Radio Frequency Interference Statement

"This equipment generates and uses radio frequency energy and if not installed and used properly, that is, in strict accordance with the manufacturer's instructions, may cause interference to radio and television reception. It has been type tested and found to comply with the limits for a Class B computing device in accordance with the specifications in Subpart J of Part 15 of FCC Rules, which are designed to proved reasonable protection against such interference in a residential installation. However, there is no guarantee that interference will not occur in a particular installation. If this equipment does cause interference to radio or television reception, which can be determined by turning the equipment off and on, the user is encouraged to try to correct the interference by one or more of the following measures:

..... reorient the receiving antenna
..... relocate the computer with respect to the receiver
..... move the computer away from the receiver
..... plug the computer into a different outlet so that computer and receiver are on different branch circuits.

If necessary, the user should consult the dealer or an experienced radio/television technician for additional suggestions.
The user may find the following booklet prepared by the Federal Communications Commission helpful: "How to Identify and Resolve Radio-TV Interference Problems."
This booklet is available from the US Government Printing Office, Washington, D.C., 20402, Stock No. 004-000-00345-4."

Note You can determine whether your computer is causing interference by turning it off. If the interference stops, it was probably caused by the computer or its peripheral devices. To further isolate the problem:

Disconnect the peripheral devices and their input/output cables one at a time. If the interference stops, it is caused by either the peripheral device or its I/O cable. These devices usually require shielded I/O cables. For Epson peripheral devices, you can obtain the proper shielded cable from your dealer. For non-Epson peripheral devices contact the manufacturer or dealer for assistance.

WARNING

This equipment has been certified to comply with the limits for a Class B computing device, pursuant to Subpart J of Part 15 of FCC Rules. Only peripherals (computer input/output devices, terminals, printers, etc.) certified to comply with the Class B limits may be attached to this computer. Operation with non-certified peripherals is likely to result in interference to radio and TV reception.

EPSON PORTABLE COMPUTER

PX-4

BASIC REFERENCE MANUAL

Trademark Acknowledgments

CP/M® is a registered trademark of Digital Research, Inc.
BASIC (Copyright 1977 – 1983 by Microsoft and EPSON) is upward compatible with the BASIC-80 specifications of Microsoft, Inc.
MICROCASSETTE™ is a trademark of Olympus Optical Co., Ltd.

TABLE OF CONTENTS

Preface i

Before using PX-4 BASIC iii

Chapter 1 INTRODUCTION TO BASIC 1-1
- 1.1 Starting and Terminating BASIC 1-1
- 1.2 Multiple Program Areas 1-11
- 1.3 Direct and Indirect Modes 1-12
- 1.4 Keyboards 1-13
- 1.5 Screen Cofiguration 1-17
- 1.6 Screen Editor 1-21

Chapter 2 PROGRAMMING 2-1
- 2.1 Program Lines and Line Numbers 2-1
- 2.2 Constants and Variables 2-2
- 2.3 Arithmetic Operations 2-10
- 2.4 Files 2-19
- 2.5 Error Messages 2-29

Chapter 3 COMMANDS AND STATEMENTS 3-1

Chapter 4 FILES 4-1
- 4.1 Program Files 4-1
- 4.2 Sequential Data Files 4-7
- 4.3 Random Data Files 4-13

Chapter 5 DATA COMMUNICATIONS SUPPORT 5-1
- 5.1 PX-4 Communications Supports 5-1
- 5.2 RS-232C Interface 5-3

Appendix A **DRIVE NAMES**
Appendix B **BASIC COMMANDS, STATEMENTS, AND FUNCTIONS AVAILABLE FOR I/O DEVICES**
Appendix C **FORMATTING CHARACTERS**
Appendix D **KEYBOARDS**
Appendix E **PROGRAMMABLE FUNCTION KEYS AND ITEM KEYS**
Appendix F **USER-DEFINED CHARACTERS**
Appendix G **MACHINE LANGUAGE PROGRAMS**
Appendix H **CONTROL CODES**
Appendix I **RESERVED WORDS**
Appendix J **CHARACTER CODES**
Appendix K **ERROR CODES AND MESSAGES**
Appendix L **DERIVED FUNCTIONS**
Appendix M **MEMORY MAP**

Configuration of This Manual

This BASIC Reference Manual is organized as follows:

Chapter 1 INTRODUCTION TO BASIC
Describes the fundamentals of the PX-4 BASIC.

Chapter 2 PROGRAMMING
Introduces a variety of topics and examples required for constructing BASIC programs.

Chapter 3 COMMANDS AND STATEMENTS
Describes in a systematic way the function and use of the BASIC commands and statements.

Chapter 4 FILES
Provides technical information pertaining to files and explains the use of files with sample programs.

Chapter 5 DATA COMMUNICATIONS SUPPORT
Shows how to perform data communications with external devices, by using the PX-4's communications facilities such as the RS-232C interface.

Appendixes
Provides supplementary information about the PX-4 BASIC facilities.

Preface

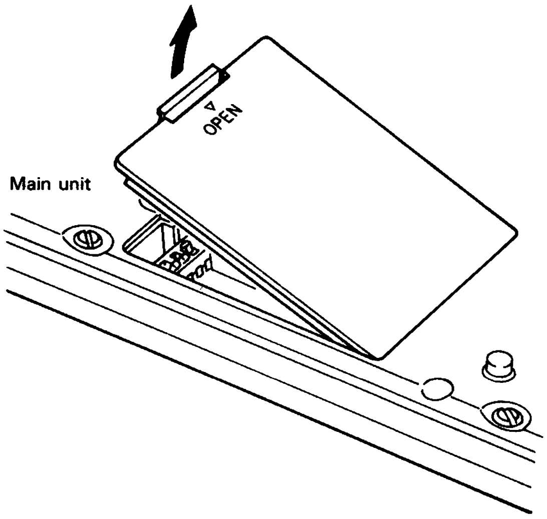
EPSON PX-4 BASIC is a derivative of widely accepted Microsoft BASIC developed by Microsoft, Corp., USA, and runs under the CP/M® operating system. Microsoft BASIC is the industry standard BASIC interpreter program for 8-bit microcomputers and enjoys well established reputation.

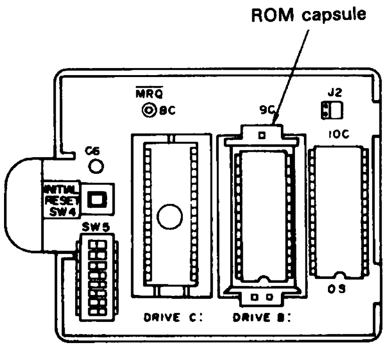
EPSON PX-4 BASIC is an enhanced version of the Microsoft BASIC interpreter program. It provides extensive graphics and timer facilities as well as powerful supports for peripheral devices unique to the PX-4. Further, inclusion of an item keyboard reinforces the PX-4's power as a dedicated computing machine and extends the range of its applications.

This manual describes all aspects of EPSON BASIC. It is also intended to be a handy guide for your BASIC programming.

Before using the PX-4 BASIC

The PX-4 BASIC interpreter program is installed in the PX-4 main unit in the form of a ROM capsule, which fits into the ROM capsule box on the back of the main unit. Before using BASIC, make sure that the BASIC ROM capsule has been installed (see figures below).





NOTE:

In the following explanations, it is assumed that the BASIC ROM capsule is installed in the socket marked "DRIVE B:."

Chapter 1

INTRODUCTION TO BASIC

1.1 Starting and Terminating BASIC

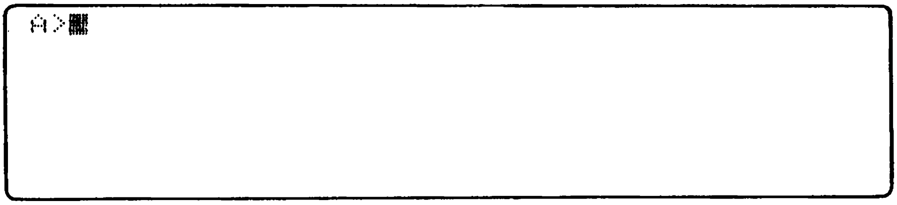
1.1.1 Starting BASIC

You can start PX-4 BASIC in two ways, with either a cold start or a warm start. Switch on your PX-4.

Cold start

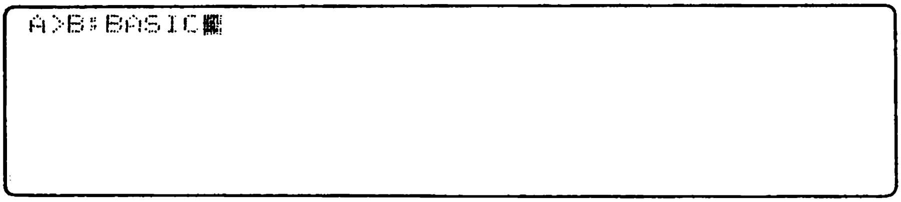
Cold start refers to starting up BASIC when the CCP screen (identified by the prompt >) or MENU screen is displayed on the computer display. The program area is initialized whenever a cold start is carried out.

(i) Starting BASIC from the CCP screen



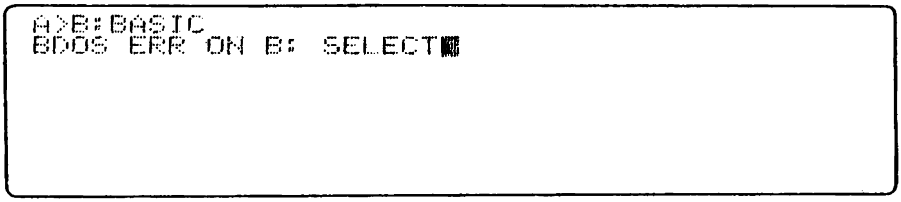
```
A>█
```

The above display is called the CCP screen. The CCP screen indicates that CP/M is waiting for a command. Type "B:BASIC" from the keyboard and press RETURN .



```
A>B:BASIC█
```

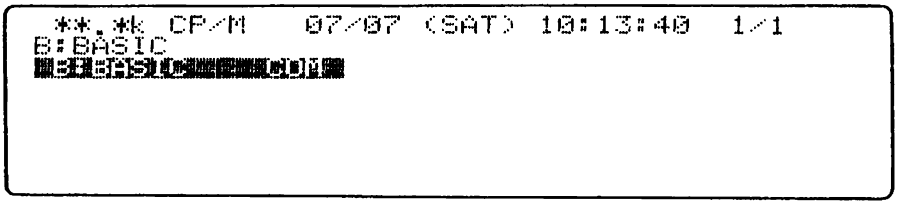
NOTE:
If the error message shown in the figure below is displayed on the screen, make sure that the BASIC interpreter ROM is correctly installed in your PX-4.



```
A>B:BASIC
BDOS ERR ON B: SELECT
```

(ii) Starting BASIC from the MENU screen

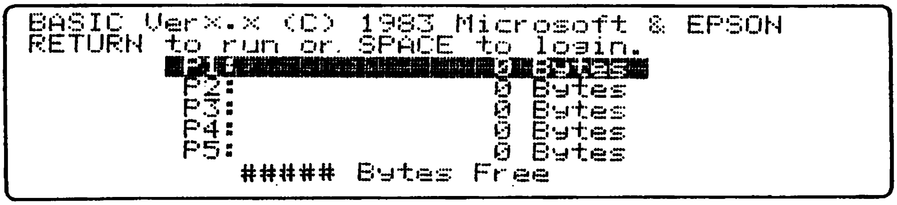
Select BASIC.COM using the cursor movement keys [→], [←], [↑], [↓] (BASIC. COM will be displayed in reverse video) and press [RETURN].



```
 **.*k CP/M    07/07 (SAT) 10:13:40   1/1
B:BASIC
[illegible]
```

NOTE:
***.* denotes the size of your CP/M system.*

When BASIC is cold-started, the following initial BASIC screen will appear:



```
BASIC Verx.x (C) 1983 Microsoft & EPSON
RETURN to run or, SPACE to login.
    [illegible]
    P2:                    0 Bytes
    P3:                    0 Bytes
    P4:                    0 Bytes
    P5:                    0 Bytes
       ##### Bytes Free
```

Initial BASIC screen

NOTE:
x.x denotes the BASIC version number and ##### the size of the memory that the user can use for programs and data.

Press the [SPACE] key.

```
BASIC Ver x.x (C) 1983 Microsoft & EPSON
20313 Bytes Free
P1:               0 Bytes
Ok
█
```

Now you can create or run your BASIC programs. This method of starting BASIC is referred to as cold start.

When BASIC is cold-started, the virtual screen is set to 40 columns by 50 lines and the programmable function keys are predefined as follows:

PF1	auto
PF2	list
PF3	edit
PF4	stat
PF5	run^M
PF6	load"
PF7	save"
PF8	system
PF9	menu^M
PF10	login

NOTE:
^M denotes a carriage return code, which you usually enter into the system by pressing the [RETURN] key. See the descriptions about the WIDTH command for the virtual screen size and the KEY command for the programmable function keys.

• **Parameters You Can Specify When Starting BASIC**

You can specify various parameters when cold-starting BASIC by using the following format:

[<drive name>:]BASIC[<file name>][/F:<number of files>][/M:<max memory address>:[S/:<max record length>] [/P:<program area number>]

<drive name>:
Specifies the name of the drive containing the BASIC.COM file. Since PX-4 BASIC runs on a ROM capsule, you must always specify drive B: or C:. If you omit this parameter, the currently logged-in drive is assumed. This parameter is automatically specified when BASIC is started from the menu screen.

NOTE:
Refer to the Operating Manual, Section 2.4.3 "CCP screen" for the logged-in drive.

<file name>:
Specifies the name of the program to be loaded and executed immediately after BASIC is started (see Section 2.4 "Files" on how to specify a program name). BASIC will automatically load and execute the specified program after it is started. BASIC assumes a file extension of BAS if the extension is omitted.

/F:<number of files>:
Specifies the maximum number of files that can be opened at any one time during the execution of a BASIC program. The maximum value you may enter is 15. This parameter defaults to 3. In following BASIC programs, you cannot specify file numbers higher than that specified here. When you set the number of files to 4 in a program, for example, you cannot use files with file numbers greater than 4 in the program. Specifying too many files will occupy a corresponding amount of memory space for the file I/O buffers, leaving less user area in memory.

/M:<max memory address>:
Specifies the maximum memory address of the work space that BASIC can use as the memory area for storing programs and variables. The memory area from this address up to the BDOS start address is reserved as a user area that BASIC will not access. You can load any machine-language programs in this area. The maximum memory address must be the BDOS start address or &H6000, whichever is lower. Notice that the BDOS start address depends on the size of the RAM disk you specify. If this parameter is omitted, BASIC assumes the BDOS start address or &H6000, whichever is lower.

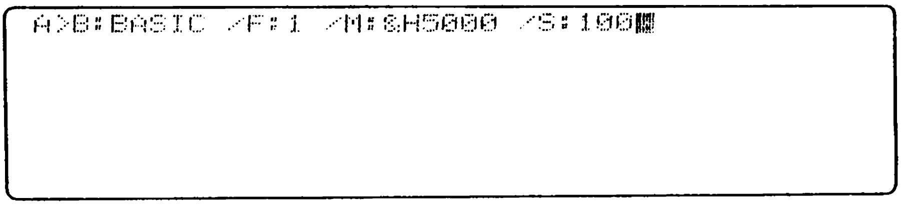
NOTE:
See Appendix N "MEMORY MAP" for the BDOS start address. Numbers preceded by &H are in hexadecimal.

/S:<max record length>:
Specifies the maximum record length in bytes for use with random files. You cannot specify a record length larger than this value with the BASIC OPEN statement. The default record length is 128 bytes.

/P:<program area number>:
Specifies the number of the program area into which BASIC will load the program you specified with its file name. You may specify a number from 1 to 5. BASIC assumes 1 if you omit this parameter. If you do not specify the file name, the program area which will be used is determined immediately when BASIC is started, and BASIC waits for a command in the program area specified by this parameter.

NOTE:
See Section 1.2 "Multiple Program Areas" for the program areas.

Example 1:



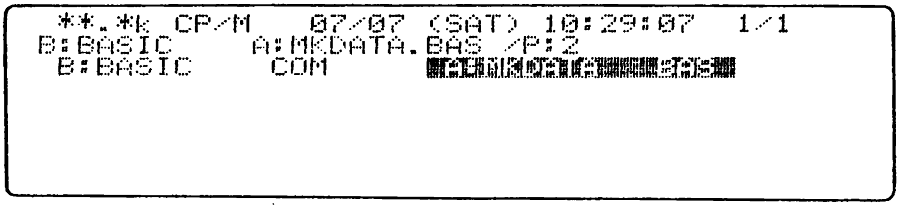
```
A>B:BASIC /F:1 /M:&H5000 /S:100█
```

The above command starts BASIC from the CCP screen. The meanings of the parameters are as follows (assuming that the BASIC interpreter is on drive B:):

/F:1	Sets the number of files that can be open to 1.
/M:&H5000	Sets the maximum memory address to address &H5000.
/S:100	Sets the record size for use with random files to 100 bytes.

Each program area is cleared when BASIC is started.

Example 2:



```
  **.*k  CP/M     07/07 (SAT) 10:29:07    1/1
 B:BASIC     A:MKDATA.BAS /P:2
  B:BASIC      COM       [illegible]
```

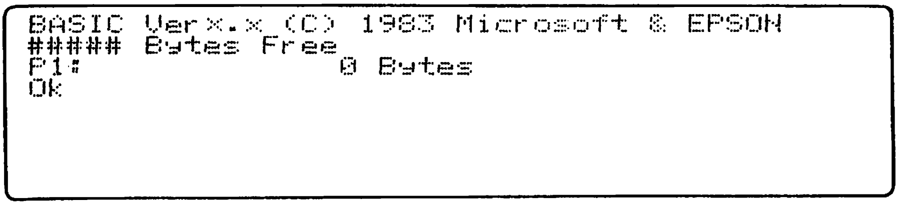
The above command starts BASIC from the menu screen (assuming that the BASIC interpreter is on drive B:). The meanings of the parameters are as follows:

A:MKDATE.BAS and "P:2	Specifies that BASIC, after starting execution, must load the program MKDATE.BAS from drive A: into program area 2 for execution.

Each program area is cleared when BASIC is started.

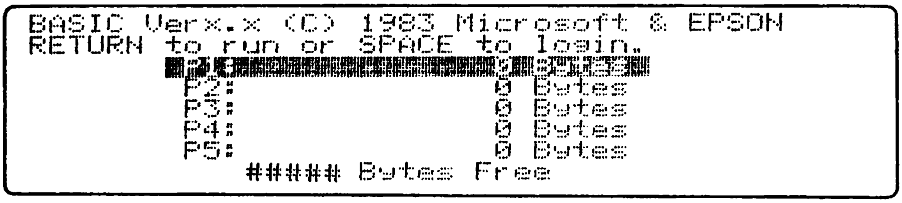
Warm Start

Execute a cold start and the following message is displayed on the screen:



```
BASIC Verx.x (C) 1983 Microsoft & EPSON
##### Bytes Free
P1:               0 Bytes
Ok
```

Now switch the computer off and on again. The display will show the initial BASIC screen instead of the CCP or MENU screen.



```
BASIC Verx.x (C) 1983 Microsoft & EPSON
RETURN to run or SPACE to login.
      [illegible]
      P2:               0 Bytes
      P3:               0 Bytes
      P4:               0 Bytes
      P5:               0 Bytes
         ##### Bytes Free
```

This means that once the computer enters the BASIC mode, it cannot exit the BASIC mode unless you execute the SYSTEM command. This method of starting up BASIC is called a "warm start."

Using this method, you can start BASIC immediately after switching on your PX-4 and make it a dedicated BASIC machine.

When BASIC is started by a warm start, the program areas retain whatever data they contained before the power was turned off. The programmable function keys also retain their assigned values.

1.1.4 Starting BASIC in other ways

You can also get BASIC to start automatically. Starting BASIC automatically when the PX-4 POWER switched is turned on is called an "auto start", and starting BASIC and running a given program automatically at a specified time is called "wake".

To use auto start and wake, see the descriptions about the AUTOST and WAKE commands in the Operating Manual, Subsections 2.5.2 "Wake" and 2.5.3 "Auto start".

1.1.2 Terminating BASIC

To terminate BASIC, use the SYSTEM command. Since the SYSTEM command clears the contents of the whole program area, you must save any wanted programs on an auxiliary storage device. After BASIC is terminated, the menu screen appears if display of the menu screen is enabled and the CCP screen appears if it is not.

1.1.3 Turning the power off

The PX-4 starts up BASIC immediately if you power it on again after having temporarily switched it off (warm start). BASIC, however, behaves in different ways depending on how you switch the power off.

(1) When you turned the POWER switch off while holding the CTRL key down (continuity mode)

When you turn the POWER switch off while holding the CTRL key down then turn the POWER switch on again, BASIC will return to the same operating state as before the switch was turned off. In this case, the programs are preserved in memory and the screen remains unchanged (i.e., the initial BASIC screen is not displayed).

This holds in the following cases:

- When you turn the POWER switch off during the execution of a program
- When the battery has run out and the power is automatically turned off (auto power-off)
- When the battery level falls below the lower limit and the power is automatically turned off
- When a POWER OFF command with the RESUME parameter is executed
- When the item keyboard is installed

(2) When you simply turn the POWER switch off (restart mode)

When you simply turn the POWER switch off, the initial BASIC screen appears on the display once you turn the switch on again. The programs are held in memory. The same is also true when you execute a POWER OFF command without the RESUME parameter.

1.2 Multiple Program Areas

PX-4 BASIC divides the program memory into five areas to allow up to five programs to be stored at a time. Each program area is assigned a number called a program area number. You must use this number to specify the area the program is to be run with the /P: parameter when starting BASIC with the BASIC command, or when starting the program from the initial BASIC screen. After BASIC is started, you can log in the specified program area by entering the LOGIN n (n denotes the program area number) command from the keyboard. You need not specify the maximum size of each program area; the BASIC interpreter manages the program areas within the limit of the program memory available for programs.

The memory areas reserved for simple, array, and string variables are used commonly by the program areas. The memory areas for these variables are cleared by executing the LOGIN command.

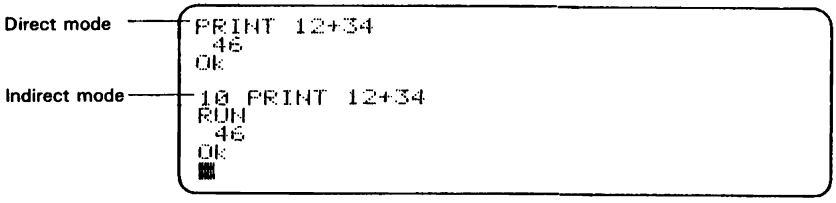
The program in the currently logged in program area can be moved to another program area by using the PCOPY command. It is advised that you should name frequently used program areas with the TITLE command. To display the initial BASIC screen after having executed a program, execute the MENU command.

The programs in all the program areas are erased when BASIC is terminated by the SYSTEM command or a cold start is performed. For further information, see the command descriptions for the LOGIN, MENU, PCOPY, SYSTEM, and TITLE commands in Chapter 3 "COMMANDS AND STATEMENTS."

1.3 Direct and Indirect Modes

When creating a program, you normally enter program lines by keying in BASIC statements preceded by a line number from the keyboard. Each statement is accepted by BASIC when you press the RETURN key at the end of the program line. At this point, the BASIC interpreter only stores the program statement in the current program area and will not execute them. To execute these statements, you have to enter a RUN command after making sure that the cursor is at the extreme left of the screen. BASIC will then execute the statements starting at the program line with the lowerest line number. This execution mode is called the indirect mode.

If you enter a command without specifying a line number and press the RETURN key, BASIC will execute the command immediately. This execution mode in which commands are executed without the use of the RUN command, one line at a time, is called the direct mode. BASIC statements DEFFN, DEFUSR, GOSUB-RETURN, GOTO, ON ERROR GOTO, ON GOSUB, READ, and WHILE-WEND can not be executed in the direct mode.

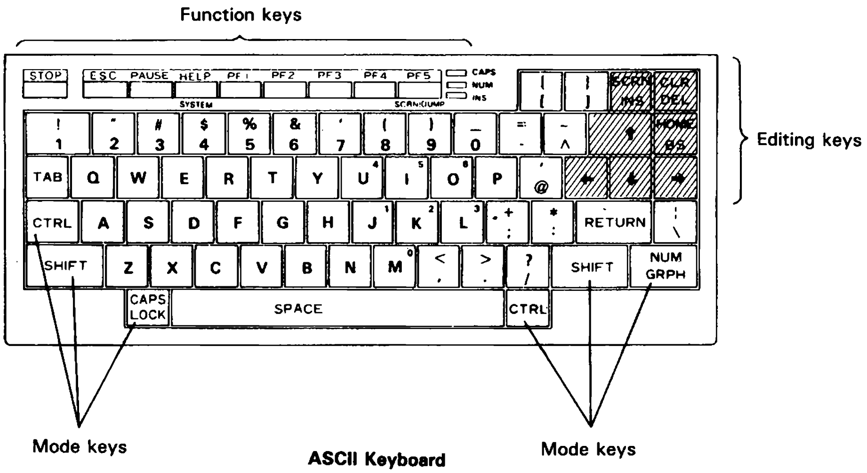


```
PRINT 12+34
 46
Ok
10 PRINT 12+34
RUN
 46
Ok
```

NOTE:
In this manual, BASIC statements are referred to as commands when they are executed in the direct mode.

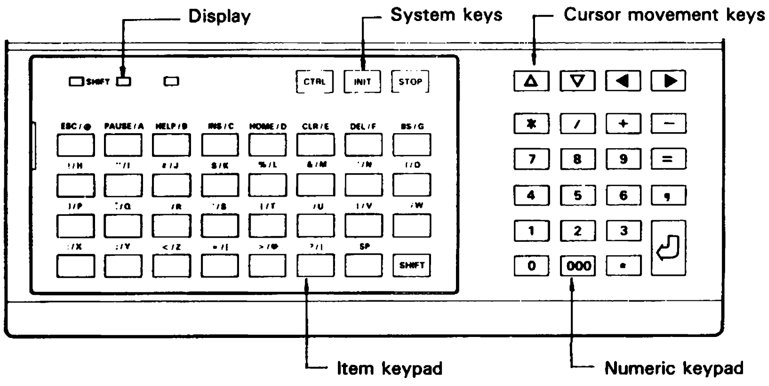
1.4 Keyboards

1.4.1 ASCII and item keyboards



ASCII Keyboard

The figure above shows the ASCII keyboard.



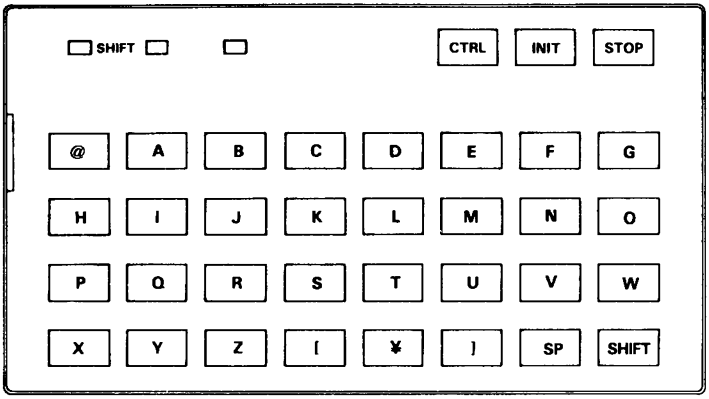
Item Keyboard

The keyboard illustrated above is called an item keyboard.

The item keyboard is designed for dedicated use on special-purpose machines. It is not suited for building BASIC programs. At power-on time, the item keyboard is initialized as follows (this facilitates minor program modification):

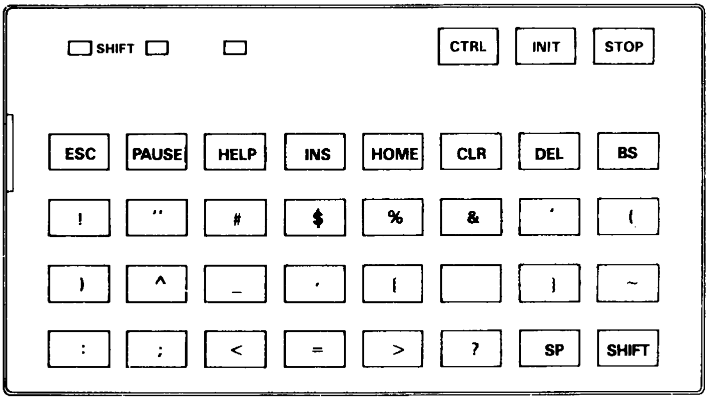
Normal mode

In the normal mode, you can use the following keys in the normal keyboard mode (SHIFT LED is off):



Shift mode

The following keys are available in the shift keyboard mode (SHIFT LED is on):



These key arrangements can be redefined from BASIC (see the KEY command description).

When the item keyboard is installed, the system display function using the CTRL + HELP keys and the screen dump function using the CTRL + PF5 are not available (you can obtain a hardcopy of the screen image by using the BASIC COPY command, however).

For details of programing item keys, see Appendix E and Operating Manual.

1.4.2 Special keys

PX-4 BASIC supports several special keys. The item keyboard however, does not have a key which corresponds to the ASCII keyboard [CTRL] key, therefore it can not provide the same special functions as those available to the ASCII keyboard.

The special keys are described below.

(1) [PF1] - [PF5], [SHIFT] + [PF1] - [PF5] (not available on the item keyboard)
These keys are called programmable function keys. You can assign a character string of up to 15 characters to each programmable function key. You can save many keystrokes by assigning often-used character strings (including BASIC commands and statements) to these keys (see the KEY, KEY LIST, and KEY LLIST commands). See also Subsection 1.1.1 "Starting BASIC" for the procedure to set up the function keys when BASIC is cold-started.

(2) [STOP] or [CTRL] + [C]
Pressing the [STOP] key or [CTRL] + [C] keys interrupts the execution of the current program and places BASIC in the wait state. To resume execution, enter the CONT command from the keyboard. The program cannot be resumed with the CONT command if it has been modified after being interrupted or if the keys were pressed while the program was doing a file I/O. These keys should be used to return BASIC into the wait state from the AUTO command entry mode. You can enable or disable these functions by using the STOP KEY command (see the STOP KEY command).

(3) [PAUSE] or [CTRL] + [S]
These keys temporarily stop the execution of a program or printing of a program listing. To resume execution or printing, press a key other than the [STOP] or [CTRL] + [C] keys.

(4) [CTRL] + [PF5] (not available on the item keyboard)
Pressing these keys simultaneously transfers the contents of the screen to the printer in a bit image format. These keys serve the same function as the COPY command (see the COPY command).

