the most accurate method of arithmetic computation available on microcomputers today. Other floating-point arithmetic methods exhibit "binary-conversion-error" which introduces strange and sometimes frustrating inaccuracies into numeric computations because of an internal conversion of numbers from decimal (base 10) to binary (base 2).

It is impossible, using straight binary methods, to represent with complete accuracy many common and precise decimal fractions, such as .1! You might assume that 10*.1 = 1. Using North Star's accurate BCD arithmetic, it always does. However, under other methods, 10*.1 frequently does not equal exactly 1!

IF ... THEN EVALUATION

Other BASICs handle the results of IF ... THEN evaluation differently than North Star BASIC when the IF statement precedes others on a multiple-statement program line. In North Star BASIC, when the IF condition is FALSE, the THEN part is skipped and execution continues with the following statement in the program text. The "following" statement may come after the IF statement on the same program line, or, when the IF is at the end of a program line, the first statement on the next line is used as the "following" statement. Thus, the program:

```
10 A=0
20 B=0
30 IF A<>0 THEN A=7 \ B=7
40 PRINT B
```

will yield

```
7
```

as output. In contrast, other BASICs may ignore the rest of line 20 when the IF condition is found to be FALSE, and will skip ahead to the following program line, bypassing the assignment to B in line 30 so that the output becomes:

```
0
```

With these other BASICs, execution always skips to the following program line when the condition in an IF statement is FALSE. The remainder of the line, if any, is executed only when the condition evaluates to TRUE.

DISCUSSION: SPECIAL ENTRY POINTS

NOTE: The following discussion concerns advanced topics and presupposes a working knowledge of the North Star DOS and a grasp of memory addressing in hexadecimal (base 16) notation. Please be sure that you are familiar with these topics before reading further in this section.

The following is a list of BASIC's entry points, and the results of re-entry to BASIC via each. (The abbreviation ORG stands for the starting address of your BASIC -- for those whose BASIC starts at 2D00H, the actual entry point addresses are given in parentheses next to the general models.)

ORG + 00H (2D00H)
BASIC is initialized. An automatic SCRatch of the program/data area is performed, erasing any BASIC program and/or data which might have existed in that area of RAM. Note that this is the entry point used by the GO BASIC command in the DOS.

ORG + 04H (2D04H)
Any previously existing program is retained, but any variables and/or other data associated with it are erased.

ORG + 14H (2D14H)
The BASIC system resumes, with all program, data, and program execution history left intact. Thus, you may interrupt a BASIC program with control-C, exit BASIC with BYE, use the DOS, re-enter BASIC at ORG + 14H, and use the CONT command to resume BASIC program execution exactly where it left off. (This assumes, of course, that your use of the DOS causes no change in BASIC's memory region.)

SEE ALSO: DISCUSSION: PERSONALIZING BASIC

DISCUSSION: PERSONALIZING BASIC

You may change certain of BASIC's internal features so that system operation is more convenient for you and/or better fits your particular computer's capabilities. For example, the limits of the memory area used by BASIC may be enlarged or constricted, leaving more or less space for user programs and data. These changes are accomplished through the modification of information stored in various memory locations within the BASIC interpreter itself.

In general, modifications of these "personalization bytes" are best handled through use of BASIC's FILL statement, and, occasionally, the built-in EXAM function. What follows is a complete, step-by-step procedure which you may use to "personalize" BASIC in your computer system. If you want the changes made to be permanent, be sure to follow ALL of the steps (from A to E). If you want only temporary modification, which will endure until the end of the current session of BASIC, then do only step C, omitting all the rest.

A. Test your system's memory by using the MONITOR memory-test function to be sure that you will not be making a copy of BASIC from bad memory. In particular, the area where BASIC and DOS reside should be tested thoroughly.

B. At this point, you should make sure that the DOS is operational, and that you are in its COMMAND mode (signified by the DOS prompt). Now, put your write-protected, original system software diskette (supplied by North Star) in drive #1 and then type

   GO BASIC<CR>

   When BASIC responds with READY, go to step C.

C. Now you are ready to make the various modifications to BASIC. In order to do so, follow the sub-steps here in exactly the order given. If you do not wish to make one or more of the individual changes listed, then simply skip it, but DON'T MIX UP THE ORDER OF THE STEPS! In any case, you must always do step #2 before attempting any higher-numbered steps.

   1. MEMORY SIZE
      Initially, the standard version of BASIC doesn't leave much room for your BASIC

program/data area -- BASIC is made to "assume" that you have only 16,384 bytes of working memory. The DOS and BASIC itself take up most of this. In order for you to write and RUN reasonably large programs, you must have more memory beyond the 16,384-byte (16K) limit. Moreover, you must inform BASIC of the extra memory availability using the MEMSET command. See COMMAND: MEMSET for detailed information on the use of this command. You may use MEMSET to enlarge or shrink the program/data area that BASIC is allowed to use. Simply determine the address (in decimal) of the highest memory-cell you want BASIC to be able to use, and employ that number as the argument to the MEMSET command. For example, if your memory extends all the way to 48K (49151 in decimal) and you want BASIC to use all that's available there, type

```
MEMSET 49151
```

(The argument to MEMSET is, among other things, translated to binary, and put into bytes ORG+Ø9H and ORG+1ØH, where "ORG" is BASIC's ORIGIN (starting address) in your system, ususally 2DØØH. In the standard version of BASIC, then, these addresses are 2Ø9H and 2DØAH, respectively. The standard default high-address for the program/data area is 5FFFH.)

2. SETTING A VARIABLE TO BASIC'S ORIGIN
   For many of the following steps, the FILL statement is used to modify memory locations within BASIC. In the examples to be given here, it will be assumed that the numeric variable S has been set to the decimal number corresponding to the address in memory where your copy of BASIC starts. If you have a version of BASIC which starts at "2DØØ" in hexadecimal, then use 11520 for BASIC's origin. Otherwise, if your BASIC starts somewhere else, determine the decimal (base 10) equivalent of the origin, and use that number. Set S in a direct-mode assignment statement. For example, for standard versions of BASIC, type

   ```
   S=11520
   ```

3. LINE LENGTH
   See STATEMENT: LINE for a description of the

significance of the input/output line length in BASIC. The standard version assumes that the console terminal has a line-width of 80 characters. If the actual per-line capacity of your terminal is smaller or larger than this, set variable L to the appropriate line length for your terminal. If that is 64, for instance, then type

```
L=64
```

Once L is set, then type

```
FILL S+14, L
```

4. VIDEO PAGING
If you have a video (CRT) terminal, it is desirable for BASIC to send only one "screen-page" at a time when providing a program LISTing to you on the video screen, and then wait for you to "ask" for the next page. If you have a printing terminal, which gives you output on paper, you won't need paging. Set variable P to the appropriate value for your terminal. For hardcopy (printing) terminals, where you don't want paging, type

```
P=0
```

and for video screens, set P to the number of lines which your screen can display at one time. The standard version of BASIC assumes that your terminal has a video screen capable of showing 24 lines at a time. If this is so, then you don't need to make any modification at all, and may skip this step. Otherwise, once the appropriate value of P is set, type

```
FILL S+19, P
```

(Note that, if you direct BASIC to page its LISTings, it will give you P-1 lines of program, then, at the bottom of the screen, at the Pth line, it will PRINT

```
PRESS RETURN TO CONTINUE
```

To get another page of LISTing, strike the RETURN key. If you'd like to terminate the LISTing at this point, press control-C.)

5. "BACKSPACE" CHARACTER
   In the standard, unmodified version of BASIC, when you press the underline, control-Q, backspace (control-H), or RUB/DEL key to delete the last character typed, BASIC types an underline (ASCII character 95) back at you to confirm the deletion. It is possible to change this "deletion confirmation" character to any other one you wish. Set variable D to the decimal ASCII value of the desired character. (If you don't know its ASCII value, use the table provided in APPENDIX 4.) For example, the ASCII value of the backspace character is 8, so to set D appropriately, type

   ```
   D=8
   ```

   Then, having set D, type

   ```
   FILL S+23, D
   ```

   Changing the "deletion confirmation" character to backspace is most useful when your terminal is a standard CRT model. However, not all use ASCII-8 as a backspace -- consult the manual for your specific terminal or video screen in order to get the exact character which causes backspacing on it.

6. CONTROL-C INHIBIT
   For some applications, you may wish to keep the user from being able to interrupt a program by striking (whether accidentally or on purpose) the control-C "PANIC BUTTON". If, for any reason you wish to disable the control-C feature, make sure that S is set to the starting address of your BASIC and type

   ```
   FILL S+24, 1
   ```

   To re-enable the feature, type

   ```
   FILL S+24, 0
   ```

   The standard copy of BASIC assumes that control-C interruptions are allowed. Note that control-C can be turned on and off during the execution of a program, if desired, using these same methods.

