NCR

Personal Computer

GW™-BASIC

Second Edition, March 1985

CUSTOMER PROGRAM LICENSE AGREEMENT

YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS BEFORE OPENING THIS PACKAGE. OPENING THIS PACKAGE INDICATES YOUR ACCEPTANCE OF THESE TERMS AND CONDITIONS. IF YOU DO NOT AGREE WITH THEM, YOU SHOULD PROMPTLY RETURN THE PACKAGE UNOPENED AND YOUR MONEY WILL BE REFUNDED.

This License Agreement applies to the Program contained in the accompanying package. Unauthorized copying is prohibited by the Copyright Law. YOU MAY NOT USE, COPY, MODIFY OR TRANSFER THE PROGRAM, IN WHOLE OR IN PART, EXCEPT AS EXPRESSLY PROVIDED FOR IN THIS LICENSE AGREEMENT. The Program is or may be considered by the copyright owner or licensor of NCR as confidential, proprietary and a trade secret and should be safeguarded by you as such. Title to the Program and copies of it remains with the copyright owner.

LICENSE

You may:

a. use the Program only on a single machine at a single location, unless the Program has been specifically designated by NCR, in writing or on this package, for your use on a group of machines.

b. copy the Program into any machine readable or printed form for backup of modification purposes only, to support your use of it on the single machine or designated group of machines. (Certain programs may include mechanisms to limit or inhibit copying. They may be designated by a "Copy Protected" notice);

c. modify the Program for your use on the single machine or designated group of machines. (Any portion of the Program merged into another program will be considered to be a modification and will be subject to the terms and conditions of this License Agreement); and

d. transfer the Program and license to another party only if the other party agrees to accept the terms and conditions of this License Agreement. You must advise NCR of the name and address of the other party and the other party must accept the terms of this License Agreement by signing a copy of it and providing NCR with the signed copy. If you transfer the Program, you must at the same time destroy all copies whether in printed or machine-readable form which you have not transferred to the other party and this includes all modifications of the Program (including portions of it contained or merged into other programs).

You must reproduce and include any copyright notice and serial number on any copy, modification or portion merged into another program.

IF YOU TRANSFER POSSESSION OF ANY COPY, MODIFICATION OR MERGED PROGRAM TO ANOTHER PARTY, YOUR LICENSE WILL BE AUTOMATICALLY TERMINATED.

TERM

Your license will be effective until terminated. You may terminate it at any time by destroying the Program, including all copies, modifications and merged portions in any form. It will also terminate upon conditions set forth elsewhere in this License Agreement or if you fail to comply with any term or condition in this License Agreement. You agree that upon any such termination you will destroy the Program, including all copies, modifications and merged portions in any form.

EXCLUSION OF WARRANTY

EXCEPT AT STATED IN THE "LIMITED WARRANTY" BELOW, THE PROGRAM IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU (AND NOT NCR OR ITS DEALER OR DISTRIBUTOR OR ANY LICENSOR OF NCR OR OWNER OF THE PROGRAM) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. There is no warranty by NCR or any other party or person that the functions contained in the Program will meet your requirements or that the operation of the Program will be uninterrupted or error free. You assume all responsibility for the selection of the Program to achieve your intended results, and for the installation, use and results obtained from it.

LIMITED WARRANTY

NCR warrants both the media on which the Program is reproduced and the reproduction of the Program on the media to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of delivery to you as evidenced by a copy of your receipt.

NCR's entire liability and your exclusive remedy shall be:

1. The repair or replacement of any media not meeting NCR's "Limited Warranty" and which is returned to NCR or an authorized NCR dealer or distributor within the 90-day period, with a copy of your receipt, or

2. If NCR or its authorized dealer or distributor is unable to deliver replacement media and repair is not practicable or cannot be timely made, you may terminate this License Agreement by returning the Program and your money will be refunded.

IN NO EVENT WILL NCR OR ANY OTHER PARTY OR PERSON BE LIABLE TO YOU OR ANYONE ELSE FOR ANY DAMAGES, INCLUDING LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM OR MEDIA EVEN IF NCR OR THE OTHER PARTY OR PERSON HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Some states do not allow limitations on how long an implied warranty lasts, so the above exclusion may not apply to you.

Some states do not allow limitations or exclusion of liability for incidental or consequential damages so the above limitation or exclusion may not apply to you.

This warranty gives you specific legal rights and you may also have other rights which vary from state to state.

MISCELLANEOUS

You may not sublicense, assign or transfer the license or the Program except as expressly provided in this License Agreement. Any attempt otherwise to sublicense, assign or transfer any of the rights, duties or obligations hereunder is prohibited, and will automatically terminate your license and right to use the Program.

This License Agreement will be governed by the laws of the State of Ohio where NCR Corporation has its principal office.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN US AND THAT IT SUPERSEDES ANY PROPOSAL OR PRIOR AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN US OR BETWEEN YOU AND ANY NCR DEALER OR DISTRIBUTOR RELATING TO THE SUBJECT MATTER OF THIS LICENSE AGREEMENT.

Should you have questions concerning this License Agreement or should you desire to obtain information on warranty performance, you may contact NCR by writing to:

NCR CORPORATION
1150 Anderson Drive
Liberty, SC 29657
USA

X-9695-1
092584

HOW TO USE THIS MANUAL

GW-BASIC is a widely used programming language that gives you complete and comfortable access to the features of your NCR PERSONAL COMPUTER. These features include disk access, printing, communications, high-resolution graphics, and even music.

With the help of the GW-BASIC full screen editor and the extensive GW-BASIC instruction set, you can create programs for a wide variety of applications. The versatile printing and drawing instructions enable your program to produce, store, and recall lists, letters, and business graphics as simple or as intricate as you require. For mathematicians there is a wealth of functions, or, if you are programming "just for fun", you might wish to add some music to your programs. GW-BASIC also provides programming facilities that enable you to make full use of a color display, a light pen, and a joystick.

Chapter 1 gives instructions for both the beginner and experienced programmers on how to get GW-BASIC started, and how to leave GW-BASIC when you have finished creating or running a program. The Chapter continues with a description of the way in which GW-BASIC communicates with you and how it stores the information you give it. The final sections of Chapter 1 show the kinds of decisions a GW-BASIC program can make, and also the help it requires from you in order to make these decisions. Each aspect covered in this introduction is accompanied by examples and exercises to assist the beginner in what might well be a first encounter with computing.

Chapter 2 introduces the GW-BASIC program editor. This is a description of the facilities provided by GW-BASIC to enable you to write new programs and change existing programs. If you are already acquainted with the line editors offered by some BASIC versions, you will especially appreciate the true full screen editing capability of GW-BASIC.

Chapter 3 presents the distinguishing feature of GW-BASIC, namely the variety of screen display possibilities. In addition to the extended character set you can use on a monochrome display, a color display allows you to use the graphic capability of GW-BASIC to its fullest. As in Chapter 1, the description is accompanied by examples and exercises.

The introductory pages to Chapter 4 give you a list of the complete GW-BASIC instruction set. Most of the instructions "speak" for themselves: BEEP obviously has something to do with making a sound; CIRCLE is concerned with drawing precisely that geometrical figure. The list is divided into sections, each dealing with a particular aspect of programming, for example, "Loading and storing programs", and "The loudspeaker." Then follows, in alphabetical sequence, a full description of each statement, command and function in the GW-BASIC instruction set. A large number of programming examples are included. Therefore, this Chapter serves both as a reference document for experienced programmers and as a practical guide for newcomers to programming. Even if an example does not fulfill your particular program requirement, you will find the information that enables you to adjust that example to create the effect you wish to see or hear.

Chapter 5 describes the different ways of storing information on disk files and how GW-BASIC communicates with external devices such as a printer. You may already know the meaning of terms such as "random access" and "sequential access" related to files, but just in case these terms are new to you, explanations and examples are included.

Chapters 6 and 7 are of interest mainly to assembler programmers who wish to incorporate machine language routines into a GW-BASIC program, or who are interested in how GW-BASIC makes use of the memory of the NCR PERSONAL COMPUTER.

The manual ends with several appendicies that contain other useful information: a list of reserved words and the character set used by GW-BASIC (A and B); error messages displayed by GW-BASIC (C); additional mathematical functions (D); a program for converting decimal values to hexadecimal values (E); and, a keyboard layout with internal key positions indicated (F).

GW-BASIC provides an environment conducive to testing programs. With RUN and GOTO you can enter a program at any point you wish, the STOP instruction does precisely what its name suggests, and a special tracing facility (TRON/TROFF) enables you to check the path your program is taking. Even if your program appears not to be working, GW-BASIC cannot damage your computer. Furthermore, GW-BASIC can give you valuable information as to what might not be working properly in a program. For example, if you tell GW-BASIC to play the note H, it will not only point out that no such

note exists, but also tell you where in the program the erroneous instruction occurred.

In summary, regardless of your knowledge base, learning GW-BASIC with this manual will enhance your programming proficiency. GW-BASIC is both simple and intrinsically helpful, as is this manual with its numerous examples and exercises.

GW-BASIC

Contents

Chapter 5 Files and Devices

Introduction

The GW-BASIC flexible disk is for use with the NCR-DOS operating system. Therefore, before you start, you should already know about some of the fundamental operations within the operating system, such as how to copy disks and how to issue commands to the operating system. It also helps if you are already familiar with the conventions of giving names to disk files. It is certainly worthwhile looking at these aspects of your operating system, before you start to write GW-BASIC programs.

NOTE: The GW-BASIC program file resides on the master NCR-DOS/GW-BASIC diskette in the NCR-DOS manual. One file with four separate names (GW-BASIC.EXE, GWBASIC.EXE, BASICA.EXE, and BASIC.EXE) exists, and you can choose to use any one of the names as desired. To copy the GW-BASIC program to another diskette, use the NCR-DOS command COPY.

HOW TO START-UP GW-BASIC

The first step is to load the NCR-DOS operating system into the memory of your computer. (If you are not yet acquainted with this procedure, consult your *NCR-DOS* manual.) Once the system prompt is displayed on the screen, you can load GW-BASIC into memory. The name of the file to be loaded is GWBASIC, so enter:

```
GWBASIC
```

(Remember, a command is complete only when you subsequently press the(↵) key. This key is referred to as the <ENTER> key in the remainder of this manual.) When GW-BASIC has been successfully loaded, it will announce its presence with a sign-on message followed by:

```
Ok
```

"OK" means that GW-BASIC is ready to accept your commands. The information appearing at the bottom of the screen relates to the Function Keys of the keyboard and need not concern us for the moment.

The following alternative methods of loading GW-BASIC are of interest to experienced programmers, so if you are new to GW-BASIC, you may wish to skip the rest of this section. Proceed then to "How to Exit GW-BASIC."

The alternative methods of loading GW-BASIC involves including one or more of the following options in the loading command:

GWBASIC filespec

where filespec represents the file specification of a BASIC program (not enclosed in quotation marks). This command tells GW-BASIC to find that file, load it, and RUN it without displaying the GW-BASIC sign-on message. The filespec must be in agreement with NCR-DOS file specification conventions. It may include a path. If you do not specify the file extension, GW-BASIC assumes that the file extension is .BAS.
Examples:

GWBASIC OLDPROG.ABC

loads GW-BASIC into memory and starts to RUN the program contained in the file OLDPROG.ABC, whereas

GWBASIC OLDPROG

loads GW-BASIC into memory and starts to run the program contained in the file OLDPROG.BAS. If you are using the AUTOEXEC.BAT file for the automatic execution of a sequence of GW-BASIC programs when loading NCR-DOS, each program must end with the GW-BASIC SYSTEM command to ensure the execution of the next command in the AUTOEXEC.BAT file.

< stdin

This option refers to the standard input device, which is normally the keyboard. 'stdin' represents the file from which GW-BASIC should accept input, instead of the keyboard. If you are using this option, it must precede any options that start with at slash (/).

> stdout

refers to the standard output device, the screen. 'stdout' represents the file to which GW-BASIC should direct output. If you are using this option, it must precede any options that start with a slash (/). If '>>' is used instead of '>', the output is appended to the file; otherwise, the file is overwritten.

/F:number

where number represents the number of disk files that may be open (maximum 15) at any one time during the execution of a BASIC program. This option is valid only with /I option (see /I option). If this option is omitted, the number of files defaults to 3. Each file requires 62 bytes of computer memory for its File Control Block, plus the buffer size (see /S option). The number of files that may be open at one time depends on the value assigned to the FILES parameter in the CONFIG.SYS file when NCR-DOS is loaded. When this parameter is not used or CONFIG.SYS file is not used, the default number for open files is 8. GW-BASIC itself requires three for stdin, stdout, stderr, stdaux, and stdprn and one for LOAD, SAVE, CHAIN, NAME, and MERGE files. So, if CONFIG.SYS has FILES = 8, the maximum value for the number of files in the /F option is 4. After all of the files have been opened, attempts to open another will result in the "Too many files" error.

/M:address,blocksize

where address represents the highest memory location that can be used by GW-BASIC. This option is useful for reserving an area in upper memory for use by your program. Obviously, this address must be a realistic one; that is, it must leave memory space at least for your program. The maximum amount of memory which can be reserved is 64KB. When omitted or 0, GWBASIC attempts to allocate all it can up to a maximum of 65536 bytes. Example:

GWBASIC/M:32768

allows GW-BASIC to use the first 32 KB of the data and stack segments setup by the operating system. You must use the second part of this option to set a maximum blocksize, when you intend to load programs above the address defined in the first part of the option. Use of 'SHELL' command requires the block size to be specified. Otherwise, 'COMMAND' will be loaded on top of your

routines when a 'SHELL' command is executed. Blocksize is the number of memory paragraphs (each of 16 bytes) required as workspace for GW-BASIC, plus the extra space you require outside the GW-BASIC program area. Example:

```
GWBASIC /M:32000,2048
```

This results in a total of 32768 (2048 x 16) bytes being reserved, of which 32000 are for GW-BASIC, and 768 for use outside GW-BASIC.

/S:size

where size is the buffer size for use with random access files. The maximum allowable size is 32767 bytes. This option determines the maximum record length which an OPEN command may set. This option is valid only with /I option. If this value is omitted the default record length will be 128 bytes.

/C:comsize

sets the size of the buffer used for receiving data under RS232C asynchronous communications. If you are not using a serial printer, this option has no effect. The maximum allowable value for this option is 32767. If you set the value 0 for this option, the RS232C support is disabled; no buffer space is allocated, therefore the portions of GW-BASIC relating to the communications facility are not loaded from disk. Any subsequent I/O attempts will result in a "Device Unavailable" error. If you are using two asynchronous communications facilities, comsize applies to both of them. The value recommended for a high-speed communications line is 1024. The buffer for transmitting data is always allocated in multiples of 128 bytes.

/D

This option tells GW-BASIC that you wish to use double precision for the following mathematical functions: ATN, COS, EXP, LOG, SIN, SQR, and TAN. The inclusion of this option requires approxmately 3000 bytes more memory than would otherwise be occupied by GW-BASIC.

/I

This option specifies that space for file operations is to be allocated statically rather than dynamically. Thus, the /S and /F options are not generally needed. However, if programs were written such that certain BASIC internal structures were made static, these programs could be run under GW-BASIC by specifying the /I option together with the /F and /S options to statically allocate space for file operations. An example would be to keep the field definition static (See FIELD Command) so that random files may be opened, closed and reopened without redefining FIELD everytime the file is reopened.

All numbers in the above options can be in decimal, octal (prefixed with &O) or hexadecimal (prefixed with &H).

The default values for all the above options when not specified can be summarized as follows: the maximum number of files (/F) which can be opened at one time is 3, there is no constraint on memory use (/M), the random file record size is 128 bytes (/S), the buffer size for asynchronous communications is 256 bytes (/C), the /D option is not in force, and space for file operations is dynamically allocated (/I).

You can combine these options in a single command when loading GW-BASIC. Here are some examples:

GWBASIC PAYROLL.BAS /F:6

Use up to 64 KB memory, load GW-BASIC, and execute the program file PAYROLL.BAS with up to 6 open files.

GWBASIC /M:32768

Load GW-BASIC and wait for further instructions. GW-BASIC itself may not use memory above location 32768.

GWBASIC DATACK /I/F:2/S:256/M:32768

Load GW-BASIC and execute the program file DATACK.BAS. No more than 2 files may be open at any one time with maximum record length of 256 bytes for random file; memory above location 32768 is out of bounds to GW-BASIC.

HOW TO EXIT GW-BASIC

To exit GW-BASIC and return to the NCR-DOS operating system, enter the command

SYSTEM

This command returns control to NCR-DOS. Note that use of the <Ctrl-Break> key combination does not have this effect. Instead, it is used for breaking out of a GW-BASIC program, and returning to the GW-BASIC "Ok" level.

SAVING AND RETRIEVING A PROGRAM

If you are a newcomer to GW-BASIC, you should read this section before going on to do the subsequent exercises.

As soon as you have loaded GW-BASIC, the "Ok" prompt appears. This is how GW-BASIC tells you that it is waiting for your instructions. At this point you can start writing a program. Before testing your program, it is usually a good idea to save it on disk. For this you need the SAVE command. At this point you must decide what you are going to call your program. The following example saves a program under the name NEWPROG.BAS on the currently active disk:

SAVE "NEWPROG"

This command can be entered at any time when "Ok" is the last message on the screen, or when the blinking cursor indicates that GW-BASIC is waiting for you to enter another program line. When the "Ok" message reappears, you know that your program is now on disk.

GW-BASIC assumes that what you are saving on disk is a GW-BASIC program and gives the program filename the extension .BAS. You could even specify the .BAS extension yourself (SAVE "NEWPROG.BAS"), but the result would be no different. You can, if you wish, specify a different extension to the filename. GW-BASIC would respect your choice, but every time you wished to do some work

on that program, you would have to enter the extension as well as the filename. Therefore, it seems only reasonable to let GW-BASIC use the .BAS extension to the filename.

When you wish to resume work on your program, first load GW-BASIC. When "Ok" appears, enter

```
LOAD "NEWPROG"
```

and your program is back again in the computer's memory. You can continue writing (editing) your program, or you can execute it using the RUN command. To do the latter simply type

```
RUN
```

If you know from the outset that you are going to run an already existing program without any further editing, you can choose between two other methods. When your NCR PC displays the NCR-DOS prompt, you can include the name of the program (without quote marks) when you load GW-BASIC:

```
GWBASIC NEWPROG
```

Alternatively, having loaded GW-BASIC, and once the sign-on message and "Ok" have appeared, you can type

```
RUN "NEWPROG"
```

In either case, the program is executed without further action on your part.

There is little point in entering these commands at the moment, as you have not yet written a program for GW-BASIC to save on disk. If you have just tried loading or running a non-existent program, you may have noticed that GW-BASIC "complained" that the program file could not be found. Such error messages are nothing to be too concerned about. They are GW-BASIC's way of telling you what information is missing or indicating where something may have gone wrong.

EXERCISES

Reset the system by means of the <Ctrl-Alt-Del> and load NCR-DOS. When the system prompt appears, start GW-BASIC by entering

GWBASIC

Remember that "enter" means to type in the words and then *press the <ENTER> key.)* GW-BASIC then announces its presence by means of the "Ok" message. Now type the following line:

20 PRINT "This is how short a program can be"

This time GW-BASIC does not return "Ok" but expects you to enter further program lines. This may be a very short program, but you can still save it on disk. Decide on a name, say, MINIPROG, and type the following GW-BASIC command (if you are not sure of NCR-DOS file naming conventions, use a name consisting of up to 8 letters):

SAVE "MINIPROG"

When "Ok" is displayed again, leave GW-BASIC and return to the operating system level by entering

SYSTEM

You can now check that MINIPROG.BAS is really in the directory by using the NCR-DOS DIR command.

Now resume working on your program by first loading GW-BASIC and then entering

LOAD "MINIPROG"

"Ok" tells you that your program has been found and loaded. You can view the contents of the program by typing

LIST

which, in this case, results in the display of the single program line followed by "Ok". Add the following entry to the program at the place where the cursor is blinking:

10 CLS

Then enter LIST once again. You have probably noticed that GW-BASIC has put the two program instructions "in the right order", that is, with the lower of the two numbers first. This is the order in which they will be executed when RUN is entered. But first, save this updated version of the program in the same way as before:

SAVE "MINIPROG"

Now for a trial run. Simply enter

```
RUN
```

The first of the two commands (the one prefixed with the number 10) Clears the Screen; the second PRINTs the text contained in the quotation marks on the screen. This short program is executed very quickly, and then the "Ok" message is displayed. Now return to the operating system level (SYSTEM), so that you can try another method of running a program. When the NCR-DOS prompt appears, enter

```
GWBASIC MINIPROG
```

First, GW-BASIC is loaded, and immediately following this, MINIPROG is loaded and executed without waiting for any further instructions on your part.

If you make a mistake when entering a program line (that is, a command prefixed by a number), simply enter the line in its incomplete or incorrect form and write the line in its correct form using the same line number. The old, incorrect version is replaced by the new, correct version, as soon as the latter is entered. If you overlook a mistake that contradicts the language rules of GW-BASIC, it will not be noticed by GW-BASIC until you execute the program. Then GW-BASIC will stop the program and indicate to you the line number in which the mistake is present. To delete a program line without replacement, simply enter DELETE with the number of the line concerned. The complete facilities of the editor are described in the *Full Screen Editor* chapter.

Line numbers tell GW-BASIC the sequence in which you wish commands to be carried out. A line number must be in the range 0 to 65529, but there is no reason why a program 5 lines long must use the lines 0, 1, 2, 3, and 4. Indeed, this would be undesirable as it would prevent subsequent insertion of additional program lines. For this reason, it is a good idea to write your programs with a line increment, perhaps of 10.

The RENUM command offers a facility for creating more "space" between program lines, should this become necessary. AUTO is a command which automatically offers you line numbers with an increment determined by you, thus saving you the trouble of typing the line number for each line of the program.

GW-BASIC uses a shorthand form for the EDIT, LIST, AUTO, and DELETE commands to refer to the current line, namely the period (.). Therefore, if you enter

```
LIST.
```

the program line you are currently working on is displayed on the screen.

The two GW-BASIC lines which comprise the program MINIPROG.BAS each consist of one command only. GW-BASIC permits lines containing more than one command; in this case, the commands must be separated by colons. For example, the following line would have the same effect as the two lines of MINIPROG.BAS together:

```
10 CLS:PRINT "This is how short a program can be"
```

When typing a program line, you may exceed the length of a display line on the screen, provided the program line is not longer than 254 characters plus the <ENTER> key.

MODES OF OPERATION

When GW-BASIC is loaded, it displays the "Ok" message to indicate that it is ready to accept your commands. At this point, you have the choice of two modes: the indirect mode or the direct mode.

The indirect mode is the one you use for entering program lines as you have done in the exercises. It is called indirect because the commands are not executed immediately, but only when you issue the command to run the program. You do not have to tell GW-BASIC that you require this mode: as soon as it sees that the command is prefixed by a line number, GW-BASIC knows that the command is not intended for immediate execution, but to be part of a program which can be run, saved, and retrieved at a later time.

The commands LOAD, SAVE, RUN, and SYSTEM, as used in the earlier exercises, were direct commands; that is, their effect came about immediately. Accordingly, they did not have program line numbers. Using GW-BASIC commands as direct commands is often a convenient way of checking what has happened in a program. This facility also enables you to use GW-BASIC and your computer as a calculator for quick computations. Try entering the following simple calculation as a direct command:

```
PRINT 128+64-5
```

The result is displayed on the screen immediately. Omitting line numbers to achieve immediate results means that the command is not retained beyond execution (even though the result is the same as if it had been executed as part of a program). Therefore, when a command or sequence of commands is required for repeated use, it makes most sense to prefix them with line numbers and store them on disk. This is why we create computer programs.

THE CHARACTER SET

The GW-BASIC character set comprises all the letters of the alphabet as well as numeric characters (the digits 0 through 9). In addition, a number of special characters belong to the GW-BASIC character set. You may recognize some of these as denoting arithmetic functions. Others are of special significance in GW-BASIC programming. They are explained in the appropriate sections of this manual. Here is a list of the special characters:

Description	Symbol	Significance in GW-BASIC
Blank		Separates syntax elements in program line
Equal sign	=	Usual function in arithmetic comparisons. Assigns contents to program variable
Plus sign	+	Usual arithmetic significance Concatenation of texts
Minus sign	-	Usual arithmetic significance
Asterisk	*	Multiplication symbol
Slash	/	Division symbol
Caret	^	Exponentiation symbol
Left and right parentheses	( )	Usual algebraic significance
Percent sign	%	Denotes an integer number

Number (or pound) sign	#	Denotes a double precision number
Exclamation mark	!	Denotes a single precision number
Dollar sign	$	Denotes a text
Comma	,	Used in screen or printer formatting
Semicolon	;	Used as a delimited between string variables (same as comma)
Period	.	Decimal point
Single quotation mark	'	Delimits a programmer's remark
Double quotation mark	"	Delimits a text
Colon	:	Separates commands within a program line
Ampersand	&	Used in declaring number bases
Question mark	?	Editor abbreviation for PRINT command
Less than Greater than	< >	Usual algebraic significance
Backslash	\	Symbol for integer division

GW-BASIC also recognizes a number of Ctrl key combinations for program editing purposes. These are described in the section which deals with GW-BASIC's Full Screen Editor.

The GW-BASIC character set is an extension of the widely known and used ASCII code. The ASCII code attributes a value to 128 individual characters. For example, the ASCII code for the uppercase letter A is 65; for the digit 3, it is 51. The so-called control characters, that is, those codes that do not directly produce a screen image of their own

(for example, the codes determining cursor movement on the screen) are also represented in ASCII.

Appendix B of this handbook lists the complete GW-BASIC character set. The characters occupying the values 32 up to 126 will be already familiar to programmers. A few little-used ASCII control characters in the range 0 to 31 are used by GW-BASIC for graphic symbols. Although the values 128 to 255 are not represented in the ASCII code, GW-BASIC uses these values to provide you with a wealth of extra letters and other symbols.

Refer to *Appendix B* and enter commands like the following:

```
PRINT CHR$(65)
```

tells GW-BASIC to display on the screen the character that corresponds to the code value 65. As a result, the uppercase letter A is displayed (as if you had issued the command PRINT "A"). Now try a character that does not appear on the keyboard, for example

```
PRINT CHR$(227)
```

results in the Greek letter pi being displayed. Finally, try one of the control (that is, the non-displayable) characters:

```
PRINT CHR$(7)
```

causes the loudspeaker to beep.

You can enter the names of GW-BASIC commands in upper or lowercase. The next time you LIST your program on the screen, you see that GW-BASIC has converted into uppercase any lowercase letters you might have used in commands. This does not apply to letters you enter within quotation marks: GW-BASIC realizes that you intend these lowercase letters for later printing on the screen or a printer. If, for example, you enter line 10 of your program as

```
10 print "UPPER lower"
```

the next time you LIST this line on the screen it will appear as

```
10 PRINT "UPPER lower"
```

CONSTANTS

Constants are actual data that you supply to GW-BASIC. This data can take the form of a string or numeric constant. A string constant is one that is enclosed in (double) quotation marks, for example

"WELCOME"
"Enter any number"

Accordingly, line 20 of your short program in the last Exercises (20 PRINT "This is how short a program can be") contained a string constant. You can even place numbers inside quotation marks, for example

"$25,000.00"

(but you cannot do any arithmetics with those numbers as they currently stand).

When you want to do calculations with numbers, you can use numeric constants for both positive and negative numbers. The simple calculation in the direct mode of GW-BASIC in the last Exercises used three numeric constants (128, 64, and 5). There are five types of numeric constants:

Integer constants

Whole numbers between -32768 and +32767 (the plus sign is optional on a positive number).

Fixed Point Constants

Positive or negative real numbers; i.e., numbers that contain decimal points.

Floating Point Constants

Positive or negative numbers represented in exponential form (similar to scientific notation). A floating point constant consists of an optionally signed (+ or -) integer or fixed point number (mantissa) followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating point constants is 2.9E-39 to 1.7E+38.

Examples:

35E-2 ("thirty-five" times "ten to the power of minus 2") = .35
235.988E-7 = .0000235988
2359E6 = 2359000000

(Double precision floating point constants use the letter D instead of E. The difference between the two is described in this section.)

Hexadecimal Constants

Numbers with the prefix &H. (Assembler programmers know all about these.)

Examples:

&H76
&H32F

Octal Constants

Numbers with the prefix &O or &.

Examples:

&O347
&1234

Non-integer numeric constants may be either single precision or double precision numbers. Single precision numeric constants are stored with 7 digits of precision and displayed with up to 7 digits (6 digit accuracy). With double precision, the numbers are stored with 17 digits of precision and displayed with up to 16 digits.

A single precision constant is any non-integer numeric constant that has

- seven or fewer digits, or
- exponential form using E, or
- a trailing exclamation mark (!)

A double precision constant is any numeric constant that has

- eight or more digits, or
- exponential form using D, or
- a trailing number sign (#)

Examples of constants

Single Precision	Double Precision
46.8	345692811
-1.09E-06	-1.09432D-06
3489.0	3489.0#
22.5!	7654321.1234

VARIABLES

A variable is a kind of storage compartment in your program in which a value (string or numeric) is placed. Your program can assign a value

to a variable and manipulate its contents. You can even assign, check, and alter variables in the direct mode.

The name of a variable must start with a letter; the remainder of the name may consist of letters, digits, decimal points and a type declaration (see below). The name of a variable can be as long as you wish, but GW-BASIC recognizes only the first 40 characters. However, this hardly poses a limitation: it is good practice to keep variable names short, thus making the program easier to type into the computer.

A variable can store either a string or numeric value. The last character of a string variable must be $. This is the type declaration for a string variable. There are three types of numeric variables, each with a special character at the end of the name:

% Integer variable
! Single precision variable
Double precision variable

If you do not specify a type declaration, GW-BASIC assumes a numeric, single precision variable. (The difference between the different types of numeric values was discussed in the section "Constants".) Here are some examples of GW-BASIC variable names:

PI#	stores a double precision value
MINIMUM!	stores a single precision value
LIMIT%	stores an integer value
N$	stores a string value
ABC	stores a single precision value

When writing a program, you can choose the names of your variables. In addition to the formalities of name choice already mentioned, there is one further constraint: the name of a variable must not be identical with a name belonging to the GW-BASIC instruction set. These names are often termed "reserved words" (there is a list of them in *Appendix A).* A reserved word may, however, be embedded in a variable name. For example, GW-BASIC does not allow you to use the name PRINT$, but you may use the name PRINTER$. If the name of a variable begins with FN, GW-BASIC assumes that it is a call to a user defined function. This is described in Chapter 4, *Commands and Functions.*

It always makes sense to give a variable a name that somehow identifies the data it is holding. For example, if you want to calculate interest on a loan, you might store the current rate of interest in a

variable called INTEREST, and the name of the loaning or borrowing bank in BANK$. Any lowercase letters you enter in a variable name are converted by GW-BASIC to uppercase at the next LIST.

There is a second method of type declaration for variables, whereby any number of variables can be declared as integer, string, single or double precision in a single command (see DEFINT, DEFSTR, DEFSNG, and DEFDBL in the chapter *Commands and Functions).*

To store values in variables, GW-BASIC has the LET command. Examples:

```
LET TEMP=80
LET WEATHER$="sunny"
```

These commands say no more than that the numeric variable TEMP should now assume the value 80 and the string variable WEATHER$ should hold the letters "sunny". In fact, the word LET is never really required, so the following would do just as well:

```
TEMP=80
WEATHER$="sunny"
```

To display the current contents of a variable on the screen, simply use the PRINT command, for example

```
PRINT TEMP:PRINT WEATHER$
```

or, perhaps a little more meaningful

```
PRINT "It is a ";WEATHER$;" day, the temperature is
";TEMP;" degrees."
```

NOTE: If you do not explicitly set a variable to a value, either in direct mode or in a program, it will give you the answer 0 (numeric variable) or nothing at all (string variable) when you display its contents. The length of a string variable is the number of characters it contains, up to a maximum of 255 characters.

If you attempt to assign a string value to a numeric variable, or vice versa, GW-BASIC points out the error ("Type mismatch"). What GW-BASIC cannot do is to check the credibility of what is in a variable: if you assign "cold" to WEATHER$, the sentence displayed by the above PRINT command will seem somewhat contradictory, while as far as GW-BASIC can see, the command is syntactically acceptable.

It is possible not only to assign constants to a variable, but also to assign combinations of the same or other variables, and even perform arithmetic or string manipulation in the assign statement. Having set TEMP to 85 and WEATHER$ to "sunny", you could undertake the following changes:

```
TEMP=TEMP-20
WEATHER$ = "nice," + WEATHER$
```

The above PRINT command would then produce

```
It is a nice, sunny day, and the temperature is 65 degrees.
```

ARRAY VARIABLES

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by the name of the array and a subscript that tells GW-BASIC which element of the array you wish to access. The subscript is an integer or an expression which yields an integer. For example, PRINT NNAME$(4) tells GW-BASIC to display one entry of what is presumably a list of names.

An array may have more than one dimension. An example of a two-dimensional array is a mileage chart giving the distances between a number of towns. The command PRINT MILES(2,5) tells GW-BASIC to display the numeric value from the table entry indicated by reading element number 2 along the top of the table and element number 5 down the side.

The maximum number of dimensions for an array is 255; the maximum number of elements per dimension is 32767. The maximum amount of space in an array is the lesser of 65536 bytes or available memory.

The procedure for using array variables differs a little from that of using simple numeric variables. If the array variable is to contain more than 11 elements (subscripts 0 through 10) or more than one dimension, it must be declared accordingly (see DIM in the chapter, *Commands and Functions).* If you use an array before it is defined, it is assumed to be of single dimension with a maximum subscript of 10. We have already established that, when assigning a value to or reading the contents of an element, you must state in parentheses which element is intended, for example

```
V(2)=65
```

It is important to note that this assigns the value 65 to the third element of the array variable V (the first element is subscripted by 0, unless you change this to 1 using OPTION BASE).

SPACE REQUIREMENTS

The space requirements in the computer memory for the various types of numeric variables are as follows:

Integer, including hexadecimal and octal numbers — 2 bytes
Single precision — 4 bytes
Double precision — 8 bytes

In the case of an array, these figures are to be understood as per element. The space requirement of a string variable is the length of the string (number of characters), plus three bytes.

It is evident that numbers of higher precision require more memory space. They also require more time when being evaluated. Therefore, a program with integer variables runs faster, especially where repeated calculations are involved.

TYPE CONVERSION

It is sometimes necessary for GW-BASIC to convert a number from one degree of precision to another. When this is the case, the rules stated in this section apply. If you attempt to assign a string variable to a numeric variable, or vice versa, a "Type mismatch" error is displayed by GW-BASIC.

- If a numeric value of one precision is assigned to a numeric variable of a different precision, the number is stored with the precision declared for the target variable (the variable on the left of the equal sign). Example:

  ```
  10 A% = 23.42
  20 PRINT A%
  ```

  When you run this short program, GW-BASIC will display the number 23. If you assign the number 23.52 to A%, GW-BASIC displays the number 24. This is because GW-BASIC does not simply truncate, but rounds the number when assigning it to a variable of lower precision. Another example of this rounding takes place when assigning a double precision number to a single precision variable:

  ```
  10 C = 55.8834567#
  20 PRINT C
  ```

  The value displayed is 55.88346.

This rounding also occurs if you forget to use pure integers in commands or functions where integers are required. Example: given that the single precision variable SUBSCR contains the value 2.5, the command PRINT NNAME$(SUBSCR) will be interpreted by GW-BASIC at the time of execution as PRINT NNAME$(3).

- Assigning a lower precision number to a variable of higher precision cannot, of course, result in any greater accuracy. In fact, there is sometimes a very slight deviation from the original number, due to the way in which GW-BASIC stores numbers. Consider the following example:

```
10 A = 2.04
20 B# = A
30 PRINT A;B#
```

When you run this program, GW-BASIC will display the original and the new form of the number side by side as follows:

```
2.04 2.039999961853027
```

The following expression is of interest to mathematicians who require exact information about the degree of deviation:

```
ABS (B#-A) < 6.3E-8 * A
```

where B# and A represent the double and single precision variables, respectively.

- When an expression is being evaluated, all the operands are converted to the degree of precision of the most precise operand involved. Here are two examples:

```
10 D# = 6#/7
20 PRINT D#
```

All the arithmetic is performed with double precision, and the result (.857142571428571) is likewise returned with double precision.

```
10 D = 6#/7
20 PRINT D
```

Again, the arithmetic is done in double precision, but this time the result is assigned to a single precision variable. Therefore, the command PRINT D will yield a single precision result (.8571429).

- Logical operators (AND, OR etc, described later in this chapter) convert their operands to integers and yield an integer result. Operands must be in the range -32768 to 32767, otherwise an "Overflow" error occurs.

EXERCISES

The following example shows you how the use of string variables can save typing time when re-arranging elements of a text:

```
 10 A$ = "The quick brown":B$="jumps over the lazy"
 20 AN1$="fox":AN2$="dog":AN3$="bear"
 30 AN4$="kangaroo":AN5$="beaver":AN6$="camel"
 40 CLS
 50 PRINT A$;AN1$;B$;AN2$:PRINT
 60 PRINT A$;AN2$;B$;AN1$:PRINT
 70 PRINT PC;" % of ";NUMBER;" = ";RESULT
 80 PRINT
 90 GOTO 40
100 END
```

Every element of the text of the 6 lines thus displayed on the screen is held in a string variable. The PRINT items are separated here by semicolons, which tells GW-BASIC to start printing the next item immediately after the position where the last print item finished. The additional PRINT command at the end of each of the lines 50 to 100 creates a line of space between each line on the screen.

The next example shows a simple percentage calculation on a sequence of numbers you input to the computer. This program uses single precision arithmetic:

```
 10 PC=5
 20 PCX=PC/100
 30 CLS
 40 INPUT NUMBER
 50 IF NUMBER=0 THEN GOTO 100
 60 RESULT=NUMBER*PCX
 70 PRINT PC;"% or ";NUMBER;" = ";RESULT
 80 PRINT
 90 GOTO 40
100 END
```

Run this program. Line 10 assigns to PC the value 5. Line 20 divides this number by 100, thus creating a number which can be directly

multiplied to gain a percentage figure, and places the new number in the variable PCX. Line 30 clears the screen. Following this, GW-BASIC prompts you by means of a question mark to enter a numeric value (which may include a decimal point or be in exponential notation); the value you input is stored in NUMBER. Line 50 checks whether NUMBER is 0; if this is the case, GW-BASIC branches (GOTO) to line 100, which is the END of the program. If NUMBER is not 0, 5% of NUMBER is calculated (line 60), the result being stored in RESULT. The calculation with result is then displayed on the screen (line 70; line 80 prints a blank line). (If the result exceeds the number of digits which can be displayed as a single precision number, GW-BASIC automatically uses the exponential form.) GW-BASIC then branches back to line 40, with the effect that GW-BASIC once again waits for you to enter a new number at the keyboard. The loop of events between lines 40 and 90 is repeated until NUMBER contains 0. To change the percentage figure which is the basis of calculation, simply alter line 10 correspondingly. If you attempt an illegal input, for example, trying to include a letter in the number, GW-BASIC immediately asks you to redo that one input ("? Redo from start").

The final example in these exercises concerns the use of an array variable. It also shows how, with a minimum of program lines, a task can be repeated a number of times, each time in a slightly different form. The program defines an array variable consisting of 10 elements in a single dimension, the first element being subscripted by the number 1 (line 10). You are then asked to INPUT 10 names to fill this array (lines 20-50). Afterwards, the screen is cleared. You are then requested to enter numbers of your choice in the range 1 to 10. After each entry, the corresponding element of the array containing the names is displayed. If you enter 0, the program terminates.

```
 10 OPTION BASE 1:DIM NNAME$(10):CLS
 20 FOR LOOP% =1 TO 10
 30 PRINT "Enter somebody's name ";
 40 INPUT NNAME$(LOOP%)
 50 NEXT LOOP%
 60 CLS
 70 PRINT "Now recall those names":PRINT
 80 PRINT "Enter number 1 to 10":PRINT:PRINT
 90 INPUT "Number";N%
100 IF N% =0 THEN GOTO 130
110 PRINT "The name entered at place";N%;"is
";NNAME$(N%)
```

```
120 GOTO 90
130 END
```

This program uses a so-called FOR...NEXT loop from lines 20 to 50. Line 20 tells GW-BASIC how often the loop has to be passed through (from 1 to 10, i.e. 10 times). The number of passes already made through the loop is held in a variable, which is here called LOOP%, but which could be any numeric (preferably integer) variable. At the outset, this number is 1, which means that on the first pass, the array variable NNAME$ in line 40 is subscripted by 1. This in turn means that the first name you INPUT during the execution of the program is stored as the first element of the array variable. After this, GW-BASIC raises the value held in LOOP% and checks whether the limit specified in line 20 (namely 10) has been exceeded. As this is clearly not the case after the first pass, GW-BASIC returns to line 20 for the next pass: you are asked for the second time to enter a name; this is stored as the second element of the array variable, and so on. GW-BASIC goes on to line 60 only after the 10th pass through the loop.

Line 90 displays the request "Number" and waits for you to enter a number. This way of including the request in the INPUT command is a GW-BASIC facility which saves the use of an explicit PRINT command. Therefore, line 90 is equivalent to

```
90 PRINT "Number ";:INPUT N%
```

One thing this program does not prevent you from doing is entering an integer number greater than 10. As there is no element in the array with a subscript greater than 10, to do so would force GW-BASIC to terminate the program with the error message "Subscript out of range". The next set of exercises will show you how to program GW-BASIC to make logical decisions which detect such eventualities and thus prevent them from causing a premature termination of the program.

EXPRESSIONS AND OPERATORS

An expression may be simply a string or numeric constant, or a variable, or it may combine constants and variables by means of operators to produce a single value.

Operators perform mathematical or logical operations. These operations are carried out mainly on numeric values, but there is no reason why strings should not be added together or compared with one

another. The operators provided by GW-BASIC can be regarded as belonging to four different categories:

- Arithmetic
- Relational
- Logical
- Functional

ARITHMETIC OPERATORS

The arithmetic operators are:

Operator	Operation	Sample Expression
^	Exponentiation	X^Y
-	Negation	-X
*,/	Multiplication, Floating Point Division	X*Y X/Y
+,-	Addition, Subtraction	X+Y

These operators are given here in their order of precedence. If you have a mathematical background, you are probably aware of the significance of this order. It reflects the sequence in which subordinate expressions are evaluated within a more complex expression. Example:

```
2 + 6 * 5
```

There is obviously a difference in result between adding 2 to 6 and then multiplying this intermediate result by 5 (= 40), and multiplying 5 by 6 and then adding 2 (= 32). The generally accepted procedure of the two is the latter: multiplication is higher in the order of precedence than addition, which means that the multiplication part of the expression must be carried out first. This is often termed "algebraic logic".

You can override this order of precedence by use of parentheses. Accordingly, the expression

```
(2 + 6) * 5
```

yields the result 40.

In the section dealing with the GW-BASIC character set, there was already mention of special characters representing mathematical functions. One example of this is the asterisk, which in GW-BASIC

denotes multiplication. Here are some examples which show how GW-BASIC represents these mathematical functions:

Algebraic Expression	GW-BASIC Expression
$X+2Y$	X+Y*2
$X-\frac{Y}{Z}$	X-Y/Z
$\frac{XY}{Z}$	X*Y/Z
$\frac{X+Y}{Z}$	(X+Y)/Z
$(X^2)^Y$	(X↑2)↑Y
X^{Y^Z}	X↑(7↑Z)
X(-Y)	X*(-Y) Two consecutive operators must be separated by parentheses.

Integer Division and Modulus Arithmetic

Two additional operators are available in GW-BASIC integer division and modulus arithmetic.

Integer division is denoted by the backslash (\). The operands are rounded to integers (must be in the range -32768 to 32767) before the division is performed, and the quotient is truncated to an integer.

For example:

```
10\4        yields the value 2
25.68\6.99  yields the value 3
```

The precedence of integer division is just after multiplication and floating point division.

Modulus arithmetic is denoted by the operator MOD. It gives the integer value that is the remainder of an integer division. GW-BASIC creates integers, where necessary, by rounding (not truncating). Examples:

```
10.4 MOD 4 = 2 (10/4=2 with a remainder 2)
25.68 MOD 6.99 = 5 (26/7=3 with a remainder 5)
```

The precedence of modulus arithmetic is just after integer division.

Overflow And Division By Zero

If, during the evaluation of an expression, a division by zero is encountered, the "Division by zero" error message is displayed, machine infinity with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an expoentiation results in zero being raised to a negative power, the "Division by zero" error message is displayed, positive machine infinity is supplied as the result of the exponentiation, and execution continues.

If overflow occurs, the "Overflow" error message is displayed, machine infinity with the algebraically correct sign is supplied as the result. In the case of integer overflow, execution stops.

GW-BASIC does not check for "underflow" (the result of an operation is so small that GW-BASIC cannot distinguish it from zero).

RELATIONAL OPERATORS

Relational operators are used to compare two values. The result of the comparison is either "true" (-1) or "false" (0). This result may then be used to make decisions regarding program flow. (See IF, Chapter 4).

Operator	Relation Tested	Sample Expressions
=	Equality	X=Y
<> or ><	Inequality	X<>Y or X><Y
<	Less than	X<Y
>	Greater than	X>Y
<= or =<	Less than or equal to	X<=Y or X=<Y
>= or =>	Greater than or equal to	X>=Y or X=>Y
(The equal sign is also used to assign a value to a variable. See LET, Chapter 4).		

When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. For example, the expression

```
X+Y<(T-1)/Z
```

is true if the value of X plus Y is less than the value of T-1 divided by Z. More examples:

```
IF SIN(X)<.5 THEN GO TO 1000
```

```
IF I MOD J<>0 THEN K=K+1
```

In the first of these two examples the outcome of evaluating whether the sine of X is less than .5 determines whether program execution will branch to line 1000. In the second example, the value held in K is increased by 1, only if the remainder resulting from dividing the contents of the variable I by that of J is not zero.

```
PRINT SECONDS%>30
```

Causes GW-BASIC to display the number -1 if the integer variable contains a value greater than 30; otherwise, 0 is displayed.

LOGICAL OPERATORS

Logical operators perform logical, or "Boolean", operations. The operator denotes the kind of comparison that two values are subjected to. The GW-BASIC words for the various kinds of logical comparison are NOT, AND, OR, XOR, IMP, and EQV.

Your program could base a decision whether or not to display a recommendation to buy a particular commodity on the following consideration: "If the quality code is higher than 3, and the price less than 200, then buy!" A corresponding GW-BASIC command would look something like this:

```
IF QUALITY% > 3 AND PRICE < 200 THEN PRINT "Conditions are good for buying"
```

Both conditions must be fulfilled if the recommendation is to be displayed. Now consider the following example:

```
IF QUALITY% > 3 OR PRICE < 200 THEN PRINT "Conditions are acceptable for buying"
```

This means that at least one of the conditions must be fulfilled. It does not matter which one; furthermore, it would be acceptable if both were fulfilled.

Now let us consider the same situation from the seller's point of view. He or she might consider selling on the basis of the following consideration: "I am prepared to sell you the commodity of a quality higher than 3, but then the price cannot be less than 200. Alternatively, I will accept the lower price, but I cannot fulfill the quality requirement." GW-BASIC uses the XOR operator to express this sentiment of "either one or the other, but not both":

IF QUALITY% > 3 XOR PRICE < 200 THEN PRINT "Selling conditions are not ideal, but good enough to do business"

The logical operators NOT and EQV (equivalent to) have counterparts in the GW-BASIC character set:

IF NOT (TEMP = 100) THEN GOTO 1000 has the same effect as
IF TEMP <> 100 THEN GOTO 1000

IF ANSWER$ EQV "YES" THEN GOTO 1500 has the same effect as
IF ANSWER$ = "YES" THEN GOTO 1500

The following list gives the result of all the permutations of each of the six logical operators. 1 stands for "true"; 0 for "false". Taking the second permutation under OR as an example, the information can be read "If the first condition is fulfilled (1), but the second condition is not fulfilled (0), the conditions of the comparison as a whole are to be regarded as fulfilled (1)". The first permutation under XOR can read "If both the first and the second condition are fulfilled, the conditions of the comparison as a whole are to be regarded as not fulfilled".

X	NOT X
1	0
0	1

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

X	Y	X IMP Y (implies)
1	1	1
1	0	0
0	1	1
0	0	1

X	Y	X EQV Y
1	1	1
1	0	0
0	1	0
0	0	1

This list also indicates the order of precedence in which logical expressions are evaluated (NOT has highest priority). As with arithmetic operations, you can override this order of precedence by the use of parentheses. Consider the following examples:

```
IF SKY$ = "clear" AND TEMP > 70 OR HUMIDITY < 75
THEN PRINT "Let's have a picnic"

IF SKY$ = "clear" AND (TEMP > 70 OR HUMIDITY < 75)
THEN PRINT "Let's have a picnic"
```

In the first example, the picnic invitation is made regardless of the appearance of the sky and the temperature, just as long as the relative humidity is below 75%. The second example would appear to be a much safer weather basis: either the temperature or the humidity (or both) must be favorable, and the sky must be clear at any rate.

The following is a detailed discussion of how GW-BASIC determines the result of a logical operation. It is not necessary to understand this process in order to program with GW-BASIC. The discussion is of interest mainly to programmers working at bit level.

Logical operators work by converting their operands to sixteen bit, signed, two's complement integers in the range -32768 to +32767. (If the operands are not in this range, an error results.) If both operands are supplied as 0 or -1, logical operators return 0 or -1. The given operation is performed on these integers in bitwise fashion; i.e., each bit of the result is determined by the corresponding bits in the two operands.

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator may be used to "mask" all but one of the bits of a status byte at a machine I/O port.

The OR operator may be used to "merge" two bytes to create a particular binary value. The following examples demonstrate how the logical operators work.

63 AND 16=16	63 = binary 111111 and 16 = binary 10000, so 63 AND 16 = 16 (binary 10000)
15 AND 14=14	15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 (binary 1110)
-1 and 8=8	-1 = binary 1111111111111111 and 8 = binary 1000, so -1 AND 8=8 (binary 1000)
4 OR 2=6	4 = binary 1000 and 2 = binary 10 so 4 OR 2 = 6 (binary 110)
10 OR 10=10	10 = binary 1010, so 1010 OR 1010 = 10 binary 1010

You can use GW-BASIC to work out the two's complement of an integer:

```
TWOSCOMP% = (NOT INTGER%)+1
```

Given that INTGER% contains 2 (= binary 10), NOT INTGER% produces the bit pattern 11111111 11111101. This is -3 in decimal. TWOSCOMP% is therefore assigned the value -2 (the result of adding 1 to -3). The general expression for calculating the two's complement of an integer is "the bit complement plus one".

