NSB8

BASIC interpreter
for Z80 family

USER MANUAL
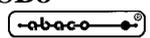
NSB8

BASIC interpreter
for Z80 family

NSB8 is a powerful software development tool which allows high-level programming in BASIC on all the **grifo®** boards based on Z80 microprocessor family.

It includes a complete floating point management with 8 significant digits plus trigonometric and trascendental functions.

The development environment is extremly friendly and achieves to reduce the development time, being anyway compliant to the operational feeling of all the BASICs. The great code performance and the rapidity of hardware intervents make the **NSB8** an unreplaceable work instrument for all the applications.

The **NSB8** interpreter supports mathematic functions, control applications, data base management, interfacing to generic consoles, operating system calls and many other features designed to solve industrial automation problems.
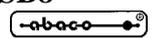
DOCUMENTATION COPYRIGHT BY grifo®, ALL RIGHTS RESERVED

IMPORTANT

SYMBOLS DESCRIPTION

In the manual could appear the following symbols:

    Attention: Generic danger

    Attention: High voltage

Trade Marks

GENERAL INDEX

FIGURE INDEX

INTRODUCTION

The use of these devices has turned - IN EXCLUSIVE WAY - to specialized personnel.

The purpose of this handbook is to give the necessary information to the cognizant and sure use of the products. They are the result of a continual and systematic elaboration of data and technical tests saved and validated from the manufacturer, related to the inside modes of certainty and quality of the information.

The reported data are destined- IN EXCLUSIVE WAY- to specialized users, that can interact with the devices in safety conditions for the persons, for the machine and for the enviroment, impersonating an elementary diagnostic of breakdowns and of malfunction conditions by performing simple functional verify operations , in the height respect of the actual safety and health norms.

The informations for the installation, the assemblage, the dismantlement, the handling, the adjustment, the reparation and the contingent accessories, devices etc. installation are destined - and then executable - always and in exclusive way from specialized warned and educated personnel, or directly from the TECHNICAL AUTHORIZED ASSISTANCE, in the height respect of the manufacturer recommendations and the actual safety and health norms.

The devices can't be used outside a box. The user must always insert the cards in a container that rispect the actual safety normative. The protection of this container is not threshold to the only atmospheric agents, but specially to mechanic, electric, magnetic, etc. ones.

To be on good terms with the products, is necessary guarantee legibility and conservation of the manual, also for future references. In case of deterioration or more easily for technical updates, consult the AUTHORIZED TECHNICAL ASSISTANCE directly.

To prevent problems during card utilization, it is a good practice to read carefully all the informations of this manual. After this reading, the user can use the general index and the alphabetical index, respectly at the begining and at the end of the manual, to find information in a faster and more easy way.

VERSIONS

The present handbook is reported to the **NSB8** version **1.2** and later. The validity of the bring informations is subordinate to the number of the firmware version. The user must always verify the correct correspondence among the two denotations. The version number is printed on the label attached on the diskette that contains the **NSB8** package; this is the only valid version number and it is not self displayed by the program.

GENERAL FEATURES

NSB8 is a powerful software development tool which allows high-level programming in BASIC on all the **grifo®** boards based on Z80 microprocessor family. The code produced by **NSB8** requires the functions and features of CP/M operating system or the similar rom-based operating system **GDOS 80**. The development environment is extremly friendly and achieves to reduce the development time, being anyway compliant to the operational feeling of all the BASICs. Unexperienced programmers will be able to take advantage of its numberouses commands and functions, becoming productive in few hours of work, while experienced programmers won't need any training. However the great code performance and the rapidity of hardware intervents make the NSB8 an unreplaceable work instrument for all the applications.

The interpreter supports mathematic functions, control applications, data base management, interfacing to generic consoles, operating system calls and many other features designed to solve industrial automation problems.

NSB8 is a programming and development environment made by a set of independent items that can be used or not by the programmer, without any limitation. Wishing to make comparisions amongst **NSB8** and other well-known BASIC programming tools, we detect that **NSB8** has an environment and instruction, command set comparable to the GWBASIC's one.

NSB8 enables to take the greatest advantage of the hardware resources from the boards you are using, because you may use them directly through the high-level instructions, with no need to develop specific firmware. For example, **NSB8** has the capacity to manage hardware resources like serial lines, printers, mass storage devices, operator interfaces, etc.

Amongst the many characteristics of this development environment, we remind:

- Numbered BASIC source code and automatic renumeration command.
- Standard syntax; it allows to reuse code written and already tested on other BASIC programming environments.
- Two different data types: floating point with 8 significant digits and string.
- Wide range of operators including mathematical, relational, logical.
- Complete set of mathematic functions including trigonometric and trascendental functions.
- Support for the scientific notation that uses the significant digits in the best way.
- Instruction set dedicated to the use of an operator interface. By means of these functions you may control the complete **QTP xxx** terrminals serie.
- Wide range of **GDOS 80** file system management instructions set. There is no more need for low-level memory and data area management. **GDOS 80** takes care of this by manipulating data files, which can be created, deleted, downloaded, etc.
- Interesting string manipulation instructions set (concatenation, conversion, test, etc.).
- Powerful control flow instructions set, which allows to perform iterations, single or multiple tests, define functions and procedure, etc.
- Basic low-level hardware resources management instructions set, like I/O instructions, direct memory access, absolute calls to external procedures etc.
- High level devices management instructions set, which, by means of **GDOS 80** features, allows easy use of peripherals like printers and serial lines.
- No license fee or overcharge, developers are free to create programs without even informing **grifo®**.

As the software packages is under continuous development, please take care of the presence of an eventual file called "READ.ME" in the work disk. This file reports additions, modifications and improvements applied to the package and not yet reported in the manual; if the file is present it must be read, printed and attached to this manual.

NSB8 REQUIREMENTS

Only three elements are required to be immediatly up and running:

1) A Z80 based control board like **GPC® 15A**,**GPC® 150**, **GPC® 15R**, **GPC® 153**, **GPC® 183**, **GPC® 154**, **GPC® 184**, etc.
2) The **GDOS 80** operating system for the desired control board or a CP/M emulator as Z80MU.
3) A personal computer, connected to the control board through a serial line.

NOTE
This manual uses the definition **target board** to refer to one of above reported **grifo®** hardware structures.

RECEIVED MATERIAL

The **NSB8** software package is made by two disks that contains the software structure needen to work:

NSB8 DISK
The content of this disk is organized in directories, whose structure recalls the software package structure; we suggest to copy all the files into an unique work directory, like **GDOS 80**'s one. Here follows a list of directories and files stored in this distribution disk:

Root
Contains all the most frequently used programs when operating with **NSB8**, which constitutes the main work structure:
NSB8.G80   ->   Executable **NSB8** basic interpreter for **GDOS 80** operating system.
LEGGIMI.ITA ->   Last minute addition to **NSB8** documentation (Italian version)
README.ENG ->   Last minute addition to **NSB8** documentation (English version)

UTILITY directory
Contains a set of utility programs and proper documentation files:
BASTRA.G80   ->   Reduce and optimize the application program
BASTRA.DOC ->   User documentation for BASTRA utility
BASYM.G80   ->   Create a table with the names of the simbols used in the application program
BASYM.DOC   ->   User documentation for BASYM utility

EXAMPLE directory
Contains a set of demostration programs that show how to use the features of **NSB8** programming language; this directory may include subdirectories containing demonstration programs for some cards, some defined hardware resources, etc.

NSB8 USER MANUAL DISK

It's the present manual and reports any technical information about **NSB8**. In detail this manual reports informations on instructions, commands, memory areas, program control, variables, etc.

HOW TO START

This chapter describes the operations to perform for a first, elementary use of the **NSB8** software package. The following list of operations refers to the descriptions reported in the previous chapters, so please read them carefully.

1) Install **GDOS 80** operating system and make experience on its use.

2) Copy into a work directory all the received disks contents. We suggest to use the **GDOS 80** directory as the **NSB8** work directory and to copy all the received files, even those saved on subdirectories, into this work directory.