(5) CTRL + HELP (not available on the item keyboard)
These keys call up the System Display, which shows you how to operate the microcassette recorder manually or how to set the alarm or wake time. Pressing the ESC key will return you to the screen displayed before the System Display appeared.

(6) CTRL + STOP
Pressing these keys simultaneously while the PX-4 is performing an I/O operation to or from the microcassette drive or RAM disk stops the I/O operation.

1.4.3 Entering characters from the keyboard

Refer to the Operating Manual, Section 2.3 ''Inputing Data from the Keyboard'' on how to enter characters from the keyboard.

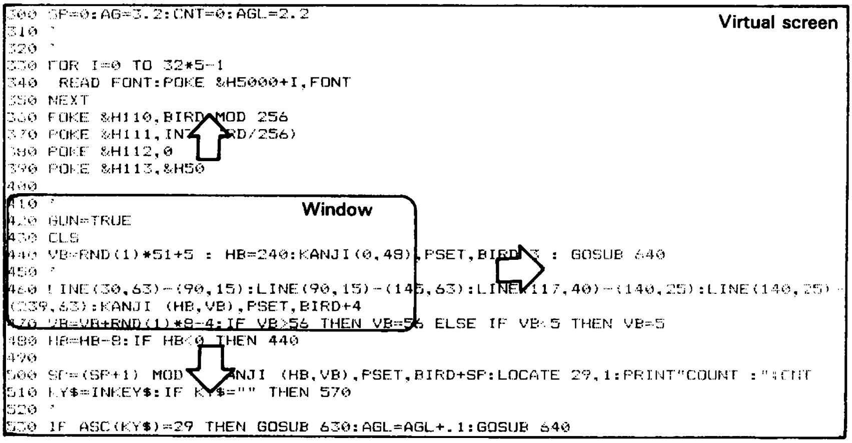
1.5 Screen Configuration

The LCD (Liquid Crystal Display) can display dot image graphics such as dots and lines as well as alphanumeric characters.

1.5.1 Virtual Screen

The LCD can display alphanumeric characters on the 40-column by 8-line screen. The PX-4 has an internal screen, called the virtual screen, which is larger than the physical screen. The contents of the virtual screen are viewed through a window. You can move the window vertically and horizontally to view the contents of the whole virtual screen. The LCD corresponds to a window, and movement of the window over the virtual screen is called scrolling. The virtual screen allows you to edit a wider area of a BASIC program than can be physically viewed with one window.

You can set the numbers of columns and lines of the virtual screen using the WIDTH command. The WIDTH command can set the number of columns to either 40 or 80 and the number of lines to any number between 8 and 50. The product of the number of columns and lines must not exceed 2000.



```
300 SP=0:AG=3.2:CNT=0:AGL=2.2
310 '
320 '
330 FOR I=0 TO 32*5-1
340  READ FONT:POKE &H5000+I,FONT
350 NEXT
360 POKE &H110,BIRD MOD 256
370 POKE &H111,IN[illegible]RD/256)
380 POKE &H112,0
390 POKE &H113,&H50
400
410 '
420 GUN=TRUE
430 CLS
440 VB=RND(1)*51+5 : HB=240:KANJI(0,48),PSET,BI[illegible] : GOSUB 640
450 '
460 LINE(30,63)-(90,15):LINE(90,15)-(145,63):LINE(117,40)-(140,25):LINE(140,25)-
(239,63):KANJI (HB,VB),PSET,BIRD+4
470 VB=VB+RND(1)*8-4:IF VB>56 THEN VB=56 ELSE IF VB<5 THEN VB=5
480 HB=HB-8:IF HB<0 THEN 440
490
500 SP=(SP+1) MOD [illegible]NJI (HB,VB),PSET,BIRD+SP:LOCATE 29,1:PRINT"COUNT :";CNT
510 KY$=INKEY$:IF KY$="" THEN 570
520 '
530 IF ASC(KY$)=29 THEN GOSUB 630:AGL=AGL+.1:GOSUB 640
```

NOTE:
Virtual screen size is set to 80 columns by 25 lines.

Scrolling

The screen has three scrolling modes: tracking, nontracking, and horizontally nontracking modes. In the tracking mode, the window scrolls following the cursor as you move the cursor. In the nontracking mode, the window maintains its position in relation to the cursor when it is moved over the virtual screen. When the width of the virtual screen is set to 80 columns, horizontal scrolling occurs at every 20 columns. The window may also be disabled for scrolling horizontally (horizontally nontracking mode).

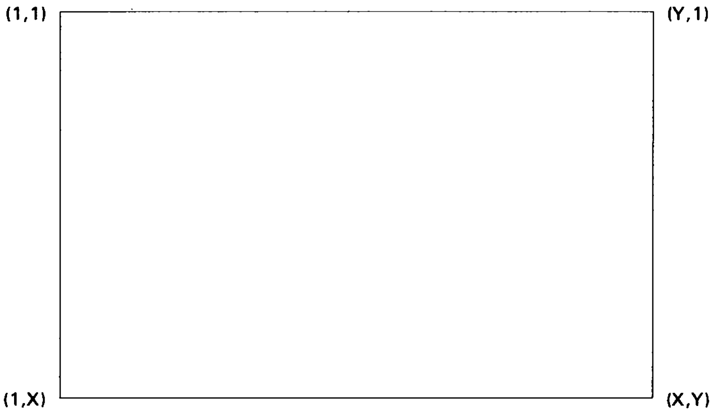
The scrolling mode can be changed by pressing the SHIFT + SCRN keys simultaneously. If the scrolling mode is switched from nontracking to tracking when the cursor is outside the screen window, the window immediately moves to the position of the cursor in the virtual screen. BASIC is placed in the tracking mode when it is started.

There are several keys which are used to control scrolling. These keys allow you to efficiently position the window in the virtual screen. Since the item keyboard does not have a key which corresponds to the CTRL key on the ASCII keyboard, you cannot use the scroll control functions. The scroll control keys are:

SHIFT + ↑	Scrolls the screen window upward.
SHIFT + ↓	Scrolls the screen window downward.
SHIFT + ←	Scrolls the screen window to the left 20 columns at a time when the virtual screen width is set to 80 columns.
SHIFT + →	Scrolls the screen window to the right 20 columns at a time when the virtual screen width is set to 80 columns.
CTRL + ↑	Scrolls the screen window to the top of the virtual screen.
CTRL + ↓	Scrolls the screen window downward one screen at a time.
CTRL + ←	Scrolls the screen window to the extreme left of the virtual screen when the virtual screen width is set to 80 columns.
CTRL + →	Scrolls the screen window to the extreme right of the virtual screen when the virtual screen width is set to 80 columns.
CTRL + SCRN	Pressing these keys simultaneously when the cursor is outside the window scrolls the screen window so that the cursor is at the center of the window. The cursor will not be centered in the window however, if it has been positioned at the top or bottom of the virtual screen.

1.5.2 Character coordinates

You must specify the position of the characters to be displayed on the screen by using the positions in the virtual screen (character coordinates). With character coordinates, the upper left corner of the virtual screen is taken as (1, 1) and the value of the x coordinate increases from left to right up to 40 or 80 columns, while the value of the Y coordinate increases from top to bottom, up to either 8 to 50 or 8 to 25 lines.

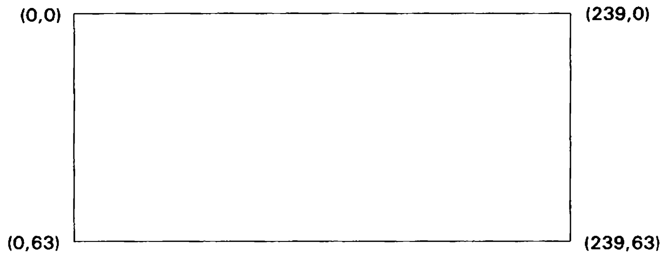


Character coordinates

X and Y can be specified with the WIDTH command.

1.5.3 Graphics coordinates

The graphics coordinates are used to draw bit image graphics using the PSET and LINE statements. In graphics coordinates, the upper left of the LCD screen is taken as the origin (0, 0) irrespective of the size of the virtual screen. You can specify x coordinate values from 0 to 239 and the y coordinate values from 0 to 63. Since the concept of a virtual screen does not apply to the graphics coordinates, once the graphics displayed on the LCD screen are scrolled out of the screen, they will not return to the original position when the screen is scrolled back.



Graphics coordinates

Dot positions can be specified using either absolute coordinates in which the coordinates of the dot positions are directly specified or relative coordinates which have their coordinates specified relative to the position of the previously specified dot.

You can specify the use of the relative coordinates by specifying the keyword STEP before the coordinate specification. For example, assuming that the LRP (Last Referenced Point or position whose coordinates are updated every time a command relating to graphics coordinates is executed) is at (x_0, y_0), the absolute coordinates of STEP (x, y) are (x_0 + x, y_0 + y).

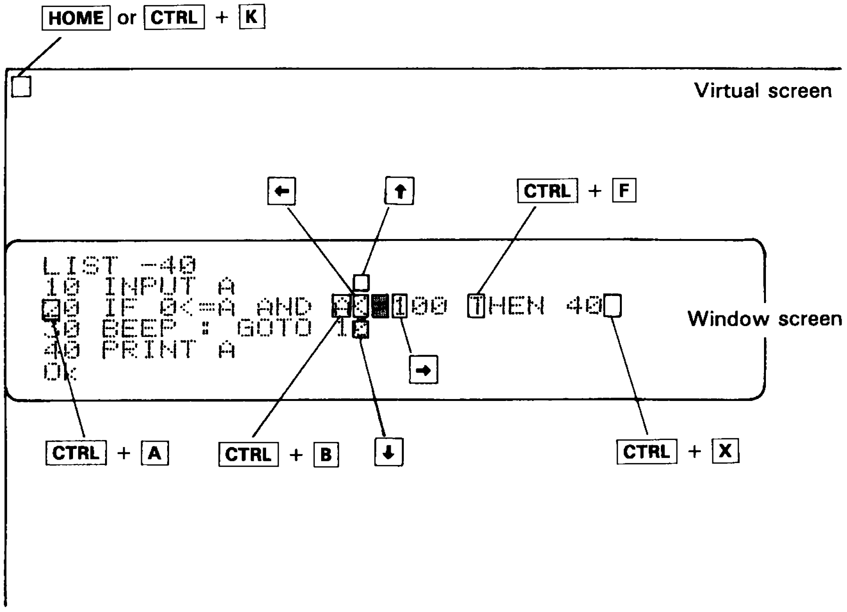
1.6 Screen Editor

The screen editor provides the user with facilities to carry out editing functions on the display screen, such as text addition and deletion, which allow you to modify your programs efficiently. With the screen editor, you can modify the character string being input through the execution of the LINE INPUT statement.

There are several function keys which are used by the screen editor. The item keyboard does not have a key which corresponds to the CTRL key of the ASCII keyboard, so you cannot use the screen editor functions. The editor function keys are:

HOME or CTRL + K :	Position the cursor at the top of the virtual screen.
CLS or CTRL + L :	Erase all text data on the virtual screen and position the cursor at the top of the virtual screen.
INS or CTRL + R :	Place the keyboard into the insert mode. When this mode is entered, an underline character ("_") appear on the screen as the cursor. Move the cursor to the character location you want to insert characters and enter them. The cursor and the characters to the right of the cursor will shift right. To exit this mode, press either INS or CTRL + R again. You may also exit the insert mode by pressing a cursor movement or RETURN key.
DEL :	Pressing this key after positioning the cursor under the character to be deleted deletes the character and shifts the characters to the right of the cursor one character to the left.

↑	: Moves the cursor one line up.
↓	: Moves the cursor one line down.
←	: Moves the cursor one character to the left.
→	: Moves the cursor one character to the right.
STOP or CTRL + C	: Cancel the editing changes made so far on the current line and position the cursor at the beginning of the following line. Also use these keys to return to the command mode if you entered the automatic line generation mode with the AUTO command. If you press these keys during execution of a program, the program will terminate and BASIC waits for another command.



```
LIST -40
10 INPUT A
20 IF 0<=A AND A<=100 THEN 40
30 BEEP : GOTO 10
40 PRINT A
Ok
```

Moving cursor

BS or CTRL + H :	Delete the character to the left of the cursor. The characters under and to the right of the cursor shift one character to the left.
TAB or CTRL + I :	Move the cursor to the next tab position. The character positions from where the cursor is located to the next tab position are filled with spaces. Tabs are placed at every eighth character position.
CLR or CTRL + L :	Clear the entire screen and move the cursor to the home position (the upper left corner of the screen).
CTRL + A :	Move the cursor to the beginning of the current line.
CTRL + B :	Move the cursor to the beginning of the preceding word.
CTRL + E :	Erase the characters from the cursor position to the end of that line.
CTRL + F :	Move the cursor to the beginning of the next word.
CTRL + X :	Move the cursor to the right of the last character on the current line.
CTRL + Z :	Erase the characters from the cursor position to the end of the virtual screen.
Scroll control keys :	Used to scroll the window over the virtual screen (see 1.5.1 ''Virtual screen'').

1.6.2 Editing a program

You can edit your programs quickly and easily by using the screen editor. To do this, display the program to be edited on the screen, by using the LIST or EDIT command. Then move the cursor to the location you want to edit your program text. You must edit only one program line at a time, and press the RETURN key after completing the editing. If you forget to press the RETURN key, the line in the program memory will remain unchanged even though the line on the screen appears to be altered.

Chapter 2

PROGRAMMING

2.1 Program Lines and Line Numbers

BASIC regards a program as comprising a collection of lines. Each line consists of a line number and a statement. For example, when you key in

$$\underbrace{10}_{\text{Line number}} \qquad \underbrace{A=10}_{\text{Statement}} \qquad \boxed{\text{RETURN}}$$

from the keyboard, BASIC stores the statement "A = 10" in program memory with a line number of 10 as its label. When you execute the RUN command, BASIC interprets and executes the program lines in program memory sequentially in numerical order.

You must specify the destination of the GOTO or GOSUB statement with a line number. You must also specify the line number when modifying, deleting, or printing a portion of a program.

NOTE:
BASIC executes all program lines in numerical order unless the program flow is changed by the GOTO, GOSUB, or IF...THEN...ELSE statement.

A line can be a maximum of 255 characters long including the line number. Within this limit, a line can contain two or more statements, which must be separated by colons (:). Line numbers must be integers between 0 and 65529.

The use of the AUTO command is convenient when entering two or more program lines. When the AUTO command is executed, BASIC will automatically generate a new line number each time you press the $\boxed{\text{RETURN}}$ key, saving you from having to enter line numbers. To terminate the AUTO command, press the $\boxed{\text{CTRL}}$ + $\boxed{\text{C}}$ or $\boxed{\text{STOP}}$ (see the description of the AUTO command).

2.2 Constants and Variables

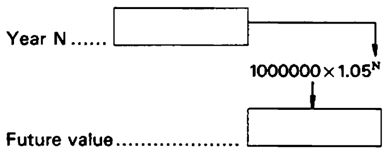
BASIC handles two types of data: constants and variables. Constants represent with numeric or alphabetic characters the actual values of some object (e.g., length, weight, amount of money, etc.). Variables are labels used to represent values.

Consider an example of depositing one million in a bank at an annual interest rate of 5 percent over a period of three years. The future value of the investment can be calculated using the following formula:

$$\begin{aligned}\textbf{(Total value after 3 years)} &= \textbf{(Principal)} \times \textbf{(1 + Interest/100)\^{}(Year)}\\ &= \mathbf{1{,}000{,}000 \times 1.05^3}\end{aligned}$$

Let us use the variable N for the compounding year and S for the future value after N years. The above formula can be expressed as shown below where the principal 1000000 and interest rate 1.05 are constants.

$$\mathbf{S = 1{,}000{,}000 \times 1.05^N}$$



Using the above formula, you can find not only the future value after three years but also future values after any years by assigning appropriate values to N. The program will look like this:

```
10  INPUT N              Gets the year.
20  S=1000000*1.05^N     Calculates future value after N years and assigns the result to S.
30  PRINT S              Displays the value of S on the screen.
40  END                  Terminates the program.
```

After entering the above program, key in:

```
RUN
```

BASIC will then run the program and display the prompt:

```
? ■
```

Eter a year.

```
? 3■
```

Press the RETURN key, and the program evaluates the expression then displays the result on the screen as follows:

```
1.15762E+06
```

NOTE:
1.15762E + 06 represents 1.15762 × 10^6.

You can apply the above program to various situations by assigning the variable F to the principal and R to the interest rate. To do so, modify lines 10 and 20 as follows:

```
10 INPUT N,F,R
20 S=F*(1+R/100)^N
```

Run the program using the RUN command. Enter variable values in the order N (year), F (principal), and R (interest rate in %), separated by commas. For example:

```
? 3,5000000,5■
```

The use of constants and variables are described on the following pages.

2.2.1 Constants

Constants are classified as follows:

```
String constants                          ┌ Decimal
Numeric constants ┬ Integer type         ┤ Octal
                  │                      └ Hexadecimal
                  │
                  └ Real type            ┌ Single precision
                                         └ Double precision
```

(1) String constants

String constants are enclosed in double quotation marks when they are entered into a program. The CHR$ function must be used to handle double quotation marks as string constants. String constants must not exceed 255 characters.

Examples:

```
PRINT "HELLO"····HELLO
PRINT "I SAID ";CHR$(34);"HELLO";CHR$(34)
         ···I SAID "HELLO"
```

(2) Numeric type constants

Numeric constants are either positive or negative numbers or 0. Negative numbers must always be preceded by a minus (−) sign but positive numbers can be preceded by a plus (+) sign simply when desired.

①Integer constants

Integer type constants are integers from −32768 to +32767. Use one of the following forms to represent integers:

• Decimal form:	Decimal numbers from 0 to 9 followed by a % mark.
• Octal form:	Octal numbers from 0 to 7 preceded by & or &O. The range of values that BASIC can handle is from &0 to &177777.
• Hexadecimal form:	Integers represented by characters 0 to 9 and A to F preceded by &H. A, B, C, D, E, and F represent 10, 11, 12, 13, 14, and 15 in decimal, respectively. The range of values that BASIC can handle is from &H0 to &HFFFF.

② Real type constants

Real type constants can be subdivided into single- and double-precision constants.

(i) Single-precision constants have three forms. Single-precision constants are rounded to 6 digits for display or printout.

(a) Numbers with not more than seven significant digits
Example: 3489.0

(b) Numbers represented using E (the exponent can take values from -38 to 37).
Example: $\underbrace{235.988}_{\text{Mantissa}}\text{E}\underbrace{-7}_{\text{Exponent}} = 235.988 \times 10^{-7} = 0.0000235988$

(c) Numbers followed by a !
Example: 279.9!

(ii) Double-precision constants have three forms. BASIC displays or prints double-precision constants as 16 digits.

(a) Numbers with 8 to 16 significant digits
Example: 345692811

(b) Numbers represented using D (the exponent can take values from -38 to 37).
Example: $-1.09432\text{D}-6 = -(1.09432 \times 10^{-6}) = -0.00000109432$

(c) Numbers followed by a #
Example: 3489.0#

Precision of numeric data

The range of values that can be internally represented inside the computer is limited. The precision of numeric values increases as the range of values to be taken increases. PX-4 BASIC supports the following types of precision:

$$\begin{cases}\text{Integer type}\\ \text{Single precision}\\ \text{Double precision}\end{cases}$$

Among numbers of the above precision types, double-precision real numbers provide the highest precision but occupy the most memory space and take the most processing time. If you know in advance that your program uses only integers, you need not use single- or double-precision representation. Use the most appropriate data representation according to the processing to be performed.

2.2.2 Variables

(1) Variable names

A variable name is a sequence of 1 to 40 alphanumeric characters which begins with an alphabetic character. BASIC does not distinguish between upper- and lower-case alphabetic characters. Reserved words cannot be used as variable names. Reserved words are keywords which are used to represent BASIC statements and functions (see Appendix I "RESERVED WORDS"). If a reserved word is used as a variable name, BASIC displays an error message "SN Error " on the screen and terminates the execution. The first two letters of variable names must not be FN. If a variable name beginning with FN is used, BASIC will call a user-defined function.

(2) Variable types

You must specify types of variables as with constants. BASIC treats variables having the same variable name but with and without a type declaration character as different variables. The type of a variable is identified by the type declaration character at the end of each variable. Variables with no type declaration character are assumed to be of single precision type.

You can declare a variable type without a type declaration character, by using type declaration statements such as DEFINT, DEFSTR, DEFSNG, and DEFDBL. For details on these statements, see Chapter 3 "COMMANDS AND STATEMENTS."

The BASIC type declaration characters for variables are described below.

① $: String variable
  Used to identify variables which are to be loaded with character strings.
  Example: A$ = "ABC"

② %: Integer variable
  Used to identify variables which are to be loaded with integers.
  Example: A% = 300

③ !: Single-precision variable
  Used to identify variables which are to be loaded with single-precision real numbers.
  Example: A! = 12.34

④ #: Double-precision variables
  Used to identify variables which are to be loaded with double-precision real number.
  Example: A# = 12.3400001525

(3) Array variable

When data items are to be processed in a program, as many variables as the data items must be prepared. When handling data items of similar type, it is often convenient to give a single name to the group of data items by using variable to refer to them collectively. In this case, each data item can be identified using one or more index expressions. A variable used to refer to such a group or table of data items is called an array variable. The number of data items (elements) of an array variable is specified by placing the number (for a single-dimension table) enclosed in parentheses after the variable name. Array variables are declared by using the DIM statement. For example, a group of ten string variables can be collectively allocated by declaring an array variable of ten elements by using the DIM A$(9) statement. This statement has the same effect as declaring ten separately named string variables A$(0) to A$(9).

The number enclosed in parentheses is called a subscript of the array variable. You can specify the base of the subscript, i.e., whether the subscript begins with 0 or 1, by using the OPTION BASE command. The default base value is 0 (see the description of the OPTION BASE command).

A$(0)	A$(1)	A$(2)	A$(3)	A$(4)	A$(5)	A$(6)	A$(7)	A$(8)	A$(9)

DIM statements for declaring array variables should normally be placed at the beginning of the program.

The ERASE command is used to clear the memory area allocated for array variables.

In the above example, a one-dimensional array variable is shown. You can declare a two-dimensional array variables by specifying two subscripts enclosed in parentheses in the DIM statement. For example, the statement DIM A$(20, 20) causes BASIC to reserve a memory area for 21 × 21 string variables assuming that an OPTION BASE 0 statement has been executed.

A$(0.0)	A$(0.1)	A$(0.2)	....	A$(0.20)
A$(1.0)	A$(1.1)	·	....	
		⋮		
⋮	⋮	⋮	⋮	

You can reserve as large an area for array variables as memory permits. You may also use integer type variables as array subscripts.

BASIC assumes a maximum subscript value of 10 when an array variable is used without the use of a DIM statement.

(4) Type conversion

BASIC will perform type conversion as required when loading numeric values into numeric variables, except between string and numeric type data. If an attempt is made to convert between string and numeric type variables, BASIC displays an error message "TM Error" on the screen and terminates processing.

① When loading numeric variables with constants of another type

Example 1:

```
10 A%=55.8
20 PRINT A%
RUN
 56
Ok
```

When a real type constant is loaded into an integer type variable, it is converted to an integer type by being rounded to the tenth place.

Example 2:

```
10 A#=12.34
20 PRINT A#
run
 12.34000015258789
Ok
```

② Arithmetic operations

Example 1:

```
10 ? 10/3
20 ? 10/3*3#
RUN
 3.33333
 9.999997615814 21
Ok
```

The equation 10/3 is performed in single precision. Its result is converted to double precision before being multiplied by 3, that is, the result is converted to the higher degree of precision.

Example 2:

```
10 PRINT 10#/3
20 PRINT 10#/3*3

RUN
 3.333333333333333
 10
Ok
```

The arithmetic 10/3 is performed in double precision. The precision of the result is compared with that of the multiplier 3; consequently, the multiplication is carried out at the higher precision, i.e., double precision.

Example 3:

```
10 A!=10/3
20 B#=10#/3
30 IF A!>B# THEN 50 ———— Comparison is performed at the higher precision,
40 PRINT "10#/3=";B# : END  i.e., double precision
50 PRINT "10/3= ";A!

RUN
10#/3= 3.333333333333333
Ok

PRINT A!
 3.33333
Ok
A#=A!
PRINT A#
 3.333333253860474
Ok
```

③Type conversion in logical operations

When a noninteger numeric value is issued in an operand of a logical operation, it is converted to integer type by being rounded to the tenth place before the operation begins.

```
10 A=10.5 AND 12.3
20 PRINT A
RUN
 8
Ok
```

2.3 Arithmetic Operations

(1) Expressions

Expressions are defined as variables or constants, or combinations of variables and constants combined with operators to yield a single value.

BASIC provides the following five operators:

- Arithmetic operators
- Relational operators
- Logical operators
- Functional operators
- String operators

(2) Operator types

① Arithmetic

Operator	Operation	Sample expression	Explanation
∧	Exponentiation	A∧B	Places A to the power of B. See (i).
−	Negation	−A	
*	Multiplication	A*B	
/	Division	A/B	See (i).
+	Addition	A+B	
−	Subtraction	A−B	
MOD	Remainder of division	A MOD B	Gives the remainder of A divided by B. See (ii).
\	Integer division	A\B	Gives the quotient of a division, with a truncated fraction. See (iii).

The precedence of the arithmetic operators are as follows:
1. Parentheses
2. Function
3. Exponentiation
4. Signs (unary +, unary −)
5. *, /, \, MOD
6. +, − (binary)

(i) When a division by 0 is performed, BASIC displays a "/0 Error" message on the screen and continues processing. It returns the maximum value that the computer can handle. The same result will be returned if an exponentiation operation results in zero being raised to a negative power.

Example 1:

```
10 A=10/0
20 PRINT A

RUN
/0 Error
 1.70141E+38
Ok
```

Example 2:

```
10 A=0^-2
20 PRINT A

Ok
RUN
/0 Error
 1.70141E+38
Ok
```

(ii) When the divisor or dividend is noninteger, it is rounded to the tenth place, to form an integer before the division is carried out.

```
10 A=10.5 MOD 2.5
20 PRINT A

RUN
 2
Ok
```

(iii) When the divisor or dividend of an integer division is a noninteger, it is rounded to the tenth place to form an integer before the division is carried out, and only the integer part of the quotient is returned.

Example 3:

```
10 A=10¥2.5
20 PRINT A

run
 3
Ok
```

(iv) When the result of an operation cannot fit in the memory area reserved for the variable storing the result, BASIC displays an "OV Error" message on the screen, and continues or terminates processing depending on the type of the operation.

```
10 A=5^100
20 PRINT A

RUN
OV Error
 1.70141E+38
Ok
```

②Relational operators

Operator	Sample expression	Explanation
=	A=B	A is equal to B.
< > or ><	A< > or A><B	6 A is not equal to B.
<	A<B	A is smaller than B
>	A>B	A is greater than B.
< = or <	A< = or A=<B	A is smaller than or equal to B.
> = or =<	A> = or A=>B	A is greater than or equal to B.

The result of relational operations is either true (−1) or false (0). Relational operations are used to alter the program flow depending on various conditions (see the descriptions of the IF...THEN...ELSE and IF...GOTO statements).

For example, in evaluating the expression

X + Y < (T − 1)/Z,

BASIC calculates X + Y and (T-1)/Z then compares the results of the preceding calculations. If the result of the former is smaller than that of the latter, the true value (− 1) is returned.

NOTE:
The equal sign is used as a relational operation only in the conditions clause of the IF statement. In addition to being used to indicate equality, the equal sign is also used to indicate the assignment of a value to a variable (see the LET statement).

Example:

```
10 B=7 : C=5
20 A=B>C
30 PRINT A
```

When the above program is executed, − 1 is displayed on the screen. On line number 20, the result (− 1 or 0) of the relational operation B > C is placed in A. Since B > C is true, when executed this program displays − 1 on the screen. Change line number 0 to:

```
10 B=5 : C=7
```

BASIC will display 0 because B < C is false.

③ Logical operators
Logical operators are used in two forms:
(i) < operand > logical operator < operand >
- When the < operand > is either a numeric constant or a variable, the logical operator performs bit manipulation on two or more operands according to the logical relation determined by the operator and returns either 1 or 0 for each bit.

  The logical operators in this form are:

NOT (Negation)

A	NOT A
1	0
0	1

AND (Logical product)

A	B	A AND B
1	1	1
1	0	0
0	1	0
0	0	0

OR (Logical sum)

A	B	A OR B
1	1	1
1	0	1
0	1	1
0	0	0

XOR (Exclusive or)

A	B	A XOR B
1	1	0
1	0	1
0	1	1
0	0	0

IMP (Implication)

A	B	A IMP B
1	1	1
1	0	0
0	1	1
0	0	1

EQV (Equivalence)

A	B	A EQV B
1	1	1
1	0	0
0	1	0
0	0	1

<operand> can take values from −32768 to 32767. Any values other than integers are rounded to integers.

The numeric value specified by the <operand> is converted to a 16-bit signed binary number in two's complement form before being submitted to the logical operation. Positive integers from 0 to 32767 are expressed as 0000000000000000 to 0111111111111111. Negative numbers from −1 to −32768 are expressed as 1111111111111111 to 1000000000000000. The bit in the leftmost position indicates the sign of the number. A 0 in this bit identifies a positive number and a 1 a negative number.

```
PRINT NOT 3                    3=0000 0000 0000 0011
-4                  ---------------------------------
                       NOT     3=1111 1111 1111 1100
Ok
```

```
PRINT 1 AND 2                  1=0000 0000 0000 0001
 0                             2=0000 0000 0000 0010
                    ---------------------------------
Ok                     1 AND   2=0000 0000 0000 0000
```

```
PRINT 3 AND 2                  3=0000 0000 0000 1111
 2                             2=0000 0000 0000 0010
                    ---------------------------------
Ok                     3 AND   2=0000 0000 0000 0010
```

```
PRINT 6 OR 5                   6=0000 0000 0000 0110
 7                             5=0000 0000 0000 0101
                    ---------------------------------
Ok                      6 OR   5=0000 0000 0000 0111
```

```
PRINT 2 XOR 5                  2=0000 0000 0000 0010
 7                             5=0000 0000 0000 0101
                    ---------------------------------
Ok                     2 XOR   5=0000 0000 0000 0111
```

```
PRINT 3 IMP 0                  3=0000 0000 0000 0011
-4                             0=0000 0000 0000 0000
                    ---------------------------------
Ok                     3 IMP   0=1111 1111 1111 1100
```

```
PRINT 2 EQV 4                  2=0000 0000 0000 0010
-7                             0=0000 0000 0000 0000
                    ---------------------------------
Ok                     2 EQV   4=1111 1111 1111 1001
```

(ii) <operand> logical operator <operand>

- When the expressions in the <operand> are connected by relational operators, the logical operator performs logical operations on the results (true or false) of the relational operations and returns a true (−1) or false (0) result. The logical operators in this form are the same as those given in (i).