FUNCTIONAL OPERATORS

A function is used in an expression to call a predetermined operation that is to be performed on an operand. GW-BASIC has "intrinsic" functions that reside in the system, such as SQR (square root) or SIN (sine). All of GW-BASIC's intrinsic functions are described in Chapter 4.

GW-BASIC also allows "user defined" functions that are written by the User. See DEF FN, Chapter 4.

EVALUATION OF EXPRESSIONS

This section summarizes the precedence of numeric operations; that is, the order in which GW-BASIC evaluates them within an expression.

1. Function calls (regardless of whether defined in your program or already provided by GW-BASIC) are evaluated first.

2. Then arithmetic operations are carried out in the following order:

   a. ^
   b. unary —
   c. *,/
   d. \
   e. MOD
   f. +,-

3. Relational operations are performed next.

4. Finally, logical operations in the following order:

   a. NOT
   b. AND
   c. OR
   d. XOR (exclusive OR)
   e. EQV (equivalence)
   f. IMP (implication)

Operations at the same precedence level are performed from left to right. To change the order of precedence for a particular expression, use parentheses: operations enclosed within parentheses are performed first; within parentheses, the usual order of evaluation (as detailed above) is observed.

STRING OPERATIONS

Strings may be concantenated (joined together) using +. For example:

```
10 A$="FILE":B$="NAME"
20 PRINT A$ + B$
30 PRINT "NEW " + A$ + B$
RUN
FILENAME
NEW FILENAME
```

Strings may be compared using the same relational operators that are used with numbers:

```
= <>  <  >  <=  >=
```

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If, during string comparison, the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant. Examples (the result is "true" in each case).

```
"AA"<"AB"
"FILENAME" = "FILENAME"
"X&">"X#"
"Cl">"CL"
"kg">"KG"
"SMYTH"<"SMYTHE"
B$<"256" (where B$ contains the string "1234")
```

Thus, string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

EXERCISES

You have now read quite a lot about the precedence of operators. Try performing some mathematical operations with a particular view to observing the effects of overriding the order of precedence by the use of parentheses. For example, observe the difference of result between

```
PRINT 5 + 6 * 12
```

and

```
PRINT (5 + 6) * 12
```

Enter these commands in the direct mode for quick results. You can even include GW-BASIC variables in an expression, but this makes sense only if you give them a value first; otherwise, GW-BASIC will assign them the value 0. Enter (in direct mode)

```
X = 12
Y = 14
```

and then a command (likewise in direct mode) to resolve an expression which uses these two variables, for example

```
PRINT Y MOD X + 3
```

The next example will remind you of the arithmetic you did at school, before you learned about decimal points. It demonstrates how you can use GW-BASIC's integer and modulus division operators to produce a result in the form of quotient and remainder.

The program makes use of "error trapping". When GW-BASIC notices during program execution that there is a syntactical error, or that a command or function is not being used correctly (for example, division by zero, or subscript out of range), a message is displayed to that effect. You can intercept non-syntactical errors by means of questioning the contents of a variable and making a branch in the program dependent on the answer. In the last set of exercises, in the program which stored a list of names and then recalled these names in response to your entering numbers, there was nothing to prevent you from entering a number higher than the maximum allowable subscript for the array. One way of intercepting such an erroneous entry in that program would be the following program line:

```
105 IF N% > 10 THEN PRINT "There aren't that many
    names":GOTO 80
```

This has the effect that if you enter a number higher than 10, first your own error notification is displayed, then GW-BASIC returns to line 80 in expectation of a legal entry. In this way, your program prevents GW-BASIC from attempting in line 110 to reference an array with a sub script that is too high.

GW-BASIC provides a more comfortable error trapping facility, which is used in the following program. Line 10 tells GW-BASIC that in the event of an error occurring anywhere in the program, GW-BASIC should branch to line 100. The types of errors that can occur in this program would result from entering an integer which is outside the allowable range for integers (resulting in an Overflow error), or trying to make GW-BASIC use 0 as a divisor (Division by Zero error). Line 110 deals with the former error, but there is no need to deal with the latter as division by zero does not force GW-BASIC to stop the program. The numbers 6 and 11 are codes GW-BASIC uses to denote these two error situations. Appendix C of this manual gives a list of all error possibilities GW-BASIC can detect. It is particularly important to trap the integer overflow error, as this error would otherwise lead to termination of the program. The RESUME 30 command tells GW-BASIC to consider the error situation as dealt with, and to resume normal program execution at line 30.

You may already have noticed that this program gives you no possibility of breaking out of the cycle of events between lines 30 and 70. This gives you the opportunity of practicing use of the <Ctrl-Break> key combination. It terminates a program and returns GW-BASIC to the "Ok" level. (You will probably make frequent use of this facility when developing programs to break out of an interminable loop.)

```
 10 ON ERROR GOTO 100
 30 INPUT "Number to be divided";Q%
 40 INPUT "Now enter the divisor";D%
 50 PRINT:PRINT "The answer is ";Q%\D%;", remainder; "Q%
    MOD D%
 60 PRINT
 70 GOTO 30
100 CLS
110 IF ERR=6 THEN PRINT "Outside the integer range! Try
    again"
130 RESUME 30
```

The section "String Operators" demonstrated that it is possible to compare not only numbers, but also strings. The outcome of the

comparison of two strings depends on a character-by-character comparison on the basis of ASCII values. This has the effect that, seen as strings, "345" is greater than "12345", because the ASCII code for the digit "3" is higher than that for the digit "1".

The following program asks you to input 10 names (lines 10 to 50). These names are stored in the array NNAME$. The screen is cleared, and the program proceeds to sort the names in rising ASCII sequence (lines 70 to 110). There are many known procedures for sorting data. This is one of the simplest, namely that of going through the list, comparing pairs of adjacent entries and carrying out an exchange where necessary. The program checks (line 110) the variable EXCH$ for a "Y" for yes or an "N" for no, to see whether an exchange was necessary on the most recent run through the list. If an exchange was necessary, another sorting run through the list is undertaken. After a clear run in which no exchange is necessary, GW-BASIC proceeds to print the list in the new order.

Remember that the sorting procedure takes into account the full ASCII code. It is not confined to letters. The program does not prevent you from entering non-alphabetical characters, nor does it stipulate whether you are to use uppercase or lowercase letters. For example, you could enter the "name" "!unusual". The sorting procedure would place this nearer the beginning of the list than any name starting with a letter, because the ASCII value for the exclamation mark is 57, and the lowest code occupied by a letter is 65 (uppercase A).

The actual exchange is carried out in a "subroutine" at line 180. This subroutine is entered (if the condition for exchanging is fulfilled) by means of the GOSUB command in line 90. The RETURN command tells GW-BASIC to go back to the command which immediately follows the one which sent GW-BASIC to the subroutine, in this case, to line 100. It would have been possible here to use a command GOTO 180 to enter the name exchanging routine, and GOTO 100 to enable GW-BASIC to find its way back. The important difference between the two methods is that the RETURN (can be used only with GOSUB) is able to put GW-BASIC back to the right place, without having to state a line number. This is a particularly useful facility in programs where a subroutine is entered from different places in the program.

The greater part of the "swapping" routine is concerned with displaying on the screen the pair of names being exchanged. The loop at line 200 presents a delay, in that GW-BASIC must pass through this loop 600 times before RETURNing to compare the next pair of names. The loop itself contains no commands, so GW-BASIC is in

effect "running on the spot", in order to give you time to read the screen before displaying the next swapped pair. If this time is too short, increase the number of times GW-BASIC must pass through the loop. You should bear in mind that displaying the swapped pairs considerably increases the time GW-BASIC requires for the total sorting procedure. To increase sorting speed, simply delete lines 180 and 200.

Line 170 does not contain any commands for GW-BASIC, but is a programmer's remark. As soon as GW-BASIC sees the word REM, it knows that it does not have to read the rest of that line, so you can write what you like in it. Here it is used for two purposes: the asterisks make a clear division between the subroutine and the main program flow, thus making the program easier for the programmer to read; the subsequent note refers to the purpose of the subroutine.

```
 10 OPTION BASE 1:DIM NNAME$(10):CLS
 20 FOR LOOP% =1 TO 10
 30 PRINT "Enter somebody's name";
 40 INPUT NNAME$(LOOP%)
 50 NEXT LOOP%
 60 CLS
 70 EXCH$="N"
 80 FOR LP% =1 TO 9
 90 IF NNAME$(LP%)>NNAME$(LP% +1) THEN
    EXCH$="Y":GOSUB 180
100 NEXT LP%
110 IF EXCH$="Y" THEN CLS:GOTO 70
120 PRINT "In 'ASCII' order":PRINT
130 FOR LP% =1 TO 10
140 PRINT NNAME$(LP%)
150 NEXT LP%
160 END
170 REM ********** subroutine — display/exchange ele-
    ments of array
180 PRINT "Swapping ";NNAME$(LP%);" and
    ";NNAME$(LP% +1)
190 SWAP NNAME$(LP%),NNAME$(LP% +1)
200 FOR DLY% =1 TO 800:NEXT DLY%
210 RETURN
```

Full Screen Editor

This section describes the use of the keyboard of your computer in relation to writing and editing GW-BASIC programs. If you are not already familiar with the layout and basic operations of the keyboard, you should first read the appropriate description in your OWNER'S MANUAL.

GW-BASIC provides a comfortable Full Screen Editor to enable you to create and alter programs. Editing is not confined to the program line currently being written. Instead, you can use the special cursor movement keys to place the cursor anywhere on the screen, and delete or add to the program at the point indicated by the cursor position. As soon as you have edited a program line, press the <ENTER> key before moving the cursor away from that line. GW-BASIC then registers the new contents of that line.

NOTE: The GW-BASIC command NEW clears the computer memory of any GW- BASIC program, but leaves GW-BASIC itself intact. NEW is used to ensure a clean memory before beginning to edit a new program.

Cursor movement is performed by a single-key action. The direction of movement is marked on each of the four keys concerned. If a left cursor move pushes the cursor off the edge of the screen, it re-appears at the far right of the line above. If a cursor right move pushes the cursor off the edge of the screen, it re-appears at the far left of the line below.

The keyboard functions supported by GW-BASIC are given below. A number of these are 2-key actions involving the use of the Ctrl key.

Key	Function
Home	The cursor moves to the top left-hand corner of the screen.
<Ctrl-Home>	The screen is cleared, and the cursor moves to the top left-hand corner.

↑	The cursor moves up one line.
↓	The cursor moves down one line.
←	The cursor moves to the left .
→	The cursor moves to the right.
<Ctrl-→>	The cursor moves to the beginning of the next word on the right. The beginning of a word is defined as the first letter or digit which follows a blank or special character (e.g. punctuation mark).
<Ctrl-←>	The cursor moves to the beginning of the previous word on the left.
End	The cursor moves to the end of the logical line, that is, to the end of the GW-BASIC line (which may extend beyond the limit of one screen line). This function is especially useful for adding to an existing program line.
<Ctrl-End>	The logical line from the current cursor position to the end of the line is erased.
Ins	This key acts as a "toggle" for the insert mode: if you press this key while the insert mode is off, this mode is turned on, and vice versa. Insert mode on means that what you type in now pushes existing characters to the right. If there is no more space in the same screen line, characters are pushed over to the next line. There is no loss of characters, either in the cursor line or in subsequent lines. While insert mode is on, the cursor covers the lower half of the character position. When insert mode is off, what you type in writes over (replaces) existing characters. In addition to the toggle effect mentioned above, pressing a cursor movement key or the <ENTER> key turns the insert mode off.

Del	The character at the current cursor position is deleted. Characters to the right of the cursor move to the left to close up the space. The closing up procedure has effect throughout the entire logical line following the current cursor position.
←	The backspace key. The character immediately to the left of the cursor is deleted. The space thus created is closed up as with the delete function.
Esc	If you press this key, the entire logical line in which the cursor is situated is removed from the screen. If, however, the line has already been passed to GW-BASIC (<ENTER> key action), it is not deleted from the program. (To delete a line from the program, simply type the line number and then press <ENTER>, or use the DELETE command.)
<Ctrl-Break>	GW-BASIC returns to the command level ("Ok") without saving any changes to the line currently being edited. The line does not disappear from the screen. (The procedure for passing the newest version of a line to GW-BASIC is simply to press <ENTER>.)
< ⇥ >	When the insert mode is off, this key has the effect of moving the cursor to the next tab stop, without displacing any characters. Tab stops are situated every eight character positions in a line, that is, at positions 1,9,17,25 etc. When the insert mode is on, blanks are inserted from the current cursor position up to the next tab stop. Text displacement takes place as with the insert function (see above).

<↵> <ENTER>	When you press <ENTER> key, the program line in which the cursor is situated is passed to GW-BASIC. Pending subsequent changes (likewise notified to GW-BASIC by pressing <ENTER>, this is the line as it will be seen by GW-BASIC when your program is executed.
<Ctrl-↵>	This key action produces a so-called line feed; that is, the cursor drops one line, but the program (logical) line is not yet passed to GW-BASIC. The blanks thus created at the end of the upper line are of no consequence to the contents of the program. This function is especially useful for creating line divisions to make a program easy to read.
<Ctrl-PrtSc>	This key combination is not quite the same as the standard <Shift-PrtSc> function. <Ctrl-PrtSc> directs to the printer a copy of everything appearing on the screen, not only during editing but also during program execution. This copying function remains in force until you press <Ctrl-PrtSc> again.

The most effective way of learning the various editor functions is to practice them. Simply load GW-BASIC in the usual way and write some program lines. Remember, each program line consists of a line number followed by at least one blank, which is followed by the program text. The lines you write while practicing the Full Screen Editor need by no means be perfect GW-BASIC syntax (syntax checking is done at the time of program execution). You will probably not want to try a program RUN. You could use the examples from the exercises in Chapter 1, building in mistakes, and then correcting them.

Here are a few suggestions about how to make program writing and editing more comfortable:

- Remember, only when you press <ENTER> is a program line passed to GW-BASIC. Regardless of where in the line the cursor is when you press <ENTER>, the whole logical line is passed to GW-BASIC. It is not necessary to move the cursor to the end of the line.

- To erase a line, simply enter the number of the line, or use DELETE. The DELETE command is useful for deleting a number of consecutive lines.

- LIST enables you to view part or all of a program. LIST uses uppercase letters wherever appropriate (e.g. for GW-BASIC reserved words) and displays the program lines in ascending numerical order. If you are viewing a large number of lines at once, you will find that the screen scrolls far too quickly for comfortable reading. Pressing the combination <Ctrl-Num Lock> suspends scrolling, giving you time for detailed reading. To resume scrolling, simply press any character key. (The same key combination also suspends program execution. This enables you to read on the screen long lists or other texts produced by a program).

- Be certain to press <ENTER> after you have corrected each line of coding using the editing keys. This action passes the corrected line to GW-BASIC.

- A program may sometimes require a number of lines with almost identical contents. In this case, it is easy to duplicate a line and then make the small changes necessary within the duplicate line.

  To duplicate a line, move the cursor to the beginning of that line. Then overwrite the line number with the new line number for the copy. When you press <ENTER>, the copy is passed to GW-BASIC; the original line is unaffected.

- To change a line number, make a copy of the line as described above and then delete the original line.

- The AUTO command provides you with line numbers, thus saving you the work of typing them. Before editing lines other than the current one, you should turn AUTO off by pressing <Ctrl-Break>.

- A number of GW-BASIC reserved words can be typed by means of Alt key combinations; for example, the key combination <Alt-P> produces the word PRINT on the screen. The following is a complete list of these special Alt key functions:

A AUTO	N NEXT
B BSAVE	O OPEN
C COLOR	P PRINT
D DELETE	Q not used
E ELSE	R RUN
F FOR	S SCREEN
G GOTO	T THEN
H HEX$	U USING
I INPUT	V VAL
J not used	W WIDTH
K KEY	X XOR
L LOCATE	Y not used
M MID	Z not used

- The ten Function Keys on the keyboard are programmed with GW- BASIC reserved words as soon as GW-BASIC is loaded into computer memory. These are commands that are especially useful in the direct mode and, therefore, are already supplied, with <ENTER> where appropriate. To activate one of these, simply press the corresponding Function Key:

F1 LIST	F2 RUN **<ENTER>**
F3 LOAD"	F4 SAVE"
F5 CONT **<ENTER>**	F6 ,"LPT" **<ENTER>**
F7 TRON **<ENTER>**	F8 TROFF **<ENTER>**
F9 KEY	F10 SCREEN 0,0,0 **<ENTER>**

GW-BASIC uses the bottom line of the screen display to show the contents of these Function Keys. You can turn this part of the display on and off, as well as change the contents of these keys, by means of the KEY statement.

- If you wish to leave important information on part of the screen and use only the other part for program editing, you should refer to the description of the VIEW statement in Chapter 4.

- Program alterations take place in the computer memory. They are not stored on disk until you issue an explicit SAVE command. When writing a long program, issue frequent intermediate SAVE commands. Normally, you will use the same filename each time, so that previous, less complete versions of the program do not clutter your disk. A program on disk is preserved even in the event of a power shut-down.

Screen Display

GW-BASIC is capable of directing the screen to display text (including the special symbols described in "The Character Set", Chapter 1) and draw points and geometric figures. GW-BASIC operates in either of two modes, character mode and graphics mode.

CHARACTER MODE

In character mode, the software considers the screen to have 25 lines (from top to bottom). Each line can accommodate 40 or 80 characters from the GW-BASIC character set. (You can set this by means of the WIDTH command.)

GW-BASIC regards the top leftmost character position as line 1, column 1; the character position in the bottom right-hand corner of the screen is 25,80 (assuming your program selects 80 columns per line). The 25th line is accessible to a GW-BASIC program, but it is not scrolled by GW-BASIC. This line normally displays the current contents of the Function Keys of your keyboard.

If your machine has a Monochrome Display Adapter, you have program control over the following screen attributes: display intensity, image inversion (dark on light and vice versa), underscoring, and blinking. On a Color Graphics Display Adapter machine, foreground and background colors can be set.

The foreground and background colors determine how an individual character area is displayed. The foreground is the character itself, the background is the small rectangular area which surrounds it.

The colors available in character mode are:

0 Black	8 Gray
1 Dark Blue	9 Light Blue
2 Dark Green	10 Light Green
3 Cyan	11 Light Cyan
4 Red	12 Light Red
5 Magenta	13 Light Magenta
6 Brown	14 Yellow
7 White	15 High-intensity White

The COLOR and SCREEN commands, with which screen attributes are determined, are also used for a monochrome display. The obvious limitation is that only two colors, black and green, can be displayed.

GRAPHICS MODE

Graphics mode is more sophisticated. To allow you to draw pictures and other shapes, the software considers the screen to be made up of points. (These points, or picture elements, are often described as "pixels".) The number of pixels per screen determines the degree of resolution. By means of the SCREEN command, you can choose between three modes of resolution, namely low, medium, and high resolution.

Low Resolution

This is the display mode that considers the screen as 200 horizontal pixel lines, each containing 320 pixels. The screen consists of 25 lines with 40 characters.

Depending on a color or monochrome display being used, you can set different colors for the foreground (the character itself) and the background (the screen) or use up to 4 gray scales.

Medium Resolution

In medium resolution the screen has 640 pixels across and 200 pixels down the screen. Each of the 25 lines on the screen accommodates 80 characters. This display mode can use only black and white, regardless of whether you have a color or a monochrome screen.

High Resolution

Again there are 25 lines per screen, each containing 80 characters.This display mode regards the screen as consisting of 400 horizontal pixel lines with 640 pixels each. There are two different high resolution modes. In high resolution black-and-white graphics (SCREEN 3) the screen display is supported only in black and white. The high resolution color graphics mode (SCREEN 4) is similar to low resolution graphics. Here, too, colors can be selected as desired, or, on a monochrome display, up to four gray scales can be used.

X and Y Coordinates

The widely used convention for addressing points on a graphics display is the use of x and y coordinates. The x coordinate is the horizontal position on the screen, the y coordinate is the vertical position. 0,0 is the first pixel position in the upper left-hand corner of the screen (origin). Usually, you can specify the coordinates in either of two forms: an absolute form where x,y specify the exact position, or an offset form where x,y are the offset values from the last point referenced. When specifying the coordinates in offset form, you must include the word STEP to let the software know you are "stepping" from the previously established point. For example, the following GW-BASIC command locates and illuminates a point near the center of the screen on a high resolution screen:

```
PSET (320,199)
```

If the next point you wish to illuminate is known in relation to the previous point (for example, you wish to illuminate the pixel 6 points to the right of and 4 pixels below the last point referenced), you can use the following addressing technique:

```
PSET STEP (6,4)
```

This saves you the trouble of calculating the absolute coordinates in relation to the 0,0 origin. If the new point is to the left of or above the last point addressed, appropriate minus values are required.

If you have a mathematical background, you will have noticed that this coordinate scheme does not use Cartesian coordinates. However, GW-BASIC includes a command (WINDOW) to enable you to define your own coordinate scheme, which can be the Cartesian scheme, if

you so wish. The foregoing discussion and the following introduction to the graphics modes refer to the screen in terms of the coordinates as initially set by GW-BASIC, that is, with the origin in the top left corner and the maximum y value at the bottom of the screen.

Color Selection in Graphics Mode

If you have a color screen, you can select different colors for the foreground (the character or graphics image) and the background (the screen itself). You can choose between two palettes, each containing three colors designated 1,2 and 3. The palettes contain the following colors:

Palette 0	Palette 1	Color
Green	Cyan	1
Red	Magenta	2
Brown	White	3

Furthermore, it is possible to switch from one palette to the other, whereupon the colors on the screen change to the colors of the newly selected palette. In addition to the palette colors, you can assign a color of your choice to the background color (color 0), which is independent of palette switching, and which can be any of the 16 colors available in character mode.

The fact that this is a graphic, and not the character mode, does not prevent you from calling on the GW-BASIC character set. With a color screen, the characters are written in Color 3 of the palette you have chosen, the background color is the one you have selected for Color 0.

	Color Display	Monochrome Display
Low Resolution		
SCREEN 1	4 colors	4 gray scales
Medium Resolution		
SCREEN 2	black/white	black/white
High Resolution		
SCREEN 3	black/white	black/white
SCREEN 4	4 colors	4 gray scales

Character Support in Graphics Mode

To ensure full character support for non-United States characters, the GRAFTABL.COM file must be loaded into memory. This is done by entering the GRAFTABL command, which is described in your NCR-DOS manual.

EXERCISES

The following examples are for use with a color display only. The first example displays the colors available, both in steady and blinking form. The colors, with the exception of black and gray, are displayed on a black background. (You could make writing invisible by choosing the same color for foreground and background.)

This program uses the character mode of display. The character displayed is the one with the code value 219 (a rectangle filling the entire space of one character). The STRING$ function in line 10 sets up the string variable B$ with 40 such rectangles. Line 20 sets the background color to black and display width to 80 characters. The remainder of the program takes the basic colors (0 to 7) one by one, first displaying the basic color itself (line 50), then the same color blinking (line 70), then the "bright" version of the color (line 90), and finally the bright version blinking. In each case, blinking is achieved by adding 16 to what would otherwise be the value for the color. The line width of 80 results in the four versions for each color being displayed over two lines.

```
 5 SCREEN 0
10 B$=STRING$(40,CHR$(219))
20 COLOR ,0:WIDTH 80
30 FOR LP% =1 TO 7
40 COLOR LP% ,0
50 PRINT B$;
60 COLOR LP% +16,0
70 PRINT B$;
80 COLOR LP% +8,0
90 PRINT B$;
100 COLOR LP% +24,0
110 PRINT B$;
120 NEXT LP%
125 COLOR 7,0
130 END
```

The following program demonstrates just some of the graphic possibilities offered by medium resolution graphics on a color display. The program carries out point by point drawing on the screen, under control of the numeric key pad on the right of the keyboard. You can determine whether the GW-BASIC screen coordinates (used so far in this chapter) or true Cartesian coordinates are to apply to graphic drawing. By changing a single program line you can even determine a different point of coordinate origin on the screen. The program

enables you to change the color palette and select a color within the chosen palette.

The program variables are used as follows (all numeric variables hold integer values (line 20)):

EXX, WYE	The X and Y coordinates of the currently addressed point on the screen.
X1,Y1 X2,Y2	The X and Y coordinates of two diagonally opposed corners of the screen. Used in the WINDOW command, these effectively define the number of addressable points along the horizontal and vertical axes of the screen. In the WINDOW SCREEN statement, X1, Y1 refers to the top left corner of the screen, while X2, Y2 refers to the bottom right corner. If the word SCREEN is omitted from the WINDOW statement, the Cartesian coordinate system applies: X1, Y1 is the bottom left corner, while X2,Y2 is the top right corner.
PAL	Contains 0 or 1 for the current palette.
COL	Contains 1, 2, or 3 for the color being used from the current palette.
K$	Stores one character read from the keyboard by means of the INKEY$ function. GW-BASIC remains in the keyboard reading routine (lines 330 to 360) until a key is pressed.
C$	If this variable contains "c", it indicates that Cartesian coordinates are being used (lines 60,160,170).
ERON$,EROFF$	Initially, ERON$ contains "N" for no, and EROFF$ contains "Y" for yes. Line 150 checks whether ERON$ has assumed

the value "Y". If this is the case, points on the screen are erased and not illuminated. The values of these two variables are exchanged (line 310), whenever the numeric key 5 is pressed during point by point drawing.

B$ — A string of 30 blanks, used for overwriting screen messages issued by the program in screen line 25.

Upon entering RUN, you are asked to enter "c" if you wish to use Cartesian coordinates, otherwise press any key. The screen is then cleared and the medium resolution graphics mode is set with color enabled (line 50). Following this, one of two screen windows is set up, according to the coordinate system selected (line 60). The values used for the numeric range of the coordinates which are to represent the height and width of the screen are determined by the values assigned to the variables X1,Y1 and X2,Y2 in line 40. The values given here make use of the maximum display definition available in medium resolution graphics. The origin (0,0) is as near to the center of the screen as possible. The alternative values suggested by the REMark line 30 would place the origin at the top or bottom left corner of the screen, depending on whether Cartesian or SCREEN coordinates apply. You might like to try these later (simply delete the word REM from line 30, and insert this word at the beginning of line 40). You could even produce a kind of horizontal or vertical exaggeration of a drawing, by varying the proportion between the X and the Y values.

Having ensured that the Num Lock key enables the numeric (not the cursor movement) functions, you can use the cluster of keys around the 5 on the numeric key pad to illuminate points in 8 directions (8 illuminates the point to the North, 9 the point North-East of the last point addressed, and so on). Using the window values in line 40, the very first point addressed is in the center area of the screen, so you will not start by getting lost! If you press the 5 key, an illuminated point still moves over the screen in accordance with your operation of the numeric key pad, but it does not leave a trail. This is so that you can go to a new drawing position. This suppression of drawing remains in force until you press the 5 key again.

The entire screen movement is carried out in lines 140 to 300. The PSET and PRESET statements are used to plot and erase points, respectively. You may wish to refer to Chapter 4 for the full

descriptions of these commands, the ON GOTO command, and the VAL function.

The coordinates of the point on the screen currently addressed are displayed at the bottom of the screen in the format X,Y (line 290). There is nothing to prevent you from falling off the edge of the screen world, but a BEEP will tell you if this has happened (line 270).

The background color is initially set to black, the drawing color to magenta (color 2 of palette 1, line 90). If, instead of pressing a number key (1 to 9), you press lowercase c, this indicates to the program that you wish to change the drawing color (line 130). This is carried out in a subroutine in lines 370 to 500: a number 0 or 1 is accepted as the palette number, a number 1, 2, or 3 as the color from that palette. If you change palettes, the whole drawing created so far changes color accordingly (green <-> cyan, red <-> magenta, brown <-> white). After completion of the color change, drawing control is returned to the numeric key pad.

If you press lowercase x during drawing control, GW-BASIC branches to line 510, whereupon the character display mode is restored, with a line width of 80 characters. You might wish to insert a selection of further graphic functions at this point, for example, CIRCLE drawing or area PAINTing. You could even save your drawing in an array variable and eventually on disk. Chapter 4 provides all the programming details you need in order to put to best use the graphic power of GW-BASIC.

Cartesian or SCREEN coordinates?

```
10 INPUT "c for Cartesian";C$
```

All numeric variables integer

```
20 DEFINT A-Z
```

WINDOWs for origin in corner and origin in center

```
30 REM X1=0:Y1=0:X2=319:Y2=199
40 X1=-160:Y1=-100:X2=159:Y2=99
```

Medium resolution, color enabled

```
50 SCREEN 1,0
```

Set WINDOW according to whether or not Cartesian coordinates

```
60 IF C$<>"c" THEN WINDOW SCREEN(X1,Y1)-(X2,Y2):
   ELSE WINDOW (X1,Y1)-(X2,Y2)
```

Initially, points to be illuminated, not erased

```
70 ERON$="N":EROFF$="Y"
```

String of 30 blanks, used for erasing "Palette?" and "Color number?"

```
80 B$=STRING$(30," ")
```

Black background, palette 1, initial plotting color is magenta

```
90 COLOR 0,1:COL=2
```

First point addressed is the origin

```
100 EXX=0:WYE=0
```

GW-BASIC returns here after any screen movement or new color setting

```
110 GOSUB 330
120 IF K$="x" THEN GOTO 510
130 IF K$="c" THEN GOSUB 370:GOTO 110
140 IF K$<"1" OR K$>"9" THEN GOTO 110
150 IF ERON$="Y" THEN PRESET (EXX,WYE)
160 IF C$<>"c" THEN ON VAL(K$) GOTO 180,190,200,
    210,220,230,240,250,260
170 IF C$="c" THEN ON VAL(K$) GOTO 240,250,260,
    210,220,230,180,190,200
180 EXX=EXX-1:WYE=WYE+1:GOTO 270
190 WYE=WYE+1:GOTO 270
200 EXX=EXX+1:WYE=WYE+1:GOTO 270
210 EXX=EXX-1:GOTO 270
220 GOTO 310
230 EXX=EXX+1:GOTO 270
240 EXX=EXX-1:WYE=WYE-1:GOTO 270
250 WYE=WYE-1:GOTO 270
260 EXX=EXX+1:WYE=WYE-1:GOTO 270
270 IF EXX>X2 OR EXX<X1 OR WYE>Y2 OR WYE<Y1
    THEN BEEP
280 PSET (EXX,WYE),COL
```

Display current coordinates, then return for next keyboard input

```
290 LOCATE 25,1:PRINT EXX;", ";WYE;
300 GOTO 110
```

GW-BASIC arrives here only if the 5 key has been pressed

```
310 SWAP ERON$,EROFF$
320 GOTO 110
```

Subroutines

```
330 REM ***** read keyboard
340 K$=INKEY$
350 IF K$="" THEN GOTO 340
360 RETURN
370 REM ***** set color
380 LOCATE 25,1
390 PRINT "palette?              ";
400 GOSUB 330
410 IF K$<"0" OR K$>"1" THEN GOTO 380
420 PAL% =VAL(K$)
430 LOCATE 25,1
440 PRINT "color number?         ";
450 GOSUB 330
460 IF K$<"1" OR K$>"3" THEN GOTO 430
470 COL% =VAL(K$)
480 COLOR ,PAL
490 LOCATE 25,1:PRINT B$:
500 RETURN
```

Pressing x while drawing sends GW-BASIC here

```
510 REM ***** other functions?
520 SCREEN 0:WIDTH 80
530 END
```

Statements, Commands and Functions

This chapter contains a detailed description of how each statement, command and function works. You may have heard of the term "statement" as well as "command" in relation to computer programming languages. The tradition which underlies this distinction is that statements are instructions within a program that are carried out at the time of execution, whereas commands are used to work on programs prior or subsequent to execution in order to, for example, load, edit, save, and run a program. This is basically the difference between communicating with GW-BASIC in direct and indirect mode.

A function converts a value into some other value by means of a fixed formula. The functions described in this chapter are built-in, or "intrinsic" to GW-BASIC. These functions may be called from any program without further definition. GW-BASIC cannot process a function on its own. It also needs a command to tell it what to do with the result, for example, display it on the screen, send it to a printer, or assign it to a variable. An argument to a function, that is, the value which the function is to work on, is always enclosed in parentheses in GW-BASIC.

If you supply a floating point value for a function where an integer value is required, GW-BASIC rounds the fractional part and uses the resulting integer. If you specify the /D option when loading GW-BASIC, the functions ATN, COS, EXP, LOG, SIN, SQR, and TAN are calculated to double precision. Otherwise single precision is used. See *Appendix D* for information about mathematical functions that are not intrinsic to GW-BASIC.

The following pages present a brief summary of all the GW-BASIC statements, commands and functions. First they are listed in groups. Each group has a title denoting the purpose common to the statements, commands and functions listed under it. Each statement, command and function is described briefly, leaving aside details of syntax. Thus, if you are a newcomer to GW-BASIC, you will be able to

find the element of GW-BASIC that conforms to a particular programming situation, simply by glancing over these few pages. You can then refer to the complete descriptions consituting the main part of this chapter.

The headings, in order of presentation, are:

GW-BASIC Management
Program Editing
Loading and Storing Programs
File Processing
The Keyboard and Other Non-Disk Input
Characters on Screen or Printer
Graphics
The Loudspeaker
Program Variables and Type Conversion
String Manipulation
Mathematical Functions
Decision Making and Branching
Event Trapping
Other Commands and Functions

A number of especially versatile statements, commands and functions appear under more than one heading.

GW-BASIC Management

CLEAR	Clears program variables, and optionally delimits the area of memory available to GW-BASIC.
CONT	Continues execution of a program after a break.
DEF SEG	Defines a segment of storage in memory.
DEF USR	Defines the starting address of a machine language program.
END	The program stops, all files are closed, "Ok" is displayed.
FRE	A function returning the amount of memory space currently not being used by GW-BASIC.

NEW	Clears programs and their variables from memory, but leaves GW-BASIC intact.
RANDOMIZE	Sets the starting point (seed) of the random number sequence used by the RND function.
RUN	Loads and begins execution of a program, or begins execution of a program already in memory from a specified program line.
SHELL	Execute an NCR-DOS command file and then return to GW-BASIC.
STOP	Terminates program execution, displaying the number of the terminating program line on the screen.
SYSTEM	Closes all files and returns to NCR-DOS system level.
TRON, TROFF	Switches the program testing (trace) facility on and off.
VARPTR$	A function returning the address in computer memory of a specified variable. This information is sometimes required by machine language programs.
WAIT	GW-BASIC suspends program execution until a specified value is present at a specified port.

Program Editing

AUTO	Generates program line numbers automatically, thus saving you the trouble of typing them.
DELETE	Deletes one program line or a number of consecutive program lines.

EDIT	Writes a program line on the screen, so that you can edit it.
LIST	Lists program lines on the screen, or directs them to a file.
LLIST	As LIST, except that the program lines appear on a printer.
NEW	Clears the computer memory, including any programs, but not GW-BASIC itself.
RENUM	Renumbers program lines.
KEY	Sets a Function Key on the keyboard.
KEY ON/OFF/LIST	Turns display of Function Key contents on and off.

Loading and Storing Programs

BLOAD	Loads binary data into memory. Used especially with machine code programs.
BSAVE	Saves binary data on disk.
LOAD	Loads a program file from disk. Optionally, the program is executed immediately.
MERGE	Merges a program from disk with a program already in memory.
SAVE	Saves a program on disk, optionally in a protected format.

File Processing

CHDIR	Changes the current directory.
CLOSE #	Closes a file to program access.

ENVIRON	Changes Operating System environment parameters. Can be used to access a program in another directory. As a function it returns these parameters.
EOF	A function which returns information as to whether the end of a specified file has been reached.
GET #	Reads a record from a random file.
FIELD #	Defines a field in a random file buffer, thus enabling different parts of a record to be assigned to different program variables.
FILES	Looks for a specific file or group of files in the disk directory, and if found, displays the filename(s).
INPUT #	Reads data from a file and immediately assigns it to one or more variables.
KILL	Erases a disk file.
LINE INPUT #	Reads an entire line from a file.
LOC	Returns information about the current state of processing of a random, sequential, or communications file.
LOF	A function returning the length of a specified file.
MKDIR	Creates a directory.
NAME...AS...	Renames a disk file.
OPEN OPEN...FOR..AS	A versatile command used for opening files for program access.
OPEN "COM	A special form of the OPEN command, opening a file for communications.

PRINT # PRINT #..USING	Writes a list of expressions or data to a file.
PUT	Writes data from a random file buffer in memory to a file.
RESET	Closes all disk files.
RMDIR	Removes a directory.
VARPTR(#)	A function returning the address of the file control block of a specified file.
WRITE #	Directs the output of data to a file.

The Keyboard and Other Non-Disk Input

DATA	A list of data items to be read sequentially by the READ command.
INKEY$	A function which reads a character from the keyboard.
INP	A function returning one byte read at a specified port of the computer. Output to a port is by means of the OUT command.
INPUT	Reads input from the keyboard into a variable.
KEY	Sets a Function Key.
KEY ON/OFF/LIST	Turn display of Function Key contents on and off.
LINE INPUT	Reads one line of input from the keyboard, ignoring commas and similar delimiters.
ON KEY...GOSUB	GW-BASIC transfers program control to the subroutine at a specified line number, when a specified cursor movement key or Function Key has been pressed.

PEN	A function returning light pen coordinates.
PEN ON/OFF/STOP	Switches the light pen function on and off.
READ	Reads the next item from the DATA list into a variable.
RESTORE	Enables DATA statements in a program to be reREAD from a specified line.
STICK	A function returning the coordinates transmitted by a joystick.
STRIG	A function checking whether a joystick button is being or has been pressed.
STRIG ON/OFF	Enables and disables the button on a joystick.

Characters on Screen or Printer (See also Program Editing)

CLS	Clears the screen.
COLOR	Sets color for character writing, background color, and border color for a color screen.
CSRLIN	A function returning the number of the screen line (1...25) in which the cursor is situated.
LCOPY	Outputs a screen to the printer.
LOCATE	Determines the appearance of and positions the cursor.
LPOS	A function returning the current position of the print head.
LPRINT	Prints data on a printer.
LPRINT USING	As LPRINT, but also specifies a print format.

POS	A function returning the number of the screen column in which the cursor is situated.
PRINT	Displays data on the screen.
PRINT USING	As PRINT, but also specifies a display format.
SCREEN	Among other things, sets character mode for screen display. As a function, SCREEN returns the character at, or the color of, a specified position on the screen.
SPC	Prints a specified number of spaces in a PRINT command.
TAB	Moves the print or screen display (cursor) position to the column specified.
WIDTH	Sets width for output to screen or printer.
WRITE	Similar to PRINT.

Graphics

CIRCLE	Draws a circle with a specified center and radius, an arc, or even an ellipse.
COLOR	Selects one of the two color palettes and the background color.
DRAW	Draws a user-defined figure on the screen.
GET	Reads the colors of all points within a specified rectangle on the screen into an array variable.
LINE	Draws lines on the screen, and, if requested, fills in a delimited area.

PAINT	Fills in a delimited area on the screen with a specified color.
PMAP	Translates coordinates between program-defined coordinate system and actual screen coordinates.
POINT	A function returning the color of a specified point.
PRESET	Resets a specified point on the screen to background color.
PSET	Illuminates a point on the screen in a specified color.
PUT	Reads graphic information from an array variable and transmits it to the screen.
SCREEN	Sets graphic modes.
VIEW	Restricts or transfers screen activity to a specified area.
WINDOW	Redefines the coordinates of the screen.

The Loudspeaker

BEEP	A beep tone is emitted by the loudspeaker.
ON PLAY	Continuous background music during the execution of a program.
PLAY	For musicians. As a function it returns the number of notes still left in the music buffer.
SOUND	For the musically less gifted.

Program Variables and Type Conversion

ASC	A function returning the ASCII code for the first character in a string.

CDBL, CSNG	A function converting a number to double precision or single precision, respectively.
CINT	A function converting a number to an integer by rounding (not truncating).
CLEAR	Clears program variables.
CHAIN	Control is passed to another program, optionally all variables may be used by the new program.
COMMON	Marks specified variables as part of the CHAINed program.
CVI, CVS, CVD	Functions converting the string representation of a number into an integer, single precision number, or double precision number respectively.
DEFINT, DEFSG, DEFDBL, DEFSTR	Define one or more variables as type integer, single precision, double precision, or string.
ERASE	Removes array variables from a program.
FIX	A function truncating a number to an integer.
DIM	Declares the number of dimensions and their maximum subscripts for an array variable.
INT	As FIX, except that negative numbers are truncated to the next lowest integer value (e.g. -3.67 becomes -4).
HEX$	A function converting a numeric value to a hexadecimal string.

LET	Assigns a value to a variable.
MKI$, MKS$, MKD$	Converts an integer, single or double precision number to a string.
OCT$	A function converting a numeric value to an octal string.
OPTION BASE	States whether 0 or 1 is to be the minimum value for subscripts to array variables.
SWAP	Exchanges the values of two variables.
STR$	A function converting a value to its string counterpart.
VAL	A function extracting a numeric value from a string.
VARPTR	A function returning the memory address of a specified variable.

String Manipulation

ASC	A function returning the ASCII code for the first character in a string.
CHR$	A function returning the ASCII character corresponding to a value.
DEF FN	Define your own string processing functions.
INSTR$	Searches a string for a particular character sequence.
LEFT$	A function returning the leftmost part of a string.
LEN	A function returning the length of a string.

LSET	Left-justifies a string.
MID$	A command and function used for extracting or replacing part of a string.
RIGHT$	A function returning the rightmost part of a string.
RSET	Right justifies a string.
SPACE$	A function returning a number of spaces in a string variable.
STRING$	A function returning a string consisting of a character, or the first character of a string variable, repeated a specified number of times.
STR$	A function converting a numeric value to its string equivalent.
VAL	A function extracting a value from a string.

Mathematical Functions

ABS	Returns the absolute value of a number.
ATN	Arctangent in radians.
COS	Cosine of an angle in radians.
DEF FN	A command enabling you to define your own mathematical functions.
EXP	Raises e to a specified power.
FIX	Truncates a number to an integer.
INT	As FIX, except that negative numbers are truncated to the next lowest integer value (e.g. -3.67 becomes -4).

LOG	The natural logarithm of a number.
RND	Returns a random number (see also RANDOMIZE).
SGN	The sign of a number.
SIN	Sine of an angle in radians.
SQR	The square root of a number.
TAN	Tangent of an angle in radians.

Decision Making and Branching

CALL	Transfers program control to a machine language program.
FOR..TO..STEP	Performs the program lines a specified number of times up to the NEXT command.
GOSUB	Transfers program control to the subroutine starting at a specified line number. The subroutine should be concluded by RETURN. See also ON GOSUB in the detailed description.
GOTO	Transfers program control to the specified line. See also ON GOTO in the detailed description.
IF..THEN..ELSE	If the specified condition is fulfilled, GW-BASIC carries out a specified command or commands. Optionally, you may determine what the program should do if the condition is not fulfilled.
NEXT	Returns program control to the beginning of the loop (FOR..TO..STEP), as long as the specified number of passes through the loop has not been exceeded.

RETURN	Concludes a subroutine, and returns program control to the line which follows the GOSUB command, or to another, specified line.
USR	A function returning a value from a machine language routine (similar to CALL).
WHILE..WEND	Encloses a sequence of program lines to be executed repeatedly, as long as a specified condition is fulfilled.

Event Trapping

COM ON/OFF/STOP	Enables and disables communications activity.
ERL	A system variable returning the line number where the last error was detected by GW-BASIC.
ERR	A system variable returning the GW-BASIC code number of the last detected error. (The GW-BASIC error codes are listed in *Appendix A.)*
ERROR	Simulates an error. Useful for testing error trapping routines.
KEY ON/OFF/STOP	Enables and disables the trapping facility for Function Keys and cursor movement keys.
ON COM GOSUB	Specifies the program line number of a subroutine to which program control transfers in the event of communications activity occurring.
ON ERROR GOTO	In the event of GW-BASIC detecting an error, program control will be passed to the specified line.

ON KEY GOSUB	In the event of a specified Function Key or cursor movement key is pressed, program control passes to the subroutine at the specified program line.
ON PEN GOSUB	In the event of light pen activity, program control passes to the subroutine at the specified program line.
ON STRIG GOSUB	In the event of a joystick button being pressed, program control passes to the subroutine at the specified program line.
ON TIMER	Determines the line to which program control will be passed in the event of a specified time elapsing (see TIME$ and TIMER in *Other Commands and Functions).*
PEN ON/OFF/STOP	Enables and disables the light pen facility.
RESUME	After dealing with an error, returns program control to the line where the error occurred, to the subsequent line, or to another, specified line.
STRIG ON/OFF/STOP	Enables and disables joystick button, or the trapping facility itself.

Other Commands and Functions

DATE$	Allows you to change or read the date.
OUT	Transmits one byte to a port of the computer.
PEEK	Returns the contents of the byte at a specified address in computer memory.
POKE	Writes a byte into computer memory at a specified address.

REM	Used to record a programmer's remark.
TIME$	Allows you to change and read the time.
TIMER	A function returning the number of seconds since the last system reset or midnight, whichever is the most recent. (System reset is achieved by switching the computer on, or the key combination <Ctrl-Alt-Del>.)

SYSTEM COMPATIBILITY

A special characteristic of your NCR PC is that it is operationally compatible with the IBM PC/PCXT. This compatibility includes GW-BASIC, which can accommodate programs created under the Advanced BASIC of the IBM computer.

The GW-BASIC version supplied is Version 2.02. This is highly compatible with IBM Advanced BASIC 2.0. This means that with GW-BASIC you can run programs written under IBM Advanced BASIC 2.0 or an earlier version, both on your NCR PC and the IBM unit. Similarly, programs you write using GW-BASIC can be run under IBM Advanced BASIC 2.0.

If you are using GW-BASIC to write programs that should also be able to execute under an earlier version of IBM Advanced BASIC, you should refer to the programming language documentation of that earlier version. A number of GW-BASIC features are not supported by these earlier versions. Characteristic enhancements of GW-BASIC 2.02 are:

- Re-direction of Standard Input (INPUT, LINE INPUT) and Standard Output (PRINT) can be specified in the NCR-DOS command line to load GW-BASIC.

- The same NCR-DOS command line allows the /M option to specify a "maximum blocksize", in order to reserve space for machine language routines.

- Execution of NCR-DOS command files from within GW-BASIC (SHELL).

- Communication with user installed devices.

- Graphic enhancements:

  Line Clipping with CIRCLE, LINE, PAINT, POINT, PRESET, and PSET, so that out-of-range drawing does not wrap round to another part of the screen.

  New graphics features: PMAP, WINDOW, and VIEW.

  Additions to DRAW (Turn through Angle, Paint), LINE (style option allows dashed and dotted lines etc.), PAINT (background option for easier tile painting), and POINT (differentiates between physical and world coordinates).

- Other new functions: PLAY and TIMER.

- A new option (/D) in the command line enables double precision calculation of ATN, COS, EXP, LOG, SIN, SQR, and TAN. RANDOMIZE is also available in double precision.

- Parity checking for communications can be enabled and disabled (PE option in OPEN "COM).

- Sound features enhanced: PLAY (raising and lowering of octave).

- New event traps: ON PLAY, ON TIMER. ON KEY now allows trapping of up to six user-specified keys.

- File processing:

  GET and PUT allow record numbers up to 16,777,215, thus providing for large files with short records.

  LOF returns the actual number of bytes allocated to a file.

  EOF can be applied to re-directed Standard Input.

  New commands (ENVIRON, MKDIR, CHDIR, and RMDIR) allow manipulation of NCR-DOS paths and access to other directories.

- DELETE command: if no line number is specified after the hyphen, the program is deleted to the end.
- The characters <, >, and \ are not permitted as part of a filename or extension, as they may now be used in the re-direction of input and output (<,>) or for specifying a path (\).

SYNTAX NOTATION

The descriptions of the GW-BASIC commands and functions use the following notation to explain the "rules" which determine the way the command or function should be written:

[ ]	Square brackets indicate that the enclosed entry is optional.
< >	Angle brackets indicate user-entered data. When the angle brackets enclose a key, press the key named by the text; for example, <Break>.
{ }	Braces indicate that the user has a choice between two or more entries. At least one of the entries enclosed in braces must be chosen unless the entries are also enclosed in square brackets.
|	Vertical bars separate choices within braces; when used with a filter indicates a pipe. Otherwise, at least one of the entries separated by bars must be chosen unless the entries are also enclosed in square brackets.
...	Ellipsis indicate that an entry may be repeated as many times as needed or desired.
CAPS	Capital letters indicate portions of statements or commands that must be entered exactly as shown.

All other punctuation, such as commas, colons, slash marks, and equal signs must be entered exactly as shown.

Each description in this chapter is formatted as follows:

Syntax	Shows the correct syntax for the instruction or function. See the introduction to this manual for syntax notation. When the term "filespec" is used as an option in the syntax, it refers to a combination of device name and filename in the correct format for the operating system.
Purpose	Tells what the instruction or function is used for.

Remarks	Describes in detail how the instruction or function is used.
Example	Shows sample programs or program segments that demonstrate the use of the instruction or function.
Note	Describes special cases or provides additional pertinent information.

ABS Function

Syntax	ABS(X)
Purpose	To return the absolute value of the expression X.
Example	PRINT ABS(7*(-5)) will yield 35

ASC Function

Syntax

ASC(X$)

Purpose

To return a numerical value that is the ASCII code of the first character in the string X$. (See *Appendix B* for ASCII codes.)

Remarks

If X$ is empty, an "Illegal function call" error is returned.

Example

```
10 X$="TEST"
20 PRINT ASC(X$)
```

will yield

```
 84
```

this being the ASCII code of uppercase T.

See the CHR$ function for details on ASCII code-to-character conversion, and the example in the description of CONT.

ATN Function

Syntax

ATN(X)

Purpose

To return the arctangent of X, where X is in radians. Result is in the range -pi/2 to pi/2.

Remarks

The expression X may be any numeric type, but the evaluation of ATN is performed in single precision, unless you specify the /D option when loading GW-BASIC.

Example

```
10 INPUT X
20 PRINT ATN(X)
```

If you enter 3, the value displayed will be 1.249046

Note

To convert degrees to radians:

RADIANS = DEGREES *PI/180

where PI (single precision) is 3.141593.

AUTO Command

Syntax

AUTO [line number[,increment]]

Purpose

To automatically generate line numbers during program entry.

Remarks

AUTO begins numbering at "line number" and increments each subsequent line number by "increment". The default for both values is 10. If "line number" is followed by a comma but "increment" is not specified, the last increment specified in an AUTO command is assumed.

If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn the user that any input will replace the existing line. However, pressing <ENTER> immediately after the asterisk will save the existing line and generate the next line number.

If the cursor is moved to another line on the screen, numbering will resume there.

AUTO is terminated by pressing <Ctrl-Break>. The line in which <Ctrl-Break> is pressed is not saved. After <Ctrl-Break> is pressed, GW-BASIC returns to command level, ("Ok").

Example

AUTO 100,50

Generates line numbers 100,
150, 200 . . . .