3) Read completely the received documentation (use ACROBAT READER on files with .PDF extension and any text editor on files with .DOC or .ENG extensions).

4) Select the upper case (caps lock) on Your P.C. keyboard.

5) Run NSB8 programming language by typing:
    C:NSB8<CR>   ->   if You load **NSB8** from hard disk
    N:NSB8<CR>   ->   if You load **NSB8** from ROM disk
and wait for the following screen:
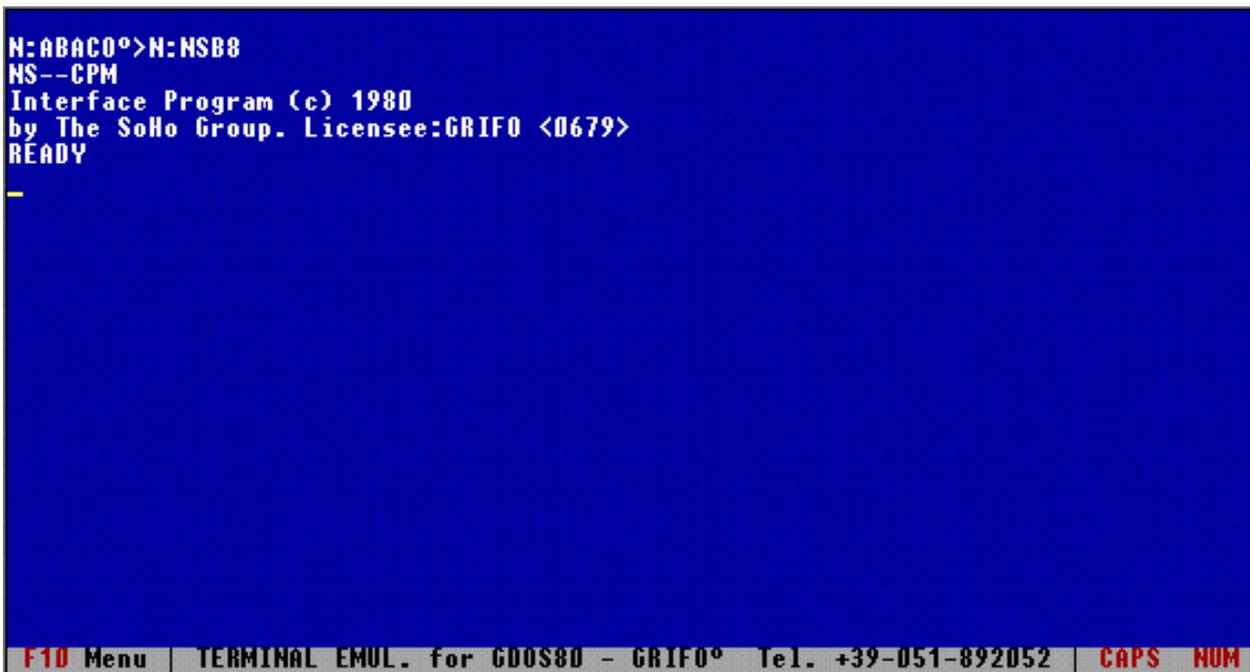
FIGURE 1: NSB8 START UP SCREEN

6) Type some **NSB8** commands like: SCR<CR>
                                          CAT3<CR>
                                          LIST<CR>
and some **NSB8** instructions like: PRINT "Hello world"<CR>
                                      PRINT 10/5,10/3,SQRT(100)<CR>
and check the results listed in the following screen:

FIGURE 2: FIRST COMMANDS AND INSTRUCTIONS SCREEN

7)　Load a tokenized demo program from hard disk, print its text and run it by typing:

LOAD DEMOT.B,3<CR>
LIST<CR>
RUN<CR>

FIGURE 3: LOAD AND RUN TOKENIZED DEMO PROGRAM SCREEN

8)　Delete previous program, load an ascii demo program from hard disk and run it by typing:

SCR<CR>
ALOAD<CR>
DEMOA◊◊◊　　　　　　　　　　　　　　　(◊ = space)
RUN<CR>

FIGURE 4: LOAD AND RUN ASCII DEMO PROGRAM SCREEN

9) Add the following two instructions lines at the ascii demo program DEMOA.SOH:
    15 PRINT "Program start"
    50 PRINT
    60 PRINT "Program stop"
By using the **GET80** integrated editor and save the obtained new source file.

10) Repeat point 8 and check the results of the previous addition.

11) At this point You are probably capable to use **NSB8**, so You can carry on testing all the instructions and all the provided demo programs.

Please remember that the **GET80** editor is really much more comfortable than **NSB8** editor, so we suggest to always use it for Your application program development.

ALOAD

COMMAND:

ALOAD

<file name> followed by a number of space up to a total of 8 characters

ACTION:

The ascii BASIC program contained in the specified file is loaded into the program/data area and becomes the current program.

EXAMPLES:

ALOAD

FILE NAME?: PROG2◊◊◊   (◊ = space)

ALOAD

FILE NAME?: TST◊◊◊◊   (◊ = space)

REMARKS:

The specified file must be an ascii file with an empty first line. The ascii format is really usefull, for example to print it on paper or to modify it with an external text editor (i.e. **GET80** one's).

Unlike LOAD command, the successful ALOAD command doesn't perform a scratch of the program/data area before loading the program.

ERROR MESSAGES:

TOO LARGE OR NO PROGRAM ERROR

HARD DISK ERROR

FILE ERROR

ARG ERROR

TYPE ERROR

See command LOAD and SAVE.

SEE ALSO:

Command ASCSAVE

Command LOAD

Command SCR

APPEND

COMMAND:
APPEND <file name>

ACTION:
Appends the tokenized basic program in the specified 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 sucessful.)

EXAMPLES:
APPEND MYPROG
APPEND TESTER,2

REMARKS:
If there is no current program, APPEND acts like LOAD. A sucessful 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 then 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 wich would result from the APPEND operation is too large to fit into available memory. In the latter case, the current program remains unmodified.

HARD DISK ERROR
FILE ERROR
ARG ERROR
TYPE ERROR
See command LOAD and SAVE.

ASCSAVE

COMMAND:
    ASCSAVE
    <file name> followed by a number of space up to a total of 8 characters

ACTION:
The current program is permanently saved into an ascii file on disk.

EXAMPLES:
    ASCSAVE
    FILE NAME:? PROG2◊◊◊   (◊ = space)
    ASCSAVE
    FILE NAME?: TST◊◊◊◊   (◊ = space)

REMARKS:
The ASCSAVE command is a special form of standard SAVE command and it can be used to save the current source program on a file with an ascii format. The ascii format is really usefull, for example to print it on paper or to modify it with an external text editor (i.e. GET80 one's).

ERROR MESSAGES:
ARG ERROR
TYPE ERROR
See command SAVE.

SEE ALSO:
Command SAVE
Command ALOAD

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 increlented 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 **NSB8** line editing capabilities to delete
       the line number from the beginning of the line);
    c) the next automatically generated line number would be greater then 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 grater than 65535 or less than 0.

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

SEE ALSO:

Command REN

BYE

COMMAND:

BYE

ACTION:

The current session with BASIC is terminated, and control returns to the **GDOS**.

EXAMPLE:

BYE

REMARKS:

The BYE command does not affect BASIC's program/data area in any way, so 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:

Section SPECIAL ENTRY POINTS

CAT

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 operating system DIR command. 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:
GDOS 80 user manual.

CONT

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 or 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:
Section CONTROL-C: THE PANIC BUTTON
Statement STOP

DEL

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 <CR> 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:
Command SCR

LIST

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 egual to the first.
For the convenience of users with CRT screens, the program listing may automatically be "paged". Refer to section 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.

LOAD

COMMAND:

LOAD <file name>

ACTION:

The tokenized BASIC program contained in the specified file is loaded into the program/data area and becomes the current program.

EXAMPLES:

LOAD PROG3.B   {load file PROG3.B from drive #1}
LOAD TEST8.B,2   {load file TEST8.B from drive #2}

REMARKS:

The specified file must be of tokenized type, or in other words composed by valid instruction codes. 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 section 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.