7. NON-STANDARD BOOTSTRAP PROM
   If your system uses a non-standard bootstrap disk-controller PROM, then you must convert the first two digits of the 4-digit hexadecimal address for your special PROM into decimal, then assign that value to a variable, say B. For example, if your PROM starts at FC00H, you would take the two-digit hex number FC and convert it to its decimal equivalent, 252. (You may use the table in APPENDIX 4 for this conversion.) Then, you would type

   ```
   B=252
   ```

   Once B has been set properly, type

   ```
   FILL S+16, B
   ```

   Note that if you have a non-standard PROM and fail to make this modification, the RND function will not work properly when given a negative argument.

8. SHRINKING BASIC
   There are many applications which do not use the special mathematical functions SIN, COS, ATN, LOG, and EXP, but do require as much free memory as they can get! To release extra memory into the program/data area, you can "chop" these functions out of BASIC by performing the modification described here. First, as you look at the table below, realize that these functions must be removed starting at ATN and continuing up through the function you select (which might itself be ATN, meaning a deletion of only one function). It is impossible, for instance, to remove the LOG function but keep SIN and COS. If you choose to remove through LOG, then SIN, COS, and ATN will also be erased. Bearing this in mind, you can indicate your choice by setting variable C to a specific value, as shown in this table:

to remove functions from ATN through ...	set C to
ATN	1
SIN-COS	2
LOG	3
EXP	4

   To illustrate, suppose you wish to eradicate

all of the listed functions. Then you should type

```
C=4
```

When C is set to the desired value, then type

```
FILL S+6, EXAM(S+24+(C*2)-1)
FILL S+7, EXAM(S+24+(C*2))
```

Note that, after this modification has been made, any attempt to use the erased functions will lead to a system crash. (The exponentiation operator, ↑, makes frequent use of the EXP function, so if you delete EXP, don't use ↑, either.)

9. PERSONALIZING FPB-BASIC FOR DIFFERENT FLOATING POINT BOARD ADDRESSES

Note: Skip this section unless you are personalizing a version of FPB-BASIC.

The North Star Hardware Floating Point Board (FPB-A) is accessed like computer memory, and has a set of addresses as does a memory board. All the FPB-A addresses have the same high byte: 239 (EFH) for the standard board. The North Star FPB-A manual tells how to change the high byte, in order to re-address the board. If you find it necessary to re-address your FPB-A, you will also have to personalize BASIC so that it will use the board at the new set of addresses. The following procedure should be done BEFORE you actually change the addresses of the board itself:

Simply determine what the decimal equivalent of the board's new high byte is, and set variable F to it. (You may find APPENDIX 4 useful in performing any necessary conversion from hexadecimal to decimal.) To illustrate, assume you wish to change the high byte from 239 (EFH) to 223 (DFH). Then type

```
F=223
```

When F has been assigned the decimal value of the board's new high byte, type

```
FILL S+33, F
```

Now, having finished all personalization, use the methods described in this DISCUSSION to save a copy of your new FPB-BASIC on diskette. Shut down the computer system and change the board's addresses. When you re-activate and re-boot the system, execute the new copy of FPB-BASIC. From now on, every time this new copy of FPB-BASIC is executed, it will "re-personalize" itself to use the FPB-A board at the new address. Older copies of FPB-BASIC, which have not been modified in the above fashion, will fail to work with the re-addressed FPB-A.

D. Type BYE in order to return to the DOS. Mount an initialized diskette, for example, a diskette which contains only a personalized copy of DOS (not your original, write-protected diskette), in drive #1, and perform the following DOS commands:

```
CR BASIC <size of BASIC file on master disk>
TY BASIC 1 <origin in hex of your BASIC>
SF BASIC <origin in hex of BASIC>
```

If you have the standard version of BASIC, then the above simplifies to the following actual commands:

```
CR BASIC 52
TY BASIC 1 2D00
SF BASIC 2D00
```

E. Now type

```
GO BASIC
```

to test your personalized copy and make sure that all the modifications have been made correctly. If not, get back into DOS and return to step A. The new copy of BASIC may now be used as your "personalized" master copy, and the disk containing it should be write-protected for this reason. Then, when you need another copy of this personalized BASIC, you need only copy it to another diskette.

TURNKEY STARTUP OF BASIC

Using methods similar to the personalization process above, you can configure a copy of BASIC so that a BASIC program begins automatically as soon as BASIC

itself is "up and running". This is especially desirable when you want to create an "automatic" software system intended for use by persons who are unfamiliar with BASIC or DOS operation.

HOW TO CREATE A TURNKEY VERSION OF BASIC:

1) Mount a diskette with a copy of BASIC on it in drive #1. Type

   GO BASIC

2) If you desire different "personalization" than that already existing in this copy of BASIC, go through the personalization procedure described in step C (above).

3) Enter or LOAD the desired BASIC program into the system.

4) Repeat substep 2 of Personalization Step C to set S to the starting address of BASIC.

5) Type

   PSIZE

   Add the number printed to the size of the BASIC interpreter itself. (This is the filesize of BASIC as listed in the diskette directory. Assume 50 for now.) Set variable X to the number you get. If the PSIZE is 20, for example, add 50 to get 70, then type

   X=70

6) Set string variable F$ to the name you wish to give to this turnkey system. If, for instance, the "automatic" BASIC program is named "SALES", then you might want to call the turnkey system "SALESBAS". Then, type

   F$="SALESBAS"

   and go on.

7) Mount a diskette with enough room on it to hold a file of size X in drive #1. Then type

   CREATE F$,X

8) Type

FILL S+15, 0

and finally type BYE, which will put you back in the DOS. In the DOS, you will need to save the turnkey system on the file you have created, and must specify BASIC's starting address by using the TY (TYpe) command. Here are two models for what you must type in the DOS:

SF <name of file> <BASIC's origin>
TY <name of file> 1 <BASIC's origin>

If "SALESBAS" is taken as an example, you might type

SF SALESBAS 2D00
TY SALESBAS 1 2D00

9) Now type

GO <file name>

to test the new version of BASIC. In the example here, you would type:

GO SALESBAS

Your BASIC program should start up without the need for a LOAD, RUN, or CHAIN.

A CHART FOR READY-REFERENCE

The following chart contains summary information about each of the "personalization bytes" discussed in this section. The addresses are given relative to the start of BASIC (the "ORG +" form), and, for those whose BASIC starts at 11520 (2D00H), the actual addresses in decimal and hex are also given.

ORG+6 & ORG+7 (11526 & 11527 or 2D06 & 2D07) [ENDBAS]
These two locations contain the low and high bytes, respectively, of the last address taken up by the BASIC interpreter itself, and may be modified to contain a lower address in order to "shrink" BASIC.

ORG+9 & ORG+10 (11529-11530 or 2D09H-2D0AH) [HIGHMEM]
Contains lower and upper bytes, respectively, of highest address in RAM which BASIC may use for program/data area. Standard value: 255 and 95 respectively (corresponding to 5FFFH).

ORG+14 (11534 or 2D0EH) [LINE]
Initial line length. Standard value: 80

ORG+15 (11535 or 2D0FH) [AUTOS]
Controls turnkey auto-start. Zero-byte means auto-start engaged. Standard value: 1 (no turnkey operation).

ORG+16 (11536 or 2D10H) [BOOTPROM]
Corresponds to first two hex-digits in bootstrap PROM address for your system. Standard value: 224 (E8H)

ORG+19 (11539 or 2D13H) [PAGES]
Controls paging-mode for program LISTings. If paging is desired, this should contain the number of lines in a terminal "page". A zero-value means no paging will occur. Standard value: 24

ORG+23 (11543 or 2D17H) [DELECHO]
Character to be "echoed" in response to a single-character deletion. Standard value: 95 (corresponds to underline character).

ORG+24 (11544 or 2D18H) [PANICOK]
Controls use of control-C for BASIC program interruption. If this byte is 0, control-C causes interruptions. When the value is non-zero, control-C interruptions are disabled. Standard value: 0

ORG+33 (11553 or 2D21H) [FPBADDR]
Specifies the high-order byte of the floating point board addresses. This byte is present only in hardware floating point versions of BASIC.

DISCUSSION: NON-STANDARD VERSIONS OF BASIC

NOTE: This discussion assumes some sophistication on the part of the reader, particularly an understanding of the term "precision" and how it relates to numbers and arithmetic in BASIC. A knowledge of computer memory addressing and the hexadecimal numbering system is also helpful. Readers unfamiliar with these topics should study other sections in this manual, namely DISCUSSION: USING NUMBERS and APPENDIX 4: DECIMAL-HEXADECIMAL-BINARY-ASCII CONVERSION TABLE.

ABOUT NON-STANDARD VERSIONS OF BASIC

The standard version of BASIC begins at address 11520 (2D00H) in memory, provides 8 digits of arithmetic precision in its representation of numbers, and does arithmetic with the help of special software routines written directly into the BASIC interpreter itself. BASIC is available, however, beginning at other addresses in memory. (From now on, the starting address of your copy of BASIC, whatever it is, will be called its ORG, for "origin".) Moreover, BASIC is available with 6, 10, 12, and 14 digits of numeric precision, as well as the standard 8 digits. North Star manufactures a Hardware Floating Point Board which will perform arithmetic with any of the above precisions far faster than equivalent microcomputer software routines. A version of BASIC is available which is designed to use the power of this board, and which, as a result, does not include the same arithmetic routines found in standard BASIC, since their functions are duplicated more efficiently in the circuitry of the board itself.