AUTO

Generates line numbers 10,
20, 30, 40 . . . .

Note

Before editing program lines other than the line currently offered by AUTO, be sure to leave AUTO by pressing <Ctrl-Break.>

BEEP Statement

Syntax	BEEP
Purpose	To produce an 830 Hz tone in the speaker for approximately 1/4 second (to be precise: 240 ms).
Remarks	BEEP has the same effect as PRINT CHR$(7); (see *Appendix B*).
Example	2430 IF X < 20 THEN BEEP

BLOAD Command

Syntax

BLOAD "filespec" [,offset]

"filespec" refers to a file in the NCR-DOS file naming conventions (see Chapter 5, *Files and Devices).*

"offset" is a numeric in the range 0 to 65535. This is the offset address at which loading is to start in the segment declared by the last DEF SEG statement.

Purpose

To load a specified memory image file into memory from disk.

Remarks

The BLOAD command allows a program or data that has been saved as a memory image file to be loaded anywhere in memory. A memory image file is a byte-for-byte copy of what was originally in memory. See BSAVE for information about saving memory image files.

If the offset is omitted, the segment address and offset contained in the file (i.e., the address specified by the BSAVE statement when the file was created) are used.

If offset is specified, the segment address used is the one given in the most recently executed DEF SEG statement. If no DEF SEG statement has been given, the GW-BASIC data segment will be used as the default.

CAUTION: BLOAD does not perform an address range check. It is therefore possible to load a file anywhere in memory. The user must be careful not to load over GW-BASIC or the operating system.

Example

```
10 DEF SEG=&H6000
```

```
20 BLOAD"PROG1",&HF000
```

This example sets the segment address at hexadecimal 6000 and loads PROG1, starting F000 bytes (hexadecimal) above the segment address. (You are not bound to using hexadecimal values: the decimal equivalents are 24576 and 61440, respectively.)

Note

BLOAD is especially useful for loading screen images from disk into video memory (screen buffer). This requires additional information about the location and structure of video memory (see Chapter 7).

BSAVE Command

Syntax

BSAVE "filespec",offset,length

"filespec" refers to a file in accordance with the NCR-DOS file naming conventions (see Chapter 5, *Files and Devices).*

"offset" is a numeric in the range 0 to 65535. This is the offset address in the segment declared by the last DEF SEG statement. Memory is saved on disk starting here.

"length" is a numeric in the range 1 to 65535. This is the length in bytes of the memory image to be saved.

Purpose

To save the contents of the specified area of memory, for example, a machine language program, as a disk file.

Remarks

The "filespec", "offset", and "length" are required in the syntax.

The BSAVE command allows data or programs to be saved as memory image files on disk . A memory image file is a byte-for-byte copy of what is in memory.

If "offset" is omitted, a "Bad file name" error is issued and the save is aborted. A DEF SEG statement must be executed before the BSAVE. The last known DEF SEG address will be used for the save.

If length is omitted, a "Bad file name" error is issued and the save is aborted.

Example

```
10 DEF SEG = &H6000
20 BSAVE"PROG1",&HF000,256
```

This example saves 256 bytes starting at memory address 6000:F000, that is, hexadecimal F000

bytes 11 above the segment address hexadecimal 6000 (You are not bound to using hexadecimal values: the decimal equivalents are 24576 and 61440, respectively.) in the file PROG1.

Note

BSAVE is especially useful for saving screen images on disk. This requires additional information about the location and structure of video memory (see Chapter 7).

CALL Statement

Syntax

CALL variable name[(argument list)]

where "variable name" contains an address that is the starting point of the subroutine. This starting point is an offset address to the last defined segment. "variable name" may not be an array variable name.

"argument list" contains the parameter names that are passed to the external subroutine.

Purpose

To call an assembly language subroutine or a compiled routine written in another high level language.

Remarks

CALL is one way to transfer program flow to an external subroutine. (See also the USR function).

CALL generates the same calling sequence used by FORTRAN and BASIC compilers.

CDBL Function

Syntax CDBL(X)

Purpose To convert X to a double precision number.

Example
```
10 A=454.67
20 PRINT CDBL(A)
```
will yield
```
454.6700134277344
```

Note This function clearly cannot make the original number more accurate than it was before conversion. In the example, the new, double precision number is still accurate to the second place after the decimal point, and that only after rounding. The section "Type Conversion" in Chapter 1 describes the factors influencing accuracy when converting from one type of number to another.

CHAIN Statement

Syntax

CHAIN [MERGE]filespec[,[line number exp] [,ALL] [,DELETE range]]

See the examples below for illustration of the syntax options.

Purpose

To call a program and pass parameters to it from the current program.

Remarks

"filespec" is the name of the program (see Chapter 5, *Files and Devices)* that is called.

COMMON may be used to pass variables.

"line number exp" is a line number or an expression that evaluates to a line number in the called program. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line. "line number exp" is not affected by a RENUM command, used on the calling program.

With the ALL option, every variable in the current program is passed to the called program. If the ALL option is omitted, the current program must contain a COMMON statement to list the parameters that are passed.

If the ALL option is used and "line number expression" is not, a comma must hold the place of "line number exp". For example, CHAIN "NEXTPROG",,ALL is correct; CHAIN "NEXTPROG",ALL is incorrect. In the latter case, GW-BASIC assumes that ALL is a variable name and evaluates it as a line number expression.

The MERGE option allows a subroutine to be brought into the GW-BASIC program as an overlay. That is, the current program and the called program are merged (see MERGE). The

called program must be an ASCII file if it is to be merged.

After an overlay is used, it is usually desirable to delete it so that a new overlay may be brought in. To do this, use the DELETE option.

The line numbers in "range" are affected by the RENUM command.

Examples

```
COMMON VAR1,VAR2,VAR$
CHAIN "NEWPROG"
```

causes GW-BASIC to load this program, and pass program control to the beginning of that program. The three variables named under COMMON are passed to the chained program.

```
COMMON VAR1,VAR2,VAR$
CHAIN "NEWPROG",1000
```

has the same effect, except that program control is passed to line 1000 of the chained program.

```
CHAIN "NEWPROG",1000,ALL
```

differs from the previous example, in that all the variables (not just three) of the current program are passed to the chained program.

```
CHAIN MERGE "OVERLY1",1000,ALL
```

has the special effect of overwriting lines in the current program with lines from OVERLY1, where line numbers between the two programs coincide (all of OVERLY1 is chained).

It is possible to clear an area of program lines, in order to provide clean loading space for the chained program. For example,

```
CHAIN MERGE
"OVERLY2",1000,ALL,DELETE 1000-2000
```

deletes lines 1000 to 2000 of the current program before loading OVERLY2.

Note

CHAIN command with MERGE option leaves the files open and preserves the current OPTION BASE setting. The chained program may, however, have an OPTION BASE of its own, if no array variables are being passed.

If MERGE is omitted, CHAIN does not preserve variable types or user-defined functions for use by the chained program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEFFN statements containing shared variables must be restated in the chained program.

When using the MERGE option, user-defined functions should be placed before any CHAIN MERGE commands in the program. Otherwise, the user-defined functions will be undefined after the merge is complete.

CHAIN does a RESTORE before the chained program is run. Therefore, the pointer to DATA items is reset. READ does not continue after the last DATA item read in the old program.

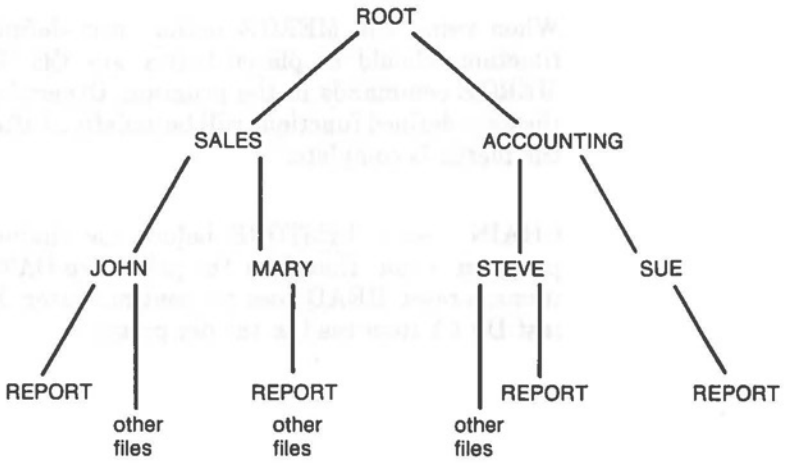
CHDIR Command

Syntax CHDIR "path"

Purpose To change the current directory.

Remarks "path" is a string expression not exceeding 128 characters identifying the new directory which is now to be the current directory. For details about paths and directories you should refer to your *NCR-DOS* manual.

Examples Given the following hierarchical structure



to change to the directory SUE from the ROOT directory, use

CHDIR "ACCOUNTING\SUE"

To change from SALES to the directory JOHN:

CHDIR "JOHN"

To change from JOHN back to SALES:

CHDIR ..

You can specify a disk drive, thus applying the CHDIR command to a disk drive other than the current one, for example

```
CHDIR "C:SALES"
```

Note

When your program refers to a file, GW-BASIC looks for the file in the current directory of the disk in the default drive, unless you have specified the drive (and pathname) as part of the filespec. (The syntax descriptions in this chapter denote file references by means of the term "filespec").

CHR$ Function

Syntax

CHR$(I)

Purpose

To return one character with a decimal equivalent ASCII code of I. (Decimal equivalent ASCII codes are listed in *Appendix B.)*

Remarks

CHR$ is commonly used to send a special character to the screen or printer. For instance, the beeping character CHR$(7) could be sent as a preface to an error message, or a form feed CHR$(12) could be sent to clear the screen and return the cursor to the home position.

Example

```
PRINT CHR$(66)
returns
B
```

See the ASC function for details of how to convert a character back to its ASCII code. See also the example in the description of CONT.

You can set a repetition factor for the character by means of the STRING$ function. The following example beeps the speaker for approximately 30 seconds:

```
10 B$=STRING$(120,CHR$(7))
20 PRINT B$
```

The following example programs the Function Key 1 to issue the GW-BASIC LIST command, without your having to press <ENTER>

```
KEY 1,"LIST"+CHR$(13)
```

Note

The CHR$ function is useful for displaying characters for which there is no single key action on the keyboard. For example, your keyboard probably does not include the square root symbol, but you can display it with

```
PRINT CHR$(251);
```

CINT Function

Syntax

CINT(X)

Purpose

To convert X to an integer by rounding the fractional portion.

Remarks

If the result is not in the range -32768 to 32767, an "Overflow" error occurs.

Example

```
PRINT CINT(45.67)
will yield
 46

PRINT CINT(-3.85)
will yield
-4
```

See the CDBL and CSNG functions for details on converting numbers to the double precision and single precision data type, respectively. See also the FIX and INT functions, both of which return integers.

CIRCLE Statement (Graphics Modes)

Syntax

CIRCLE (x,y), radius [,color[,start,end[,aspect]]]

Purpose

Draws an ellipse on the screen according to the following definitions:

x,y
Specifies the coordinates of the center of the ellipse. The coordinates can be absolute, or relative to the last point addressed on the screen (using STEP).

radius
Specifies the radius (major axis) in points.

color
In low and high resolution color graphics, this selects the color (1 to 3) from the current palette. The background color (0) is also allowed. If you do not specify the color, color 3 is used. In medium and high resolution black-and-white graphics, the color can be 1 for white, or 0 for black (default is 1).

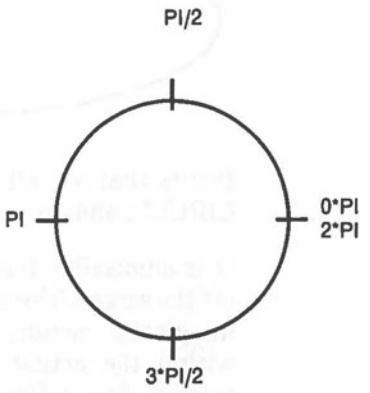
start,end
Specifies in radians where the drawing is to begin and end. The values may range from -2*PI to 2*PI, where PI = 3.141593. (See also remarks.)

aspect
Specifies the ratio of the X radius to the Y radius. If the ratio is less than 1, the radius is the X (horizontal) radius; if the ratio is greater than 1, the radius is the Y (vertical) radius.

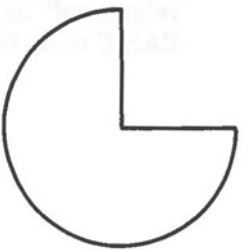
GW-BASIC produces a circle on the screen without your having to specify a value for "aspect." If you specify a value for "aspect" which is not 5/6 in low and high resolution graphics or 5/12 in medium resolution graphics, an ellipse is displayed. If the value you specify is less than 5/6 (medium resolution: 5/12), the ellipse has the form of a circle which has been stretched along the horizontal axis (see example).

Remarks

The first two arguments (x,y coordinates and radius) are the only ones required to draw a circle. Use the last two arguments to draw other "curved" shapes. Start and end, for example, allow you to control how much of the circle is to be drawn. The values of start and end are in radians, positioned in the standard mathematical way.



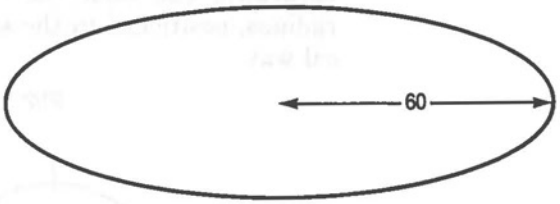
Either start or end value may be negative (-0, however, is not allowed) in which case the angle is connected to the center point with a line. For example, start and end values of -PI/2, -2PI would draw part of a circle.



Use the aspect argument to draw an ellipse other than a circle. Remember, if the aspect ratio is less than 1, then r is the X radius; if the aspect ratio is greater than 1, then r is the Y radius. For example,

```
10 SCREEN 1
20 CIRCLE (160,100),60,,,,5/18
```

will draw an ellipse like this:



Note

Points that are off the screen are not drawn by CIRCLE, and do not cause an error situation.

It is admissible for the center coordinates to be off the screen. The ellipse is then drawn using the imaginary center, whereby points which lie within the actual screen coordinates are displayed. The following example draws an arc across the top right corner of the screen in high resolution:

```
10 SCREEN 2
20 CIRCLE (650,-10),100
```

After an ellipse has been drawn, the "last point referenced" on the screen is considered by GW-BASIC to be the center of the ellipse.

CLEAR Command

Syntax

CLEAR [,[expression1] [,expression2]]

Purpose

To set all numeric variables to zero, all string variables to null, and to close all open files; and, optionally, to set the end of memory and the amount of stack space.

Remarks

"expression1" is a memory location that, if specified, sets the highest location of the workspace available for use by GW-BASIC. You can thus put aside space for machine language programs.

"expression2" sets aside stack space for GW-BASIC. The default is 572 bytes or one-eighth of the available memory, whichever is smaller. The specification of a larger stack may be necessary if your program uses deeply-nested GOSUB routines, or a lot of FOR...NEXT loops, or does extensive PAINTing.

CLEAR performs the following actions:

Closes all files.
Clears all COMMON variables.
Resets numeric variables and arrays to zero.
Resets the stack and string space.
Resets all string variables and arrays to null.
Releases all disk buffers.
Cancels all DEFinitions (DEF FN, DEF USR, DEF SEG, DEFINT, DEFDBL, DEFSNG, DEFSTR)
Turns off sound and resets to Music Foreground.
PEN and STRIG are reset to OFF.

It does not erase the program in memory.

Examples

CLEAR

performs the above-stated actions only.

CLEAR ,32768

has the additional effect of setting the maximum workspace to 32KB.

CLEAR ,,2000

has the special effect of setting aside 2000 bytes for the stack.

CLEAR ,32768,2000

performs the CLEAR actions, sets maximum workspace to 32KB, and sets aside 2000 bytes for the stack.

Note

To free space in memory, your program can use the ERASE statement on specified array variables.

CLOSE Command

Syntax

CLOSE [[#]file number[,[#]file number...]...]

Purpose

To conclude I/O to a file or device. The CLOSE statement is complementary to the OPEN statement.

Remarks

"file number" is the number under which the file was opened. A CLOSE with no arguments closes all open files and devices.

The association between a particular file and file number terminates upon execution of a CLOSE statement. The file may then be reopened using the same or a different file number; also that file number may now be reused to open any file.

A CLOSE for a sequential output file (or device) writes the final buffer of output.

The END statement and the NEW command always close all disk files automatically. (STOP does not close disk files.)

Access to files and devices is discussed in Chapter 5.

Examples

CLOSE

closes all open files and devices.

CLOSE #1,#2,#3

closes the files and devices associated with the numbers 1, 2, and 3. (The inclusion of the # sign is optional).

Note

END, NEW, RESET, SYSTEM and RUN (without the R option) all have the effect of automatically closing open files and devices. Pressing <Ctrl-Break> during program execution has the same effect. STOP does not close any files or devices.

CLS Statement

Syntax

CLS <n>
where <n> may be 1 or 2

Purpose

Erases the screen to the currently selected background color.

Remarks

If the KEY ON statement is in effect when you use the CLS statement, the screen is cleared; however, the function line at the bottom of the screen is renewed with the currently active background/foreground colors.

In character mode, the cursor is placed in the top left corner of the screen. In character mode, tl screen buffer (video memory) can store 8 scre pages. CLS clears only the active screen page (u SCREEN to determine which page is active

In the graphics modes, only one screen page is present in the screen buffer. CLS clears the screen buffer completely. The last referenced point on the screen is then considered to be 160,100 in low, 320,100 in medium, and 320,20 in high resolution. A subsequent graphic comman using STEP refers to this point.

The <n> parameter may be specified to selectively clear portions of the screen. <n> may be 1 or 2. CLS 1 results in a graphics viewport being cleared, while CLS 2 clears a text window.

Example

```
10 COLOR 12,1
20 CLS
```

in character mode, clears the screen and sets the background color to blue (1). Subsequent writing (including the GW-BASIC "Ok" message) appears in light red (12).

Note

In character mode, the color statement alone does not set the screen to the new background color: the new background color is used as subsequent screen writing progresses. CLS has the effect of

setting the entire character area of the screen to the background color. Similarily, CLS is not required in the graphics modes in order to change the background color of the screen.

The SCREEN and WIDTH commands, used to set the screen mode and character line WIDTH, also have the effect of clearing the screen.

COLOR Statement (Character Mode)

Syntax

COLOR [writing] [,[background]

"writing" is a numeric expression in the range 0...31. This represents the color in which characters are to be displayed.

"background" is a numeric expression in the range 0...7 for the background color.

Purpose

To alter one or more of the two color factors (writing, background,) which make up the display on a color screen. On a monochrome display, COLOR can be used to invert the video display (i.e., black characters on white, instead of white characters on a black background; or vice versa). On both monochrome and color screens, writing can be set to increased brightness or be made to blink.

Remarks

The significance of the values 0 to 7:

0 Black	4 Red
1 Blue	5 Magenta
2 Green	6 Brown
3 Cyan	7 White

The values 8 to 15 refer to high-intensity versions of the same colors:

8 Gray	12 Light Red
9 Light Blue	13 Light Magenta
10 Light Green	14 Yellow
11 Light Cyan	15 High Intensity

For the writing color, you may add 16 to the required color, thus yielding a value 16 to 31. This causes subsequent screen writing to blink in the selected color.

With a monochrome display adapter, color values are used as follows (references to the color white are to be understood as the standard writing color used by your display).

	Writing	Background
0	Black	Black
1	White, and underlined	Black
2...6	White	Black
7	White	White

When using a white background (7), you may use for writing 0, 8, 16 or 24 (the latter two produce a blinking display). You cannot set white writing on a white backgound. Furthermore, there is now high intensity and no underlining.

When using a black background (0...6), you can set normal white, high-intensity white, normal white blinking, and high-intensity white blinking (7, 15, 23, or 31, respectively). In each case, subtracting 6 from the value gives the additional effect of underlining.

It usually makes sense to select colors so as to produce an acceptable contrast between background and writing. However, GW-BASIC does not prevent you from using black for both. To create this invisible writing effect, specify 0 for background and 0, 8, 16, or 24 for writing. You could use this possibility for entering, say, passwords which are not intended to be displayed on the screen.

Combinations other than those stated here produce white writing on black background effect.

With a color graphics display adapter, you can specify any color in the range 0 to 7 as the common color for <writing> and <background> in order to make writing invisible.

Examples
(With color graphics display adapter)

COLOR 12

sets the writing color to light red; the background color remains as it was before.

COLOR ,1

influences only the background color, changing it to blue.

COLOR 12,1

sets writing to light red and background to blue.

The following example is for both monochrome and color displays. It asks you to enter a known text in such a way that you cannot see what you are typing on the screen, but you may backspace and correct if you suspect you have made a mistake. Your typing speed is timed, and displayed (in seconds), assuming that the text you have typed and entered with the <CR> key is identical to the string contained in X$. After your "score" is displayed, press any key to return to "Ok".

```
10 X$="The quick brown fox jumps over the
   lazy dog"
20 SCREEN 0:WIDTH 80
30 COLOR 7,0
40 CLS
50 PRINT "Start typing...";
60 FOR DLY% =1 TO 700:NEXT DLY%
70 COLOR 0,7
80 PRINT " NOW"
90 TIME$="00:00:00"
100 COLOR 0,0
110 INPUT I$
120 T$ =RIGHT$(TIME$,2)
130 SLOW$=RIGHT$(TIME$,4)
```

```
140 CLS
150 IF I$<>X$ THEN GOTO 220
160 COLOR 23,0
170 PRINT X$:PRINT:PRINT
180 PRINT " WELL DONE ";
190 IF ASC(SLOW$)<>48 THEN PRINT "But
    you took at least a minute":GOTO 260
200 PRINT "You took ";T$;" seconds":PRINT
210 GOTO 260
220 COLOR 7,0
230 PRINT "Not quite right...":PRINT
240 PRINT "You should have typed:":PRINT
    X$:PRINT
250 PRINT "... this is what you wrote:":PRINT
    I$
260 COLOR 7,0
270 IF INKEY$="" THEN GOTO 270
```

Note

The final item in a COLOR command should not be a comma.

A value outside the permitted range may lead to an "Illegal function call" error.

The display mode (character, low, medium or high resolution graphics) can be set through the SCREEN command.

COLOR Statement (Graphics Mode)

Syntax

COLOR [background] [,[palette]

"background" is a numeric expression specifying the screen background color. Values 0 to 15 are allowed (see COLOR (Character Mode) for the color significance of these values).

"palette" is a numeric expression which selects one of the two available color palettes. An even number selects palette 0, an odd number selects palette 1. On each palette there are three colors:

Color Number	Palette 0	Palette 1
1	Green	Cyan
2	Red	Magenta
3	Brown	White

Purpose

When COLOR is executed, the background color changes immediately. When COLOR is used to change the palette, the colors of drawings currently displayed on the screen change accordingly: if you change from palette 0 to 1, what was green is now cyan, red becomes magenta, and brown becomes white. The reverse is true if you change from palette 1 to 0 (cyan becomes green, etc.).

The CIRCLE, DRAW, LINE, PAINT, PRESET AND PSET commands can use either the background color or one of the three colors from the current palette. COLOR is used to select a palette for these graphics commands.

Remarks

Characters written on the screen in low or high resolution color graphics use color number 3 from the currently selected palette, that is, brown or white.

Example

```
10 SCREEN 4
20 COLOR 2,0
```

sets screen mode 4 (high resolution color graphics), the background color to green, and selects color palette 0.

```
10 SCREEN 4
20 COLOR 4
```

sets screen mode 4 and the background color to red. The color palette remains as it was before.

```
10 SCREEN 4
20 COLOR ,1
```

sets screen mode 4 and selects color palette 1. The background color does not change.

The example at the end of the Chapter *Screen Display* demonstrates just a few of the possibilities of GW-BASIC color graphics.

Note

COLOR is not applicable to medium and high resolution black-and-white graphics, as this screen mode uses black and white only. If you attempt to use COLOR in this mode, an 'Illegal function call' error will result.

An "Illegal Function call" error also results from a value exceeding 255.

COM Command

Syntax

COM(n) ON
COM(n) OFF
COM(n) STOP

where n is the number of the communications channel, either 1 or 2.

Purpose

To enable or disable event trapping of communications activity on the specified channel.

The COM(n) ON enables communications event trapping by an ON COM command (see ON COM). While trapping is enabled, and if a non-zero line number is specified in the ON COM statement, GW-BASIC checks before the execution of every command to see if activity has occurred on the communications channel. If it has, GW-BASIC transfers program control to the line indicated by the ON COM command.

COM(n) OFF disables communications event trapping. If an event takes place, it is not recorded.

COM(n) STOP disables communications event trapping, but if an event occurs, it is recorded and ON COM will affect transfer of program control as soon as trapping is enabled.

Example

```
10 COM(1) ON
```

Enables error trapping of communications activity on channel 1.

COMMON Statement

Syntax

COMMON list of variables

Purpose

To pass variables to a chained program.

Remarks

COMMON is used with CHAIN. COMMON statements may appear anywhere in a program, though it is recommended that they appear at the beginning. The same variable cannot appear in more than one COMMON. Array variables are specified by appending "()" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit COMMON.

CHAIN should not state the dimensions of an array variable. Such a statement is ignored by GW-BASIC.

Example

```
100 COMMON A,B,C,D(),G$
110 CHAIN "PROG3",10
```

chains the program PROG3, passes the variables A, B, C, G$, and the array variable D, and starts execution of the chained program at line 10.

CONT Command

Syntax

CONT

Purpose

To continue program execution after <Ctrl-Break> has been pressed, a STOP has been executed, or an untrapped error has occurred.

Remarks

Execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt ("?" or prompt string).

CONT is usually used in conjunction with STOP for program testing. When execution is stopped, you can examine intermediate values of variables and change them using direct mode commands. Execution may be resumed with CONT or a direct mode GOTO, which resumes execution at a specified line number. CONT may be used to continue execution after an error has caused GW-BASIC to terminate program execution.

CONT is invalid if the program has been edited during the break.

Example

The following program displays the characters of the GW-BASIC character set which have a code value of 128 or higher (see *Appendix B*). Each character, along with its code value, is displayed in a line of its own. A scrolling delay is built in (line 50), but to examine a character for any length of time, you will need to press <Ctrl-Break>. When you enter CONT as a direct command, the display of characters continues. You may interrupt and continue the program as often as you wish. WIDTH 40 means that characters are displayed in large format.

```
10 WIDTH 40
20 FOR LOOP% = 128 TO 255
```

```
30 PRINT "The character for code
   ";LOOP%;"is ";CHR$(LOOP%)
40 PRINT
50 FOR DLY% = 1 TO 500: NEXT DLY%
60 NEXT LOOP%
```

Note

RUN, even with a line number, is not suitable for continuing a program after a break, as it has the same effects on memory contents as CLEAR; notably, closing all files, erasing definitions, and clearing out variables.

COS Function

Syntax COS(X)

Purpose To return the cosine of X, where X is in radians.

Remarks The calculation of COS(X) is performed in single precision, unless you specify the /D option when loading GW-BASIC.

Example
```
10 X=2*COS(.4)
20 PRINT X
```
will yield
```
 1.842122
```

Note To convert radians to degrees:

DEGREES = RADIANS *180/PI

To convert degrees to radians:

RADIANS = DEGREES *PI/180

where PI (single precision) is 3.141593

CSNG Function

Syntax CSNG(X)

Purpose To convert X to a single precision number.

Example
```
10 A# = 482.3421222#
20 PRINT CSNG(A#)
```
will yield
```
482.3422
```

See the CINT and CDBL functions for converting numbers to the integer and double precision data types, respectively.

The section "Type Conversion" in Chapter 1 gives more information about conversion accuracy.

CSRLIN Function

Syntax

CSRLIN

Purpose

This is really equated to a variable containing the current line position of the cursor (0 to 24).

Example

In the following example, line 10 reads the current line position into L%. Line 20 reads the current column position into C%; line 30 displays HELLO in the middle line of the screen, and line 40 restores the position of the cursor to the former line and column.

```
10 L% = CSRLIN
20 C% = POS(0)
30 LOCATE 13,30 :PRINT "HELLO"
40 LOCATE L%,C%
```

Note

POS is similar to CSRLIN, returning the current column of the cursor instead. LOCATE is the command used for positioning the cursor.

CVI, CVS, CVD Functions

Syntax

CVI(2-byte string)
CVS(4-byte string)
CVD(8-byte string)

Purpose

To convert string values to numeric values.

Remarks

Numeric values that are read in from a random disk file must be converted from strings back into numbers. CVI converts a 2-byte string to an integer. CVS converts a 4-byte string to a single precision number. CVD converts an 8-byte string to a double precision number.

The result in each case is stored in the numeric variable; the string itself is unaffected by the conversion.

Example

```
70 FIELD #1,4 AS N$, 12 AS B$
80 GET #1
90 Y=CVS(N$)
```

The record read from the random file in line 80 is divided into two string variables, N$ and B$, by the FIELD declaration of line 70. Line 90 regards N$ as the string form of a single precision number, and assigns the equivalent numeric value to Y. Presumably, N$ was originally a number written to the file using the MKS$ function.

Note

The MKI$, MKS$, and MKD$ functions perform the inverse operations, that is, they convert numeric values to strings.

DATA Statement

Syntax

DATA constant[,constant]...

Purpose

To store the numeric and string constants that are accessed by the program's READ commands.

Remarks

DATA does not actually instruct GW-BASIC to carry out any action, so it may be placed anywhere in the program. A DATA list may contain as many constants as will fit on a line (separated by commas). Any number of DATA lines may be used in a program. READ commands access DATA lines in order (by line number). The data contained in the various DATA lines may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

A DATA line may contain numeric constants in any format; i.e., fixed-point, floating-point, integer, decimal, octal or hexadecimal. (No numeric expressions are allowed in the list.) String constants in a DATA line must be surrounded by double quotation marks only if they contain commas, colons, or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA list; otherwise, a "Syntax error" (not "Type mismatch") is issued by GW-BASIC.

A line of DATA May be reread from the beginning by use of the RESTORE statement.

Example

See READ

DATE$ Statement

Syntax

DATE$="string expression"

"string expression" returns a string in one of the following forms:

mm-dd-yy
mm-dd-yyyy
mm/dd/yy
mm/dd/yyyy

Purpose

To set the current date. This statement complements the DATE$ function, which retrieves the current date.

Remarks

The year must be in the range 1980 to 2099. If the "string expression" contains only one digit for the day or month, GW-BASIC assumes a zero in front of it. If only two digits for the year are given, GW-BASIC assumes the year falls in the twentieth century and places "19" in front of them.

Example

```
10 DATE$="07-13-1984"
```

The current date is set at July 13, 1984.

Note

The date may have been set already by NCR-DOS, before you loaded GW-BASIC. However, this does not prevent you from overwriting while in GW-BASIC.

DATE$ Function

Syntax

DATE$

Purpose

This is really equal to a variable containing the current date. (To set the date, use the DATE$ statement.)

Remarks

The DATE$ function returns a ten-character string in the form mm-dd-yyyy, where mm is the month (01 through 12), dd is the day (01 through 31), and yyyy is the year (1980 through 2099).

Example

```
10 PRINT DATE$
```

following the setting of DATE$, the display produces:

```
07-13-1984
```

The separators are hyphens, even if the separators used when entering the date were slashes.

DEF FN Statement

Syntax

DEF FNname[(parameter list)]=
function definition

Purpose

To define and name a function in addition to the functions provided by GW-BASIC.

Remarks

"name" must be a legal variable name. This name, preceded by FN, becomes the name of the function.

"parameter list" consists of those variable names in the function definition that are to be given values when the function is called. The items in the list are separated by commas.

"function definition" is a single expression that performs the operation of the function. Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name. A variable name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

The variables in the parameter list represent, on a one-to-one basis, the argument variables or values that will be given in the function call.

DEF FN may define either numeric or string functions. If the function is numeric, the result of evaluating the expression comprising the function definition is returned to the calling command with the precision inherent in the function name. If the calling command attempts to assign a numeric function result to a string variable, or vice versa, a "Type mismatch" error occurs.

GW-BASIC must encounter the DEF FN statement before the program makes use of the corresponding function, otherwise an "Undefined user function" error occurs. A function may be defined more than once. GW-BASIC always refers to the most recently encountered definition.

Example

To calculate the hypotenuse of a right-angled triangle (the side opposite the right-angle) you could define a function as follows:

```
10 DEF FNHYPOT(S1,S2)=SQR(S1∧2+S2∧2)
```

To make use of this function, you could continue the program with

```
20 INPUT "Length of one side adjacent to right
   angle?";SIDE1
30 INPUT "Length of other side?";SIDE2
40 PRINT "Length of hypotenuse is
   ";FNHYPOT(SIDE1,SIDE2)
```

Note

GW-BASIC does not accept DEF FN in the direct mode.

A function may be recursive (that is, it may call itself), but you must then provide a way of stopping it; otherwise, an error situation ("Out of memory") will occur.

DEFINT/SNG/DBL/STR Statements

Syntax

DEF<type> <range(s) of letters>

where <type> is INT, SNG, DBL, or STR

Purpose

To declare variable types as integer, single precision, double precision, or string.

Remarks

Any variable names beginning with the letter(s) specified in <range of letters> will be considered the type of variable specified in the <type> portion. However, a type declaration character at the end of the actual name of the variable (%, !, #, or $) always takes precedence over DEFtype. (See the section "Variables" in Chapter 1.)

If no type declaration commands are encountered, GW-BASIC assumes that all variables without declaration characters are single precision variables.

Examples

```
10 DEFDBL L-P
```

All variables beginning with the letters L, M, N, O and P will be double precision variables.

```
10 DEFSTR A
```

All variables beginning with the letter A will be string variables.

```
10 DEFINT I-N,W-Z
```

All variables beginning with the letters I, J, K, L, M, N, W, X, Y, Z will be integer variables.

Note

GW-BASIC does not recognize the type declaration stated in DEF until the DEF statement is actually encountered during program execution. It is therefore a good idea to place these type DEFinitions at the beginning of the program.

DEF SEG Statement

Syntax

DEF SEG [=address]

where "address" is a numeric expression in the range 0 to 65535.

Purpose

The address specified is saved for use as the segment required by BLOAD, BSAVE, CALL, POKE, USR, and PEEK.

Remarks

Entry of any value outside the "address" range 0 through 65535 will result in an "Illegal function call" error, and the previous value will be retained.

If the "address" option is omitted, the segment to be used is set to the GW-BASIC data segment (DS). This is the initial default value.

If the "address" option is given, it should be based on a 16-byte boundary. GW-BASIC multiplies this value by 16 and uses the result of this multiplication as the actual memory address of the beginning of the segment. GW-BASIC does not check the validity of the specified address.

Example

```
10 DEF SEG=&HB800
```

This program line sets the segment to the hexadecimal number B800 (in decimal: 47104), which represents a true memory address of 16 times that value -B8000 (753664). In fact, this is the beginning of the screen buffer for the color display, so you would probably follow up this statement with BLOAD or BSAVE. Later in the program you would probably change the segment back to GW-BASIC's Data Segment.

Note

DEF and SEG must be separated by a space. Otherwise, GW-BASIC will interpret DEFSEG=100 to mean "assign the value 100 to the variable DEFSEG."

DEF USR Statement

Syntax

DEF USR[digit]=integer expression

Purpose

To specify the starting address of an assembly language subroutine.

Remarks

"digit" may be any digit from 0 to 9. The digit corresponds to the number of the USR subroutine whose address is being specified. If "digit" is omitted, DEF USR0 is assumed. The value of "integer expression" is the starting address of the USR subroutine offset to the segment value which applies when the USR subroutine is called.

It is admissible to use the same "digit" in more than one DEF USR statement, then assign a new address to that "digit". GW-BASIC always recognizes the address most recently assigned. This enables you to access more than 10 subroutines.

Example

```
200 DEF USR0=24000
210 X=USR0(Y^2/2.89)
```

Line 200 defines the starting address of a machine language subroutine as 24000. Line 210 assigns to the variable X the result of whatever the subroutine does with the value of the expression given in parentheses (see USR).

If you need to access a subroutine by absolute memory address, consider the following example:

```
200 DEF SEG=0
210 DEF USR0=ABSADDR%
 .
 .
 .
300 RESULT=USR0(INFO)
```

where ABSADDR% contains the absolute memory address of the subroutine to be accessed in line 300.

DELETE Command

Syntax

DELETE [line number1] [-line number2]
DELETE [line number1-]

Purpose

To delete program lines.

Remarks

GW-BASIC always returns to command level ("Ok") after a DELETE is executed. If a specified line number does not exist, an "Illegal function call" error occurs.

Examples

DELETE 40
deletes line 40.

DELETE 40-100
deletes lines 40 through 100, inclusive.

DELETE -40
deletes all lines up to and including line 40.

DELETE 40-
deletes line 40 and any subsequent lines in the program.

DELETE
deletes the current line.

DIM Statement

Syntax

DIM variable (subscripts)
[,variable(subscripts)]...

Purpose

To specify the maximum values for array variable subscripts and allocate storage accordingly.

Remarks

If an array variable name is used without a DIM statement, the maximum value of the array's subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is always 0, unless otherwise specified with OPTION BASE.

The DIM statement sets all the elements of the specified numeric arrays to an initial value of zero. Elements of a string array are initially empty (zero length).

The maximum number of dimensions allowed in a DIM statement is 255, hardly a practical limitation. The number of dimensions is more likely to be limited by the amount of available memory, and the maximum admissible length for a program line. The maximum number of elements per dimension is 32767.

If you attempt to issue a DIM statement more than once for the same array variable, or if GW-BASIC encounters DIM after the implicit definition of that array variable (i.e., use of the array variable with maximum subscript 10 without a prior DIM), a "Duplicate Definition" error occurs.

Example

See the Exercises after the section "Array Variables" in Chapter 1.

DRAW Statement (Graphics Modes)

Syntax

DRAW "string expression"

Purpose

Draws an object as specified by the string expression.

Remarks

With the Draw statement you can draw an object using object definition language commands. A language command is a single character within a string, optionally followed by one or more arguments. The string expression defines an object which is drawn on the screen when GW-BASIC executes the DRAW statement.

The following movement commands begin movement from the coordinates of the last point referenced by another language command, or another GW-BASIC graphics statement (e.g., LINE or PSET).

U [<n>] Move up
D [<n>] Move down
L [<n>] Move left
R [<n>] Move right
E [<n>] Move diagonally up and right
F [<n>] Move diagonally down and right
G [<n>] Move diagonally down and left
H [<n>] Move diagonally up and left

The n in the preceding commands indicates the distance to move. The number of points moved is n times the scale factor (see S below). If you do not specify n, movement is one scale unit.

M<x,y>
Move absolute or offset (see Chapter 3 for discussion of x and y coordinates). If x is preceded by a + or -, x and y are added to the coordinates of the last point referenced. The point thus referenced is connected to the last point referenced by a line. If no + or - is added, a line is drawn to point (x,y) from the last point referenced.

The following prefix commands may precede any of the above movement commands:

B
Move but do not plot any points.

N
Move but return to original position when plotting is finished.

When considering how a drawing will look when displayed on the screen, you should take into account the "aspect ratio" of the screen. The aspect ratio is the vertical exaggeration factor of 1.2 in low and high resolution graphics, and 2.4 in medium resolution graphics. For example, the following short program DRAWs a square in medium resolution graphics:

```
10 SCREEN 2
20 DRAW "L96D40 R96 U40"
```

The vertical sides are specified with lower values than the horizontal sides because horizontal drawing illuminates more pixels per inch (2.4 times as many).

A square in low resolution graphics would be

```
10 SCREEN 1
20 DRAW "L96 D80 R96 U80"
```

In addition to simple straight line drawing, DRAW offers the following graphic commands.

TA<n>
Turns the drawing angle through <n> degrees in the counter clockwise direction. (A negative value for <n> turns the drawing angle clockwise). The result of

```
DRAW "TA5;U50"
```

is that the "upward" line is in fact leaning five degrees to the left, as we perceive it on

the screen. If <n> is outside the range -360 to +360, an "Illegal function call" error occurs.

A<n>

Similar to the TA command. The difference is that <n> represents a number 0, 1, 2 or 3, for 0.90, 180, and 270 degrees, respectively. Both TA and A compensate for drawing exaggeration which would otherwise occur when specifying 0 or 180 degrees. See the following example.

C<n>

Determines the drawing color. In low an high resolution color graphics, <n> may be 1 2, or 3, this being the color from the current ly selected palette, or 0, the current back ground color (see COLOR). If you do no specify a color, GW-BASIC uses 3. In mediun and high resolution black-and-white graphics <n> can be 0 (black) or 1 (white). If you d not specify a color, GW-BASIC uses 1.

S<n>

Sets the scale (magnification) factor for the U, D, L, R, E, F, G, H, and M (offset) movement commands. <n> divided by 4 is the scale factor. If you do not specify a scale factor, GW-BASIC uses 4 for <n>, that is, scale factor 1. <n> must be an integer value in the range 1 to 255.

<X string variable>

When DRAW encounters this command, it carries out the drawing commands contained in "variables" before proceeding with the rest of the command string. This enables you to execute a second string from within a string.

<P paint outline>

Paints the area comprising the drawing in which the last addressed point is enclosed. The drawing outline must already have the

color 'outline'; otherwise, painting goes beyond the outline. In low and high resolution color graphics, the colors may be 0, 1, 2, or 3; in medium and high resolution black-and-white graphics 0 or 1 (see 'C <n>' above). Both 'paint' and 'outline' must be specified.

The items contained in a DRAW command string do not require separators (not even blanks), with the exception that a semicolon must separate the name of a variable from the next item. However, spaces and/or semicolons may be used between items, and they do make the string easier to read.

Variables are allowed for all n and x,y values indicated in this description of DRAW. A variable is preceded by an equal sign, except after the X drawing command.

Examples

```
10 SCREEN 1
20 DRAW "E15 F15 L30"
```

draws an isosceles triangle.

```
10 SCREEN 1
20 V=50
30 DRAW "U=V; R=V; D=V; L=V;"
```

draws a box. (This time we have used variables (V), so the semicolons are necessary.) The following lines move the "last referenced point" into the confines of the box, and paint the box with color 2 and the outline with color 1 from the currently selected palette:

```
30 DRAW "C2"
40 DRAW "BE10"
50 DRAW "P1,2"
```

The following example draws spokes:

```
10 SCREEN 1
20 FOR L%=0 TO 360 STEP 10
30 DRAW "TA=L%;NU60"
40 NEXT L%
```

If you now draw a circle using the same center and the radius 60

```
50 CIRCLE (160,100),60
```

you will notice that the circle is too small to encompass the spokes. To compensate this you should instruct TA to draw the first spoke in a horizontal direction. This is because the radius applied to the circle represents a number of horizontal, not vertical screen points (see CIRCLE).

Note

It is possible to specify variables through the VARPTR$ function.

An error situation does not result from DRAWing off the edge of the screen (exception: TA<n>). However, DRAWing off the right of the screen in medium resolution will result in wrap round to the next horizontal line.

The X drawing command is useful, in that it enables you to call drawings which are to be used more than once, as required.

EDIT Command

Syntax

EDIT line number

"line number" specifies the line number of a line in the program. If there is no such line, an "Undefined Line Number" error message is displayed.

Purpose

Displays a line for editing.

Remarks

The EDIT command simply displays the line specified and positions the cursor under the first digit of the line number. You may then modify the line using the keys described in Chapter 2, *Full Screen Editor*.

A period (.) always refers to the current line. If you have just entered a line and want to go back and edit it, you may enter EDIT to redisplay the line.

Note

To display a block of program lines for editing purposes, you can use the LIST command.

END Statement

Syntax

END

Purpose

To terminate program execution, close all files, and return to command level ("Ok").

Remarks

END may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a "Break" message to be printed. Furthermore, END closes all files. An END statement at the end of a program is optional.

Example

```
520 IF K>1000 THEN END ELSE GOTO 20
```

ENVIRON Statement

Syntax

ENVIRON "parameter-id[=text]"

Purpose

To modify parameters in GW-BASIC's Environment String Table, especially the "PATH" parameter. This enables your program to call up another program (NCR-DOS calls this a child process), even though the program is in a different directory (see SHELL).

Remarks

For details of pathing and other NCR-DOS Environment parameters, you should refer to the *NCR-DOS* Manual and *NCR-DOS Programmer's Manual.*

"parameter-id" is the name of the parameter, for example, PATH.

"text" is the new parameter text. If "text" is omitted, then the parameter is removed from the Environment String Table and the Table is compressed. If "text" is present, it must be enclosed in double quotation marks and preceded by an equal sign or a blank. If "text" contains only a semicolon, it is regarded as non-existent.

Example

The following statement gives GW-BASIC access to the directory SALES on drive A.

```
ENVIRON "PATH=A:\SALES"
```

Note

ENVIRON allows only strings as parameters. Failure to use strings results in a "Type mismatch" error. "Out of memory" indicates that the Environment String Table is full.

Unless changed by the NCR-DOS PATH command, GW-BASIC's Environment String Table is initially empty.

ENVIRON$ Function

Syntax

ENVIRON$ ("parameter-id")
ENVIRON$ (n)

Purpose

To retrieve an Environment parameter from GW-BASICS Environment String Table.

Remarks

"parameter-id" is the name of the parameter enclosed in double quotation marks.

n is an integer expression yielding a value in the range 1 to 255. This value represents the nth parameter in the Table.

If the parameter is not found, the ENVIRON$ statement returns an empty string.

Examples

Assuming that the only parameter in the Table is that assigned in the example under the ENVIRON statement,

```
PRINT ENVIRON$ ("PATH")
```

displays

```
A:\SALES
```

The statement

```
PRINT ENVIRON$(1)
```

displays

```
PATH=A:\SALES
```

The following program saves BASIC's Environment String Table in an array so that it may be modified for a child process. After the child process completes, the Environment is restored.

```
10 DIM ENV.TBL$(10) 'Assume no more than
   10 parms
20 N.PARMS= 1 'init number of parms
30 WHILE LEN(ENVIRON$(N.PARMS)) > 0
40 ENV.TBL$(N.PARMS)=
   ENVIRON$(N.PARMS)
```

```
50 N.PARMS= N.PARMS+1
60 WEND
70 N.PARMS= N.PARMS-1 'adjust to correct
   number
80 'Now store new Environment
90 ENVIRON "MYCHILD.PARM1=SORT BY
   NAME"
100 ENVIRON "MYCHILD.PARM2=LIST BY
    NAME"
                .
                .
                .
1000 SHELL "MYCHILD" 'Runs
     "MYCHILD.EXE"
1010 FOR I= 1 TO N.PARMS
1020 ENVIRON ENV.TBL$(I) 'Restore parms
1030 NEXT I
                .
                .
                .
```

Note

If "parameter-id" is not a string, a "Type mismatch" error occurs. If the string is too long, a "String too long" error occurs. If there are too few parameters in the Table for "n" to make sense, an "Illegal function call" error occurs.

EOF Function

Syntax

EOF (file number)

Purpose

To test whether the end of a file has been reached.

Remarks

EOF returns true (-1) if there is no more data in the file. The file is empty if the next input operation (for example INPUT#, LINE INPUT#) would cause an "Input past end" error.

In the case of a communications file, the value is returned if the input buffer is empty.

The EOF condition is not significant for random access files.

Example

This example displays each record of a sequential file "NAMES", which would already have to exist. The end-of-file situation is detected as soon as it arises, thus preventing an error situation.

```
10 OPEN "NAMES" FOR INPUT AS #1
20 IF EOF(1) THEN PRINT "That's all": END
30 INPUT #1, N$
40 PRINT N$
50 GOTO 20
```

Note

EOF can also apply to redirected I/O on standard input devices. In this case, specify 0 as the "file number".

ERASE Command

Syntax ERASE <list of array variables>

Purpose To eliminate arrays from memory.

Remarks Arrays may be redimensioned after they are erased, or the previously allocated array space in memory may be used for other purposes. If an attempt is made to redimension an array without first erasing it, a "Duplicate definition" error occurs.

Example The following program uses the FRE function to demonstrate just how much space can be saved when you ERASE a large array variable which is no longer required.

```
10 PRINT "Bytes free before DIMensioning
   large array: ";FRE("")
20 DIM DINOSAUR (100,100)
30 PRINT "Bytes free after DIMensioning large
   array: ";FRE("")
40 ERASE DINOSAUR
50 DIM DINOSAUR (10,10).
60 PRINT "It's now a much smaller array.
   Bytes free: ";FRE("")
```

You see that the difference in memory requirement between the large and the re-DIMensioned array is approximately 40000 bytes.

Note CLEAR erases all program variables.

ERR AND ERL System Variables

Purpose

To establish where in the program an error occurred and the nature of that error.

Remarks

When an error handling routine is entered, the variable ERR contains the error code for the error and the variable ERL contains the line number of the line in which the error was detected. The ERR and ERL variables are usually used in IF...THEN decisions to direct program flow in the error handling routine. *Appendix C* lists the GW-BASIC error codes.

If the error-producing command was entered in direct mode, ERL contains 65535. To test whether a direct mode command was responsible for the error, enter

```
IF 65535 = ERL THEN...
```

Otherwise, ERL is written on the left side of the relational operator (e.g.=), so that the line number stated on the right will not be left out by the RENUM command during program editing, for example:

```
IF ERL < 20 THEN PRINT "The error
occurred in a line very near to the beginning of
the program"
```

Example

See ON ERROR

Note

ERR and ERL are system, not program, variables. You cannot assign values to them. You can only look to see what GW-BASIC has put in them.

ERROR Statement

Syntax

ERROR <integer expression>

Purpose

To simulate the occurrence of a GW-BASIC error, or to enable you to define error codes.

Remarks

The value of <integer expression> must be greater than 0 and less than 255. If the value of <integer expression> equals an error code already in use by GW-BASIC (see *Appendix C*), the ERROR statement will simulate the occurrence of that error and the corresponding error message will be printed. (See Example 1.)

To define your own error code, use a value that is greater than any used by GW-BASIC error codes. (It is preferable to use the highest available values, so compatibility may be maintained when more error codes are added to GW-BASIC.) This user-defined error code may then be conveniently handled in an error handling routine. (See Example 2.)

If an ERROR statement specifies a code for which no error handling has been defined, GW-BASIC responds with the "Unprintable error" error message, and terminates program execution.

Example 1

```
10 S=10
20 T=5
30 ERROR S+T
40 END
will yield
String too long in 30
```

Example 2

```
 .
 .
 .
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET";B
130 IF B>5000 THEN ERROR 210
```

```
.
.
.
400 IF ERR=210 THEN PRINT "HOUSE
    LIMIT IS $5000"
410 IF ERL=130 THEN RESUME 120
.
.
.
```

Note

ERROR is useful for testing your error handling routines. Like most other commands, ERROR can be entered in the direct mode.

EXP Function

Syntax EXP(X)

Purpose To return e (base of natural logarithms) to the power of X. X must be <= 88.02969.

Remarks If x is greater than 88.02969, the "Overflow" error message is displayed, but execution continues.