FILE ERROR
ARG ERROR
TYPE ERROR

See command SAVE. If an attempted LOAD results in any of these 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.

SEE ALSO:

Command SAVE
Command SCR

MEMSET

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.

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
This error occurs because of one of the following four reasons:
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:
Section PERSONALIZING BASIC

PSIZE

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 requires to store the current program. This figure is helpfull in creating command files, together with the **GDOS 80** SAVE command.

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

ERROR MESSAGES:

None.

SEE ALSO:

Command SAVE

Appendix A: GDOS 80 COMMAND FILE CREATION

REN

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   the 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 10.
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

RUN

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

SAVE

COMMAND:
     SAVE <file name>

ACTION:
The current program is permanently saved into a tokenized BASIC program file on disk.

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

REMARKS:
The specified disk must have sufficient free 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 tokenized BASIC program file.

HARD DISK ERROR
Refer to command CAT.

SEE ALSO:
Command ASCSAVE
Command LOAD
Command ALOAD
GDOS 80 user manual

SCR

COMMAND:

   SCR

ACTION:

SCR erases (scratches) the current program and any existing variables from the user workspace.

EXAMPLES:

   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 theprogram on diskette before using SCR, if that program, or parts of it, will be used later.

ERROR MESSAGES:

None.

SEE ALSO:

Command SAVE
Command ASCSAVE
Command LOAD
Command ALOAD
Command DEL

CHAIN

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 tokenized BASIC program file name, as described in section FILES.

EXAMPLES:

     10 CHAIN "PROG.B,2"
     100 CHAIN P\$ + D\$
     200 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.

Bacause CHIAN 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:

Section CHAINING
Command LOAD
Command RUN

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
Section FILES

DATA

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

DEF

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 letter FN followed by a string or numeric variable name (such as FNA$, FNQ7, etc.)

EXAMPLE:

Single line

70 DEF FNH(X,Y)=SQRT((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 forwared to the first statement after the definition. Function definitions may be located anywhere in the program text. Function definitions occurs before program execution begins.

ERROR MESSAGE:

FUNCTION DEF ERROR

An (apparently) single line function was defined improperly, or an attempt was made to define function within the definition of a multi line function.

SEE ALSO:

Section USER FUNCTION

Statement RETURN

Statement FNEND

DIM

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 lenght 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 section 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:
Section USING STRINGS
Section USING ARRAYS

END

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
Section THE FOR NEXT LOOP

ERRSET

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 software 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; 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 section 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:
Section ERROR TRAPPING
Section ERROR MESSAGES
Section PERSONALIZING BASIC
Section CONTROL-C, THE PANIC BUTTON

EXIT

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 section 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
OUT OF BOUNDS ERROR
See statement GOTO

SEE ALSO:
Section THE FOR NEXT LOOP
Statement NEXT
Statement FOR
Statement GOTO

FILL

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*16^0) ,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. **NSB8** does not accept hexadecimal (base 16) numbers. If you wish to use hexadecimal form when specifying addresses of byte values, you should make use of a hex to decimal conversion function.
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 module 256 = 1: FILL X, 257 would put a byte with value 1 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 GDOS, BASIC, or the program/data area. Thus, FILL gives the programmer power to make some very bad mistakes.

ERROR MESSAGES:
OUT OF BOUND ERROR
This error occurs because of one of the following four reasons:
    1) The byte value or the memory address (or both) is less than zero.
    2) The memory address is greater than 65535.

SEE ALSO:

Section PERSONALIZING BASIC
Section USER FUNCTIONS
Section MACHINE LANGUAGE SUBROUTINES

FNEND

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:

Section USER FUNCTIONS
Statement DEF
Statement RETURN

FOR

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 section 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:

Section FOR NEXT LOOP

Statement NEXT

Statement EXIT

GOSUB

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, elesewhere 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 (remember that the location of the return point takes memory space).

ERROR MESSAGES:
LINE NUMBER ERROR
OUT OF BOUNDS ERROR
See statement GOTO

SEE ALSO:
Statement RETURN
Statement GOTO
Section SUBROUTINES

GOTO

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 erroneos line is typed!).

SEE ALSO:
Section EXECUTION AND FLOW CONTROL
Statement EXIT
Statement ON ... GOTO

IF

STATEMENT:

IF <logical expression> THEN <statement>

IF <logical expression> 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 statement 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 misformation of the logical expression. Check the section on the appropriate type of statement or feature to track down tha cause of each individual error.

SEE ALSO:

Section USING NUMBERS

Statemen GOTO

INPUT

STATEMENT:

INPUT <list of variables>
INPUT <string constant>, <var. list>
INPUT #<device expression>, <var. list>
INPUT #<device expression>, <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 <CR> 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 <CR> 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 ures, 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 INPUT1 statements must be used (see statement: INPUT1).
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 strinking the homonymous key.) After <CR> 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 <CR> is struck, BASIC will type a double question mark

(??) as auxilliary prompt, and await further INPUT for the given varianble list. It will repeat this step as long as necesary until all variables naled 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:
LENGHT 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:
Section USING NUMBERS
Section CONTROL-C, THE PANIC BUTTON
Section FUNCTIONS (in detail the built in INP function)
Statement INPUT1

INPUT1

STATEMENT:

      INPUT1 <list of variables>
      INPUT1 <string constant>, <var. list>
      INPUT1 #<device expression>, <var. list>
      INPUT1 #<device expression>, <string const.>, <var. list>

ACTION:

Exactly the same as INPUT statement, except that when the user strikes the <CR> key to terminate an input line, no carriage return is echoed to the terminal. Subseguent input or output will occur on the same line.

EXAMPLES:

      50 INPUT1 Z, W, B7, A(3)
      25 INPUT1 #D (Q), "GUESS?", G

REMARKS:

See statement INPUT

ERROR MESSAGES:

See statement INPUT

LET

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 eexpression on the right hand site 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$ ("145-549-0858")
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 the line 10 above, the apparent mathematical impossibility of X=X+1. However, as an assignment, this make sense: the right hand expression is evaluated with the current value of X, and the result obtained then become 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 assignement if one resists reading LET statements as "Q gets Q+1", or "Z becomes M+173".)

Only single variable names are legal on the left sode 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 at 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:

Section USING NUMBERS
Section USING STRINGS
Section USING ARRAYS

LINE

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

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 beceuse 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, some video display devices (QTP xxx) provide for 20 or 40 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 carriare 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 section 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:
Section FUNCTIONS (in detail the built in TAB function)
Statement INPUT
Statement INPUT1
Section PERSONALIZING BASIC

NEXT

STATEMENT:
>   NEXT
>   NEXT <numeric variable>

ACTION:
Terminates execution of the loop which starts with the matching FOR statement. For a complete description of this loop, see section 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 NSB8, 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 the 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
Section FOR NEXT LOOP

ON

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
OUT OF BOUNDS ERROR

See statement GOTO

SEE ALSO:

Statement GOTO

OPEN

STATEMENT:

    OPEN #<file number expression>, <file name>
    OPEN #<file number expression>, <file name>, <size variable>
    OPEN #<file number expression> %<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 BASIC data type 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 section 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 #17%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 (data type, 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:

Section DATA FILES
Statement CLOSE

OUT

STATEMENT:

OUT <port number>, <byte value>

ACTION:

The byte value is sent to the indicated 8080 or Z80 output port. Both port number and byte values must be numeric expression 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:

Section FUNCIONS (in detail built in INP function)
Statement FILL

PRINT

STATEMENT:

PRINT

PRINT <list of string and/or numeric expression>

PRINT #<device expression>

PRINT #<device expression>,<list of string and/or numeric expression>

ACTION:

The data indicated in theoutput 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 Section 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 keyboard 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 section FUNCTIONS (in details built-in TAB).

SEE ALSO:
Section FORMATTED PRINTING
Section MULTIPLE I/O DEVICES
Statement LINE

READ#

STATEMENT:

READ #<file number expression>, <variable list>
READ #<file number expression> %<random address>, <variable 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 percent 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:
Section DATA FILES
Statement WRITE#

READ

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 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 noo 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 or set to end of data. After this happens, should a READ be attemped, 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 ad the variable to which it was to be assigned.