NOT (Negation)

Result of relational operation(A)	NOT A
True (−1)	False (0)
False (0)	True (−1)

AND (Logical product)

Result of relational operation(A)	Result of relational operation(B)	A AND B
True (−1)	True (−1)	True (−1)
True (−1)	False	(0)False (0)
False (0)	True (−1)	False (0)
False (0)	False (0)	False (0)

OR (Logical sum)

Result of relational operation(A)	Result of relational operation(B)	A AND B
True (−1)	True (−1)	True (−1)
True (−1)	False (0)	True (−1)
False (0)	True (−1)	True (−1)
False (0)	False (0)	False (0)

XOR (Exclusive or)

Result of relational operation(A)	Result of relational operation(B)	A AND B
True (−1)	True (−1)	False (0)
True (−1)	False (0)	True (−1)
False (0)	True (−1)	True (−1)
False (0)	False (0)	False (0)

IMP

Result of relational operation(A)	Result of relational operation(B)	A AND B
True (−1)	True (−1)	True (−1)
True (−1)	False (0)	False (0)
False (0)	True (−1)	True (−1)
False (0)	False (0)	True (−1)

EQV (Equivalence)

Result of relational operation(A)	Result of relational operation(B)	A AND B
True (−1)	True (−1)	True (−1)
True (−1)	False (0)	False (0)
False (0)	True (−1)	False (0)
False (0)	False (0)	True (−1)

```
100 IF D<200 AND F<4 THEN 80
```

The above program line causes control to transfer to line 80 because the results of relational operations D<200 and F<4 are both true and the logical operation performed on these results also gives a true value.

```
100 IF J<10 OR K<0 THEN 50
```

The above program line causes control to transfer to line 50 because the result of the OR operation is true if either J<4 or K<0 is true.

④Functions

A function performs a predefined operation on the given parameters and returns the result of the operation. For example, SIN(X) returns the sine of numeric value X in radians. PX-4 BASIC provides a number of functions. The BASIC functions are described in detail in Chapter 3.

You can define your own functions by using the DEF FN statement. See Chapter 3 for further information.

⑤String operators

(i) The "+" string operator concatenates character strings together.

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

RUN
FILENAME
NEW FILE NAME
Ok
```

(ii) Character strings can also be compared in the same way as numeric values are, by using relational operators. BASIC compares two strings by matching them one character at a time starting at the leftmost character position. It regards two characters which match in all character positions as equal. If a mismatch is found in a character position, BASIC terminates the comparison and regards the character string with the higher character code value in that character position as the larger string. When one character string is phsically shorter than the other, and the character positions of both strings match, the longer character string is regarded as larger string. Blanks are also compared in the magnitude of their code (see Appendix J, "CHARACTER CODES").

Example 1:

"AA"<"AB"

"FILENAME"="FILENAME"

"X$">"X#"

"kg">"KG"

"SMYTH"<"SMYHE"

Example 2:

```
10 A$="alpha"
20 B$="beta"
30 IF A$>B$ THEN 60
40 PRINT A$;" is lower than ";B$
50 END
60 PRINT B$;" is lower than ";A$

RUN
alpha is lower than beta
Ok
```

2.4 Files

The PX-4 computer treats its input/output devices as files. This section briefly describes the input/output devices that the PX-4 can handle and also shows how BASIC identifies them.

2.4.1 File Specification

A file is identified by means of a file specification which consists of a drive name, a file name, and a file extension.

(1) Drive name

The drive name identifies the input/output device on which a file is to be mounted for processing. The default drive is the logged in drive.

The relationship between the input/output devices and the drive names is shown below.

A:	RAM disk
B:	ROM capsule-1
C:	ROM capsule-2
D:,E:,F:,G:	Floppy disk
H:	Microcassette
I:	RAM cartridge
J:	ROM cartridge-1
K:	ROM cartridge-2
SCRN:	LCD display
LPT0:	Printer
COM0:	RS-232C
COM1:	SIO
COM2:	RS232C SIO
COM3:	Cartridge serial
KYBD:	Keyboard
CAS0:	External cassette

(2) File name
The file name identifies a file on a single input/output device. A file name is made up of one to eight alphanumeric characters followed by a period and a 1- to 3-character file extension (or file type). When we refer to a file name, we normally include the file extension.

You need not specify a file name for the keyboard, RS-232C port, and those input/output devices which cannot handle more than one file at a time.

(3) File extension
The file extension is usually used to indicate the use of a file (e.g., for storing programs or data). The file extension can consist of one to three alphanumeric characters. The LOAD and SAVE commands assume a file extension of BAS when no file extension is specified. The file name and extension must be separated by a period.

Example:

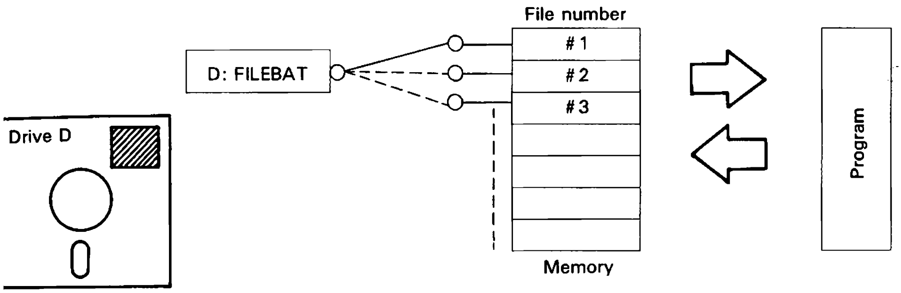
```
SAVE "A:PROG1.BAS"
```

The above command saves the program in the currently selected program area onto the RAM disk with a file name of PROG1 and a file extension of BAS.

2.4.2 File number

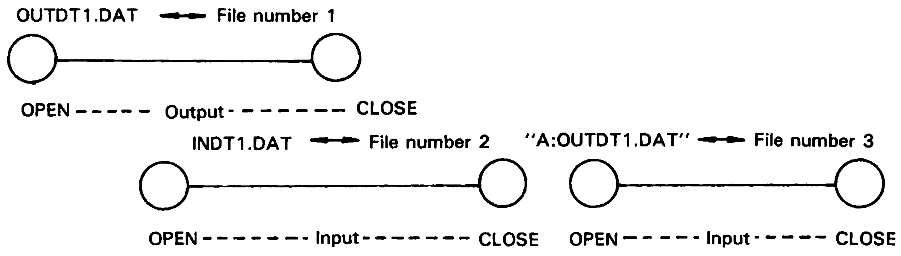
A file must be given a file number by the OPEN statement when it is to be used by a program for input/output processing. The program performs input/output operations to and from the file via the memory area (buffer) associated with the file number that is uniquely assigned to the file. The association between the file and the file number is released by the CLOSE statement.

File numbers begin with 1. The maximum file number is determined by the /F: < number of files > parameter that is specified when BASIC is cold-started. 3 is assumed when the /F: parameter is omitted.



Let us consider an example program for handling input/output. If the /F: parameter (i.e, the number of files that can be opened at a time) is specified as 4 at BASIC start-up time, we can specify integers 1 to 4 as file numbers.

①The first code segment creates a file on the RAM disk with a file name of OUTDT1.DAT, and we have decided to make the file number 1.
②The second code segment reads data from the file INDT1.DATA on the floppy disk. Since file number 1 is already being used for the output file, we must select a second file number for this file from file numbers 2, 3, and 4. Here we have selected file number 2.
③The third code segment reads data from the file OUTDT1.DAT on the RAM disk. We can use any number from 1 to 4 for this file. We used file number 3.



```
100 OPEN "O",#1,"A:OUTDT1.DAT"
 .         .
 .         .
 .         .
 .         .
 .         .
200 OPEN "I",#2,"A:INDT1.DAT"
 .         .
 .         .
250 CLOSE #1
 .         .
 .         .
300 CLOSE #2
 .         .
 .         .
340 OPEN "I",#3,"A:OUTDT1.DAT"
 .         .
 .         .
 .         .
 .         .
 .         .
 .         .
 .         .
 .         .
500 CLOSE #3
```

2.4.3 Input/output devices

The PX-4 computer is furnished with a number of input/output devices. These devices can be accessed by using standardized I/O statements and functions. A brief description of the PX-4 I/O devices follows.

①LCD display
The LCD display is an write-only device which is used to show data and program contents. When using it as a file, specify SCRN: as the drive name.

②Keyboard
The keyboard is an read-only device used to receive data. When using it as a file, specify KYBD: as the drive name.

③RAM disk
The RAM disk is used to store programs and data. The RAM disk is part of the PX-4 main memory which simulates a floppy disk but provides far higher input/output processing speed than normal floppy disk drives. Its drive name is A:. The size of the RAM disk can be specified during system initialization. The contents of the RAM disk are preserved when the PX-4 is powered off.

④RS-232C, SIO, and cartridge serial ports
The PX-4 computer can exchange data with a variety of external devices (e.g., QX-10, PX-8, etc.) via their communications facilities such as the RS-232C, SIO, and cartridge serial interfaces. See Chapter 6, "Data Communications Facilities" for details. The drive names COM0: to COM3: are reserved for these interfaces.

⑤ROM capsule
The ROM capsule is read-only device which reads programs and data stored in the computer. It may also be used as part of main memory. The drive name of the ROM capsule is B: or C:.

2.4.4 Other peripheral devices

The PX-4 computer also includes a buzzer and a clock. These devices cannot be used as files so they are not assigned drive names. The following statements and functions are available for these devices:

Buzzer:	BEEP, SOUND
Clock:	TIME$, DATE$, DAY

2.4.5 Microcassette drive (Optional)

The optional microcassette can be used to store data and programs in the same way as the RAM disk. Not only as a storage device, but also as an input/output device. The microcassette is assigned the drive name of H:.

The PX-4 controls files on the microcassette by using a directory (a table used to store information about the files). It reads this directory into memory from the microcassette tape whenever performing an I/O operation (i.e., read or write) on the microcassette. This procedure is known as mounting. Once the directory is loaded, the PX-4 accesses the files on the microcassette based on the directory information in memory. It alters the contents of the directory in memory each time it writes data onto the microcassette tape. This means that you have to save the contents of the directory back onto the tape before removing the microcassette from the cassette deck following an I/O operation. This procedure is known as remove. The mount and remove procedures can be carried out in the system display mode; however, they may also be performed from within a program by using the BASIC REMOVE and MOUNT commands.

The directory of a new microcassette tape must be initialized (DIRINIT) before use. For detailed handling procedures of the microcassettes, refer to Section 2.8, "Microcassette Handling" in the PX-4 Operating Manual.

Although microcassettes can be opened in the random mode, they do not allow accesss. I/O operations on microcassette random data files must be carried out in record number sequence. The PX-4 can access only one microcassette file at a time. For example, if one microcassette file is opened in the input mode and then another microcassette file is opened, the PX-4 can only access the latter file. Once a file is opened for output, no other files can be opened until the opened file is closed.

The file specification for microcassette files has the following format:

[H:][(<options>)][<file name>][.<extension>]

You can specify two types of options for microcassette files:

① $\left\{\begin{array}{l}\text{S: Stop mode}\\ \\ \text{N: Nonstop mode}\end{array}\right.$

When this option is omitted, the microcassette file is processed as follows:

(i) When the file is opened in the input ("I") mode
Data is read from the file in the mode in which the data was written.
(ii) When the file is opened in the output ("O") mode
Data is written to the file in the stop mode.
(iii) When the SAVE command is used
Data is written to the file in the nonstop mode.
(iv) When the random ("R") mode is specified in the OPEN statement
If the file name specified in the OPEN statement exists on the microcassette, data is read from that file in the mode in which the data was written. If the OPEN statement is used to create a new file, data is written to the file in the stop mode.

② V: Specifies that when the file is closed, a CRC check is to be made after rewinding the tape (verify mode).
N: Specifies that no CRC check is to be made (nonverify mode). When this option is omitted, the mode specified in the system display mode is becomes active.

Example:

```
SAVE "H:(SV)TEST.BAS"
```

When specifying only option ②, place a blank in the position where option ① is to be specified.

NOTE:
When an item keyboard is installed on the PX-4, the system display cannot be obtained, so the microcassette drive cannot be controlled manually.

2.4.6 RAM cartridge (Optional)

The RAM cartridge is an external storage device which allows both input and output. Like the RAM disk in the main unit, it is used to store data and programs. The RAM cartridge is optional and connected to the main unit via a cartridge interface.

Since the RAM cartridge is battery-backed up, it can preserve data for up to one year when detached from the main unit. The RAM cartridge has a capacity of 16KB and the assigned drive name is I:.

2.4.7 Printer (Optional)

This optional printer is a write-only device which is used to print data and program listings. It is assigned the drive name LPT0: when used as a file.
See Appendix N.

2.4.8 Floppy disk (Optional)

Floppy disks are commonly used auxiliary storage devices for storing data and programs. They can be used for both input and output operations. The floppy disk drives for the PX-4 are given the drive names D:, E:, F:, and G:.

2.4.9 ROM cartridges (Optional)

ROM cartridges are read-only external storage devices which are attached to the main unit via the cartridge interface. The drive names J: and K: are reserved for the ROM cartridges.

2.4.10 External audio cassette

The PX-4 can use a commercial audio cassette tape recorder as an external storage device by connecting it with an optional cable (#732). The audio cassette tape recorder is used as a sequential input/output unit and assigned the drive name CAS0:. Its file specification has the format:

```
CAS0:[<option>][<file name>][.<extension>]
```

The option must be either the stop or nonstop mode.

{ S: Stop mode
{ N: Nonstop mode

The audio cassette recorder allows no manual operation; it is only controlled through BASIC commands. The BASIC commands that support the audio cassette record are: OPEN, CLOSE, INPUT #, LINE INPUT #, PRINT #, PRINT USING, SAVE, LOAD, LOAD?, LIST, RUN, MERGE, MOTOR, BLOAD, BSAVE, EOF, and INPUT$.

Examples:

```
SAVE "CAS0:PROGRAM.BAS"
SAVE "CAS0:(S)PROGRAM.BAS"
```

• Connecting a cassette recorder to the PX-4

The REM (remote) terminal need not be used when saving onto or loading from a cassette tape. When handling a data file, however, data cannot be written to the cassette file unless the REM terminal is used and the S (stop mode) option is specified in the file specification. Since the PX-4 reads data from a cassette file one block at a time, it must be able to control the starting and stopping the cassette recorder with the REM terminal in order to position the read/write head at the next block.

When the REM terminal is activated, the PLAY button on the cassette recorder is disabled and tape will not move when the PLAY button is pressed. To start the recorder when the REM terminal is used and the PLAY button is pressed down, execute the MOTOR ON command. To stop the tape, execute the MOTOR OFF command.

• Saving a program onto cassette tape

To save a program onto a cassette tape, press down the REC and PLAY buttons on the tape recorder then execute the following command:

SAVE "CAS0:[(<option>)]<file name>[.<file extension>]"[,A | P]

BAS is assumed if <file extension> is omitted and nonstop mode (N) is assumed if <option> is omitted.

• **Program verification**
After a program is saved on a cassette tape, a check should be made to verify whether the program has been saved properly. To do this, rewind the tape up to the point where the program saving started (by visually checking the tape counter), then press down the PLAY button, and execute the command:

LOAD?[“CAS0:[<file name>.<file extension>]”]

Program verification is not achieved by comparing the program in memory with that which has been saved on tape. Instead, the LOAD? command loads the saved program while making CRC checks, and displays the error message “IO Error” on the screen if it detects a CRC check error.

• **Loading a program**
To load a program, rewind the tape up to the point where the program was saved, press down the PLAY button, and execute the command:

LOAD[“CAS0:[<file name>.<file extension>]”]

BASIC will load the program into memory in the mode in which it was saved if <option> is omitted. If <file name>.<file extension> is omitted, BASIC will load the first file found.

• **Input/output to and from a data file**
When <option> is omitted in an I/O statement executed for a data file, if the file is opened in the input (“I”) open mode, it will be read in the mode in which the file was created. The file will be written in the stop (S) mode if the file has been opened in the output (“O”) open mode.

2.5 Error Messages

BASIC displays error messages whenever it detects errors during execution of BASIC statements, commands, and functions. If an error is detected while executing a BASIC program, BASIC immediately interrupts the program execution and returns to the command mode. You can prevent your program from being interrupted by such errors by including in your program some error handling routines which use the ON ERROR statement and the ERROR and ERL functions. See Appendix K, ‘‘ERROR CODES AND MESSAGES’’ for a full description of the BASIC error codes and messages.

Chapter 3

COMMANDS AND STATEMENTS

This Chapter describes the commands, statements, and functions used with PX-4 BASIC.

Commands and statements are words in the BASIC language which control operation of the computer or which set up parameters which are manipulated during computer operation. The distinction between commands and statements is as follows.

Commands — Generally executed in the direct mode, and used in manipulating BASIC program files. The LOAD command, which is used to bring a program into memory from external storage, is an example of a command.

Statements — Instructions which are included in a program to control operation of a computer or establish parameters which are manipulated during program operation. For example, execution of the GOTO statement causes execution to branch from one part of a program to another.

In practice, most commands and statements can be executed in either the direct mode or the indirect (program execute) mode, so the distinction between them is more traditional than qualitative.

Functions are procedures built into the BASIC language which return specific results for given data. Functions differ from commands and statements in that the former controls operation of the computer, while the latter produces a result and passes it to the program. An example is the SIN function, which returns the sine of a specified value. Functions may be used at any time, either from within a program or in the direct mode; there is no need for definition on the part of the user.

The following format is used in describing commands, statements, and functions in this Chapter.

Format	Illustrates the general format for specification of the statement or function concerned. Meanings of the symbols used in the format descriptions are described in "Format Notations" below.
Purpose	Explains the purpose of the statement or function.
Remarks	Gives detailed instructions for using the statement or function.
See also	Refers the reader to descriptions of other statements or functions whose operation is related in some way to that of the statement or function being described.
Example	Gives examples of use of the statement or function in programs.

NOTE:
Outlines precautions which should be observed when using the statement/function, or presents other related information.

Format Notations

The following rules apply to specification of commands, statements, and functions.

(1) Items shown in capital letters are BASIC reserved words, and must be input letter for letter exactly as shown. Any combination of upper- or lowercase letters can be used to enter reserved words; BASIC automatically converts lowercase letters to uppercase except when they are included between quotation marks or in a remark statement.

e.g. **CLS PRINT STEP**
PRINT
STEP

(2) Angle brackets "< >" indicate items which must be specified by the user.

(3) Items in square brackets "[ ]" are optional.

In either case the brackets themselves have no meaning except as format notation, and should NOT be included when the statement/function is entered. If angle brackets are included inside square brackets, this means optional items to be specified by the user.

e.g. **PSET [STEP] (X,Y),<function code>**

might be typed to appear with various values as follows. These are particular cases, to show what is actually typed.

PSET (1Ø,1Ø)
is the minimum format for plotting a point at position (10,10) on the screen.

PSET (1Ø,1Ø), 1
plots the same point, but with <function code> having a value of 1.

PSET STEP (1Ø,1Ø)
plots a point relative to the last plotted point.

PSET STEP (1Ø,1Ø),7
plots the point relative to the last plotted point, with <function code> set to 7.

ALARM [<date>,<time>,<string> [,W]]
means that it is optional to input the <date>, <time> and a <string>; however, if it is required to use one or other of these three options, the others must be typed in as well. It is optional to use the "W" extension, but this option cannot be used without the other three options. Examples of the three valid types of statement are

ALARM
ALARM "/**/**","**:13:ØØ","LUNCH TIME"**
ALARM,"/**/**","**:Ø9:3Ø","APPOINTMENT",W**

(4) All punctuation marks (commas, parentheses, semicolons, hyphens, equal signs, and so forth) must be entered exactly as shown.
When round brackets ( ) are included they MUST be typed in as shown.

(5) Where a set of full stops "..." is included, the items may be repeated any number of times, provided the length of the logical line is not exceeded.

e.g. **CLOSE [[# <filenumber>][, # <filenumber>]....]**

means that any number of files can be closed. Valid examples are

```
CLOSE
CLOSE #4
CLOSE #1, #3, #4
```

(6) Items included between vertical bars are mutually exclusive; and only one of the items shown can be included when the statement is executed.

e.g. **STOP KEY** $\left| \begin{array}{l} \textbf{ON} \\ \textbf{OFF} \end{array} \right|$

The following abbreviations are used in explaining the arguments or parameters of commands, statements, and functions.

X or Y	Represent any numeric expressions.
J or K	Represent integer expressions.
X$ or Y$	Represent string expressions.

With functions, any floating point value specified as an argument will be automatically rounded to the nearest integer value if the function in question only works with integer values.

ABS

Format **ABS (X)**

Purpose Returns the absolute value of expression X.

Remarks Any numeric expression may be specified for X.

Example

```
10 CLS
20 A = 25
30 B = -25
40 C = 2.545
50 D = -2.545
60 PRINT "VARIABLE","VALUE","ABSOLUTE VALUE"
70 PRINT "A",A,ABS(A)
80 PRINT "B",B,ABS(B)
90 PRINT "C",C,ABS(C)
100 PRINT "D",D,ABS(D)
```

```
VARIABLE      VALUE         ABSOLUTE VALUE
A              25            25
B             -25            25
C              2.545         2.545
D             -2.545         2.545
Ok
```

ALARM

Format **ALARM [<date>,<time>,<message>[,W]]**

Purpose Specifies the alarm or wake time. Only one of these can be set at a time.

Remarks Executing the ALARM statement without the W option sets the alarm time, and executing it with the W option sets the wake time. An alarm or wake time set with the ALARM statement is the same as one specified from the System Display; execution of an ALARM statement will cancel any alarm or wake time setting made from the System Display, and vice versa. It is not possible to set both an alarm and wake time simultaneously.

The alarm or wake date is specified in <date> in the same format as with DATE$ (a system variable), and the alarm/wake time is specified in <time> in the same format as with TIME$.

Asterisks can be specified as wildcard characters for any of the digits in <date> or <time>. When asterisks are specified, those positions in <date> and/or <time> will be regarded as always matching the corresponding digit in the DATE$ or TIME$ system variable. For example, executing the following statement will result in alarm operation every day at ten minute intervals from 8:00 AM to 8:50 AM.

```
ALARM "**/**/**","ø8:*ø:ø*",A$
```

Note that the two year digits are always handled as if they were specified as asterisks, and that the lower second digit is always handled as if it were specified as "0".

<message> must be specified as a string expression whose result is no more than 32 characters; this message is displayed on the System Display when the alarm time is reached, or is assumed as the auto start string when the wake time is reached.

The alarm or wake time setting can be cleared by executing the ALARM statement without specifying any parameters.

MO error (Missing operand) — A required operand was not specified in the statement.

FC error (Illegal function call) — The <date> or <time> parameters were incorrectly specified.

See also **ALARM$, AUTO START, POWER**

ALARM$

Format **ALARM$ (<function>)**

Purpose Used to check the information set by the ALARM statement.

Remarks <function> is specified as a numeric expression whose result is a value from 0 to 2. The value returned by the ALARM$ function varies according to <function> as follows.

0: Returns the status of the setting made by the ALARM statement as a 1-character string. Characters returned and their meanings are as follows.

"N" —	No alarm setting has been made.
"B" —	An alarm setting has been made, but the specified time has not yet been reached.
"P" —	An alarm setting has been made and the specified time has been reached.

1: Returns the date set by the ALARM statement in the same format as the date returned by the DATE$ function.

2: Returns the time set by the ALARM statement in the same format as the time returned by the TIME$ function.

3: Returns the message set by the ALARM statement as a character string.

Note that, once "P" has been returned by executing ALARM$(0), "B" is returned when ALARM$(0) is subsequently executed.

FC error (Illegal function call) — The value specified for <function> was outside the prescribed range.

See also **ALARM, AUTO START, POWER**

ASC

Format **ASC(X$)**

Purpose Returns the numeric value which is the ASCII code for the first character of string X$. (See Appendix F for the ASCII codes.)

Remarks X$ must be a string expression. An FC error (Illegal function call) will occur if X$ is a null string (a string variable which contains no data, or a pair of quotation marks without any intervening characters or spaces).

See also **CHR$**

Example

```
10 CLS
20 A$ = "A"
30 B$ = "BOO"
40 C$ = "1234"
50 D$ = ""
60 PRINT "STRING", "ASCII value of first letter"
70 PRINT A$, ASC(A$)
80 PRINT B$, ASC(B$)
90 PRINT C$, ASC(C$)
100 PRINT D$, ASC(D$)

STRING        ASCII value of first letter
A              65
BOO            66
1234           49

FC Error in 100
Ok
```

ATN

Format ATN(X)

Purpose Returns the arc tangent in radians of X.

Remarks This function returns an angle in radians for expression X as a value from $-\pi/2$ to $\pi/2$. The angle will be returned as a double precision number if X is a double precision number, and as a single precision number if X is a single precision number or an integer. ATN(X) can also be used to derive a value for the constant PI. From elementary trigonometry PI = 4 * ATN(1).

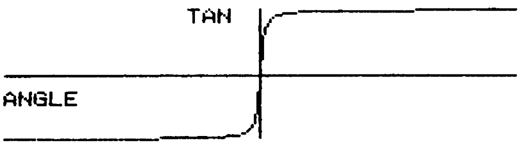
As PI times radius equals 180 degrees, conversion of radians to degrees, is a matter of simple proportion. Lines 100 and 110 in Example 1 show how to obtain angles in degrees.

Example 1

```
10 CLS
20 INPUT "Type in the tangent of an angle" ; T
30 Y = ATN(T)
40 PRINT "The angle whose tangent is";T "is";Y;"RADIANS"
100 PI = 4 * ATN(1)
110 Z = Y*180/PI
120 PRINT "The angle whose tangent is";T "is";Z;"DEGREES"

Type in the tangent of an angle? 0.7071
The angle whose tangent is .7071 is .615475 RADIANS
The angle whose tangent is .7071 is 35.2641 DEGREES
Ok
```

Example 2

```
10 'Graphic representation of angles whose tangents
20 'range from -99 to 100.  Range of angles is from
30 '-1.5607 radians to +1.5608 radians.
40 WIDTH 40,8:CLS
50 LINE (100,0)-(100,62)
60 LINE (0,32)-(200,32)
70 LOCATE 1,6:PRINT"ANGLE"
80 LOCATE 13,1:PRINT"TAN"
90 I=-100
100 X=I+100
110 Y=63-(ATN(I)+1.5708)*20.3718
120 PSET(X,Y)
130 FOR I=-99 TO 100 STEP 1
140 X=I+100
150 Y=63-(ATN(I)+1.5708)*20
160 LINE -(X,Y)
170 NEXT
```



AUTO

Format **AUTO [<line number>][,[<increment>]]**

Purpose Entered in the direct mode to initiate automatic program line number generation during program entry.

Remarks Executing this command causes program line numbers to be generated each time the RETURN key is pressed to complete the input of a program line. Numbering starts at <line number>, and subsequent line numbers differ by the value specified for <increment>. If either <line number> or <increment> is omitted, 10 is assumed as the default value; however, if a comma is specified following <line number> but no <increment> is specified, the increment specified for the last previous AUTO command is assumed.

If the currently selected program area already contains a line whose line number is the same as one generated by AUTO numbering, an asterisk (*) is displayed immediately following the number to warn the user that that line contains statements. If the RETURN key is pressed without entering any characters, the line is skipped without affecting its current contents; if any characters are entered before the RETURN key is pressed, the former contents of the line are replaced with the characters entered.

AUTO line number generation is terminated and BASIC returned to the command level by pressing CTRL and C or the STOP key; the contents of the last line number displayed at this time are not stored in the program area.

Example 1 **AUTO 1ØØ,5Ø**
Generates line numbers in increments of 50, starting with line number 100. (100, 150, 200 ...)

Example 2 **AUTO**
Generates line numbers in increments of 10, starting with line number 10. (10, 20, 30...)

AUTO START

Format **AUTO START <auto start string>**

Purpose The AUTO START statement is used to specify parameters which are passed to BASIC when a hot start is made.

Remarks The <auto start string> is specified as a string expression whose result is no more than 32 characters. When the power switch is turned on and BASIC is started by a hot start, this string is passed to BASIC in the same manner as if it were input from the keyboard. If a null string is specified for <auto start string>, the auto start function is cancelled.

BEEP

Format **BEEP [<duration>]**

Purpose Causes the speaker to beep.

Remarks The BEEP statement causes the speaker built into the keyboard to make a beeping sound. The numeric value specified in <duration> determines the length of the sound; numbers specified must be in the range from 0 to 255. The length of the sound generated is equal to <duration> × 10 msec, with the result rounded off to the nearest millisecond.

If <duration> is omitted, "10" is assumed.

BLOAD

Format **BLOAD <file descriptor>,[,[<load address>][,R]]**

Purpose Loads machine language programs or data.

Remarks This statement is used to load the machine language program or data file specified in <file descriptor> into memory. Ordinarily, the load address is the same as that specified when the file was saved with the BSAVE statement; however, the file can be loaded into a different area by specifying the starting address of that area in <load address>. Therefore, a machine language program which is loaded into an area other than that specified when it was saved must be fully relocatable (it must be capable of being executed properly in a different area in memory).

When the R option is specified, program execution begins immediately after loading has been completed. Any files which are open at the time remain open. If <load address> is not specified, execution begins at the starting address which was specified when the program was saved; otherwise, execution begins at the address specified in <load address>.

This statement will only load programs or data into the area from the address following that specified in the CLEAR statement to that immediately preceding the beginning of BDOS (or to the beginning of the user BIOS area, if any).

This statement can also be used to load user-defined character font files.

FC error (Illegal function call) — An attempt was made to load a file into an inhibited area.

BSAVE

Format **BSAVE <file descriptor>,<starting address>,<length>**

Purpose Saves machine language programs or data to files.

Remarks This statement saves the contents of the machine language programs or data files from memory to the file specified in <file descriptor>. The number of bytes specified in <length> is saved, starting at the address specified in <starting address>.

If the machine language program is written so that execution can begin at the address specified in <starting address>, the program can be executed immediately upon loading with the BLOAD statement.

This statement can also be used to save user-defined character fonts.

If CAS0: is specified as the device in <file descriptor>, the save can be verified by executing the LOAD? statement.

CALL

Format **CALL <variable name>[(<argument list>)]**

Purpose Calls a machine language subroutine.

Remarks The CALL statement is one method of transferring BASIC program execution to a machine language subroutine. (See also the discussion of the USR function.) <variable name> is the name of a variable which indicates the machine language subroutine's starting address in memory. The starting address must be specified as a variable (not as a numeric expression), and the variable name specified must not be an element of an array. <argument list> is the list of parameters which is passed to the machine language subroutine by the calling program. See Appendix D for further details on use of the CALL statement.

See also **USR**, Appendix D

Example The following program is an example of the use of the CALL function. It simply increments by one the number in location &HC009, and then displays the new value found by the PEEK function in line 140.