Example
```
10 X=5
20 PRINT EXP(X-1)
```
will yield
```
 54.59815
```

FIELD Statement

Syntax

FIELD [#]<file number>,<field width> AS <string variable>...

Purpose

To allocate space for variables in a random file buffer.

Remarks

Before GET or PUT can be executed, a FIELD statement must be executed to format the random file buffer.

<file number> is the number under which the file was opened. <field width> is the number of characters to be allocated to <string variable>.

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was opened. Otherwise, a "Field overflow" error occurs. (The default record length is 128 bytes.)

Any number of FIELD statements may be executed for the same file. All FIELD statements that have been executed will remain in effect throughout the program.

FIELD does not actually place any data in the random file buffer (this is done by LSET and RSET), nor does it fetch data from the random file (this is done by GET).

Examples

FIELD 1, 20 AS N$, 10 AS ID$, 40 AS ADD$, 58 AS LEFTOVER$

tells GW-BASIC that as soon as a record has been read from the random file allocated the number 1, the first 20 bytes of that record can be regarded as belonging to the string variable N$, the next 10 bytes as belonging to ID$, and so on. To view the first 20 bytes, you could issue the command PRINT N$.

One thing you must not do is give these variables contents (e.g. by using LET or INPUT), as they

have already been allocated to the random file buffer.

To write a record to a random file, you need the LSET (or RSET) statement to place the data in the random buffer, and the PUT statement to write the contents of the buffer to disk. The following example writes a new subscriber and his number as the fourth record in the file. (LSET places data left-justified in the area of the buffer delimited by the field variable.)

```
200 OPEN "R", #1, "TELNUMS",35
210 FIELD 1,25 AS NNAME$, 10 AS
    PHONENO$
220 LSET NNAME$="ISAAC NEWTON"
230 LSET PHONENO$="1234"
240 PUT 1,4
250 END
```

FILES Command

Syntax FILES ["filespec"]

Purpose To print the names of files residing on the specified disk.

Remarks If "filespec" is omitted, all the files in the current directory on the currently selected drive will be listed. "filespec" is a string which may contain question marks (?) or asterisks (*) used as universal characters. A question mark will match any single character in the filename or extension. An asterisk will match one or more characters starting at that position. The asterisk is a shorthand notation for a series of question marks. It is also required to include a pathname in "filespec" if the file is in another directory.

Examples FILES

shows all files in the current directory on the currently selected drive.

FILES "*.BAS"

shows all files with extension .BAS in the current directory of the currently selected drive

FILES "B:*.*" or FILES "B:"

shows all files on drive B.

FILES "TEST?.BAS"

shows all five-letter files whose names start with "TEST" and end with the .BAS extension in the current directory of the currently selected drive.

The filenames are displayed in a format which indicates the position of a file in its immediate surroundings in the NCR-DOS hierarchical directories. If the filename is in fact a

sub-directory, this is denoted by "<DIR>" following the directory name. Referring to the hierarchical structure shown in the following illustration, the command

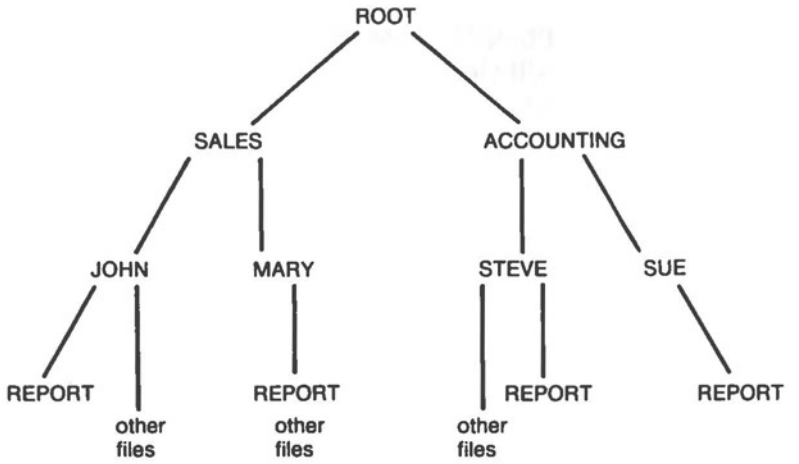
FILES "\SALES"

displays the directory entry

SALES <DIR>

The command
FILES "\SALES\MARY\"

displays the names of all the files in the directory MARY.



FIX Function

Syntax

FIX(X)

Purpose

Returns the value of the digits to the left of the decimal point in the number X, and ignores any digits to the right of the decimal point.

Remarks

The difference between FIX and INT is that FIX does not return the next lower number for negative X (see INT,CINT).

Examples

```
PRINT FIX(58.75)
will yield
 58

PRINT FIX(-58.75)
will yield
-58
```

FOR...NEXT Statement

Syntax

FOR variable=x TO y [STEP z]

.
.
.

NEXT [variable] [,variable...]

where x, y, and z are numeric expressions.

Purpose

To allow a series of instructions to be performed in a loop a given number of times.

Remarks

"variable" is used as a counter. The first numeric expression (x) is the initial value of the counter. The second numeric expression (y) is the final value of the counter. The program lines following the FOR statement are executed until NEXT is encountered. Then the counter is adjusted by the amount specified by STEP. A check is performed to see if the value of the counter is now greater than the final value (y). If it is not greater, GW-BASIC branches back to the command after the FOR statement and the process is repeated. If it is greater, execution continues with the command following NEXT. This sequence of events is often called a FOR...NEXT loop.

If STEP is not specified, the increment is assumed to be one. If STEP is negative, the counter is decreased each time GW-BASIC passes through the loop. The loop is executed until the counter is less than the final value.

The counter must be an integer or single precision numeric constant. If a double precision numeric constant is used, a "Type mismatch" error will result. Using an integer as the counter gives better program performance.

The body of the loop is skipped if the initial value of the loop exceeds the final value, assuming a positive value for STEP. In the case of STEP

being a negative value, the body of the loop is skipped if the initial value is less than the final value.

Nested Loops

FOR...NEXT loops may be nested; that is, a FOR...NEXT loop may be placed within the context of another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT may be used for all of them.

The variable(s) belonging to NEXT may be omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT is encountered before its corresponding FOR, a "NEXT without FOR" error message is issued and execution is terminated.
Using the FOR variable name with its corresponding NEXT causes a marginal loss of execution speed, but makes your program much more readable.

Example 1

```
10 K=10
20 FOR I=1 TO K STEP 2
30 PRINT I;
40 K=K+10
50 PRINT K
60 NEXT
```

will yield

```
 1   20
 3   30
 5   40
 7   50
 9   60
```

Example 2

```
10 J=0
20 FOR I=1 TO J
30 PRINT I
40 NEXT I
```

```
50 ....
```

In this example, the loop does not execute because the initial value of the loop exceeds the final value. The program skips to line 50.

Example 3

```
10 I=5
20 FOR I=1 TO I+5
30 PRINT I;
40 NEXT
```

will yield

```
 1  2  3  4  5  6  7  8  9  10
```

In this example, the loop executes ten times. The final value for the loop variable is always set before the initial value is set.

Example 4

The versatility of a FOR...NEXT loop is to be seen in its use as a means of addressing the elements of an array variable. The following program lets you enter the names of 6 animals, and for each animal 3 menus from which it can choose. The first part lets you enter the animals and their favorite dishes, and stores this information in ZOO$.

```
10 OPTION BASE 1
20 DIM ZOO$(6,4)
30 CLS
40 INPUT "What is today's date";DAY$
50 CLS
60 LOCATE 23,33:PRINT "The animals' menu"
70 FOR A% =1 TO 6
80 PRINT
100 INPUT "Which animal";ZOO$(A%,1)
110 FOR M% =2 TO 4
130 PRINT "Menu ";M%-1;" for the
    ";ZOO$(A%,1)"?  ";
140 INPUT ZOO$(A%,M%)
150 NEXT M%
160 NEXT A%
```

The second part displays the entire menu for the ZOO.

```
170 CLS
180 LOCATE 1,26
190 PRINT "The animals' menu for
    today",,,,DAY$
210 FOR A% =1 TO 6
220 PRINT
230 PRINT "The";ZOO$(A%,1);" can choose
    from",
240 FOR M% =2 TO 4
250 PRINT ZOO$(A%,M%),
260 NEXT M%
270 NEXT A%
```

FRE Function

Syntax

FRE(0)
FRE("")

Purpose

To find out the amount of memory still free, and to economize on string space.

Remarks

With a numeric argument, FRE returns the number of bytes in memory that are not being used by GW-BASIC. Arguments to FRE are dummy arguments; that is, the syntax requires them, but they are not processed by the function.

FRE("") releases memory space occupied by strings which are no longer needed, before returning the number of free bytes.

GW-BASIC does not initiate memory economizing until all free memory has almost been used up. Left this late, it can take quite some time, so using FRE periodically can shorten delays.

Example

See ERASE

Note

The number of bytes returned by the FRE function does not take into account the workspace in memory required by the GW-BASIC interpreter. Even when nothing is in the workspace, GW-BASIC reserves between 2.5KB and 4KB.

GET (Files) Statement

Syntax

GET [#]file number[,record number]

Purpose

To read a record from a random disk file into a random buffer.

Remarks

"file number" is the number under which the file was OPENed. If "record number" is omitted, the next record (after the last GET) is read into the buffer. The largest possible record number is 16,777,215; the smallest is 1. "record number" may be in the form of a mathmatical expression or variable name.

Once the record is in the buffer, your program can read it with INPUT#, LINE INPUT#, or by referring to the variables used in a FIELD definition for the buffer.

Example

See FIELD

Note

You may also use GET for reading bytes from a communications file. "record number" here has nothing to do with records; instead, it represents the number of bytes to be read from the communications buffer, provided that this number is not greater than the value set by the LEN option at OPEN "COM...

GET (Graphics) Statement

Syntax GET (x1,y1)-(x2,y2),array

Purpose To read screen graphics information (graphics modes only) into an array variable.

Remarks x1,y1 and x2,y2 are opposite corners of an imaginary rectangle. The color of each point within this rectangle is read into the specified array.

The equation
BYTES = 4 + INT
((XLEN*RESOLUTION+7)/8)*YLEN

gives the required size of the array in bytes. XLEN represents the horizontal length of the rectangle, YLEN its vertical length. RESOLUTION is 2 for low and high resolution color graphics, 1 for medium and high resolution black-and-white graphics. (This is the number of bits required to store on screen point in video RAM.)
If, for example, you wish to store a low resolution graphic design of size 15 horizontal by 12 vertical pixels, the number of bytes required is

4+INT((15*2+7)/8)*12

which yields a result of 60 bytes.

Now all you have to do is decide upon the type of numeric array in which you wish to store the design. In the section "Space Requirements" in Chapter 1, the bytes per element of an array were given as follows:

- integer array — 2
- single precision array — 4
- double precision array — 8

This means that an integer array of 26 elements is large enough to store the 15x12 graphic design. Using an integer, rather than a single or double precision array, offers the advantage that you can examine the horizontal and vertical dimen-

sion of the graphic design: the first element contains the horizontal length; the second element contains the vertical length.

Example

The following program stores a 15x12 low resolution rectangle from the top left corner of the screen in the array variable A%, and displays the contents of the first two elements of the array.

```
10 DIM A%(52)
20 SCREEN 1
30 GET (0,0)-(14,11)
40 SCREEN 0:WIDTH 80
50 PRINT A%(0),A%(1)
```

The leftmost of the two numbers displayed is the horizontal length times 2; the rightmost number is the vertical length of the rectangle.

```
30  12
```

Change line 20 to SCREEN 2 (don't forget to press <CR>). This tells GW-BASIC to use medium resolution graphics. Now RUN the program again. This time the two numbers displayed are the horizontal and vertical length of the rectangle:

```
15  12
```

Note

The complementary command, PUT, can be used for putting the contents of an array on the screen. Both GET and PUT work more efficiently if the x1 is a number that can be divided by 8 (in low and high resolution color graphics) or 16 (in medium and high resolution black-and-white graphics) without remainder.
You can also use offset coordinates, for example

```
GET (100,100)-STEP(15,-12),A%
```

determines that the graphic information of a rectangle of which the top left corner is the point 100,100 is to be read into the array variable A%.

GOSUB...RETURN Statements

Syntax

GOSUB <line number>
.
.
.
RETURN [line number]

Purpose

To branch to, and return from, a subroutine.

Remarks

"line number" in the GOSUB command is the first line of the subroutine.

A subroutine may be called any number of times in a program. A subroutine also may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

Simple RETURN statement(s) in a subroutine cause GW-BASIC to branch back to the command following the most recently encountered GOSUB. A subroutine may contain more than one RETURN, so GW-BASIC does not have to branch to the last line of the subroutine in order to return.

The "line number" option may be included in the RETURN statement to return to a specific line number from the subroutine. Use this type of return with care, however, because any other GOSUBs, WHILEs, or FORs that were active at the time of the GOSUB will remain active, and errors such as "FOR without NEXT" may result.

Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. It is often a good idea to head a subroutine with a REM line, stating what the subroutine does. To prevent inadvertent entry into the subroutine, precede it with a STOP, END, or GOTO statement that directs program control around the subroutine.

Example

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT " IN";
60 PRINT " PROGRESS"
70 RETURN
```

will yield

```
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
```

Note

The ON...GOSUB statement can be used to select a subroutine in accordance with the result of a preceding operation.

GOTO Statement

Syntax

GOTO <line number>

Purpose

To branch unconditionally out of the normal program sequence to a specified line number.

Remarks

If <line number> is the line number of an executable command, that command and those following are executed. If it is non-executable (e.g. REM) execution proceeds at the first executable command encountered after <line number>.

Example

```
10 READ R
20 PRINT "R =";R,
30 A=3.14*R^2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5,7,12
will yield
R = 5        AREA = 78.5
R = 7        AREA = 153.86
R = 12       AREA = 452.16
Out of data in 10
```

The GOTO 10 statement sets up an infinite loop. The way GW-BASIC breaks out of it is when it runs out of DATA to READ. (In fact, the loop is executed only three times.)

The following example is a true infinite loop.

```
10 THRU% = 1
20 PRINT "This is run number";THRU%;
   "through the loop"
30 THRU%=THRU% +1
40 GOTO 20
```

However, you do not have to switch off your computer to break out of the loop. For situations like this, GW-BASIC has provided a break-out possibility, namely the <Ctrl-Break> combination. This even leaves the variables intact, so you can inspect them (using PRINT...in direct mode).

Note

Entering GOTO with a line number in direct mode is a way of re-entering a program at the beginning of a line of your choice after a break in execution. It does not matter whether you interrupted the program (STOP statement, <Ctrl-Break>), or GW-BASIC was forced to stop due to an error.

The ON...GOTO statement can be used to select a program line to which GW-BASIC must branch, depending on the result of a preceding operation.

HEX$ Function

Syntax

HEX$(X)

Purpose

To return a string that represents the hexadecimal value of the decimal argument.

Remarks

X, if not already an integer, is rounded to an integer before HEX$(X) is evaluated. This integer must be in the range -32768 to 65535.

Example

```
10 INPUT X
20 A$=HEX$(X)
30 PRINT X "DECIMAL IS " A$ "
   HEXADECIMAL"
```

will yield

```
? 32
 32 DECIMAL IS 20 HEXADECIMAL
```

Note

If the value of X is negative, HEX$ returns the "two's complement". This is the same as regarding X as positive, and applying the HEX$ function to the difference between 65536 and X.

For hexadecimal to decimal conversion, see *Appendix E.*

IF...THEN...ELSE...
IF...GOTO Statements

Syntax

IF expression [,] THEN {commands(s) / line number}

[ELSE {command / line number}]

or

IF expression GOTO line number

[ELSE {command(s) / line number}]

Purpose

To make a decision regarding program flow based on the answer for an expression.

Remarks

If the "expression" is evaluated to be TRUE, the THEN or GOTO clause is executed. THEN may be followed by either a line number for branching or one or more commands to be executed. GOTO is always followed by a line number. If "expression" is FALSE, the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. If no ELSE is present, execution continues with the next executable command. A comma is allowed before THEN.

Examples

```
10 INPUT "What is the outside temperature in
   Fahrenheit";TEMP
20 IF TEMP <70 THEN PRINT "It's too cold
   for a picnic": GOTO 100
30 PRINT "It's warm enough for a picnic"
40 END
100 PRINT "So we're staying at home, today"
110 END
```

Note that only if TEMP <70 is TRUE is the GOTO 100 instruction performed.

Another way of coming to the same decision is

```
10 INPUT "What is the outside temperature in
   Fahrenheit";TEMP
```

```
20 IF TEMP <70 THEN PRINT "It's too cold
   for a picnic":PRINT "So we're staying at
   home, today" ELSE PRINT "It's warm
   enough for a picnic"
```

The following example uses line numbers instead of commands to do the same thing:

```
10 INPUT "What is the outside temperature in
   Fahrenheit";TEMP
20 IF TEMP<90 THEN 50
30 PRINT "Switch on the air conditioning"
40 END
50 IF TEMP <70 THEN 100 ELSE 80
60 PRINT "GW-BASIC will never encounter
   this line!"
80 PRINT "Conditions are right for a picnic"
90 END
100 PRINT "It's still too cold"
110 END
```

Nesting of IF...commands

IF...THEN...ELSE commands may be nested, that is, the final course of action depends on a multiple decision.

Examples

```
10 IF X>Y THEN PRINT "X IS GREATER"
   ELSE IF Y>X THEN PRINT "X IS LESS"
   ELSE PRINT "EQUAL"
```

If the statement does not contain the same number of ELSE and THEN CLAUSES, each ELSE is matched with the closest unmatched THEN. Try the following example:

```
10 INPUT "How many ounces of
   sugar";SUGAR
20 INPUT "How many measures of
   milk";MILK
30 INPUT "How many plums";FRUIT
40 IF SUGAR>4 THEN IF MILK=6 THEN
   100 ELSE IF FRUIT>10 THEN 120
50 PRINT "The mixture isn't sweet enough. Or
   it's a case of both the wrong milk quantity
```

```
    and not enough fruit. Perhaps you got every-
    thing wrong!"
60 END
100 PRINT "This cake mixture has at least
    enough sugar and the milk quantity is
    right, so I don't care how many plums are
    in it"
110 END
120 PRINT "There's enough sugar. The milk
    isn't right, but there are enough plums"
```

If there isn't enough sugar, the judgement of line 120 cannot apply, even if there are enough plums.

Note

If an IF...THEN statement is followed by a line number in direct mode, an "Undefined line" error results, unless a statement with the specified line number has previously been entered in indirect mode.

When using IF to test equality for a value that is the result of a floating-point computation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS(A-1.0)<1.0E-6 THEN...
```

This test is true if the value of A is 1.0 with a relative error of less than 1.0E-6.

The value zero is considered to represent "false"; a non-zero value to represent "true". You could therefore make a decision dependent on the contents of a numeric variable. Example:

```
210 IF IOFLAG THEN PRINT A$ ELSE
    LPRINT A$
```

This statement causes printed output to go either to the screen or the line printer, depending on the value of the variable IOFLAG. If IOFLAG is zero, output goes to the line printer; otherwise, output goes to the screen.

INKEY$ Function

Syntax

INKEY$

Purpose

Returns a character from the keyboard buffer.

Remarks

INKEY$, unlike the INPUT statement, does not wait for you to press a key. If you "miss your chance", that is, if there are no characters waiting in the keyboard buffer, INKEY$ returns a null string (length zero). Otherwise, a single character from the keyboard buffer is returned.

A number of keys return an extended string of not one but two characters. In these cases, the first character is undisplayable (ASCII code 0), only the second character is meaningful. The keys concerned are cursor movement keys, and certain shift, Ctrl and Alt key combinations (see *Appendix B).*

A further difference between INKEY$ and INPUT is that INKEY$ does not echo your keyboard entry on the screen.

The character returned by INKEY$ must be assigned to a string variable before it can be examined.

Example

The following example is useful for programs in which the user is to be allowed as much time as he or she wishes to look at a screen display. Program execution continues when any key is pressed.

```
210 PRINT "Press any key to continue"
220 K$=INKEY$:IF LEN(K$)=0 THEN GOTO
    220
```

The next example repeatedly reads the keyboard to see if you have pressed the space bar. But it does not go on checking forever: after INKEY$ has looked at the keyboard LIMIT% number of times, the program gives up waiting and BEEPs you for being so slow. You might use something like this in video games.

```
10 LIMIT% = 1000
20 FOR I% = 1 TO LIMIT%
30 K$ = INKEY$:IF K$ <>" " THEN 60
40 PRINT "Just in time!"
50 GOTO 20
60 NEXT I%
70 BEEP:CLS:PRINT "Too Slow!"
80 GOTO 20
```

To check for and read one of the extended strings:

```
10 K$ = INKEY$
20 IF LEN(K$)=2 THEN K$=RIGHT$(K$,1)
```

INKEY$ does not annul the special functions of the following key combinations:

<Ctrl-Break> (break out of program)
<Ctrl-NumLock> (system pause)
<PrtSc> (print contents of screen)
<Alt-Ctrl-Del> (system reset)

INP Function

Syntax	INP(I)
Purpose	To return the byte read from port I. I must be in the range 0 to 65535.
Remarks	INP is the complementary function to the OUT statement.
Example	100 A% = INP(64) reads a byte from port 64 and assigns it to the variable A%.
Note	INP performs the same function as the assembly language IN instruction.

INPUT Statement

Syntax

INPUT[;] [prompt string;]variable [,variable]...

Purpose

To allow input from the keyboard during program execution.

Remarks

When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data. If "prompt string" is included, the string is printed before the question mark. GW-BASIC then waits for you to type data at the keyboard with a concluding <ENTER>.

A comma may be used instead of a semicolon after the prompt string to suppress the question mark. For example, INPUT "ENTER BIRTHDATE",B$ will print the prompt with no question mark.

The data entered is assigned to the variable(s) given in "variable". The number of data items you enter must be the same as the number of variables in the list. You must type a comma before each data item other than the first.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. String items input need not be surrounded by quotation marks unless they contain commas or start or end with a significant number of blanks.

Responding to INPUT with too many or too few items or with the wrong type of value (numeric instead of string, etc.) causes the message "?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given. If only a single variable is specified in the INPUT statement, you may omit the data item

and simply press <ENTER>. GW-BASIC then supplies a zero for a numeric variable, or an empty string for a string variable.

Examples

```
10 INPUT X
20 PRINT X "SQUARED IS"; X∧2
30 END
will yield
? 5          (The 5 is an example of what you
              might enter in response to the
              question mark.)
 5 SQUARED IS 25

10 PI=3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A=PI*R∧2
40 PRINT "THE AREA OF THE CIRCLE
IS";A
50 PRINT
60 GOTO 20
will yield
WHAT IS THE RADIUS? 7.4 (You enter 7.4)
THE AREA OF THE CIRCLE IS 171.9464

WHAT IS THE RADIUS?
and so on.
```

To break-out of this program, use <Ctrl-Break>

Note

The >stdin option, which you may include in the NCR-DOS command to load GW-BASIC, allows you to appoint a file which will provide data for INPUT in place of the keyboard. In this case, each response must be concluded by a <Ctrl-Z>. If this character is missing, a "Read past end" error occurs, all files are closed, and GW-BASIC returns control to the NCR-DOS (not "Ok") command level. Similarly, if you press <Ctrl-Break>, the return is to the NCR-DOS command level.

If you use <Ctrl-Break> to interrupt execution of an INPUT statement, and then return to the program with CONT, GW-BASIC resumes execu-

tion with the INPUT (not the subsequent) statement.

The following example assumes that a file exists containing information about telephone subscribers (name, number) in records each 35 bytes in length. The first line opens the file for random access.

```
10 OPEN "R",#1, "TELNUMS",35
```

The first record contains no more than the number of subscribers up to a maximum of 99. A FIELD definition suitable for this first record is:

```
20 FIELD 1,2 AS SUBSCRIB$,33 AS UNUSED$
```

The remaining records, containing actual names and numbers, could conform to the following divisions.

```
30 FIELD 1,25 AS NNAME$,10 AS
   PHONENO$
```

The first thing to do is to GET the first record, and look at the first two bytes of that record to see how many subscriber records are in the file (this number is converted from a string to an integer value):

```
40 GET #1
50 TOTAL% = CVI(SUBSCRIB$)
```

The rest of the program GETs each record from the disk file and displays each name and phone number of the screen:

```
60 FOR LOOP% = 2 TO TOTAL%
70 GET #1, LOOP%
80 PRINT NNAME$,PHONENO$
90 NEXT LOOP%
100 END
```

INPUT# Statement

Syntax

INPUT#file number,variable list

Purpose

To read data items from a sequential device or file and assign them to program variables.

Remarks

"file number" is the number used when the file was OPENed for input. "variable list" contains the variable names to which items in the file will be assigned. (The variable type must match the type specified by the variable name.) With INPUT#, no question mark is printed, as with INPUT.

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. With numeric values, leading spaces, carriage returns, and linefeeds are ignored. The first character encountered that is not a space, carriage return, or linefeed is assumed to be the start of the number. The number concludes with a space, carriage return, linefeed, or comma.

If GW-BASIC is scanning the sequential data file for a string item, it will also ignore leading spaces, carriage returns, and linefeeds. The first character encountered that is not a space, carriage return, or linefeed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string, concluded by a comma, carriage return, or linefeed (or after 255 characters have been read). If end-of-file is reached when a numeric or string item is being INPUT, the item is terminated.

Example

See Chapter 5, *Files and Devices*".

Note **INPUT# can also be used with a random access file.**

INPUT$ Function

Syntax

INPUT$(X[,[#]Y])

Purpose

To return a string of X characters, read from file number Y. If the file number is not specified, the characters will be read from the keyboard.

Remarks

If the keyboard is used for input, no characters will be echoed on the screen. All control characters are passed through except <Ctrl-Break>, which can be used to interrupt the execution of the INPUT$ function. When responding to INPUT$ via the keyboard, it is not necessary to press <ENTER>. X limits the number of input characters anyway.

Example

```
5 'LIST THE CONTENTS OF A SEQUENTIAL
  FILE IN HEXADECIMAL
10 OPEN"I",1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
```

Note

INPUT$ and INKEY$ read all keyboard entries and not just printable characters. To detect, for example, cursor movement keys, use these functions and not INPUT or LINE INPUT.

Similarly, you should use INPUT$ for reading communications files (rather than INPUT# or LINE INPUT#), as any ASCII character may be significant.

INSTR Function

Syntax

INSTR([I,]X$,Y$)

Purpose

To search for the first occurrence of string Y$ in X$ and return the position at which the match is found. Optional offset I sets the position X$ for starting the search.

Remarks

I must be in the range 1 to 255. If I is greater than the number of characters in X$ or if X$ is null or Y$ cannot be found, INSTR returns 0. If Y$ is null, INSTR returns I, or 1 if I was not specified. X$ and Y$ may be string variables, string expressions, or string constants.

Example

```
10 X$="ABCDEB"
20 Y$="B"
30 PRINT INSTR(X$,Y$);INSTR(4,X$,Y$)
will yield
 2  6
```

The first INSTR searches for "B" from the beginning of the string variable X$, and finds one at position 2. The second INSTR starts searching at position 4, and, therefore, cannot find the "B" at position 2 but does find the "B" at position 6.

Note

If I is out of range, that is, beyond the length of X$, an "Illegal function call" error occurs. To find the length of a string you can use the LEN function.

INT Function

Syntax INT(X)

Purpose To return the largest integer that is less than or equal to the numerical value X.

Examples
```
PRINT INT(99.89)
will yield
 99

PRINT INT(-12.11)
will yield
-13
```

Note See the CINT and FIX functions, which also return integer values.

KEY Statement

Syntax

Function Keys
KEY key number,"string expression"
KEY ON
KEY OFF
KEY LIST
or
User Defined Key Traps
KEY key
number,CHR$(mode)+CHR$(Keyboard)

Purpose

Function Keys

Allows you to assign a string expression to programmable function keys. You may assign a string of up to 15 characters to any one or all of the keys. When you later press the key, the string will be input to GW-BASIC. KEY ON/OFF enables you to view or hide the contents of the programmable Function Keys. KEY LIST displays the Function Keys and their definitions.

Remarks

"key number

Specifies the number of a programmable Function Key in the range 1 to 10 (see list below).

"string expression"

Specifies the string expression which will be assigned to the programmable Function Key.

Initially, the programmable function keys are assigned the following values by GW-BASIC:

F1 LIST	F6,"LPT1<**ENTER**>
F2 RUN<**ENTER**>	F7 TRON<**ENTER**>
F3 LOAD"	F8 TROFF<**ENTER**>
F4 SAVE"	F9 KEY
F6 CONT<**ENTER**>	F10 SCREEN 0,0,0<**ENTER**>

KEY "key number","string expression"
Assigns the string expression to the specified

key. The string expression may be 1 to 15 characters in length. If it is longer than 15 characters, only the first 15 characters are assigned.

If you specify a value for "key number" which is not in the range 1 to 10, an "Illegal Function Call" error occurs. The previous key string assignment is retained.

Assigning a string of length 0 to a programmable Function Key disables the key. It will remain disabled until another string expression is assigned to it.

KEY ON
The first six characters of each Function Key are displayed on the 25th line of the screen, just as after loading GW-BASIC. If you are using a display WIDTH of 40, only five Function Keys are displayed.

KEY OFF
Erases the programmable Function Key display from the 25th line, but it does not disable the function keys.

KEY LIST
Lists all 10 programmable Function Key values on the screen. All 15 characters of each Function Key are displayed.

When a programmable Function Key is assigned, the INKEY$ function returns one character of the string each time it is called. If the programmable Function Key is disabled, INKEY$ returns a string of length 2. The first character is binary zero, and the second is the key scan code (see *Appendix B).*

Examples

In the following example, the statement in line 10 assigns the string 'MENU' with a concluding <ENTER> to Function Key 1. This assignment might be used in a program to select a menu

display when entered by the user. Line 20 disables the key.

```
10 KEY 1,"MENU"+CHR$(13)
20 KEY 1,""
```

Purpose

User Defined Key Traps
GW-BASIC allows you to define six additional key traps. The trapped key must be in the Ctrl, Shift or Alt mode.

Remarks

To set a trap, a KEY command containing the following elements is required:

KEY key number,CHR$(mode) + CHR$(keyboard)

"key number"
A number you choose to define in the range 15 to 20 (not a Keyboard key, etc).

"mode"

A hexadecimal value as follows:

Caps Lock	&H40
Num Lock	&H20
Alt	&H08
Ctrl	&H04
Shift	&H01 or &H02 or &H03

Combined mode key actions are accomplished by combining the appropriate codes. For example &H04 + &H01 means <Ctrl-Shift>.

"keyboard"
A number representing the position on the keyboard of the key to be trapped (see *Appendix F*).

Example

The following program sets up a trap for the key combination <Ctrl-Shift-X>:

```
10 KEY 15, CHR$(&H04+&H03)+CHR$(45)
20 ON KEY(15) GOSUB 1000
30 KEY (15) ON
 .
 .
 .
1000 PRINT "Somebody has pressed Ctrl-
     Shift-X!"
 .
 .
 .
```

Line 20 states that whenever this key combination is pressed, and provided that the trap is enabled (line 30, see KEY(n) statement), GW-BASIC will branch to the subroutine at line 1000.

Note

Key trapping cannot be used to override the Ctl-PrtSc function, nor does it affect Function or cursor movement keys.

However, you can override the GW-BASIC Ctrl-Break function, with the result that you can no longer use this key combination to break out of a programmed return to the GW-BASIC "Ok" level. Similarly, you can override the Ctrl-Alt-Del (system reset) combination.

KEY(N) Statement

Syntax

KEY(n) ON
KEY(n) OFF
KEY(n) STOP

where (n) is the number of a programmable Function Key, a cursor movement key, or one of the six user-defined key traps (see KEY Statement).

Purpose

To enable or disable trapping the operation of one of the above mentioned keys or key combinations.

Remarks

n is a numeric expression in the range 1 to 20:

1 — 10	the appropriate Function Key
11	Cursor Up
12	Cursor Left
13	Cursor Right
14	Cursor Down
15 — 20	User defined trappable key combination.

The KEY(n) ON statement enables trapping of the key or key combination specified by n. While trapping is enabled, and if a non-zero line number is specified in the ON KEY statement, GW-BASIC checks before every command to see if the specified key has been used. If it has, the GW-BASIC branches to the subroutine starting at the line given in the ON KEY statement.

KEY(n) OFF disables the event trap. If an event takes place, it is not recorded.

KEY(n) STOP disables the event trap, but if an event occurs, it is recorded and an ON KEY statement will be executed as soon as trapping is enabled.

Example

See KEY.

Note

A KEY(n) statement may not precede an ON KEY(n) statement.

KEY(n) ON does not have any effect on whether the Function Keys are displayed in the 25th screen line.

KILL Command

Syntax

KILL "filespec"

Purpose

To delete a file from disk.

Remarks

If a KILL command is given for a file that is currently OPEN, a "File already open" error occurs.

KILL can be used for all types of disk files both program and data files. The filespec may contain question marks (?) or asterisks (*) (universal characters). A question mark will match any single character in the filename or extension. An asterisk will match one or more characters starting at its position.

Examples

```
200 KILL "A:OBSOLETE.DAT"
```

deletes the file with the name OBSOLETE.DAT on drive A.

```
200 KILL
"LOCAL1\LOCAL1A\UNUSED.BAS"
```

deletes the file UNUSED.BAS from the subdirectory LOCAL1A on the current disk.

Note

KILL is similar to the NCR-DOS DELETE command.

LCOPY Command

Syntax LCOPY

Purpose To output the screen display to a printer.

Remarks The LCOPY command enables you to "dump" a screen, that is, to print what is currently displayed on a printer.

The time required for printing depends on the degree of resolution that you have selected.
To print screen graphics an a printer you can also use the GRAPHICS command. Refer to your NCR-DOS manual for further in formation on how to invoke this command.

LEFT$ Function

Syntax

LEFT$(X$,I)

Purpose

To return a string comprising the leftmost I characters of X$.

Remarks

I must be in the range 0 to 255. If I is greater than the number of characters in X$, the entire string (X$) will be returned. If I = 0, the null string (length zero) is returned.

Example

```
10 A$="BASIC"
20 B$=LEFT$(A$,3)
30 PRINT B$
```

will yield

```
BAS
```

Also see the MID$ and RIGHT$ functions.

Note

To find out the length of a string, you can use the LEN function.

LEN Function

Syntax LEN(X$)

Purpose To return the number of characters in X$, including blanks and special characters.

Example
```
10 X$="PORTLAND,  OREGON"
20 PRINT LEN(X$)
```
will yield
```
 16
```

LET Statement

Syntax [LET] variable=expression

Purpose To assign the value of an expression to a variable.

Remarks Notice that the word LET is optional; that is, the equal sign is sufficient for assigning an expression to a variable name.

Example

```
110 LET D=12
120 LET E=12^2
130 LET F=12^4
140 LET SUM=D+E+F
150 LET TEXT$="Words"
 .
 .
 .
```

is the same as

```
110 D=12
120 E=12^2
130 F=12^4
140 SUM=D+E+F
150 TEXT$="Words"
 .
 .
 .
```

Note The same variable may appear on both sides of the equal sign, for example:

```
100 INPUT "Enter any number", NUMB
110 NUMB = NUMB/3
120 PRINT "Your number divided by three =
    ";NUMB
```

If the expression to the right of the equal sign does not yield a result of the same type as that of the variable to the left of the equal sign, a "Type mismatch" error occurs.

LINE Statement

Syntax LINE [(x1,y1)](x2,y2)[,[color] [,B[F]] [,style]]

Purpose Draws a line, box, or filled-in box on the screen.

(x1,y1),(x2,y2)
Specifies the coordinates in either absolute or offset form (see Chapter 3 "Screen Display"). If (x1,y1) point coordinates are not specified, the beginning point of the line is the last point referred to in a graphics command.

'color'
Specifies color of line, box, or filled-in box. In low and high resolution color graphics, the number must be in the range 1 - 3, this representing a color from the color palette, or 0 (the background color). Default is 3. In medium and high resolution black-and-white graphics, 0 indicates black, 1 indicates white. Default is 1.

B or BF
Specifies box or filled-in box. The B tells BASIC to draw a rectangle with the points (x1,y1) and (x2,y2) as opposite corners. This avoids having to give four LINE commands to perform the same function:

```
LINE (x1,y1)-(x2,y1)
LINE (x1,y1)-(x1,y2)
LINE (x2,y1)-(x2,y2)
LINE (x1,y2)-(x2,y2)
```

The BF tells BASIC to draw the same rectangle as B and also to fill in the interior points in the same color as B. (You cannot use "style" with B.)

"style" is an option for drawing the line or box not with a continuous line but with a dotted or dashed line, or any other pattern you choose. "style" is a number in the range 0 to 65535. The style of drawing corresponds to the 16 bit binary representation of the number. A dotted line requires the bit pattern

1010101010101010 or
0101010101010101

This corresponds to the decimal number 43690 (first pattern) or 21845 (second pattern). Experienced programmers will probably prefer to use the hexadecimal notations &HAAAA and &H5555

The pattern set by style applies to the first 16 screen points of the line, and is repeated thereafter until the end of the line. Any screen points not set retain their old appearance. Therefore, before drawing an intermittent line, you may wish to first draw a continuous line using the background color (0), in order to achieve a single background color for the whole length of the line. This is a consideration which applies in particular to color graphics.

Examples

```
10 INPUT "X2 and Y2 and color";
   X2%,Y2%,COL%
20 SCREEN 1,0:COLOR 0,1
30 FOR D% =1 TO 400:NEXT D%
40 LINE -(X2%,Y2%),COL%
50 IF INKEY$ ="" THEN 40
60 SCREEN 0:WIDTH 80
```

This program asks you for the end coordinates and the color of a line to be drawn from the center of the screen (the "last point referenced" following the SCREEN 1 statement). The line is then drawn and remains on the screen until you press any key. (The short delay caused by line 30 is merely to give the display time to settle after changing mode.)

The following example connects the corners of the screen with one another using high resolution line drawing:

```
10 SCREEN 2 :CLS
20 LINE (0,0)-(639,199),,B
30 LINE (0,0)-(639,199)
40 LINE (0,199)-(639,0)
50 IF INKEY$ ="" THEN 50
```

```
60 SCREEN 0
```

As "style" is not specified, continuous lines are drawn. Now re-write lines 20 to 40 as follows:

```
20 STYLE% =21845 :LINE
   (0,0)-(639,199),,B,STYLE%
30 LINE (0,0)-(639,199),,,STYLE%
40 LINE (0,199)-(639,0),,,STYLE%
```

This now produces dotted lines. Change the value of STYLE% to 1 and the dots will be few and far between. The value 3855 produces dashed lines. The binary equivalent is

0000111100001111 (hexadecimal 0F0F).

The following example draws a line using offset coordinates, that is, relative to the last point referenced (in this case 150,100):

```
10 LINE (150,100)-STEP (30,—30)
```

The following draws random size boxes up to a maximum of 50 X 50 points at random positions in random colors

```
10 SCREEN 1,0
20 COLOR 0, RND*4
30 LINE
(RND*319,RND*199)-STEP(RND*50,RND*50)
,RND*3, BF
40 GOTO 20
```

Note

If LINE attempts to draw beyond the edges of the graphic display area, there is no wrap around to another part of the screen. Those parts of the drawing that are out of bounds are simply clipped away. GW-BASIC does not regard this as an error condition.

LINE INPUT Statement

Syntax

LINE INPUT[;] ["prompt string";]
string variable

Purpose

To read an entire line (up to 254 characters) from the keyboard to a string variable, ignoring delimiters.

Remarks

"prompt string" is a string constant that is displayed on the screen before input is accepted. A question mark is not printed unless it is part of "prompt string". The subsequent keyboard input is assigned to "string variable".

If LINE INPUT is immediately followed by a semicolon, then the cursor remains in the same line, even after you have pressed <ENTER>.

A LINE INPUT statement may be aborted by typing <Ctrl-Break>. GW-BASIC then returns to command level ("Ok").

Example

See "LINE INPUT#".

Note

If you interrupt execution of a LINE INPUT statement by pressing <Ctrl-Break>, and then return to the program with CONT, GW-BASIC resumes execution with the LINE INPUT (not the subsequent) statement.

LINE INPUT # Statement

Syntax

LINE INPUT# file number,string variable

Purpose

To read an entire line (up to 254 characters), ignoring delimiters, from a sequential disk data file to a string variable.

Remarks

"file number" is the number under which the file was OPENed. "string variable" is the variable name to which the line will be assigned. LINE INPUT# reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/linefeed sequence. The next LINE INPUT# reads all characters up to the next carriage return. (If a linefeed/carriage return sequence is encountered, it is preserved.)

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a GW-BASIC program saved in ASCII format (see SAVE) is being read as data by another program.

Example

```
10 OPEN "O",1,"LIST"
20 LINE INPUT "CUSTOMER
INFORMATION? ";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "I",1,"LIST"
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
will yield
CUSTOMER INFORMATION? LINDA
JONES   234,4   MEMPHIS
LINDA JONES   234,4   MEMPHIS
```

This example reads information with commas and other delimiters from the keyboard into C$. This information is written to the file LIST. Afterwards the sequential file is re-opened and the information is read back in C$ and then displayed.

LIST Command

Syntax

LIST [line number 1] [-[line number 2]] [filespec]

"line number 1","line number 2"
Numbers in the range 0 to 65529 which specify the span of program lines to be displayed.
"filespec"
A file or device to which the program listing is to be directed. If filespec is omitted, the program lines are listed on the screen.

Purpose

Allows a program to be listed, usually on the screen.

Remarks

If you do not specify line numbers, the entire program is listed.

You may terminate any listings to the screen by pressing <Ctrl-Break>. <Ctrl-Num Lock> temporarily suspends listing, that is, until you press an key.

Examples

LIST

displays the entire program.

LIST 100

displays line 100 only.

LIST 80-

The hyphen has the effect that not only line 80 but also every line with a higher number is displayed.

LIST -120

Every line up to and including line 120 is displayed.

LIST 50-210

This time, listing is confined to this inclusive line span.

LIST 1000-"COPY2.BAS"

Writes a copy of the program in ASCII format (see SAVE) to the file COPY2.BAS, starting at line 1000.

Note

If LIST is executed in indirect mode, GW-BASIC returns immediately afterwards to the command level ("Ok").

Refer to Chapter 5 *Files and Devices* for information about using device names (e.g. LPT1:) for "filespec".

LLIST Command

Syntax	LLIST [line number 1] [-[line number 2]]
Purpose	To list a program on the printer.
Remarks	LLIST works in an identical way to LIST, except that you cannot use "filespec".

LOAD Command

Syntax LOAD filespec[,R]

Purpose To load a file from disk or a device into memory.

Remarks The "filespec" must include the filename that was used when the file was SAVEd, but you need not specify the extension if it is .BAS.

The R option automatically runs the program after it has been loaded.

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program. However, if the R option is used with LOAD, the program is RUN after it is LOADed, and all open data files remain, so that the newly run program can use them.

Example LOAD "STRTRK",R

Loads and runs the program STRTRK.BAS. (This has the same effect as RUN "STRTRK".)

LOAD "B:MYPROG"
Loads the program MYPROG.BAS from the disk in drive B, but does not run the program.

Note "filespec" may include a pathname.

LOC Function

Syntax

LOC(file number)

where "file number" is the number under which the file was opened.

Purpose

With random access files, LOC returns the number of the last record read or written.

With sequential access files, LOC returns the number of records read from, or written to, the file since it was opened.

Remarks

When a file is opened for sequential input, GW-BASIC reads the first sector of the file, so LOC will return a 1 even before any input from the file occurs.

For a communications file, LOC is used to determine if there are any characters in the input queue waiting to be read. If there are more than 255 characters in the queue, LOC returns 255. Since interpreter strings are limited to 255 characters, this practical limit eliminates the need to test for string size before reading data into it. If fewer than 255 characters remain in the queue, the value returned by LOC is the actual number of characters waiting to be read.

Examples

```
200 IF LOC(1)>50THEN STOP
```

stops the program if reading has gone beyond the 50th record.

The following example is useful for re-writing a random access file record which has just been read:

```
200 PUT #1,LOC(1)
```

LOCATE Statement

Syntax

LOCATE[row] [,[col] [,[cursor] [,[top] [,bottom]]]]

Purpose

To position the cursor on the screen, define its size, and to make it visible and invisible.

Remarks

"row"
Specifies the screen line number in the range 1 to 25 where the cursor is to appear.

"col"
Specifies the screen column number in the range 1 to 40 (using width 40) or 1 to 80 (using width 80) where the cursor is to appear.

"cursor"
A value indicating whether the cursor is to be visible. Specify 1 for visible, 0 for invisible. "cursor" does not apply to the graphics modes.

'top'
A character position on the screen consists of 14 or 8 scan lines, depending on whether you are using a monochrome display adapter or a color graphics display adapter. The scan lines are numbered from the top of the character position as 0 to 13 and 0 to 7, respectively. 'top' determines the topmost scan line of the cursor.

"bottom"
Determines the lowermost line of the cursor. The area between "top" and "bottom" is filled in at whatever character position the cursor is situated. If "top" is given as a higher scan line number than "bottom", the cursor wraps round to the upper portion of the character position, giving a cursor in two parts. "top" and "bottom" do not apply to the graphics modes.

Positioning the cursor determines where character output to the screen is displayed.

You may omit any of the command parameters. The current value continues to apply for an omitted parameter.

If you turn off the Function Key display in the bottom screen line (KEY OFF), you can use all 25 lines. GW-BASIC does not normally write in line 25, but it will do so, if you place the cursor there.

When a program is running, GW-BASIC normally turns the cursor off. The command LOCATE ,,1 turns it back on.

Examples

```
LOCATE 1,1
```

Moves the cursor to the top left corner of the character display area.

```
LOCATE ,,,0,7
```

Sets a block cursor on a color screen, but does not change the position or visibility of the cursor.

```
LOCATE 24,1,1,6,1
```

Places a two-part visible cursor at the beginning of the 24th character display line

Note

Allowable ranges for LOCATE parameters are 1 to 25 for "row", 1 to screen WIDTH for "col", 0 or 1 for "cursor", and 1 to 31 for "top" and "bottom". Out of range values produce an "Illegal function call" error.

LOF Function

Syntax LOF(file number)

Purpose To return the length of the file in bytes.

Remarks You can also apply LOF to a communications file. In this case, the function returns the amount of free space in the input buffer. The maximum amount of free space is normally 256 bytes, but you can change this using the /C option when loading GW-BASIC.

Example The following example reads the last record of a random access file into the buffer. The record length must already have been stored in RECSIZ%:

```
10 OPEN "AFILE" AS #1
20 GET #1, LOF(1)/RECSIZ%
```

Note If you apply LOF to files created under IBM BASIC 1.10, the length returned is a multiple of 128. For example, a true length of 290 yields the result 384.

LOG Function

Syntax LOG(X)

Purpose To return the natural logarithm of X. X must be greater than zero.

Example
```
PRINT LOG(45/7)
```
will yield
```
 1.860752
```

Note LOG is performed in single precision unless you specify the /D option when loading GW-BASIC. This option results in LOG and other "Resident" functions being calculated with double precision.

LPOS Function

Syntax LPOS(X)

where X is the identifier for the line printer.

Purpose To return the current character in the print buffer that is ready to be printed.

Remarks LPOS does not necessarily give the physical position of the print head.

The value of X determines which printer is being tested:

0 or 1	LPT1:
2	LPT2:
3	LPT3:

Example

```
100 IF LPOS(X)>60 THEN LPRINT CHR$(13)
```

This program line ensures that no more than 60 characters are printed in any line CHR$(13) produces a carriage return <ENTER>.

LPRINT and LPRINT USING Statements

Syntax

LPRINT [list of expressions][;]

LPRINT USING string expression;list of expressions[;]

Purpose

To print data at the line printer (LPT1:)

Remarks

"list of expressions"
contains the items which are to be printed. These items are numeric and/or string expressions, and must be separated from one another by commas or semicolons.

"string expression"
is a string constant or variable giving the format to be used for printing.

Details regarding printing formats are the same as those for displaying characters on the screen. Please refer to PRINT and PRINT USING.

LPRINT assumes that you are using a printer with a line width of 80 characters and automatically inserts a carriage return/line feed sequence accordingly. You may change this width value by means of the WIDTH"LPT1:" command.

LPRINT issues a "Device Timeout" error if the printer or other device receiving the LPRINT output is slow in responding. You can give the device more time by trapping the error, for example:

```
9900 IF ERR = 24 THEN RESUME
```

(error number 24 is the "Device Timeout" error).

Note

Use of LPRINT is not confined to printable characters. Your program can use LPRINT to activate printer functions, such as head movement and character font selection. A number of the codes are standardized, for example:

```
LPRINT CHR$(12);
```

should produce a form feed on the printer. Other codes are specified to the printer being used. For these refer to your printer documentation.

LSET and RSET Statements

Syntax

LSET string variable=string expression
RSET string variable=string expression

Purpose

To move data from memory to a random file buffer (in preparation for a PUT statement).

Remarks

"string variable"
is a variable which has already been defined by FIELD.

"string expression"
contains the information to be placed in the random access file buffer at the position indicated by string variable.

If "string expression" requires fewer bytes than specified for "string variable", LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions.) If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before they are LSET or RSET (see MKI$, MKS$, MKD$).

Examples

```
150 LSET W$ = "Very + WEATHER$
```

places the string expression in the buffer at the position indicated by the FIELD variable W$. The string is set to the left of the FIELD area and, at the right of this area, it is padded with blanks or truncated, if necessary.

The following example converts a numeric value to a string before placing it right-justified in the buffer:

```
30 ABC$ = MKI$(14)
40 RSET A$ = ABC$
```

Note

You may apply LSET and RSET to non-FIELDed variables. This is useful for formatting text which is to be displayed on the screen or printed.

MERGE Command

Syntax MERGE "filespec"

Purpose To merge a specified disk file into the program currently in memory.

Remarks The "filespec" must include the filename used when the file was saved. The file must have been saved in ASCII format (see SAVE). If it was not, a "Bad file mode" error occurs. "filespec" may include a pathname.

If any lines in the disk file have the same line numbers as lines in the program in memory, the lines from the file on disk will replace the corresponding lines in memory.

GW-BASIC always returns to command level ("ok") after executing a MERGE command.

Example MERGE "NUMBERS"

merges the GW-BASIC program file NUMBERS.BAS residing in the current directory of the current disk into the program already present in memory.

MID$ Statement

Syntax

MID$(string expression1,n[,m])=string expression2

where n and m are integer expressions.

Purpose

To replace a portion of one string with another string.

Remarks

The characters in "string expression1", beginning at position n, are replaced by the characters in "string expression2. The optional "m" refers to the number of characters from "string expression2" that will be used in the replacement. If "m" is omitted, all of "string expression2" is used. However, regardless of whether "m" is omitted or included, the replacement of characters never goes beyond the length of "string expression1".