SEE ALSO:
Statement DATA
Statement RESTORE

REM

STATEMENT:

     REM <optional line of any text>

ACTION:

      None. REM statement are ignored by BASIC.

EXAMPLES:

     10 REM THE REM STATEMNT IS USED TO
     20 REM INSERT COMENTS 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 is ignored by BASIC. Therefore, any statements which appear after a REM on the same line will not be executed.

As with other **NSB8** statements, the characters ":", ";", "[" and "]" are traslated to "\", ",", "(" and ")", respectively , within REM text.

ERROR MESSAGES:

None.

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 tha 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 at statement GOTO

SEE ALSO:
Statement READ
Statement DATA

RETURN

STATEMENT:
RETURN

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

EXAMPLE:
1099 RETURN

REMARKS:
There are two versions of the RETURN statement in **NSB8**. This version is for use with subroutine only. Another is used wit user funtions. See section USER 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
Section SUBROUTINE

RETURN

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.

EXAMPLE:
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
There are two versions of the RETURN statement in **NSB8**. This version is for use with subroutine only. Another is used wit user funtions. See section USER FUNCTIONS, for details on that version of RETURN.

ERROR MESSAGES:
SYNTAX ERROR
The return expression doesn't match the function type.

SEE ALSO:
Statement FNEND
Statement DEF
Statement RETURN
Section USER FUNCTION

STOP

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 time which LET and PRINT may be used as direct statements in ordr 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 interrruption will retain the new values as the program resues.

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 to not change any of the program text (edit, insert, or delete program lines) during the interim. If you do, CONT will not work, and you wil forced to RUN the program alla over again.

ERROR MESSAGES:

None.

SEE ALSO:

Statement END
Command CONT
Section CONTROL C, THE PANIC BUTTON
Section SOME BASIC CONCEPTS

WRITE

STATEMENT:

WRITE #<file number>, <expression list>

WRITE #<file number> %<random address>, <expression 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 and 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 endmark by using the reserved word NOENMARK 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 expression, so long as the file number evaluates to an interger from 0 to 7 (corresponding to an opened file), and the random address is an interger between 0 an the last byte address in the file. Any numeric expression in the expressionlist may be prefixed with an ampersand (&) character. This signals BASIC to convert the vale to a single byteand WRITE in to the file (any value so prefixed must evaluate to an interger 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 writting 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 occured:

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:
Section FILES
Statement READ#
Statement OPEN
Statement CLOSE
Section IMPLEMENTATION NOTES

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 card whenever you hold down the control key then press the C key at the same time on your PC.

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

<div align="center">

STOP IN LINE XXXXX

</div>

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 completed, and the message

<div align="center">

STOP

</div>

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 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> command. This is accomplished by changing certaine internal data in the BASIC interpreter itself, a procedure described in section 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 you program is fully debugged.

SEE ALSO:
Command CONT
Statement STOP
Section SOME BASIC CONCEPTS
Section PERSONALIZING BASIC

MULTIPLE I/O DEVICES

A system may iclude several input/output (I/O) devices, such as a video terminal, printer, graphics display, etc. **NSB8** provides a convenient means for BASIC programs to make use of up to 8 separate I/O devices. A unique integer number 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 the terminal emulation window of **GET80**. When your **GDOS 80** 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 statements.

A PRINT, INPUT1 or LINE statement accomodates an optional device expression, which consist 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 divice 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 display. the following program causes a different message to be printed on each of the three devides:

```
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 REMOTE PRINTER"
50 PRINT #2, "THIS SHOWS UP ON THE REMOTE DISPLAY".
```

The PRINT/INPUT device expression, characterized by a cross hatch, should not be confused with begins with a percent sign (%).

SEE ALSO:
Statement PRINT
Statement INPUT
Statement INPUT1
GDOS 80 user manual

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 inludes 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 0, write COS(0). 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(0) 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 **NSB8** 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=2*SQRT(100)

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 section USING STRINGS). COS(2,3) causes a SYNTAX ERROR because the COS function wants only one numeric argument.

The following pages contain a list and description of all the functions built into **NSB8**. 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 section describes the value represented by the function as well as how the arguments relate to that value.

MATHEMATIC FUNCTIONS

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.

STRING FUNCTIONS

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$="" then LEN(A$) 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 (<sring 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. Note that CHR$ and ASC are inverse functions.
>   ASC("B")=66, ASC("CLUNK")=67

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. Note that if any non numeric characters follow the numeric constant which is at beginning of the string expression, they will be ignored.
>   VAL("123") = (the number) 123, VAL("00000")=0, VAL("abcde"), VAL(" ") and VAL(" ")
>   will cause errors, VAL("123XYZ")=123, 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 section PRINT FORMATTING for further details.

SPECIALIZED INPUT OUTPUT FUNCTIONS

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 section 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:

```
10 T$=INCHAR$(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.

        CIO=INP(128)

<status>=CALL(17200,<acquire>)
This function checks if the console input device has a character ready and it manages its acquisition according with <acquire> numeric parameter. The result of this check is returned in the numeric variable <status>, with the following meaning:

        <status> = 0   ->   no character available;
        <status> > 255 ->   one character available; in this condition if:
                                 <acquire> = 0 -> the available character is not acquired;
                                 <acquire> = 1 -> the available character is acquired and returned on
                                                  the <status> low byte.

To correctly use this function, the special keys management must be disabled (see discussion: CONTROL-C, THE PANIC BUTTON).
Example:

        10 IF CALL(17200,0)=0 THEN GOTO 10 \ REM Wait one character with no acquisition
        20 IF CALL(17200,1)=0 THEN GOTO 20 \ REM Wait one character with acquisistion
        30 T=CALL(17200,1) \ REM Check if character available and acquire it
        40 IF T>255 THEN T=T-256 \ REM If character available, calculates its numeric code

<not used>=CALL(17152,<I/O add.>)
This function define the 16 bit I/O address for the other input output functions. The <I/O add.> parameter should evaluate to an integer from 0 to 65535, while the returned value is not significant.

        10 H=CALL(17152,12) \ REM Sets I/O address = 12 = 000CH

<data in>=CALL(17168)
This function perform an input operation, at the 16 bit address specified by the previous function, and it returns the acquired value (an integer from 0 to 255) on the numeric variable <data in>.

        20 I=CALL(17168) \ REM Get input at 000CH address and saves it on I variable

<not used>=CALL(17184,<data out>)
This function perform an output operation, at the 16 bit address defined by the first function, with the value of the numeric variable <data out> (an integer from 0 to 255).

        30 H=CALL(17184,0) \ REM Performs output at 000CH address with the value 0

DISK FILES FUNCTIONS

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 section 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 section 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.B" is the name of a BASIC program file on a disk in drive 2, then FILE ("ABC.B,2") will return the value 2.

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, when 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 section MACHINE LANGUAGE SUBROUTINES.

USER FUNCTIONS

Functions may be written in **NSB8** 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 Pth 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 includen 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.

      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 didrect 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
Statement FNEND

USING NUMBERS

This section describes numbers and how to use them in conjunction with the standard version of NSB8. Those with non standard version of BASIC should read the section called 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:

        0   347   -33.333   .00176   1.003
        .1   -8   123.4567   -.3   0.2

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 unconfortably long. Here are the same numbers as in the examples above, but written in scientific notation:

        0E+00   3.47E+02   -3.3333E+01   1.76E-03   1.003E+00
        1E-01   8E+00   1.234567E+02   -3E-01   2E-01

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-03 would be read as "1.76 times 10 to the power of -3"

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 **NSB8** consists of a single capital letter, or a single capital letter followed by a single digit from 0 to 9. Here are some legal **NSB8** variable names:

        A   B7   C3   Z   Q   N8   P0

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

PRECISION

Numbers in the standard version of **NSB8** are stored with 8 digits precision. Other precisions are available, see section NON STANDARD VERSION OF BASIC for details. **NSB8** uses the most accurate form of microcomputer arithmetic available: binary coded decimal (BCD), see section COMPATIBILITY WITH OTHER BASICS. All arithmetic operations are rounded to 8 digits in the standard version of **NSB8**, e.g., the sum of .12345678 and .011111111 would be rounded to .13456789, since .134567891 requires 9 digits.

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 local number of digits is now 8. It is impossible to get a more accurate representation of 2/3 in non standard **NSB8**. The fraction 1/2, on the other hand, needs only a single digit (.5) to represent it exactly!

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. **NSB8** 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).

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.

VERY SMALL NUMBER

The number .00000000123 will not be rounded by **NSB8**, but .00000000123456789 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 1.0000000012345679E+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 digits) precision **NSB8** 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.

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
^	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 then 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 **NSB8**:

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, while 0 will be treated as false. The result of a boolean operation is either true or false. 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 an expression (see sections FUNCTION and USING ARRAYS for complete details concerning function calls and array element names. These are two advanced features of **NSB8** 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, -   (negates a number, unary minus)
   ^   (exponentiation)
   * , /   (multiplication and division)
   + , -   (addition and subtraction)
   =,<,>,<>,<=,>=   (relationals)
   AND   (boolean sum)
   OR   (boolean multiplication)

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.

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 evaluates 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 coexist 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
```