```
C000   3A 09 C0  LD A, (C009)     ;Load register A with contents of
                                   location &HC009 which has been
                                   POKEd in by a BASIC program.
C003   C6 01     ADD A, 01H       ;Add 1 to it.
C005   32 09 C0  LD (C009), A     ;Move the contents of register A
                                   back to location &HC009.
C008   C9        RET              ;RETURN to BASIC.
C009   00        NOP              ;The value obtained by the BASIC
                                   program and the location used for it.

10 CLS
20 CLEAR ,&HBFFF
30 ADRS = &HC000
40 FOR J = 0 TO 9
50 READ A
60 POKE ADRS+J,A
70 NEXT J
90 DATA &h3a, &h09, &hc0, &hc6, &h01, &h32, &h09, &hc0, &hc9
, &h00
110 INPUT "Type in a number in the range 1 to 254";B
120 POKE &HC009,B
130 CALL ADRS
140 C = PEEK(&HC009)
150 PRINT B "+ 1 ="; C

Type in a number in the range 1 to 254? 99
 99 + 1 = 100
Ok
```

CDBL

Format CDBL(X)

Purpose Converts numeric expression X to a double precision number.

Remarks This function converts the values of integer or single precision numeric expressions to double precision numbers. Significant decimal places added to converted numbers will contain random numbers.

Example

```
10 CLS
20 INPUT "TYPE IN TWO NUMBERS ";X,Y
30 PRINT "THE VALUE OF X multiplied by Y is ";X*Y
40 PRINT "CONVERTED TO DOUBLE PRECISION IT IS "; CDBL(X*Y)


run
TYPE IN TWO NUMBERS ? 3.45,3.141597
THE VALUE OF X multiplied by Y is  10.8385
CONVERTED TO DOUBLE PRECISION IT IS  10.83850955963135
Ok
```

CHAIN

Format **CHAIN [MERGE] <filename> [,[<line number exp>][,ALL] [,DELETE<range>]]**

Purpose Calls the BASIC program designated by <filename> and passes variables to it from the program currently being executed.

Remarks The CHAIN statement makes it possible for one BASIC program to call (load and execute) another one. <filename> is the name of the program being called by this statement. The program called may be stored on floppy disk, in RAM disk, or on microcassette tape. However, the program called must be one which is not contained in program memory (in another program area).

If the MERGE option is not specified, the program called replaces the calling program in the program area from which the call is made. If the MERGE option is specified, the program called is brought into program memory as an overlay; statements in program lines of the calling program are replaced by similarly numbered lines in the program called. In this case, the program called must be an ASCII file (see the explanation of the SAVE command). Since it is usually desirable to ensure that the range of program line numbers in the two programs are mutually exclusive, the DELETE option may be used to delete unneeded lines in the calling program.

<line number exp> is a variable or constant indicating the line number at which execution of the called program is to begin. If omitted, execution begins with the first line of the program called.

If the ALL option is specified, all variables being used by the calling program are passed to the program called. If the ALL option is omitted, the calling program must contain a COMMON statement to list variables that are to be passed. (See the explanation of the COMMON statement.)

Note that user defined functions and variable type definitions made with the DEFINT, DEFSNG, DEFDBL, or DEFSTR statements are not preserved if the MERGE option is omitted. Therefore, these statements must be restated in the program called that program is to use the corresponding variables or functions.

See also **COMMON, MERGE, SAVE**

Example 1 The first example shows how the chained program can replace the calling program but still preserve the variables.

```
150 'prog 2
160 X = X * X
170 PRINT "The values of X,Y, and Z from the chained program
are :- "
180 PRINT X,Y,Z
```

Save the above lines to the RAM disk using SAVE "A: PROG2",A. Delete them then type in and execute the following.

```
50 'The first program to be run
100 READ Y , Z
110 X = Y + Z
120 PRINT "The value of X,Y and Z from the first program
are:-"
130 PRINT X,Y,Z
140 DATA 2, 5
150 CHAIN "prog2",,ALL
```

```
run
The value of X,Y and Z from the first program are:-
 7             2             5
The values of X,Y, and Z from the chained program are :-
 49            2             5
Ok
```

Example 2 The second example shows how lines can be merged and lines of the original calling progam deleted. The line number from which the chained program is executed is also included in this example.

```
80 PRINT "This line is printed after line 100"
90 RETURN
100 PRINT "*** This is the chained program ***"
110 GOSUB 80
```

Save the above lines to the RAM disk using SAVE"A: SUB", A. Delete them then type in and execute the following.

```
200 PRINT "*** This is the first program *** ": PRINT
210 CHAIN MERGE "SUB",100,DELETE 200-210

Ok
run
*** This is the first program ***

*** This is the chained program ***
This line is printed after line 100
Ok
```

CHR$

Format **CHR$(J)**

Purpose Returns the character whose ASCII code equals the value of integer expression J. (See Appendix J for the ASCII codes.)

Remarks The CHR$ function is frequently used to send special characters to terminal equipment such as the console or printer. For example, executing PRINT CHR$(12) clears the entire virtual screen and returns the cursor to the home position; executing PRINT CHR$(7) causes the speaker to beep; and executing PRINT CHR$(11) moves the cursor to the home position (the upper left corner of the virtual screen) without clearing the screen. See the description of the ASC function for conversion of ASCII characters to numeric values.

It is also easier to program using numbers, so that often manipulations with ASCII codes are used in programs instead of using the actual alphabetic characters.

Example

```
10 PRINT CHR$(12)                        :'clear screen
20 FOR J = 65 TO 90                      :'Display characters A-Z
30 PRINT CHR$(J);
40 NEXT J
50 PRINT
60 FOR J = 97 TO 122                     :'Display characters a-z
70 PRINT CHR$(J);
80 NEXT J
90 '
100 PRINT CHR$(7)                        :'sound buzzer


ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
Ok
```

CINT

Format CINT(X)

Purpose Rounds the decimal portion of numeric expression (X) to the nearest whole number and returns the equivalent integer value.

Remarks X must be a numeric expression which, when rounded, is within the range from -32768 to +32767; otherwise, an "Overflow" error will occur.

See the descriptions of the FIX and INT functions for other methods of converting numbers to integers.

See the descriptions of the CDBL and CSNG functions for conversion of numeric expressions to double and single precision numbers.

NOTE:
Differences between the CINT, FIX, and INT functions are as follows.

CINT(X)	*Rounds X to the nearest integer value.*
FIX(X)	*Truncates the decimal fraction of X.*
INT(X)	*Returns the integer value which is less than or equal to X.*

Number	Result of Function		
	CINT	FIX	INT
−1.6	−2	−1	−2
−1.2	−1	−1	−2
1.2	1	1	1
1.6	2	1	1

Although numbers are printed to the screen as you would expect, they are not always stored as such in the computer. This can lead to erroneous results with INT(X) and FIX(X) as the following program shows:

```
10 CLS
20 K1 = 2.6 : K2 = .2
30 PRINT "X","X%","INT(X)","FIX(X)","CINT(X)"
40 N = 2 : GOSUB 100
50 N = 12: GOSUB 100
60 PRINT :PRINT "The value of X - INT(X) is "; X - INT (X)
80 END
90 '
95 'subroutine to print values
100 X = K1 * N - K2
110 X% = K1 * N - K2
120 PRINT X,X%,INT(X),FIX(X),CINT(X)
130 RETURN
```

```
X             X%            INT(X)        FIX(X)        CINT(X)
 5             5             5             5             5
 31            31            30            30            31

The value of X - INT(X) is  .999998
Ok
```

The values of INT(X) and FIX(X) are apparently wrong since simple mental arithmetic will show that 12 × 2.6 − 0.2 = 31. However, the output from line 60 shows that X is stored in the computer as 30.999998 and so the functions INT (X) and FIX (X) are returning logically correct values. The error is due to the fact that numbers are converted to and handled as binary numbers by the computer. Such rounding errors are overcome by adding a small number to the answer before executing INT (X) or FIX (X) — eg: if line 120 is altered to:

120 PRINT X, X%, INT (X+0.0005), FIX (X+0.0005), CINT (X)

all values would be correct.

```
10 CLS
20 K1 = 2.6 : K2 = .2
30 PRINT "X","X%","INT(X)","FIX(X)","CINT(X)"
40 N = 2 : GOSUB 100
50 N = 12: GOSUB 100
60 PRINT :PRINT "The value of X - INT(X) is "; X - INT (X)
80 END
90 '
95 'subroutine to print values
100 X = K1 * N - K2
110 X% = K1 * N - K2
120 PRINT X,X%,INT(X + .0005),FIX(X + .0005),CINT(X)
130 RETURN
```

```
X             X%            INT(X)        FIX(X)        CINT(X)
 5             5             5             5             5
 31            31            31            31            31

The value of X - INT(X) is  .999998
Ok
```

CLEAR

Format

CLEAR [[<dummy>][,[upper memory limit>][,<stack size>]]]

Purpose

Clears numeric variables to 0, sets null strings to all string variables, specifies the upper limit of the BASIC memory area, and sets the size of the stack area.

Remarks

This statement clears all numeric variables to 0 and sets all null strings in all string variables; at the same time, it cancels all variable type definitions made with the DEFINT, DEFSNG, DEFDBL, and DEFSTR statements and closes all files which are currently open.

<upper memory limit> specifies the highest address in memory which can be used by BASIC. The maximum value which can be specified is &H6000. This is the value which is effective when BASIC is started (unless a different address has been specified with the /M: option). The area from <upper memory limit> to the beginning of BDOS can be used for storage of machine language programs, user-defined character sets, or other data. The upper memory limit is not changed by execution of the CLEAR statement if this parameter is omitted.

The <stack size> parameter specifies the size of the stack area which is used by BASIC. The initial size is 256 bytes. If this parameter is omitted, the stack area size is not affected by execution of the CLEAR statement.

CLOSE

Format **CLOSE[[#]<file number>[,[#]<file number>...]]**

Purpose Executing this statement terminates access to device files.

Remarks This statement terminates access to files opened previously under specified file numbers. If no file numbers are specified, this statement closes all files which are currently open.

Once opened, a file must be closed before it can be reopened under a different file number or in a different mode, or before the file number under which that file was opened can be used to open a different file. An FE error (File already open) will occur if an attempt is made to open a file which is already open, or if an attempt is made to open a different file using the file number assigned to a file which is already open; a BF error (Bad file mode) will occur if an attempt is made to use a different file number to open a file which is already open.

Executing this statement to close a random file or a sequential file which has been opened in the output mode causes the contents of the output buffer to be written to the file's end. Therefore, be sure to close any disk or microcassette files which are open for output before removing the medium from the drive; otherwise, data stored in the file will not be usable, and there is a possibility that the contents of other files may be destroyed when a CLOSE statement is executed if another disk or magnetic tape is inserted in that drive.

All files are closed automatically upon execution of an END, CLEAR, or NEW statement.

See also **END, OPEN,** Chapter 4

Example

```
10 OPEN "O",#1, "A:TEST.DAT"      :' Opens the file "TEST"
20 '                                 for output on drive A:
30 FOR J = 65 TO 90
40 PRINT #1, CHR$(J) ;                :' Writes letters A to Z
50 NEXT J                         :' to the file
60 '
70 CLOSE #1                       :' Closes file "A:TEST.DAT"
80 '
90 OPEN "I" ,#1, "A:TEST.DAT"     :' Opens file for input
100 IF EOF (1) THEN 160           :' Checks whether End of File
110 '                                marker of file 1 has been
120 '                                reached, and if so goes to 160.
130 A$ = INPUT$(2,1)              :' Inputs two characters from file #1
140 PRINT A$ : GOTO 100               :' Prints the characters input
150 '                                from the file and returns for more.
160 END                           :' Ends program, Closing file
```

```
run
AB
CD
EF
GH
IJ
KL
MN
OP
QR
ST
UV
WX
YZ
Ok
```

CLS

Format	**CLS**
Purpose	Clears the screen.
Remarks	The CLS statement clears the virtual screen. This is the same as executing PRINT CHR$(12);.

COMMON

Format **COMMON <list of variables>**

Purpose Passes variables to a program executed with the CHAIN statement.

Remarks The COMMON statement is one method of passing variables from one program to another when execution of programs is chained with the CHAIN statement, the other being to specify the ALL option in the CHAIN statement of the calling program. The COMMON statement may be included anywhere in the calling program, but is usually placed near its beginning. More than one COMMON statement may be specified in a program, but the same variables cannot be specified in more than one COMMON statement. Array variables are specified by appending "( )" to the array name.

See also **CHAIN**

Example

```
10 PRINT "Main program"
20 A$ = "Tom"
30 B$ = "Dick"
40 C$ = "Harry"
50 COMMON A$,B$,C$
60 CHAIN "A:COMMON2"
```

```
10 PRINT "The COMMON statement passes 3 variables to this pr
ogram"
20 PRINT "The first is ";A$
30 PRINT "The second is ";B$
40 PRINT "The third is ";C$
50 END
```

```
Main program
The COMMON statement passes 3 variables to this program
The first is Tom
The second is Dick
The third is Harry
Ok
```

COM(n) ON/OFF/STOP

Format

```
COM(n) | ON   |
       | OFF  |
       | STOP |
```

Purpose

Enables, disables, or defers interrupts from the communication line.

Remarks

This statement enables, disables, or defers external interrupts from the communication line. The n in COM(n) indicates the communication port, and is specified as a number from 0 to 3. COM(n) ON enables interrupts. After this statement has been executed, the specified communication port is checked each time a statement is executed and, if any signal has been received, an interrupt is generated and processing branches to the communication trap routine designated by the ON COM(n) GOSUB statement.

COM(n) OFF disables communication interrupts. After this statement has been executed, processing does not branch to the communication trap even if an external signal is received.

COM(n) STOP defers generation of communication interrupts. If an external signal is received following execution of this statement, the event is noted but processing does not branch to the communication trap routine. However, processing does branch to the communication trap routine the next time interrupts are enabled by executing the COM(n) ON statement.

CONT

Format CONT

Purpose Resumes execution of a program which has been interrupted by a STOP or END statement, or by pressing CTRL + C or the STOP key.

Remarks This command causes program execution to resume at the point at which it was interrupted. If execution was interrupted while a prompt ("?" or a user-defined prompt string) was being displayed by an INPUT statement, the prompt is displayed again when program execution resumes.

The CONT command is often used together with the STOP command or the STOP key during programming debugging. When execution is interrupted by the STOP statement or the STOP key, statements can be executed in the direct mode to examine or change intermediate values, then execution can be resumed by executing CONT (or by executing a GOTO statement in the direct mode to resume execution at a different line number). The CONT statement can also be used to resume execution of a program which has been interrupted by an error; however, program execution cannot be resumed if any changes are made in the program while execution is stopped.

COPY

Format **COPY**

Purpose Copies the contents of the LCD screen to the printer.

Remarks The COPY statement outputs the contents of the LCD screen to the printer. This is the same as pressing CTRL + PF5 .

COS

Format **COS(X)**

Purpose Returns the cosine of angle X, where X is in radians.

Remarks The cosine of angle X is calculated to the precision of the type of numeric expression specified for X.

Example

```
PRINT COS (1.5)
.07073711
```

NOTE:
The value returned by this function will not be correct if (1) X is a single precision value which is greater than or equal to 2.7E7, or (2) if X is a double precision value which is greater than or equal to 1.2D17.

CSNG

Format CSNG(X)

Purpose Returns the single precision number obtained by conversion of the value of numeric expression X.

Remarks See the descriptions of the CDBL and CINT functions for conversion of numeric values to double precision or integer type numbers.

Example

```
PRINT CSNG(5.123456789#)
      5.12346
```

CSRLIN

Format **CSRLIN**

Purpose The CSRLIN function returns the current vertical coordinate of the cursor.

Remarks The value returned by the CSRLIN function indicates the current location of the cursor in the virtual screen. The meaning and range of values returned is the same as for the LOCATE statement.

CVI/CVS/CVD

Format

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

Purpose

These functions are used to convert string values into numeric values.

Remarks

Numeric values must be converted to string values for storage in random access files. This is done using the MKI$, MKS$, or MKD$ functions depending on whether the numeric value being converted is an integer, single precision number, or double precision number. When such strings are then read back in from the file, they must be converted back into numeric values for display or use as operands in numeric operations. This is done using the CVI, CVS and CVD functions.

CVI returns an integer for a 2-byte string, CVS returns a single precision number for a 4-byte string, and CVD returns a double precision number for an 8-byte string.

See also

MKI$/MKS$/MKD$, Chapter 4

Example

```
a$=mki$(12849)
Ok
?a$
12
Ok
?cvi(a$)
 12849
Ok
```

DATA

Format **DATA <list of constants>**

Purpose Lists numeric and/or string constants which are substituted into variables by the READ statement. (See the explanation of the READ statement.)

Remarks DATA statements are non-executable, and may be located anywhere in the program. Constants included in the list must be separated from each other by commas, and are substituted into variables upon execution of READ statements in the order in which they appear in the list. A program may include any number of DATA statements.

When more than one DATA statement is included in a program, they are accessed by READ statements in the order in which they appear (in program line number order); therefore, the lists of constants specified in DATA statements can be thought of as constituting one continuous list, regardless of the number of constants on each individual line or where the lines appear in the program.

Constants of any type (numeric or string) may be included in <list of constants>; however, the types of the constants must be the same as the types of variables into which they are to be substituted by the READ statements.

Numeric DATA statements can contain negative numbers but no operators. String constants must be enclosed in quotation marks if they include commas, colons or significant leading or trailing spaces; otherwise, quotation marks are not required.

Once the <list of constants> of a DATA statement has been read, it cannot be read again until a RESTORE statement has been executed.

See also **READ, RESTORE**

Example

```
10 CLS
20 FOR J = 1 TO 5
30 READ A$,B
40 PRINT A$,B
50 NEXT
60 END
70 DATA "ANGELA:ANGIE",12,"  BRIAN",-20,CHARLIE,39,DIANA,
-16,ERIC,34
```

```
ANGELA:ANGIE   12
  BRIAN       -20
CHARLIE        39
DIANA         -16
ERIC           34
Ok
```

DATE$

Format **DATE$**

Purpose Reads the date of the PX-4's built-in clock.

Remarks The DATE$ function returns the date of the built-in clock as a character string in the following format.

"MM/DD/YY"

Here, MM indicates the month as a value from "01" to "12", DD indicates the day of the month as a value from "01" to "31", and YY indicates the last two digits of the year as a value from "00" to "99".

DATE$ is a system variable, and can be set by executing DATE$="MM/DD/YY".

DAY

Format

As a statement
DAY = <W>

As a variable
X% = DAY

Purpose

DAY is a system variable which maintains the day of the week of the PX-4's built-in calendar clock.

Remarks

As a variable, DAY returns the day of the week of the PX-4's calendar clock as a number from 0 to 6. Sunday is represented by 0, 6 is used to represent Saturday, and so forth.

The day can be set independently of the value assigned as the calendar date (by the DATE$ statement); therefore, as a statement DAY can be used to assign any number from 0 to 6 to the current day of the week. However, if the current day of the week is altered from the above representation, the System Display and other software will print out the day incorrectly. For example if you choose to assign the first day of January 1985 as 6, the System Display will show Saturday when it should in fact be a Sunday.

Example

```
10 PRINT "Day is ";DAY
20 DAY=3
30 PRINT "Day is now";DAY
40 END

run
Day is  5
Day is now 3
Ok
```

DEF FN

Format **DEF FN<name>(<parameter list>)=<function definition>**

Purpose Used to define and name user-written functions.

Remarks A user defined function is a numeric or string expression which can be executed by BASIC programs in the same manner as intrinsic functions (e.g., TAN or SIN). When such a function is called, the variables specified as its arguments (either in the function definition or in the parameter list of the calling statement) are substituted into the expression and the equivalent value is returned as the result of the function.

<parameter list> comprises those variables in the function definition that are to be replaced when the function is called. The items in the list are separated by commas.

If a <parameter list> is included in the <function definition>, then a list with a corresponding number of parameters must be specified in the statement calling the function; the values of variables specified in the calling statement's parameter list are then substituted into the <parameter list> of the function definition on a one-to-one basis.

<function definition> is an expression that performs the operation of the function. It is limited to one program line. 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.

If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a TM error (Type mismatch) occurs.

The DEF FN statement must be executed before the corresponding user function can be called: otherwise, a UF error (Undefined user function) will occur.

DEF FN statements cannot be executed in the direct mode.

Examples showing extensive use of the DEF FN command are shown in the example programs in the appendixes. The following programs outline simpler applications.

Example

```
10 DEF FN SQ(X) = X * X    :' define the function SQ to give
20 '                          the square of a number
30 PRINT FN SQ(9)          :' print the square of 9
40 X = 12                  :' set the variable X equal to 12
50 PRINT FN SQ(X)          :' and print the square of X
60 Y = 10
70 PRINT FN SQ(Y)          :' use the value of the variable Y
80 '                          as a substitute for X
90 '
100 PRINT:PRINT
110 DEF FN NM(X,Y) = X * X + Y:' define a function NM to give
120 '                            a function of two numbers
130 PRINT FN NM(10,20)     :' print the value using 10 and 20
140 X = 5 : Y = 6
150 PRINT FN NM(X,Y)       :' Print the value of the function
160 '                         using the variables X and Y
170 PRINT FN NM(Y,X)       :' The values of the variables are
180 '                         used according to position and
190 '                         NOT VARIABLE NAME


run
 81
 144
 100


 120
 31
 41
Ok
```

The only inverse trigonometric function available is ARC TAN. DEF FN is useful to provide functions for ARC COS etc.,and the formulae for obtaining such functions are listed in Appendix L.

The following program illustrates the use with ARC SIN, and also in converting from radians to degrees. (See ATN for this computation.)

Example

```
10 DEF FN ARCSIN(X) = ATN (X / SQR (1-X * X) )  :' Defines a
20 '                  function to give the angle from its SINE
30 '
40 DEF FN DEG (X) = X * 45 / ATN(1)    :' function to convert
50 '                                        radians to degrees
60 CLS
70 INPUT "Type in SINE of angle "; X
80 R = FN ARCSIN (X) : D = FN DEG (R) :' Find the values of
the angle
90 PRINT "The angle whose SINE is "; X ; " is " ; R ;
"Radians or "; D ; "Degrees"


Type in SINE of angle ? .5
The angle whose SINE is .5  is  .523599 Radians or 30 Degrees
Ok
```

DEFINT/SNG/DBL/STR

Format **DEFINT <range(s) of letters>**
DEFSNG <range(s) of letters>
DEFDBL <range(s) of letters>
DEFSTR <range(s) of letters>

Purpose Declares the type of variables specified in <range(s) of letters>.

DEFINT as an INTEGER variable
DEFSNG as a SINGLE PRECISION variable
DEFDBL as a DOUBLE PRECISION variable, and
DEF STR as a STRING variable

Remarks This statement defines the type of the specified variable or ranges of variables, making it unnecessary to indicate their type by appending the type definition characters (%, !, #, and $). Type declarations made using this statement apply to all variable names which begin with the letters included in <range(s) of letters>. For example, execution of DEFSTR A-C declares all variables whose names begin with the letters A, B, and C as string variables even though the declaration character $ is not appended to their names. Variable types specified in DEF statements do not become effective until those DEF statements have been executed; therefore, the BASIC interpreter assumes that all variables without type declaration characters are single precision variables until a type definition statement is encountered. When a DEF statement is encountered, variables without type definition characters are cleared if the first letter of their names is specified in that statement.

NOTE:
Note that trying to assign a numerical value to R when it has been declared as a string variable results in a TM error (Type mismatch error).

DEF USR

Format **DEF USR[<digit>]=<integer expression>**

Purpose Specifies the starting address in memory of a user-written machine language program.

Remarks Machine language programs whose starting addresses are defined with the DEF USR statement can be used as functions in BASIC programs. This is done using the USR function; see the explanation of the USR function and Appendix D for more information. <digit> is a number from 0 to 9 by which the machine language program is identified when called with the USR function. If <digit> is not specified, 0 is assumed.

<integer expression> is the starting address of the machine language program. Up to 10 starting addresses (USR0 to USR9) may be concurrently defined; if more addresses are required, additional DEF USR statements may be executed to redefine starting addresses for any of USR0 to USR9.

Machine language programs used as subroutines by BASIC programs must be written into memory before they can be called; further, the starting address of the area into which machine language programs are written must be specified with the CLEAR statement.

See also **USR, CALL**

NOTE:
Appendix G describes how to use DEF USR.

DELETE

Format **DELETE [<line number 1>][-<line number 2>]**

Purpose Deletes specified lines of the program in the currently logged in BASIC program area.

Remarks If both <line number 1> and <line number 2> are present, all the lines from <line number 1> to <line number 2> inclusive will be deleted. If the second parameter is omitted, only the line specified in <line number 1> is deleted. If the first parameter is omitted, all lines from the beginning of the program to <line number 2> will be omitted.

An FC error (Illegal function call) will result if a specified line number does not exist or if a hyphen is specified without specifying the second line number.

Although DELETE can be used in a BASIC program, control always returns to the command level after execution.

If you wish to delete the last lines of a program, you cannot specify a line number greater than that of the last line of the program, otherwise an FC error (Illegal function call) will be printed and no action will be taken. Thus, if the last line number in a program is 190 and you wish to delete lines greater than 100, DELETE 1Ø1-19Ø is correct but DELETE 1Ø1-2ØØ will generate an error. However, line 101 does not have to exist.

Examples **DELETE 1Ø** will remove line 10.
DELETE 1Ø–9Ø will remove lines 10 to 90.
DELETE –9Ø will remove lines up to line 90.

DIM

Format **DIM <list of subscripted variables>**

Purpose Specifies the maximum range of array subscripts and allocates space for storage of array variables.

Remarks The DIM statement defines the extent of each dimension of variable arrays by specifying the maximum value which can be used as a subscript for each dimension; it also clears all variables in the specified array(s). For example, DIM A(25,50) defines a two-dimensional array whose individual variables are designated as A(N1,N2), where the maximum value of N1 is 25 and the maximum value of N2 is 50. Since the minimum value of a subscript is 0 (unless otherwise specified with the OPTION BASE statement), this array includes 26 × 51 = 1346 individual variables. Any attempt to access an array element with subscripts greater than those specified in the DIM statement for that array will result in a BS error (Subscript out of range); if no DIM statement is specified, the maximum value which can be used for subscripts is 10. Once an array has been dimensioned with the DIM statement it cannot be redimensioned until it has been erased by a CLEAR or ERASE statement.

See also **ERASE, OPTION BASE, CLEAR**

Example 1 **10 DIM A(20, 15)**
Defines two-dimensional array A and specifies 20 and 15 as its maximum subscript values. Unless otherwise specified by a DEF <type> statement, BASIC will handle this as a single precision numeric array.

Example 2 **10 DIM A$(30)**
Defines one-dimensional string array A$ and specifies 30 as its maximum subscript value.

Example 3 **10 DIM G%(25), F%(25)**
Defines one-dimensional arrays G and F and specifies 25 as the maximum values of their subscripts.

DSKF

Format **DSKF (<disk device name>)**

Purpose Returns the amount of free space on a disk.

Remarks The DSKF function returns the amount of free space on the disk specified in (<disk device name>) in kilobyte (1024-byte) units. The device specified in <disk device name> must be that of a disk device which is supported by PX-4 BASIC.

FC error (Illegal function call) — The <disk device name> argument was incorrectly specified.

DU error (Device unavailable) — The specified device was not available.

Example PRINT DSKF("A:") will return the amount of space available for drive A:.

EDIT

Format **EDIT [<line no.>]**

Purpose Places BASIC in the edit mode.

Remarks This command makes it possible to edit program lines on-screen by displaying the program line specified in <line no.> and positioning the cursor at the beginning of that line. Following occurrence of an error, "EDIT." can be executed to display the line in which the error occurred. The error line is displayed only if the command is executed immediately following occurrence of the error; in this case, the cursor is positioned at the point at which the error occurred.

UL error (Undefined line number) — The program line specified is not included in the program.

END

Format END

Purpose Stops program execution, closes all files and returns BASIC to the command level.

Remarks END statements may be included anywhere in a program to stop execution. However, it is not necessary to place an END statement in the last line of the program if the last program line is always the last line executed.

The END statement is often used together with the IF ... THEN ... ELSE statement to terminate program execution under specific conditions.

As with the STOP statement, program execution terminated by the END statement can be resumed by executing a CONT command. However, the END statement does not result in display of a BREAK message.

It often happens that subroutines are placed at the end of a program. An END command is often placed before the first such subroutine so that the program does not continue into the subroutine when the main part of the program has been completed. The following example illustrates this.

See also STOP

Example

```
10 GOSUB 50
20 PRINT "Having executed the subroutine at line 50"
30 PRINT "this program halts at the END statement on line 40"
40 END
50 PRINT "The program is now executing the subroutine at line
50"
60 PRINT
70 RETURN
run
The program is now executing the subroutine at line 50

Having executed the subroutine at line 50
this program halts at the END statement on line 40
Ok
```

EOF

Format **EOF (<file number>)**

Purpose Returns a value indicating whether the end of a sequential file has been reached during sequential input.

Remarks During input from a sequential file, an "Input past end" error will occur if INPUT# statements are executed against that file after the end of the file has been reached. This can be prevented by testing whether the end of file has been reached with the EOF function.

<file number> is the number under which the file was opened. The function will return "false" (0) if the end of file has not been reached, and "true" (−1) if the end of file has been reached.

Example