Example

See MID$ Function.

Note

The values n and m must be in the range 1 to 255, otherwise an "Illegal function call" error occurs.

MID$ Function

Syntax

MID$(X$,n[,m])

Purpose

To return a string of length m characters from X$, beginning with the nth character.

Remarks

n and m must be integer expressions in the range 1 to 255. If m is omitted or if there are fewer than m characters to the right of the nth character, all rightmost characters beginning with the nth character are returned. If n is greater than the number of characters in X$ (LEN(X$)), MID$ returns a null string.

Example

```
10 A$="GOOD "
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
```

will yield

```
GOOD EVENING
```

The MID$ function is useful for looking at the characters of a string one by one. The following program asks you to enter a text. The program checks whether there are any characters in that string that are not letters. As soon as such a character is found, its position is displayed on the screen.

```
10 INPUT "Please enter a text ",T$
20 IF LEN(T$) = 0 THEN GOTO 10
30 FOR L% = 1 TO LEN(T$)
40 CHAR$ = MID$(T$,L%,1)
50 IF CHAR$<"A" OR CHAR$> "z" OR
   CHAR$> "Z" AND CHAR$< "a" THEN
   P% =L%:GOTO 100
60 NEXT L%
70 PRINT "Text is all letters" :END
100 PRINT "A non-letter character" ;CHAR$;
    "was found at position ",P%
110 END
```

Note

If n or m is out of range, an "Illegal function call" error occurs.

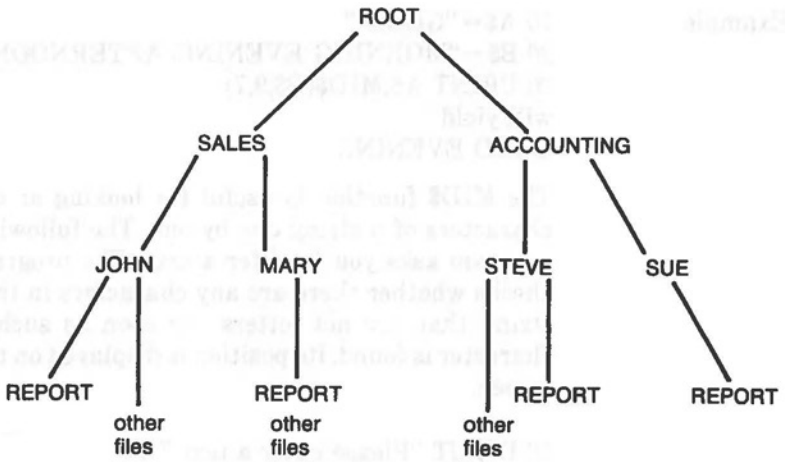
MKDIR Command

Syntax MKDIR "path"

Purpose To create a directory on the specified drive.

Remarks "path" is a string expression not exceeding 128 characters identifying a new directory which is to be created. For details about paths and directories you should refer to your *NCR-DOS* manual.

Examples Given the following hierarchical structure, and assuming you are presently in the root directory



the command

MKDIR "RESEARCH"

Creates a sub-directory of that name to ROOT (that is, on the same level as SALE and ACCOUNTING).

MKDIR "RESEARCH\MARTHA"

Creates a sub-directory MARTHA in the directory RESEARCH.

Note MKDIR works in the same way as the NCR-DOS command of the same name.

MKI$, MKS$, MKD$ Functions

Syntax

MKI$ (integer expression)
MKS$ (single precision expression)
MKD$ (double precision expression)

Purpose

To convert numeric values to string values.

Remarks

Any numeric value that is to be placed in a random file buffer with an LSET or RSET statement must first be converted to a string. MKI$ converts an integer to a 2-byte string. MKS$ converts a single precision number to a 4-byte string. MKD$ converts a double precision number to an 8-byte string.

Example

```
90 AMT=(K+T)
100 FIELD #1,4 AS D$,20 AS N$
110 LSET D$=MKS$(AMT)
120 LSET N$=A$
130 PUT #1
```

converts the single precision value in AMT into a 4-byte string, places it in the random access file buffer, and writes it as part of the record to a file.

Note

The complementary functions, that is, the functions which convert strings to numeric values are CVI, CVS, and CVD.

The STR$ function also converts a numeric to string value, but it does not necessarily retain the length in bytes of the original number.

NAME Command

Syntax NAME <old filename> AS <new filename>

Purpose To change the name of a disk file.

Remarks <old filename> must exist and <new filename> must not exist; otherwise, an error will result.

A file may not be renamed with a new drive designation. If this is attempted, a "Rename across disks" error is generated. After a NAME command, the file exists on the same disk, in the same area of disk space, but with the new name.

Example NAME "ACCTS" AS "LEDGER"

In this example, the file that was formerly named ACCTS will now be named LEDGER.

Note When you are using the name command, GW-BASIC does not assume a file extension of .BAS.

The specification of a pathname is not allowed.

Before using NAME, ensure that the file is closed.

NEW Command

Syntax NEW

Purpose To delete the program currently in memory, close all files, and clear all variables. If the TRacer is ON, it is turned off.

Remarks NEW is entered in direct mode to clear memory before entering a new program. GW-BASIC always returns to command level after a NEW is executed.

NEW closes all files and turns tracing off.

OCT$ Function

Syntax OCT$(X)

Purpose To return a string that represents the octal value of the decimal argument. X is rounded to an integer before OCT$(X) is evaluated.

Example
```
PRINT OCT$(24)
```
will yield
```
 30
```

See the HEX$ function for details on hexadecimal conversion.

ON COM(n) Statement

Syntax

ON COM(n) GOSUB line

n
Communications channel number (1 or 2).

line
Line number of the beginning of the trap routine. A line number of 0 disables trapping for the specified channel.

Purpose

Allows your program to set a trap for communications activity: as soon as information comes into the communications buffer, GW-BASIC branches to the subroutine at the specified line number.

Remarks

The following statements control the activation or deactivation of communications trapping:

COM(n) ON
Must be executed to activate the ON COM(n) statement. If you specify a non-zero line in ON COM (n), every time the program starts a new command, GW-BASIC checks to see if any characters have come into the specified channel. If there is at least one character, GW-BASIC performs a GOSUB to the specified line.

COM(n) OFF
When executed, no trapping takes place for the channel. If information arrives on the communications channel, it is not noticed by GW-BASIC.

COM(n) STOP
When executed, no trapping takes place for the channel. However, any characters received by the channel are saved in memory so that an immediate trap takes place when COM(n) ON is executed.

When a trap occurs, GW-BASIC automatically executes a COM(n) STOP so that recursive traps can never take place. The RETURN from the trap routine automatically executes COM(n) ON

unless an explicit COM(n) OFF has been executed within the trap routine.

Trapping never takes place unless GW-BASIC is executing a program.

When an error trap takes place (see ON ERROR), all trapping is automatically disabled. This means that communications events are disregarded by GW-BASIC.

Typically, the communications trap routine reads an entire message from the communications channel before returning. Using the communications trap for single character messages is not advisable because at high baud rates (speeds) the overhead of trapping and reading for each individual character may cause the interrupt buffer for communications to overflow.

Specifying a line number with the RETURN statement at the end of the trap routine is optional. Use it to go back to the program at a fixed line number. This action eliminates the GOSUB entry which the trap created. Use RETURN line with care! Any other GOSUB, WHILE, or FOR which was active at the time of the trap will remain active.

Example

```
30 ON COM(1) GOSUB 9900
40 COM(1) ON
 .
 .
 .
9900 REM trap routine to deal with incoming
     characters
 .
 .
 .
9990 RETURN
```

Line 30 means that if communications activity and trapping has been enabled for this communications channel at the time of the event, GW-BASIC will branch to the subroutine at line 9900. Line 40 enables communications trapping for this channel.

ON ERROR GOTO Statement

Syntax

ON ERROR GOTO line number

Purpose

To enable error handling and specify the first line of the error handling routine.

Remarks

Once error handling has been enabled, all errors detected, including direct mode errors, will cause a jump to the specified error handling routine. If "line number" does not exist, an "Undefined line" error results.

To disable error handling, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error handling routine causes GW-BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error handling routines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

If an error occurs during execution of an error handling routine, that error message is printed and execution terminates. Error trapping does not occur within the error handling routine.

Examples

```
10 ON ERROR GOTO 500
 .
 .
 .
500 REM Error trap handling routines
510 ON ERROR GOTO 0
520 IF ERR=24 THEN RESUME 'Device too
slow
 .
 .
 .
690 RESUME
```

Line 10 states that if any error occurs GW-BASIC is to branch to line 500. Line 520 asks that if a communications device did not respond in time

send GW-BASIC back to try to establish communications again. This is just one example of an error. Refer to *Appendix C* and decide which error possibilities you think you should provide for in each program you create.

You may want an audible signal to attract your attention if an error occurs. In this case, do the following:

```
10 ON ERROR GOTO 9900
 .
 .
 .
9900 REM Beep until a key is pressed
9910 BEEP
9920 IF INKEY$ = "" THEN 9910
9930 END
```

ON...GOSUB and ON...GOTO Statements

Syntax

ON <expression> GOTO <list of line numbers>

ON <expression> GOSUB <list of line numbers>

Purpose

To branch to one of several specified line numbers, depending on the value returned when "expression" is evaluated.

Remarks

The value of "expression" determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a non-integer, the fractional portion is rounded.)

In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine. Where you are using ON...GOSUB, you must ensure that GW-BASIC can find its way back after completing execution of the subroutine. This is achieved by the RETURN statement.

If the value of "expression" is zero or greater than the number of items in the list (but not greater than 255), GW-BASIC continues with the next executable command. If the value of expression is negative or greater than 255, an "Illegal function call" error occurs.

Examples

```
100 ON L-1 GOTO 150,300,320,390
```

If the value of the expression after any necessary rounding is 1, GW-BASIC branches to line 150. If this value is 2, GW-BASIC branches to line 300, and so on. If the value of L-1 is greater than 4, GW-BASIC simply goes on to the next line after 100.

The following program asks you to enter a number 1 to 4 according to the simple arithmetic operation to be performed. Line 110 tells GW-

BASIC to enter one of four subroutines, depending on your choice. After doing the arithmetic and telling you the answer, GW-BASIC RETURNs to the command following the ON...GOSUB statement. Press a key to go back to the beginning of the program. Entering zero for a number ends the program.

```
10 CLS:PRINT "Let me do your arithmetic
   homework":PRINT
20 INPUT "Enter the first of two numbers";N1
30 INPUT "Now the second number";N2
40 IF N1=0 OR N2=0 THEN END
50 CLS:PRINT "Press 1,2,3, or 4":PRINT
60 PRINT "1) Add the numbers"
70 PRINT "2) Subtract the second number
   from the first"
80 PRINT "3) Multiply the numbers"
90 PRINT "4) Divide the first number by the
   second"
100 INPUT CHOICE:PRINT
110 ON CHOICE GOSUB 150,170,190,210
120 IF INKEY$="" THEN 120
130 GOTO 10
140 REM ****** the subroutines
150 PRINT N1;" plus ";N2" equals ";N1+N2
160 RETURN
170 PRINT N1;" minus ";N2:" equals ";N1-N2
180 RETURN
190 PRINT N1;" times ";N2" equals ";N1*N2
200 RETURN
210 PRINT N1;" divided by ";N2;" equals
";N1/N2
220 RETURN
```

ON KEY(n) Statement

Syntax

KEY(n) GOSUB line number

n is a number in the range 1-20 referring to a programmable Function Key, a cursor movement key, or a user-defined trappable key combination:

1-10	The ten Function Keys
11	Cursor Up
12	Cursor Left
13	Cursor Right
14	Cursor Down
15-20	User-defined trappable key combination (see KEY).

"line number"
is the first line of the subroutine to which GW-BASIC will branch in the event of the key denoted by n being pressed. If "line number" is 0, the pressing of that key is no longer trapped.

Purpose

To instruct GW-BASIC to branch to a subroutine at a specified line, if a specified key or key combination is pressed.

Remarks

The following statements control the activation or deactivation of key trapping:

KEY(n) ON
Must be executed if ON KEY(n) is to have any effect. If you specify a non-zero line for the trap with ON KEY(n), every time the program starts a new command, GW-BASIC checks to see if the specified key was pressed. If you have pressed the key, GW-BASIC executes a GOSUB to the specified line.

KEY(n) OFF
If executed, no trapping takes place for the specified key. If you press the key, GW-BASIC does not react.

KEY(n) STOP

If executed, no trapping takes place for the specified key. However, if you press the specified key, an immediate trap takes place when KEY(n) ON is performed.

When a trap occurs, GW-BASIC automatically causes a KEY(n) STOP for the trapped key so that recursive traps can never take place. The RETURN from the trap routine automatically performs a KEY(n) ON unless an explicit KEY(n) OFF has been performed within the trap routine.

Trapping never takes place unless GW-BASIC is executing a program.

When an error trap takes place, all trapping is automatically disabled. This means that keyboard events are ignored.

No type of trapping is activated when GW-BASIC is in direct mode. In particular, Function Keys resume their programmed contents.

A key that causes a trap cannot be subsequently tested with the INPUT or INKEY$, so the trap routine for each key must be different if you want different functions.

Specifying a line number with the RETURN statement at the end of trap routine is optional. Use RETURN "line" to go back to the program at a fixed line number. This action eliminates the GOSUB entry which the trap created. Use RETURN "line" with care! Any other GOSUB, WHILE, or FOR which was active at the time of the trap will remain active.

Example

The following example prevents <Ctrl-Break> from having its usual effect:

```
10 KEY 20,CHR$(&H04)+CHR$(70)
20 ON KEY(20) GOSUB 1000
30 KEY(20) ON
40 PRINT "Try to get back to Ok"
50 GOTO 40
```

```
1000 PRINT "You tried to break out but you
     didn't succeed"
1010 RETURN
```

Before running this program, make sure that any other program in memory is saved on disk, as you will need a system restart <Ctrl-Alt-Del> to break out.

ON PEN Statement

Syntax

ON PEN GOSUB line number

Purpose

To set a line number where light pen trap handling starts.

Remarks

"line number"
is the first line of the subroutine to which GW-BASIC will branch if light pen activity is detected. If you specify a "line number" of 0, light pen trapping is disabled.

Assuming that "line number" was not specified as 0, and that a PEN ON statement has been executed, then every time a new command is about to be executed GW-BASIC checks to see if the light pen was activated. If so, GW-BASIC branches to the subroutine at "line number".

PEN OFF means that light pen trapping is cancelled. Furthermore, light pen activity is not recorded.

PEN STOP means that light pen trapping is cancelled, but pen activity is recorded by GW-BASIC. Therefore, an immediate trap takes place as soon as PEN ON is executed, if there has been interim light pen activity.

When the trap occurs, GW-BASIC automatically executes a PEN STOP so that recursive traps cannot take place. The RETURN from the trap handling subroutine automatically effects PEN ON, unless the subroutine contains an explicit PEN OFF statement.

Event trapping does not take place when GW-BASIC is not executing a program.

When an error trap takes place (see ON ERROR), all trapping is automatically disabled. This means that light pen events are ignored by GW-BASIC.

The PEN function is not affected when light pen activity causes a trap.

Specifying a line number with the RETURN statement at the end of the light pen handling subroutine is optional. This causes GW-BASIC to RETURN to the specified line number. This eliminates the GOSUB entry which the trap created, but it should be used with care! Any other GOSUB, WHILE or FOR which was active at the time of the trap will remain active.

Example

```
10 ON PEN GOSUB 1000
20 PEN ON
 .
 .
 .
1000 REM light pen handling
 .
 .
 .
1190 RETURN
```

This shows in lines 10 and 20 the commands necessary to create and enable a trap for light pen activity. Line 1190 shows the RETURN from the light pen handling routine.

ON PLAY(n) Statement

Syntax

ON PLAY(n) GOSUB line number

Purpose

To enable execution of other GW-BASIC commands while background music is playing.

Remarks

n is an integer expression in the range 1 to 32. A trap occurs when n notes in the background music buffer are left to play.

"line number" is the first line of the subroutine to which GW-BASIC branches when a trap occurs. A line number of 0 prevents music trapping.

Assuming that "line number" was not specified as 0, and that a PLAY ON statement has been executed, then every time a new command is about to be executed GW-BASIC checks to see if the background music buffer has gone from n to n-1. If so, GW-BASIC branches to the subroutine at "line number".

PLAY OFF has the effect that background music trapping no longer takes place. Furthermore, background music activity is not recorded.

PLAY STOP has the effect that background music trapping no longer takes place, but background music activity is recorded by GW-BASIC. Therefore, an immediate trap takes place as soon as PLAY ON is executed, if there has been interim background music activity.

When the trap occurs, GW-BASIC automatically executes a PLAY STOP, so that recursive traps cannot take place. The RETURN from the trap handling subroutine automatically effects PLAY ON, unless the subroutine contains an explicit PLAY OFF command.

Event trapping does not take place when GW-BASIC is not executing a program.

When an error trap takes place (see ON ERROR), all trapping is automatically disabled. This means that music events are ignored by GW-BASIC.

Specifying a line number with the RETURN statement at the end of the music event handling subroutine is optional. This causes GW-BASIC to RETURN to the specified line number. This eliminates the GOSUB entry which the trap created, but it should be used with care! Any other GOSUB, WHILE, or FOR which was active at the time of the trap will remain active.

Example

```
10 PLAY MB string
20 ON PLAY(5) GOSUB 1000
30 PLAY ON
40 GOSUB 1000
.
1000 REM Execute the following during back-
     ground music
.
.
.
1190 RETURN
```

The trap occurs when there are five notes remaining in the background music buffer.

Note

A music event can occur only when PLAY is in the background music, not the foreground music mode. A music event is not issued if the background music buffer is empty. This is the purpose of line 30 in the above example.

Do not choose too high a value for n. For example, ON PLAY(32) causes so many event traps that there is hardly time for the rest of the program.

ON STRIG(n) Statement

Syntax

ON STRIG(n) GOSUB line number

where (n) is a number denoting one of a maximum of four joystick buttons. Valid numbers are 0, 2, 4, and 6.

where "line number" is the number of the first line of a subroutine that is to be performed when the joystick button is pressed.

Purpose

To specify the first line number of a subroutine to be performed when a joystick button is pressed.

Remarks

A "line number" of zero disables the event trap.

The ON STRIG statement will only have effect if a STRIG ON has been executed (see STRIG Statement) to enable event trapping for that button. If event trapping is enabled, and if the "line number" in ON STRIG is not zero, GW-BASIC checks between commands to see if the joystick button has been pressed. If it has, a GOSUB is executed to the specified line.

If a STRIG OFF command has been executed for the specified button (see STRIG statement), the GOSUB is not executed and GW-BASIC does not record that a button was pressed.

If a STRIG STOP command has been executed for the specified button (see STRIG statement), the GOSUB is not executed, but will be performed as soon as the appropriate STRIG ON statement is executed.

When an event trap occurs (i.e., the GOSUB is performed), an automatic STRIG STOP is executed so that recursive traps cannot take place. The RETURN from the trapping subroutine will automatically perform a STRIG ON statement unless an explicit STRIG OFF was executed inside the subroutine.

Specifying a line number with the RETURN command at the end of the trap handling subrou-

tine is optional. It returns GW-BASIC to a specific line number from the trap handling subroutine. Use this type of return with care, because any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

Event trapping does not take place when GW-BASIC is not executing a program. Event trapping is automatically disabled when an error trap occurs. GW-BASIC then ignores joystick events.

The STRIG function is not affected by the occurrence of a joystick trap.

Example

```
10 ON STRIG(2) GOSUB 2200
20 STRIG(2) ON
 .
 .
 .
2200 REM deals with pressing of a joystick
button
 .
 .
 .
2290 RETURN
```

ON TIMER(n) Statement

Syntax ON TIMER(n) GOSUB line number

Purpose To set the line number where execution of a subroutine starts at the expiration of a specified period of time.

Remarks n is a numeric expression in the range 1 to 86,400 representing the state of the TIMER counter which will cause a timer event. This range of seconds means that a period of up to 24 hours can be set. A timer event occurs every n seconds following a TIMER ON command.

"line number" is the first line of the subroutine executed by GW-BASIC when a timer event occurs. A "line number" of 0 prevents timer trapping.

When n seconds have elapsed, the timer event occurs, whereupon GW-BASIC starts counting again from zero to n, and branches to the subroutine at line number.

TIMER OFF has the effect that timer trapping no longer takes place. Furthermore, timer activity is not recorded.

TIMER STOP means that timer trapping no longer takes place but timer activity is recorded by GW-BASIC. Therefore, an immediate trap takes place as soon as TIMER ON is executed, if there has been interim timer activity.

When the trap occurs, GW-BASIC automatically executes a TIMER STOP so that recursive traps cannot take place. The RETURN from the trap handling subroutine automatically effects TIMER ON, unless the subroutine contains an explicit TIMER OFF command.

Event trapping does not take place when GW-BASIC is not executing a program.

When an error trap takes place (see ON ERROR), all trapping is automatically disabled. This

means that timer events are disregarded by GW-BASIC.

Specifying a line number with the RETURN command at the end of the timer handling subroutine is optional. GW-BASIC then RETURNS to the specified line number. This eliminates the GOSUB entry which the trap created, but it should be used with care! Any other GOSUB, WHILE, or FOR which was active at the time of the trap will remain active.

Example

The following program issues a timer event once every minute. The contents of the screen are saved in an array variable, and the current time is displayed on the screen. After this, the old screen contents are restored. Lines 70 and 80 are included so you can write something on the screen (you can start typing as soon as the screen clears following RUN). When GW-BASIC BEEPs (line 110), stop writing for the moment. A little time is needed to store the screen in PIC$ (lines 130 to 160). Resume writing as soon as the screen is restored after the time has been displayed. If you wish, you can set the clock which governs the display of time in this program to the current time (see TIME$ function).

```
10 DEFINT A-Z
20 DIM PIC$(24,80)
30 SCREEN 0:WIDTH 80
40 ON TIMER(60) GOSUB 100
50 TIMER ON
60 CLS
70 K$=INKEY$:IF K$="" THEN 70
80 PRINT K$;
90 GOTO 70
100 REM ***** Deal with timer event
110 BEEP
120 ROW=CSRLIN:COL=POS(0)
130 FOR V=1 TO ROW
140 FOR H=1 TO 80
150 PIC$(V,H)=CHR$(SCREEN(V,H))
```

```
160 NEXT H:NEXT V
170 CLS:LOCATE 1,1
180 PRINT TIME$;
190 FOR DLY=1 TO 4000:NEXT DLY
200 CLS
210 FOR V=1 TO ROW
220 FOR H=1 TO 80
230 PRINT PIC$(V,H);
240 NEXT H:NEXT V
250 LOCATE ROW,COL
260 RETURN
```

OPEN Statement

Syntaxes

OPEN "mode",[#]file number, "filespec"[,record length]

OPEN "filespec"[FOR mode] AS [#]file number[LEN=record length]

In the first syntax form, "mode" is a string expression whose first character is one of the following:

O	Specifies sequential output mode.
I	Specifies sequential input mode.
R	Specifies random input/output mode.

In the second syntax form, mode is *not* a string expression (no quotes), but is as follows:

OUTPUT	Specifies sequential output mode
INPUT	Specifies sequential input mode
APPEND	Specifies sequential output mode and sets the file pointer at the end of file and the record number as the last record of the file. A PRINT# or WRITE# command will then extend (append) the file.

Note that in the first form, mode and filespec are enclosed in double quotation marks. In the second form, only filespec needs quotation marks. If mode is omitted in either form, GW-BASIC assumes random access. Sequential and random access files are discussed in Chapter 5, *Files and Devices.*

Whichever syntax form you use, "file number" is an integer expression whose value is between 1 and the maximum number set using the /F option

when GW-BASIC was loaded. The number is then associated with the file for as long as it is OPEN and is used to refer other disk I/O commands to the file, for example, FIELD for a random access file.

"filespec" is a string expression containing a name that conforms to DOS rules for disk filenames. "filespec" may include a pathname (see Chapter 5).

"record length" is an integer expression that, if included, sets the record length for random and sequential files. The default length is 128 bytes. Any value you set may not exceed that set by the /S option when loading GW-BASIC.

Purpose

To allow I/O to a file or device.

Remarks

A disk file must be opened before any disk I/O operation can be performed on that file. OPEN allocates a buffer for I/O to the file or device and determines the mode of access that will be used with the buffer.

The GW-BASIC commands which require a prior opening of the file are FIELD, GET, INPUT$, INPUT#, LINE INPUT#, PRINT#, PRINT#USING, PUT, and WRITE#.

A disk file may be either random or sequential access. The same applies to a printer. Other devices are normally accessed in sequential mode. APPEND is valid for disk files only.

Unless "'filespec" specifies another disk drive or device, GW-BASIC assumes the file to be opened is on the current disk drive.

The opening of a communications file is described separately (see OPEN COM).

A file can be opened for sequential input or random access under more than one file number at a time. However, once a file is open, it cannot be further opened for sequential output or appending.

If a printer (LPT1:, LPT2:, or LPT3:) is opened as a random file with a WIDTH of 255, the line feed normally accompanying a carriage return is suppressed. The print head can then make a second pass over the line and add effects such as underlining.

Examples

The following command is an example of opening a new sequential access file:

```
10 OPEN "WORDS" FOR OUTPUT AS #1
```

Using the other syntax form, the equivalent is

```
10 OPEN "O", #1, "WORDS"
```

Do not use these commands for a file already existing, as it would be erased and then opened as a new file. Instead use

```
20 OPEN "WORDS" FOR APPEND AS #1
```

The following example opens a random access file. If the file STORY.TIM already exists on drive C in the current directory, it is opened for further processing by your program (it is not erased). If the file does not already exist, it is opened as a new, empty file. A record length of 256 bytes is specified.

```
10 OPEN "C:STORY.TIM" AS #1 LEN=256
```

Using the other syntax form, the equivalent is

```
10 OPEN "R", #9, "C:STORY.TIM", 256
```

Here is an example of a pathname being specified:

```
30 OPEN "BITOPLVL\BOTLVL\INFO.OLD"
   FOR APPEND AS 2
```

Note

The following error situations can arise when opening a file:

"File not found"
A file opened for input does not exist. If a file opened for output, append, or random access does not exist, a new file is created.

"Illegal function call"
A value in the OPEN command is outside the permitted range.

A string variable may be given for "filespec", for example

```
10 INPUT "Which sequential file needs append-
   ing"; F$
20 OPEN F$ FOR APPEND AS 1
```

OPEN "COM" Statement

Syntax

OPEN "dev:[speed] [,parity] [,data] [,stop] [,RS] [,CS[n]] [,DS[n]] [,CD[n]] [,LF] [,PE]" AS [#]file number [LEN=length]

Purpose

Opens a communications file. Allocates a buffer for I/O in the same manner as OPEN for disk files. Supports RS-232 asynchronous communication with other computers and peripherals.

Remarks

dev
Specifies one of the following communications devices: COM1 or COM2.

speed
An integer constant which specifies the number of transmit or receive bits per second (baud rate). Valid speeds are: 75, 110, 150, 300, 600, 1200, 1800, 2400, 4800, and 9600. Default is 300 bps.

parity
A one-character constant which specifies the parity for transmit and receive, as follows:

S	SPACE: parity bit is always transmitted and received as a space (0 bit).
O	ODD: odd transmit and receive parity checking.
M	MARK: parity bit is always transmitted and received as a mark (1 bit).
E	EVEN: even transmit and receive parity checking.
N	NONE: no transmit or receive parity checking.

The default for parity is even (E).

data
An integer constant which indicates the number of transmit or receive data bits. Valid values are: 4, 5, 6, 7, and 8. The default is 7. If you specify 4, you may not specify N for parity, otherwise a "Bad File Name" error occurs. If you specify 8 bits, you must specify N (none) parity.

stop
An integer constant which indicates the number of stop bits. Valid values are 1 and 2. The default stop bits for 75 and 110 bps is 2. The default for all others is 1. If you specify 4 or 5 for data, a 2 entered for stop will mean 1 1/2 stop bits.

RS
Suppresses Request To Send (RTS) line signal. If you include RS, the RTS line is not turned on when an OPEN COM statement is run.

CSn
Controls Clear To Send (CTS) line signal. If you include CS without n, the system waits for the line signal for 1 second, without returning an error. If you enter CSn, n specifies the amount of time (in milliseconds) the system waits before returning a "Device Timeout" error. Using n=0 is the same as entering CS without n.

DSn
Controls Data Set Ready (DSR) line signal. If you include DS, the system waits for the line signal for 1 second, without returning an error. If you enter DSn, n specifies the amount of time (in milliseconds) the system waits before returning a "Device Timeout" error. Using n=0 is the same as entering DS without n.

CDn
Controls Carrier Detect (CD) line signal, also known as Received Line Signal Detect (RLSD). If you enter CDn, n specifies the amount of time (in

milliseconds) the system waits before returning a "Device Timeout" error. If you use n=0, or you omit the option, the line signal is not checked.

The maximum value for n which may be specified with CS, DS, or CD is 65535.

LF
Sends a line feed following each carriage return. Specify LF when using communication files to print to a serial line printer. Note that INPUT# and LINE INPUT#, when used to read from a communications file which was opened with the LF option, ignore the line feed and stop when they detect a carriage return.

PE
Enables parity checking. If not included, no parity checking takes place. Assuming you are using 7 data bits or less and that parity checking is enabled, a parity error will set the high order bit and cause a "Device I/O Error". (Framing and overrun errors always set the high order bit and cause "Device I/O Error", regardless of data length.)

file number
An integer expression which returns a valid file number. The number is associated with the file for as long as it is open and is used by other communications I/O statements to refer to the file. A communications device may be open under only one file number at a time.

length
The maximum number of bytes which can be read from the communications buffer when using GET or PUT. The default is 128 bytes.

Any syntactical errors within the part of the command enclosed in double quotation marks result in a "Bad File Name" error. An indication of which parameter is in error is not given.

See Chapter 5, *Files and Devices* for information on communications I/O. Success with communications depends to a great extent on getting the hardware connections right. Therefore, you should refer to the hardware documentation for the communications device you are using.

Examples

In the following example, file number 1 is opened for communication with defaults: 300 bps, even parity, and 7 data bits with 1 stop bit. However, parity checking will not actually take place since PE is not included.

```
10 OPEN "COM1:" AS #1
```

The following command opens file number 2 for communication at 2400 bps. Defaults are: even parity, 7 data bits, and 1 stop bit. Again, PE is not included.

```
10 OPEN "COM1:2400" AS #2
```

The following command opens file number 1 for asynchronous I/O at 1200 bps. No parity is produced or checked. 8-bit bytes will be sent and received. The stop bit defaults to 1.

```
10 OPEN "COM2:1200,N,8" AS #1
```

The next example opens COM1 for 4800 bits per second, defaulting to even parity and 7 data bits. RTS is to be transmitted, CTS will not be checked, and a "Device Timeout" error will arise if DSR is not detected within three seconds. Parity checking is enabled. Commas are required for the missing (defaulting) parameters: parity, data, and stop. (If you omit one or more of the parameters RS, CS, DS, CD, LF, and PE, replacing commas are not required).

```
10 OPEN "COM1:4800,,,,CS,DS3000,PE" AS #1
```

Note

Error trapping a "Device Timeout" error is useful if you want to give the communications device

more time to respond (see ON ERROR). However, you probably do not want to wait indefinitely for the device. In this case, it is not enough for the trap routine to contain only a RESUME command. Your program should include a counter to limit the number of tries to be made. The trap routine can adjust this counter each time GW-BASIC passes through it. Example:

```
10  ATTEMPT% =5
20  ON ERROR GOTO 500
30  OPEN "COM1:,,,,CS,DS,CD5000" AS #1
 .
 .
 .
500 IF ERR<>24 THEN GOTO 600
510 ATTEMPT% =ATTEMPT%-1
520 IF ATTEMPT% =0 THEN GOTO 540
530 RESUME
540 BEEP:PRINT"Check device and start pro-
    gram again"
550 STOP
 .
 .
 .
600 REM other error handling routines
```

OPTION BASE Statement

Syntax

OPTION BASE n

where n is 1 or 0

Purpose

To declare the minimum value for array subscripts.

Remarks

The default base is 0. If the statement

```
OPTION BASE 1
```

is executed, the lowest value an array subscript may have is 1.

OPTION BASE must be encountered by GW-BASIC before an array is defined or used.

OUT Statement

Syntax OUT I,J

where I is the port number. It must be an integer expression in the range 0 to 65535.

J is the data to be transmitted. It must be an integer expression in the range 0 to 255.

Purpose To send a byte to a machine output port.

Example 100 OUT 128,255

transmits the value 255 via port 128.

Note OUT has the same effect as the assembly language OUT instruction.

PAINT Statement

Syntax

PAINT (x,y[effect] [,[outline] [,background]]

x,y
The coordinates of any point within the area to be filled in. The coordinates may be given in absolute or relative (using STEP) form.

effect
If a numeric expression, it can be either the background color (0), or one of the colors 1 to 3 from the current palette (see COLOR). This applies to low and high resolution color graphics, where the default is 3. In medium and high resolution black-and-white graphics, a numeric value of 0 denotes black; the default value of 1 denotes white. Alternatively, 'effects' may be a string expression, in which case 'tiling' is performed. This is explained later in this description of PAINT.

outline
The color belonging to the outline of the area to be filled. Possible colors are as given above under "effect". If the "outline" color is incorrect, painting goes beyond the area in which x,y is enclosed. In medium and high resolution black-and-white graphics "outline" need not be stated as it defaults to the same value as "effect".

Purpose

In low and high resolution color graphics, to fill an area with a specified color. In medium and high resolution black-and-white graphics, this means filling in an area enclosed by white with white.

Remarks

The starting point x,y must be inside and completely enclosed by the figure to be painted. If the specified point already has the "outline" color, no painting takes place, therefore move x,y just inside the figure after DRAWing it and before PAINTing it.

When PAINTing a figure with lots of corners, GW-BASIC needs a greater than usual amount of stack space. You can give GW-BASIC more stack space, using the CLEAR command.

Tiling

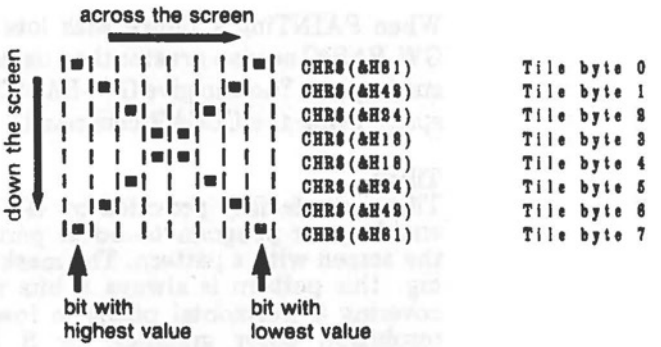
Tiling is a facility provided by GW-BASIC to enable your program to cover part or all of the screen with a pattern. The mask determining this pattern is always 8 bits wide, thus covering 4 horizontal pixels in low and high resolution color graphics, or 8 horizontal pixels in medium and high resolution black-and-white graphics.

GW-BASIC recognizes that you wish the PAINT command to do tiling when "effect" is a string expression. "effect" then consists of up to 64 2-character hexadecimal values, each of which represents a pattern for 4 or 8 horizontal pixels. Thus, you can define a pattern of up to 4-horizontal by 64 vertical pixels in low and high resolution color graphics, or 8 by 64 pixels in medium and high resolution black-and-white graphics. This pattern is repeated uniformly over the entire screen or within a figure, if that figure encloses the point x,y.

In medium and high resolution black-and-white graphics, the following string value for "effect" produces x's over the whole screen by means of an 8 by 8 pixel pattern.

```
10 SCREEN 2:CLS
20 PAINT (320,100),CHR$(&H81)+
  CHR$(&H42)+CHR$(&H24)+
    CHR$(&H18)+CHR$(&H18)+
    CHR$(&H24)+CHR$(&H42)+CHR$(&H81)
```

To understand how these hexadecimal values create an x, some knowledge of binary and hexadecimal values is useful. Perhaps the following diagram can help:



The first CHR$ value following the x,y coordinates in line 20 is tile byte 0, the last is tile byte 7.

GW-BASIC does not necessarily start with tile byte 0 at the specified x,y coordinates. Instead, the number of the tile byte painted at x,y is equal to the remainder of dividing the y coordinate by the number of bytes in the pattern. In the example given here, this means that pattern painting starts in vertical screen line 100 with the byte number 4 (remainder of dividing 100 by 8).

In low and high resolution color graphics, one tile byte represents only four screen points in the horizontal direction. This is because two bits are required for each point, so that GW-BASIC knows what color that point is. Therefore, the eight bits of each tile byte must be regarded as four pairs. The binary value of a pair can be 01 for color 1 of the current palette, 10 for color 2, or 11 for color 3. This gives the color for a single screen point. Therefore, you will most probably be using the following three hexadecimal values for the CHR functions which go to make up the "effect" string:

&H55	produces four consecutive horizontal points in color 1 (palette 0: green, palette 1: cyan)
&HAA	produces four consecutive horizontal points in color 2 (palette 0: red, palette 2: magenta)
&HFF	produces four consecutive hori-

zontal points in color 3 (palette 0: brown, palette 1: white)

You may have occasion to tile paint over an area already painted that is the same color as two consecutive bytes in the pattern. Normally, PAINT terminates when it encounters two consecutive bytes of the same color as the point being set, because the point is then surrounded by the same color. "background" specifies a background tile color enabling you to skip up to two consecutive bytes in the tile string. By using "background" your program can, for example, draw alternating blue and red lines on a red background with a minimum of GW-BASIC commands.

If more than two consecutive lines in the pattern match "background", GW-BASIC recognizes an "Illegal function call" error.

Examples

The first example draws a circle in red on black screen background. Since the last point referenced is then the center of the circle, PAINT is already positioned inside the circle and need not be moved: STEP (0,0). The circle is then filled in with green (color 1 of palette 0). The "outline" color must be set to 2, the color the circle was drawn with, otherwise painting will continue outside the circle.

```
10 SCREEN 1:COLOR 0,0
20 CLS
30 CIRCLE (160,100),30,2
40 PAINT STEP (0,0),1,2
```

The next example uses a string expression for "effect" to paint cyan and magenta horizontal stripes in the box drawn in line 40. STEP (5,-5) moves the position for PAINT from the bottom left corner of the box to just inside it. As the box was drawn in white, "outline" must be set to white (3), otherwise painting would go beyond the confines of the box.

```
10 SCREEN 1:COLOR 0,1
```

```
20 CLS
30 DRAW "c3"
40 DRAW "u50 r50 d50 150"
50 PAINT STEP (5,-5),
   CHR$(&H55)+CHR$(&H55)+CHR$
   (&HAA)+CHR$(&HAA),3
```

The following example shows the use of "background" to paint stripes of cyan and magenta on a magenta background. Lines 30 and 40 draw a box and fill it with magenta. Line 60 paints the stripes using the skipping effect provided by "background". Line 70 contains in a programmer's REMark the same command, but without this skipping. Try making 60 the REM (non-executed) line and let GW-BASIC execute line 70 instead. You will see that painting stops as soon as the first magenta line is to be drawn. Note that the tiling pattern in this example was taken from a string variable, PATT$. The delay created by line 50 gives you time to observe what is happening on the screen.

```
10 SCREEN 1:COLOR 0,1
20 PATT$=CHR$(&H55)+CHR$(&H55)+
   CHR$(&HAA)+CHR$(&HAA)
30 DRAW "u90 r90 d90 190"
40 PAINT STEP (5,-5),2,3
50 FOR DLY% =1 TO 1000:NEXT DLY%
60 PAINT STEP (5,-5),PATT$,3,CHR$(&HAA)
70 REM PAINT STEP (5,-5),PATT$,3
```

PEEK Function

Syntax

PEEK(I)

Purpose

To return the byte read from the indicated memory location (I).

Remarks

The returned value is an integer in the range 0 to 255. I must be in the range 0 to 65535. I is the offset from the current segment, which was defined by the last DEF SEG command.

PEEK is the complementary function of the POKE statement.

Example

A%=PEEK(&H5A00)

assigns the value of the byte at the hexadecimal address 5A00 to the integer variable A%.

You do not have to use a hexadecimal value for I. The decimal equivalent of this command is

A%=PEEK(23040)

Note

To assign a value to a specific memory location, use the POKE command.

PEN Statement

Syntax

PEN ON
PEN OFF
PEN STOP

Purpose

To enable and disable light pen reading.

Remarks

The light pen is initially off. PEN ON enables data from the light pen to be read by means of the PEN function, and enables the ON PEN event trap.

PEN OFF disables light pen reading and the ON PEN event trap. Issue this command as soon as light pen reading is no longer required, as this results in better GW-BASIC execution times. PEN events are then not recorded by GW-BASIC.

PEN STOP disables the ON PEN event trap, but GW-BASIC records any light pen activity. Consequently, a trap is activated as soon as PEN ON is executed, if there has been interim light pen activity.

PEN Function

Syntax

PEN(n)

where n is a numeric expression in the range 0 to 9, selecting a particular light pen value to be read.

Remarks

The significance of the values 0 to 9:

0 This is a flag indicating whether the pen switch has been set to down since the PEN function was last called upon to give information.

1 Returns the x coordinate of the position in which the pen was last activated. The number thus read can be in the range 0 to 319 in low resolution, 0 to 639 in medium and high resolution.

2 Returns the y coordinate of the position in which the light pen was last read (low and medium resolution 0 to 199, high resolution 0 to 399).

3 Returns -1 if the pen switch is down, 0 if it is up.

4 Returns the last known valid x coordinate (low resolution 0 to 319, medium and high resolution 0 to 639).

5 Returns the last known valid y coordinate (low and medium resolution 0 to 199, high resolution 0 to 399).

6 Returns a value in the range 1 to 24 for the line position where the light pen was last activated.

7 Returns the character column position where the light pen was last activated. This is a value in the range 1 to 80 or 1 to 40, depending on the current WIDTH setting.

8 Returns the last known valid character row in the range 1 to 24.

9 Returns the last known valid character column position in the range 1 to 80 or 1 to 40, depending on the current WIDTH setting.

Example

```
10 PEN ON
20 PENLIN% = PEN(6)
```

enables light pen reading (and event trapping) and puts the number of the screen line in which the light pen was last activated into the variable PENLIN%.

Note

Attempting to read the light pen while PEN OFF is in force results in an "Illegal function call" error.

PLAY Statement

Syntax

Format 1
PLAY <string expression>
or
Format 2
PLAY ON
PLAY OFF
PLAY STOP

Purpose

Format 1: To create a tune by defining its characteristics in the string expression. Format 2: To enable or disable the ON PLAY event trapping.

Remarks

FORMAT 1

The expression may consist of any of the following commands, which you may specify in any order unless stated otherwise in the description.

A to G [#,+,-]
Plays the specified note. # or + after a note specifies a sharp; - after a note specifies a flat. In either case, the note must be an actual piano key.

L <n> - Length
Sets the length (duration) of the note (or notes), where n may be from 1 to 64. As examples, L1 specifies a whole note, L2 specifies a half note...and L64 specifies a sixty-fourth note. You may specify the length before a group of notes or after a single note to change the length of that note only. For example, A16 is the same definition as L16A.

MB - Music Background
Sets music to run in the background. A buffer of up to 32 notes plays in the background while GW-BASIC is executing other commands.

MF - Music Foreground
Sets music to run in the foreground. Each subsequent note or sound is not started until the previous note or sound is finished. MF is the initial default value.

MN - Music Normal
Plays each note 7/8ths. of the time specified in L (length). This is the default setting.

ML - Music Legato
Plays each note the full length (as specified in L).

MS - Music Staccato
Plays each note 3/4ths. of the time specified in L (length).

N <n> - Note
Plays the note specified by <n>. <n> may range from 0 to 84 thus covering the semi-tones of 7 octaves, starting two octaves below middle C. <n> may equal 0 to specify a pause. Using this command provides an alternative way to specify the note other than by name (A to G) and octave.

O <n> - Octave
Sets the octave, where <n> may range from 0 to 6. Each of the seven octaves start with C. Middle C is at the beginning of octave 3; the default octave is octave 4.

P <n> - Pause
Sets the length of the pause, where <n> may range from 1 to 64. The <n> value is the same as the <n> value in the L(ength) command; for example, P1 causes a pause the length of a whole note, P2 causes a pause the length of a half note, and so on.

T <n> - Tempo
Sets the number of quarter notes <n> that can be played in a minute. <n> may range from 32 to 255; the default value is 120.

. - dot or period
Used after a note, plays the note as a dotted note; that is, its length is multiplied by 3/2. More than one dot may be used after the note, in which case its length is adjusted accordingly. As examples, A.. plays 9/4 as long as L specifies, A... plays 27/8 as long, etc. Dots may also be used after a pause (P) to scale the pause length in the same way.

X variable;
Executes the specified string containing valid PLAY commands.

> note
Raises the scale by one octave and plays the note A to G specified in the new octave. If the octave is already six, the note is played in octave 6.

< note
Lowers the scale by one octave and plays the note A to G specified. If the octave is already 0, the note is played in octave 0.

In all commands, the <n> value can be a constant or a numeric variable preceded by an equal sign, followed by a semicolon. The semicolon (;) is required when you use a variable in this way, and when you use the X command; otherwise, a semicolon is optional between commands, except it is not allowed after MF, MB, MN, ML, or MS. Blanks in a string are ignored.

You can also specify variables in the form VARPTR$ (<variable>), instead of =<variable>;. This method is useful in programs that will later be compiled.

Examples

```
10 MARY$="GFE-FGGG"
20 PLAY "MB T100 O3 L8;XMARY$;P8 FFF4"
30 PLAY "GB-B-4;XMARY$;GFFGFE-."
```

The following example shows the use of the ON PLAY event trap to produce a continuous background tune while screen activity is in progress. The screen activity simply builds up a random

pattern of character blanks as you press the space bar (lines 40 to 60).

Line 30 determines that whenever the number of background music notes left to play goes from 1 to 0, GW-BASIC will branch to the subroutine starting at line 160. The subroutine tells GW-BASIC to play the background tune (again). Line 40 enables the trap. Line 50 starts the PLAYing of the background tune. Without this command, the condition for music trapping given in line 30 would never be fulfilled, and there would be no music.

```
10 SCREEN 0:WIDTH 80
20 KEY OFF
30 ON PLAY (1) GOSUB 160
40 PLAY ON
50 GOSUB 160
60 CLS
70 IF INKEY$=" " THEN GOTO 70
80 COLOR INT(32*RND)
90 LI%=INT(25*RND+1)
100 PO%=INT(80*RND+1)
110 IF LI%>25 THEN LI%=24
120 IF PO%>80 THEN PO%=80
130 LOCATE LI%,PO%
140 PRINT CHR$(219) ;
150 GOTO 70
160 REM *********
170 PLAY "o2 mb t140 f aa c aa"
180 RETURN
```

Remarks

FORMAT 2

PLAY OFF has the effect that background music trapping by an ON PLAY statement no longer takes place. Furthermore, background music activity is not recorded.

PLAY STOP has the effect that background music trapping by an ON PLAY statement no longer takes place, but background music activity

is recorded by GW-BASIC. Therefore, an immediate ON PLAY trap takes place as soon as PLAY ON is executed, if there has been interim background music activity.

When the ON PLAY trap occurs, GW-BASIC automatically executes a PLAY STOP, so that recursive traps cannot take place. The RETURN from the trap handling subroutine automatically effects PLAY ON, unless the subroutine contains an explicit PLAY OFF command.

Event trapping does not take place when GW-BASIC is not executing a program.

When an error trap takes place (see ON ERROR), all trapping is automatically disabled. This means that music events are ignored by GW-BASIC.

Example

```
10 PLAY MB string
20 ON PLAY(5) GOSUB 1000
30 PLAY ON
40 GOSUB 1000
 .
1000 REM Execute the following during background music
 .
 .
 .
1190 RETURN
```

The trap occurs when there are five notes remaining in the background music buffer.

Note

A music event can occur only when PLAY <string> is in the background music (MB), not the foreground music (MF) mode. A music event is not issued if the background music buffer is empty. This is the purpose of line 30 in the above example.

PMAP Function

Syntax

PMAP (coord,transaction)

Purpose

To translate world coordinates (set by WINDOW) into physical coordinates (see VIEW), and vice versa.

Remarks

"translation" can be one of four values:

0 returns a physical x coordinate for a world x coordinate specified in "coord".

1 returns a physical y coordinate for the world coordinate specified in "coord".

2 returns a world x coordinate for the physical x coordinate specified in "coord".

3 returns a world y coordinate for the physical y coordinate specified in "coord".

Example

The physical coordinates of the usable screen are 0,0-199,319 in low resolution (medium resolution: 0,0-199,639, high resolution: 0,0-399,639), as long as you do not alter this with VIEW.

If you first use WINDOW to determine a Cartesian coordinate scheme in low resolution graphics

```
10 SCREEN 1: WINDOW (-1,-1)-(1,1)
```

and then execute

```
20 PRINT  PMAP(-1,0),PMAP(1,0),PMAP(-1,1),
   PMAP(1,1)
30 PRINT PMAP(0,1),PMAP(0,0)
```

GW-BASIC displays

```
0     319   199   0
100   160
```

Refer to the WINDOW description for full details of how this command sets the coordinate scheme.

POINT Function

Syntax

POINT(x,y)
POINT (coord)

Purpose

To read the color or a coordinate of the currently addressed point on the screen.

Remarks

x and y must specify an absolute screen position. The value returned in low and high resolution color graphics is 0 for background, or a value 1, 2, or 3 for the corresponding color of the current palette. In medium and high resolution black-and-white graphics, the possible values are 0 and 1.

"coord" is a value 0 to 3;

0 returns the physical x coordinate

1 returns the physical y coordinate

2 if WINDOW is currently active, the coordinate returned is the world x coordinate. Otherwise, the physical x coordinate is returned.

3 is the same as 2, except that a y coordinate is returned.

Examples

The following program creates a random pattern of red dots in a 50 by 50 area in the top left corner of the screen (lines 20 to 40). Then, each of the 2500 graphic points is read using the POINT function. If a point is red, it is changed to black and vice versa.

```
10 SCREEN 1:COLOR 0,0:CLS
20 FOR CHANCE% =1 to 1000
30 PSET (RND*50,RND*50),2
40 NEXT CHANCE%
50 FOR X% = 0 TO 50
60 FOR Y% = 0 TO 50
70 PSET(X%,Y%),ABS(POINT(X%,Y%)-2)
80 NEXT Y%:NEXT X%
```

The program given below asks you to enter an x and a y coordinate. This point is then illuminated red using the standard coordinate system of medium resolution graphics (origin top left, 320 points in the horizontal direction, 200 in vertical direction). Line 40 then sets a Cartesian coordinate system with the origin as near as possible to the center of the screen. The center point is illuminated brown, and the x and y value you entered are used to plot a point in green in accordance with the new (Cartesian) coordinate system. The top of the screen then displays on the left the coordinates of the green point in terms of the physical screen, and on the right the coordinates of the same point in terms of the WINDOW definition.

```
10 INPUT "X and Y";X%,Y%
20 SCREEN 1:COLOR 0,0:CLS
30 PSET (X%,Y%),2
40 WINDOW (-160,-100)-(159,99)
50 PSET (0,0),3
60 PSET (X%,Y%),1
70 PRINT POINT(0);POINT(1);" -->WINDOW
   -->";POINT(2);POINT(3)
80 IF INKEY$="" THEN 80
90 SCREEN 0:WIDTH 80
100 LIST
```

POKE Statement

Syntax

POKE I,J

where I and J are integer expressions.