Any combination of these three options (different origin, different precision, and FPB arithmetic) may be ordered in a special, NON-STANDARD version of BASIC for a nominal fee. This section discusses the explicit details and the ramifications of the differences between these special BASICs and the standard BASIC.

DIFFERENT ORIGIN

BASIC may be "re-located" to begin at any of the sixty-four 1024-byte address boundaries in memory. It is, of course, advisable to avoid certain areas of memory, most notably those which contain the DOS and the bootstrap PROM. If you have any other system software (such as special I/O routines in PROM, etc.)

which must exist in a certain region of memory, you should also avoid re-locating BASIC into these areas as you avoid the DOS and North Star PROM regions.

DIFFERENT PRECISIONS

Within RAM and in diskette data files, numeric elements of differing precision will take different amounts of storage space. Standard 8 digit numbers require 5 bytes, for example, while 14 digit numbers require 8 bytes.

Because of this size difference between numbers of different precisions, it is not possible for a BASIC program which is operating under a BASIC of precision X to READ numeric elements from data files created under a BASIC of precision Y using the READ# statement in normal fashion. That is,

```
READ #1,A
```

under 8 digit BASIC will not return a correct value if used to retrieve a numeric element created under 14 digit BASIC. It is possible to read "foreign" files such as these by accepting data byte-by-byte and reconstructing appropriate values, making allowances for difference in precisions.

FLOATING POINT BOARD (FPB) BASICS

Versions of BASIC which use the North Star FPB to perform arithmetic typically operate much faster than those which use software to do the same calculations. Moreover, FPB BASICs are somewhat smaller than software-arithmetic versions. Depending upon the precision required, an FPB BASIC is approximately 750 bytes smaller than the corresponding version which does arithmetic with software. Except for the increased speed of computation which is realized with Hardware Floating Point versions of BASIC, there is no operational difference between FPB and non-FPB BASICs. In particular, BASIC programs written under an FPB system will run without modification (although more slowly) on a non-FPB system, as long as the numeric precisions are the same, and other considerations are equal. However, FPB BASIC interpreters themselves will not operate correctly in computers which do not include the Floating Point Board.