```
10 OPEN "O",#1,"TEST"
20 FOR J = 1 TO 5
30 PRINT #1,J
40 NEXT
50 CLOSE #1
60 OPEN "I",#1,"TEST"
70 IF EOF(1) THEN 110
80 INPUT #1,J
90 PRINT J,
100 GOTO 70
110 PRINT "The end of the file has been reached"
120 CLOSE #1

run
 1             2             3             4             5
The end of the file has been reached
Ok
```

ERASE

Format **ERASE <list of variables>**

Purpose Cancels array definitions made with the DIM statement.

Remarks The ERASE statement erases the specified variable arrays from memory, allowing them to be redimensioned and freeing the memory they occupied for other purposes.

An ''Illegal function call error'' will result if an attempt is made to erase a non-existent array.

It is not possible to redimension an array without destroying it completely using ERASE.

See also **DIM**

ERL

Format **ERL**

Purpose Used in an error processing routine to return the line number of the program line at which an error occurred during command or statement execution.

Remarks The ERL function returns the line number of the command/statement causing an error during program execution.

If an error occurs during execution of a command or statement in the direct mode, this function returns the number 65535 as the line number.

The ERL function is normally used with IF ... THEN statements in an error processing routine to control the flow of program execution when an error occurs.

See also **ERROR, ON ERROR GOTO, RESUME, ERR**

Example See under ERROR.

ERROR

Format **ERROR <integer expression>**

Purpose Simulates the occurrence of a BASIC error. Also allows error codes to be defined by the user.

Remarks The value of <integer expression> must be greater than 0 and less than 255. If the value specified equals one of the error codes which is used by BASIC (see Appendix A), occurrence of that error is simulated and the corresponding error message is displayed. If the value specified is not defined in BASIC, the message "UL error" (Unprintable error) is displayed.

You can also use the ERROR statement to define your own error messages; this is illustrated in the example below. When using the ERROR statement for this purpose the value of <integer expression> must be a number which does not correspond to any error code which is defined in BASIC. Such user-defined error codes can then be handled in an error processing routine.

See also **ERR, ERL, ON ERROR GOTO, RESUME**

Example

```
10 ON ERROR GOTO 80
20 X = "FRED"
30 X$(20) = 23
40 ERROR 199
50 PRINT
60 PRINT "There are no more errors to demonstrate"
70 END
80 IF ERR = 13 THEN PRINT "There is a Type Mismatch in line
20":RESUME 30
90 IF ERL = 30 THEN PRINT "There is a Subscript error in lin
e 30":RESUME 40
100 IF ERR = 199 THEN PRINT "I defined this error number 199
 myself":RESUME 50

There is a Type Mismatch in line 20
There is a Subscript error in line 30
I defined this error number 199 myself

There are no more errors to demonstrate
Ok
```

ERR

Format

ERR

Purpose

Used in an error processing routine to return the code of an error occurring during program execution.

Remarks

The ERR function returns the code of errors occurring during command or statement execution.

As with the ERL function, the ERR function is normally used with IF ... THEN statements in an error processing routine to control the flow of program execution when an error occurs. An example of program control using the ERR function is shown below.

See also

ERL, ON ERROR GOTO, RESUME

Example

See under ERROR

```
IF ERR=11 THEN RESUME 1ØØØ
```

When included in an error processing routine, this line causes program execution to resume at line 1000 if the error being processed is a /Ø error (Division by zero).

EXP

Format **EXP(X)**

Purpose Returns the value of the natural base *e* to the power of X.

Remarks The value specified for X must not be greater than 87.3365; otherwise an OV error (Overflow) will occur.

To raise another number to a power use the operator "^". See the program below for an example.
EXP can also be used to obtain antilogarithms.

See also **LOG**

Example

```
10 FOR J = 0 TO 80 STEP 10
20 LPRINT "e^";J;"=";EXP(J)
30 NEXT
40 LPRINT
50 LPRINT "The cube of 2 is: ";2^3
```

```
e^ 0 = 1
e^ 10 = 22026.5
e^ 20 = 4.85165E+08
e^ 30 = 1.06865E+13
e^ 40 = 2.35385E+17
e^ 50 = 5.1847E+21
e^ 60 = 1.142E+26
e^ 70 = 2.51544E+30
e^ 80 = 5.54063E+34

The cube of 2 is:  8
```

FIELD

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

Purpose Assigns string variables to specific positions in a random file buffer.

Remarks When a random access file is opened, a buffer is automatically reserved in memory which is used for temporary storage of data while it is being transferred between the storage medium (RAM disk or other disk device) and the computer's memory. Data is read into this buffer from the storage medium during input, and is written from the buffer to the storage medium during output. During input, data is read into the buffer upon execution of a GET statement, and is output to the buffer upon execution of a PUT statement.

However, before any GET or PUT statement can be executed, a FIELD statement must be executed to assign positions in the file access buffer to specific variables. Doing this causes data substituted into that variable by a PUT statement to be stored in assigned positions in the file access buffer, rather than in normal string space. Conversely, items brought into the buffer from the file by a GET statement are accessed by checking the contents of variables to which positions in the buffer have been assigned. See Chapter 5 for a detailed description of procedures for accessing random access files.

The <file number> which is specified in the FIELD statement is that under which the file was opened. <field width> specifies the number of positions which is to be allocated to the specified <string variable>. For example:

```
FIELD 1,2Ø AS N$,1Ø AS ID$,4Ø AS ADD$
```

assigns the first 20 positions of the buffer to string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$.

The total number of positions assigned to variables by the FIELD statement cannot exceed the record length that was specified when the file was opened; otherwise, an FO error (Field overflow) will occur (the default record length is 128 bytes).

If necessary, any number of FIELD statements may be executed for the same file. If more than one such statement is executed, all assignments made are effective concurrently.

See also **GET, LSET/RSET, OPEN, PUT**
For full details of use of the FIELD command see Chapter 5.

Example

```
10 OPEN "R",#1,"E:EMPDAT.DAT"
20 '   Opens file "EMPDAT.DAT" in drive E: as a random
30 '   access file.
40 FIELD #1,6 AS NO$,2 AS DP$,120 AS RE$
50 '   Assigns the first six bytes of file buffer #1 to
60 '   variable NO$, the second two bytes to variable
70 '   DP$, and the last 120 bytes to variable RE$.
80 '   Data can now be placed in the buffer with the
90 '   LSET/RSET statement, then stored in the file with
100 '  the PUT statement. Or, data can be brought into
110 '  the buffer from the file with the GET statement,
120 '  after which the contents of the buffer variables
130 '  can be displayed with the PRINT statement or
140 '  used for other processing.
```

NOTE:
Once a variable name has been specified in a FIELD statement, only use RSET or LSET to store data in that variable. FIELDing a variable name assigns it to specific positions in the random file buffer; using an INPUT or LET statement to store values to FIELDed variables will cancel this assignment and reassign the names to normal string space.

FILES

Format **FILES[<ambiguous file name>]**

Purpose Displays the names of files satisfying the <ambiguous file name>.

Remarks If <ambiguous file name> is omitted, this command displays the names of all files in the disk device which is the currently selected drive.

When specified, <ambiguous file name> is composed of the following elements.

[<drive name>:][<file name>[<.extension>]]

In this case, the FILES command lists the names of all files which satisfy the <ambiguous file name>. An ambiguous file name is used to find files whose file names and/or extensions include common character strings. The form for specifying ambiguous file names is similar to that used for normal file descriptors, except that the question mark (?) can be used as a wild card character to indicate any character in a particular position, and the asterisk (*) can be used as a wild card character to indicate any combination of characters for a file name or extension. Some examples of ambiguous file names are shown below.

If there are no files on a particular disk, the "File not found" message will be displayed.

Example 1 **FILES "A:L???.BAS"**
Displays all files on disk drive A: whose file names begin with L, are up to four characters long and are also BASIC files.

Example 2 **FILES "L???????.BAS"**
Displays the names of all files on the currently active disk device whose file names begin with the letter L and whose extensions are ".BAS", because all possible character positions are used with the '?' wild card.

Example 3 **FILES "D:*.*" or FILES "D:"**
Displays the names of all files on the disk in drive D.

Example 4 **FILES "D:D???.*"**
Displays the names of all files on the disk in drive D which begin with the letter D and which include not more than four characters in the file name.

Example 5 **FILES "D:*.COM"**
Displays all the files on disk drive D: which have .COM as their extension.

PRINTING FILE NAMES FROM BASIC

In order to print the directory of a disk from BASIC, a screen dump must be carried out. If there are more files than the screen will show at one time, scroll the screen using the cursor keys and perform multiple screen dumps. Selecting screen size of 40 × 50 by WIDTH command is the best to use for screen dumps of files because it can be scrolled, and will hold the greatest number of file names at any one time.

FIX

Format **FIX(X)**

Purpose Returns the integer portion of numeric expression X.

Remarks The value returned by FIX(X) is equal to the sign of X times the integer portion of the absolute value of X. Thus −1 is returned for −1.5, −2 is returned for −2.33333 and so forth. Compare with the INT function, which returns the largest integer which is less than or equal to X.

See CINT for an explanation and comparison of CINT, FIX and INT, and also other problems associated with their use.

FONT

Format **FONT [STEP](X,Y),<function>,<user defined character code>[,<user defined character code>......]**

Purpose Displays user defined characters at the specified location.

Remarks This statement displays user defined characters assigned to the specified user defined character codes. The coordinates (X,Y) indicate a point which is the upper left corner of the first character displayed. Y is an integer from 0 to 13 but X must be a multiple of 8 (0, 8, 16,...) between 0 and 232. If X is other than multiples of 8, the greatest one of multiples of 8 which is less than X is used as the X coordinate.
When the specified location is out of the window, the specified characters are not displayed or part of the characters is displayed.

<function> is one of the following.

PSET:	Displays characters as they are.
PRESET:	Displays reversed characters.
OR:	Displays the result of logical sum (OR) of the dot pattern existing on the screen and that of the specified characters.
AND:	Displays the result of logical product (AND) of the dot pattern existing on the screen and that of the specified characters.
XOR:	Displays the result of XOR (exclusive OR) operation on the dot pattern existing on the screen and that of the specified characters.

FOR...NEXT

Format

FOR <variable> = <expression 1> TO <expression 2> [STEP <expression 3>]

•

•

•

NEXT [<variable>][,<variable>...]

Purpose

The FOR...NEXT statement allows the series of instructions between FOR and NEXT to be repeated a specific number of times.

Remarks

This statement causes program execution to loop through the series of instructions between FOR and NEXT a specific number of times. The number of repetitions is determined by the values specified following FOR for <expression 1>, <expression 2>, and <expression 3>.

<variable> is used as a counter for keeping track of the number of loops which have been made. The initial value of this counter is that specified by <expression 1>. The ending value of the counter is that specified by <expression 2>. Program lines following FOR are executed until the NEXT statement is encountered, then the counter is incremented by the amount specified by <expression 3>. An increment of 1 is assumed if STEP <expression 3> is not specified; however, a negative value must be specified following STEP if the value of <expression 2> is less than that of <expression 1>.

Next, a check is made to see if the value of the counter is greater than the value specified by <expression 2>. If not, execution branches back to the statement following the FOR statement and the sequence is repeated. If the value is greater, execution continues with the statement following NEXT. Otherwise statements in the loop are skipped and execution resumes with the first statement following NEXT.

FOR...NEXT loops may be nested; that is, one FOR...NEXT loop may be included within the body of another one. When loops are nested, different variable names must be specified for <variable> at the beginning of each loop. Further, the NEXT statement for inner loops must appear before those for outer ones.

If nested loops end at the same point, a single NEXT statement may be used for all of them. In this case, the variable names must be specified following NEXT in reverse order to that in which they appear in the FOR statements of the nested loops; in other words, the first variable name following NEXT must be that which is specified in the nearest preceding FOR statement, the second variable name following NEXT must be that which is specified in the next nearest preceding FOR statement, and so forth.

If a NEXT statement is encountered before its corresponding FOR statement, an NF error (NEXT without FOR) message is displayed and execution is aborted. If a FOR statement without a corresponding NEXT statement is encountered, an FN error (FOR without NEXT) message is displayed and execution is aborted.

See also **WHILE...WEND**

Example

```
10 PRINT "1. Single loop incremented in steps of 2"
20 FOR J = 1 TO 9 STEP 2
30 PRINT J,
40 NEXT
50 FOR L = 1 TO 100:NEXT
60 PRINT:PRINT "2. Single loop with default increment of 1"
70 FOR K = 1 TO 5
80 PRINT K,
90 NEXT
100 FOR L = 1 TO 100:NEXT
110 PRINT:PRINT "3. Nested loop with two NEXT statements"
120 FOR J = 1 TO 5
130 FOR K = 1 TO 3
140 PRINT J;"/";K,
150 NEXT:NEXT
160 FOR L = 1 TO 100:NEXT
170 PRINT:PRINT "4. Nested loop with one NEXT statement
    specifying both variable
180 FOR J = 1 TO 5
190 FOR K = 1 TO 3
200 PRINT J;"/";K,
210 NEXT K,J
```

```
1. Single loop incremented in steps of 2
 1          3          5          7          9

2. Single loop with default increment of 1
 1          2          3          4          5

3. Nested loop with two NEXT statements
 1 / 1      1 / 2      1 / 3      2 / 1      2 / 2
 2 / 3      3 / 1      3 / 2      3 / 3      4 / 1
 4 / 2      4 / 3      5 / 1      5 / 2      5 / 3

4. Nested loop with one NEXT statement specifying both
   variables
 1 / 1      1 / 2      1 / 3      2 / 1      2 / 2
 2 / 3      3 / 1      3 / 2      3 / 3      4 / 1
 4 / 2      4 / 3      5 / 1      5 / 2      5 / 3
```

Note that when the program is run, line 10 shows there are 23665 bytes of memory. When line 20 has allocated a value to variable B, available memory is reduced to 23657 bytes. After string manipulation, line 70 shows the memory is reduced a great deal further. In executing the FRE(X$) command, much of this memory can be retrieved.

```
10 PRINT "Free memory for programs and variables using FRE(X)
is";FRE(X)
20 B= 10
30 PRINT "Free memory for programs and variables using FRE(X)
is";FRE(X)
40 FOR J = 65 TO 75
50 A$ = A$ + CHR$(J)
60 NEXT J
70 PRINT "Free memory for programs and variables using FRE(X)
is";FRE(X)
80 PRINT "and using FRE(X$) is ";FRE(X$)

run
Free memory for programs and variables using FRE(X) is 23665
Free memory for programs and variables using FRE(X) is 23657
Free memory for programs and variables using FRE(X) is 23565
and using FRE(X$) is  23631
Ok
```

GET

Format **GET[#]<file number>[,<record number>]**

Purpose The GET statement reads a record into a random file access buffer from a random disk file.

Remarks This statement reads a record into a random file access buffer from the corresponding random access file. <file number> is the number under which the file was opened and <record number> is the number of the record which is to be read into the random file buffer. Both <file number> and <record number> must be specified as integer expressions.

If <record number> is omitted the record read is that following the one read by the preceding GET statement. The highest possible record number is 32767.

Note that records must be read sequentially if the file being accessed is a microcassette file. Also note that the FIELD statement must be executed to assign space in the random file buffer to variables prior to executing a GET statement.

See also **FIELD, LSET/RSET, OPEN, PUT**
For full details of use of the FIELD command see Chapter 4.

Example

```
10 'Lines 20-80 create a random data file with 5 records.
20 OPEN"R",#1,"A:TESTDAT",10
30 FIELD#1,10 AS A$
40 FOR I=1 TO 5
50 PRINT"Type 1-10 characters for record";I;:INPUT B$
60 LSET A$=B$
70 PUT#1,I
80 NEXT
90 '
100 'Lines 110 to read specified records from the file.
110 INPUT"Enter record no.(1-5)";R
120 GET#1,R
130 PRINT A$
140 GOTO 110
```

```
run
Type 1-10 characters for record 1 ? Alfie
Type 1-10 characters for record 2 ? Betty
Type 1-10 characters for record 3 ? Charlie
Type 1-10 characters for record 4 ? Dean
Type 1-10 characters for record 5 ? Edward
Enter record no.(1-5)?



Enter record no.(1-5)? 5
Edward
Enter record no.(1-5)? 4
Dean
Enter record no.(1-5)? 3
Charlie
Enter record no.(1-5)? 2
Betty


Enter record no.(1-5)? 1
Alfie
Enter record no.(1-5)?
```

GOSUB...RETURN

Format

GOSUB <line number>

•

•

RETURN

Purpose

The GOSUB and RETURN statements are used to branch to and return from subroutines.

Remarks

The GOSUB statement transfers execution to the program line number specified in <line number>. When a RETURN statement is encountered, execution then returns to the statement following the one which called the subroutine, either on the same line or the next one. Subroutines may be located anywhere in a program; however, it is recommended that they be made readily distinguishable from the main routine. Since an RG error (RETURN without GOSUB) will occur if a RETURN statement is encountered without a corresponding GOSUB statement, care must be taken to ensure that execution does not move into a subroutine without it having been called. This can be avoided with the STOP, END or GOTO statements; the STOP and END statements halt execution when encountered, while the GOTO statement can be used to route execution around the subroutine.

Subroutines may include more than one RETURN statement if the program logic dictates a return from different points in the subroutine. Further, a subroutine may be called any number of times in a program, and one subroutine may be called by another. Nesting of subroutines in this manner is limited only by the amount of stack space available for storing return addresses. An OM error (Out of memory) will occur if the stack space is exceeded. The stack space size may be changed with the CLEAR statement if it is insufficient to accomodate the number of levels of subroutine nesting used by a program, but this must be executed at the beginning of the program outside the subroutines as CLEAR destroys all references to RETURN line numbers on the stack and you will not be able to RETURN from a subroutine if CLEAR is used within it.

See also **CLEAR**

Example

```
10 GOSUB 70
20 PRINT "Resuming execution after returning from subroutine
at line 70"
30 GOSUB 60
40 PRINT "Resuming execution after return from the nested
subroutines startin at line 50"
50 END
60 GOSUB 70:RETURN
70 PRINT "Now executing the subroutine at line 70"
80 RETURN

run
Now executing the subroutine at line 70
Resuming execution after returning from subroutine at line 70
Now executing the subroutine at line 70
Resuming execution after return from the nested subroutines st
arting at line 50
Ok
```

GOTO or GO TO

Format GOTO <line number>
GO TO <line number>

Purpose Unconditionally transfers program execution to the program line specified by <line number>.

Remarks This statement is used to make unconditional "jumps" from one point in a program to another. If the first statement on the line specified by <line number> is an executable statement (other than a REM or DATA statement), execution resumes with that statement; otherwise, execution resumes with the first executable statement encountered following <line number>.

It is also possible to leave out the GOTO in conditional statements, e.g., line 20 in the following program.

A UL error (Undefined line number) will occur if <line number> refers to a non-existent line.

GOTO can also be used in direct mode. In this case, variables are not destroyed (unlike RUN <line number> which does destroy variables), and can in fact be assigned from the command line in the direct mode.

Example

```
10 READ A,B:'Reads numbers into A and B from line 70
20 IF A=0 AND B=0 THEN 80:'Jumps to line 80 if A and B
25 '                        both equal 0.
30 PRINT "A=";A,"B=";B
40 S=A*B
50 PRINT "Product is";S
60 GOTO 10                :'Jumps to line 10.
70 DATA 12,5,8,3,9,0,0,0
80 END

run
A= 12         B= 5
Product is 60
A= 8          B= 3
Product is 24
A= 9          B= 0
Product is 0
Ok
```

HEX$

Format **HEX$(X)**

Purpose Returns a character string which represents the hexadecimal value of X.

Remarks The value of the numeric expression specified in the argument must be a number in the range from −32768 to 65535. If the value of the expression includes a decimal fraction, it is rounded to the nearest integer before the string representing the hexadecimal value is returned.

To convert from hexadecimal to decimal use &H before the hexadecimal value. This will give a numerical constant.

HEX$ is a string.
&H is a numeric.

See also **OCT$**

Example 1

```
10 CLS
20 LOCATE 1,1: PRINT "Convert Hex to Decimal (H) or ":
PRINT "Decimal to Hex (D)"
30 INPUT C$
40 IF C$ = "H" OR C$ = "h" THEN GOSUB 100 ELSE IF C$ =
"D" OR C$ = "d" THEN GOSUB 200 ELSE 20
50 INPUT "Any more (yes/no) "; YN$
60 IF LEFT$(YN$,1) <> "y" AND LEFT$(YN$,1) <> "Y" THEN
END ELSE RUN
100 INPUT "Type in number in hexadecimal ";H$
110 V  = VAL ("&H" + H$)
120 PRINT V
130 RETURN
200 INPUT "Type in number in decimal "; D
210 V$ = HEX$ (D)
220 PRINT V$
230 RETURN
```

```
run
Convert Hex to Decimal (H) or
Decimal to Hex (D)
? h
Type in number in hexadecimal ? 4d
 77
Any more (yes/no) ?


Convert Hex to Decimal (H) or
Decimal to Hex (D)
? d
Type in number in decimal ? 34
22
Any more (yes/no) ?
```

IF...THEN [...ELSE]/IF...GOTO

Format Possible alternatives are

IF<logical expression>THEN<statement>[ELSE <statement>]

IF<logical expression>THEN<line No.>[ELSE<line No.>]

IF<logical expression>THEN<statement>[ELSE<line No.>]

IF<logical expression>THEN<line No.>[ELSE <statement>]

IF<logical expression>GOTO<line No.>[ELSE <statement>]

IF<logical expression>GOTO<line No.>[ELSE<line No.>]

Purpose Changes the flow of program execution according to the results of a logical expression.

Remarks The THEN or GOTO clause following <logical expression> is executed if the result of <logical expression> is true (–1). Otherwise, the THEN or GOTO clause is ignored and the ELSE clause (if any) is executed; execution then proceeds with the next executable statement.

When a THEN clause is specified, THEN may be followed by either a line number or one or more statements. Specifying a line number following THEN causes program execution to branch to that program line in the same manner as with GOTO. When a GOTO clause is specified, GOTO is always followed by a line number.

IF...THEN...ELSE statements may be nested by including one such statement as a clause in another. Such nesting is limited only by the maximum length of the program line.

For example, the following is a correctly nested IF...THEN statement

```
20 IF X>Y THEN PRINT "X IS LARGER THAN Y" ELSE
IF Y>X THEN PRINT "X IS SMALLER THAN Y" ELSE
PRINT "X EQUALS Y"
```

Because of the logical structure of the line only one of the strings can be printed.

If a statement contains more THEN than ELSE clauses, each ELSE clause is matched with the nearest preceding THEN clause. For example, the following statement displays "A=C" when A=B and B=C. If A=B and B< >C it will display "A< >C". And if A< >B, it displays nothing at all.

```
IF A=B THEN IF B=C THEN PRINT "A=C" ELSE
PRINT "A< >C"
```

It is also possible to have a number of statements where <statement> occurs in the above format expressions. For example it is common to have a line such as:

```
IF A=2 THEN B=3:C=7:A$="ORANGE"
```

Only if A has the value 2 will B be set equal to 3. The value of C will also only be set equal to 7 and A$ given the value "ORANGE" if A=2. If the expression "A=2" is false then all the rest of the line will be ignored.

This also applies if the sequence of statements exists in a line such as the following:

```
IF A=2 THEN PRINT "TRUE":B=5:A$="APPLES":
GOTO 200 ELSE PRINT "FALSE":B=7:A$="PEARS":
GOTO 300
```

In this case if A has the value 2 the expression "A=2" is true and the variable B is made equal to 5, the value "APPLES" is assigned to A$ and the program branches to line 200. However, if A does not have the value 2, the expression "A=2" is false and the commands following ELSE are executed; B is set equal to 7, A$ is assigned the value "PEARS" and the program will branch to line 300.

When using IF together with a relational expression which tests for equality, remember that the results of arithmetic operations are not always exact values. For example, the result of the relational expression SIN(1.5708)=1 is false even though "1" is displayed if PRINT SIN(1.5708) is executed. Therefore, the relational expression should be written in such a way that computed values are tested over the range within which the accuracy of such values may vary. For example, if you are testing for equality between SIN(1.5708) and 1, the following form is recommended:

IF ABS(1 – SIN(1.5708)) < 1.0E – 6 THEN...

This relational expression returns "true".

Example

```
10 SCREEN 0,0,0:CLS
20 RANDOMIZE VAL(RIGHT$(TIME$,2))
30 '     Reinitializes the sequence of numbers returned
40 '     by the RND function
50 '
60 PRINT "Guess what number I am thinking of."
70 '
80 N=INT(RND(1)*9999)
90 '      Generates a random 4-digit number between
100 '     0 and 9999 and stores it in variable N
110 '
120 INPUT"Enter your guess";G
130 I=I+1
140 '     Keeps track of the number of guesses
150 '
160 IF G=N THEN PRINT "That's just right--in";I;"guesses!":E
LSE IF G<N THEN PRINT "You're too low! Try again.":GOTO 120:
ELSE PRINT "Sorry--you're too high! Try again.":GOTO 120
170 '     Displays the first message and the number
180 '     of guesses you have made if you correctly
190 '     guess the number generated on line 40; if
200 '     your guess is too low, displays the second
210 '     message and branches to line 50; if your
220 '     guess is too high, displays the third
230 '     message and branches to line 50
```

INKEY$

Format **INKEY$**

Purpose Checks the keyboard buffer during program execution and returns a null string if no key has been pressed.

Remarks INKEY$ returns a null string if the keyboard buffer is empty. If any key whose code is included in the ASCII code table has been pressed, INKEY$ reads that character from the keyboard buffer and returns it to the program. Characters read from the keyboard buffer by INKEY$ are not displayed on the screen.

INKEY$ simply examines the keyboard buffer. It does not wait for a key to be pressed. If this function is required, use INPUT$(1).

See also **INPUT$**

Example

```
10 SCREEN 0,0,0
20 WIDTH 80,8:CLS
30 X=1:Y=1:LOCATE X,Y:PRINT "*";:'Displays an asterisk
40 '                               in the upper left
50 '                               corner of the screen.
60 '
70 A$=INKEY$:IF A$="" THEN 70
80 '     Checks for input from the keyboard; repeats
90 '     until input is detected.
100 '
110 ON INSTR(CHR$(30)+CHR$(31)+CHR$(29)+CHR$(28),A$) GOSUB 2
50,300,350,400
120 '     Checks whether the key pressed is one of the
130 '    cursor control keys; if so, goes to the
140 '    corresponding subroutine.  Otherwise,
150 '    continues to the GOTO statement on the line
160 '    below.
170 '
180 GOTO 70:'Transfers execution to line 70.
190 '
200 'The four subroutines below move the asterisk
210 'in the direction indicated by the arrow on
220 'the applicable cursor key.
230 '
240 'Move asterisk up
```

```
250 Y=Y-1:IF Y<1 THEN Y=8
260 LOCATE X,Y:PRINT "*";:IF Y=8 THEN LOCATE X,1:PRINT " ":E
LSE LOCATE X,Y+1:PRINT " "
270 RETURN
280 '
290 'Move asterisk down
300 Y=Y+1:IF Y>8 THEN Y=1
310 LOCATE X,Y:PRINT "*";:IF Y=1 THEN LOCATE X,8:PRINT " ";:
ELSE LOCATE X,Y-1:PRINT " "
320 RETURN
330 '
340 'Move asterisk left
350 X=X-1:IF X<1 THEN X=80
360 LOCATE X,Y:PRINT "*";:IF X=80 THEN LOCATE 1,Y:PRINT " ";
:ELSE LOCATE X+1,Y:PRINT " ";
370 RETURN
380 '
390 'Move asterisk right
400 X=X+1:IF X>80 THEN X=1
410 LOCATE X,Y:PRINT "*";:IF X=1 THEN LOCATE 80,Y:PRINT " ";
:ELSE LOCATE X-1,Y:PRINT " ";
420 RETURN
```

INP

Format INP (J)

Purpose Returns one byte of data from machine port J.

Remarks The machine port number specified for J must be an integer expression in the range from 0 to 255.

The full use of this command is beyond the scope of this manual. Please see the System Documentation for details of the ports.

See also OUT

Example

```
10 'See I/O port &H02 if LED is ON
20 A=INP(&H2)
30 LED=A AND &H4       :'See BIT 2 status
40 IF LED=0 GOTO 60    :'If LED is OFF, set LED
50 END
60 OUT &H2,&H4         :'Set LED
```

INPUT

Format

INPUT[;"<prompt string>"]|;|<list of variables>
|,|

Purpose

Makes it possible to substitute values into variables from the keyboard during program execution.

Remarks

Program execution pauses when an INPUT statement is encountered to allow data to be substituted into variables from the keyboard. One data item must be typed in for each variable name specified in <list of variables>, and each item typed in must be separated from the following one by a comma. If any commas are to be included in a string substituted into a given variable, that string must be enclosed in quotation marks when it is typed in from the keyboard. The same applies to leading and trailing spaces; leading and trailing spaces are not substituted into a string variable by the INPUT statement unless the string is enclosed in quotation marks.

If the number or type of items entered is incorrect, the message "?Redo from start" is displayed, followed by the prompt string (if any). When this occurs, all the data items must be re-entered. No values are substituted into variables until a correct response has been made to all the items of the list.

When BASIC executes the line a question mark prompt is displayed if no prompt string is specified. However, if a prompt string is specified, the string is displayed, but whether a question mark is displayed depends on the character before the list of variables. It is possible to enter a comma or a semi-colon before the list of variables. In the latter case, a question mark will be displayed. If the prompt string is followed by a comma, the question mark is suppressed. If no prompt string is given it is not possible to suppress the question mark.

The optional semicolon following INPUT prevents the cursor from advancing to the next line when the user types a RETURN on completion of entry of the data. The next PRINT statement or error statement will be printed directly after the last character input by the user before pressing RETURN .

When more than one variable name is specified in <list of variables>, each variable name must be separated from the following one by a comma. Items entered in response to an INPUT statement are substituted into the variables specified in <list of variables> when the RETURN key is pressed. The user must input each variable and separate it from the following one by a comma. If the user tries to press RETURN after each variable, a "?Redo from start" message will be printed, and the user will have to begin at the first item of the list of variables. The values entered are only substituted into the variables when the RETURN key is pressed at the end of the list.

When the RETURN key is pressed for a single item or for the last item of a list, the variable is set to a null string if it is a string variable and to zero if it is a numeric variable.

Example

```
10 INPUT A                  :'Inputs value from keyboard
20                          :'into A, then moves cursor
30                          :'to next line.
40                          :'
50 INPUT;B                  :'Inputs value into B and
60                          :'keeps cursor on current
70                          :'line.
80                          :'
90 INPUT"Enter C",C         :'Displays prompt without
100                         :'question mark and inputs
110                         :'value into C.
120                         :'
130 INPUT"Enter D,E";D,E    :'Displays prompt and
140                         :'question mark and inputs
150                         :'values into D and E.
160                         :'
170 INPUT;"Enter F,G";F,G:'Displays prompt, inputs
180                         :'values into F and G, and
190                         :'keeps cursor on current line.
200 PRINT "END"
```

INPUT#

Format INPUT# **<file number>, <variable list>**

Purpose This statement is used to read items from a sequential disk file in a similar way to that in which the INPUT statement reads data from the keyboard.

Remarks The sequential file from which data is to be read with this statement must have been previously opened for input by executing an OPEN statement. <file number> is the number under which the file was opened.

As with INPUT, <variable list> specifies the names of variables into which items of data are to be read when the INPUT# statement is executed. Variables specified must be of the same type as data items which are read. Otherwise, a "Type mismatch" error will occur.

Upon execution of this statement, data items are read in from the file in sequence until one item has been assigned to each variable in <variable list>. When the file is read with this statement, the first character encountered which is not a space is assumed to be the start of a data item. With string items, the end of one item is assumed when the following character is a comma or a carriage return, however, individual string items may include commas and carriage returns if they are enclosed in quotation marks when they were saved to the file. The end of a data item is also assumed if 255 characters are read without encountering a comma or carriage return.

With numeric items, the end of each item is assumed when a space, comma, or carriage return is encountered. Therefore, care must be taken to ensure that proper delimiters are used when the file is written to the disk file with the PRINT# statement.

Examples of use of the INPUT# statement are shown in the program below.

See also **INPUT, LINE INPUT, LINE INPUT#, OPEN, PRINT#, WRITE, WRITE#**
Chapter 4

Example