Purpose

To write a byte into a memory location.

Remarks

I and J are integer expressions. The expression I represents the address of the memory location and J is the data byte. I must be in the range 0 to 65535. I is the offset from the current segment, which was set by the last DEF SEG statement.

The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read.

POKEing and PEEKing is a fast way of storing and reading data, or passing information to machine language routines. You can even write machine language routines using POKE (see Chapter 6).

Note

POKE gives you immense power over your computer, which means that you should use it with care. It is advisable to confine POKEing to an area of memory which you have especially set aside for your own use.

POS Function

Syntax	POS(I)
Purpose	To return the current horizontal (column) position of the cursor
Remarks	The leftmost position is 1. I is a dummy argument so you can use a numeric constant in its place. To return the current line position of the cursor, use the CSRLIN function.
Example	IF POS(X)>30 THEN BEEP causes GW-BASIC to BEEP for as long as the cursor is to the right of the 30th screen column.
Note	The screen can consist of 40 or 80 columns, depending on the setting of WIDTH.

PRESET and PSET Statements

Syntax

PRESET(x,y)[,color]
PSET(x,y)[,color]

where x and y specify a point on the screen, and "color" specifies the background color (0) or a color 1 to 3 from the current palette (medium resolution). In high resolution, the values 0 and 2 denote black, 1 and 3 denote white.

Purpose

To illuminate a point on the screen in a specified color and/or to determine the point which subsequent graphic drawing is to regard as the last point referenced.

The default "color" for medium resolution graphics is 3, for high resolution graphics it is 1.

The only difference between PSET and PRESET is that if no "color" is specified for PRESET, the background color (0) is used, thus plotting an invisible point.

You may specify offset coordinates using STEP, that is, a point relative to the last point referenced.

Example

The following example sends a dash, defined as six horizontal pixels, from left to right across the center of the screen. The PRESET command in line 70 means that a trail is not left.

```
10 SCREEN 2:CLS
20 FOR X% =0 TO 5
30 PSET (X%,100)
40 NEXT X%
50 FOR X% =6 TO 639
60 PSET (X%,100)
70 PRESET (X%-6,100)
80 NEXT X%
```

See also the Exercises at the end of Chapter 3, *Screen Display.*

Here is another example that draws a line across the screen, then erases it.

```
SCREEN 2:CLS
20 FOR X% =0 TO 639
30 PSET (X%,100)
40 NEXT X%
50 FOR X% =639 TO 0 STEP-1
60 PRESET (X%,100)
70 NEXT
```

See also the Exercises at the end of Chapter 3, *Screen Display.*

Note

GW-BASIC does not recognize an error if you try to address points outside the range of coordinates available for plotting.

Specifying a greater value than 3 for "color" causes an "Illegal function call" error.

PRINT Statement

Syntax

PRINT [list of expressions]

Purpose

To display data on the screen.

Remarks

If "list of expressions" is omitted, a blank line is printed. If "list of expressions" is included, the values of the expressions are displayed on the screen. The expressions in the list may be numeric and/or string expressions. (Strings must be enclosed in quotation marks.)

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. GW-BASIC divides the line into zones of 14 spaces each. In the list of expressions, a comma causes the next value to be displayed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value. One or more spaces between expressions have the same effect as a semicolon.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins displaying on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage return is carried out at the end of the line. If the line to be displayed is longer than the screen WIDTH, GW-BASIC goes to the next physical line and continues with the rest of the line.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single precision numbers that can be represented with 6 or fewer digits in the unscaled (fixed point or integer) format no less accurately than they can be represented in the scaled (floating point) format are output using the unscaled format. For

example, 1E-7 is output as .0000001 but 1E-8 is output as 1E-08. Double precision numbers that can be represented with 16 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format are output using the unscaled format. For example, 1D-16 is output as .0000000000000001 and 1D-16 is output as 1D-17.

A question mark may be used in place of the word PRINT in a PRINT statement. GW-BASIC automatically replaces it with the word PRINT at the next LISTing.

Examples

```
10 X=5
20 PRINT X+5,X-5,X*(-5),X^5
30 END
will yield
 10          0          -25          3125
```

In the above example, the commas in the PRINT statement cause each value to be displayed at the beginning of the next print zone.

In the next example, the semicolon at the end of line 20 causes the PRINT items of lines 20 and 30 to be displayed on the same line. Line 40 causes a blank line to be printed before the next prompt.

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2 "AND";
30 PRINT X "CUBED IS" X^3
40 PRINT
50 GOTO 10
will yield
? 9
 9 SQUARED IS 81 AND 9 CUBED IS 729

? 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261

?
```

In the following example, the semicolons in the PRINT statement cause each value to be displayed immediately after the preceding value. (Don't forget, a number is always followed by a space.) In line 40, a question mark is used instead of the word PRINT.

```
10 FOR X=1 TO 5
20 J=J+5
30 K=K+10
40 ?J;K;
50 NEXT X
will yield
 5  10  10  20  15  30  20  40  25  50
```

Note

If the end of the last item of a PRINT statement is displayed in the rightmost screen position (column 40 or 80 according to WIDTH) and if that PRINT statement is not concluded by a semicolon, a blank line will be apparent between the line just displayed and the next item displayed.

LPRINT displays data on a printer instead of on the screen.

PRINT USING Statement

Syntax

PRINT USING string exp;list of expressions

Purpose

To print strings or numbers using a specified format.

Remarks/
Examples

"list of expressions" is comprised of the string expressions or numeric expressions that are to be printed, separated by semicolons or commas.

"string exp" is a string constant or variable composed of special formatting characters. These formatting characters (see below) determine the field and the format of the printed strings or numbers.

String Fields

When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field:

"!"
Specifies that only the first character of each item in the given "list of expressions" is to be printed.

"\n spaces\"
Specifies that 2 + n characters from each item of the "list of expressions" are to be printed. If the backslashes are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string will be left-justified in the field and padded with spaces on the right.

Example:

```
10 A$="LOOK":B$="OUT"
```

```
30 PRINT USING "!";A$;B$
40 PRINT USING "\     \";A$;B$
50 PRINT USING "\     \";A$;B$;"!!"
will yield
LO
LOOKOUT
LOOK OUT !!
```

"&"
Specifies a variable length string field. When the field is specified with "&", the string is output without modification.

Example:

```
10 A$="LOOK":B$="OUT"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
will yield
LOUT
```

Numeric Fields

When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

(number sign)
A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.

. (decimal point)
A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0, if necessary). Numbers are rounded as necessary.

```
PRINT USING "##.##";.78
0.78
```

```
PRINT USING "###.##";987.654
987.65

PRINT USING "##.##    ";10.2,5.3,66.789,.234
10.20    5.30   66.79    0.23
```

In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.

+ (plus sign)
A plus sign at the beginning or end of the format string will cause the sign of the number (plus or minus) to be printed before or after the number.

— (minus sign)
A minus sign at the end of the format field will cause negative numbers to be printed with a trailing minus sign.

```
PRINT USING "+##.##    ";-68.95,2.4,55.6,-.9
-68.95    +2.40    +55.60    -0.90

PRINT USING "##.##- ";-68.95,22.449,-7.01
68.95-    22.45     7.01-
```

** (double asterisk)
A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The ** also specifies positions for two more digits.

```
PRINT USING "**#.#    ";12.39,-0.9,765.1
*12.4    *-0.9    765.1
```

$$ (double dollar sign)
A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.

```
PRINT USING "$$###.##";456.78
$456.78
```

**$

The **$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces will be asterisk-filled and a dollar sign will be printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.

```
PRINT USING "**$##.##";2.34
***$2.34
```

, (comma)

A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with exponential (^^^^) format.

```
PRINT USING "####,.##";1234.5
1,234.50

PRINT USING "####.##,";1234.5
1234.50,
```

^^^^

Four carets may be placed after the digit position characters to specify exponential format. The four carets allow space for E+xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

```
PRINT USING "##.##^^^^";234.56
2.35E+02
```

```
PRINT USING ".####^^^^-";-888888
.8889E+06-

PRINT USING "+.##^^^^";123
+.12E+03
```

An underscore in the format string causes the next character to be output as a literal character.

```
PRINT USING "  !##.##  !";12.34
!12.34!
```

An underscore at the beginning of the format string may be omitted.

```
PRINT USING "Your file has been assigned
  ##";2

Your file has been assigned #2
```

The literal character itself may be an underscore if you place " " in the format string.

% (percent sign)
If the number to be printed is larger than the specified numeric field, a percent sign is printed in front of the number. If rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number.

```
PRINT USING "##.##";111.22
%111.22

PRINT USING ".##";.999
%1.00
```

If the number of digits specified exceeds 24, an "Illegal function call" error will result.

PRINT# and PRINT# USING Statements

Syntax

PRINT#file number,[USING string exp;] list of expressions

Purpose

To write data to a sequential file.

Remarks/
Examples

"file number" is the number used when the file was opened for output. "string exp" consists of formatting characters as described in PRINT USING. The expressions in "list of expressions" are the numeric and/or string expressions that will be written to the file.

PRINT# does not compress data. An image of the data is written to the file, just as it would be displayed on the screen with PRINT. For this reason, care should be taken to delimit the data, so that later it will be input correctly from the file.

In the "list of expressions", numeric expressions should be delimited by semicolons. For example:

PRINT#1,A;B;C;X;Y;Z

(If commas are used as delimiters, the extra blanks that are inserted between print fields will also be written to the file.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly in the file, use explicit delimiters in the list of expressions.

For example, let A$="CAMERA" and B$="93604-1". The command

PRINT#1,A$;B$

would write CAMERA93604-1 to the file. Because there are no delimiters, this could not be input as

two separate strings. To correct the problem, insert explicit delimiters into the PRINT# command as follows:

```
PRINT#1,A$;",";B$
```

The image written to the file is

```
CAMERA,93604-1
```

which can be read back from the file into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or linefeeds, write them to the file surrounded by explicit quotation marks, CHR$(34).

For example, let A$="CAMERA, AUTOMATIC" and B$=" 93604-1". The command

```
PRINT#1,A$;B$
```

would write the following image to the file:

```
CAMERA, AUTOMATIC 93604-1
```

And the statement

```
INPUT#1,A$,B$
```

would input "CAMERA" to A$ and "AUTOMATIC93604-1" to B$. To separate these strings properly in the file, write double quotation marks to the file image using CHR$(34). The command

```
PRINT#1,CHR$(34);A$;CHR$(34);CHR$(34);B$
   ;CHR$(34)
```

writes the following image to the file:

"CAMERA, AUTOMATIC" 93604-1"

And the statement

INPUT#1,A$,B$

would input "CAMERA, AUTOMATIC" to A$ and " 93604-1" to B$.

PRINT# may also be used with the USING option to control the format of the file. For example:

PRINT#1,USING"$$###.##,";J;K;L

Note

See also WRITE# which does not require explicit delimiters.

PUT (Files) Statement

Syntax

PUT [#]file number [,record number]

Purpose

To write a record from a random buffer to a random access file.

Remarks

"file number" is the number under which the file was opened. If "record number" is omitted, the record will assume the next available record number (after the last PUT). The largest possible record number is 32,767. The smallest record number is 1.

LSET, RSET, PRINT#, PRINT# USING, and WRITE# may be used to put characters in the random file buffer before executing a PUT statement. In the case of WRITE#, GW-BASIC pads the buffer with spaces up to the carriage return.

Any attempt to read or write past the end of the buffer causes a "Field overflow" error.

You can also use PUT for a communications file. Then "record number" is not a record number, but the number of bytes for output. Take care that this number is not greater than that set by the LEN option in OPEN COM.

Note

A single PUT command does not necessarily mean that the disk drive of your computer is activated immediately. This is because GW-BASIC and the operating system try to collect a number of records before writing them to disk.

PUT (Graphics) Statement

Syntax

PUT (x,y),array[,image]

Purpose

To set the colors of points on the screen using data stored in an array.

Remarks

"x,y" are the coordinates of the top left position of the screen area to be affected.

"array" is the name of the numeric array containing the data. The GET (Graphics) description explains how graphic data is stored in such an array.

"image", if included, offers a choice of effects.

PSET puts the graphic data on the screen just as it was when GET was used to store it in the array.

PRESET has the same effect as PSET, except that a negative image is produced: a value 0 in the array sets the screen point to color 3 of the current palette, and vice versa; 1 sets the screen point to 2, and vice versa.

AND has the effect that only those points of the array, which are also already illuminated in a non-background color on the screen, are displayed. The remaining parts of an image already present on the screen are cleared.

OR superimposes the array image on the existing screen image.

XOR is particularly useful for creating animated images. Where a point on the screen has the same color as the corresponding point in the array, an inverse image is produced. If an image is PUT twice using XOR at the same position, the former background is restored. You can use this characteristic to move an object around the screen without affecting the background:

1. Using XOR, PUT the image on the screen.

2. Calculate the next position for the image.

3. PUT the image on the screen again using XOR at the position used in 1.

4. Go to step 1, using the new image location.

The default "image" is XOR.

The following tables show the results of PUTting a screen point for AND, OR, and XOR in low and high resolution color graphics. Color 0 is the background; colors 1, 2, and 3 are the colors of the current palette.

AND		Array Value for Point			
		0	1	2	3
Old Color of Screen Point	0	0	0	0	0
	1	0	1	0	1
	2	0	0	2	2
	3	0	1	2	3

OR		Array Value for Point			
		0	1	2	3
Old Color of Screen Point	0	0	1	2	3
	1	1	1	3	3
	2	2	3	2	3
	3	3	3	3	3

XOR		Array Value for Point			
		0	1	2	3
Old Color of Screen Point	0	0	1	2	3
	1	1	0	3	2
	2	2	3	0	1
	3	3	2	1	0

Example

The following program makes use of the XOR "image" of the PUT statement while moving a green ball around the screen under control of the numeric keypad of the keyboard (press Num Lock to activate the numerical function of these keys instead of their cursor movement function).

Line 50 draws a circle in red, line 55 fills it in green. As the diameter of the circle is 20 screen points, an area 20 by 20 screen points is stored by the GET command (line 70), after the top left hand corner of that area has been located (line 60). Line 150 displays the green ball again on the screen, line 160 calls a subroutine which assigns offset values to X% and Y% according to the numeric keys pressed ("7" yields minus in the x direction and minus in the y direction, "9" yields plus in the x direction and minus in the y direction, etc.). Line 170 PUTs the ball image at the old position using XOR, thus removing the image and immediately proceeds to display the ball at the next position, using the offset values contained in X% and Y%.

The error trapping facility traps the "Illegal function call" error, as this is issued when an attempt is made to PUT an image outside the screen.

```
5 ON ERROR GOTO 1100
10 DIM BALL%(64)
20 PI=3.141593
30 SCREEN 1:CLS
40 COLOR 0,0
50 CIRCLE (160,100),10,2
55 PAINT STEP (0,0),1,2
60 PRESET STEP (-10,-10)
70 GET
   (POINT(0),POINT(1))-STEP(20,20),BALL%
100 CLS
150 PUT STEP(X%,Y%),BALL%,XOR
160 GOSUB 1000
170 PUT STEP(0,0),BALL%,XOR
180 GOTO 150
500 REM
1000 REM ******* Look at keyboard
```

```
1001 IF INKEY$="" THEN GOTO 1001
1005 X%=0:Y%=0
1010 FOR LOOK%=1 TO 5
1020 K$=INKEY$
1030 X%=X%-4*(K$="9" OR K$="6" OR
     K$="3")+4*(K$="7" OR K$="4" OR
     K$="1")
1040 Y%=4*(K$="3" OR K$="2" OR
     K$="1")+4*(K$="9" OR K$="8" OR
     K$="7")
1050 NEXT LOOK%
1060 RETURN
1100 REM ******* Trap ball going off screen
1110 IF ERR=5 THEN BEEP:RESUME 50
1120 ON ERROR GOTO 0
```

Flicker is kept to a minimum by displaying the new image immediately after removing the old one (line 180).

Note

If it is not important to preserve background, you can PUT a single array containing both the new image and enough background points to cover up the old image, using PSET as "image". This saves one of the two PUT commands needed if using XOR "image", but it means that the array must be correspondingly greater.

RANDOMIZE Statement

Syntax

RANDOMIZE [numeric expression]
RANDOMIZE TIMER

Purpose

To reseed the random number generator.

Remarks

If "numeric expression" is omitted, GW-BASIC suspends program execution and asks for a value by printing

Random Number Seed (-32768 to 32767)?

before executing RANDOMIZE.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is run. To change the sequence of random numbers every time the program is run, place a RANDOMIZE statement at the beginning of the program and change the "numeric expression"with each run.

Example

```
10 RANDOMIZE
20 FOR I=1 TO 5
30 PRINT RND;
40 NEXT I
will yield
Random Number Seed (-32768 to 32767)?
```

Enter 3 in response. You will see five "random" numbers displayed. Now run the program again and enter another number in the permitted range. GW-BASIC displays a different set of numbers.

If you run the program again entering 3, you will find that the numbers displayed are the same as those displayed the first time you ran the program. The numbers would appear not to be truly random (see RND).

Note

It is often inconvenient if operator action is required to reseed the random number generator.

For this reason, GW-BASIC allows the TIME$ function to supply the "numeric expression". In this case, it makes most sense to read the seconds counter:

```
RANDOMIZE VAL (RIGHT$(TIME$,2))
```

GW-BASIC 2.02 (see "System Compatibility" in this chapter) allows you to use the TIMER function. This saves the VAL transformation and offers a wider range of values with which to reseed the random number generator. Care should be taken to ensure that the value does not exceed 32767 as follows:

```
RANDOMIZE TIMER MOD 32767
```

READ Statement

Syntax

READ list of variables

Purpose

To read items from a DATA list and assign them to variables.

Remarks

A READ statement must always be used in conjunction with DATA. READ assigns DATA items to variables on a one-to-one basis. An item READ from a DATA list must be of the same type as the variable to which it is being assigned. If they do not agree, a "Syntax error" will result.

A single READ statement may access one or more DATA items (they will be accessed in order), or several READ statements may access the same DATA items. If the number of variables in "list of variables" exceeds the number of DATA items, an "Out of data" error message is displayed. If the number of variables specified is fewer than the number of DATA items, subsequent READ statements will begin reading data at the first unread item. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA from the start, use the RESTORE command.

Examples

```
.
.
.
80 FOR I=1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
.
.
.
```

This program segment READs the values from the DATA lists into the (implicitly defined) array

A. After execution, the value of A(1) will be 3.08, and so on.

```
10 PRINT "CITY", "STATE", "ZIP"
20 READ C$,S$,Z$
30 DATA "DENVER,", COLORADO, 80211
40 PRINT C$,S$,Z$
```

will yield

```
CITY          STATE ZIP
DENVER,  COLORADO 80211
```

This program reads string and numeric data from the DATA items in line 30. Note that DATA items which include commas, semicolons or significant leading or trailing blanks must be enclosed in double quotation marks.

REM Statement

Syntax

REM remark

Purpose

To allow explanatory remarks to be inserted in a program.

Remarks

REM lines are not executed but are output exactly as entered when the program is listed.

REM lines may be branched into from a GOTO or GOSUB statement. Execution will continue with the first executable command after the REM line.

Remarks may also be added to the end of a line by preceding the remark with a single quotation mark instead of REM.

Example

```
.
.
.
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I=1 TO 20
140 SUM=SUM + V(I)
.
.
.
```

or

```
.
.
.
120 FOR I=1 TO 20 'CALCULATE AVERAGE
VELOCITY
130 SUM=SUM+V(I)
140 NEXT I
.
.
.
```

Note

In a program line containing more than one command, REM, if present, must be the last command in that line.

REM lines or appended remarks preceded by a quotation mark can be used to divide a program listing into sections and explain how the program works. However, it is not a good idea to oversaturate your program with remarks as they require memory space and increase execution time.

RENUM Command

Syntax

RENUM [new number] [,[old number] [,increment]]

Purpose

To renumber program lines.

Remarks

"new number" is the first line number to be used in the new sequence. The default is 10. "old number" is the line in the current program where renumbering is to begin. The default is the first line of the program. "increment" is the increment to be used in the new sequence. The default is 10.

RENUM also changes all line number references following GOTO, GOSUB, THEN, ELSE, ON...GOTO, ON...GOSUB, RESTORE, and RESUME commands to reflect the new line numbers.

Note

RENUM cannot be used to change the order of program lines (for example, RENUM 15,30 when the program has three lines numbered 10, 20 and 30) or to create line numbers greater than 65529. An "Illegal function call" error will result.

Examples

RENUM

Renumbers the entire program. The first new line number will be 10. Lines will be numbered in increments of 10.

RENUM 300,,50

Renumbers the entire program. The first new line number will be 300. Lines will be numbered in increments of 50.

RENUM 1000,900,20

Renumbers the lines from 900 up so they start with line number 1000 and are numbered in increments of 20.

RESET Command

Syntax RESET

Purpose To close all files on all drives.

Remarks RESET closes all open files on all drives and writes the directory track to every disk for which there were open files.

All files must be closed before a disk is removed from its drive.

Note To close individual files use CLOSE.

RESTORE Statement

Syntax

RESTORE [line number]

Purpose

To allow DATA statements to be reread from a specified line.

Remarks

After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA list in the program. If "line number" is specified, the next READ statement accesses the first item in the specified DATA line.

Example

```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
50 PRINT A;B;C;D;E;F
```

will yield

```
57 68 79 57 68 79
```

RESUME Statement

Syntaxes

```
RESUME
RESUME 0
RESUME NEXT
RESUME line number
```

Purpose

To continue program execution after an error recovery procedure has been performed.

Remarks

Any one of the four syntaxes shown above may be used, depending upon where execution is to resume:

RESUME or RESUME 0
Execution resumes at the command that caused the error.

RESUME NEXT
Execution resumes at the command immediately following the one that caused the error.

RESUME "line number"
Execution resumes at "line number".

A RESUME that is not in an error handling routine causes a "RESUME without error" message to be printed.

Example

```
10 ON ERROR GOTO 900
 .
 .
 .
900 IF (ERR=230)AND(ERL=90) THEN
    PRINT "TRY AGAIN":RESUME 80
 .
 .
 .
```

Note

RENUMBER does not attempt to change the number 0 in a RESUME 0 statement. To let you know this, GW-BASIC issues an "Undefined line number" error message.

RETURN Statement

Syntax

RETURN [line number]

"line number" specifies the number of the program line to which GW-BASIC returns after executing a subroutine.

Purpose

Returns program from a subroutine, entered by means of GOSUB or ON GOSUB.

Remarks

The use of non-local RETURNs, that is, RETURN with a "line number" requires particular care. You must make sure that such a RETURN statement does not bypass the RETURN for another subroutine which is still active.

RIGHT$ Function

Syntax RIGHT$(string expression,number)

Purpose To return the rightmost "number" of characters of the "string expression".

Remarks If "number" is greater than or equal to the number of characters in the "string expression", the result is the entire string. If "number" is 0, the null string (length zero) is returned.

Example
```
10 A$="DISK BASIC"
20 PRINT RIGHT$(A$,5)
```
will yield
```
BASIC
```

Also see the LEFT$ and MID$ functions.

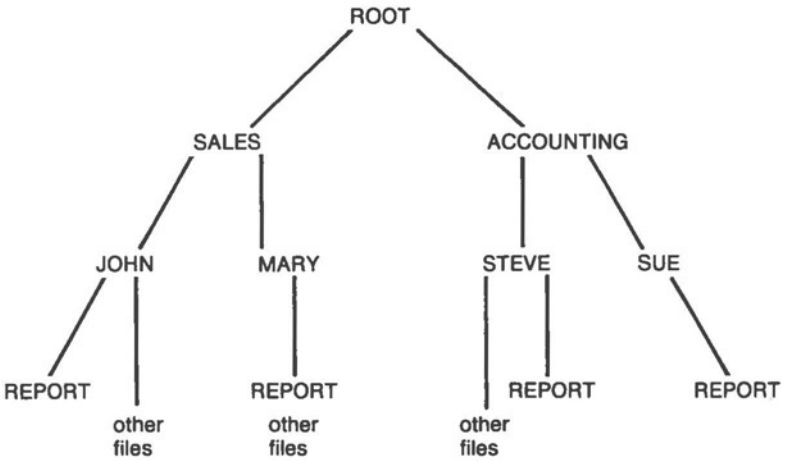
RMDIR Command

Syntax RMDIR"path"

Purpose To remove a directory from the specified drive.

Remarks "path" is a string expression not exceeding 128 characters identifying a sub-directory which is to be removed. For details about paths and directories you should refer to your *NCR-DOS* manual.

Example Given the following hierarchical structure, and assuming you are presently in the root directory



the command

RMDIR "ACCOUNTING/SUE"

removes the directory SUE following the specified path, on the condition that there are no files and no sub-directories under SUE. In this example, it would be necessary to KILL the file REPORT before issuing the RMDIR command.

Note GW-BASIC does not allow you to remove the parent directory of the directory you are currently working in. If you attempt this, a "Path/

file access" error occurs. The same error occurs if subdirectories of the directory to be removed still exist, or if files still exist in the directory.

A "File not Found" error also occurs if you try to remove a directory using KILL.

RND Function

Syntax

RND[(X)]

Purpose

To return a single precision random number between 0 and 1.

Remarks

The RND function draws upon a pseudo-random sequence of numbers stored internally by GW-BASIC. This sequence is so constructed that it, to all intents and purposes, presents a sequence of random numbers.

The same sequence of random numbers is generated each time the program is run unless the random number generator is reseeded (see RANDOMIZE).

If X is greater than 0 or X is omitted, the next random number in the sequence is generated. If X is 0, the last number generated is repeated.

A negative value of X reseeds the random number generator, but is unaffected by RANDOMIZE.

RND never quite attains the value 1. Therefore, to return a random integer value in the range 0 to 10 inclusive, for example, issue the command

```
PRINT INT(RND*11))
```

(This is not strictly necessary where the command or function calling upon RND rounds to the nearest integer.)

Examples

The following program gives you an idea of the pseudo-random nature of the "random" number sequence. (You will need <Ctrl-Break> to breakout.)

```
10 SCREEN 2:CLS
20 PSET (RND*639,RND*199)
30 GOTO 20
```

Now the same program again, this time with a greater element of chance:

```
10 SCREEN 2:CLS
20 RANDOMIZE TIMER MOD 32767
30 PSET (RND*639,RND*199)
40 GOTO 20
```

The following program repeatedly throws two dice, adds the two numbers together and builds on a column 2 to 12 according to the result:

```
10 DEFINT A-Z
20 DIM STAT(12)
30 FOOT=180
40 SCREEN 1:COLOR 0,0:CLS
50 PRESET (93,14)
60 DRAW "c2 r28 d28 128 u28 br56"
70 DRAW "r28 d28 128 u28"
80 LOCATE 25,2:PRINT "2 3 4 5 6 7 8 9 10 11 12"
90 FOR DICE=1 TO 1000
100 RANDOMIZE TIMER MOD 32767
110 D1=INT(RND*6)+1:D2=INT(RND*6)+1
120 LOCATE 3,13:PRINT D1
130 LOCATE 3,20:PRINT D2
140 THROW=D1+D2
150 LOCATE 3,30
160 PRINT"-->";THROW
170 STAT(THROW)=STAT(THROW)+1
180 PSET
    (24*THROW-36,FOOT-STAT(THROW)),1
190 FOR D=1 TO 800:NEXT D
200 NEXT DICE%
```

RUN Command

Syntax 1	RUN [line number]
Purpose	To execute the program currently in memory.
Remarks	If "line number" is specified, execution begins with that line. Otherwise, execution begins at the lowest line number.
Syntax 2	RUN "filespec"[,R]
Purpose	To load a file from disk into memory and run it.
Remarks	The "filespec" must include the filename used when the file was saved. (GW-BASIC appends the filename with the extension .BAS, if you do not supply one.) RUN closes all open files and deletes the current contents of memory before loading the designated program. However, with the "R" option, all data files remain open.
Note	RUN, even with a line number, is not suitable for continuing a program after a break, as it has the same effects on memory contents as CLEAR; notably, closing all files, erasing definitions, and clearing out variables.

SAVE Command

Syntax

SAVE "filespec" [,A / ,P]

Purpose

To save a program file on disk.

Remarks

"filespec" is a string expression that conforms to the NCR-DOS conventions for naming files. GW-BASIC appends a default filename extension .BAS if one is not supplied in the SAVE command. If a filename already exists, the file is written over.

The A option saves the file in ASCII format. If the A option is not specified, GW-BASIC saves the file in a compressed binary format. ASCII format takes more space on the disk, but some disk access requires that files be in ASCII format. For instance, the MERGE command requires an ASCII format file and ASCII files can be processed by text editors.

The P option protects the file by saving it in an encoded binary format. When a protected file is later RUN (or LOADed), any attempt to list or edit it will fail ("Illegal function call"). This characteristic cannot be reversed, so always keep an unprotected version of the program, so that you can list or edit it at a later date.

Examples

SAVE "B:COM2",A

Saves the program COM2 in ASCII format on the disk in drive B.

SAVE "ENIGMA",P

Saves the program ENIGMA as a protected file which cannot be altered.

SCREEN Statement

Syntax SCREEN [mode] [,active] [,outpage] [,seepage]

Purpose To set screen attributes for a color screen.

Remarks "mode" is an integer expression 0, 1, 2, 3, or 4 which sets the display mode: character mode, low resolution graphics, medium resolution graphics, high resolution black-and-white graphics, and high resolution color graphics, respectively.

"active" is an integer expression 0 or 1. This attribute is connected with enabling and disabling color in earlier versions of BASIC. It has no effect on the screen display of your NCR PC when using the version of GW-BASIC supplied.

"outpage" refers to the number of the page in character mode to which GW-BASIC is to write screen output. Admissible values are 0 to 7, if you are using a line width of 40 characters, or 0 to 3, if the line width is 80 characters.

"seepage" refers to the number of the page in character mode which GW-BASIC displays. If you do not specify a value, "outpage" and "seepage" are the same.

Upon execution of a valid SCREEN command, GW-BASIC sets the foreground color to white and the background color to black. (You can subsequently alter colors by means of the COLOR command.)

If the "mode" differs from the previously active mode, the screen is cleared. If you require a clear screen when setting "mode", it is a good idea to include a subsequent CLS command, as this works regardless of the previous mode.

Specifying different page numbers for "seepage" and "outpage" enables your program to write to "outpage" without affecting the current screen display. A subsequent SCREEN command can then give "seepage" the same number as

"outpage", and the screen page built up in the background appears instantaneously. There is only one cursor for all pages. Therefore, use POS and CSRLIN if you later wish to restore the cursor at a specific position on a page, before making another page the "seepage". Use LOCATE to restore the cursor to the position thus stored.

You may omit any parameter, using a comma in its place. The old value for that parameter is then retained, except "seepage" which defaults to "outpage".

Examples

```
10 SCREEN 0,1,0,0
```

selects character mode and sets "outpage" and "seepage" to 0.

```
20 SCREEN ,,1,0
```

leaves the display "mode" as it was before, and sets "outpage" and "seepage" to 1 and 0, respectively.

```
10 SCREEN 1:CLS
```

switches to or confirms low resolution graphics, and clears the screen at least once.

```
10 SCREEN 3
```

switches to or confirms high resolution black-and-white graphics.

Note

See WIDTH regarding the size of characters displayed in the graphics modes.

If your program is intended to run on both a monochrome and a color display, specify

```
SCREEN 0
```

SCREEN Function

Syntax

SCREEN(row,col[,attr])

Purpose

To return the ASCII code (0 to 255) of the character currently displayed at a specified position on the screen.

Remarks

"row" is a numeric expression in the range 1 to 25 specifying the line number.

"col" is a numeric expression in the range 1 to 40 or 1 to 80 (according to WIDTH) specifying the column number.

"attr" is permitted in character mode only. If "attr" is a non-zero value, then a number representing the display characteristics of the specified character position is returned. This number is in the range 0 to 255:

- the remainder from dividing this number by 16 (number MOD 16) gives the code of the character color (see COLOR).
- the background color is calculated as follows

  ((number-writing)/16) MOD 128

  where writing is the color code 0...15 of the character.
- if the number is greater than 127, then the character is blinking.

The SCREEN function in graphics mode returns the ASCII code, if an ASCII character is displayed at the specified position. If the position contains part of a graphic design (points, lines, etc.), the value returned is zero.

Examples

```
200 X% = SCREEN (10,1)
```

assigns to X% the ASCII code of the character in line 10, column 1.

```
210 C% = SCREEN (10,1,1)
```

assigns to C% a number in the range 0 to 255 representing the display characteristics of the same character position.

SGN Function

Syntax SGN(X)

Purpose To indicate the value of X, relative to zero:

If X>0, SGN(X) returns 1.
If X=0, SGN(X) returns 0.
If X<0, SGN(X) returns -1.

Example ON SGN(X)+2 GOTO 100,200,300

Branches to 100 if X is negative, 200 if X is 0, and 300 if X if positive.

SHELL Command

Syntax

SHELL [command string]

Purpose

To load and execute an NCR-DOS .EXE, .COM, or .BAT file and afterwards return to the GW-BASIC program command following the SHELL command.

Remarks

SHELL without a "command string" gives you the NCR-DOS system prompt, for example A>. You can then call upon NCR-DOS .EXE, .COM, or .BAT files. To return to GW-BASIC, enter the NCR-DOS EXIT command.

SHELL may include a command string in accordance with the NCR-DOS conventions for issuing commands. If you are executing your own command file, you should make sure that it does not end with the condition "Terminate and stay resident", otherwise GW-BASIC will issue the error message "Can't continue after SHELL". If there is not enough room in memory to retain GW-BASIC and execute the SHELLed program, an "Out of memory" error will occur.

If the SHELLed program is to process a file, make sure that this file is not in an open state at the time the SHELL command is executed. GW-BASIC can reopen the file as soon as the SHELLed program is terminated.

Example

```
10 OPEN "SORTIN.DAT" FOR OUTPUT AS #1
 .
 .
 .
1000 CLOSE #1
1010 SHELL      "SORT      <SORTIN.DAT
     >SORTOUT.DAT"
1020 OPEN "SORTOUT.DAT" FOR INPUT AS
     #1
 .
 .
 .
```

This example uses the NCR-DOS SORT command to sort data entered in the course of the GW-BASIC program. Note that the file must be closed (line 1000) before SHELLing the NCR-DOS command.

Note

Programmers writing their own NCR-DOS .COM or .EXE files which are to be capable of being SHELLed by GW-BASIC should take care that the interrupt vector is saved immediately upon entry to the SHELLed program, and that it is restored just before returning to GW-BASIC.

SIN Function

Syntax SIN(X)

Purpose To return the sine of X, where X is in radians.

Remarks The sine is evaluated to single precision, unless you specify the /D option when loading GW-BASIC.

Example
```
PRINT SIN(1.5)
```
will yield
```
 .9974951
```

Note To convert radians to degrees:

DEGREES = RADIANS*180/PI

where PI (single precision) is 3.141593.

To convert degrees to radians:

RADIANS = DEGREES*PI/180

SOUND Statement

Syntax

SOUND frequency,duration

"frequency"
Specifies the frequency in Hertz (cycles per second). Specify the desired number from 37 to 32767 (see Notes and Frequencies table below).

"duration"
Specifies desired length of the sound measured in clock ticks. (1 clock tick = 55 ms.) Specify the number of clock ticks in the range 0 to 65535 (see Tempo table below).

Purpose

Generates sound through the speaker.

The following table correlates notes with their frequencies. The tuning note A has a frequency of 440.

Note	Freq.	Note	Freq.
Pause	32767	F#	740
A	220	G	784
A#	233	G#	830
B	247	A	880
C	262	A#	930
C#	277.2	B	987.8
D	293.6	C	1046.4
D#	311.6	C#	1106
E	329.6	D	1174.6
F	349.2	D#	1244
F#	370	E	1318.6
G	392	F	1397
G#	416	F#	1480
A	440	G	1568
A#	466	G#	1660
B	493.2	A	1760
C	523.2	A#	1864
C#	554.8	B	1975.6
D	587.4	C	2093
D#	622	C#	2217.4
E	659.2	D	2349.4
F	598.4		

* Middle C

Remarks

SOUND produces a sound that continues until another SOUND is reached. If a SOUND statement with a "duration" of 0 is encountered, any currently running sound is turned off. (If no SOUND statement is running, SOUND "frequently", 0 has no effect.)

You can cause sounds to be buffered so program execution does not stop when a new SOUND is encountered. (See the MB command under PLAY.)

To create periods of silence, use SOUND 32767, "duration".

The "duration" for one beat is calculated from beats per minute. Divide the beats per minute into 1092 (the number of clock ticks in a minute). The following table shows typical tempos in terms of clock ticks (duration).

	Tempo	Beats/ Minute	Ticks/ Beat (Duration)
very slow	Larghissimo		
↓	Largo	40-60	27.3-18.2
↓	Larghetto	60-66	18.2-16.55
↓	Grave		
↓	Lento		
↓	Adagio	66-76	16.55-14.37
slow	Adagietto		
↓	Andante	76-108	14.37-10.11
medium	Andantino	108-120	10.11-9.1
↓	Moderato		
fast	Allegretto		
↓	Allegro	120-168	9.1-6.5
↓	Vivace		
↓	Veloce		
↓	Presto	168-208	6.5-5.25
very fast	Prestissimo		

Example

The following program creates a glissando up and down.

```
10 FOR I=220 TO 2200 STEP 20
```

```
20 SOUND I, 0.5
30 NEXT
40 FOR I=2200 TO 220 STEP -20
50 SOUND I, 0.5
60 NEXT
```

SPACE$ Function

Syntax

SPACE$(X)

Purpose

To return a string consisting of X spaces.

Remarks

The expression X is rounded, if necessary, to an integer which must be in the range 0 to 255.

Example

```
10 FOR I=1 TO 5
20 X$=SPACE$(I)
30 PRINT X$;I
40 NEXT I
will yield
   1
     2
       3
         4
           5
```

This program prints one space at the beginning of the first line, two spaces at the beginning of the second line, and so on. The additional space in each case arises through the fact the GW-BASIC prefixes each number with a space of its own in the PRINT command.

See also the SPC function.

SPC Function

Syntax SPC(I)

Purpose To skip spaces in a PRINT or LPRINT command. I is the number of spaces to be skipped.

Remarks SPC may only be used with PRINT and LPRINT. I must be in range 0 to 255. The spaces are displayed or printed as if the equivalent string of spaces were concluded by a semicolon, that is, with no automatic carriage return.

Example
```
PRINT "OVER" SPC(15) "THERE"
```
will yield
```
OVER               THERE
```

Also see SPACE$.

SQR Function

Syntax SQR(X)

Purpose To return the square root of X.

Remarks X must not be a negative number.

Example
```
10 FOR X%=10 TO 25 STEP 5
20 PRINT X%, SQR(X%)
30 NEXT X%
```
will yield
```
 10          3.162278
 15          3.872984
 20          4.472136
 25          5
```

STICK Function

Syntax

STICK(n)

n is a numeric expression returning an integer in the range 0 to 3.

Purpose

To return the x and y coordinates of the two joysticks.

Remarks

The values for n can be:

0 — returns the x coordinate for joystick A. Also prepares the x and y values for both joysticks for the following function calls:

1 — Returns the y coordinate of joystick A.

2 — Returns the x coordinate of joystick B.

3 — Returns the y coordinate of joystick B.

Even if you only wish to read the values for joystick B, you must execute a dummy command using STICK(0).

Example

```
50 DISCARD=STICK(0)
60 X%=STICK(2)
70 Y%=STICK(3)
80 PRINT X%,Y%
```

This program samples the x and y coordinates of joystick B.

Note

STRIG is for use in connection with joystick buttons.

STOP Statement

Syntax

STOP

Purpose

To terminate program execution and return to command level.

Remarks

STOP statements may be used anywhere in a program to terminate execution. STOP is often used for debugging. Following STOP you can inspect and alter program variables, and then continue with CONT.

When a STOP is encountered, the following message is printed:

Break in nnnnn

Unlike END, STOP does not close files.

Example

The following loop is executed until you press a key. You can then see how far the counter has progressed by entering PRINT CT as a direct command. CONT enables the program to continue where it left off.

```
10 CT=0
20 IF INKEY$<>""THEN STOP
30 CT=CT+1
40 FOR SLOTH% =1 TO 200:NEXT SLOTH%
50 GOTO 20
```

STR$ Function

Syntax

STR$(X)

Purpose

To return a string representation of the value yielded by the numeric expression X.

Remarks

If X is positive, the string representation is preceded by a single blank. Therefore, the length of the string returned by STR$ is one character greater than a positive number it represents.

Example

The following program doubles any number you enter, provided the number is not longer than two digits:

```
10 REM Arithmetic for kids
20 INPUT "Enter an EASY number";N
30 IF LEN(STR$(N))>3 THEN PRINT "I said
   EASY. Try again":GOTO 20
40 PRINT N;"doubled is";N*2
```

Note

The complementary function of STR$ is VAL.

STRIG Statement

Syntax

```
STRIG ON
STRIG OFF

STRIG (n) ON
STRIG (n) OFF
STRIG (n) STOP
```

Purpose

To enable and disable the joystick buttons.

To enable and disable the trapping of joystick buttons.

Remarks

STRIG ON has the effect that every time GW-BASIC is about to execute a command, it performs a check to see if a button has been pressed.

STRIG OFF tells GW-BASIC to ignore the presence of the joystick buttons.

STRIG (n) ON enables trapping of the button specified by n. Up to four buttons may be trapped, using the values 0, 2, 4, and 6 for n.

STRIG (n) OFF disables trapping for the specified button 0, 2, 4, or 6. If a button is pressed between execution of this command and the next STRIG (n) ON, GW-BASIC does not remember the event.

STRIG (n) STOP disables trapping for the specified button 0, 2, 4, or 6. If a button is pressed after execution of this command, GW-BASIC will branch to the event handling routine as soon as it has encountered the next STRIG (n) ON statement.

STRIG Function

Syntax

STRIG (n)

Purpose

To return information as to whether a joystick button is being pressed or has been pressed since the last time it was checked.

Remarks

n is a numeric expression in the range 0 to 7. One of four buttons can be checked, according to whether n is 0, 2, 4, 6. STRIG (n) returns the value -1, if the button has been pressed since the last time it was checked; otherwise, it returns 0.

If n is 1, 3, 5, or 7, STRIG (n) returns the value -1, if that button is currently pressed.

STRIG ON must have been executed before button checking can take place.

Example

```
10 STRIG ON
20 IF STRIG(2) THEN PRINT "Somebody has
   pressed button 2"
30 IF STRIG(3) THEN PRINT "Now please
   release button 2": GOTO 50
40 PRINT "Have you forgotten the existence of
   button 2?"
50 END
```

STRING$ Function

Syntax

```
STRING$(I,J)
STRING$(I,X$)
```

Purpose

To return a string of length I whose characters all have ASCII code J or the first character of the string expression X$.

Example

```
10 X$=STRING$(10,45)
20 PRINT X$ "MONTHLY REPORT" X$
```

will yield

```
----------MONTHLY REPORT----------
```

SWAP Statement

Syntax SWAP variable1,variable2

Purpose To exchange the values of two variables.

Remarks Any type variable may be swapped (integer, single precision, double precision, string), but the two variables must be of the same type or a "Type mismatch" error results.

If the second variable is not already defined when SWAP is executed, an "Illegal function call" error will result.

Example

```
10 A$="ONE" : B$="ALL" : C$="FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
will yield
ONE FOR ALL
ALL FOR ONE
```

SYSTEM Command

Syntax **SYSTEM**

Purpose **To close all open files and return control to NCR-DOS.**

Remarks **Any GW-BASIC program in memory is lost as soon as SYSTEM is executed, so consider SAVEing the program if you have updated it since the last SAVE.**

TAB Function

Syntax

TAB(I)

Purpose

To move the display or print position to I.

Remarks

If the current print position is already beyond space I, TAB goes to that position on the next line. Space 1 is the leftmost position; the rightmost position is the defined WIDTH. I must be in the range 1 to 255. TAB may only be used in PRINT, PRINT #, and LPRINT commands.

TAB at the end of a list of PRINT or LPRINT items is regarded by GW-BASIC as having a semicolon; therefore, no automatic carriage return occurs.

Example

```
10 PRINT "NAME" TAB(25) "AMOUNT"
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G. T. JONES", "$25.00"
```

will yield

```
NAME                    AMOUNT

G. T. JONES             $25.00
```

This program shows TAB being used to create neat display columns.

TAN Function

Syntax TAN(X)

Purpose To return the tangent of X. The angle X is in radians.

Remarks The tangent is calculated to single precision, unless you specify the /D option when loading GW-BASIC.

Example
```
10 RADS=0.78
20 PRINT TAN(RADS)
```
will yield
.9892613

Note To convert radians to degrees:

DEGREES = RADIANS*180/PI

where PI (single precision) is 3.141593.

To convert degrees to radians:

RADIANS = DEGREES*PI/180

TIME$ Statement

Syntax

TIME$=string expression

"string expression" represents a string in one of the following forms:

hh
(sets the hour; minutes and seconds default to 00)

hh:mm
(sets the hour and minutes; seconds default to 00)

hh:mm:ss
(sets the hour, minutes, and seconds)

Purpose

To set the time. This command complements the TIME$ function, which retrieves the time.

Remarks

A 24-hour clock is used; 8:00 p.m., therefore, would be entered as 20:00:00.

Example

```
10 TIME$="08:00:00"
```

The current time is set at 8:00 a.m.

Note

If you have set the time at the NCR-DOS command level, there is no need to set it again from within GW-BASIC.

TIME$ Function

Syntax

TIME$

Purpose

To retrieve the current time. (To set the time, use the TIME$ command.)

Remarks

The TIME$ function returns an eight-character string in the form hh:mm:ss, where hh is the hour (00 through 23), mm is minutes (00 through 59), and ss is seconds (00 through 59). A 24-hour clock is used; 8:00 p.m., therefore, would be shown as 20:00:00.

Example

```
10 ALARM$ = TIME$
20 IF LEFT$ (ALARM$,2)="06" AND MID$
   (ALARM$,4,2) ="30"THEN BEEP:PRINT
   "Your early morning call":GOTO 40
30 GOTO 10
40 IF INKEY$="" THEN GOTO 10 ELSE END
```

This example repeatedly checks the clock. At 6:30 a.m. it starts BEEPing and continues to do so for a whole minute or until you press a key. (You might prefer to replace BEEP with some music.)

TIMER Function

Syntax

TIMER

Purpose

To return a single precision number representing the number of seconds that have elapsed since midnight or the last time you switched on or reset your computer.

Remarks

TIMER starts counting at 0, and starts again with zero a fraction of a second before 86400 would be attained. TIMER returns whole seconds and fractions of a second.

Example

The following program gives you an idea of how much time elapses in a GW-BASIC delay loop:

```
10 INPUT "How many runs through loop";R
20 TIME$ = "00:00:00:"
30 FOR X% = 1 TO R: NEXT X%
40 PRINT TIMER; "seconds"
```

TRON AND TROFF Commands

Syntax

TRON

TROFF

Purpose

To trace the execution of program commands.

Remarks

As an aid in debugging, TRON (executed in either direct or indirect mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace flag is disabled with the TROFF command (or when a NEW command is executed).

Example

TRON

```
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINT J;K;L
50 K=K+10
60 NEXT
70 END
```

will yield

```
[10] [20] [30] [40] 1  10  20
[50] [60] [30] [40] 2  20  30
[50] [60] [70]
```

TROFF

The numbers not enclosed in brackets are the result of PRINT statements.

USR Function

Syntax

USR[digit] [(argument)]

where "digit" specifies which USR routine is being called. See DEF USR for rules governing "digit". If "digit" is omitted, USR0 is assumed.

"argument" is the value passed to the subroutine. It may be any numeric or string expression.

Purpose

To call an assembly language subroutine.

Remarks

If a segment other than the default segment (data segment DS) is to be used, a DEF SEG statement must be executed prior to a USR function call. The address given in the DEF SEG statement determines the machine location of the beginning of the segment to which the address specified in DEF USR is offset.

For each USR function, a corresponding DEF USR must be executed to define the USR call offset. This offset and the currently active DEF SEG segment address determine the starting address of the subroutine.

Example

```
100 DEF SEG=&H8000
110 DEF USR0=0
120 X=5
130 Y = USR0(X)
140 PRINT Y
```

Line 130 calls the machine language subroutine at the beginning (address 0) of the segment which starts at the machine memory address hexadecimal 8000. A single value returned by the subroutine is assigned to the variable Y.

Note

If your machine language program is not required to return a value to your BASIC program, the variable to the left of the equal sign in the command containing the USR function plays only a dummy role.

Another way of accessing a machine language subroutine is to use the CALL command.

Chapter 6 contains more information about using machine language routines in GW-BASIC programs.

VAL Function

Syntax

VAL(X$)

Purpose

To return the numerical value of string expression X$. The VAL function strips blanks, tabs, and linefeeds from the argument string. For example,

```
VAL("   -3")
```

returns -3.

Remarks

If the string does not begin with numeric characters, VAL returns 0.

The constraints of the argument string when using CVI, CVS, or CVD do not apply to VAL, so it is especially useful for converting strings of variable length to numeric values.

Example

You might wish to evaluate the numeric significance of information which has been read into a random file buffer as a string, for example, dates:

```
10 FIELD #1,<4 AS YEAR$, 2 AS MONTH$, 2
   AS DAY$
 .
 .
 .
120 GET #1,1
130 IF VAL(YEAR$+MONTH$+DAY$) < 195-
    40713 THEN PRINT "Older than I am"
 .
 .
 .
```

Note

The complementary function STR$ converts numeric values to strings.

VARPTR Function

Syntax 1 VARPTR(variable name)

Syntax 2 VARPTR(#file number)

Purpose Syntax 1

Returns the address of the first byte of data identified with "variable name". A value must be assigned to "variable name" prior to execution of VARPTR; otherwise, an "Illegal function call" error results. Any type variable name may be used (numeric, string, array). For string variables, the address of the first byte of the string descriptor is returned (see Chapter 6). The address returned is an integer in the range 0 to 65535.

VARPTR is usually used to obtain the address of a variable or array so that it may be passed to an assembly language subroutine. A function call of the form VARPTR(A(0)) can be specified when passing an array, so that the lowest-addressed element of the array is returned.

Note All simple variables should be assigned before calling VARPTR for an array, because the addresses of the arrays change whenever a new simple variable is assigned.

Syntax 2

For sequential files, VARPTR# returns the starting address of the disk I/O buffer assigned to "file number". For random files, the address of the FIELD buffer assigned to "file number" is returned (see Chapter 6).

VARPTR$ Function

Syntax

VARPTR$(variable name)

where "variable name" is the name of a variable in the program.

Purpose

To return a character form of the memory address of the variable.

Remarks

VARPTR$ is primarily used with the DRAW and PLAY statements in programs that will be compiled.

A value must be assigned to "variable name" prior to execution of VARPTR$; otherwise, an "Illegal function call" error results. Any type variable (numeric, string, or array) may be used.

VARPTR$ returns a three-byte string in the form:

byte 0 = type: 2 = integer, 3 = string, 4 = single precision, 5 = double precision
byte 1 = low byte of address
byte 2 = high byte of address

Example

```
10 PLAY "X"+VARPTR$(A$)
```

uses the subcommand X, plus the address of A$, as the string expression in the PLAY statement.

In the GW-BASIC interpreter supplied, this is the same as

```
10 PLAY "XA$;"
```

Note

Because array addresses change whenever a new variable is assigned, always assign all simple variables before calling VARPTR$ for an array element.

VIEW Statement

Syntax

VIEW [[SCREEN] [(x1,y1)-(x2,y2)[,[filling] [,[outline]]]]]

Purpose

To define subsets of the screen ("viewports") in the graphics display modes in order to limit screen activity to a specified area.

Remarks

VIEW without any parameters defines the entire screen as the viewport. The RUN command has the same effect.

(x1,y1)-(x2,y2) are the coordinates of the top left and bottom right corners of the rectangular viewport, respectively. Unlike most other coordinate specifications in GW-BASIC, these coordinates must represent screen points actually available, otherwise an "Illegal function call" error occurs.

"filling" allows you to fill the viewport with color. "filling" is therefore a number 0 to 3 in low and high resolution color graphics (0: background; 1 to 3: color from color palette), or 0 (black) or 1 (white) in medium and high resolution black-and-white graphics. If you do not specify a color, no filling is performed.

"outline" allows you to draw a boundary line in a specified color (see "filling"), if space is available.

The viewport specified may not extend beyond the screen. The coordinates specified for the two diametrically opposed corners must not be identical.

If the word SCREEN is not included in the VIEW command, the coordinates of subsequent graphics drawing are relative to the viewport. Thus, it is possible to display the same graphics design with different scaling, if you have previously issued an explicit WINDOW command.

If the word SCREEN is included, the physical screen area addressed by graphics commands is unaltered by the viewport, but only those parts

which fall within the viewport are actually displayed.

Only one viewport can be active at a given time.

CLS affects only the current viewport. To clear the entire physical screen, first disable the viewport by means of VIEW without parameters.

Examples

The first example shows the use of VIEW to draw a circle, first with the normal screen display scale, then removed and reduced. The circle using normal low resolution graphics' screen coordinates is drawing in green (line 30), then a small viewport is defined and outlined in red (line 40). Finally, a circle is drawn in brown. Note that this latter circle uses the same coordinates for its center and the same radius as the first circle, but is now scaled to the new viewport.