SEE ALSO: APPENDIX 3: IMPLEMENTATION NOTES

APPENDIX 1

The following are sample programs written in North Star BASIC. Each has been fully tested and thoroughly debugged, and is guaranteed to run on any version of North Star BASIC, Release 4 or later, which has at least 8-digit precision and has not been stripped of trigonometric and exponential functions.

```
100 REM
110 REM PRINT a sine wave vertically on the page
111 REM
115 FOR J=1 TO 100 STEP .1
120 T=SIN(J)
130 S=INT(30*T)
140 PRINT TAB(30+S),"*"
150 NEXT
160 END
```

```
100 REM Input a string and check that it is a legal integer.
105 REM
110 DIM A$(72)
115 PRINT \ INPUT "TYPE AN INTEGER: ",A$
120 IF LEN(A$)=0 OR LEN(A$) > 8 THEN GOTO 500
130 FOR J=1 TO LEN(A$)
140 IF A$(J,J) < "0" THEN 500
145 IF A$(J,J) > "9" THEN 500
150 NEXT J
155 PRINT "THE INTEGER IS OK:",VAL(A$)
160 GOTO 115
500 REM Case not ok
510 PRINT "NOT A POSITIVE INTEGER WITH AT LEAST ONE"
515 PRINT "DIGIT AND NO MORE THAN 8 DIGITS.  TRY AGAIN."
520 GOTO 115
```

```
100 REM
110 REM    Print a table of formatted values.
120 REM
130 FOR J=1 TO 100
140 PRINT %3I,J,
150 PRINT %6F3,SIN(J),%7F4,COS(J),
160 PRINT %10E3,EXP(J),
170 PRINT %12F10,RND(0)
180 NEXT
```

```
100 REM   Construct a file containing numeric squares,
110 REM   and then use random access to compute squares
120 REM   of typed input values.
125 REM   Program assumes file "SQTABLE" exists and will
126 REM   fail if it doesn't.
127 REM   Both sequential and random access are used here.
130 OPEN #0,"SQTABLE"
140 FOR J=0 TO 500
150 WRITE #0, J↑2
160 NEXT
170 INPUT "X=",X
180 IF X<0 OR X>500 OR X<>INT(X) THEN END
190 READ #0%5*X,X2 \ REM Each number takes 5 bytes in file.
200 PRINT "X SQUARED:",X2
220 GOTO 170
```

```
 10 REM Various utility functions which may be handy
 20 REM in writing programs.
 30 REM
300 DEF FNC1(X) \ REM Returns ARCCOS(X) in radians.
305 REM X must lie in range -1 ... 1
310 IF X=-1 THEN RETURN 3.1415926 \ REM ARCCOS(-1)=PI
320 IF X=0 THEN RETURN 3.1415926/2 \ REM ARCCOS(0)=PI/2
330 RETURN ATN(SQRT(1-X↑2)/X) \ REM All other cases of X
340 FNEND
350 REM
360 REM
400 DEF FNS1(X) \ REM Returns ARCSIN(X) in radians.
405 REM X must lie in range -1 ... 1
410 IF ABS(X)=1 THEN RETURN X*(3.1415926/2)
415 REM ARCSIN(+\- 1) = +\- PI/2
420 RETURN ATN(X/SQRT(1-X↑2)) \ REM All other cases of X
430 FNEND
440 REM
450 REM
500 DEF FNB(B,P)=INT(B/2↑P)/2<>INT(INT(B/2↑P)/2)
510 REM Returns the Pth bit in byte B -- 0 or 1
520 REM
530 REM
```

```
600 DEF FND(H$)
605 REM Converts hex string in H$ to decimal value
606 REM and returns that.  Error condition occurs
610 REM if H$ is null or contains non-hex digits.
615 REM *** Uses variables T, E, and C without
616 REM     restoring them at return!
620 IF H$="" THEN 675
625 T=0
630 FOR E=LEN(H$) TO 1 STEP -1
635   C=ASC(H$(E,E))
640   IF (C < ASC("0")) OR (C > ASC("F")) THEN EXIT 675
645   IF (C >= ASC("0")) AND (C <= ASC("9")) THEN C=C-48
650   IF (C >= ASC("A")) AND (C <= ASC("F")) THEN C=C-55
655   IF (C > ASC("9")) AND (C < ASC("A")) THEN EXIT 675
660   T=T+C*(16↑(LEN(H$)-E))
665 NEXT
670 RETURN T
675 PRINT "BAD HEX NUMBER"
680 RETURN -1
685 FNEND
690 REM
695 REM
700 DEF FNh$(D)
705 REM Given decimal value D, returns string value
710 REM corresponding to hex form of D.
715 REM Negative arguments are turned positive,
716 REM non-integer numbers are truncated.
720 REM Uses variables H1$, D2, H, and I without
725 REM restoring them upon return.
730 D=INT(ABS(D)) \ H1$=""
735 H=INT(LOG(D)/LOG(16)+.5)
740 FOR I=H TO 0 STEP -1
745   D2=INT(D/(16↑I))
750   IF D2 >= 10 THEN H1$=H1$+CHR$(ASC("A")+D2-10)
755   IF D2 < 10 THEN H1$=H1$+CHR$(ASC("0")+D2)
760   D=D-(D2*(16↑I))
765 NEXT
770 RETURN H1$
775 FNEND
780 REM
785 REM
```

```
 10 PRINT "QUICKSORT-A TEST PROGRAM -- NUMBERS"
 11 PRINT "VERSION 1.0 -- 3/20/78"
 12 PRINT "NORTH STAR COMPUTERS, INC."
 15 REM Sorts array A of N numbers into ascending order.
 20 REM  Uses the array-partitioning scheme as
 30 REM  explained in section 2.2.6 (pp. 76-82) of
 35 REM  Wirth, ALGORITHMS + DATA STRUCTURES = PROGRAMS
 36 REM  (Prentice-Hall - 1976).
 50 REM  The quicksort mirrors Wirth's non-recursive
 60 REM  version (program 2.11, p. 80), and includes
 70 REM  the modifications suggested in the text --
 80 REM
 90 REM a) Comparand X is selected at random (line 1030)
100 REM   to avoid Quicksort's poor worst-case behavior.
110 REM
120 REM b) The size of the stack which holds accumulated
130 REM   partitioning information has been limited to
135 REM   log2(N) by incorporation of the program segment
140 REM   on page 82.  (See Wirth, Fig. 2.16, this
150 REM   corresponds to lines 1090-1160 here.)
160 REM
170 REM Note that the stack array, S9, is declared in
180 REM a DIM statement (line 210) before the Quicksort
190 REM routine is called, and the random-number
195 REM generator is "randomized" at this time also.
200 N=1000 \ REM Sort N numbers
210 DIM A(N), S9(INT(LOG(N)/LOG(2)+1),2)
215 REM A is main array, S9 is stack.
220 Q=RND(-1) \ REM Randomize PRN generator.
230 FOR Q=1 TO N\A(Q)=RND(0)\NEXT
235 REM Above fills A with random numbers.
240 FOR Q=1 TO N\PRINT A(Q)\NEXT \ REM Verify randomness.
250 PRINT "BEGIN SORT"
260 GOSUB 1000
270 PRINT "END SORT"
280 FOR Q=1 TO N\PRINT A(Q)\NEXT \ REM Show sorted array.
299 END
1000 REM BEGIN Quicksort in North Star BASIC
1001 REM Relies on existence of N, and arrays S9 and A
1002 REM Uses L, R, I, J, X, S, and W without
1003 REM  restoring them.
1010 S=1 \ REM S is stackpointer.
1015 S9(1,1)=1 \ S9(1,2)=N \ REM S9 is pre-DIMmed stack.
1020 L=S9(S,1) \ R=S9(S,2) \ S=S-1
1025 REM L and R are left and right partition boundaries.
1030 I=L \ J=R \ X=A(INT(RND(0)*(R-L)+.5)+L) \ ! "sort",
1040 IF A(I) >= X THEN 1050 \ I=I+1 \ GOTO 1040
1050 IF X >= A(J) THEN 1060 \ J=J-1 \ GOTO 1050
```

```
1060 IF I > J THEN 1080
1070 W=A(I) \ A(I)=A(J) \ A(J)=W \ I=I+1 \ J=J-1
1080 IF I <= J THEN 1040
1090 IF J-L >= R-I THEN 1140
1110 IF I >= R THEN 1130
1120 S=S+1 \ S9(S,1)=I \ S9(S,2)=R
1130 R=J \ GOTO 1170
1140 IF L >= J THEN 1160
1150 S=S+1 \ S9(S,1)=L \ S9(S,2)=J
1160 L=I
1170 IF L < R THEN 1030
1180 IF S > 0 THEN 1020
1199 PRINT \ RETURN \ REM END Quicksort.
```

```
    1 !"QUICKSORT-B TEST PROGRAM -- STRINGSORT"
    2 !"VERSION 1.0 -- RELEASE DATE: 3/20/78"
    3 !"NORTH STAR COMPUTERS, INC."
    4 REM Generates N strings of length G7, each containing random
    5 REM characters.  Strings held in "super string" R1$.
    6 REM Uses same algorithm as numeric Quicksort-A program above,
    7 REM except that this has been modified to sort strings using
    8 REM North Star substring conventions and user-functions.
    9 REM Many of the variable names have been changed, but sort
   10 REM is the same.
   12 G7=10\N=50
   13 DIM R$(G7),K$(G7),Q$(G7),R1$(G7*N)
   14 DIM U8(INT(LOG(N)/LOG(2)+.5),2) \ REM U8 is stack.
   15 W8=RND(-1) \ REM randomize the random number generator
   73 DEF FNX(X)=(X-1)*G7+1
   74 DEF FNY(Y)=Y*G7
   75 REM FNX and FNY are pointers to individual substrings of
   76 REM simulated array R1$.
   99 REM Below fills R1$ with random strings.
  100 FOR I=1 TO N
  110 Q$(1,1)=":"
  120 FOR J=2 TO G7-1
  130 Q$(J,J)=CHR$(INT(RND(0)*25+.5)+65)
  150 NEXT J
  170 Q$(G7,G7)="*"\R1$(FNX(I),FNY(I))=Q$
  175 !%3I,I,"  ",Q$
  180 NEXT I
  190 !"CREATION PHASE ENDED -- SORTING BEGINS"
  200 GOSUB 1000 \ REM Quicksorts R1$
  300 !"SORTING PHASE ENDED -- RESULTANT ARRAY:"
  320 FOR I=1 TO N
  330 !%3I,I,"  ",R1$(FNX(I),FNY(I)),
  332 IF I=N THEN 336
  335 IF I/15<>INT(I/15) THEN 336\INPUT "",X$\GOTO 340
  336 ! \ REM Above line and this are for output paging.
  340 NEXT I
  360 END
 1000 REM Quicksort of R1$, using FNX and FNY to point
 1005 REM to substrings.
 1010 N8=1\U8(1,1)=1\U8(1,2)=N
 1015 REM N8 is stack pointer.
 1020 L=U8(N8,1)\R=U8(N8,2)\N8=N8-1
 1030 I=L\J=R\Z8=INT((R-L)*RND(0)+.5)+L\!".",
 1035 K$=R1$(FNX(Z8),FNY(Z8))
 1040 IF R1$(FNX(I),FNY(I))>=K$ THEN 1050\I=I+1\GOTO 1040
 1050 IF K$>=R1$(FNX(J),FNY(J)) THEN 1060\J=J-1\GOTO 1050
```

```
1060 IF I>J THEN 1090
1070 R$=R1$(FNX(I),FNY(I))
1071 R1$(FNX(I),FNY(I))=R1$(FNX(J),FNY(J))
1080 R1$(FNX(J),FNY(J))=R$\I=I+1\J=J-1
1090 IF I<=J THEN 1040
1110 IF J-L>=R-I THEN 1150
1120 IF I>=R THEN 1140
1130 N8=N8+1\U8(N8,1)=I\U8(N8,2)=R
1140 R=J\GOTO 1180
1150 IF L>=J THEN 1170
1160 N8=N8+1\U8(N8,1)=L\U8(N8,2)=J
1170 L=I
1180 IF L<R THEN 1030
1190 IF N8>0 THEN 1020
1200 ! RETURN
```

```
  10 REM Test program for string search
  20 REM Version 1.0 -- 11/01/78
  30 REM North Star Computers, Inc.
  40 B=1000 \ REM Maximum length of any string used in program
  50 DIM A1$(B),A2$(B) \ REM These will hold arguments to FNS.
  60 DIM M$(B),N$(80)
  65 M$=CHR$(3)
  66 REM Control-C's will separate names in master list.
  70 REM M$ is main string, N$ is one name, F$ is name to find.
  80 REM Test program will input names (or arbitrary strings),
  90 REM rejecting duplications, and adding new ones to end
  95 REM of master list.
 100 GOSUB 1000 \ REM Give directions.
 110 GOSUB 2000 \ REM Get a name, put in N$.
 120 IF N$="" THEN 160 \ REM N$ will be null if time to quit.
 130 P=FNS(M$,CHR$(3)+N$+CHR$(3))
 140 GOSUB 3000 \ REM Add N$ to M$ if P=0, Otherwise, advise
 145 REM user that it is already in the main string.
 150 GOTO 110
 160 PRINT "QUIT"
 199 END
1000 PRINT "This program compiles a list of names which"
1010 PRINT "you type in from the keyboard.  Duplications"
1015 PRINT "are caught and rejected.  Be sure to strike"
1020 PRINT "the RETURN key after typing every name."
1030 PRINT "Striking RETURN alone when I ask for a name"
1040 PRINT "will quit the program."
1999 RETURN
2000 REM Get a name, put in N$.
2001 REM N$ will be null if time to quit.
2005 PRINT
2010 INPUT "Name (just strike <CR> to quit): ",N$
2999 RETURN
3000 REM Add N$ to M$ if P=0, Otherwise, advise user
3010 REM that it's already in the main string.
3020 REM Add-new-name fails if no more room in M$
3030 IF P=0 THEN 3060
3040 PRINT "*** Already in main string!"
3050 GOTO 3999
3060 REM Now, check to see if addition is physically possible.
3070 IF LEN(M$)+LEN(N$)+1 <= B THEN 3110
3080 PRINT "*** No room in main string to add.  Add rejected."
3090 GOTO 3999
```

```
3110 REM Now, REALLY add string and separator to main string.
3120 M$=M$+N$+CHR$(3)
3130 PRINT "<",N$,"> : added."
3999 RETURN
4000 DEF FNS(A1$,A2$)
4005 REM Uses variable T without preserving it.
4010 REM Looks for A2$ in A1$.  Value returned is
4020 REM first character position in A1$ where A2$
4030 REM is found.  Zero is returned if A2$ not found.
4040 IF LEN(A2$)> LEN(A1$) THEN 4090
4045 REM If A2$ longer than A1$, can't be contained in A1$
4050 IF A2$="" THEN 4090
4055 REM Null string is not substring of any non-null string.
4056 REM Scan down the string until a match is found.
4060 FOR T=1 TO LEN(A1$)-LEN(A2$)+1
4070   IF A1$(T,T+LEN(A2$)-1)=A2$ THEN EXIT 4095
4080 NEXT
4090 RETURN 0 \ REM A2$ not in A1$
4095 RETURN T
4096 REM T is char position in A1$ where A2$
4097 REM is first found.
4999 FNEND
```

```
   10 REM Magic Squares Program
   20 REM Version 1.0 -- 11/01/78
   30 REM North Star Computers, Inc.
   40 REM *** Demonstrates Array Handling in BASIC ***
   50 REM
   60 REM Global Variables Used --
   70 REM S -- number of elements in one side of square
   80 REM M -- flag, 0 if square not magic, nonzero if magic
   90 REM A -- array which holds the suspected square
   95 REM
  100 REM Main routine.
  110 GOSUB 1000 \ REM Give Directions.
  120 GOSUB 2000
  125 REM  Get DIM of side from user, and DIM A.
  130 REM Length of one side of square now in S.
  140 GOSUB 3000 \ REM Have user fill array elements.
  150 GOSUB 4000 \ REM Determine if square is magic.
  160 REM M is nonzero if square is magic.
  170 GOSUB 5000 \ REM Report results to user.
  199 END \ REM End of main routine.
 1000 REM Give directions for this program to the user.
 1010 PRINT "**** North Star Magic Squares Program ****"
 1020 PRINT
 1030 PRINT "A magic square is a grid of numbers in which"
 1035 PRINT "all the rows, all the columns, and both"
 1040 PRINT "diagonals add up to the same number."
 1050 PRINT "This program tests to see if a given square"
 1060 PRINT "of numbers is magic."
 1064 PRINT
 1065 PRINT "You may choose to input a square of up"
 1070 PRINT "to 5x5 numbers.  I will tell you whether"
 1075 PRINT "or not the square you give me is a magic"
 1080 PRINT "square.  Please be sure to type your"
 1090 PRINT "answers to me when I ask.  Conclude each"
 1110 PRINT "response by striking the RETURN key."
 1120 PRINT
 1999 RETURN
 2000 REM Get DIM of side, S, from user.  Use S to DIM A.
 2010 PRINT
 2020 INPUT "Type the length of one side: ",S
 2030 IF S>=1 AND S <= 5 AND S=INT(S) THEN 2070
 2040 PRINT "*** BAD INPUT"
 2050 PRINT "Your answer must be an integer from"
 2051 PRINT "1 to 5.  Please Try again."
 2060 GOTO 2010
 2070 DIM A(S-1,S-1) \ REM 0-element is used to save space.
 2999 RETURN
 3000 REM Have user fill array elements, and re-display
```

```
3005 REM the input as a square.
3010 PRINT
3020 PRINT "Please give me the appropriate number to"
3021 PRINT "fill each co-ordinate of the proposed magic"
3030 PRINT "square.  I will give co-ordinates in row-"
3040 PRINT "column form:"
3045 PRINT TAB(3),"(row,column)= <you type number here>"
3050 PRINT
3060 FOR R0=0 TO S-1
3070   FOR C0=0 TO S-1
3080     PRINT "(",%1I,R0+1,",",C0+1,")= ",
3090     INPUT "",A(R0,C0)
3110   NEXT
3120 NEXT
3130 PRINT
3140 PRINT "All square positions have been filled."
3150 PRINT "Thank you!"
3151 PRINT
3160 PRINT "Here is your proposed magic square:"
3170 PRINT
3175 REM Now scan through and display square in grid format.
3180 FOR R0=0 TO S-1
3190   FOR C0=0 TO S-1
3210     PRINT TAB(C0*12),A(R0,C0),
3215 REM  Item field widths are 12 columns.
3220   NEXT
3230   PRINT \ PRINT \ PRINT
3240 NEXT
3999 RETURN
4000 REM Determine if square in array A is magic.
4010 REM On return, M <> 0 if magic, M=0 if not.
4020 REM Add up rows, columns, and diagonals.
4030 REM "Master" total kept in T1,
4040 REM Temporary Row, Column, and Diagonals
4050 REM totals kept in R1, C1, D1, D2.
4110 M=0 \ REM Assume not magic until we prove it is.
4130 D1=0 \ D2=0 \ REM Initialize diagonals.
4140 FOR R0=0 TO S-1
4150   D1=D1+A(R0,R0) \ D2=D2+A(R0,S-1-R0)
4155 REM  Above updates diagonals
4160   R1=0 \ C1=0 \ REM Initialize row, column temp totals.
4170   FOR C0=0 TO S-1
4180     R1=R1+A(R0,C0) \ C1=C1+A(C0,R0)
4181 REM  Above updates row and column.
4190   NEXT
4210   IF R0=0 THEN T1=R1
4211 REM  Arbitrarily choose 1st row as master total.
4220   IF (R1<>T1) OR (C1<>T1) THEN EXIT 4999
4225 REM If row or column <> master, return M=0.
4230 NEXT
```

```
4240 IF (D1<>T1) OR (D2<>T1) THEN 4999
4245 REM If diagonals don't match master, return with M=0.
4250 M=1 \ REM If here, all totals have matched master.
4999 RETURN
5000 REM Report results to user.
5020 PRINT "This square is ",
5030 IF M=0 THEN PRINT "NOT ",
5040 PRINT "a magic square."
5050 PRINT
5999 RETURN
```