```
10 OPEN "O",#1,"a:test1.dat"
20 FOR I=1 TO 16:'   Saves numeric data items "1"
30 '                 to "16" to file "a:test1.dat"
40 PRINT#1,I
50 NEXT I
60 PRINT#1,"a";CHR$(13);"b"
70 '   Saves "a" and "b" to file "a:test1.dat"
80 '   as separate data items.  Items are separated
90 '   by a carriage return code (CHR$(13)).
100 PRINT#1,CHR$(34)+"c,d,e"+CHR$(34)
110 '   Saves "c,d,e" to file "a:test1.dat" as one
120 '   data item.  This is regarded as one item because
130 '   it is enclosed in quotation marks (CHR$(34))
140 '   to indicate that the commas are part of the
150 '   string, and not delimiting characters.
160 PRINT#1,"f,g"
170 '   Saves "f,g" to file "a:test1.dat" as separate
180 '   data items.  The reason for this is that commas
190 '   are regarded as delimiters unless quotation marks
200 '   are saved to the disk to indicate that the commas
210 '   are part of a string.  Quotation marks are saved
220 '   to a file by specifying their ASCII codes with
230 '   the CHR$ function as shown above with "c,d,e".
240 CLOSE
250 DIM A(15)
260 OPEN "I",#1,"a:test1.dat"
270 FOR I=0 TO 15
280 INPUT#1,A(I)
290 NEXT I
300 FOR I=0 TO 15
310 PRINT A(I);
320 NEXT
330 PRINT
340 INPUT#1,A$,B$,C$,D$,E$
350 PRINT A$:PRINT B$:PRINT C$:PRINT D$:PRINT E$
360 CLOSE

run
 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16
a
b
c,d,e
f
g
Ok
```

INPUT$

Format

INPUT$(X[,[#]<file number>])

Purpose

Reads a string of X characters from the keyboard buffer or the file opened under <file number>.

Remarks

INPUT$(X) reads the number of characters specified by X from the keyboard buffer and returns a string consisting of those characters to the program. If the keyboard buffer does not contain the specified number of characters, INPUT$ reads those characters which are present and waits for other keys to be pressed. Characters read are not displayed on the screen.

Unlike the INPUT and LINE INPUT statements, INPUT$ can be used to pass control characters such as RETURN (character code 13) to the program.

INPUT$(X,[#]<file number>) reads the number of characters specified by X from a sequential file opened under the specified file number. As with the first format, characters which would be recognized as delimiters between items by the INPUT # or LINE INPUT # statements are returned as part of the character string.

Execution of the INPUT$ function can be terminated by pressing the STOP key.

The BASIC statement

```
A$=INPUT$(1)
```

is useful for waiting for a single key to be pressed, in contrast to

```
100 A$=INKEY$ : IF INKEY$="" THEN 100
```

whereas INKEY$ can scan the keyboard buffer simply to test if a key has been pressed without waiting for it to be pressed.

Example

```
10 OPEN"O",#1,"test"
20 PRINT#1,"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30 CLOSE
40 OPEN"I",#1,"test"
50 '
60 '
100 A$=INPUT$(10,#1) :'Inputs 10 characters into
110                  :'A$ from sequential file opened
120                  :'under file number 1.
125 PRINT A$
130                  :'
140 B$=INPUT$(10)    :'Inputs 10 characters into B$
150                  :'from keyboard.
160 PRINT B$

run
ABCDEFGHIJ
qwertyuiop
Ok
```

INSTR

Format **INSTR([J,]X$,Y$)**

Purpose Searches for the first occurrence of string Y$ in string X$ and returns the position at which a match is found.

Remarks If J is specified, the search for string Y$ begins at position J in string X$. J must be specified as an integer expression in the range from 1 to 255; otherwise, an "Illegal function call" error will occur. If a null string is specified for Y$, INSTR returns the value which is equal to that specified for J. A value of "0" is returned if J is greater than the length of X$, if X$ is a null string, or if Y$ cannot be found. Both X$ and Y$ may be specified as string variables, string expressions or string literals.

Example

```
10 'Example of using INSTR with ON...GOTO to control
20 'flow of program execution.
30 '
40 INPUT"Enter a,b,or c";X$
50 ON INSTR(1,"abc",X$) GOTO 70,90,110
60 PRINT"Illegal entry, try again.":GOTO 40
70 PRINT"Character entered is ";CHR$(34);"a.";CHR$(34)
80 END
90 PRINT"Character entered is ";CHR$(34);"b.";CHR$(34)
100 END
110 PRINT"Character entered is ";CHR$(34);"c.";CHR$(34)
120 END

run
Enter a,b,or c? a
Character entered is "a."
Ok
```

INT

Format INT(X)

Purpose Returns the largest integer which is less than or equal to X.

Remarks Any numeric expression may be specified for X.

See also CINT, FIX

The explanation of CINT also contains information on the differences between CINT, FIX and INT and describes why some problems arise from conversion and storage of numbers in connection with these functions.

Example

```
10 PRINT "I","INT(I)"
20 FOR I=-5 TO 5 STEP .8
30 PRINT I,INT(I)
40 NEXT I
I              INT(I)
-5             -5
-4.2           -5
-3.4           -4
-2.6           -3
-1.8           -2
-1             -1
-.2            -1
 .6             0
 1.4            1
 2.2            2
 3              3
 3.8            3
 4.6            4
```

KEY

Format

KEY <programable function key no.>,<character string>
KEY <item function key no.>,<character string>
KEY <item function key switch>
KEY LIST
KEY LLIST

Purpose

Used to define the programmable function keys and item function keys. Also used to output a list of character strings assigned to all function keys to the screen or printer.

Remarks

The first two formats indicated above assign the specified contents of <character string> to the function key specified in <programmable function key no.> or <item function key no.>. For the programmable function keys, the function key number is specified as a number from 1 to 10. For the item function keys, the function key number is specified as a number from &H40 to &H7E.

<character string> is specified as any combination of up to 15 characters. If more than 15 characters are specified in <character string>, the 16th and following characters are ignored when the statement is executed.

The third format indicated above is used to clear, disable, or enable the item function keys. All item functions are cancelled when 255 is specified in <item function key switch>, all item function keys are disabled when 254 is specified, and all item function keys are enabled when 253 is specified. However, the item function keys can be temporarily enabled after executing KEY 254 by pressing them together with the CTRL and SHIFT keys. The KEY LIST statement outputs the definitions of the programmable function keys to the display screen, and the KEY LLIST outputs a similar list to the printer.

Initial definitions of the programmable function keys are as follows.

PF1	auto	PF6	load"
PF2	list	PF7	save"
PF3	edit	PF8	system
PF4	stat	PF9	menu^M
PF5	run^M	PF10	login_

FC error (Illegal function call) — The number specified in one of the statement operands was outside of the prescribed range.

MO error (Missing operand) — A required operand was not specified in the statement.

KILL

Format KILL **<file descriptor>**

Purpose Used to delete files from a disk device.

Remarks The KILL command can be used to delete any type of disk file. The full file descriptor must be specified if the file to be deleted is in a drive other than that which is currently selected. Otherwise, only the file name and extension need to be specified.

Example

```
KILL "A:GRAPH.BAS"
```

This will delete the file in drive A: "GRAPH" which has the extension ".BAS".

NOTE:
Operation of the KILL command is not assured if it is issued against a file which is currently OPEN.

LEFT$

Format **LEFT$(X$,J)**

Purpose Returns a string composed of the J characters making up the left end of string X$.

Remarks The value specified for J must be in the range from 0 to 255. If J is greater than the length of string X$, the entire string will be returned. If J is zero, a null string of zero length will be returned.

See also **MID$, RIGHT$**

Example 1 A$=LEFT$("CARROT",3) will return "CAR" to be stored in A$.

Example 2

```
10 A$ = "EPSON"
20 FOR J = 1 TO 6
30 PRINT LEFT$(A$,J)
40 NEXT
```

```
E
EP
EPS
EPSO
EPSON
EPSON
```

LEN

Format **LEN(X$)**

Purpose Returns the number of characters in string X$.

Remarks The number returned by this function also indicates any blanks or non-printable characters included in the string (such as the return and cursor control codes).

Example

```
10 CLS
20 INPUT "Type in a word or phrase";A$
30 PRINT "The length of :- "
40 PRINT A$
50 PRINT "is"; LEN(A$); "characters"
60 GOTO 20

Type in a word or phrase? FRED
The length of :-
FRED
is 4 characters
Type in a word or phrase?



Type in a word or phrase? CHARLIE IS SUPER
The length of :-
CHARLIE IS SUPER
is 16 characters
Type in a word or phrase?
```

LET

Format **[LET] <variable> = <expression>**

Purpose Assigns the value of <expression> to <variable>.

Remarks Note that the word LET is optional. Thus, in the example below, the variables A$ and B$ give the same result when printed, as do A and B.

Example

```
10 CLS
20 LET A$ = "THIS IS A STRING"
30 B$ = "THIS IS A STRING"
40 PRINT A$
50 PRINT B$
60 LET A = 3*4
70 B = 3 * 4
80 PRINT A,B

THIS IS A STRING
THIS IS A STRING
 12            12
Ok
```

LINE

Format **LINE[[STEP] (X1,Y1)] – [STEP](X2,Y2)[,[<function code>] [,[B[F]][,<line style>]]]**

Purpose Draws a line between two specified points.

Remarks This statement is a graphics command which can only be used for graphic screen (LCD display). It draws a straight line between two specified points on the graphic screen. The coordinates of the first point are specified as (X1, Y1) and those of the second point are specified as (X2, Y2).

If STEP is omitted, (X1, Y1) and (X2, Y2) are absolute screen coordinates; if STEP is specified, (X1, Y1) indicates coordinates in relation to the last dot specified by the last graphic display statement executed (PSET, PRESET or LINE). The coordinates of the last previously specified dot are maintained by a pointer referred to as the last reference pointer (LRP); this pointer is updated automatically whenever a PSET, PRESET, or LINE statement is executed.

For example

LINE (0,0) – (239,63)
draws a line diagonally from the top left hand corner to the bottom right hand corner.

LINE – (100,50)
draws a line from the last plotted point (i.e. the LRP) to the point (100,50).

LINE (10,10) – STEP (100,30)
draws a line from point (10,10) to a point 100 points to the right and 30 down from point (10,10); i.e., to point (110,40).

LINE – STEP (100,50)
draws a line 100 points to the right and 50 points down from the coordinates of the LRP.

LINE STEP (10,10) – (100,50)
draws a line from a point 10 to the right and 10 down from the LRP to the absolute point (100,50).

LINE STEP (10,10) – STEP (100,40)
draws a line from a point 10 to the right and 10 down from the LRP. The LRP is then updated and the line drawn 100 points to the right and 50 down from the first end point of the line. Thus if the last point plotted before this command was executed was (5,3), the line would be drawn from the point (15,13) to (115,53).

<function code> is a number from 0 to 7 which specifies the line function. If 0 is specified, the LINE statement resets (turns off) dots along the line between the specified coordinates. If a number from 1 to 7 is specified, dots along the line are set (turned on) when the statement is executed. If no <function code> is specified, 7 is assumed.

Specifying "B" causes the LINE statement to draw a rectangle whose diagonal dimension is defined by the two points specified. If the F option is specified together with the B option, the rectangle is filled in. However, the BF option cannot be specified together with <line style>, although simply using B will allow rectangles to be drawn using different line types.

If you want to use the B or BF function without using the <function code>, a comma must be used as separator.

For example

LINE (0,0) — (20,15) ,,BF
will fill a box 20 points wide and 15 points high in the top left hand corner of the screen.

The <line style> option is a parameter which determines the type of line drawn between the two specified points. The line style is specified as any number which can be represented with 16 binary digits; i.e., the line style can be specified as any number from 0 to 65535 (in hexadecimal notation, from &H0 to &HFFFF). There is a one-to-one correspondence between the settings of the binary digits of <line style> and the settings of each 16-dot segment of the line drawn when the statement is executed. When the

<function code> is 1 to 7 or defaults to 7 because no value is inserted, all points corresponding to "1" bits are set (i.e., plotted). When the <function code> is specified as 0, all points corresponding to "1" bits are reset (i.e., erased). This is illustrated in the second example program below. In both cases where dots correspond to "0" bits no action is taken. When the length of the line is greater than 16 dots, the pattern is repeated for each 16-dot segment.

For example, dot settings are as follows when <line style> is specified as 1, 43690, and 61680.

```
<line style>       Binary equivalent
1 (&H1)            0000000000000001
                   ---------------* Dot settings
                                  (* for on, − for off)
43690 (&HAAAA)     1010101010101010
                   *-*-*-*-*-*-*-*-  Dot settings

61680 (&HF0F0)     1111000011110000
                   ****----****----  Dot settings
```

The LINE statement can only be executed during display in the graphic mode (screen mode 3).

See also **PRESET, PSET**

LINE INPUT

Format

LINE INPUT[;][<prompt string>;]<string variable>

Purpose

Used to substitute string data including all punctuation into string variables from the keyboard during program execution.

Remarks

The LINE INPUT statement is similar to the INPUT statement in that it is used to substitute values into variables from the keyboard during program execution.
However, whereas the INPUT statement can be used to input both numeric and string values, the LINE INPUT statement can only be used for input of string values. Further, only one such string can be input each time the LINE INPUT statement is executed. It is not possible to specify a list of variables separated by commas as is the case with the INPUT statement, because commas are accepted as part of the string.

INPUT allows commas to be entered if the first character typed is the quotation marks character. Quotation marks can be entered as long as they are not input as the first character. LINE INPUT on the other hand allows all characters to be substituted into the specified variable exactly as entered. Further, no question mark is displayed when a LINE INPUT statement is executed unless one has been included in <prompt string> by the user.

As with the INPUT statement, a semicolon immediately following LINE INPUT suppresses the carriage return typed by the user. The cursor is positioned after the last character entered by the user before pressing RETURN.

See also

INPUT

Example

```
5 CLS
10 LINE INPUT "TYPE IN SOME CHARACTERS ";A$
15 PRINT " This is on the next line"
20 PRINT "The characters were ";A$
30 LINE INPUT;"AND SOME MORE ";B$
40 PRINT " This is on the same line"
50 PRINT "The characters were ";B$
```

```
TYPE IN SOME CHARACTERS CHARLIE IS MY DARLING
 This is on the next line
The characters were CHARLIE IS MY DARLING
AND SOME MORE OVER THE SEA TO SKYE This is on the same line
The characters were OVER THE SEA TO SKYE
Ok
```

LINE INPUT#

Format **LINE INPUT# <file number>,<string variable>**

Purpose Used to read data into string variables from a sequential access file, in the same way that LINE INPUT is used to read strings from the keyboard.

Remarks The LINE INPUT# statement is similar to the INPUT# statement in that it is used to read data into variables from a sequential access file. The value of <file number> is the number under which the file was opened, and <string variable> is the name of the variable into which data is read when the statement is executed.

Whereas the INPUT# statement can be used to read both numeric and string values, the LINE INPUT# statement can only be used to read character strings. Further, only one such string can be read each time the LINE INPUT# statement is executed. It is not possible to specify a list of variables, as is possible with the INPUT# statement.

Another difference between the INPUT# and LINE INPUT# statements is that whereas the former recognizes both commas and carriage returns as delimiters between data items, the LINE INPUT# statement regards all characters up to a carriage return (up to a maximum of 255 characters) as one data item. Any commas encountered are regarded as part of the string being read. The carriage return code itself is skipped, so the next LINE INPUT# statement begins reading data at the character following the carriage return.

This statement can be used to read all values written by a PRINT# statement into one variable. It also allows lines of a BASIC program which has been saved in ASCII format to be input as data by another program.

See also **INPUT#**

Example See Chapter 5.

LIST

Format

1) LIST[*] [[<line no. 1>][-[<line no. 2>]]
2) LIST[*] <file descriptor>
[,[<line no. 1>][-[<line no. 2>]]]
3) LLIST[*] [[<line no. 1>][-[<line no. 2>]]

Purpose

Outputs all or part of a program in memory to an external device.

Remarks

The LIST command outputs all or part of the program in the currently selected program area to the screen or other external device.With format 1) above, output is to the screen; with format 2), output is to the specified device; when it is omitted, output is to the display screen.

If <line no. 1> is specified by itself, only that line of the program is output.

If <line no. 1> and the hyphen are specified, the line specified and all following lines are output.

If the hyphen and <line no. 2> are specified, all lines from the beginning of the program to the specified line are output.

If both <line no. 1> and <line no. 2> are specified, all lines within that range are output.

When the asterisk is specified, program lines are output without line numbers.Further, any remark statements which are specified using an apostrophe (‘) are printed without the apostrophe, making it possible to use the LIST statement in simple word processing applications.

In the case of the LLIST command, output is always directed to the printer; however, in other respects it is exactly the same as the LIST command.

LLIST

Format **LLIST[*][<line number>][– <line number>]**

Purpose Lists all or part of the lines of the program in the currently logged in program area to a printer.

Remarks The LLIST command is used in the same manner as LIST, but output is always directed to the printer connected to the PX-4. BASIC always returns to the command level after execution of a LLIST command.

See also **LIST**

Example 1 **LLIST**
Prints all lines of the program in the currently logged in program area.

Example 2 **LLIST ***
Same as above, but prints program lines without line numbers.

Example 3 **LLIST 500**
Prints program line 500.

Example 4 **LLIST 150-**
Prints all program lines from line 150 to the end of the program.

Example 5 **LLIST -1000**
Prints all lines from the beginning of the program to line 1000 (inclusive).

Example 6 **LLIST 150-1000**
Prints program lines from 150 to 1000 (inclusive).

LOAD

Format **LOAD<file descriptor>[,R]**

Purpose Loads a program into memory from a disk drive, RAM disk, the RS-232C interface, RAM cartridge, ROM cartridge, or the microcassette drive.

Remarks Specify the device name, file name, and extension under which the program was saved in <file descriptor>. If the device name is omitted, the currently selected drive is assumed; if the file name extension is omitted, ".BAS" is assumed.

When a LOAD command is executed without specifying the "R" option, all files which are open are closed, all variables are cleared, and all lines of any program in the currently logged in program area are cleared; after loading is completed, BASIC returns to the command level.

However, if the "R" is specified, any files which are currently open remain open and program execution begins as soon as loading has been completed. Thus, LOAD with the "R" option may be used to chain execution of programs which use the same data files. The following restrictions must be noted when using LOAD with the "R" option to chain execution of programs.

- All variables are cleared by execution of the LOAD command, regardless of whether the "R" option is specified. Further, the COMMON statement cannot be used to pass variables to the program called. Therefore, some other provision must be made for passing data to the program called (for example, intermediate data could be saved in a file in RAM disk).
- All assignments of variables to positions in random file buffers are cancelled even though the random access files to which the buffers belong remain open. Therefore, the FIELD statement must be executed in the called program to remake these assignments.

See also **CHAIN, MERGE, RUN, SAVE**

Example 1 **LOAD"A:PROG1.BAS"**

Example 2 (Example of program call using LOAD)

```
10 CLS
20 PRINT "This is the calling program, sometimes called the
loader"
30 PRINT "It will now load the program LOAD2.BAS...."
40 LOAD "A:LOAD2.BAS",R

This is the calling program, sometimes called the loader
It will now load the program LOAD2.BAS....
```

Example 3 (Example of program called by Example 2)

```
10 PRINT
20 PRINT"This is the program called LOAD2.BAS which has been
 loaded by the loading program LOAD1.BAS"
30 END

This is the program called LOAD2.BAS which has been loaded
by the loading program LOAD1.BAS
Ok
```

LOAD?

Format **LOAD? [<file descriptor>]**

Purpose Verifies the contents of a file.

Remarks This statement is used to check whether the file specified in <file descriptor> has been properly recorded. No device name other than CAS0: may be specified in <file descriptor>.

The LOAD? statement reads the contents of the specified file in the mode (stop or non-stop) in which it was written to the external audio cassette and verifies its contents by making CRC checks. If any CRC error is detected, an IO error (Device I/O error) occurs. Programs are not actually loaded by this statement, so the contents of memory remain unaffected.

When this statement is executed, all files preceding that specified in <file descriptor> are skipped; after the file is checked, the head of the external cassette recorder is positioned to the end of the file checked (to the beginning of the next file).

If <file descriptor> is omitted, the first file found is checked.

LOC

Format **LOC (<file number>)**

Purpose With random access files, returns the record number which will be used by the next GET or PUT statement if that statement is executed without specifying a record number. With sequential files, returns the number of file sectors (128-byte areas) which have been read or written since the file was opened.

When the specified file is the PX-4's RS-232C interface, the LOC function returns the number of bytes of data in the RS-232C receive buffer.

Remarks This function can be used to control the flow of program execution according to the number of records or file sectors which have been accessed by a program since the file was opened.

Example

```
10 ON ERROR GOTO 160
20 OPEN"R",#1,"A:LOCTEST",5
30 PRINT "OUTPUT"
40 FIELD#1,5 AS A$
50 FOR A=1 TO 20:LSET A$=STR$(A):PRINT STR$(A);:PUT#1,A:NEXT
60 PRINT
70 CLOSE
80 OPEN"R",#1,"A:LOCTEST",5
90 PRINT "INPUT"
100 FIELD #1,5 AS A$
110 IF LOC(1)>10 THEN 150
120 GET#1
130 PRINT A$;
140 GOTO 110
150 ERROR 230
160 IF ERR=230 THEN PRINT:PRINT "INPUT PAST LIMIT"
170 END

OUTPUT
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
INPUT
 1    2    3    4    5    6    7    8    9    10   11
INPUT PAST LIMIT
Ok
```

LOCATE

Format **LOCATE [<X>],[,[<Y>][,<cursor switch>]]**

Purpose Moves the cursor to the specified position.

Remarks This statement moves the cursor to the screen position whose horizontal character coordinate is specified by <X> and whose vertical character coordinate is specified by <Y>. The value specified for <X> must be in the range from 1 to Xmax and that specified for <Y> must be in the range from 1 to Ymax, where Xmax and Ymax are determined by the size of the virtual screen.

<cursor switch> is a switch which determines the status of the cursor following execution of the LOCATE statement. Cursor display is turned off if 0 is specified for <cursor switch>, and cursor display is turned on if 1 is specified. Normally, the cursor is not displayed during execution of BASIC programs; however, it can be displayed by executing a LOCATE statement with 1 specified for <cursor switch>. The BASIC interpreter also forcibly sets the cursor switch to 1 whenever it returns to the command input mode. (The cursor is also always displayed during execution of an INPUT or LINE INPUT statement, regardless of the status of the cursor switch.)

MO error (Missing operand) — A required operand was not specified in the statement.

FC error (Illegal function call) — The number specified in one of the statement operands was outside of the prescribed range.

OV error (Overflow) — The number specified in one of the statement operands was outside of the prescribed range.

LOF

Format **LOF (<file no.>)**

Purpose Returns the size of a file.

Remarks When the file specified in <file no.> is a disk file, the LOF function returns the size of that file. When the file is open in the "R" mode, the size of the file is calculated based on the record size specified when the file was opened and the maximum record number which has been written to that file. When the file is open in the "I" or "O" mode, its size is calculated based on the number of records written and a record size of 128 bytes.

If the file specified in <file no.> is the RS-232C interface, the LOF function returns the number of bytes remaining in the receive buffer.

Under CP/M, the size of files is determined in 128-byte units. Therefore, if a record size of less than 128 bytes is used for a random access file, the value returned by the LOF function will be greater than that indicated by the maximum record number output.

NOTE:
When the LOF function is executed against a disk file, the disk must be accessed in order to determine the size of the file. Therefore, when a random access file is used in a retrieval program or the like, execution time will be greatly increased if the LOF function is executed repeatedly. To prevent this, substitute the LOF value into a variable at the beginning of the retrieval loop, then refer to the variable inside the loop instead of the LOF function.

LOG

Format **LOG(X)**

Purpose Returns the natural logarithm of X.

Remarks The value specified for X must be greater than zero. LOG(X) is calculated to the precision of the numeric type of expression X.

To obtain the logarithm to another base the mathematical conversion has to be carried out as in the first example below.

To obtain a number from its logarithm (i.e. its antilogarithm) use EXP (X) as shown in the second example.

Example 1

```
10 CLS
20 INPUT "What base logarithm do you want ";N
30 PRINT:INPUT "What number do you want the log of ";X
40 Z = (LOG(X)/LOG(N)):'THIS IS THE FORMULA FOR CONVERTING
NATURAL LOGS TO OTHER BASES
50 PRINT: PRINT "Log to the base ";N;" of ";X;" is ";Z
60 END


What base logarithm do you want ? 10
What number do you want the log of ? 100
Log to the base 10 of 100 is 2
Ok


What base logarithm do you want ? 8
What number do you want the log of ? 34
Log to the base 8 of 34 is 1.69582
Ok
```

Example 2

```
10 CLS
20 INPUT "What is the number of which you want the natural
   log ";X
30 PRINT:PRINT"The log to the base e of ";X;" is ";LOG(X)
40 PRINT:PRINT"The antilog (given by EXP(X)) is ";EXP(LOG(X))
50 END
```

```
What is the number of which you want the natural log ? 23

The log to the base e of  23  is  3.13549

The antilog (given by EXP(X)) is  23
Ok
```

```
What is the number of which you want the natural log ? 657

The log to the base e. of  657  is  6.48769

The antilog (given by EXP(X)) is  657
Ok
```

LOGIN

Format **LOGIN <program area no.>[,R]**

Purpose Switches between BASIC program areas.

Remarks A number from 1 to 5 is specified in <program area no.>, indicating one of the five BASIC program areas. When the R option is not specified, executing this command causes the BASIC interpreter to switch to the specified program area and stand by for entry of commands in the direct mode. In this case, the variable area is cleared and all open files are closed.

When the R option is specified, the specified program area is selected and the program in that area is executed immediately, starting with its first line. In this case, the variable area is not cleared and any files which are open at the time of command execution remain open.

FC error (Illegal function call) — A number other than 1 to 5 was specified in <program area no.>.

MO error (Missing operand) — A required operand was not specified in the command.

LPOS

Format **LPOS(X)**

Purpose Returns the current position of the print head pointer in the printer output buffer.

Remarks The maximum value returned by LPOS is determined by the line width which has been set by the WIDTH LPRINT statement, and does not necessarily correspond to the physical position of the print head. This is especially true if a control character has been sent to the printer; see the program below for an example of this.

X is a dummy argument, and may be specified as any numeric expression.

Example In the example below, at the end of line 100 ten characters have been printed. The position in the buffer is thus 11. Line 20 adds two control characters compatible with EPSON printers to cause the printer to change the print style. The first character is a control character, which is ignored by the LPOS function. The second character is used by the printer but not printed; it is in fact the letter "E". The position in the buffer as returned by LPOS is now 12 because only this character has been added. Line 30 adds another ten characters to the line, and thus LPOS returns a value of 22.