```
10 SCREEN 1:CLS:COLOR 0,0
20 WINDOW SCREEN (0,0)-(319,199)
30 CIRCLE (160,100),70,1
40 VIEW (40,30)-(90,70),,2
50 CIRCLE (160,100),70,3
```

While still in the graphics mode enter CLS as a direct command. Only the small, brown circle disappears.

The following program shows VIEW with the SCREEN option. First, a circle is drawn in normal screen characteristics (line 20). Then a viewport is defined in the top left quarter of the screen (line 30). Only the part of the second circle which lies within the viewport is then actually drawn (line 40).

```
10 SCREEN 1:CLS:COLOR 0,0
20 CIRCLE (160,100),96,1
30 VIEW SCREEN (1,1)-(159,99),,2
40 CIRCLE (160,100),90,3
```

WAIT Command

Syntax

WAIT port number,I[,J]

where I and J are integer expressions.

Purpose

To suspend program execution while monitoring the status of a machine input port.

Remarks

The WAIT command causes execution to be suspended until a specified machine input port develops a specified bit pattern. The data read at the port is XORed with the integer expression J, and then ANDed with I. If the result is zero, GW-BASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next command. If J is omitted, it is assumed to be zero.

Example

```
100 WAIT 32,2
```

Suspends program execution until the value 2 is present at port 32.

Note

There is no error trapping facility in the event that the specified bit pattern (value) does not appear at the port, but you can break out with Ctrl-Break.

WHILE AND WEND Statements

Syntax

```
WHILE expression
.
.
[loop commands]
.
.
WEND
```

Purpose

To execute a series of commands in a loop as long as a given condition is true.

Remarks

If "expression" is true (that is, resulting in a non-zero value) , "loop commands" are executed until WEND is encountered. GW-BASIC then returns to the WHILE statement and checks "expression". If it is still true, the process is repeated. If it is not true, execution resumes with the command following WEND.

WHILE/WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE causes a "WHILE without WEND" error, and an unmatched WEND a "WEND without WHILE" error.

Example

```
 90 'bubble sort array a$ containing J elements
100 FLIPS=1 'force one pass thru loop
110 WHILE FLIPS
115     FLIPS=0
120     FOR I=1 TO J-1
130         IF A$(I)>A$(I+1) THEN
                SWAP A$(I),A$(I+1):FLIPS=1
140     NEXT I
150 WEND
```

This example sorts the elements of the array A$ into alphabetical, or, to be more precise, ascending ASCII sequence. The leading spaces in lines 115 to 140 are a programming convention which has no effect on the way GW-BASIC executes the commands. They simply reflect the depth of nesting.

WIDTH Statement

Syntax

WIDTH file number,size
WIDTH "device",size
WIDTH size

"file number"
Numeric expression in the integer range 1 to 15. This is the number of a file opened for a device.

"size"
Numeric expression in the integer range 1 to 255. This is the new width. If you specify 0, GW-BASIC understands 1.

"device"
String expression which identifies the device. Valid devices are "SCRN:", "LPT1:","LPT2:", "LPT3", "COM1:", and "COM2:".

Purpose

Sets the output line width in number of characters.

Remarks

WIDTH file number,size
If the file is open to LPT1:, the line printer's printed line width is immediately changed to the new size specified. This command allows you to change the width at will while the file is open. This form of the WIDTH statement is also meaningful for any of the other devices specified above.

WIDTH device,size
Used as a deferred width assignment for the line printer, this form of the WIDTH statement stores the new width value without actually changing the current width setting. A subsequent OPEN statement for the specified device will use the new size specified while the file is open.

Note that the LPRINT, LLIST, and LIST commands perform and automatic OPEN. You do not have to explicitly open the device.

WIDTH size
or
WIDTH "SCRN:",size

Sets the number of characters which can be displayed in a screen line. You may specify either 40 or 80. If the screen display is in either of the graphics modes, WIDTH 40 automatically selects or confirms low resolution graphics. When entered while in low resolution mode, WIDTH 80 selects high resolution graphics. If the screen display is in medium or high resolution graphics, specifying WIDTH 80 has no effect.

If you enter any value outside the legal ranges an "Illegal Function Call" error occurs. The previous value is retained.

No data is lost by using the WIDTH command. GW-BASIC simply adds a carriage return after sending the number of characters specified as "size". For example, if you have a 60-character line and a 40-character printer, and if you issue WIDTH 40, the first 40 characters will be printed on one line and the next 20 characters on the next line.

The transmit and receive buffer of a communications file are not altered by WIDTH. The only effect is that GW-BASIC adds a carriage return as soon as "size" characters are in the buffer.

The default WIDTH for printing devices is 80; for communications files 255 with no line folding.

WINDOW Statement

Syntax

WINDOW [[SCREEN](x1,y1)-(x2,y2)]

Purpose

To redefine the coordinates of the screen in medium and high resolution graphics.

Remarks

x1,y1 and x2,y2 represent in single precision numbers the coordinates of two diametrically opposite corners of the screen. For example, your program can determine that the screen in low resolution graphics is to be regarded by GW-BASIC as consisting of 240 (horizontal) by 150 points. The result is that a graphics drawing appears larger than it did when using the default 320 by 200 coordinate system, without you having to change the values in the drawing commands. This is often called a zoom effect, a term which you may know from photography.

Exactly which two corners are defined by the "world coordinates" x1,y1 and x2,y2 depends on whether or not you specify SCREEN in the WINDOW statement. If you include SCREEN, the GW-BASIC convention is retained; that is, x1,y1 is the upper left corner, and x2,y2 is the bottom right corner. WINDOW sorts the two coordinate pairs, placing the smaller values for x and y first, even if you specify them the other way around. This means that movement from left to right across the screen increases the x value, while movement down the screen increases the y value.

If you do not include SCREEN, x1,y1 refers to the bottom left corner of the screen, and x2,y2 refers to the top right corner. The Cartesian scheme then applies, namely, that movement down the screen *decreases* the y value.

Figure 1 shows the coordinate scheme as set automatically when high resolution graphics mode is selected. The explicit statement to set this scheme would be

```
WINDOW SCREEN(0,0)-(639,199)
```

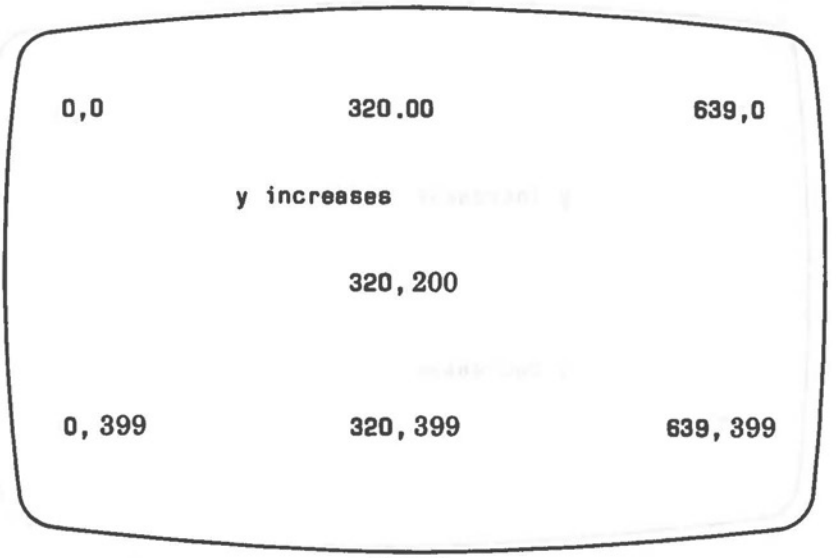


Figure 1

Figure 2 shows the Cartesian coordinate scheme with the origin (0,0) in the center of the screen. The coordinates are here given symbolically in terms of 1, 0, and -1 to denote the transition from negative to positive values of x and y, and, at the same time, to remind you that you are by no means obliged to use (in high resolution graphics)

WINDOW (-320,-200) - (319,199)

although this offers the maximum degree of graphic resolution for that graphics mode. Furthermore, there is no reason why the origin must be in the center of the screen.

The general formulation of the WINDOW command for Figure 2 is

WINDOW (-1,-1)-(1,1)

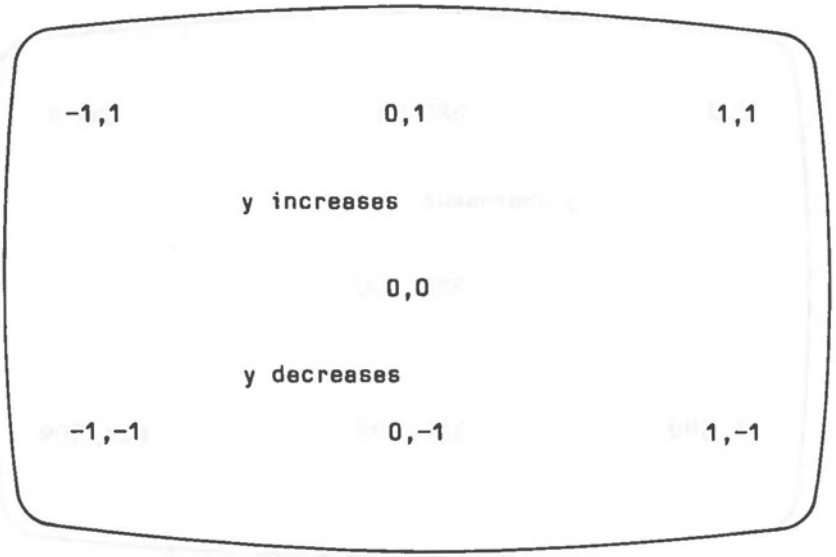


Figure 2

Figure 3 shows in generalized form the standard high resolution graphics coordinate scheme given in Figure 1. The equivalent WINDOW command is

WINDOW SCREEN (-1,-1)-(1,1)

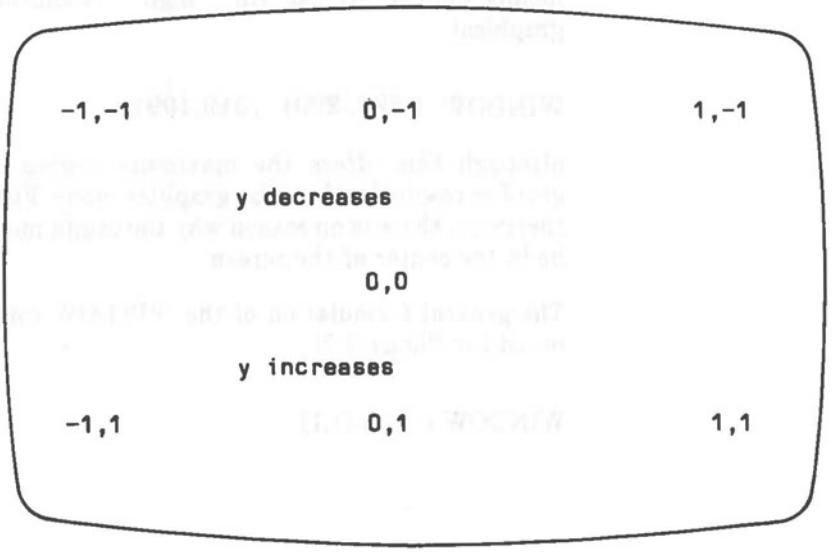


Figure 3

The effect of WINDOW sorting the coordinate pairs is that if you, for example, specify

WINDOW (200,200)-(10,10)

this is interpreted as

WINDOW (10,10)-(200,200)

A newly set window applies to subsequent graphics drawing. It does not affect existing screen contents.

The two sets of coordinates in the WINDOW statement must not be identical.

WINDOW uses "clipping"; that is, if your graphics design extends beyond the coordinate range defined for the screen, those parts outside the range are not displayed (there is no wrap around to another part of the screen).

If you specify WINDOW without any parameters, normal physical screen coordinates are restored. The RUN command and SCREEN statement have the same effect.

Example

The following program demonstrates zooming, panning, and clipping in medium resolution graphics.

First, Cartesian coordinates are set with the origin (0,0) at the center of the screen (line 20), and the axes are drawn in green through the origin. Two boxes are then drawn, one in the bottom left quadrant, and the other in the top right quadrant. You can then specify a point in terms of the current Cartesian coordinate scheme of (-160,-100)-(159,99).This point is to become the new focus for zooming. Then, you are asked to specify a zoom factor. A value greater than 1 produces zoom in; a value less than 1 produces zoom out. The boxes drawn in line 143 are suitable for zoom in. For zoom out, try making this the REM line and removing REM from line 145.

```
5 X1=-160:Y1=-100:X2=159:Y2=99
10 SCREEN 1:CLS:COLOR 0,0
20 WINDOW (X1,Y1)-(X2,Y2)
40 GOSUB 130
50 LOCATE 1,1:INPUT;"zoom/pan to x,y posi-
   tion? ";X,Y
60 LOCATE 1,1:INPUT;"zoom factor - zoom
   out:< 1?          ";ZP
61 LOCATE ,,0
62 FOR SC= 1 TO ZP STEP (ZP<=1)*ZP/30-
   (ZP>1)*(ZP-1)/60
70 CLS
80 WINDOW
   ((X*SC+X1)/SC,(Y*SC+Y1)/SC)-
   ((X*SC+X2)/SC,(Y*SC+Y2)/SC)
100 GOSUB 130
102 NEXT SC
110 IF INKEY$="" THEN 110
120 STOP
125 REM ***** Draw axes and boxes
130 LINE (0,50)-(0,-50),1:LINE (-60,0)-(60,0),1
143 LINE (20,20)-(30,30),2,BF:LINE
   (-20,-20)-(-5,-5),3,BF
145 REM LINE (20,20)-(60,60),2,BF:LINE
   (-40,-40)-(-5,-5),3,BF
170 RETURN
```

WRITE Statement

Syntax

WRITE [list of expressions]

Purpose

To output data to the screen.

Remarks

If "list of expressions" is omitted, a blank line is output. If "list of expressions" is included, the values of the expressions are output to the screen. The expressions in the list may be numeric and/or string expressions. They must be separated by commas or semicolons.

When the items are output, each item is separated from the last by a comma. When displayed, strings are delimited by quotation marks. Positive numbers are not preceded by blanks. After the last item in the list is printed, GW-BASIC inserts a carriage return/linefeed. These are the features which distinguish WRITE from PRINT.

WRITE outputs numeric values using the same format as PRINT.

Example

```
10 A=80:B=90:C$="THAT'S ALL"
20 WRITE A,B,C$
will yield
 80, 90,"THAT'S ALL'
```

WRITE# Statement

Syntax

WRITE#file number,list of expressions

Purpose

To write data to a sequential file.

Remarks

"file number" is the number under which the file was OPENed. The expressions in the list are string or numeric expressions. They must be separated by commas or semicolons.

The difference between WRITE# and PRINT# is that WRITE# inserts commas between the items as they are written to the file and delimits strings with quotation marks. Therefore, it is not necessary to put explicit delimiters in the list. A carriage return/linefeed sequence is inserted after the last item in the list is written to the file.

Example

Let A$="CAMERA" and B$="93604-1"

The statement:

WRITE#1,A$,B$

writes the following image to the file:

"CAMERA","93604-1"

A subsequent INPUT$ statement, such as

INPUT#1,A$,B$

would input "CAMERA" to A$ and "93604-1" to B$.

Files and Devices

The term "file" refers not only to the name under which we SAVE and LOAD GW-BASIC programs. We also use this term for any collection of data stored on disk which is capable of being processed by a program. A "device" normally exists outside the computer cabinet and is capable of receiving and/or transmitting data, or even converting data from one form to another. Examples of devices are the keyboard, a printer, a telephone modem. Even the screen inside the cabinet can be considered to be a device.

Files and devices are discussed in a common chapter because GW-BASIC addresses them in the same way. Any type of input/output can be treated as if it is related to a disk file. Obviously, the special physical characteristics of the device must be taken into account. For example, you can easily inspect a record already written to disk, but you cannot expect a printer to roll back six pages and read back to your program what is written on that page.

GW-BASIC expects you to say which files you wish to access. You do this by means of the OPEN statement. This statement asks you to state the name of a file and a number with which that file is to be associated as long it is open. Normally, GW-BASIC allows you to have up to three data files open at any time, but you can change this number using the /F option when loading GW-BASIC (see "Starting-up GW-BASIC" in Chapter 1). When you have finished working with a file you should CLOSE it. This frees some memory space and ensures that important information, such as the latest state of a disk directory, has been noted by GW-BASIC.

EVERY FILE NEEDS A NAME

A filename may be up to eight characters long. All letters of the alphabet and all digits are allowed characters. In addition, the filename may include the following characters:

( ) { } @ # $ % ^ & ! - _ ' / ~ |

If you wish, you can append a period (.) to the filename followed by an extension consisting of up to three of the legal characters. GW-BASIC does this automatically to program files (.BAS), if you do not specify a different extension.

It makes sense to give a file a name which has something to do with its existing or prospective contents. For example, you might call a program which carries out a market analysis MARKET.BAS, and the files containing the reports that are processed in the course of the analysis MKTRPT1, MKTRPT2, and so on.

At the same time as stating the name of a file (with an extension, if one is provided), you may wish to specify in which drive the disk containing that file is situated. In this case, you must precede the filename with the drive letter and a colon, for example

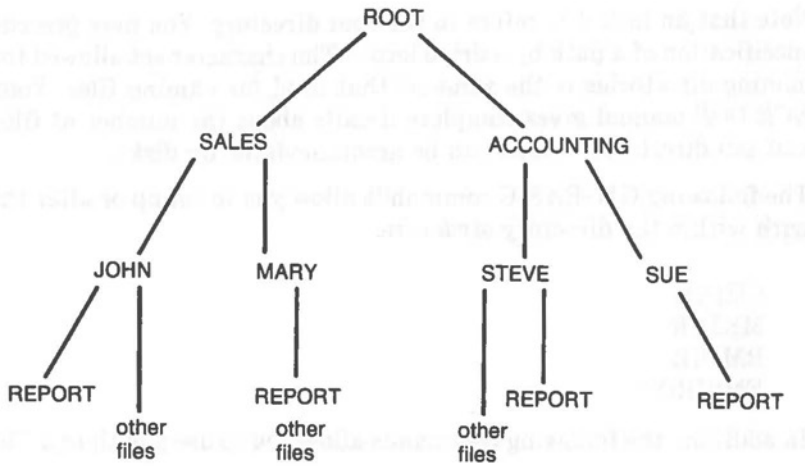
B:MARKET

You will normally specify the drive if you know the file is not present on, or is not to be created on, the currently active disk.

The version of GW-BASIC supplied also enables you to specify a path of access to a file. Your *NCR-DOS* manual tells you all about directories and paths. Here is a quick summary of this facility.

A single disk may contain not just one but a number of directories arranged in a tree-like structure. The main directory is called the root directory. This is the directory to which all paths must lead back. A path is the route you have to follow through the structure in order to access a particular file. You can specify such a path either from the root directory to a file or from the current directory to a file. A directory may contain sub-directories and/or files. If you do not specify a path, GW-BASIC assumes that the file is to be found in the current directory.

The path to a file is denoted by one or more directory names separated by \ and concluded by a filename (.extension). The symbol .. denotes the parent directory, that is, the directory immediately above in the hierarchical structure. A path may be preceded by a drive letter.



Given the structure in the previous illustration and assuming that the current directory is JOHN, the path

```
REPORT
```

or

```
\SALES\JOHN\REPORT
```

or

```
..\JOHN\REPORT
```

all reference the REPORT in the directory JOHN. To gain access to the REPORT under MARY, you would have to specify the path as

```
..\MARY\REPORT
```

or

\SALES\MARY\REPORT. To access the REPORT under SUE, you can specify the path

```
..\..\ACCOUNTING\SUE\REPORT
```

or

```
\ACCOUNTING\SUE\REPORT
```

Note that an initial \ refers to the root directory. You may precede specification of a path by a drive letter. The character set allowed for naming directories is the same as that used for naming files. Your *NCR-DOS* manual gives complete details about the number of files and sub-directories which can be accommodated on disks.

The following GW-BASIC commands allow you to set up or alter the path within the directory structure:

CHDIR
MKDIR
RMDIR
ENVIRON

In addition, the following commands allow you to use a path to a file:

BLOAD	MERGE
BSAVE	NAME
CHAIN	OPEN
FILES	RUN
KILL	SAVE
LOAD	

If you have not as yet specified any directories, either in GW-BASIC or outside GW-BASIC at the NCR-DOS command level, GW-BASIC assumes the root directory. In this case, you do not have to explicitly state the root directory when accessing files. Therefore, you do not need to be concerned with different directories and the paths to them, if you do not yet wish to use this facility of the NCR-DOS Operating System. You can refer to both program and data files using filename, extension where provided, and drive letter where appropriate.

DEVICE NAMES

Unlike filenames, device names are already determined by GW-BASIC:

A: }
B: } the disk drives: the flexible disk drives (or drive) are
C: } designated A: and B:, the first fixed disk is designated C:, and so on.
D: }

KYBD: the keyboard.

SCRN: the screen display.

LPT1: }
LPT2: } printers (if present).
LPT3: }

COM1: } adapters for asynchronous communications
COM2: }

REDIRECTION OF STANDARD INPUT/OUTPUT

The standard input device is the keyboard, the standard output device is the screen display. You can redirect standard input and output to files or suitable devices. This is achieved by specifying the < or > option in the NCR-DOS command line which loads GW-BASIC.

Examples:

```
GW-BASIC ANYPROG >PROTOCOL.DSK
```

means that any data which would normally appear on the screen will be sent to the disk file PROTOCOL.DSK instead.

```
GW-BASIC FASTKEY <REPLACE.KEY >PROTOCOL.DSK
```

has the effect that data which would normally appear on the screen will now be sent to the disk file PROTOCOL.DSK. Input from the keyboard is suspended. In its place, input is derived from the disk file REPLACE.KEY.

If you specify not one but two >> when redirecting standard output, the output does not replace but is appended to the existing file. Example:

```
GW-BASIC SECRET >>COLLECT.DAT
```

appends what would otherwise be the screen output to the disk file COLLECT.DAT.

Regardless of input and output redirection:

- error messages are still displayed on the screen.
- INPUT$ and input from the specified device KYBD: are still derived from the keyboard.
- Output explicitly directed to the output device SCRN: is displayed on the screen.

- Trapping of keys set up by an ON KEY(n) statement is still in force.

Ctrl-PrtSc does not copy the screen as long as standard output is redirected. The redirection of standard output is terminated if you press Ctrl-Break.

HOW TO USE DISK DATA FILES

You can create and access two types of disk data file. Files for sequential access (for the sake of brevity usually called "sequential files") and files for random access ("random files") both store data in units of records. Your program can determine the length of the record to suit the data you wish to store. For example, you may decide that a file should store weather observations of the last 24 hours. Your record might then look something like this:

10 bytes for the name of the weather station
4 bytes for time of observation
3 bytes for wind direction
2 bytes for wind force
3 bytes for temperature
2 bytes for relative humidity
4 bytes for atmospheric pressure
4 bytes for visibility

Accordingly, you would choose a record length of at least 32 bytes. If you do not specify a record length, GW-BASIC assumes 128 bytes.

Then you must decide whether you would like to store and access the data as a sequential file or as a random file. The characteristics of these two types of file are as follows.

- Sequential files

  As the name suggests, data is written to and read from a sequential file in a fixed sequence. The first record you write is record 1, the second record written is record 2, and so on. It is not possible to write, say, six records and then ask GW-BASIC to note that you might wish to insert a record between records 2 and 3 at a later date.

  The same fixed sequence applies when reading the file. Your program must read the records one by one from the beginning until it finds the record it is looking for. It can then read the record, but not alter it. This is because a sequential file can be

open for input, output, or appending at any one time, but not for more than one of these functions. Appending allows you to add records at the end of an existing file, but does not allow you to change the sequence of existing records.

- Random files

  Random files allow you to specify a record number which deviates from the ascending sequence of record numbers associated with sequential files. You can, for example, write records 1 to 6, and then continue with record 9, thus leaving "space" for two additional records which you can insert at a later date. You can read the records in any order you wish, and you can alter a record without having to read the file from the beginning. Random files usually store numeric items in a compressed format, so if you are working mainly with numbers, using random files can save disk space.

When comparing the advantages of the two types of file access it should also be mentioned that random files require more programming than do sequential files. Returning to the example of the weather reports, you would probably decide that the observations of the last 24 hours, or the last week, are required at present for fast random access in order to do the calculations necessary for making a forecast. Observations which go further back in time are for the archives and can be stored in chronological order in sequential files. They need no longer be accessed quickly, but still provide source information from which statistics can be derived.

SEQUENTIAL FILES

The following commands and functions are used with sequential files:

OPEN, CLOSE (the OPEN command can be written in two different ways, see Chapter 4)
INPUT$, INPUT#, LINE INPUT# - reading data from the file
PRINT#, PRINT# USING, WRITE# - writing data to the file
EOF, LOC, LOF - end of file, location in file, length of file.

Creating a Sequential File.

Here is an example of a new sequential file being opened to receive data from a program. The GW-BASIC default record length of 128 bytes applies. Each record consists of the concatenation of the strings N$ (name), D$ (department), and H$ (date hired), with separating commas. A record is written each time line 50 is executed.

```
10 OPEN "O",#1,"DATA"
20 INPUT "NAME";N$
25 IF N$="DONE" THEN CLOSE:END
30 INPUT "DEPARTMENT";D$
40 INPUT "DATE HIRED";H$
50 PRINT#1,N$;",";D$;",";H$
60 PRINT:GOTO 20
```

Start the program with RUN and enter the following sample data in response to the prompts NAME, DEPARTMENT, and DATE HIRED:

```
NAME? MICKEY MOUSE
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? EBENEEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/26/78

NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78

NAME? DONE
```

Reading a Sequential File

The following program reads the sequential file created in the previous section and displays the names of all people hired in 1978.

```
10 OPEN "I",#1,"DATA"
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2)="78" THEN PRINT N$
40 GOTO 20
RUN
EBENEEZER SCROOGE
SUPER MANN
Input past end in 20
ok
```

When the program tries to INPUT# beyond the end of the file, an "Input past end" error occurs. To bring the program to an orderly conclusion, add the program line

```
15 IF EOF(1) THEN PRINT "File search complete":END
```

and change line 40 to

```
40 GOTO 15
```

It is always advisable to check for end of file before (not after) reading a record, just in case there are no records in the file at all.

Continuing a Sequential File

Although adding data to a file is essentially an output operation, you must not specify "O" or OUTPUT when opening the file, otherwise the existing file is destroyed. Instead, you should open the file for APPEND. Records subsequently written to the file are added to the existing records.

Inserting Records in a Sequential File

To insert data in an existing sequential file an additional, temporary sequential file is required.

1. OPEN the original file for input and the temporary file for output.

2. Read a record from the original file and write that record to the temporary file.

3. Repeat step 2, checking each time whether the current record is the one after which the record insertion is to take place. If this is so, proceed to step 4.

4. Write the record(s) for insertion to the temporary file.

5. Resume reading the original file, writing each record to the temporary file, until EOF is detected.

6. CLOSE both files.

7. Delete the original file (KILL). Then rename the temporary file to the name of the original file just deleted (NAME).

RANDOM FILES

The following statements, commands and functions are used with random files:

OPEN, CLOSE (the OPEN command can be written in two different ways, see Chapter 4)
FIELD - relates program variables to the file buffer
LSET, RSET — alignment of data in the buffer
MKI$, MKS$, MKD$ — convert numeric data to string form in preparation for writing to the file
CVI, CVS, CVD — convert string representations of numeric values read from the file
GET — reads a record from the disk into the file buffer
PUT — writes a file from the file buffer to disk
LOC, LOF — location in file, end of file.

Creating a Random File

Creation of a random file requires the following program steps.

1. OPEN the file for random access ("R" mode). This example specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes. Example:

   ```
   OPEN "R",#1, "FILE",32
   ```

   or

   ```
   OPEN "FILE" AS #1 LEN=32
   ```

2. Use the FIELD statement to allocate space in the random buffer for variables that will be written to the random file. Example:

   ```
   FIELD#1,20 AS N$, 4 AS A$,8 AS P$
   ```

3. Use the LSET statement to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions. MKI$ makes an integer value into a string, MKS$ makes an integer value into a single precision value, and MKD$ makes an integer value into a double precision value. Example:

   ```
   LSET N$=M$
   LSET A$=MKS$(AMT)
   P$=TEL$
   ```

4. Write the data from the buffer to the disk using the PUT command Example:

```
PUT #1,CODE%
```

The LOC function, with random access files, returns the "current record number." The current record number is one plus the last record number that was used for a GET or PUT statement. For example:

```
IF LOC(1)>50 THEN END
```

ends program execution if the current record number in file #1 is higher than 50.

The following program asks you to enter a record number (line 30). Your subsequent input is set up in the file buffer (lines 70 to 90) and written to the file "FILE" in line 100. This process is repeated until you enter a record number less than 1.

```
10 OPEN "R",#1,"FILE",32
20 FIELD#1,20 AS N$,4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";RECORD%
35 IF RECORD% <1 THEN CLOSE:END
40 INPUT "NAME";X$
50 INPUT "AMOUNT";AMT
60 INPUT "PHONE";TEL$:PRINT
70 LSET N$=X$
80 LSET A$=MKS$(AMT)
90 LSET P$=TEL$
100 PUT#1,RECORD%
110 GOTO 30
```

NOTE: Do not use a FIELDed string variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of into the random access file buffer.

Accessing a Random File

The initial steps for accessing an existing random file are the same as those for the original creation of the file. If the file is still open from previous use, these two steps are not required:

1. OPEN the file in "R" mode.

```
OPEN "R",#1, "FILE",32
```

or

```
OPEN "FILE" AS #1 LEN=32
```

2. Execute a FIELD statement to allocate space in the random buffer for the variables that will be read from the file.

```
FIELD#1,20 AS N$, 4 AS A$,8 AS P$
```

You can now read any record into the file buffer and then evaluate the contents of the buffer using the FIELDed variables:

3. Use the GET statement to move the desired record into the random buffer.

```
GET #1,RECORD%
```

4. The data in the buffer may now be accessed by the program. Numeric values must be converted back to numbers using the "convert" functions. CVI converts numeric values to integer values, CVS converts numeric values to single precision values, and CVD converts numeric values to double precision values.

```
PRINT N$
PRINT CVS(A$)
```

The following program gives you access to the data written to the disk file in the example given in the section "Creating a random file". All you have to do is enter the number of the record you wish to be displayed. You do not have to read the records one by one from the beginning of the file, as you would have to with a sequential file.

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1,20 AS N$,4 AS A$,8 AS P$
30 INPUT "2-DIGIT CODE";RECORD%
35 IF RECORD% <1 THEN CLOSE:END
40 GET#1,RECORD%
50 PRINT N$
60 PRINT USING "$$###.##";CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

A Sample Random Access Program

Here is an inventory program that illustrates random file access. In this program, the record number is used as the part number, and it is assumed the inventory will contain no more than 100 different part

numbers. Lines 900 through 960 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 140 through 210 display the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

```
110 REM INVENTORY
120 OPEN"R",#1,"INVEN.DAT",39
130 FIELD#1,1 AS F$,30 AS D$,2 AS Q$,2 AS R$,4 AS P$
140 PRINT:PRINT "Choose from:":PRINT
150 PRINT 1,"INITIALIZE FILE"
160 PRINT 2,"CREATE A NEW ENTRY"
170 PRINT 3,"DISPLAY INVENTORY FOR ONE PART"
180 PRINT 4,"ADD TO STOCK"
190 PRINT 5,"SUBTRACT FROM STOCK"
200 PRINT 6,"DISPLAY ALL ITEMS BELOW REORDER
    LEVEL"
205 PRINT 7,"END PROGRAM"
210 PRINT:PRINT:INPUT"Your choice";FUNCTION
220 IF (FUNCTION<1)OR(FUNCTION>7) THEN PRINT
    "Valid choices are 1 to 7":GOTO 140
230 ON FUNCTION GOSUB 900,250,390,480,560,680,245
240 GOTO 210
245 CLOSE:END
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC($)<>255 THEN INPUT"ENTER Y TO OVER-
    WRITE";A$ IF A$:<>"Y" THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESC$
300 LSET D$=DESC$
310 INPUT "QUANTITY IN STOCK";Q%
320 LSET Q$=MKI$(Q%)
330 INPUT "REORDER LEVER";R%
340 LSET R$=MKI$(R%)
350 INPUT "UNIT PRICE";P
360 LSET P$=MKS$(P)
370 PUT#1,PART%
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
```

```
410 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
420 PRINT USING "PART NUMBER###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND#####";CVI(Q$)
450 PRINT USING "REORDER LEVEL#####";CVI(R$)
460 PRINT USING "UNIT PRICE $$##.##";CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD";A%
520 Q% =CVI(Q$)+A%
530 LSET Q$=MKI$(Q%)
540 PUT#1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";S%
610 Q% =CVI(Q$)
620 IF (Q%-S%)<0 THEN PRINT "ONLY ";Q%;" IN
    STOCK":GOTO 600
630 Q% =Q%-S%
640 IF Q% =<CVI(R$) THEN PRINT "QUANTITY
    NOW";Q%; "REORDER LEVEL";CVI(R$)
650 LSET Q$=MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 REM DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I=1 TO 100
710 GET#1,I
720 IF CVI(Q$)<CVI(R$) THEN PRINT D$;"QUANTITY";
    CVI(Q$) TAB(50) "REORDER LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER"PART%
845 REM GET RECORD FOR PART
850 IF(PART% <1)OR(PART% >100) THEN PRINT "BAD
    PART NUMBER":GOTO 840 ELSE
    GET#1,PART%:RETURN
```

```
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE";B$:IF B$<>"Y" THEN
    RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
```

COMMUNICATIONS

This section describes the GW-BASIC program steps required to support RS-232 asynchronous communication with other computers and peripherals (with or without XON-XOFF Protocol).

OPENING A COMMUNICATIONS FILE

The OPEN "COM command allocates a buffer for input/output in the same manner as the OPEN for disk files. Refer to OPEN COM in Chapter 4.

COMMUNICATION I/O

Because the communications buffer is opened as a file, all input/output commands which are valid for disk files are valid for communications.

Communications sequential input commands are the same as those for disk files. They are:

```
INPUT#
LINE INPUT#
INPUT$
```

Communications sequential output commands are also the same as those for disk files. They are:

```
PRINT#
PRINT# USING
WRITE#
```

GET and PUT can be used for fixed length input/output. Obviously, you cannot specify a record number; instead, you state the number of bytes to be transferred either into or out of the file buffer (see GET, PUT and the LEN option in OPEN "COM, Chapter 4).

I/O Functions

The most difficult aspect of asynchronous communication is processing characters as fast as they are received. At rates above 1200 bps it may be necessary to suspend character transmission from the input device long enough for characters already received to be processed. This can be done by sending CHR$(19) (XOFF) and CHR$(17) (XON) to the computer or device transmitting data to your NCR PC. XOFF tells the input device to stop sending; XON tells it to resume sending.

There are three functions which help to determine when an "overrun" condition may occur:

LOC(x) Returns the number of characters in the input buffer which are waiting to be read. If more than 255 characters are in the buffer, LOC(x) returns 255. (The input buffer can hold more than 255 characters, as determined by the /C option when loading GW-BASIC.) If fewer than 255 characters remain in the buffer, LOC(x) returns the actual amount.

LOF(x) Returns the amount of free space in the input buffer. This is the same as the size of the buffer minus the value returned by LOC. The size of the communications buffer can be set by the /C option when loading GW-BASIC. The default size of the buffer is 256 bytes. Attempting to read data into a full buffer can cause a "Communication buffer overflow" error.

EOF(x) Returns true (-1) if the input buffer is empty; returns false (0) if there are any characters waiting to be read.

INPUT$ FUNCTION

As a recommendation, use the INPUT$ function instead of the INPUT# and LINE INPUT# statements when reading communications files, because it allows all characters read to be assigned to a string. INPUT# stops input when it detects a comma or <ENTER>.

INPUT$ returns a string of a specified number of characters read from a file specified by number. The following statements are efficient in reading a communications buffer:

```
10 WHILE NOT EOF(1)
20 A$=INPUT$(LOC(1),#1)
```

```
30 ...
40 ...
50 ...
60 WEND
```

If there are characters in the input buffer, the above statements return the characters in the buffer into A$ and process them (lines 30, 40, 50, etc.). If there are more than 255 characters, only 255 at a time will be returned to prevent a "String overflow" error. Further, if there are more than 255 characters, EOF(1) is false, and input into A$ continues until the buffer is empty.

NOTE: When developing a communications program, you should consider both the host computer's and satellite computer's baud rates. If a "Device I/O" error occurs, this usually indicates an overrun on the hardware interface, and you should adjust your program.

CONTROL SIGNALS

This paragraph contains information about control signals which you may need to know in order to communicate with another computer or pheripheral.

Output Signals

When you start GW-BASIC on your NCR PC, the Request To Send (RTS) and Data Terminal Ready (DTR) signal lines are not turned on until an OPEN"COM command is performed. You can suppress the RTS signal by specifying the RS option in the OPEN"COM. Unless suppressed, the line stays on until the communications file is closed by CLOSE, END, NEW, RESET, SYSTEM, or RUN without the R option. If an OPEN COM statement fails, the lines remain on. You may then retry the OPEN"COM without a prior CLOSE command.

Input Signals

If either the Clear To Send (CTS) or Data Set Ready (DSR) signal lines are off, you cannot perform an OPEN"COM. GW-BASIC returns a "Device Timeout" error after one second. You can, however, specify if and how you want these lines tested by using the CS and DS options in the OPEN"COM statement.

If the CTS or DSR line signals are off while a program is running, I/O commands associated with the communications file do not work, and a "Device Fault" or "Device Timeout" error occurs.

SAMPLE PROGRAM

The following program enables your NCR PC to be used as a conventional terminal. In addition to full-duplex communication, the program allows data to be down loaded (written) to a file, and conversely, a file may be up-loaded (transmitted) to another machine.

In addition to demonstrating the elements of asynchronous communications, this program should be useful in transferring GW-BASIC programs and data to and from the NCR PC.

Notes on the Sample Program

Line No.	Comments
	When starting GW-BASIC, set the /F option to 3. There is no need to set the /C option.
10	Sets the screen to character mode.
20	Turns off the programmable function key display, clears the screen, and makes sure that all files are closed.

NOTE: Asynchronous implies character I/O as opposed to line or block I/O. Therefore, all PRINTs (either to the communications file, the screen, or a disk file) are terminated with a semicolon (;). This stops the <ENTER> normally issued at the end of a PRINT statement.

30	Defines all numeric variables as integers. This is primarily for use in the subroutine at lines 500-660. Any program looking for speed optimization should use integer counters in loops wherever possible.
35-40	Clears the 23rd line starting at column 1.
50	Defines Boolean true and false.
70	Defines the ASCII XON and XOFF characters.
100-130	Prints program identification and asks for baud rate (speed). Opens communications to file number 1 with even parity, 7 data bits, and a line feed (LF) following every <ENTER>.

200-280 This section gives you a menu for receiving data at your screen or on a file, or for transmitting data from your keyboard or from one of your files.

1. You are asked how many characters have to be received on your communications line before they are displayed on the screen.

2. Reads one or more characters from the keyboard into A$ and transmits A$. You are guided by the menu to continue.

3. If only a space is entered, wait for n characters and print them when received.

4. If the character was M only, then the user is ready to down-load a file, so get file name.

5. If you entered an E only, the program will stop at 9000-9040.

6. If the input (A$) is not M, E, or space, send it by writing to the communications file (PRINT #1. . .). as described in step 2, and at line 230 go back to menu.

7. At lines 250-260, read and display contents of communications buffer (as much as selected by n) on screen. Continue with 1.

300-310 Get disk file name to be used.

400-430 Asks if file name is to be transmitted (up-loaded) or received (down-loaded) and opens file.

490-540 The received data will fill an array of 126 positions unless an end-of-file character (line 530) was received, which closes the file.

550-620 Before writing to the selected disk file, an XOFF is sent to the transmitter. Two additional characters (lines 560-590) may be read after the 126 positions are filled and before the transmitter gets the XOFF.

625 When the array is completely written to disk file and XON is sent to the transmitter, the transmitter continues sending.

630 Continue receiving as at line 500.

640-680 For end-of-file, write last characters to file and close it. Continue again at the menu.

800-880 This is a waiting routine used when the transmitter also receives characters. If the transmitter receives an XOFF, wait until XON is received before continuing transmission.

1000-1060 This is a transmit routine. Until the end of the disk file:

Read one character into A$ with INPUT$ function. Send character to communications device in 1015. If a character is received, the waiting routine for XON in case of XOFF is called, line 1015.) Send a <Ctrl-Z> at the end-of-file in line 1040 in case the receiving device needs one to close its file. Finally, in lines 1050 and 1060, close disk file, print completion message, and go back to conversation mode in line 200.

9000-9040 These lines are run if you enter E in response to the menu. They close the communications file and the screen output file, restore the programmable function key display, and end the program.