APPENDIX [illegible]

This [illegible] lists all the [illegible] messages printed by North Star BASIC. For [illegible] in parentheses after ERRSET statement, the error [illegible] the identifying error-message.

The brief discussion of the [illegible] of each error is [illegible] ERROR MESSAGE [illegible] large part to [illegible] sections for each [illegible] as given [illegible] treated in DISCUSSION [illegible] (and as also [illegible]) sections.

[illegible]
[illegible] attempt has been made to [illegible] invalid argument to a [illegible] or function.

ARG [illegible] ERROR (3)
[illegible] for a user-defined [illegible] number of arguments [illegible] parameters for that [illegible] does not match [illegible]

CONTINUE ERROR (non-trappable)
[illegible] illegal attempt has been made to CONTinue program [illegible] not be CONTinued if the [illegible] Program execution [illegible] error, if any editing of [illegible] execution stopped [illegible] during an interruption, [illegible] program has taken place [illegible] END statement. [illegible] the program has executed [illegible]

[illegible] ERROR (non-trappable)
[illegible] improper nesting of FOR and [illegible] error occurs when there [illegible] statements, or multi-line [illegible] statements, GOSUB and RETURN statements. It also occurs [illegible] function calls and RETURN statement in a program. [illegible] a FOR statement is the last statement [illegible]

DIM [illegible] ERROR (2)
[illegible] attempt has been made to redimension an array or string, [illegible] use the DIMension statement in some other, illegal, [illegible]

DIVIDE [illegible] ERROR (9)
[illegible] attempt has been made to divide by zero.

DOUBLE [illegible] ERROR (non-trappable)
[illegible] exists more than one definition for the same user-[illegible] functions are defined at RUN [illegible] in the same program. [illegible] before program execution [illegible] this message will occur [illegible] begins.

FILE [illegible]
[illegible] diskette file which [illegible] program is trying to access [illegible] type. This error will also [illegible] exist or is [illegible]

occur when you try to LOAD a BASIC program from a type 2 file which has never before held a BASIC program. File errors occur when attempts are made to use file numbers which are less than Ø or greater than 7, or when a file is being OPENed, but the file number specified is already in use. Attempts to CREATE or NSAVE files onto diskettes too full to hold them also yield a FILE ERROR. Finally, a FILE ERROR can occur if any attempts are made to store information on, or erase information from, a write-protected diskette.

FORMAT ERROR (5)
An illegal format string has been used in a PRINT statement. Either the format string is formed incorrectly, or the field specifications are too big or are inconsistent. Also, an attempt to PRINT a value which won't fit into a specified field, or to PRINT a non-integral value using I-format will result in this error.

FUNCTION DEF ERROR (non-trappable)
This means that BASIC has encountered the beginning of a new user-function definition (a DEF statement) before the previous definition has been concluded. Generally, the function defined immediately above the offending DEF statement does not include (but needs) a FNEND statement. This error also occurs when an attempt is made to call an undefined user-function.

HARD DISK ERROR (8)
An impossible disk access was attempted. This can result from not having a properly mounted diskette, or from having a diskette with unreadable data. See the DOS manual for further discussion.

ILLEGAL DIRECT ERROR (non-trappable)
An attempt was made to use a statement in direct mode which can only be used as part of a program. See DISCUSSION: SOME BASIC CONCEPTS for a list of those statements which may be used in direct mode. Note that user-functions may not be used in direct mode.

INPUT ERROR (12)
During the execution of an INPUT statement, the user typed an improprely formed numeric constant in response to a programmed request for numeric input.

INTERNAL STACK OV (non-trappable)
This message should not occur in normal BASIC programs. It means that an unanticipated amount of internal BASIC memory was required to process the STATEMENT or COMMAND. Please report the circumstances to North Star (in writing) if this error occurs.

LENGTH ERROR (16)
This error occurs if an attempt is made to type a longer line of text than BASIC allows. (This limit may be reset by using the LINE statement.) Typically, LENGTH ERRORs may occur when typing in response to INPUT statements, or when entering program statements or commands to BASIC. Unless otherwise personalized or informed by the LINE statement, BASIC assumes that a line may be no longer than 80 characters.

LINE NUMBER ERROR (6)
There is a missing or improperly formed line number in the erroneous COMMAND or STATEMENT. Also, if a line number is specified in a COMMAND or STATEMENT, but that line cannot be found in the current BASIC program, a LINE NUMBER ERROR will be generated.