50   PRINT \ REM Print <CR> before starting next row
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 dimension 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 automatically initialized so that all of its elements contain the value 0.

ARRAYS MAY NOT BE REDIMENSIONED

No matter how created, either by an explicit declaration in a DIM statement or automatically, by BASIC, no array may be redimensioned 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 redimension an existing array will result in a DIMENSION ERROR.

ARRAY REFERENCES IN NUMERIC EXPRESSIONS

As mentioned in the section 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:
Section USING NUMBERS
Statement DIM

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 (0-9), 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 redimensioned 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 string 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 yeld 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 CONCATENATION

The concatenation operation may be performed on strings, 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 tigether. For example, "CAR"+"LOAD" represents the same value as "CARLOAD". Any string value may be concatenated with any other string value to yeld a third value which consists of the two linked together. A$(2,3)+A$(2) yelds 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 YELD"

which gives the value "ACCEDE MEANS YELD".

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 section 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. 1) Two string values are equal only if they have the same number of characters, and have matching characters in each character position.
2. 2) Strings are compared character by character, left to right, until a difference occurs or one of the strings ends.
3. 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. 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. 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 comparison 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 a standard table of ascii codes 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 automaticstring 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$)). where LEN(L$) 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 AND 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 **NSB8** 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$ 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:
Section FUNCTIONS
Section USING NUMBERS (EXPRESSIONS)
Statement DIM
Ascii table

FORMATTED PRINTING

NOTE:
Read section 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
    -8.03
    -.04

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 the two digits of the exponent itself (the first digit may be 0). Here are some numbers in E format:

    1.4073749E+14
    -2E-09
    -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 the 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 percent 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 with. 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 accomodate 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 percent 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 nth (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   (The field is 7 positions wide)
            51.00
           987.12
          1234.56

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 31% 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 AND 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. There after, 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 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 section FUNCTIONS, in detail the 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 yeld 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

SEE ALSO:

Statement PRINT
Section MULTIPLE I/O DEVICES
Section USING NUMBERS

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 ususally 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 certainsituations, 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.

SEE ALSO:
Statement GOTO
Statement IF
Statement ON
Statement STOP
Statement END
Section THE FOR NEXT LOOP
Section PROCEDURES
Section FUNCTIONS

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 sections on EXIT).

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 #1, 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 caled 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 chenge in I it will have been repeat 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:

```
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 be 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 on clarifying the program text (determining which NEXT goes with which FOR). If the optional control variable is specified on the NEXT statement, BASIC will perform a syntax check during program execution and will cause a program error of 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 performaded if an EXIT statement is executed. EXIT is used to transfer program control to 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 elements array A.
        20 REM The following searches A from element 1 to 10 for the first non zero element.
        30 REM The index of this element will be N.
        40 REM If all elements are 0, N will also be 0.
        50 REM A FOR loop is used for the scan,and EXIT stops scan if non zero found.
        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 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 30 in the inner loop of a two deep nest (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

        10 FOR I=0 TO 9
        20   FOR J=0 TO 9
        30   PRINT I, J
        35   EXIT 45
        40   NEXT
        45   EXIT 600
        50 NEXT

SEE ALSO:
Statement FOR
Statement NEXT
Statement EXIT

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 (Y) or no (N) answer in A$.
20 INPUT "PLEASE ANSWER YES OR NO: ",A$
30 IF A$=" " THEN 20 \ REM No answer given.
40 A$=A$ (1,1)
50 IF A$="Y" THEN 70 \ REM OK answer
60 IF A$<>"N" THEN 20 \ REM Not = Y or N either.
70 REM At this point, answer 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 rexecute 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.

NSB8 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 (Y) or no (N) answer in A$.
1020 INPUT "PLEASE ANSWER YES OR NO: ",A$
1030 IF A$=" " THEN 1020 \ REM No answer given.
1040 A$=A$(1,1) \ REM Examine 1st char only.
1050 IF A$="Y" THEN 1070 \ REM OK answer.
1060 IF A$<>"N" THEN 1020 \ REM Not = Yor N 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.

SEE ALSO:
Statement GOSUB
Statement RETURN
Section USER FUNCTION

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. All this file information is stored in a special place on the diskette called the directory. You can list the name information for each file on diskette by using the CAT command. Each catalog listing is of the following format:

NAME1	EXT1	NAME2	EXT2	NAME3	EXT3
NAME4	EXT4	NAME5	EXT5	NAME6	EXT6
:	:	:	:	:	:
NAMEn	EXTn				

For example, the listing

PROG1	B	DEMOA	SOH

denotes the presence of two files named PROG1.B and DEMOA.SOH on the current directory of the current disk.

FILE NAMES

The name of a file consists of two series of printable characters (the printable characters include the upper and lower case alphabets, the digits 0 to 9, and some punctuation symbols).The first group is the real name and it is not more than 6 characters length, while the second group is the file extension, that is not more than 3 characters length. Generally the file extension give information to the user about the file contents, in fact the following standard extension are used:

B	->	Tokenized **NSB8** source file
SOH	->	Ascii **NSB8** source file
DOC	->	Documentation file
TXT	->	Text file
G80	->	**GDOS 80** executable file
EXE	->	MS-DOS executable file
	etc.	

Any characters may be used in any order, with the exception of the space, the comma and the point. 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.B" is on the diskette in drive #2, the proper way to write its name is "PROG.B,2". File "POP.B" in drive #3 would be called "POP.B,3'. If no suffix is given, then the system assumes that the file is on the diskette in drive #1 or current selected drive. The file names "SYNCRO" and "SYNCRO,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 and extension on diskette, but also the drive in which it is located. Thus, a complete file name consists of an actual name of no more than 6+3 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 128 bytes of information. Each file in **NSB8** occupies a portion of disk storage. A file may be any number of file blocks in length, provided that there is sufficient free storage space for it on the diskette. **NSB8** has no functions nor statements to obtain the file size, so if this information is necessary it is preferible to save it directly into the file.

FILE TYPES

Every file has a type, which can be used to classify a file according to how it is used. For example, the **NSB8** convention is that a type 2 file always holds a program written in BASIC tokenized. 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 0 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 **NSB8** allow you to determine a file's type when accessing or creating it.

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 "STOR" is opened as file #2. Then, all BASIC statements in your program which are intended to access "STOR" 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 disk.
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 file's 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 0 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

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 statements READ# and 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 chapter 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 data file of unknown structure.
20 REM Assume no string bigger than 500 chars.
30 DIM S$(500),F$(10)
40 REM F$ will hold file name, S$ will hold string values read and 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. If not, string is next.
140 REM READ and PRINT string.
150 READ #1,S$
160 PRINT S$
170 REM Go back for more data.
180 GOTO 90
190 REM READ and 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 reopened 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 and READ them back again.
30 REM First, WRITE them!
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 OPEN  DAT7 file with file number 1.
30 OPEN #1,"DAT7"
40 REM First byte goes into X.
50 READ #1, &X
60 REM Byte value 65 goes to file #1.
70 WRITE #1,&65
80 CLOSE modified file
90 CLOSE #1
100 END
```