```
10 LPRINT "1234567890";:GOSUB 100
20 LPRINT CHR$(27);CHR$(69);:GOSUB 100
30 LPRINT "1234567890";:GOSUB 100
40 END
100 A = LPOS(X):PRINT"Print head pointer is at position ";A
110 RETURN
```

```
12345678901234567890
```

```
Ok
run
Print head pointer is at position  11
Print head pointer is at position  12
Print head pointer is at position  22
Ok
```

LPRINT/LPRINT USING

Format **LPRINT [<list of expressions>]**
LPRINT USING <format string>;<list of expressions>

Purpose These statements are used to output data to a printer connected to the PX-4.

Remarks These statements are used in the same manner as the PRINT and PRINT USING statements, but output is directed to the printer instead of the display screen.

See also **PRINT, PRINT USING**

Example 1

```
10 A = 3
20 A$ = "There are "
30 LPRINT A$;A;" vowels in 'computer'"
40 END

There are  3  vowels in 'computer'
```

Example 2

```
10 'THE LPRINT COMMAND
20 A = 3
30 A$ = "There are "
40 LPRINT A$;A;" vowels in 'computer'"
50 LPRINT
60 'THE LPRINT USING "!" COMMAND
70 LPRINT USING "!";"AAA";"BBB";"CCC"
80 LPRINT
90 'THE LPRINT USING "\  \" COMMAND
100 A$ = "123456"
110 B$ = "ABCDEF"
120 LPRINT USING "\ \";A$;B$
130 LPRINT USING "\  \";A$;B$
140 LPRINT
150 'THE LPRINT USING "&" COMMAND
160 LPRINT USING "&";A$;" = ";B$
170 LPRINT
```

```
180 'THE LPRINT USING "#" COMMAND
190 LPRINT USING "####";1;.12;12.6;12345
200 LPRINT
210 LPRINT USING "###.##   ";123;12.34;123.456;.12
220 LPRINT
225 'THE LPRINT USING "+#" COMMAND
230 LPRINT USING "+#### ";123
240 LPRINT
245 'THE LPRINT USING "#-" COMMAND
250 LPRINT USING "####- ";345;-456
260 LPRINT
265 'THE LPRINT USING "**" COMMAND
270 LPRINT USING "**####.##  ";12.35;123.555;555555.88#
280 LPRINT
290 'THE LPRINT USING "$$" COMMAND
300 LPRINT USING "$$####.##  ";12.35;123.555;555555.88#
310 LPRINT
320 'THE LPRINT USING "$**" COMMAND
330 LPRINT USING "$**####.##  ";12.35;123.555;555555.88#
340 LPRINT
350 'THE LPRINT USING "**$" COMMAND
360 LPRINT USING "**$####.##  ";12.35;123.555;555555.88#
370 LPRINT
380 'THE LPRINT USING "##,.##" COMMAND
390 LPRINT USING "#######,.##";555555.88#
400 LPRINT
405 'THE LPRINT USING "##.##^^^^^" COMMAND
410 LPRINT USING "###.##^^^^  ";123.45;12.345;1234.5
415 LPRINT
425 'USING THE UNDERSCORE
435 LPRINT USING "###_%";123
445 LPRINT
455 'USING OTHER CHARAACTERS
465 LPRINT USING "##/##/##";12;34;56
475 LPRINT
485 LPRINT USING "(###)";123
495 LPRINT
505 LPRINT USING "<###>";123
515 LPRINT
```

LSET/RSET

Format

LSET <string variable> = <string expression>
RSET <string variable> = <string expression>

Purpose

These statements move data into a random file buffer to prepare it for storage in a random access file with the PUT statement.

Remarks

<string variable> is a variable which has been assigned to positions in a random file buffer with the FIELD statement. <string expression> is any string constant or string variable.

If the length of <string expression> is less than the number of bytes which were assigned to the specified variable with the FIELD statement, the LSET (Left SET) statement left-justifies the string data in the variable and the RSET (Right SET) statement right-justifies it. The positions following left-justified data and those preceding right-justified data are padded with spaces.

If the length of <string expression> is greater than the number of bytes assigned to the specified variable, excess characters are truncated from the right end of <string expression> when it is moved into the buffer. (This is true for both the LSET and RSET statements.)

Numeric values must be converted to strings before they can be moved into a random file buffer with the LSET or RSET statements. This is done using the MKI$, MKS$, and MKD$ functions described elsewhere in this Chapter, and Chapter 5.

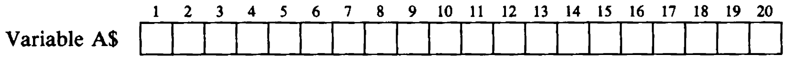
See also

FIELD, GET, OPEN, PUT

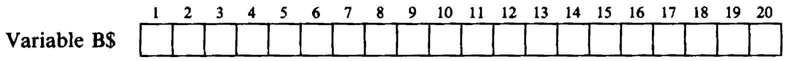
Example See Chapter 5 for examples of use of LSET/RSET in a program.

NOTE:
The LSET and RSET statements can also be used to left or right justify a string in a string variable which has not been assigned to a random file buffer. For example, the following program left-justifies character string "CAMERA" in a 20-character field prepared in variable A$ and right-justifies character string "MAY 8, 1984" in variable B$. This procedure can be very useful when formatting data for output.

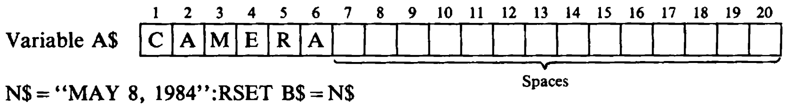
```
10 A$=STRING$(20, " ")
```



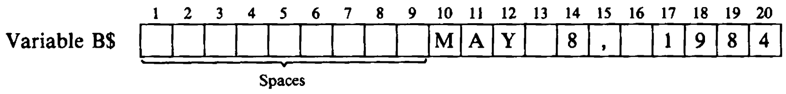
```
20 B$=A$
```



```
30 N$="CAMERA":LSET A$=N$
```



```
40 N$="MAY 8, 1984":RSET B$=N$
```



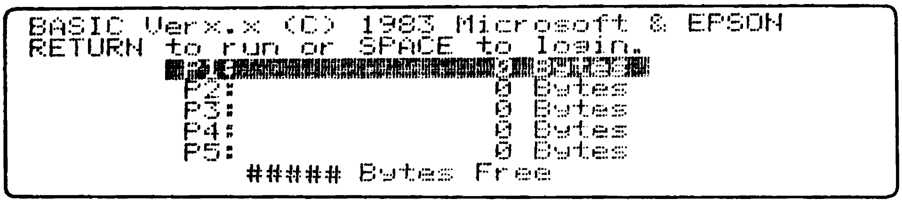
MENU

Format **MENU**

Purpose Returns BASIC to the start-up menu.

Remarks Executing this command returns the BASIC interpreter to the BASIC menu screen which is displayed at the time of start-up. It also clears all variables and closes any files which are open. This command resets the virtual screen size to 40 columns × 50 lines.

See also **LOGIN, WIDTH**



```
BASIC Verx.x (C) 1983 Microsoft & EPSON
RETURN to run or SPACE to login.
      [illegible]
       P2:                  0 Bytes
       P3:                  0 Bytes
       P4:                  0 Bytes
       P5:                  0 Bytes
          ###### Bytes Free
```

MERGE

Format **MERGE <file descriptor>**

Purpose Merges a program from a disk device or the RS-232C interface with the program in the currently logged in program area.

Remarks Specify the device name, file name, and file name extension under which the file was saved in <file descriptor>. The device name can be omitted if the file is in the currently active drive (the drive which was logged in under CP/M at the time BASIC was started). If the file name extension is omitted, ".BAS" is assumed.

The file being merged must have been saved in ASCII format. Otherwise, a BF error (Bad file mode) will occur.

If any lines of the program being merged have the same numbers as lines of the program in memory, the merged lines will replace the corresponding lines in memory. Thus, a program brought into the BASIC program area with the MERGE statement may be thought of as an overlay which replaces corresponding lines previously included in the program area.

BASIC always returns to the command level following execution of a MERGE command.

See also **SAVE**

Examples First type in and save the following programs, making sure they are saved in ASCII format by using the ",A" extension as shown below.

```
10 'PROGRAM "MERGE1" TO BE MERGED WITH "MERGE2" THEN "MERGE3"
20 PRINT "HELLO MUM"
30 PRINT
Ok
SAVE "A:MERGE1",A
Ok
```

```
NEW
Ok
30 'PROGRAM "MERGE3" TO BE MERGED WITH "MERGE1"
40 PRINT "GOODBYE DAD"
50 END
SAVE "A:MERGE3",A
Ok
```

Clear the program with the NEW command, and type in the following program.

```
50 'PROGRAM "MERGE 2" WITH WHICH "MERGE1" WILL BE MERGED
60 PRINT "GOODBYE MUM"
70 END
```

Now type

```
MERGE "A:MERGE1"
```

If the program in memory is listed it will be seen to consist of the lines from both programs as follows:

```
10 'PROGRAM "MERGE1" TO BE MERGED WITH "MERGE2" THEN "MERGE3"
20 PRINT "HELLO MUM"
30 PRINT
50 'PROGRAM "MERGE 2" WITH WHICH "MERGE1" WILL BE MERGED
60 PRINT "GOODBYE MUM"
70 END
```

Now type in the following.

```
MERGE "A:MERGE3"
```

List to see that the result is as follows.

```
10 'PROGRAM "MERGE1" TO BE MERGED WITH "MERGE2" THEN "MERGE3"
20 PRINT "HELLO MUM"
30 'PROGRAM "MERGE3" TO BE MERGED WITH "MERGE1"
40 PRINT "GOODBYE DAD"
50 END
60 PRINT "GOODBYE MUM"
70 END
```

MID$

Format

As a statement
MID$ (<string expression 1>,n[,m])=<string expression 2>

As a function
MID$ (X$,J[,K])

Purpose

As a statement, replaces a portion of one string with another. As a function, returns the character string from the middle of string expression X$ which consists of the K characters beginning with the Jth character.

Remarks

When MID$ is used as a statement, n and m are integer expressions and <string expression 1> and <string expression 2> are string expressions. In this case, MID$ replaces the characters beginning at position n in <string expression 1> with the characters of <string expression 2>. If the m option is specified, the first m characters of <string expression 2> are used in making the replacement; otherwise, all of <string expression 2> is used. However, the number of characters replaced cannot exceed the length of the portion of <string expression 1> which starts with character n.

For example:

```
10 A$="ABCDEFG" : LET B$="wxyz"
20 MID$(A$,3)=B$
30 PRINT A$
```

will give the result "ABwxyzG" for A$. Whereas,

```
10 A$="ABCDEFG" : LET B$="wxyz"
20 MID$(A$,3,2)=B$
30 PRINT A$
```

will give a value of "ABwxEFG".

When MID$ is used as a function, J and K must be integer expressions in the range from 1 to 255. If K is omitted, or there are fewer than K characters to the right of the Jth character,the string returned consists of all characters to the right of the Jth character. If the value specified for J is greater than the number of characters in X$, MID$ returns a null string.

See also **LEFT$, RIGHT$**

Example **INSERT MID$**

```
10 A$ = "Computers  is great!"
20 B$ = "are"
30 PRINT A$
40 MID$(A$,11) = B$
50 PRINT A$
60 PRINT
70 B$ = "super-duper"
80 MID$(A$,15,5) = B$
90 PRINT A$
100 PRINT
110 B$ = MID$(A$,7,12)
120 PRINT B$
130 END
```

```
Computers  is great!
Computers are great!

Computers are super!

ers are supe
```

MKI$/MKS$/MKD$

Format **MKI$(<integer expression>)**
MKS$(<single precision expression>)
MKD$(<double precision expression>)

Purpose Converts numeric values to string values for storage in random access files.

Remarks Numeric values must be converted to string values before they can be placed in a random file buffer by a LSET or RSET statement for storage with a PUT statement. MKI$ converts integers to 2-byte strings, MKS$ converts single precision numbers to 4-byte strings, and MKD$ converts double precision numbers to 8-byte strings.

Unlike the STR$ function, which produces an ASCII string whose characters correspond to the digits of the decimal representation of a number, these functions convert numeric values to characters whose ASCII codes correspond to the binary coded decimal values with which corresponding values are stored in variable memory. In many instances, less disk space is required for storage of numbers which are converted to strings using the MKI$/MKS$/MKD$ functions.

See also **CVI/CVS/CVD**

For examples of the use of these functions, see the section of Chapter 6 dealing with random access files.

```
FILES "A:
MERGE1   .BAS  MERGE3   .BAS
Ok
NAME "A:MERGE1.BAS" AS "A:CHAIN.BAS"
Ok
```

```
FILES "A:
CHAIN    .BAS  MERGE3   .BAS
Ok
```

MOTOR

Format **MOTOR [<switch>]**

Purpose Controls the motor of the external audio cassette.

Remarks This statement turns on or off the external audio cassette's REM (remote) terminal. The REM terminal is turned on if ON is specified for <switch>, and is turned off if OFF is specified. If <switch> is not specified, execution of this statement reverses the setting of the REM terminal.

MOUNT

Format **MOUNT**

Purpose Reads the microcassette tape directory into memory and enables the tape for use.

Remarks A MOUNT command must be executed before data can be written to or read from a microcassette tape. The MOUNT command prepares the tape for access by reading its directory into memory.

Before executing the MOUNT command, any previously mounted tape must be unmounted by executing the REMOVE command.

IO error (Device I/O error) — The drive does not contain a cassette tape.

DR error (Disk read error) — An error occurred while the microcassette tape was being read.

AC error (Tape access error) — The MOUNT command was executed without unmounting the previous tape with the REMOVE command.

See also **REMOVE**
Section 2.4.5

NAME

Format **NAME <old filename> AS <new filename>**

Purpose Changes the name of a disk device file.

Remarks Both <old filename> and <new filename> are specified as a device name, file name, and extension. The device name may be omitted if the file resides on the disk device which is currently active.

The file name specified in <old filename> must be that of a currently existing file, and that specified in <new filename> must be a name which is not assigned to any other file belonging to the applicable disk device. If <old filename> is not the name of an existing file, an NE error (File not found) will occur; if <new filename> is already assigned to an existing file, an FE error (File already exists) will occur. If the file being renamed has a file name extension, that extension must be specified in <old filename>.

If the NAME command is executed with a file which is already open, there is no guarantee that the file will remain intact. Using the CLOSE command will ensure that all files are closed.

This command changes only the name of the specified file; it does not rewrite the file to another area in the storage medium.

See also **FILES**

NEW

Format NEW

Purpose Deletes the program in the currently logged in program area and clears all variables.

Remarks Enter NEW at the command level to clear the memory before starting to enter a new program. BASIC always returns to the command level upon execution of a NEW command.

An FC error (Illegal function call) will occur when this command is executed if program editing has been disabled by executing a TITLE command with the P (protect) option specified.

See also **TITLE**

Example

```
LOGIN 1
P1:DEMOPROG      23 Bytes
Ok
NEW
Ok
```

```
LOGIN 1
P1:               0 Bytes
Ok
```

OCT$

Format **OCT$(X)**

Purpose Returns a string which represents the octal value of X.

Remarks The numeric expression specified in the argument is rounded to the nearest integer value before it is evaluated.

A description of using numbers and numeric variables is given in Chapter 2.

Example

```
10 CLS
20 INPUT "What value do you want to convert ";X
30 PRINT
40 PRINT "The octal value of ";X;" is ";OCT$(X)
50 FOR J = 1 TO 3000:NEXT
60 GOTO 10
```

```
What value do you want to convert ? 23
The octal value of  23  is 27
```

```
What value do you want to convert ? 983
The octal value of  983  is 1727
```

ON COM(n) GOSUB...RETURN

Format **ON COM(n) GOSUB [<line number>]...**
RETURN [<line number>]

Purpose Defines the starting point of the communication trap routine to which processing branches when a communication line interrupt is generated.

Remarks This statement defines the starting point of the communication trap routine to which processing branches when a communication port input interrupt is generated. The n in COM(n) indicates the communication port number, and is specified as a number from 0 to 3. <line number> is the first line of the communication trap routine. Communication interrupts are disabled if 0 is specified for <line number> or the parameter is omitted. Processing is returned to the main routine from the communication trap by the RETURN statement. If RETURN is executed by itself, processing resumes from the point at which it was interrupted; if a line number is specified following RETURN, processing resumes with the first statement in that line.

Since only one communication device (COMn:) can be open at a time, there is no possibility of interrupts being generated from two communication ports simultaneously. The COM(n) ON statement must be executed to allow branching to the communication V trap. Communication interrupts are not generated if COM(n) OFF is executed. If COM(n) STOP is executed, no interrupt is generated but data received is saved; an interrupt is then generated the next time COM(n) ON is executed. Once one interrupted has been generated, other are deferred just as if COM(n) STOP were executed. Unless the COM(n) OFF statement is executed in the communication trap routine, generation of deferred interrupts is automatically reenabled upon return to the main routine in the same manner as when COM(n) ON is interrupted. Communication interrupts are not generated except during execution of a BASIC program. Further, all interrupts are automatically disabled if any error occurs. When using the RETURN <line number> statement, remember that the status of any subroutines (GOSUB) or loops (WHILE or FOR/NEXT) being processed remains unchanged when processing branches to the communication trap.

ON ERROR GOTO

Format **ON ERROR GOTO [<line number>]**

Purpose Causes program execution to branch to the first line of an error processing routine when an error occurs.

Remarks Execution of the ON ERROR GOTO statement enables error trapping; that is, it causes execution of a program to branch to a user-written error processing routine beginning at the program line specified in <line number> whenever any error (such as a syntax error) occurs. This error processing routine then evaluates the error and/or directs the course of subsequent processing. For example, it may be written to check for a certain type of error or an error occurring in a certain program line, then to resume execution at a certain point in the program depending on the result.

If subsequent error trapping is to be disabled, execute ON ERROR GOTO 0. If this statement is encountered in an error processing routine, program execution stops and BASIC displays the error message for the error which caused the trap. It is recommended that all error processing routines include an ON ERROR GOTO 0 statement for errors for which are not provided for in the error recovery procedures. If <line number> is not specified, the effect is the same as executing ON ERROR GOTO 0.

See also **ERROR, RESUME, ERL/ERR, Appendix A**

Example See the program example under ERROR.

NOTE:
BASIC always displays error messages and terminates execution for errors which occur in the body of an error processing routine; that is, error trapping is not performed within an error processing routine itself.

ON...GOSUB/ON...GOTO

Format

ON <numeric expression> GOSUB <list of line numbers>
ON <numeric expression> GOTO <list of line numbers>

Purpose

Transfers execution to one of several program lines specified in <list of line numbers> depending on the value returned when <numeric expression> is evaluated.

Remarks

The value of <numeric expression> determines to which of the line numbers listed execution will branch. If the value of <numeric expression> is 1, execution will branch to the first line number in the list; if it is 2, execution will branch to the second line number in the list; and so forth. If the value is a non-integer, the fractional portion is rounded.

An FC error (Illegal function call) will occur if the value of <numeric expression> is negative or greater than 256.

With the ON...GOSUB statement, each program line indicated in <list of line numbers> must be a line of a subroutine.

See also

GOSUB...RETURN, GOTO

Example 1

```
10 CLS
20 INPUT "Type in a number from 5 to 10 ";X
30 Y = X - 4
40 ON Y GOTO 60,80,100,120,140,160
50 END
60 PRINT X;"- 4 = ";Y;" so this is line 60"
70 GOTO 20
80 PRINT X;"- 4 = ";Y;" so this is line 80"
90 GOTO 20
100 PRINT X;"- 4 = ";Y;" so this is line 100"
110 GOTO 20
120 PRINT X;"- 4 = ";Y;" so this is line 120"
130 GOTO 20
140 PRINT X;"- 4 = ";Y;" so this is line 140"
150 GOTO 20
160 PRINT X;"- 4 = ";Y;" so this is line 160"
170 GOTO 20
```

```
Type in a number from 5 to 10 ? 7
 7 - 4 =  3  so this is line 100
Type in a number from 5 to 10 ? 9
 9 - 4 =  5  so this is line 140
```

Example 2

```
10 CLS
20 INPUT "Type in a number from 1 to 5 ";X
30 ON X GOSUB 50,60,70,80,90
40 GOTO 20
50 PRINT "ONE":RETURN
60 PRINT "TWO":RETURN
70 PRINT "THREE":RETURN
80 PRINT "FOUR":RETURN
90 PRINT "FIVE":RETURN
```

```
Type in a number from 1 to 5 ? 2
TWO
Type in a number from 1 to 5 ? 4
FOUR
Type in a number from 1 to 5 ? 3
THREE
```

NOTE:
Only numeric expressions can be used to control branching with the ON...GOSUB and ON...GOTO statements. However, it is possible to derive numeric values from string values by using functions such as ASC and INSTR$. For example, the following sample program derives numeric results for the ON...GOSUB and ON...GOTO statements based on input of string values.

Example 3

```
10 CLS
20 INPUT "Type in a word beginning with A, B or C ";A$
30 X = ASC(A$)
40 Y = X - 64
50 ON Y GOSUB 70,110,150
60 END
70 PRINT "The ASCII code for A is 65, so line 40 subtracts
80 PRINT "64 from this code to give 1, thus causing the"
90 PRINT "subroutine at line 70 to be executed"
100 RETURN
110 PRINT "The ASCII code for B is 66, so line 40 subtracts
120 PRINT "64 from this to give 2, causing the subroutine"
130 PRINT "at line 110 to be executed"
140 RETURN
150 PRINT "The ASCII code for C is 67, and line 40 subtracts
160 PRINT "64 from this giving 3 so that the subroutine on"
170 PRINT "line 150 is executed."
180 RETURN

Type in a word beginning with A, B or C ? Albatross
The ASCII code for A is 65, so line 40 subtracts
64 from this code to give 1, thus causing the
subroutine at line 70 to be executed
Ok
Type in a word beginning with A, B or C ? Carousel
The ASCII code for C is 67, and line 40 subtracts
64 from this giving 3 so that the subroutine on
line 150 is executed.
Ok
```

OPEN

Format **OPEN "<mode>",[#]<file number>,<file descriptor>, [<record length>]**

Purpose The OPEN statement enables input/output access to a disk device file or other device.

Remarks Disk device files must be OPENed before any data can be input from or output to such files. The OPEN statement allocates a buffer for I/O to the specified file and determines the mode of access in which that buffer will be used. <mode> is a string expression whose first character is one of the following.

O or o	Specifies the sequential output mode.
I or i	Specifies the sequential input mode.
R or r	Specifies the random input/output mode.

Any mode can be specified for a disk device file, but only the "I" or "O" modes can be specified for devices such as the RS-232C interface or printer.

<file number> is an integer expression from 1 to 15 which specifies the number by which the file is to be referenced in I/O statements as long as the file is open. The value of <file number> is limited to the maximum specified in the /F: option if this option is used when BASIC is started up. Since this is 3 in the default mode, it is necessary to ensure the /F: option is specified before a program is run if more than 3 files are required.

<file descriptor> is a string expression which conforms to the rules for naming files (see Chapter 2).

<record length> is an integer expression which, if specified, sets the record length for random access files. If not specified, the record length is set to 128 bytes.

Disk files and files on the RAM disk can be open for sequential input or random access under more than one file number at a time. However, a given sequential access file can only be opened for sequential output under one file number at a time, and such a file cannot be open in both the sequential input and sequential output modes concurrently.

Microcassette files can only be open for sequential input, random access, or sequential output under one file number at a time. Further, only one microcassette file can be open at any given time.

The RS-232C interface can be open concurrently in the sequential input mode and the sequential output mode, but cannot be opened in the random access mode.

See also Chapters 5 and 6.

Example For programming examples, see Chapters 5 and 6 and the explanation of EOF.

OPTION BASE

Format **OPTION BASE <base number>**

Purpose Declares the minimum value of array subscripts.

Remarks When BASIC is started, the minimum value of array subscripts is set to 0; however, in certain applications it may be more convenient to use variable arrays whose subscripts have a minimum value of 1. Specifying 1 for the value of <base number> in this statement makes it possible to set the minimum subscript base to one.

Once the subscript base has been set by executing this statement, it cannot be reset until a CLEAR statement has been executed; executing a CLEAR statement restores the option base to 0. Further, OPTION BASE 1 cannot be executed if any values have previously been stored in any array variables. A DD error (Duplicate Definition) will occur if the OPTION BASE statement is executed under either of these conditions.

See also **DIM**

Example

```
10 CLEAR
20 PRINT "Memory free following CLEAR:";FRE(0)
30 OPTION BASE 0
40 DIM A(5,5,5,5)
50 PRINT "Memory free after DIM A(5,5,5,5) with OPTION BASE
0:";FRE(0)
60 CLEAR
70 OPTION BASE 1
80 DIM A(5,5,5,5)
90 PRINT "Memory free after DIM A(5,5,5,5) with OPTION BASE
1:";FRE(0)

run
Memory free following CLEAR: 7324
Memory free after DIM A(5,5,5,5) with OPTION BASE 0: 2125
Memory free after DIM A(5,5,5,5) with OPTION BASE 1: 4809
Ok
```

OPTION COUNTRY

Format **OPTION COUNTRY <string>**

Purpose Selects the international character set.

Remarks This statement selects the international character set which is used for keyboard input/output, display, and output to the printer. The character set selected by this statement is determined by the first character of the <string> parameter as follows.

"U" or "u"......	U.S.A
"F" or "f"	France
"G" or "g"......	Germany
"E" or "e"......	England
"D" or "d"......	Denmark
"W" or "w"	Sweden
"I" or "i"	Italy
"S" or "s"	Spain
"N" or "n"......	Norway

After execution of the OPTION COUNTRY statement, the specified character set is used for all output to the LCD screen or printer. The currency symbol output with the PRINT USING statement is also changed to the corresponding symbol in the applicable character set.

The OPTION COUNTRY statement can only be used with the European version of PX-4 BASIC; if it is executed with the American version, a Syntax error will result.

Country selection can be changed by DIP switch setting or CONFIG.COM of CP/M utility program. See Appendix N and Operating Manual.

OPTION CURRENCY

Format **OPTION CURRENCY <string>**

Purpose Changes the currency symbol.

Remarks This statement specifies the character which is output for the currency symbol when two consecutive currency symbol codes (&H5C with the Japanese version, and &H24 with the export versions) are specified at the beginning of the numeric formatting string in the PRINT[#] USING statement.

The character output for the currency symbol is the first character in <string>, and can be any character which is included in the character set.

OUT

Format **OUT <integer expression 1>,<integer expression 2>**

Purpose Used to send data to a machine output port.

Remarks The data to be output is specified in <integer expression 2> and the port to which it is to be output is specified in <integer expression 1>. Both values must be in the range 0 to 255.

See also **INP**

NOTE:
Use of this statement requires sound knowledge of the PX-4 firmware. Incorrect use may corrupt programs or data held in memory, including BASIC itself.

PCOPY

Format **PCOPY <program area no.>**

Purpose Copies the contents of the currently selected program area to another program area.

Remarks This command copies the contents of the currently selected program area to the program area whose number is specified in <program area no.>. The number specified must be in the range from 1 to 5, and must be the number of an area which does not contain a program. It must also be the number of an area other than the currently selected program area.

Programs which have been saved using the protect save function cannot be transferred between program areas with this command.

FC error (Illegal function call) — A number other than 1 to 5 was specified for <program area no.>; the currently selected program area was specified as the destination area; the specified program area was not empty, or; an attempt was made to PCOPY a program saved using the protect save function.

OM error (Out of memory) — The amount of free memory available was not sufficient to allow the program to be copied.

PEEK

Format **PEEK(J)**

Purpose Returns one byte of data from the memory address specified for J as an integer from 0 to 255.

Remarks As the name suggests, PEEK is a function to look at memory locations and return the value of the contents of the location PEEKed. The contents of the location are not changed by inspecting it. For the beginner learning BASIC, PEEK and the allied command POKE (which allows the contents of a location to be changed) are commands which are difficult to understand, because it is not always easy to see the function of the values used. They can be used in a large number of ways. Also the values are very computer dependent. It is often possible to type many BASIC programs into the PX-4 when they have been written for other computers even if the BASIC is another version of MICROSOFT BASIC. When PEEK and POKE commands are used, they are invariably not directly translatable. For example with many computers it is possible to PEEK and POKE the memory reserved for the screen. This is not possible directly with the PX-4.

The integer value specified for J must be in the range from 0 to 65535.

See also **POKE**

Example If location 4 is PEEKed, the number returned will correspond to the drive which is the default drive when returning to CP/M. This is not the default drive for BASIC. If the value returned is "0" then A: is the default drive, if it is "1" then it is drive "B" and so on.

POINT

Format **POINT (<horizontal coordinate>,<vertical coordinate>)**

Purpose Returns the status of the dot at the specified screen coordinates.

Remarks The POINT function returns the status of the dot at the specified graphic coordinates as a function code. If the dot is set (turned on), the value returned is 7; if the dot is reset (turned off), the value returned is 0. If the specified graphic coordinates are outside the screen, the value returned is −1.

OV error (Overflow) — An argument specified was not in the range from −32768 to 32767.

POKE

Format **POKE <integer expression 1>,<integer expression 2>**

Purpose Writes a byte of data into memory.

Remarks The address into which the data byte is to be written is specified in <integer expression 1> and the value which is to be written into that address is specified in <integer expression 2>. The value specified in <integer expression 2> must be in the range from 0 to 255.

The complement of the POKE statement is the PEEK function, which is used to check the contents of specific addresses in memory. Used together, the POKE statement and PEEK function are useful for accessing memory for data storage, writing machine language programs into memory, and passing arguments and results between BASIC programs and machine language routines.

See also **PEEK**

Example An example of using BASIC to POKE a machine code routine is described under the CALL command.

In the example under the PEEK command, it was shown how to find out which drive would be the active drive when exiting to CP/M. This can be altered with the BASIC POKE command. If you have any BASIC programs in memory which you want to save, save them first. In direct mode type "POKE 4,1", then type "SYSTEM" and press RETURN. You will be transferred to either the system menu or the CP/M command line. If the menu is active, use ESC to return to the CP/M command line. The active drive should be shown as "B>" which will normally be assigned to one of the ROM sockets.

WARNING:
Since this statement changes the contents of memory, the work area used by BASIC may be destroyed if it is used carelessly. This can result in erroneous operation, so be sure to check the memory map to confirm that the address specified is in a usable area.

POS

Format **POS(<file no.>)**

Purpose Returns the current position of the print head, cursor, or file output buffer pointer.

Remarks When "0" is specified for <file no.>, this function returns the current horizontal position of the cursor. The value returned ranges from 1 to the number of columns in the virtual screen currently being output.

When a number other than "0" is specified for <file no.>, this function returns the current position of the pointer in the output buffer for the specified file. Here, the file must be one which has been opened in the sequential output mode or the random access mode. The value returned for a file opened in the sequential input mode will have no meaning.

When the value specified for <file no.> is other than "0", the value returned by the POS function will be a number in the range from 1 to 255; immediately after the file is opened or a carriage return is output, the value returned is "1".

If the file specified is "LPT0:", this function returns the same value as the LPOS function.

POWER

Format

1) POWER OFF [,RESUME]
2) POWER | <duration> |
| CONT |

Purpose

Allows the power to be turned off by program and controls the auto power-off function.

Remarks

Executing POWER OFF turns off the power in the restart mode. When the power goes off in the restart mode, BASIC is restarted by a hot start when the power is turned back on.

When POWER OFF, RESUME is executed, the power goes off in the continue mode. When the power is then turned back on (by moving the power switch from ON to OFF, and then back ON again), BASIC program execution resumes with the statement following the POWER statement which turned off the power.

When the power goes off in the restart mode, it is turned back on again if the wake-up time set by a WAKE statement is reached before the power is manually turned back on.

When the wake time is reached after the power has gone off in the continue mode, program execution resumes with the statement following the POWER statement which turned off the power.

POWER <duration> specifies the amount of time which will elapse before the auto shut-off function automatically turns off the power when a certain amount of time passes in the direct mode without anything being entered from the keyboard.

Executing POWER CONT disables the auto power-off function. The auto power-off function can later be reenabled by executing POWER <duration>.

FC error (Illegal function call) — The value specified for <counter value> was outside the prescribed range.

PRESET

Format **PRESET [STEP] (X,Y)[,<function code>]**

Purpose Resets (turns off) the dot at the specified graphic screen coordinates.

Remarks This statement resets the dot at the graphic screen coordinates specified by (X,Y). When STEP is specified, relative coordinates are used.

When <function code> is specified, this statement sets or resets the dot at the specified position in the same manner as the PSET statement. If <function code> is omitted, the specified dot is reset.

After execution of the PRESET statement, the LRP (last reference pointer) is updated to the values specified for (X,Y).

FC error (Illegal function call) — The number specified in one of the statement operands was outside of the prescribed range.

OV error (Overflow) — The number specified in one of the statement operands was outside of the prescribed range.

PRINT

Format **PRINT [<list of expressions>]**

Purpose Outputs data to the LCD screen.

Remarks Executing a PRINT statement without specifying any expressions advances the cursor to the line following that on which it is currently located without displaying anything.

When a <list of expressions> is included, the values of the expressions are output to the display screen. Both numeric and string expressions may be included in the list. The positions in which items are displayed is determined by the delimiting punctuation used to separate items in the list.

Under BASIC, the screen is divided up into zones consisting of 14 spaces each. When items in <list of expressions> are delimited with commas, each succeeding value is displayed starting at the beginning of the following zone. When items are delimited with semicolons, they are displayed immediately following one another. Including one or more spaces between items has the same effect as a semicolon. Other display formats can be obtained by including the TAB, SPACE$, and SPC functions.

If a semicolon or comma is included at the end of the list of expressions, the cursor remains on the current display line and values specified in the next PRINT statement are displayed starting on the same line. If the list of expressions is concluded without a semicolon or comma, the cursor is moved to the beginning of the next line.

If values displayed by the PRINT statement will not fit on one display line, display is continued on the next line.

See also **LPRINT, PRINT USING, SPACE$, SPC, TAB**

Example

```
10 PRINT 123;456;789
20 PRINT 123,456,789
30 PRINT "123";"456";"789"
40 PRINT "ABC",
50 PRINT "DEF"
60 PRINT "ABC";
70 PRINT "DEF"

run
 123  456  789
 123           456             789
123456789
ABC            DEF
ABCDEF
Ok
```

NOTE:
A question mark (?) may be typed in place of the word PRINT when entering the PRINT statement. BASIC automatically converts question marks encountered during statement execution to PRINT statements.

PRINT USING

Format **PRINT USING <format string>;<list of expressions>**

Purpose Displays string data or numbers using a format specified by <format string>.

Remarks <format string> consists of special characters which determine the size and format of the field in which expressions are displayed. <list of expressions> consists of the string expressions or numeric expressions which are to be displayed. Each expression in <list of expressions> must be delimited from the one following it by a semicolon.

The characters which make up the <format string> differ according to whether the expressions included in <list of expressions> are string expressions or numeric expressions. The characters and their functions are as follows:

Format strings for string expressions

"!"
Specifies that the first character of each string included in <list of expressions> is to be displayed in a 1-character field.

Example 1

```
10 PRINT USING "!";"Aa";"bB";"Dc"

run
AbD
Ok
```

"\n spaces\"
Specifies that 2+n characters of each string in <list of expressions> is to be displayed. Two characters will be displayed if no spaces are included between the backslashes, three characters will be displayed if one space is included between the backslashes, and so on. Extra characters are ignored if the length of any string in <list of expressions> is greater than 2+n. If the length of the field is greater than that of a string, the string is left-justified in the field and padded on the right with spaces.

Example 2

```
10 A$="1234567"
20 B$="ABCDEFG"
30 PRINT USING "\   \";A$;B$
40 PRINT USING "\         \";A$;B$

run
12345ABCDE
1234567    ABCDEFG
Ok
```

"&"
Specifies that strings included in <list of expressions> are to be displayed exactly as they are.

Example 3

```
10 READ A$,B$
20 PRINT USING "&";A$;" ";B$
30 DATA EPSON,PX-4

run
EPSON PX-4
Ok
```

Format strings for numeric expressions

With numeric expressions, the field in which digits are displayed by a PRINT USING statement is determined by a format string consisting of the number sign (#) and a number of other characters. When the format string consists entirely of # signs, the length of the field is determined by that of the format string.

If the number of digits in numbers being displayed is smaller than the number of positions in the field, numbers are right justified in the field. If the number of digits is greater than the number of # signs, a percent sign (%) is displayed in front of the number and all digits are displayed.

Minus signs are displayed in front of negative numbers, but (ordinarily) positive numbers are not preceded by a plus sign.

The following is an example of use of the # sign in the numeric format string of a PRINT USING statement.

Example 4

```
10 PRINT USING "####   ";1;.12;12.6;12345
20 END

run
   1     0     13   %12345
Ok
```

Other special characters which may be included in numeric format strings are as follows:

"."

A decimal point may be included at any point in the format string to indicate the number of positions in the field which are to be used for display of decimal fractions. The position to the left of the decimal point in the field is always filled (with 0 if necessary). Digits to the right of the decimal point are rounded to fit into positions to the right of the decimal point in the field.