MEMORY FULL ERROR (non-trappable)
The total amount of memory available to BASIC is insufficient to contain the current program, its variables, and temporary storage. The MEMSET command may be used to expand the available memory area. Note that, when performing string concatenations, BASIC reserves as temporary storage an area in memory as large as the concatenated string itself. BASIC also reserves this temporary storage when PRINTing expressions, so PRINTing large string expressions may sometimes result in this error.

MISSING NEXT ERROR (non-trappable)
Within an executing program, a FOR statement is encountered for which no matching NEXT can be found.

NO PROGRAM ERROR (non-trappable)
This error occurs when an attempt is made to RUN and there is no current program.

NUMERIC OV ERROR (14)
This error occurs whenever an arithmetic operation results in a number larger than 9.9999999E+62. Numbers larger than this cannot be represented in standard versions of North Star BASIC. (Numbers smaller than 1E-64 are converted to 0.)

OUT OF BOUNDS ERROR (3)
This message occurs when a numeric argument is not within legal range, e.g., when an array subscript is too large or too small, or when an argument used with CALL, EXAM, FILL, INP, or OUT is not in the correct range. When dealing with diskette files, an OUT OF BOUNDS ERROR will occur as attempts are made to READ from or WRITE to a file beyond its absolute end (determined by the file size).

READ ERROR (11)
When using the READ statement, if an attempt is made to READ a numeric value into a string variable or vice versa, or to READ any value when there is no more DATA available, a READ ERROR will occur.

STOP (15)
This is not really an error, but when control-C is enabled and pressed while an ERRSET statement is in effect, the attempted program interruption is treated as a program error, with 15 as its code. In other words, "error 15" means that control-C was pressed while ERRSET is in effect.

SYNTAX ERROR (10)
This is the most commonly-generated error message. It occurs when a language feature has been used improperly, or has been improperly formed (typed incorrectly). Most of these mistakes become obvious upon brief (but careful) examination of the faulty COMMAND or STATEMENT (as compared with its manual description). Refer to the appropriate exposition or DISCUSSION section to determine the correct form of the language feature in question, and make sure that all keywords are correctly spelled.

TOO LARGE OR NO PROGRAM ERROR (non-trappable)
This message occurs when an attempt is made to LOAD, APPEND, or CHAIN to a program which either is too large to fit in the program/data area, or is not a valid BASIC program.

TYPE ERROR (4)
TYPE ERRORs happen when a string value appears where a numeric value is expected, or vice versa. With regard to disk file operations, an attempt to OPEN a file whose actual type doesn't agree with the type specified in the program, or to READ a value on disk into a program variable of the wrong type, will lead to this error.

IMPLEMENTATION NOTES

APPENDIX B

This appendix is designed to provide important details concerning some of the internal workings of North Star BASIC, and the internal representations of data within BASIC, in order to help you better understand the operation of the system, and to facilitate writing of programs which perform tasks which would be difficult or impossible to undertake without such information.

DISKETTE DATA-STORAGE FORMATS

All NUMBERS which have been written to diskette by a BASIC of a given precision will have a standard fixed storage size in bytes. However, the storage size of a number written to disk by 6-digit BASIC, for example, will be smaller in size than that of a number written by 10-digit BASIC. Here is a chart which tells how many bytes a number will require on disk, depending upon the precision of the BASIC writing it:

PRECISION	BYTES
6	4
8	5
10	6
12	7
14	8

Numbers are stored in packed, binary-coded-decimal (BCD) form. The representation is as follows:

first byte:
bits 7-4 = most signicant digit of value in BCD coding
bits 3-0 = next most significant digit of value

middle bytes:
bits 7-4 = next significant digit of value in BCD coding
bits 3-0 = next significant digit of value

last byte:
bit 7 = sign (1=negative, 0=positive)
bits 6-0 = exponent in excess 64 binary representation (If all bits in the last byte are 0, the entire number is 0.)

All values are normalized.

The decimal value of the first byte in a number stored on disk will always be greater than 15, even when the number is zero. (This is how the TYP function determines if the next data element is numeric.)

STRINGS are stored using a number of bytes equal to the length of the string plus two or three overhead bytes. Strings of length less than or equal to 255 are stored with two overhead bytes, the first one being of decimal value 3, and the second containing the number of characters in the string. The information bytes -- the string itself -- follow the overhead bytes. A string value of length greater than 255 is stored with three overhead bytes, the first one being of value 2, and the second two being the low and high bytes, respectively, of the length of the string, expressed as a 16-bit integer. Again, the string itself follows the overhead.

The ENDMARK for a sequential file is a single byte of value 1.

FILE BUFFER SIZES -- LIFETIMES OF BUFFERS

When each file is OPENed, an area of RAM memory is reserved as a high-speed data-transfer "buffer" between BASIC and the disk drive. A buffer of 256 bytes is reserved when OPENing a single-density file. With double-density files, the buffer size is 512 bytes. Buffers are used to make disk access as efficient and quick as possible. When the file is CLOSEd, its buffer region does not return to free-memory, but is reserved for later use by any files which will be opened under the file number associated with the buffer.

TYPE-DEPENDENT INFORMATION IN A TYPE-2 FILE DIRECTORY ENTRY

Those familiar with the DOS and the details of diskette directory entries will realize that 3 bytes are reserved in each entry for what is termed "type-dependent" information. For a type 1 file, this area is used to store the GO address for the file. For type 2 files -- that is, BASIC program files -- the information stored in the "type-dependent" slot is the actual size of the program in disk blocks. This information, stored as part of a program's directory entry, and updated every time a program is SAVEd or NSAVEd into that file, allows BASIC to make economical use of its time when LOADing a BASIC program -- it may read only as much program data as actually exists in a file, and need not waste time attempting to LOAD information from beyond the end of the program. This number is stored in byte 13 in a type 2 file's directory entry. See the DOS section of the NORTH STAR SYSTEM SOFTWARE MANUAL for more information about directory entries.

PRINT HEAD TABLE

At memory addresses ORG+17 and ORG+18 (ORG+11H and ORG+12H) there exists a pointer containing the low and high bytes,

respectively, of the address in memory where BASIC's "print-head-table" is stored. Each of the 8 bytes in this table contains the current cursor position for one of BASIC's 8 possible I/O devices (starting with device #0). For some applications, such as plotting, some users may wish to EXAM or FILL these bytes to avoid LENGTH ERROR messages or the automatic carriage-return which BASIC supplies when enough characters to fill a line have been PRINTed on a given device. Users with standard versions of BASIC may use the following user-function to return the address of the table entry for any of the 8 devices. EXAM or FILL this address to determine or change the value of the print-head counter for the given device.

```
DEF FNH(D)=EXAM(11537)+(EXAM(11538)*256)+D
REM D IS DEVICE NUMBER FROM 0 TO 7
```

FILE-HEADER TABLE

This table follows immediately the 8 bytes of the print-head table described above. The file-header table is 80 bytes long, and contains one 10-byte entry for each of the 8 possible open files (0 to 7). Each entry has the following format:

a) byte 0: status byte
b) bytes 1-2: buffer address for the file (low/high)
c) bytes 3-4: disk address of the open file (the number of the file's beginning disk block)
d) bytes 5-6: filesize in blocks
e) bytes 7-9: current file pointer -- this points to the next byte to be accessed, expressed as an offset from the start of the file. Because three bytes (arranged as middle byte, high byte, low byte) are used to represent the pointer value, BASIC may access files as large as an entire diskette side (single or double density).

BASIC PROGRAM PRE-PROCESSING

Once program lines are typed into BASIC, they are pre-processed automatically into a more compact, efficient form where each reserved word maps onto a single byte value, and line number references in GOTO, GOSUB, RESTORE and similar statements are collapsed into 16-bit values. This permits faster execution, and more efficient use of storage space in both RAM (when the program is RUNning or under development) and disk (when the program is SAVEd or NSAVEd). When the program is LISTed, the compaction process is reversed, and the complete text of the program is restored for the user. The conversion of program-line text into compacted form even extends to REM statements. REMs which include instances of

keywords will take up less memory space than REMs of equivalent length which contain no embedded keywords. For example

REM FOR THE NEXT ORIGIN, LETS TRY 2000H

will be compcted into a much smaller internal form than

REM 2000H HEX IS THE NEW STARTING PLACE

because the former includes instances of FOR, NEXT, OR, and LET -- all keywords which will be compacted to single-byte form. The second REM includes no embedded keywords, so will be stored in exactly the same form as it is written. Spaces are retained in the number and order typed in the program line to preserve the author's style and any indentation. Compaction does not occur within quoted strings.

Throughout the evolution of North Star BASIC, certain single-byte keyword codes have had their meanings changed. As a result, REMs in programs which were written under earlier versions of BASIC may undergo small changes when the programs are LISTed under release 4 or later versions of BASIC. This is because these REMs included embedded keywords which were compacted to single bytes, and now, these codes are translated back into different keywords. In particular, instances of CREATE, DUMP, and NULL in older REM statements will become AUTO, MEMSET, and NSAVE respectively. To correct this, just retype the correct form of the altered REM statement and re-SAVE the program.

Note also that programs written under later versions of BASIC will not always list properly under earlier BASICs, especially if they include some of the newer keywords, such as CREATE, ERRSET, etc.