```
10 SCREEN 0:WIDTH 80
20 KEY OFF:CLS:CLOSE
30 DEFINT A-Z
35 LOCATE 23,1
40 PRINT STRING$(60," ")
50 FALSE=0:TRUE= NOT FALSE
70 XOFF$=CHR$(19):XON$=CHR$(17)
100 LOCATE 23,1:PRINT "Async TTY Program ":
110 LOCATE 1,1:LINE INPUT "speed?";SPEED$
120 REM
130 OPEN"COM1:"+SPEED$+",E,7,,LF LC AS #1
140 OPEN "scrn:" FOR OUTPUT AS #2
```

```
200 LOCATE 1,1:LINE INPUT "on receiving, how many characters
    are to be read";N$
203 N% = VAL(N$)
205 LOCATE 3,1:PRINT "press any keys for transmission"
206 PRINT "except: M      for file i/o"
207 PRINT "or         space for receiving"
208 PRINT "or         E       for ending program"
209 LINE INPUT;A$
210 IF A$=" " THEN 250
211 IF A$="M" THEN 300
212 IF A$="E" THEN 9000
220 PRINT #1,A$;
230 GOTO 200
250 A$=INPUT$(N%,#1)
260 PRINT #2,A$;
280 GOTO 200
300 LOCATE 8,1
310 LINE INPUT"file? "DSKFIL$
400 LOCATE 9,1
410 LINE INPUT"(T)ransmit or (R)eceive?";TXRX$
420 IF TXRX$="T" THEN OPEN DSKFIL$ FOR INPUT AS
#3:GOTO 1000
430 OPEN DSKFIL$ FOR OUTPUT AS #3
490 DIM BUF$(128)
500 FOR J=1 TO 126
520 BUF$(J)=INPUT$(1,#1)
530 IF BUF$(J)=CHR$(26) THEN GOTO 640 'checks for Ctrl-Z
540 NEXT J
550 PRINT #1,XOFF$;
560 IF LOC(1)=0 THEN K=126:GOTO 600
570 BUF$(127)=INPUT$(1,#1)
580 IF LOC(1)=0 THEN K=127:GOTO 600
585 BUF$(128)=INPUT$(1,#1)
590 K=128
600 FOR I=1 TO K
610 PRINT #3,BUF$(I);
620 NEXT I
625 PRINT #1,XON$;
630 GOTO 500
640 FOR I=1 TO J
650 PRINT #3,BUF$(I);
```

```
660 NEXT I
670 CLOSE #3:CLS:LOCATE 24,10:PRINT "* download complete
*"
680 GOTO 200
800 B$=INPUT$(1,#1)
810 IF B$=XOFF$ THEN GOTO 850
820 PRINT #2,B$;
830 IF LOC(1)=0 THEN RETURN
840 GOTO 800
850 B$=INPUT$(1,#1)
860 IF B$=XON$ THEN RETURN
870 PRINT #2,B$;
880 GOTO 850
1000 WHILE NOT EOF(3)
1010 A$=INPUT$(1,#3)
1012 IF VAL(SPEED$)>4000 THEN FOR I=1 TO 10:NEXT
1015 PRINT #1,A$;
1020 IF LOC(1)>0 THEN GOSUB 800
1030 WEND
1040 PRINT #1,CHR$(26); 'ctrl-Z to close file.
1050 CLOSE #3:CLS:LOCATE 23,10:PRINT "** upload complete
***";
1060 GOTO 200
9000 CLOSE #1
9010 CLOSE #2
9030 KEY ON
9040 END
```

Running Machine Language

This chapter is intended for the machine language (assembler) programmer who wishes to use machine language routines from within a GW-BASIC program. You will find information about where and how you can reserve memory for these routines, how to load them into memory, and how GW-BASIC can pass parameters to and read results from these routines.

Your NCR PC contains an 8088 microprocessor. There is a wealth of available literature on programming with the 8086 family of microprocessors, to which the 8088 belongs, including publications by Intel Corporation.

RESERVING MEMORY

GW-BASIC uses up to 64 KB of computer memory. Not only your program is stored in this area, but also the variables it sets up. Furthermore, space is required for GW-BASIC to interpret your program commands and carry out calculations. Depending on what other files, apart from GW-BASIC, are currently held in memory by NCR-DOS, you can use memory above that kept aside by NCR-DOS for GW-BASIC. Alternatively, you can use part of the GW-BASIC memory area.

To use memory outside the GW-BASIC area for machine language subroutines, define the starting address of an area where you wish to load the subroutine using the DEF SEG statement. You can then refer to this area using offset values from within the GW-BASIC program. This does not actually protect this area from being overwritten by other applications running under NCR-DOS, nor does it prevent you from accidentally writing your subroutines to an area of memory where they can upset GW-BASIC or the operating system. For this reason you should specify an area you wish to reserve by means of the second parameter in the /M option when loading GW-BASIC (see the section "How to Start-Up GW-BASIC" in Chapter 1). For example,

```
GWBASIC /M:,4112
```

allows GW-BASIC 64 KB of memory, and reserves 256 bytes of memory immediately above GW-BASIC for your use.

An alternative method of reserving memory for subroutines is to put aside space within the GW-BASIC area. You can use the /M option when loading GW-BASIC. For example,

```
GWBASIC /M:65000
```

takes away from the top of the GW-BASIC area a little more than 500 bytes and places them at your disposal. To reserve space dynamically within a GW-BASIC program, use the CLEAR command, for example

```
10 CLEAR ,65000
```

You may specify hexadecimal instead of decimal values in any of these methods, using the &H prefix.

USING RESERVED MEMORY

You can use the memory you have reserved to store any kind of information you want. For example, if you have a very long series of integer numbers of which none is greater than 255, you could POKE them one by one into your reserved area. This saves memory as an integer array would require two bytes for each element. You can then read the individual numbers by means of the PEEK function.

You can apply POKE to a series of bytes which go to make up a machine language routine, or you can BLOAD these bytes from a disk file.

POKEing

Write the byte values for the machine language instructions in GW-BASIC DATA lists. You will probably prefer to use hexadecimal values prefixed by &H.

State in a DEF SEG command the memory address of the first byte to which a DATA item is to be written.

Using a FOR...NEXT loop with its control variable starting with zero and STEPping up by one until the number of DATA items is exhausted, READ each DATA item into an integer variable and POKE that value using the current value of the control variable as the address to be POKEd.

As an alternative, you can store the byte values in an integer array and POKE the elements of the array one by one.

The methods described are suitable for coding relatively short subroutines.

BLOADing

If you are doing extensive machine language programming, you are probably using a symbolic or macro assembler and then producing an .EXE file by means of the NCR-DOS linker.

If you write a truly relocatable subroutine, that is, a program which can execute from any memory address, you need not be concerned about loading it to an address which differs from the one intended by the linker. You can use BLOAD following a DEF SEG statement, or, if the subroutine is to be situated not more than 64 KB above the start of the GW-BASIC program area, you can load it to an address specified in terms of the offset to the beginning of that area.

If the BLOADing address of your machine language subroutine is dependent on the location determined by the linker, you must first ascertain where the operating system wants that subroutine to be loaded. This requires use of the DEBUG utility which is described in your *NCR-DOS PROGRAMMER'S MANUAL*.

1. Make sure that the subroutine was linked for loading at the HIGH end of memory.

2. Load DEBUG, including GWBASIC.EXE in the command line as the file to be loaded under DEBUG. Display and note the contents of the registers. Then load the .EXE file produced by the linker; display and note the contents of CS and the Instruction Pointer.

3. Restore the registers to the state prior to loading the .EXE file and, still under DEBUG, start execution of GW-BASIC (not the subroutine) using the DEBUG G command.

4. Load your GW-BASIC program from which the subroutine is to be called. Edit the program so that the value of the CS register noted after loading the .EXE file is in the DEF SEG statement. The DEF USR or CALL statement should refer to the address contained in the Instruction Pointer as noted after loading the .EXE file.

5. In direct mode set DEF SEG in accordance with the CS value noted after loading the .EXE file. Then BSAVE the subroutine, specifying the offset as the contents of the Instruction Pointer noted after loading the .EXE file.

6. The BLOAD command in your GW-BASIC program which loads the subroutine need not specify the offset at which it is to be located. GW-BASIC assumes the offset value used in the BSAVE command for that file.

It is possible to locate your machine language subroutines within the GW-BASIC memory area. Possible locations are an unused file or screen buffer, or a string variable. You can find out the location of a file buffer or a string variable by means of VARPTR# and VARPTR, respectively.

HOW GW-BASIC CALLS SUBROUTINES

Your GW-BASIC program can call machine language subroutines by means of the CALL command and the USR function. Regardless of which method you are using, the DS, ES, and SS processor registers are all set to the address of the GW-BASIC data area upon entry to the subroutine. The CS register contains the value specified in the most recent DEF SEG command. If none has been executed, or DEF SEG was executed without a specified value, CS is set to the same address as the other segment registers.

The stack available to the subroutine can accommodate up to 8 PUSHes. If more are required, a separate stack must be set up.

GW-BASIC regards all machine language routines as far procedures. Therefore, an intersegment RET instruction should conclude the subroutine. The segment registers and the Stack Pointer must be restored before returning to GW-BASIC. It is therefore important that the subroutine note the values of these five registers before altering their contents.

If the subroutine disables interrupts, they must be enabled before the return to GW-BASIC.

CALL

Upon execution of the CALL statement GW-BASIC does the following:

The address (offset to GW-BASIC's data area) of each variable specified in the CALL statement is PUSHed onto the stack. Using

these addresses the subroutine can accept data from and return data to the GW-BASIC program. If the variable is a string variable, the address on the stack is that of a 3-byte string descriptor. The first byte contains the length of the string (0 to 255), the second byte contains the eight least significant bits of the offset of the string in GW-BASIC's data area, the eight most significant bits are stored in the third byte. The subroutine must not alter the length of the string.

If your subroutine is to influence the content of a string variable, it is a good idea to ensure that GW-BASIC first copies the string variable into its own workspace by performing an operation on the string. For example, if your subroutine is to return a value to A$, issue the command

```
10 A$="String long enough?"+""
```

If you do not include such a command, the string descriptor points to the occurrence of the string in your program text. This could lead to an unwanted modification of the program.

A return address specified in the CS register and the offset are likewise PUSHed.

Processor control is passed to the subroutine using the contents of the last DEF SEG and the offset value specified in the CALL command. The stack entry for the last variable in the parameter list is now 6 bytes above the current Stack Pointer value, the entry for the parameter before the last one is 8 bytes above the current Stack Pointer value, and so on.

The assembler RET instruction at the end of the subroutine must specify a value which is two times the number of items in the CALL variable list.

The following example shows a simple arithmetic operation performed in an assembler subroutine. The two numbers for the subtraction are passed by the GW-BASIC program in the integer variables I% and J%, the result is returned to R%.

GW-BASIC program CALL command:

```
CALL SUBTR (I%,J%,R%)
```

The assembler subroutine:

```
CSEG    SEGMENT
        ASSUME CS:CSEG
```

```
        ;
SUBTR   PROC FAR
        ;
        PUSH BP
        MOV BP,SP
        MOV SI,[BP]+10 ;address of I% in SI
        MOV DX,[SI]    ;value I% in DX
        MOV SI,[BP]+8  ;address of J% in SI
        MOV AX,[SI]    ;value J% in AX
        SUB DX,AX
        MOV SI,[BP]+6  ;points to memory location of R%
        MOV [SI],DX    ;puts two byte result in R%
        POP BP
        RET 6
        ;
SUBTR   ENDP
CSEG    ENDS
```

USR

You can enter a subroutine by means of the USR function. A single parameter can be passed which can be any constant or variable. If a parameter is not required by the subroutine, the GW-BASIC command calling upon the USR function must specify a dummy parameter.

Upon entry to the subroutine the AL register contains the value 2 for a two byte integer in two's complement notation, 3 for a string, 4 for a single precision number, or 8 for a double precision number.

If the parameter is a string, the DX register points to a 3 byte string descriptor. The first byte contains the length of the string (0 to 255), the second byte contains the eight least significant bits of the offset of the string in GW-BASIC's data area, the eight most significant bits are stored in the third byte.

If your subroutine is to influence the content of a string variable, it is a good idea to ensure that GW-BASIC first copies the string variable into its own workspace by performing an operation on the string. For example, if your subroutine is to return a value to A$, issue the command

```
10 A$="String long enough?"+""
```

If you do not include such a command, the string descriptor points to the occurrence of the string in your program text. This could lead to an unwanted modification of the program.

If the parameter is a number, the value is placed in the 8 byte Floating Point Accumulator in the GW-BASIC data area. The BX register then points to the fifth byte of the FAC.

- If the number is an integer, the fifth and sixth byte of the FAC contain the least significant and the most significant bits of the number, respectively.

- If the number is a single precision number, the last byte of the FAC contains the exponent minus 128. The fifth, sixth and seventh bytes contain the mantissa: the least significant bit is bit 0 of the fifth byte, the most significant bit is bit 6 of the seventh byte. Bit 7 of the seventh byte indicates a positive number with 0, a negative number with 1. The mantissa is to be understood as having a leading 1, the exponent as a whole number.

- The structure of the FAC for a double precision number is the same as that for a single precision number, with the difference that the mantissa occupies all the first seven bytes (bit 0 of the first byte is the least significant bit).

The result returned by the USR function call is the contents of the BX register.

For PEEKers and POKErs

This chapter presents information as to the way GW-BASIC makes use of the hardware facilities of your NCR PC. The GW-BASIC memory map and information about the structure of variables is also included.

Programming with GW-BASIC does not mean that you have to read this Chapter. Controlling the hardware is a task GW-BASIC entrusts to "drivers". Drivers are programs hidden in GW-BASIC or the operating system which convert the GW-BASIC commands you issue into detailed machine instructions.

For example, let us assume that GW-BASIC encounters the command CLS somewhere in your program. First, the GW-BASIC "interpreter" checks that the term CLS is included in its dictionary of commands. Then it calls upon an internal routine driving the screen display, with the effect that the screen pixels are set one by one to the background color. The comfort of CLS and the other GW-BASIC commands and functions is that you do not have to be concerned about how the pixels are changed. You need not even be aware of the fact that there is a copy of screen contents in random access memory.

If you want to know where the screen buffers are and other facts related to the hardware of your NCR PC, or if you are curious about the way GW-BASIC uses the memory allotted to it, then read on. You can apply the information in this chapter to the GW-BASIC PEEK and IN functions, and the POKE and OUT commands.

PEEK allows you to ascertain the value of an individual byte in memory; IN lets you observe the way the microprocessor receives information from the machine ports (for example, status signals from a printer). POKE allows you to influence individual memory bytes, OUT is used for writing information to machine ports. POKE and OUT give you immense power over your computer, but they require careful use, otherwise your computer might behave in a strange and unpredictable way.

NCR offers a *SYSTEM TECHNICAL* manual for your PC. These contain detailed information about the way the hardware and the software drivers operate. The examples given in this chapter are just some of the effects you can achieve when by-passing GW-BASIC.

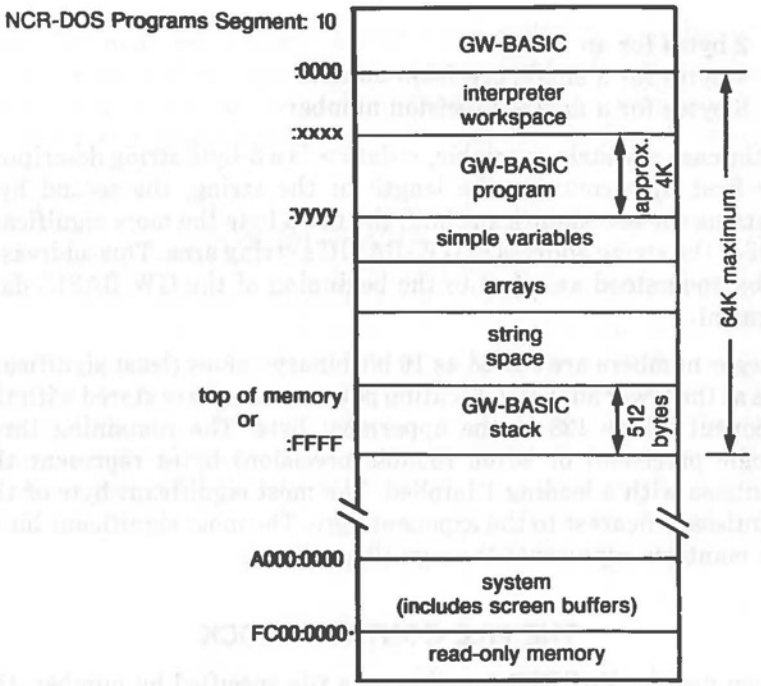
GW-BASIC AND PC MEMORY

NCR-DOS loads GW-BASIC into a program segment just as it loads other .COM and .EXE files. The absolute machine address of the program segment is of no consequence: the value for GW-BASIC's data area (segment), that is, the value set or confirmed by DEF SEG without parameters, is automatically assigned by NCR-DOS. The following diagram shows the state of memory for the program segment immediately after loading GW-BASIC. Where a paragraph value is not specified to the left of the colon, the value is the paragraph address of the GW-BASIC data segment.

Notes:

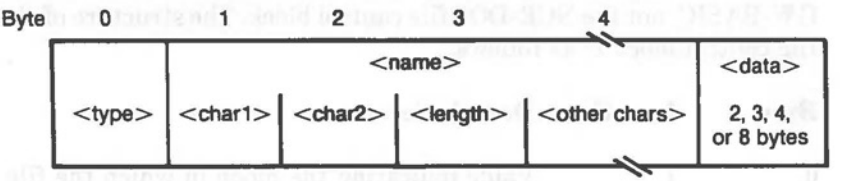
- The offset values xxxx and yyyy are stored at the locations 30H-31H and 358H-359H, respectively. In both cases, the lower byte is the less significant.

- The size of the GW-BASIC stack can be set by means of the CLEAR command.



VARIABLES

Variables are stored by GW-BASIC as follows:



<type> identifies the type of variable as follows:

2 integer
3 string
4 single precision
8 double precision

<name> is the name of the variable. The first two characters of the name are stored in <char1> and <char2>, <length> states how many more characters are in the variable name. These <other chars> start at byte 4.

Immediately after the last character of the name is the first byte of the actual <data> contained in the variable. This is the position to which the VARPTR function points. The length of this data is

2 bytes for an integer
4 bytes for a single precision number
8 bytes for a double precision number

In the case of a string variable, <data> is a 3-byte string descriptor: the first byte contains the length of the string, the second byte contains the less significant half, the third byte the more significant half of the string address in GW-BASIC's string area. This address is to be understood as offset to the beginning of the GW-BASIC data segment.

Integer numbers are stored as 16 bit binary values (least significant bits at the lower address). Floating point numbers are stored with the exponent minus 128 in the uppermost byte. The remaining three (single precision) or seven (double precision) bytes represent the mantissa with a leading 1 implied. The most significant byte of the mantissa is nearest to the exponent byte. The most significant bit of the mantissa represents the sign (0: positive).

THE FILE CONTROL BLOCK

If you use the VARPTR function on a file specified by number, the value returned is the address of the first byte of the file control block for that file. This address represents an offset to the beginning of the GW-BASIC data area. It is important to remember that this is a GW-BASIC, not the NCR-DOS file control block. The structure of the file control block is as follows:

Byte	Length	Description
0	1	Value indicating the mode in which the file was opened: 1 - Input only 2 - Output only 4 - Random 16 - Append only
1	38	NCR-DOS file control block
39	2	The number of sectors read or written for sequential access files. 1 plus the last record number read or written for random access files.

41	1	Number of bytes in sector when read or written.
42	1	Number of bytes left in input buffer.
43	3	(reserved)
46	1	Device number: 0, 1 - Disk drives A: and B: 248 - LPT3: 249 - LPT2: 250 - COM2: 251 - COM1: 253 - LPT1: 254 - SCRN: 255 - KYBD:
47	1	Device width
48	1	Position in buffer for PRINT#.
49	1	Internal use during LOAD and SAVE. Not used for data files.
50	1	Output position used during tab expansion.
51	128	Physical data buffer. Used to transfer data between NCR-DOS and GW-BASIC. Use this offset to check data in sequential I/O mode.
179	2	Variable length record size. Default: 128. Set by length parameter in OPEN command.
181	2	Current physical record number.
183	2	Current logical record number.
185	1	(reserved)
186	2	Disk files only. Position for PRINT#, INPUT#, and WRITE#.

188	n	Actual FIELD data buffer. Size n is determined by the /S option when loading GW-BASIC. Use this offset to examine file data in random access mode.

THE KEYBOARD

The keyboard buffer can store up to 15 characters. Attempting to enter more causes your NCR PC to beep.

You can clear the keyboard buffer with

```
DEF SEG=0
```

followed by

```
POKE 1050,PEEK(1052)
```

The DEF SEG command is required so that PEEK and POKE refer to absolute addresses counting from the start of physical RAM.

A subsequent

```
DEF SEG
```

restores the segment value to GW-BASIC's data segment.

SETTING SCREEN ATTRIBUTES

The memory area from the paragraph:offset of A000:0000 to B000:FFFF is used for various screen buffers.

CHARACTER DISPLAY MEMORY

The area from B000:0000 to B000:7FFF contains the 8 screen pages which are supported in character mode with a monochrome display adapter. With the line width 80, the default display page occupies the 4000 bytes starting at B000:0000. The next page starts on the 1000H boundary at B000:1000, and so on. Characters are stored at even addresses. The attribute byte for a character position is the odd byte immediately above the character byte.

The attribute byte is made up as follows (values are decimal):

```
Normal video is active when only bits 0,1,2 are set -->
        value 7
Inverse video is active when only bits 4,5,6 are set -->
```

value 112

Add 8 to the value for high intensity and/or add 128 for blinking.

If bits 0,1,2,4,5,6 are all set or all zero, there is no contrast between writing and background.

The color graphics display adapter has its buffers from B000:8000 to B000:FFFF. The bytes up to B000:8FFF hold the default display page in character mode.

Display colors are each made up of a combination of red, green, and blue. These are the colors produced by the three color guns of the cathode ray tube inside your computer. The color combination of each of the 8 basic colors available in character modes is as follows:

Black - none
Blue - Blue only
Green - Green only
Cyan - Green and Blue
Red - Red only
Magenta - Blue and Red
Brown - Red and Green
White - Red, Green, and Blue

The use of alternating addresses for character and attribute is as with the monochrome display adapter. The difference is that the attribute value has a color effect on the screen:

```
Writing color -          bit:     0       1       2
                                blue    green   red

Background color -       bit:     4       5       6
                                blue    green   red

Bit 3 gives the high intensity colors when set.
Bit 7 set yields character blinking.
```

Example:

```
DEF SEG=&HB000:POKE &H8F9E,&H39:POKE &H8F9f,132
```

produces a blinking red digit 9 in the bottom right corner of the screen, when character mode with WIDTH 80 is in force.

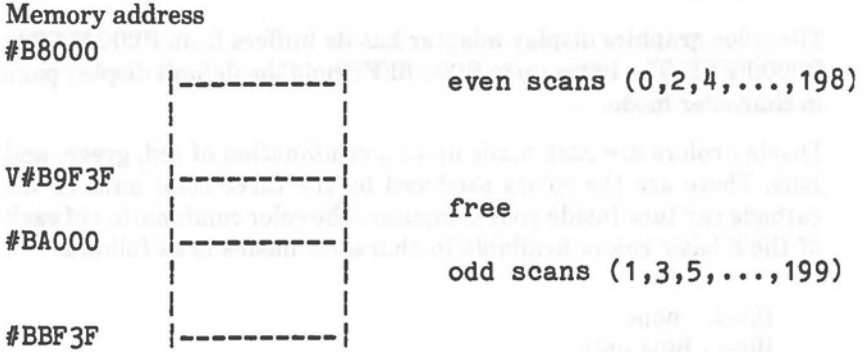
```
DEF SEG=&HB000:POKE &H87CE,&H39:POKE &H87CF,132
```

has the same effect in character mode with WIDTH 40.

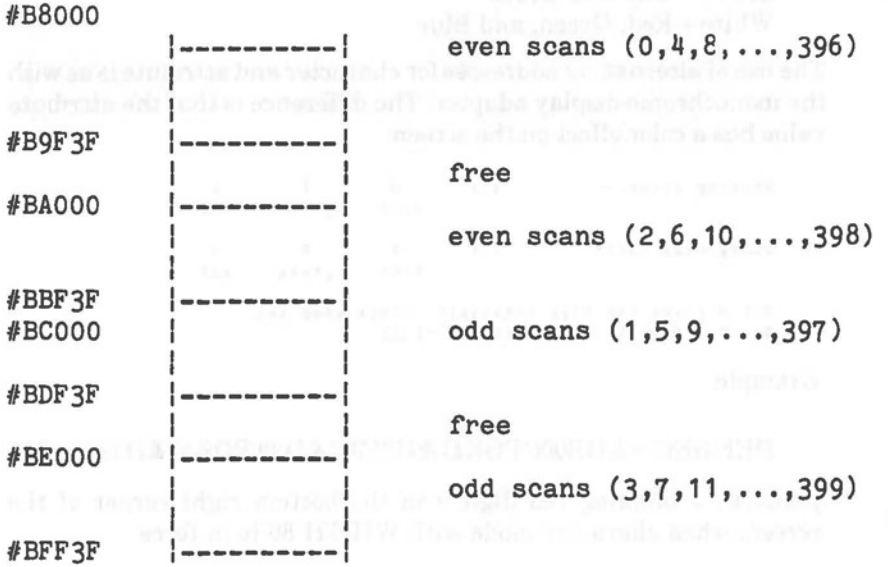
GRAPHICS DISPLAY MEMORY

The screen image is built up in two scans, each running from the top to the bottom of the screen.

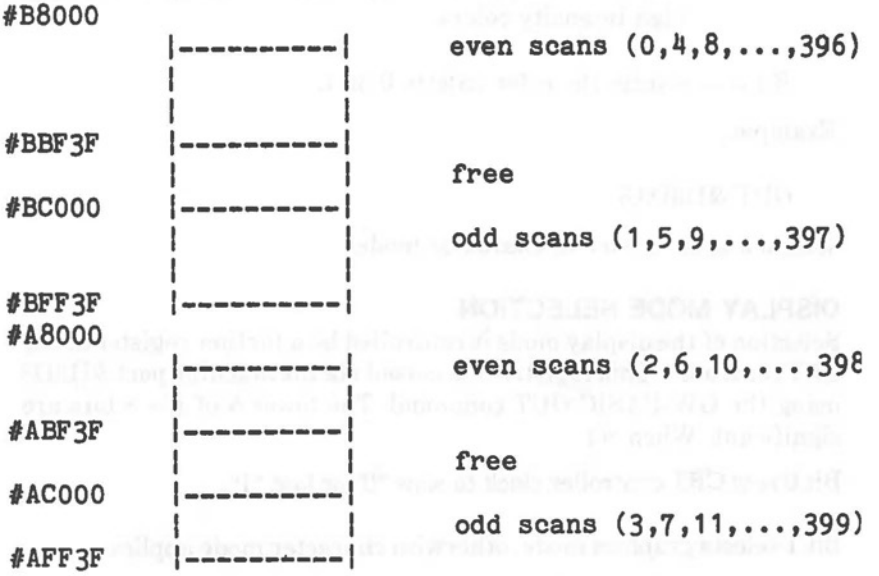
Storage Map for Low and Medium Resolution Graphics



Storage Map for High Resolution Black-and-White Graphics



Storage Map for high Resolution Color Graphics



In low and high resolution color graphics, each byte can be regarded as consisting of four bit pairs. Each bit pair contains a binary value 0 to 3, representing the color to be plotted for a screen point (see COLOR in Chapter 4).

In medium and high resolution black-and-white graphics, each bit represents one screen point.

COLOR SELECTION

The integrated circuit which looks after the screen display contains a register where it notes color attributes in terms of the three color guns of the cathode ray tube. You can address this register via the machine port &H3D9, using the GW-BASIC OUT command. The lower 6 of the 8 bits in this register are significant. When set

Bit 0 — activates the blue gun for background in graphics mode

Bit 1 — activates the green gun for background in graphics mode

Bit 2 — activates the red gun for background in graphics mode

Bit 3 — displays in high intensity
the background color in graphics mode

Bit 4 — selects the high intensity colors for background in character mode; in the graphics modes, too, it selects the high intensity colors.

Bit 5 — selects the color palette 0 or 1.

Example:

```
OUT &H3D9,3
```

selects a cyan border in character mode.

DISPLAY MODE SELECTION

Selection of the display mode is controlled by a further register of the CRT controller. This register is accessed via the machine port &H3D8 using the GW-BASIC OUT command. The lower 6 of the 8 bits are significant. When set

Bit 0 sets CRT controller clock to slow "0" or fast "1".

Bit 1 selects graphics mode, otherwise character mode applies.

Bit 2 selects a black and white display, otherwise a color display is selected.

Bit 3 enables the screen display. While the display mode is being changed, this bit should be 0.

Bit 4 selects medium and high resolution black-and-white graphics.

Bit 5 ensures that the blinking attribute can be used by GW-BASIC. If you reset this bit to 0, blinking can no longer be achieved, but you then use the high intensity colors for background, in addition to the normal intensity colors.

Bit 6 selects 400 pixel lines for high resolution graphics.

Bit 7 enables display of page 1 or 2 ("0") and page 3 or 4 ("1") in low and medium resolution.

Example:

```
OUT &HD38,9
```

When executed in character mode, this command replaces the blinking facility by the extended range of background colors.

The following command sequence switches over to a monochrome screen:

```
10 DEF SEG=0
20 POKE &H410,(PEEK(&H410) OR &H30)
30 SCREEN 0:WIDTH 40:WIDTH 80
```

The underline cursor is then created by

```
40 LOCATE ,,1,12,13
```

To switch over to a color screen:

```
10 DEF SEG=0
20 POKE &H410,(PEEK(&H410) AND &HCF) OR &H10
30 SCREEN 1,0,0,0
40 SCREEN 0:WIDTH 40
50 LOCATE ,,1,6,7
```

Line 50 sets up the underline cursor.

The following commands enable you to specify the value 1,2, or 3 in COL% as the foreground color:

```
10 DEF SEG
20 POKE &H4E,COL%
```

THE CHARACTER SET

You can examine and reproduce the bit patterns of the standard ASCII part of the GW-BASIC character set. These characters are stored in the read only memory (ROM) of your NCR PC. The ROM is located at the memory address F000:C000 (paragraph:offset). The 8 by 8 pixel character set starts at F000:FA6E and occupies addresses up to F000:FE6D. The 8 by 16 character set resides in memory from F000H:D000H to F000H:D7FFH. The bit pattern for each character is held in eight adjacent bytes. The ASCII character for value 0 occupies the first eight bytes, the ASCII character for value 1 occupies the next eight bytes, and so on.

The following program reads the bit pattern of an 8 by 8 character you enter, and displays a corresponding pattern of dots on the screen:

```
10 DEFINT A-Z
20 OPTION BASE 1
30 DIM PATT(8)
40 DEF SEG=&HF000
50 INPUT "Character":CH$
60 FOR X=1 TO 8
70 PATT(X)=PEEK(ASC(CH$)*8+X+$HFA6D)
80 NEXT X
90 CLS
100 FOR X=1 TO 8
110 BYTE=PATT(X)
120 SHTF=256
140 FOR Y=1 TO 8
150 SHFT=SHFT/2
160 IF INT(BYTE/SHFT)=1 THEN BYTE=BYTE-
    SHFT:PRINT CHR$(249); ELSE PRINT " ";
170 NEXT Y
175 PRINT
180 NEXT X
```

Reserved Words

Reserved words are words recognized by GW-BASIC as belonging to commands and functions. For this reason, you cannot use them as names for variables (appending %, !, #, or $, as a type declaration does not change this). A reserved word may, however, be part of a variable name. For example, you cannot use the word AND$ as the name of a variable, but SAND$ and CANDY$ are allowed.

The GW-BASIC reserved words are:

ABS	COMMON	END
AND	CONT	ENVIRON
ASC	COS	ENVIRON$
ATN	CSNG	EOF
AUTO	CSRLIN	EQV
BEEP	CVD	ERASE
BLOAD	CVI	ERDEV
BSAVE	CVS	ERDEV$
CALL	DATA	ERL
CDBL	DATE$	ERR
CHAIN	DEF	ERROR
CHDIR	DEFDBL	EXP
CHR$	DEFINT	FIELD
CINT	DEFSNG	FILES
CIRCLE	DEFSTR	FIX
CLEAR	DELETE	FNxxxxxxxx
CLOSE	DIM	FOR
CLS	DRAW	FRE
COLOR	EDIT	GET
COM	ELSE	GOSUB

GOTO
HEX$
IF
IMP
INKEY$
INP
INPUT
INPUT#
INPUT$
INSTR
INT
IOCTL
IOCTL$
KEY
KEY$
KILL
LCOPY
LEFT$
LEFT$
LEN
LET
LINE
LIST
LLIST
LOAD
LOC
LOCATE
LOF
LOG
LPOS
LPRINT
LSET
MERGE
MID$
MKDIR
MKD$
MKI$
MKS$
MOD
MOTOR
NAME
NEW
NEXT
NOT
OCT$
OFF
ON
OPEN
OPTION
OR
OUT
PAINT
PEEK
PEN
PLAY
PMAP
POINT
POKE
POS
PRESET
PRINT
PRINT#
PSET
PUT
RANDOMIZE
READ
REM
RENUM
RESET
RESTORE
RESUME
RETURN
RIGHT$
RMDIR
RND
RSET
RUN
SAVE
SCREEN
SGN
SHELL
SIN
SOUND
SPACE$
SPC(
SQR
STEP
STICK
STOP
STR$
STRIG
STRING$
SWAP
SYSTEM
TAB(
TAN
THEN
TIME$
TIMER
TO
TROFF
TRON
USING
USR
VAL
VARPTR
VARPTR$
VIEW
WAIT
WEND
WHILE
WIDTH
WINDOW
WRITE
WRITE#
XOR

The Character Set

This appendix consists of a list of the characters available to GW-BASIC. For each character, the decimal and hexadecimal equivalent of the ASCII code value is given. For example, the code value for the question mark is 63 (3F in hexadecimal) so

```
PRINT CHR$(63);
```

displays a question mark on the screen. Where the character is a word in parentheses, it is not a displayable character as such, although it may have an effect on the screen display (e.g. cursor movement).

If you know ASCII code, you will notice that the first 32 items in the list include graphic symbols which replace the standard interpretation of these values as control and communications functions.

When editing a program, you can produce on the screen a character for which there is no key on your keyboard by entering the three digit code while the Alt key is depressed. In this way, you can include non-keyboard characters in a string constant.

The values stated in the column &H of the list are the hexadecimal equivalents of the decimal codes.

ASCII Decimal	&H	Character	Control character	ASCII Decimal	&H	Character
000	00	(null)	NUL	032	20	(space)
001	01	☺	SOH	033	21	!
002	02	☻	STX	034	22	"
003	03	♥	ETX	035	23	#
004	04	♦	EOT	036	24	$
005	05	♣	ENQ	037	25	%
006	06	♠	ACK	038	26	&
007	07	(beep)	BEL	039	27	'
008	08	■	BS	040	28	(
009	09	(tab)	HT	041	29	)
010	0A	(line feed)	LF	042	2A	*
011	0B	(home)	VT	043	2B	+
012	0C	(form feed)	FF	044	2C	,
013	0D	(carriage return)	CR	045	2D	-
014	0E	♫	SO	046	2E	.
015	0F	☼	SI	047	2F	/
016	10	►	DLE	048	30	0
017	11	◄	DC1	049	31	1
018	12	↕	DC2	050	32	2
019	13	‼	DC3	051	33	3
020	14	¶	DC4	052	34	4
021	15	§	NAK	053	35	5
022	16	▬	SYN	054	36	6
023	17	↨	ETB	055	37	7
024	18	↑	CAN	056	38	8
025	19	↓	EM	057	39	9
026	1A	→	SUB	058	3A	:
027	1B	←	ESC	059	3B	;
028	1C	(cursor right)	FS	060	3C	<
029	1D	(cursor left)	GS	061	3D	=
030	1E	(cursor up)	RS	062	3E	>
031	1F	(cursor down)	US	063	3F	?

ASCII Decimal	&H	Character	ASCII Decimal	&H	Character
064	40	@	096	60	`
065	41	A	097	61	a
066	42	B	098	62	b
067	43	C	099	63	c
068	44	D	100	64	d
069	45	E	101	65	e
070	46	F	102	66	f
071	47	G	103	67	g
072	48	H	104	68	h
073	49	I	105	69	i
074	4A	J	106	6A	j
075	4B	K	107	6B	k
076	4C	L	108	6C	l
077	4D	M	109	6D	m
078	4E	N	110	6E	n
079	4F	O	111	6F	o
080	50	P	112	70	p
081	51	Q	113	71	q
082	52	R	114	72	r
083	53	S	115	73	s
084	54	T	116	74	t
085	55	U	117	75	u
086	56	V	118	76	v
087	57	W	119	77	w
088	58	X	120	78	x
089	59	Y	121	79	y
090	5A	Z	122	7A	z
091	5B	[	123	7B	{
092	5C	\	124	7C	|
093	5D	]	125	7D	}
094	5E	^	126	7E	~
095	5F	_	127	7F	⌂

ASCII Decimal	&H	Character	ASCII Decimal	&H	Character
128	80	Ç	160	A0	á
129	81	ü	161	A1	í
130	82	é	162	A2	ó
131	83	â	163	A3	ú
132	84	ä	164	A4	ñ
133	85	à	165	A5	Ñ
134	86	å	166	A6	ª
135	87	ç	167	A7	º
136	88	ê	168	A8	¿
137	89	ë	169	A9	⌐
138	8A	è	170	AA	¬
139	8B	ï	171	AB	½
140	8C	î	172	AC	¼
141	8D	ì	173	AD	¡
142	8E	Ä	174	AE	«
143	8F	Å	175	AF	»
144	90	É	176	B0	▒
145	91	æ	177	B1	▒
146	92	Æ	178	B2	▓
147	93	ô	179	B3	│
148	94	ö	180	B4	┤
149	95	ò	181	B5	╡
150	96	û	182	B6	╢
151	97	ù	183	B7	╖
152	98	ÿ	184	B8	╕
153	99	Ö	185	B9	╣
154	9A	Ü	186	BA	║
155	9B	¢	187	BB	╗
156	9C	£	188	BC	╝
157	9D	¥	189	BD	╜
158	9E	Pt	190	BE	╛
159	9F	ƒ	191	BF	┐

ASCII Decimal	&H	Character	ASCII Decimal	&H	Character
192	C0	└	224	E0	α
193	C1	┴	225	E1	ß
194	C2	┬	226	E2	Γ
195	C3	├	227	E3	π
196	C4	─	228	E4	Σ
197	C5	┼	229	E5	σ
198	C6	╞	230	E6	µ
199	C7	╟	231	E7	τ
200	C8	╚	232	E8	Φ
201	C9	╔	233	E9	Θ
202	CA	╩	234	EA	Ω
203	CB	╦	235	EB	δ
204	CC	╠	236	EC	∞
205	CD	═	237	ED	φ
206	CE	╬	238	EE	ε
207	CF	╧	239	EF	∩
208	D0	╨	240	F0	≡
209	D1	╤	241	F1	±
210	D2	╥	242	F2	≥
211	D3	╙	243	F3	≤
212	D4	╘	244	F4	⌠
213	D5	╒	245	F5	⌡
214	D6	╓	246	F6	÷
215	D7	╫	247	F7	≈
216	D8	╪	248	F8	°
217	D9	┘	249	F9	∙
218	DA	┌	250	FA	·
219	DB	▉	251	FB	√
220	DC	▄	252	FC	n^2
221	DD	▌	253	FD	■
222	DE	▐	254	FE	(blank 'FF')
223	DF	▀	255	FF	

A number of codes read by INKEY$ are two-code characters, and therefore not part of the ASCII code. If INKEY$ reads one of these special characters, the first character is a null character (code 000). In this case, your program should examine the second character of the string returned by INKEY$. This character is usually the key code relating to the position on the keyboard, and only then if the mode for that key (Shift, Ctrl, Alt, or none at all) is the one indicated in the following list.

Second Character	Key(s)
3	(null character) NUL
15	(shift tab) ↑ ←
16-25	Alt- Q, W, E, R, T, Y, U, I, O, P
30-38	Alt- A, S, D, F, G, H, J, K, L
44-50	Alt- Z, X, C, V, B, N, M
59-68	Function Keys F1 through F10 (when disabled as soft keys)
71	Home
72	Cursor Up
73	Pg Up
75	Cursor Left
77	Cursor Right
79	End
80	Cursor Down
81	Pg Dn
82	Ins
84	Del
84-93	F11-F20 (Shift- F1 through F10)
94-103	F21-F30 (Ctrl- F1 through F10)
104-113	F31-F40 (Alt- F1 through F10)
114	Ctrl-PrtSc
115	Ctrl-Cursor Left (Previous Word)
116	Ctrl-Cursor Right (Next Word)
117	Ctrl-End
118	Ctrl-Pg Dn
119	Ctrl-Home
120-131	Alt- 1,2,3,4,5,6,7,8,9,0,-,=
132	Ctrl-Pg Up

Error Messages

If GW-BASIC detects an error in your program, execution of the program usually stops and an error message is displayed telling you just what went wrong. The usual cause of an error situation is that your program has asked GW-BASIC to contradict the rules of the language, but a difference of opinion between GW-BASIC and an external device can sometimes be the cause.

If you wish, you can trap error situations using the ON ERROR statement and the two GW-BASIC variables ERR and ERL. In this case, your error event handling routine should do something about the error situation, if you want program execution to continue as if nothing had happened. If it does not, either GW-BASIC will return the same error message, or the results of your program will be unreliable.

The first section of this appendix consists of an overview of error numbers. When considering what error possibilities to cover in your program, simply look down this list. For a more detailed description of an error number refer then to the second section. Remember, you can also define error situations and allocate error numbers yourself (see ERROR).

The second section deals with GW-BASIC error messages in alphabetical order, and gives you indications as to what might have caused the error.

OVERVIEW OF ERROR NUMBERS

Number	Error Message
1	NEXT without FOR
2	Syntax error
3	RETURN without GOSUB
4	Out of data
5	Illegal function call

6	Overflow
7	Out of memory
8	Undefined line number
9	Subscript out of range
10	Duplicate Definition
11	Division by zero (This error cannot be trapped)
12	Illegal direct
13	Type mismatch
14	Out of string space
15	String too long
16	String formula too complex
17	Can't continue
18	Undefined user function
19	No RESUME
20	RESUME without error
22	Missing operand
23	Line buffer overflow
24	Device Timeout
25	Device Fault
26	FOR without NEXT
27	Out of paper
29	WHILE without WEND
30	WEND without WHILE
50	FIELD overflow
51	Internal error
52	Bad file number
53	File not found
54	Bad file mode
55	File already open
57	Device I/O error
58	File already exists
61	Disk full
62	Input past end
63	Bad record number
64	Bad file name
66	Direct statement in file
67	Too many files
68	Device unavailable
69	Communication buffer overflow
70	Disk Write Protect
71	Disk not ready
72	Disk media error

74	Rename across disks
75	Path/file access error
76	Path not found
—	Unprintable error

The following list shows each error message and its number, followed by an explanation of the message.

Bad file mode 54

You attempted to use PUT or GET with a sequential or closed file, to LOAD a random file or to execute an OPEN statement with a file mode other than Input, Output, Append or Random.

This error may also occur when an attempt is made to read from a file opened for output or appending or if you attempt to MERGE a non-ASCII File.

Try checking the OPEN command in your program.

Bad file name 64

An illegal form was used for the filename with a KILL, NAME or FILES command (e.g. a filename with too many characters).

Bad file number 52

A command references a file with a file number that is not OPEN or is out of the range of file numbers specified when loading GW-BASIC. Alternatively, the device name or filename was too long or illegal.

Check the name and number of the file in the OPEN statement for that file.

Bad record number 63

In a PUT or GET statement, the record number was either greater than the maximum allowed (16,777,215) or equal to zero.

Can't continue 17

You attempted to use CONT to continue a program that:

1. Has halted due to an error.
2. Has been modified during a break in execution
3. Does not exist.

Make sure that the program is loaded and start it with RUN.

Communication buffer overflow 69
Occurs when a communication input statement is executed and the input buffer is already full.

Use an ON ERROR trap to retry the input. If it is a case of characters being received faster than the program can process them, consider the following measures:

1. Increase the size of the communications receive buffer via the /C option when loading GW-BASIC.
2. If the transmitting device can support a "handshaking" protocol, include one in your communications program. This gives time for input processing to catch up.
3. Use a lower baud rate for transmitting and receiving.

Device Fault 25
An interface adapter returned a hardware error. When transmitting data to a communications file, it indicates that one or more of the signals being tested (specified on the OPEN "COM... command) was not found within the specified period of time.

Device I/O Error 57
An I/O Error (overrun, parity, framing or break) was detected during device I/O. When character length is 7 or less data bits, the highest order bit is turned on in the byte in error.

Device Timeout 24
Occurs if one or more of the signals to be tested by the OPEN "COM statement was not found in the specified period of time.

You can direct a trap handling routine to retry the operation (RESUME), but you should limit the number of attempts or your program could loop indefinitely.

Device Unavailable 68
You attempted to open a file to a non-existent device. Perhaps the hardware simply does not exist, or your program has disabled communication to the device. For example, COM1 statement was disabled by specifying the /C option with the value 0 when loading GW-BASIC. If this is the case you will have to return to operating system level (SYSTEM) and re-load GW-BASIC.

Direct statement in file 66

A direct statement (command) was encountered while LOADing or CHAINing a file in ASCII format. The LOAD or CHAIN is terminated.

The ASCII file should consist only of GW-BASIC commands with line numbers. The error may have been caused by a line feed character in the input stream.

Disk full 61

All disk storage space is in use. Upon encountering this error situation, GW-BASIC closes all files.

Erase any unnecessary files or use a new disk. Then retry the disk operation or run the program again from the beginning.

Disk Media Error 72

Occurs when the disk controller detects a hardware or media fault. This usually indicates a damaged disk.

Copy any existing files to a new disk and reformat the damaged disk.

Disk not Ready 71

The most likely problem is that the disk is not inserted properly.

Disk Write Protect 70

Occurs when you attempt to write to a disk that is write-protected. The error may also be caused by a hardware failure.

Check if you are using the right disk. Then remove the write protection and retry the operation.

Division by zero 11

A division by zero or the raising of zero to a negative power was encountered in an expression. If the cause of the error was division by zero, the display indicates machine infinity and the sign of number causing the error. A defective exponentiation yields positive machine infinity. This error cannot be trapped.

Duplicate Definition 10

Two DIM commands are given for the same array; or, a DIM command is given for an array after the default dimension of 10 has been established for that array by implicit definition; or, the OPTION BASE command was encountered by GW-BASIC after the first definition or use of an array.

FIELD overflow 50

A FIELD command attempted to allocate more bytes than were specified for the record length of a random file in the OPEN statement; or, the end of the FIELD buffer was encountered while doing sequential I/O (PRINT#, WRITE#, INPUT#) to a random file.

Check if the OPEN statement and the FIELD statement correspond. If you are doing sequential I/O to a random file, the length of the data read or written may not exceed the record length of the random file.

File already exists 58

The filename specified in a NAME command is identical to a filename already in use on the disk.

Retry the NAME command with a different name.

File already open 55

A sequential output mode OPEN command was issued for a file that is already open, or a KILL command was given for a file that is open.

Check that you only executed one OPEN to a file if you are writing to it sequentially or appending it. Close a file before you use KILL.

File not found 53

A LOAD, KILL, NAME, or OPEN command references a file that does not exist on the current disk.

Check that the correct disk is in the drive specified, and that the file specification was entered correctly, including a path if necessary. Retry the operation.

FOR without NEXT 26

A FOR command was encountered without a matching NEXT. Perhaps a FOR loop was active when the physical end of the program was reached.

Include a NEXT command in the program.

Illegal direct 12

You attempted to enter a command invalid in direct mode as a direct mode command (e.g DEF FN).

Enter the command as part of a program line.

Illegal function call 5

A parameter that is out of range is passed to a system function. This error can also be caused by:

1. A negative or improbable subscript to an array.
2. A negative or zero argument for a numeric function where none is allowed.
3. A call to a USR function for which the starting address has not yet been defined with DEF USR.
4. An improper argument to a string processing command or function.
5. A negative record number used with GET or PUT.
6. Trying to list or edit a protected BASIC program.
7. Trying to delete line numbers which don't exist.

Correct the program.

Input past end 62

An INPUT statement is executed after all the data in the file has been INPUT, or for a null (empty) file.

To avoid this error, use the EOF function to detect the end-of-file. The error also occurs if you try to read from a file that was opened for output or append.

If you want to read from a sequential output or append file, you must close it and open it again for input.

Internal error 51

An internal malfunction has occurred in GW-BASIC.

Recopy your disk. Check the hardware, and retry the operation.

Line buffer overflow 23

You attempted to input a line that has too many characters.

Separate multiple commands so they are on more than one line; or you may use string variables instead of constants.

Missing operand 22

An expression contains an operator with no operand following it.

Check that all the required operands are included in the expression.

NEXT without FOR 1
A variable in a NEXT command does not correspond to any previously executed, unmatched FOR command variable.

Adjust the program so the NEXT has a matching FOR.

No RESUME 19
An error handling routine is entered but contains no RESUME command.
Check to include RESUME in your error trapping routine to continue program execution. It is possible to add an ON ERROR GOTO 0 command to your error trapping routine so BASIC displays the message for any untrapped error.

Out of data 4
A READ statement was executed when there is no DATA left to be read.

Correct the program so that there are enough items in the DATA lists for all the READ commands in the program.

Out of memory 7
A program is too large, or has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated. Extensive PAINTing with jagged edges can also produce this error condition.

It is possible to use CLEAR at the beginning of your program to set aside more stack space or memory area.

Out of paper 27
The printer device is out of paper, or the printer is not switched on.

Insert paper, check that the printer is properly connected, and that the power is on; then, continue the program.

Out of string space 14
String variables have caused GW-BASIC to exceed the amount of free memory remaining. BASIC will allocate string space dynamically, until it runs out of memory.

Overflow 6
The result of a calculation is too large to be displayed by GW-BASIC. Integer overflow causes execution to stop. Otherwise, machine infinity with the appropriate sign is supplied as the result and execution continues. Integer overflow is the only type of overflow that

can be trapped. To correct integer overflow, you need to use smaller numbers, or change to single- or double-precision variables.

If a numeric result is so small that GW-BASIC cannot display it (underflow), the result is zero and execution continues without an error.

Path/file access error 75
During an MKDIR, CHDIR, or RMDIR operation, NCR-DOS was unable to make a correct path to filename connection. The error occurred when you tried to create a directory, remove the current directory or change a directory. During an OPEN operation you tried to open a read only file for output.

Path not found 76
During an OPEN, MKDIR, CHDIR, or RMDIR operation, NCR-DOS was unable to find the path specified.

Rename across disks 74
You attempted to specify two different disks when reNAMEing a file.

RESUME without error 20
A RESUME command was encountered although no error had been trapped. A common cause, the program has run on into the error trapping routines. To prevent this, use a STOP or END command at the point(s) where execution should stop.

RETURN without GOSUB 3
A RETURN command was encountered for which there is no previous, unmatched GOSUB statement. A common cause, the program has run on into subroutines. To prevent this, use a STOP or END command where execution should stop.

String formula too complex 16
A string expression is too long or too complex.

The expression should be broken into smaller pieces.

String too long 15
You attempted to create a string more than 255 characters long.

Use smaller strings.

Subscript out of range 9
An array element is referenced either with a subscript that is outside the dimensions of the array or with the wrong number of subscripts. See also "Illegal function call".

Check the usage of the array variable. It is possible that you subscripted a variable that is not an array.

Syntax error 2
A line is encountered that contains some incorrect sequence of characters (such as unmatched parentheses, misspelled command, incorrect punctuation, etc.). This error can also occur when a READ command tries to assign DATA of the wrong type (e.g. string for numeric) to a variable.

Too many files 67
You attempted to create a new file (using SAVE or OPEN) when all directory entries were full, or the filespec was incorrect.

Type mismatch 13
Your program tried to assign a string value to a numeric variable, or vice versa; or the wrong type of argument was passed to a function.

Undefined line number 8
A nonexistent line number is referenced in a command.

Use an existing line number (this may be a REM line).

Undefined user function 18
You called a function before you gave the function definition (DEF FN). Check that the program executes the DEF FN statement before you use the function.

Unprintable error _
An error message is not available for the error condition that exists. This may be caused by an ERROR command with an undefined error code.

Make sure you handle all error codes that you create, if you want your program to continue execution without intervention.

WEND without WHILE 30
A WEND command was encountered without a matching WHILE.

WHILE without WEND 29

A WHILE command does not have a matching WEND. WHILE was still active when the physical end of the program was reached.

Correct the program so that each WHILE has a corresponding WEND.

Additional Functions

This appendix contains mathematical functions in GW-BASIC language which are not intrinsic to GW-BASIC. A useful way of storing and calling such "user defined" functions is by means of the DEF FN command and the function FN. For example, to define a function returning the cotangent of a value, use the following statement.

```
DEF FN COTAN(X)=1/TAN(X)
```

To call up the defined function, a command like the following is needed:

```
RESULT = FNCOTAN(ANGLE)
```

Note that you can use GW-BASIC intrinsic functions as part of your function definition. This example makes use of TAN. You can give your function any name you wish, within the usual variable naming conventions. For example, the following function definition would serve the same purpose:

```
DEF FN SUNTAN(LOTION)=1/TAN(LOTION)
```

and could be called up with

```
ENIGMA=SUNTAN(SOMETHIN)
```

but later, you probably would not be able to remember why you defined the function in the first place. Therefore, it makes sense to use meaningful function names.

Logarithm to base B	LOGB(X) = LOG(X)/LOG(B)
Secant	SEC(X) = 1/COS(X)
Cosecant	CSC(X) = 1/SIN(X)
Cotangent	COT(X) = 1/TAN(X)
Inverse sine	ARCSIN(X) = ATN(X/SQR(1-X*X))
Inverse cosine	ARCCOS(X) = 1.570796

	-ATN(X/SQR(1-X*X))
Inverse secant	ARCSEC(X) = ATN(SQR(X*X-1)) +(X<0)*3.141593
Inverse cosecant	ARCCSC(X) = ATN(1/SQR(X*X-1)) +(X<0)*3.141593
Inverse cotangent	ARCCOT(X) = 1.57096-ATN(X)
Hyperbolic sine	SINH(X) = (EXP(X)-EXP(-X))/2
Hyperbolic cosine	COSH(X) = (EXP(X)+EXP(-X))/2
Hyperbolic tangent	TANH(X) = (EXP(X)-EXP(-X)) /(EXP(X)+EXP(-X))
Hyperbolic secant	SECH(X) = 2/(EXP(X)+EXP(-X))
Hyperbolic cosecant	CSCH(X) = 2/(EXP(X)-EXP(-X))
Hyperbolic cotangent	COTH(X) = (EXP(X)+EXP(-X)) /(EXP(X)-EXP(-X))
Inverse hyperbolic sine	ARCSINH(X) = LOG(X+SQR(X*X+1))
Inverse hyperbolic cosine	ARCCOSH(X) = LOG(X+SQR(X*X-1))
Inverse hyperbolic tangent	ARCTANH(X) = LOG((1+X)/(1-X))/2
Inverse hyperbolic secant	ARCSECH(X) = LOG((1+SQR(1-X*X))/X)
Inverse hyperbolic cosecant	ARCCSCH(X) = LOG((1+SGN(X) *SQR(1+X*X))/X)
Inverse hyperbolic cotangent	ARCCOTH(X) = LOG((X+1)/(X-1))/2

Decimal and Hexadecimal Numbers

Conversion of a decimal number to its hexadecimal equivalent is provided for by GW-BASIC in the form of the HEX$ function (see Chapter 4, *Statements, Commands and Functions).* This function returns the hexadecimal equivalent of a decimal number in the range -32768 to 65535 (if the number is negative, a two's complement form is used),

An example of using the HEX$ function:

```
PRINT HEX$(255)
will yield
FF
```

GW-BASIC does not itself provide a function for converting hexadecimal to decimal numbers, but you could use the following program, which converts a hexadecimal number to a positive decimal value. Enter any hexadecimal number, using uppercase letters for A to F. Do not prefix the number with 8H.

```
9900 INPUT "Hex number";H$
9905 DEC=0
9910 FOR C% = 1 TO LEN(H$)
9920 CH$=MID$(H$,C%,1)
9930 IF (CH$<"0" OR CH$>"9") AND (CH$<"A" OR
     CH$>"F") THEN GOTO 9900
9940 DEC=16*DEC-(CH$<"A")*(ASC(CH$)-48)-
     (CH$>"9")*(ASC(CH$)-55)
9950 NEXT C%
9960 PRINT "Hex ";H$;" = ";DEC;" decimal"
```

Use high line numbers outside your normal programming range. You can then leave this program in memory or MERGE it from disk while editing your main program. This enables you to do quick hexadecimal to decimal conversions during programming, simply by issuing the direct statement GOTO 9900.

Keyboard Positions

Your NCR Personal Computer refers internally to the keys of the keyboard by means of a keyboard position. In the normal course of GW-BASIC programming, you refer to a key by means of the "name" printed on it, for example

```
10 K$=INKEY$: IF K$ = "N" THEN GOTO 10
```

However, there are two sets of circumstances in which knowledge of the key position is required: when reading a two-code character from the keyboard (see Appendix B), and when defining your own key trap (see KEY).

The diagram shows the position number on the top of each key on your keyboard.

INDEX

NCR Corporation
Dayton, Ohio 45479

150-0000605 0185