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 access. 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 fata 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 file length value in bytes. 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 section 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 percent 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 $(1-1) * 42 = 0$. 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 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#
Section FUNCTIONS (in detail the built in TYP, FILE functions)

MACHINE LANGUAGE SUBROUTINES

NSB8 provides a method through wich 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 operating system, 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 functions 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 comunicated to your routine in the D and E registers 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 routine in fact 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, whic 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 and L registers pair whenever the machine language routine terminates. Thus, it is possible to communicate a single numeric value to your subroutine 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 supposethere exists a machine language routine at address 6000, and that it will require the optional argument value. The foolowing line effects a transfer to that routine, passing the value of variable A as argument in the D E registers as a positive, 16 bits binary integer:

        10 Q=CALL(6000,A)

If, in this istance, the binary value of 578 is in the H L registers pair when the machine language routine returns, then the variable Q will be set to 578 when BASIC resume 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 instance of CALL, no specific argument value is sent to the machine language routine in the D E registers 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 routine correctly is difficult and should only be attempted by experienced programmers and only then if no other alternative is available.

SEE ALSO:
Statement FILL
Section FUNCTION
Section SPECIALIZED INPUT OUTPUT FUNCTIONS

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 console. Thus, a sequence of programs may operate virtually unattended for long periods (uless, 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 programa 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 bettween 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 moremodules 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 wich 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 name, calendar items, switchboard messages, products informations, parameters, 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 programs 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 many good techniques for utilizing RAM memory in this way, most involve the use of EXAM function and the fill statement.

TESTING 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 disk to be certain that it is in perfect condition, but the built in file functions 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 situation.

SEE ALSO:
Statements CHAIN, READ#, WRITE#, FILL, ERRSET
Section FUNCTIONS (in detail the built in EXAM, FILE functions)
Section ERROR TRAPPING AND RECOVERY

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 rexecute 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, **NSB8** 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 execution of 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 reinsert 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. **NSB8** program errors and their codes are listed in appendix B.

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 time 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 section CONTROL-C, and section 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
Section CONTROL-C, THE PANIC BUTTON
Apeendix B: ERROR MESSAGES

THE LINE EDITOR

Never forget that **NSB8** is capable to load and save ascii source BASIC program that can be edited through an external ascii editor. So if you retain that the internal editor is poor and uncomfortable you are free to use your favourite editor ensuring that the ascii source file length is a multiple of 128 bytes (use **GDOS 80** program utility **DOS2GDOS** or the **GET80** integrated editor).

NSB8 INTERNAL EDITOR

Anyone who has used the **NSB8** 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 **NSB8**. 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 <CR>, 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, **NSB8** 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 <CR> 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 <CR>, 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 <CR> 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 <CR>.

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 <CR> 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 <CR>! Now strike <control-G>, you should see the following on your terminal:

        20 PRINT "TOTAL RECEIPTS TO DATE: " ,T2

If you strike <CR>, a new line 20 will be added to you 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 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 can not 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 <CR> 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 discussed 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 <CR> 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 start 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
```

in the following paragraphs there 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 FOM 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 FOM 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 <CR> 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 <CR>.

CONTROL-N: CANCEL AND REEDIT 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:
Section HOW TO START
Section COMPATIBILITY WITH OTHER BASIC

COMPATIBILITY WITH OTHER BASICS

This section provides some information which may be useful to you if you are attempting to convert programs into **NSB8** 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. Section USING STRINGS details the system implemented in **NSB8**, 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 **NSB8** conventions. In general:

OTHER BASICS		NSB8
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)

ARRAY OF STRINGS

Some versions of BASIC implement arrays of strings with the syntax which is used for substring referencing in **NSB8**. An array of strings may be achieved in **NSB8** 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 **NSB8**, 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 **NSB8**, are automatically translated into other characters. This is done to help minimize the effort of converting programs written for other BASIC systems into **NSB8**. This conversion is not performed upon text within quoted strings. The following chart summarizes the translation process.

	becomes	
[	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"

BCD ARITHMETIC

NSB8 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), **NSB8**'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 **NSB8**'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 **NSB8** when the IF statement precedes others on a multiple statement program line. In **NSB8**, 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 \ B=0
    20 IF A<>0 THEN A=7 \ B=7
    30 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 20 so that the output becomes 0. With these other BASICs, execution always skips to the following program line when the condition is FALSE. The remainder of the line, if any, is executed only when the condition evaluates to TRUE.

SPECIAL ENTRY POINTS

NOTE:
The following discussion concerns advanced topics and presupposes a working knowledge of the operating system 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 reentry to BASIC via each. The abbreviation ORG stands for the starting address of your BASIC; for those whose BASIC starts at E00H, as standard NSB8 version does, the actual entry point addresses are given in parentheses next to the general models.

ORG + 00H   (E00H)

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 default entry point used by the **NSB8**<CR> command in the **GDOS 80**.

ORG + 04H   (E04H)

Any previously existing program is retained, but any variables and/or other data associated with it are erased.

ORG + 14H   (E14H)

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 operating system, reenter 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 operating system causes no change in BASIC's memory region).

SEE ALSO:
Section PERSONALIZING BASIC

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 an operating system 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 **GDOS 80** reside should be tested thoroughly.

B)   At this point, you should make sure that the **GDOS 80** is operational, and that you are in its command mode (signified by the **GDOS 80** prompt). Now, run the original **NSB8** saved in the received disk, by typing:

       NSB8<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 substeps of the following paragraphs 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.

D)   Type BYE in order to return to the **GDOS 80** and save the modified version of BASIC to a new file on one of the managed drives. This operation is performed through the following **GDOS** command:

       SAVE <size of BASIC file in 256 bytes blocks> <drive:filename><CR>

       With the standard version, then the above simplifies to the following actual command:

       SAVE 43 C:NSB8MOD.G80<CR>

E)   Now type:

       C:NSB8MOD<CR>

       to test your personalized copy and make sure that all the modifications have been made correctly. If not, get back into **GDOS 80** and return to step A. The new copy of BASIC may now be used as your personalized master copy, and it is a good practise to make a back up copy for example on write protect disk.

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 65536 bytes of working memory. The **GDOS 80** and BASIC itself take up a large part of this. In order for you to write and RUN reasonably large programs, you must have more memory beyond the 16384 bytes (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+09H and ORG+0AH, where ORG is BASIC's origin (starting address) in your system, usually 0E00H. In the standard version of BASIC, then, these addresses are 0E09H and 0E0AH, respectively. The standard default high address for the program/data area is E3FFH.

2 SETTING A VARIABLE TO BASIC 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 0E00H in hexadecimal, then use 3584 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=3584

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 40, for instance, then type

    L=40

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 <CR> 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. 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 reenable 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 digits hex number FC and convert it to its decimal equivalent, 252. 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 PROGRAM AUTOSTART

For some applications, you may wish to automatically start the program execution without manually loading and running it. This possibility is really useful especially when a control program is installed on embedded systems without a human presence. For detailed information on autostart procedure, please refer to proper appendix A: COMMAND FILE FOR GDOS 80.
To enable the autostart feature, make sure that S is set to the starting address of your BASIC and type

    FILL S+15, 0

To disable the feature, type

    FILL S+15, 1

The standard copy of BASIC assumes that autostart is disabled.

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 3584 (0E00H), the actual addresses in decimal and hex are also given.

ORG+6 & ORG+7 (3590 & 3591 or 0E06 & 0E07)   [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 (3593 & 3594 or OE09H & 0E0AH)   HIGHMEM]
Contains lower and upper bytes, respectively, of highest address in RAM which BASIC may use for program/data area. Standard value: 255 and 227 respectively (corresponding to E3FFH).

ORG+14 (3598 or 0E0EH)   [LINECT]
Initial line length. Standard value: 80

ORG+15 (3599 or 0E0FH)   [AUTOST]
Controls turnkey autostart. Zero byte means autostart engaged. Standard value: 1 (turnkey operation required).

ORG+16 (3600 or 0E10H)   [BOOTPR]
Corresponds to first two hex digits in bootstrap PROM address for your system. Standard value: 232 (E8H)

ORG+19 (3603 or 0E13H)   [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 (3607 or 0E17H)   [DELECHO]
Character to be echoed in response to a single character deletion. Standard value: 08 (corresponds to backspace character).

ORG+24 (3608 or 0E18H)   [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 that enables interruption management.

NON STANDARD VERSIONS OF BASIC

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 USING NUMBERS, SPECIAL ENTRY POINTS and PERSONALIZING BASIC.

ABOUT NON STANDARD VERSIONS OF BASIC

The standard version of BASIC begins at address 3584 (0E00H) 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 going 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.
Any combination of these three options (different origin and different precision) 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 relocated 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 **GDOS 80** or reserved memory areas. 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 relocating BASIC into these areas.

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.

IMPLEMENTATION NOTES

This appendix is designed to provide important details concerning some of the internal workings of **NSB8**, 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 performs 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 digits BASIC, for example, will be smaller in size than that of a number written by 10 digits 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 followa:

first byte: bits 7÷4 =   most significant 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 bits 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 128 bytes is reserved when opening a file. 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.

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 bytes 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.

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 (named token) 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). When the program is listed, the comaction 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 memeory space than REMs of equivalent length which contain no embedded keywords. For example

>   REM FOR THE NEXT ORIGIN, LET TRY 2000H

will be compacted 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.

THE INTERNAL FORM OF A PROGRAM

In RAM and on disk, a program is represented as a series of program llines which have been converted to the tokenized 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 tokenized 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 maintaings 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.

ERROR MESSAGE

This section lists all the possible error messages printed by **NSB8**. For any errors which are trappable using the ERRSET statement, the error number is given in parentheses after the error message. The brief discussion of the general causes of each error is intended in large part to supplement the ERROR MESSAGE descriptions as given in the exposition sections for each language feature (and as also occasionally treated in DISCUSSION sections).

ARG ERROR   **(1)**
An attempt has been made to give an invalid argument to a command or function.

ARG MISMATCH ERROR   **(13)**
The number of arguments in the call for a user defined function does not match the number of parameters for that function.

CONTINUE ERROR   **(non trappable)**