THE INTERNAL FORM OF A PROGRAM

In RAM and on disk, a program is represented as a series of program lines which have been converted to the compacted form mentioned above. Each line is arranged as follows:

a) byte 0: contains the binary representation of the number of bytes in the program line (called "N" here for purposes of discussion).
b) bytes 1-2: the program-line number expressed as a 16-bit binary integer (low byte/high byte).
c) bytes up to N-2: the program line in its compacted form.
d) byte N-1: A carriage-return character (byte value 13 or 0DH).

There is a standard ENDMARK (byte value 1) after the last

line in the program.

USE OF RAM DURING PROGRAM EXECUTION

When a program is executing, BASIC maintains two variable-size data storage areas at opposite ends of memory. These are the GENERAL DATA AREA and the BASIC CONTROL STACK. The general data area begins immediately above the last byte in the current BASIC program. This storage area contains BASIC's symbol table, and static storage space which has been allocated for numeric variables, arrays, and strings. The general data area grows from low memory to high memory.

BASIC's control stack begins at the highest byte available to the BASIC system, and grows downward, into low memory. The stack contains highly transient information such as FOR-NEXT, GOSUB, and user-function call linkages. Whenever program conditions lead to the case that one of these areas is made to "grow" into the other, a MEMORY FULL ERROR occurs.

APPENDIX 4

DECIMAL-ASCII-HEX-BINARY CONVERSION TABLE

The following table is intended to ease the task of conversion between the various numeric representations commonly used in programming, as well as between numbers (of any kind) and the ASCII character code.

Note that the ASCII character set only goes as far as decimal 127 (7FH, 01111111 B). Also, many "characters" in ASCII are non-printing CONTROL CHARACTERS. Whenever a code corresponds to a printable character, that will be given. In the case of control characters, a description or name for the special character will be given in parentheses.

DECIMAL	HEX	BINARY	ASCII
0	00H	00000000	(NUL)
1	01H	00000001	(CONTROL-A)
2	02H	00000010	(CONTROL-B)
3	03H	00000011	(CONTROL-C)
4	04H	00000100	(CONTROL-D)
5	05H	00000101	(CONTROL-E)
6	06H	00000110	(CONTROL-F)
7	07H	00000111	(CONTROL-G, RINGS BELL)
8	08H	00001000	(CONTROL-H, BACKSPACE)
9	09H	00001001	(CONTROL-I, TAB)
10	0AH	00001010	(CONTROL-J, LINEFEED)
11	0BH	00001011	(CONTROL-K)
12	0CH	00001100	(CONTROL-L, FORMFEED)
13	0DH	00001101	(CONTROL-M, CARRIAGE RETURN)
14	0EH	00001110	(CONTROL-N)
15	0FH	00001111	(CONTROL-O)
16	10H	00010000	(CONTROL-P)
17	11H	00010001	(CONTROL-Q)
18	12H	00010010	(CONTROL-R)
19	13H	00010011	(CONTROL-S)
20	14H	00010100	(CONTROL-T)
21	15H	00010101	(CONTROL-U)
22	16H	00010110	(CONTROL-V)
23	17H	00010111	(CONTROL-W)
24	18H	00011000	(CONTROL-X)
25	19H	00011001	(CONTROL-Y)
26	1AH	00011010	(CONTROL-Z)
27	1BH	00011011	(ESCAPE)
28	1CH	00011100	(NON-PRINTING)
29	1DH	00011101	(NON-PRINTING)
30	1EH	00011110	(NON-PRINTING)
31	1FH	00011111	(NON-PRINTING)

CONVERSION TABLE (Continued)

DECIMAL	HEX	BINARY	ASCII
32	20H	00100000	(SPACE)
33	21H	00100001	!
34	22H	00100010	"
35	23H	00100011	#
36	24H	00100100	$
37	25H	00100101	%
38	26H	00100110	&
39	27H	00100111	'
40	28H	00101000	(
41	29H	00101001	)
42	2AH	00101010	*
43	2BH	00101011	+
44	2CH	00101100	,
45	2DH	00101101	-
46	2EH	00101110	.
47	2FH	00101111	/
48	30H	00110000	0
49	31H	00110001	1
50	32H	00110010	2
51	33H	00110011	3
52	34H	00110100	4
53	35H	00110101	5
54	36H	00110110	6
55	37H	00110111	7
56	38H	00111000	8
57	39H	00111001	9
58	3AH	00111010	:
59	3BH	00111011	;
60	3CH	00111100	<
61	3DH	00111101	=
62	3EH	00111110	>
63	3FH	00111111	?

CONVERSION TABLE (Continued)

DECIMAL	HEX	BINARY	ASCII
64	40H	01000000	@
65	41H	01000001	A
66	42H	01000010	B
67	43H	01000011	C
68	44H	01000100	D
69	45H	01000101	E
70	46H	01000110	F
71	47H	01000111	G
72	48H	01001000	H
73	49H	01001001	I
74	4AH	01001010	J
75	4BH	01001011	K
76	4CH	01001100	L
77	4DH	01001101	M
78	4EH	01001110	N
79	4FH	01001111	O
80	50H	01010000	P
81	51H	01010001	Q
82	52H	01010010	R
83	53H	01010011	S
84	54H	01010100	T
85	55H	01010101	U
86	56H	01010110	V
87	57H	01010111	W
88	58H	01011000	X
89	59H	01011001	Y
90	5AH	01011010	Z
91	5BH	01011011	[
92	5CH	01011100	\
93	5DH	01011101	]
94	5EH	01011110	↑ OR ^
95	5FH	01011111	_

CONVERSION TABLE (Continued)

DECIMAL	HEX	BINARY	ASCII
96	60H	01100000	`
97	61H	01100001	a
98	62H	01100010	b
99	63H	01100011	c
100	64H	01100100	d
101	65H	01100101	e
102	66H	01100110	f
103	67H	01100111	g
104	68H	01101000	h
105	69H	01101001	i
106	6AH	01101010	j
107	6BH	01101011	k
108	6CH	01101100	l
109	6DH	01101101	m
110	6EH	01101110	n
111	6FH	01101111	o
112	70H	01110000	p
113	71H	01110001	q
114	72H	01110010	r
115	73H	01110011	s
116	74H	01110100	t
117	75H	01110101	u
118	76H	01110110	v
119	77H	01110111	w
120	78H	01111000	x
121	79H	01111001	y
122	7AH	01111010	z
123	7BH	01111011	{
124	7CH	01111100	|
125	7DH	01111101	}
126	7EH	01111110	~
127	7FH	01111111	(DELETE, RUB OUT)

CONVERSION TABLE (Continued)

DECIMAL	HEX	BINARY	ASCII
128	80H	10000000	
129	81H	10000001	
130	82H	10000010	
131	83H	10000011	
132	84H	10000100	
133	85H	10000101	
134	86H	10000110	
135	87H	10000111	
136	88H	10001000	
137	89H	10001001	
138	8AH	10001010	
139	8BH	10001011	
140	8CH	10001100	
141	8DH	10001101	
142	8EH	10001110	
143	8FH	10001111	
144	90H	10010000	
145	91H	10010001	
146	92H	10010010	
147	93H	10010011	
148	94H	10010100	
149	95H	10010101	
150	96H	10010110	
151	97H	10010111	
152	98H	10011000	
153	99H	10011001	
154	9AH	10011010	
155	9BH	10011011	
156	9CH	10011100	
157	9DH	10011101	
158	9EH	10011110	
159	9FH	10011111	

CONVERSION TABLE (Continued)

DECIMAL	HEX	BINARY	ASCII
160	A0H	10100000	
161	A1H	10100001	
162	A2H	10100010	
163	A3H	10100011	
164	A4H	10100100	
165	A5H	10100101	
166	A6H	10100110	
167	A7H	10100111	
168	A8H	10101000	
169	A9H	10101001	
170	AAH	10101010	
171	ABH	10101011	
172	ACH	10101100	
173	ADH	10101101	
174	AEH	10101110	
175	AFH	10101111	
176	B0H	10110000	
177	B1H	10110001	
178	B2H	10110010	
179	B3H	10110011	
180	B4H	10110100	
181	B5H	10110101	
182	B6H	10110110	
183	B7H	10110111	
184	B8H	10111000	
185	B9H	10111001	
186	BAH	10111010	
187	BBH	10111011	
188	BCH	10111100	
189	BDH	10111101	
190	BEH	10111110	
191	BFH	10111111	

CONVERSION TABLE (Continued)

DECIMAL	HEX	BINARY	ASCII
192	C0H	11000000	
193	C1H	11000001	
194	C2H	11000010	
195	C3H	11000011	
196	C4H	11000100	
197	C5H	11000101	
198	C6H	11000110	
199	C7H	11000111	
200	C8H	11001000	
201	C9H	11001001	
202	CAH	11001010	
203	CBH	11001011	
204	CCH	11001100	
205	CDH	11001101	
206	CEH	11001110	
207	CFH	11001111	
208	D0H	11010000	
209	D1H	11010001	
210	D2H	11010010	
211	D3H	11010011	
212	D4H	11010100	
213	D5H	11010101	
214	D6H	11010110	
215	D7H	11010111	
216	D8H	11011000	
217	D9H	11011001	
218	DAH	11011010	
219	DBH	11011011	
220	DCH	11011100	
221	DDH	11011101	
222	DEH	11011110	
223	DFH	11011111	