An illegal attempt has been made to continue program execution. Program execution may not be continued if the previous execution stopped on an error, if any editing of the current program has taken place during an interruption, or if the program has executed an END statement.

CONTROL STACK ERROR   **(non-trappable)**
This error occurs when there is improper nesting of FOR and NEXT statements, GOSUB and RETURN statements, or multi line user function calls and RETURN statements. It also occurs when a FOR statement is the last statement in program.

DIMENSION ERROR   **(2)**
An attempt has been made to redimension an array or string, or to use the dimension statement in some other, illegal, fashion.

DIVIDE ZERO ERROR   **(9)**
An attempt has been made to divide by zero.

DOUBLE DEF ERROR   **(non trappable)**
There exists more than one definition for the same user function in the same program. Functions are defined at RUN time, so this message will occur before program execution actually begins.

FILE ERROR   **(7)**
The program is trying to access a diskette file which doesn't exist or is of incorrect type. This error will also 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 0 or greater than 7, or when a file is begin opened, but the file number specified is already in use. 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 encontered 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 **GDOS 80** 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 section STATEMENTS in appendix B 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 improperly 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 cincatenated 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   (4)

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 **NSB8**. (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.

APPENDIX A: COMMAND FILE FOR GDOS 80

If you need to create a command file (a directly executable program) for the **GDOS 80** operating system, starting from a source program written for **NSB8**, the following steps must be performed. This methods is similar to the personalization process above described, and 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 **GDOS** operation, or in embedded systems.

a) The **NSB8** program must be created and completely tested in all its parts ensuring that any syntax and logical errors are eliminated. In other words you must obtain the final version of your program.

b) Inside **NSB8** environment, with the tested program already loaded, type the command:

PSIZE<CR>

that shows the program length in blocks of 256 bytes. Call SIZE_APPL the shown value and use it in the following steps.

c) Execute the following commands by directly using the BASIC on line interpreter:

FILL 3599,00<CR>
FILL 256,195<CR>
FILL 257,00<CR>
FILL 258,14<CR>

This commands set the autostart flag (address 3599 = E0FH) and put a jump to **NSB8** default entry point (address E00H) at the beginning of **GDOS 80** command file.

d) Type BYE in order to return to the **GDOS 80**, indicated by the proper prompt **ABACO(r)>** representation.

e) Type the **GDOS** direct comand:

SAVE nn <drive:filename.G80><CR>

where **nn** correspond to the result of the sum **67+SIZE_APPL** in **hexadecimal**.
At this point, when the operating system prompt is redisplayed, the command file has been created and it is the file **filename.G80** created on the specified **drive**. This file can be directly tested thanks to operating system capability to load and execute a command file:

<drive:filename><CR>

For example to create a **GDOS 80** autorun program on target card RAM disk, with a BASIC program length SIZE_APPL=3, the command:

SAVE 46 M:AUTORUN.G80<CR>

must be typed. At the following reset or power on the card will automatically execute the program.
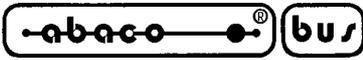
SEE ALSO:
Command PSIZE
GDOS 80 User manual

APPENDIX B: QUICK REFERENCE

FLAGS AND PARAMETER

ADDRESS	NAME	FUNCTION	DEFAULT
3584	ENTRY1	First entry point (erase program and data)	-
3588	ENTRY2	Second entry point (erase only data)	-
3590,3591	ENDBAS	End of BASIC	17280
3593,3594	HIGHMEM	End of available memory	58367
3598	LINECT	Current max line length	80
3599	AUTOST	Autostart flag	1
3600	BOOTPR	Bootsrap PROM high address	232
3601,3602	LINETB	Print head (cursor) table address	-
3603	PAGES	Number of line per screen page	24
3604	ENTRY3	Third entry point (no erase)	-
3607	DELECHO	Delete character	8
3608	PANICOK	<control-C> panic button flag	0
3993	OPENFILE	Number of open file (max 5)	0

MEMORY SIZE SETTING

MEMSET VALUE	HEX HIGHMEM ADDRESS	SIZE (KBytes)
MEMSET 24575	5FFF	24
MEMSET 28671	6FFF	28
MEMSET 32767	7FFF	32
MEMSET 36863	8FFF	36
MEMSET 40959	9FFF	40
MEMSET 45055	AFFF	44
MEMSET 49151	BFFF	48
MEMSET 53247	CFFF	52
MEMSET 57343	DFFF	56
MEMSET 58367	E3FF	57
MEMSET 59391	E7FF	58
MEMSET 61439	EFFF	60
MEMSET 63487	F7FF	62
MEMSET 65535	FFFF	64

INSTRUCTION CODES TABLE

The below table reports all the statement code (token) of **NSB8** instructions:

Most Significant Nibble

	8	9	A	B	C	D	E	F
0	LET	FN	CLS	STEP			(	<=
1	FOR	DEF		TO			^	<>
2	PRINT	!		THEN		ATN	*	
3	NEXT	ON		TAB		FILESIZE	+	
4	IF	OUT		ELSE	SORT	FILEPTR		<
5	READ	FILL		CHR$		ADDR	-	=
6	INPUT	EXIT		ASC	INT			>
7	DATA	OPEN	APPEND	VAL			/	NOT
8	GOTO	CLOSE		STR$		FREE		
9	GOSUB	WRITE		NONENDMARK		INP		
A	RETURN			INCHAR$	SGN	EXAM		
B	DIM	CHAIN		FILE	SIN	ABS		
C	STOP	LINE			LEN	COS	AND	
D	END				CALL	LOG	OR	
E	RESTORE				RND	EXP		
F	REM	ERRSET				TYP	>=	

(Left side label: Less Significant Nibble)

FIGURE B-1: INSTRUCTION CODES

SOURCE FILE SYNTAX

The tokenized **NSB8** sorce files must have the following syntax:

<file name>.B

Where <file name> is a 6 character sequence valid for the used operating system.

DATA FILE SYNTAX

The data files managed by **NSB8** must have the following syntax:

<file name>.<ext>

Where <file name> and <ext>, are respective 6 and 3 character sequences valid for the used operating system.

ARITHMETIC OPERATORS

OPERATORS	OPERATION	EXAMPLE
^	Exponentation	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

RELATIONALS OPERATORS

OPERATORS	RELATION	EXAMPLES
>	Greather than	(6>1)=1 (true)
		(2>3)=0 (false)
<	Less than	(0<0)=0 (true)
		(1<3)=1 (false)
<=	Less than or Equal to	(5<=5)=1
		(3<=5)=1
		(3<=5)=1
>=	Greather 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

AND	Logic AND (= boolean sum)
OR	Logic OR (= boolean multiplication)
NOT	Logic negation

OPERATORS ORDER OF EVALUATION

NOT, **-**	negates a number,unary minus
^	exponentiation
* , **/**	multiplication and division
+ , **-**	addition and subtraction
=,<,>,<>,<=,>=	relationals
AND	boolean sum
OR	boolean multiplication

FORMATTED PRINTING

Syntax rules:

   PRINT %<format specifier string><format character>,<variables, expressions, ...>

format specifier string: **nFm** Floating format
   nI   Integer format
   nEm Scientific format

n = number of integer digits
m = number of decimal digits

format character:
	A	Counting
	C	Comma for each 3 digits group
	Z	Suppress trailing zeroes
	+	Sign value
	$	Dollar sign
	#	Default format

LINE EDITOR

<control-G>	Copy to end of line
<control-N>	Delete for a new insertion
<control-A>	Copy a character
<control-Q>	Go one character back
<control-Z>	Delete one character
<control-D> X	Copy till X character
<control-Y>	Enable and disable the insert mode

INTERNAL FUNCTIONS

MATHEMATIC FUNCTIONS

ABS	ABSolute value (<expression>)
ATN	ArcTaNgent (<numeric expression>)
COS	COSine (<numeric expression>)
EXP	EXPonential value (<numeric expression>)
INT	INTeger value (<numeric expression>)
LOG	LOGarithmic value (<numeric expression>)
SIGN	SIGN of a number (<numeric expression>)
SIN	SINe (<numeric expression>)
SQRT	SQuare RooT (<numeric expression>)

STRING FUNCTIONS

ASC	ASCii value (<string expression>)
CHR$	CHRaracter$ (<numeric expression>)
LEN	LENgth of a string (<string name>)
STR$	STRing$ (<numeric expression>)
VAL	VALue (<string expression>)

INPUT OUTPUT FUNCTIONS

INCHAR$	INput a CHARacter$ (<device#>)
INP	INPut a byte (<port address>)
<status>=CALL(17200,<acquire>)	Acquire console status
<not used>=CALL(17152,<I/O add.>)	Set 16 bits I/O address
<data in>=CALL(17168)	Perform input from a 16 bits port address
<not used>=CALL(17184,<data out>)	Perform output to a 16 bits port address

FILE FUNCTIONS

FILE	FILE type (<file name>)
FILEPTR	FILEPoinTeR position (<file#>)
FILESIZE	FILESIZE (<file#>)
TYP	TYPe of file pointer (<file#>)

MISCELLANEOUS FUNCTIONS

CALL	CALL machine language (<memory address> [,<argument>])
EXAM	EXAMine memory (<memory address>)
FREE	FREE memory (<argument>)
TAB	TABulate (<#expression>)
RND	RaNDom (<#expression>)

USER DEFINED FUNCTIONS

DEF <function name> (<parameter list>) [=<expression>]
RETURN <numeric or string variable>
FNEND

TERMS MEANING

LINE#	BASIC line number
DEVICE#	input, output device number
DRIVE#	disk drive number
FILENAME	name of the file
#EXPR	numeric expression
LOGEXPR	logic expression
TYPEXPR	a type of expression
FILE#	file identification number
[ ]	optional

DIRECT COMMANDS

ALOAD
ASCSAV
AUTO [<LINE>] [,<INCREMENTAL VALUE>]
BYE
CAT [#<DEVICE#>] [,<DRIVE#>]
CONT
DEL <LINE#>,<LINE#>
LIST [#<DEVICE#>] [,<LINE#>] [,<LINE#>]
LOAD <FILENAME>
MEMSET <MEMORY ADDRESS>
PSIZE
REM [<LINE#>] [,<INCREMENTAL VALUE>]
RUN [<LINE#>]
SAVE <FILENAME>
SCR

STATEMENTS

* IT CAN BE DIRECTLY USED AS A COMMAND

STATEMENTS FOR PROGRAM INTERNAL DATA MANAGEMENT

      DATA <CONSTANTS LIST>
      READ <VARIABLES LIST>
*   **RESTORE** [<LINE#>]

INPUT AND OUTPUT STATEMENTS

      INPUT [<DEVICE#>] [,<STRING PROMPT>,] <VARIABLE LIST>
      INPUT1 [<DEVICE#>] [,<STRING PROMPT>,] <VARIABLE LIST>
*   **OUT** <PORT ADDRESS>,<BYTE VALUE>
*   **PRINT** [@,(ROW,COLUMN)][#<DEVICE#>] [,<LIST OF EXPRESSIONS>]
*   **!** [@(ROW,COLUMN)] [#<DEVICE#>] [,<LIST OF EXPRESSION>]

CONTROL STATEMENTS
      FOR <VARIABLE>=<INITIAL> **TO** <LIMIT> [**STEP** <VALUE>]
      GOSUB <LINE#>
      GOTO <LINE#>
*   **IF** < LOGEXPR> **THEN** <STATEMENT> [**ELSE** <STATEMENT>]
      ON <#EXPR> **GOSUB** <LIST OF LINE NUMBERS>
      ON <#EXPR> **GOTO** <LIST OF LINE NUMBERS>
      RETURN

FILE STATEMENTS

*   **APPEND** [LINE#>,]<FILENAME>
*   **CHAIN** <FILENAME>
*   **CLOSE** #<FILE#>
*   **OPEN** #<FILE#>[%<TYPEXPR>],<FILENAME>[,<SIZEVAR>]
*   **READ#** #<FILE#>[%<RANDOM ADDRESS>],<LIST OF VARIABLES>
*   **WRITE#** #<FILE#>[%<RANDOM ADDRESS>],<LIST OF EXPRESSIONS>

GENERAL STATEMENTS

*   **CLS** [#<DEVICE#>]
*   **DIM** <VARIABLE NAME (<ARRAY OR STRING DIMENSION>)
      END
      ERRSET [<LINEA#>,<ERROR LINE NUMBER>,<ERROR NUMBER>]
*   **FILL** <MEMORY ADDRESS>,<BYTE VALUE>
*   [**LET**] <NUMERIC VARIABLE>=<numeric expression>
*   [**LET**] <STRING VARIABLE>=<string expression>
*   **LINE** [#<DEVICE#>,]<#expression>[,<#expression>]
      REM <EXPLANATION COMMENT>
      STOP

TRAPPABLE ERRORS

ARGument	Error 1
DIMENSION	Error 2
OUT OF BOUNDS	Error 3
TYPE	Error 4
FORMAT	Error 5
LINE NUMBER	Error 6
FILE	Error 7
HARD DISK	Error 8
DIVIDE by ZERO	Error 9
SYNTAX	Error 10
READ	Error 11
INPUT	Error 12
ARGument MISMATCH	Error 13
NUMERIC OVerflow	Error 14
STOP/control C	Error 15
LENGTH	Error 16

APPENDIX C: ALPHABETICAL INDEX

G

H

I

L

M

N

O

P