CONVERSION TABLE (Continued)

DECIMAL	HEX	BINARY	ASCII
224	E0H	11100000	
225	E1H	11100001	
226	E2H	11100010	
227	E3H	11100011	
228	E4H	11100100	
229	E5H	11100101	
230	E6H	11100110	
231	E7H	11100111	
232	E8H	11101000	
233	E9H	11101001	
234	EAH	11101010	
235	EBH	11101011	
236	ECH	11101100	
237	EDH	11101101	
238	EEH	11101110	
239	EFH	11101111	
240	F0H	11110000	
241	F1H	11110001	
242	F2H	11110010	
243	F3H	11110011	
244	F4H	11110100	
245	F5H	11110101	
246	F6H	11110110	
247	F7H	11110111	
248	F8H	11111000	
249	F9H	11111001	
250	FAH	11111010	
251	FBH	11111011	
252	FCH	11111100	
253	FDH	11111101	
254	FEH	11111110	
255	FFH	11111111	

BASIC TOPICS INDEX

APPENDIX 5

This is the index to topics and discussion sections in the BASIC section of the System Software Manual, and is designed to help the reader study North Star BASIC from a topical standpoint.

Listings to DISCUSSION sections are given in all-capital letters. Those which refer to general topics are given in lower-case.

The page reference format is a hyphenated one, with the chapter designation as a capital letter appearing on the left side of the hyphen, and the page number within the chapter appearing in arabic form on the right side. For example, the listing

constant, numeric D-1

indicates that the term "Numeric constant" is discussed in chapter D, page 1. If a topic consumes a whole chapter, only the chapter letter is given as the page reference. Page intervals are denoted by inserting an ellipsis (...) between the page references of the first and last pages in the interval. Whenever information about a given topic appears on more than one separate page, the pages with the most important information are listed in order first, then those with less important information. A semi-colon (;) separates the list of more-important references from the less-important ones within an entry.

argument list K-1
arguments K-1; B-9
arrays E
  default dimensions E-3
  re-dimensioning E-3
ASCII character set F-7
AUTOMATIC PROGRAM SEQUENCING M-6...M-7

BCD (Binary Coded Decimal) D-1; N-2, A3-1
bootstrap PROM, non-standard O-7

CHAINING M-6...M-7
character deletion, changing echo for O-5
character set F-7
command B-9
COMMUNICATING WITH BASIC B-2
COMPATIBILITY WITH OTHER BASICS N
  BCD arithmetic N-2
  IF...THEN evaluation N-3
  input translation N-2
  string handling N-1
concatenation F-3
console terminal B-2
constant, numeric D-1
constant, string F-1

control-c inhibit O-5
control characters B-5
control statement J-1
current format H-7
current length F-2; F-7
current program B-9

DATA FILES L-1...L-9
  appending to sequential files L-6
  closing L-3
  creating L-3
  endmark L-4...L-7
  file name L-1
  file number L-3
  file pointer L-7; L-14, L-16
  file size L-2
  file type L-2
  opening L-3
  random access L-7
  random address (expression) L-8
  sequential access L-4
  sequential byte access L-7
  types of elements in -- L-4
data pointer I-2; I-4
default format H-7
device expression H-12
dimension E-2
direct mode B-9; J-5
dimensioning, strings F-1
drive number suffix L-1

E-format H-3; D-1
endmark L-4...L-7
ENTERING A BASIC PROGRAM B-6...B-8
ERROR TRAPPING AND RECOVERY M-9...M-10
EXECUTION AND CONTROL FLOW J-1
exponent D-1
exponential format D-1
expression, numeric D-6
expression, string F-3

field width H-4
file block L-2
file buffer A3-2
file header table A3-3
file name L-1
file number L-3
file pointer L-7; L-14, L-16
file size L-2
file type L-2
Floating Point Board (FPB) BASICs O-13; O-7, O-11
FOR-NEXT LOOP, THE J-7...J-11

  body of -- J-7
  control variable J-7
  exiting from nested loops J-11
  limit value J-7
  nesting J-9
  optional control variable in NEXT J-10
  step value J-7
format specification H-4
FORMATTED PRINTING H-3...H-8
  allowable formats (chart) H-6
  format characters H-7
free format H-3
FUNCTIONS K
  built-in K-1...K-7
  string F-3
  user -- K-8...K-11
    function call K-8
    multi-line K-10
    names K-8
    numeric parameters K-9
    single-line K-8
    string parameters K-9

hexadecimal C-17

I-format H-4
IMPLEMENTATION NOTES A3
index number E-1

justification, right H-5

LINE EDITOR, THE M-13...M-19
  new line M-13
  old line M-13
  specifics and functions M-15...M-19
line length O-4; C-18
line number B-5, B-10; J-2
LOADING BASIC B-1

MACHINE LANGUAGE SUBROUTINES M-4...M-5
mantissa D-1
maximum length F-1; F-7
memory size O-2
memory usage during program execution A3-5
MULTIPLE I/O DEVICES H-12...H-13

nesting
  of FOR-NEXT loop J-9
  of IF statements J-3
  of subroutines J-17
new line M-13
NON-STANDARD VERSIONS OF BASIC O-12...O-13

null string F-1
numbers D

old line M-13
open-ended substring F-3
operators
  arithmetic D-4
  boolean D-5...D-6
  numeric D-4...D-6
  numeric, order of evaluation D-6
  relational D-4...D-5
  string F-3
output data list H-1

paging (video) O-4
PERSONALIZING BASIC O-2...O-11
precedence D-6...D-7
precision, numeric D-1; O-13
print-head table A3-2
program B-6; B-9
  internal form of -- A3-4
  -- pre-processing A3-3
program/data area C-17; G-1
program line B-6; B-10
program mode B-9

random address (expression) L-8
range, numeric D-3
regular format H-3
relocation of BASIC O-12

scientific notation D-1
sector, diskette L-2
sequential execution J-1
shrinking BASIC O-6
SPECIAL ENTRY POINTS O-1
statement B-9
strings F
  assignment to substrings and -- F-5...F-7
  comparisons F-4
  compatibility with other BASICs N-1
  current length F-2; F-7
  functions F-3
  maximum length F-1; F-7
SUBROUTINES J-15...J-16
SUBROUTINES, MACHINE LANGUAGE M-4...M-5
subroutines, nesting J-17
subscript E-1
substring F-2...F-3
  open-ended -- F-3
  -- interval F-2
  -- notation F-2

truncation E-1; F-5
turnkey startup of BASIC O-8
typing to BASIC B-2

upper case bias of BASIC B-2
user-functions K-8...K-11
USING ARRAYS E
USING NUMBERS D
USING STRINGS F

value, numeric D-3
variable
-- name D-3, F-1
simple -- D-3...D-4
numeric -- D-3
string -- F-1
string --, dimensioning of F-1

zero-element E-1

APPENDIX 6

This is an index of the statements, commands, and functions in North Star BASIC, and is included to facilitate the manual's use by experienced programmers needing to look up specific information in a hurry. Page number references follow the convention set in APPENDIX 5; refer to that APPENDIX if you are unfamiliar with the format.

ABS K-2
APPEND C-11
ASC K-3
ATN K-3
AUTO C-6

BYE C-20

CALL K-6; M-4
CAT C-7
CHAIN M-8
CHR$ K-3; F-7...F-8
CLOSE L-13
CONT C-15; J-5...J-6
CONTROL-C C-13
COS K-3
CREATE L-10

DATA I-1
DEF K-12
DEL C-2
DESTROY L-11
DIM G-1; E-2, F-1

EDIT M-15
END J-6
ERRSET M-11
EXAM K-6
EXIT J-14; J-10
EXP K-3

FILE K-5
FILL M-1
FNEND K-14
FOR J-12
FREE K-6

GOSUB J-17
GOTO J-2

IF...THEN...ELSE J-3
INCHAR$ K-4
INP K-5; H-10

INPUT H-9
INPUT1 H-11
INT K-2

LEN K-3; F-7
LET G-4
LINE C-18
LIST C-1
LOAD C-10
LOG K-2

MEMSET C-17

NEXT J-13
NOENDMARK L-7; L-9, L-16
NSAVE C-9

ON...GOTO J-4
OPEN L-12
OUT M-3

PANIC BUTTON (CONTROL-C) C-13
PRINT H-1
PSIZE C-16

READ I-2
READ# L-14; L-4
REM G-3
REN C-4
RESTORE I-4
RETURN
  subroutines J-16; J-18
  user-functions K-13
RND K-5
RUN C-12; I-4

SAVE C-8
SCR C-3
SGN K-2
SIN K-3
SQRT K-3
STOP J-5
STR$ K-4

TAB K-6; H-5
TYP K-5; L-5

VAL K-4

WRITE# L-16; L-4