```
10 PRINT USING "###.##   ";123;12.34;123.456;.12
20 END

run
123.00   12.34   123.46    0.12
Ok
```

"+"

A plus sign (+) at the beginning or end of the format string causes the sign of the number (plus or minus) to be displayed in front of or behind the number.

Example 5

```
10 PRINT USING "+####";123;-123
20 END

run
 +123 -123
Ok
```

"_"

A minus sign at the end of the format string causes negative numbers to be displayed with a trailing minus sign.

Example 6

```
10 PRINT USING "####- ";345;-456
20 END

run
 345   456-
Ok
```

"**"

A double asterisk at the beginning of the format string causes leading spaces to be filled with asterisks. The asterisks in the format string also represent two positions in the display field.

Example 7

```
10 PRINT USING "******.##  ";
12.35;123.555;555555.88#
20 END

run
****12.35  ***123.56  555555.88
Ok
```

"$$"

A double dollar sign at the beginning of the format string causes the dollar sign (or other character selected with the OPTION CURRENCY statement) to be displayed immediately to the left of numbers displayed. The dollar signs in the format string also represent two positions in the display field (one of which is used for display of the dollar sign).

Example 8

```
10 PRINT USING "$$####.##  ";
12.35;123.555;555555.88#
20 END

run
   $12.35    $123.56  %$555555.88
Ok
```

"**$"

Specifying **$ at the beginning of the format string combines the effect of the dollar sign and asterisk. Numbers displayed are preceded by a dollar sign, and empty spaces to the left of the dollar sign are filled with asterisks. The symbols * *$ also represent three positions in the display field (one of which is used for display of the dollar sign).

Example 9

```
10 PRINT USING "**$####.##   ";12.35;
123.555;555555.88#
20 END

run
****$12.35  ***$123.56  $555555.88
Ok
```

","

Including a comma to the left of the decimal point in a format string causes commas to be displayed to the left of every third digit to the left of the decimal point. If the format string does not include a decimal point, include the comma at the end of the format string; in this case, numbers are rounded to the nearest integer value for display.

The comma represents the position of an additional position in the display field, and each comma displayed occupies one position.

Example 10

```
10 PRINT USING "#######,.##";555555.88#
20 END

run
 555,555.88
Ok
```

"^^^^"
Four carets (exponentiation operators) at the right end of the format string cause numbers to be displayed in exponential format. The four carets reserve space for display of E+XX.

The decimal point may also be included in the format string at any position desired. Significant digits are left-justified, and the exponent and fixed point constant are adjusted as necessary to allow the number to be displayed in the number of positions in the field.

Unless a leading + or trailing + or − sign is included in the format string, one digit position to the left of the decimal point will be used to display a + or − sign.

Example 11

```
10 PRINT USING "###.##^^^^  ";
   123.45; 12.345; 1234.5
20 END
```

```
run
 12.35E+01    12.35E+00    12.35E+02
Ok
```

"_"

An underscore mark in the format string causes the following character to be output as a literal together with the number.

Example 12

```
10 PRINT USING "###_%";123
20 END
```

```
run
123%
Ok
```

Other characters:
If characters other than those described above are placed at the beginning or end of a format string, those characters will be displayed in front of or behind the formatted number. Operation varies from case to case if other characters are included within the format string; however, in general including other characters in the string has the effect of dividing the string up into sections, with formatted numbers displayed in each section together with the delimiting character.

```
10 PRINT USING "##/##/##";12;34;56
20 PRINT USING "(###)";123
30 PRINT USING "<###>";123
40 END
```

```
Ok
run
12/34/56
(123)
<123>
Ok
```

NOTE:
The formatting characters shown above apply to the ASCII character set. If you select a character set other than ASCII with the Option Country statement some of the formatting characters will be output differently as shown below.

Hex.	Dec.	U.S.A	France	Germany	England	Denmark	Sweden	Italy	Spain	Norway
23H	35	#	#	#	£	#	#	#	₧	#
24H	36	$	$	$	$	$	¤	$	$	¤
5CH	92	\	ç	Ö	\	Ø	Ö	\	Ñ	Ø
5EH	94	^	^	^	^	Ü	Ü	^	^	Ü

Example 13

See listing under **OPTION COUNTRY.**

PRINT#/PRINT# USING

Format **PRINT# <file number>,[<list of expressions>]**

PRINT# <file number>, USING <format string>;<list of expressions>

Purpose These statements write data to a sequential output file.

Remarks The value of <file number> is the number under which the file was opened for output. The specification of <format string> is the same as that described in the explanation of the PRINT USING statement, and the expressions included in <list of expressions> are the numeric expressions which are to be written to the file.

Both of the formats above write values to the disk in display image format; that is, data is written to the disk in exactly the same format as it is displayed on the screen with the PRINT or PRINT USING statements. Therefore, care must be taken to ensure that data is properly delimited when it is written to the file (otherwise, it will not be input properly when the file is read later with the INPUT# or LINE INPUT# statements).

Numeric expressions included in <list of expressions> should be delimited with semicolons. If commas are used, the extra blanks that would be inserted between display fields by a PRINT statement will be written to the disk.

String expressions included in <list of expressions> must be delimited with semicolons; further, a string expression consisting of an explicit delimiter (a comma or carriage return code) should be included between each expression which is to be read back into a separate variable. The reason for this is that the INPUT# statement regards all characters preceding a comma or carriage return as one item. Explicit delimiters can be included using one of the following formats.

PRINT#1,<string expression>;",";<string expression>...

PRINT#1,<string expression>;CHR$(13);<string expression>...

If a string which is to be read back into a variable with the INPUT# statement includes commas, significant leading spaces or carriage returns, the corresponding expression in the PRINT# statement must be enclosed between explicit quotation marks CHR$(34). This is done as follows.

```
PRINT#1,CHR$(34);"SMITH, JOHN";CHR$(34);CHR$(34);

"SMITH, ROBERT";CHR$(34);...
```

This would actually be printed to the disk as

```
"SMITH, JOHN","SMITH, ROBERT".....
```

When the LINE INPUT# statement is to be used to read items of data back into variables, delimit string expressions in <list of expressions> with CHR$(13) (the carriage return code) as shown in the example above.

See also **INPUT#**, **LINE INPUT#**, **WRITE#**, and Chapter 6.

PSET

Format **PSET [STEP] (X,Y)[,<function code>]**

Purpose Sets (turns on) the dot at the specified graphic screen coordinates.

Remarks This statement sets the dot at the graphic screen coordinates specified by (X,Y). When STEP is specified, relative coordinates are used.

<function code> is a number from 0 to 7 which specifies whether the dot at the specified coordinates is set or reset. If 0 is specified, the PSET statement resets (turns off) the specified dot; if 1 to 7 is specified, the specified dot is set (turned on). If omitted, 7 is assumed.

After execution of the PSET statement, the LRP (last reference pointer) is updated to the values specified for (X,Y).

MO error (Missing operand) — A required operand was not specified in the statement.

FC error (Illegal function call) — The number specified in one of the statement operands was outside of the prescribed range.

OV error (Overflow) — The number specified in one of the statement operands was outside of the prescribed range.

PUT

Format **PUT[#]<file number>[,<record number>]**

Purpose Writes the contents of a random file buffer to one record of a random access file.

Remarks The random access file must be opened in the "R" mode under the number specified in <file number> before this statement can be executed. Further, data must be set in the random file buffer with the LSET and/or RSET statements.

<file number> is the number under which the file was opened, and <record number> is the number of the file record to which the buffer contents are to be written.

<record number> must have a value in the range from 1 to 32767. If <record number> is omitted, the contents of the random file buffer will be written to the record following the record accessed by the previous PUT or GET statement.

See also **GET, LSET/RSET, OPEN** and Chapter 6.

Example

```
10 OPEN "R",#1,"RANDOM",20
20 FIELD #1,10 AS A$
30 LSET A$="ABCDEFGHIJ"
40 PUT #1
50 WRITE#1,1,2,"YZa"
60 PUT #1
70 PRINT#1,"KLMNOPQ";
80 PUT #1
90 FOR I=1 TO 3
100 GET #1,I
110 PRINT A$
120 NEXT

run
ABCDEFGHIJ
1,2,"YZa"
KLMNOPQa"
Ok
```

NOTE:
String data to be written to a random access file with PUT can be placed in the random file buffer with the PRINT#, PRINT# USING, and WRITE# statements, as well as with the LSET/RSET statements. An example of this is INCLUDED IN THE PROGRAM ABOVE. When the WRITE # statement is used for this purpose, extra positions in the buffer are padded with spaces. An FO error (Field overflow) will occur if an attempt is made to read or write past the end of the buffer.

RANDOMIZE

Format **RANDOMIZE [<expression>]**

Purpose Reinitializes the sequence of random numbers generated by the RND function using the seed number specified in <expression>.

Remarks The value specified in <expression> must be a number in the range from −32768 to 32767. If <expression> is omitted, BASIC suspends program execution and displays the following message to prompt the operator to enter a value from the keyboard.

Random number seed ?

If the random number sequence is not reinitialized, the RND function will return the same sequence of random numbers each time a given program is executed. This can be overcome by placing a RANDOMIZE command at the beginning of each program so that the user can input a random number. It is better however, if the computer changes the random number seed in <expression>. This can be achieved by using the TIME$ function as this is a continually changing string of numbers. A description of this is given in the example program in RND.

See also **RND**

Example

```
10 FOR J=1 TO 3
20 RANDOMIZE
30 PRINT
40 FOR I=1 TO 5
50 PRINT RND;
60 NEXT I
70 PRINT
80 NEXT J
```

```
run
Random number seed (-32768 to 32767)? 1
 .58041  .128928  .928324  .901162  .532818
Random number seed (-32768 to 32767)? 2
 .89341  .823736  .964563  .674916  .963391
Random number seed (-32768 to 32767)? 1
 .826124  .915422  .0593067  .381003  .511101
Ok
```

READ

Format **READ <list of variables>**

Purpose Reads values from DATA statements and assigns them to variables.

Remarks The READ statement must always be used in conjunction with one or more DATA statements. The READ statement assigns items from the <list of constants> of DATA statements to variables specified in <list of variables>. Items from the <list of constants> are substituted into variables in the <list of variables> on a one-to-one basis, and the type of each variable to which data is assigned must be the same as the type of the corresponding constant in <list of constants>.

A single READ statement may access one or more DATA statements, or several READ statements may access the same DATA statement.

If the number of variables specified in <list of variables> is greater than the number of constants specified by DATA statements, an OD error (Out of data) will occur. If the number of variables specified in <list of variables> is smaller than the number of constants specified in DATA statements, subsequent READ statements begin reading data at the first item which has not previously been read. If there are no subsequent READ statements, the extra items are ignored.

The next item to be read by a READ statement can be reset to the beginning of the first DATA statement on any specified line by means of the RESTORE statement.

See also **DATA, RESTORE**

Example

```
10 FOR J=1 TO 5
20 READ A(J),B$(J),C(J)
30 NEXT J
40 FOR J=1 TO 5
50 PRINT A(J),B$(J),C(J)
60 NEXT J
70 END
80 DATA 1,aaaa,11
90 DATA 2,bbbb,22
100 DATA 3,cccc,33
110 DATA 4,dddd,44
120 DATA 5,eeee,55

run
 1             aaaa          11
 2             bbbb          22
 3             cccc          33
 4             dddd          44
 5             eeee          55
Ok
```

REM

Format **REM <remark>**
' <remark>

Purpose Makes it possible to insert explanatory remarks into programs.

Remarks Either of the above formats may be used to insert explanatory remarks into a program. Remark statements are ignored by BASIC during program execution, but are output exactly as entered when the program is listed.

If program execution is branched to a line which begins with a remark statement, execution resumes with the first subsequent line which contains an executable statement.

If a remark statement is to be appended to a line which includes an executable statement, be sure to precede it with a colon (:). Also, note that any executable statements following a remark statement on a given program line will be ignored.

NOTE:
When a program is listed using LIST or LLIST*, apostrophes indicating remark statements are not output. However, "REM" is output at the beginning of any remark statements beginning with REM.*

REMOVE

Format **REMOVE**

Purpose Writes the directory to the microcassette tape and terminates microcassette I/O.

Remarks A REMOVE command must be executed before taking a cassette tape out of the microcassette drive. The reason for this is that, if the cassette tape is taken out of the drive without executing the REMOVE command, the directory is not updated and data which has been written to the cassette up to that point may be lost. Further, if another tape is inserted in the microcassette drive without executing the REMOVE command for the previous tape, the contents of the new tape may be destroyed.

The REMOVE command cannot be executed if any files are open or the tape in the drive has not been mounted.

DW error (Disk write error) — An error occurred while data was being written to the microcassette drive.

AC error (Tape access error) — The REMOVE command was executed before the tape in the drive had been installed by executing the MOUNT command; or, the REMOVE command was executed while a tape file was still open.

See also **MOUNT**
Section 2.4.5

RENUM

Format **RENUM[[<new line number>][,[<old line number>] [,<increment>]]]**

Purpose Renumbers the lines of programs.

Remarks This command renumbers the lines of a program according to the values specified in <new line number>, <old line number>, and <increment>. <new line number> is the first line number to be used in the new sequence of program lines and <old line number> is the line number in the current program with which renumbering is to begin. <increment> is the amount by which each successive line number in the new sequence is to be increased over the number of the preceding line.

The default value for <new line number> is 10, that for <old line number> is the first line of the current program, and that for <increment> is 10.

The RENUM command also changes all line number references included in GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the message "UL error in xxxxx" is displayed. The incorrect line number in the program line is not changed, but that indicated by xxxxx may be changed.

Examples **RENUM**
Renumbers the entire program. The first line number of the new sequence will be 10, and the numbers of subsequent lines will be increased in increments of 10.

RENUM 300,50
Renumbers program lines starting with existing line 50. The number of line number 50 in the current program will be changed to 300, and all subsequent lines will be increased in increments of 10.

RENUM 1000,900,20
Renumbers the lines beginning with 900 so that they start with line number 1000 and are increased in increments of 20.

NOTE:
The RENUM command 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 FC error (Illegal function call) will result if this rule is not observed.

RESET

Format RESET

Purpose Resets the READ/ONLY condition which results when the floppy disk in a disk drive has been exchanged for another one. When the floppy disk in an external disk drive is replaced with another one when that drive has previously been accessed, subsequent writes to that drive are inhibited. This is to protect the contents of the disk's directory. Executing the RESET command resets the read-only condition and re-enables access to the new disk. It also closes any files which are open in the same manner as the CLOSE command.

Also RESET enables a new ROM capsule for read access after replacement.

The default drive is the default drive for CP/M until a RESET command is executed. The execution of a RESET command sets the default drive to drive A: and so programs should specify the drive to which the data is to be saved.

Example

```
100 CLOSE
110 PRINT "Replace disk in drive E: and press
    enter when ready"
120 A$=INPUT$(1)
130 RESET
140 OPEN"O",#1,"E:FILE1"
```

RESTORE

Format **RESTORE [<line number>]**

Purpose Allows DATA statements to be re-read from a specified program line.

Remarks If <line number> is specified, the next READ statement will access the first item in the DATA statement on the specified line or on the first subsequent line which contains a DATA statement if there is no DATA statement in the specified line. If <line number> is not specified, the next READ statement accesses the first item in the first DATA statement in the program.

See also **DATA, READ**

Example

```
10 FOR I=1 TO 2
20 FOR X=1 TO 8
30 READ A:SOUND A,50
40 NEXT X,I
50 SOUND 0,100:RESTORE
60 FOR I=1 TO 2
70 FOR X=1 TO 8
80 READ A:SOUND A*2,50
90 NEXT X,I
100 DATA 256,288,320,341,384,426,480,512
110 DATA 512,480,426,384,341,320,288,256
```

RESUME

Format

RESUME
RESUME Ø
RESUME NEXT
RESUME <line number>

Purpose

Used to continue program execution after execution has branched to an error processing routine.

Remarks

The RESUME statement makes it possible to resume program execution at a specific line or statement after error recovery processing has been completed. The point at which execution is resumed is determined by the format in which the statement is executed as follows:

RESUME or **RESUME Ø**	Resumes program execution at the statement which caused the error.
RESUME NEXT	Resumes program execution at the statement immediately following that which caused the error.
RESUME <line number>	Resumes program execution at the program line specified in <line number>.

An RW error (RESUME without error) message will be generated if a RESUME statement is encountered anywhere in a program except in an error processing routine.

See also

ERROR, ON ERROR GOTO, ERR/ERL

Example

See the example program under ERROR.

RIGHT$

Format **RIGHT$(X$,J)**

Purpose Returns a string composed of the J characters making up the right hand end of string X$.

Remarks The value specified for J must be in the range from 0 to 255. If J is greater than the length of X$ the entire string will be returned. If J is equal to 0 a null string of zero length is returned.

See also **LEFT$, MID$**

Example

```
10 A$="Epson PX-4"
20 FOR I=1 TO 10
30 PRINT RIGHT$(A$,I)
40 NEXT
run
4
-4
X-4
PX-4
 PX-4
n PX-4
on PX-4
son PX-4
pson PX-4
Epson PX-4
Ok
```

RND

Format RND[(X)]

Purpose Returns a random number with a value between 0 and 1.

Remarks RND returns a random number from a sequence determined mathematically by the random number generator in the BASIC interpreter. The same sequence of numbers is generated each time a program containing the RND function is executed. If X is omitted or the number specified for X is greater than 0, the next random number in the sequence is generated. If 0 is specified for X, RND repeats the last random number generated. If the number specified for X is less than 0, RND starts a new sequence whose initial value is determined by the value specified for X.
It is sometimes necessary to generate numbers in a given range. The following examples show how this may be done.

INT (RND (X) * 100)	Generates numbers in the range 0—99.
INT (RND (X) * 100) + 1	Generates numbers in the range 1—100.
INT (RND (X) * 100) + 50	Generates numbers in the range 100—149.
INT (RND (X) * 10) − 5	Generates numbers in the range −5 to +4.

See also RANDOMIZE

Example The following program shows how to use RANDOMIZE and the value returned by TIME$, to give a random number which is as random as the computer will allow. Run the program a number of times to see that the numbers output from line 30 are the same each time the program is run. The first value will be the same for the five circuits of the repeating loop (lines 30 to 50) because a negative value of X repeats the number by continually reseeding with the same number. The next values will be the same as these values because X = 0 which repeats the last random number. When X is from 1 to 3 a different number is produced each time in the loop lines 30 − 50, but this number will be the same each time the program is run.

The second set of numbers, generated by lines 100 – 130, is again the same every time the program is run, despite RANDOMIZE having been used.

The last set of numbers are different every time the program is run because the number used to generate the seed is different. It is obtained from the string returned by TIME$, by multiplying the hours by the seconds. This requires not only string manipulation to remove the characters from each end of the string, but also using the VAL function to convert them into numbers. More complex algorithms could be used.

```
10 FOR X = -1 TO 3
20 FOR N = 1 TO 5
30 PRINT RND(X);
40 NEXT N
45 PRINT
50 NEXT X
60 '
70 '
80 ' first randomize routine
90 RANDOMIZE(456)
100 FOR J = 1 TO 10
110 PRINT INT(RND(1)*1000);
120 NEXT
130 PRINT
140 '
150 ' second randomize routine
160 FOR J = 1 TO 10
170 RANDOMIZE(VAL(LEFT$(TIME$,2))*VAL(RIGHT$(TIME$,2)))
180 PRINT INT(RND(1)*1000);
190 NEXT
200 PRINT
```

```
 .308601   .308601   .308601   .308601   .308601
 .308601   .308601   .308601   .308601   .308601
 .498871   .670127   .98706   .739354   .783018
 .949844   .55241   .681371   .823571   .244878
 .421504   .775332   .310637   .346463   .056878
 422  292  906  614  667  833  595  910  875  173
 86  120  948  4  839  687  0  507  224  514
Ok
```

RUN

Format **RUN [<line number>]**
RUN <file descriptor>[,R]

Purpose Initiates program execution.

Remarks The first format is used to start execution of the program in the currently selected program area. Execution begins at the first line of the program unless <line number> is specified. If <line number> is specified, execution begins at the specified line. All files are closed and variables cleared, even if the <line number> specified is not the first line of the program. To restart a program from a particular line number without using RUN, use GOTO <line number>.

The second format is used to load and execute a program from a disk device, including the microcassette drive and RS-232C interface. Specify the name under which the program was saved in <file descriptor>; if the extension is omitted, ".BAS" is assumed. For the RS-232C interface, specify "COMØ:[(<options>)] as the file descriptor.

The RUN command normally closes all files which are open and deletes the current contents of memory before loading the specified program. However, all data files will remain open if the R option is specified although the variables will be cleared.

See also **GOTO, LOAD, MERGE**

Examples **RUN 3ØØ**
runs the program from line 300.

RUN "ADDRESS.BAS"
loads and runs the program "ADDRESS.BAS" in the default drive.

RUN "D:ENTRY.BAS",R
loads and runs the program "ENTRY.BAS" from disk drive D: without closing the files which were opened by the previous program.

SAVE

Format

SAVE <file descriptor>[,A]
SAVE <file descriptor>[,P]

Purpose

Used to save programs to disk device files or the RS-232C communications interface.

Remarks

This command saves BASIC programs to disk files or the RS-232C communications interface. In the former case, specify the drive name, file name and extension in <file descriptor>.
The currently active drive is assumed if the drive name is omitted, and ".BAS" is assumed if the extension is omitted. In the latter case, specify "COMØ:[(<options>)]" as the file descriptor.

If the A option is specified, the program will be saved in ASCII format; otherwise it will be saved in compressed binary format. The ASCII format requires more disk space for storage than binary format, but some file access operations require that the file be in ASCII format (for example, the file must be in ASCII format if it is to be loaded with the MERGE command).

If the P (protect) option is specified, the program will be saved in an encoded binary format. When a file is saved using this option it cannot be edited or listed when it is subsequently loaded. Once a program has been saved with the P option the protected condition cannot be cancelled.

See also

LOAD, MERGE

Examples

```
SAVE "ADDRESS"
SAVE "B:ADDRESS.ASC",A
SAVE "COMØ:(A8N3FXN)"
SAVE "SECRET",P
```

SCREEN

Format **SCREEN (<horizontal position>,<vertical position>)**

Purpose Returns the character code of the character displayed at the specified position on the screen.

Remarks The SCREEN function returns the value of character code for the character displayed at the virtual screen coordinates specified by <horizontal position> and <vertical position>. <horizontal position> must be specified as a numerical expression with a value in the range from 1 to Xmax; <vertical position> is also specified as a numerical expression, but with a value in the range from 1 to Ymax. The values of Xmax and Ymax are determined by the size of the screen currently being used as the write screen.

FC error (Illegal function call) — An argument specified was not in the prescribed range.

SGN

Format SGN(X)

Purpose Returns the sign of numeric expression X.

Remarks If X is greater than 0, SGN(X) returns 1. If X equals 0, SGN(X) returns 0. If X is less than 0, SGN(X) returns −1. Any numeric expression can be specified for X.

Example

```
10 A=1:  PRINT SGN(A)
20 B=1<0:PRINT SGN(B)
30 C=1=1:PRINT SGN(C)

run
 1
 0
-1
Ok
```

SIN

Format SIN(X)

Purpose Returns the sine of X, where X is an angle in radians.

Remarks The sine of angle X is calculated to the precision of the type of the numeric expression specified for X.

Example

```
10 CLS
20 INPUT "Enter angle in degrees";A
30 PI=4*ATN(1)
40 D=PI/180
50 PRINT "SIN(";A;")=";SIN(A*D)
60 GOTO 20
```

```
Enter angle in degrees? 0
SIN( 0 )= 0
Enter angle in degrees? 30
SIN( 30 )= .5
Enter angle in degrees? 45
SIN( 45 )= .707107
Enter angle in degrees?
```

SOUND

Format **SOUND <pitch>,<duration>**

Purpose Generates a sound of the specified pitch and duration.

Remarks The <pitch> parameter is specified as a value from 0 to 4500. Values from 100 to 4500 cause a sound to be generated at the equivalent pitch, and values from 0 to 99 result in no sound.

The <duration> parameter is specified as a value from 0 to 2554; the length of the sound generated is equal to <duration> × 10 msec, with the result rounded off to the nearest millisecond.

When a SOUND statement is executed, the BASIC interpreter waits for execution of that statement to be completed before going on to the next statement.

The relationship between values specified in <pitch> and the notes of the musical scale is as shown in the table below.

Unit: Hz

Note	1	2	3	4	5
C	130	261	523	1046	2093
C#	138	277	554	1108	2217
D	146	293	587	1174	2349
D#	155	311	622	1214	2489
E	164	329	659	1318	2637
F	174	349	698	1396	2793
F#	184	369	739	1479	2959
G	195	391	783	1567	3135
G#	207	415	830	1661	3322
A	220	440	880	1760	3520
A#	233	466	932	1864	3729
B	246	493	987	1975	3951

FC error (Illegal function call) — A parameter specified was outside the permissable range.

MO error (Missing operand) — A required operand was not specified in the statement.

SPACE$

Format **SPACE$(J)**

Purpose Returns a string of spaces whose length is determined by the value specified for J.

Remarks The value of J must be in the range from 0 to 255. If other than an integer expression is specified for J, it is rounded to the nearest integer.

See also **SPC**

Example

```
10 FOR J=1 TO 7
20 READ A$,B$
30 P$=A$+SPACE$(20-LEN(A$+B$))+B$
40 PRINT P$
50 NEXT
60 DATA Angie,523-2121,Alfie,456-1010,Robert,21-4444
70 DATA Susan,223-1234,Charlie,234-2324,John,703-7654
80 DATA Randolph,631-1360

run
Angie         523-2121
Alfie         456-1010
Robert         21-4444
Susan         223-1234
Charlie       234-2324
John          703-7654
Randolph      631-1360
Ok
```

SPC

Format **SPC(J)**

Purpose Returns a string of spaces for output to the display or printer.

Remarks The SPC function can only be used with an output statement such as PRINT or LPRINT — unlike the SPACE$ function, it cannot be used to assign spaces to variables. The value specified for J must be in the range from 0 to 255.

See also **SPACE$**

Example

```
10 FOR J=1 TO 7
20 READ A$,B$
30 PRINT A$;SPC(20-LEN(A$+B$));B$
40 NEXT
50 DATA Angie,523-2121,Alfie,456-1010,Robert,21-4444,Susan,2
23-1234,Charlie,234-2324,John,703-7654,Randolph,631-1360

run
Angie       523-2121
Alfie       456-1010
Robert       21-4444
Susan       223-1234
Charlie     234-2324
John        703-7654
Randolph    631-1360
```

SQR

Format SQR(X)

Purpose Returns the square root of X.

Remarks The value specified for X must be greater than or equal to 0.

Example

```
10 PRINT "X","SQR(X)"
20 FOR X=0 TO 100 STEP 10
30 PRINT X,SQR(X)
40 NEXT
run
X                SQR(X)
 0                0
 10               3.16228
 20               4.47214
 30               5.47723
 40               6.32456
 50               7.07107
 60               7.74597
 70               8.3666
 80               8.94427
 90               9.48683
 100              10
Ok
```

STAT

Format

```
STAT [ | <program area no.> | ]
       | ALL                |
```

Purpose

Displays the status of program areas.

Remarks

Executing the STAT command without specifying any parameters displays the status of the currently selected program area. If <program area no.> is specified, the status of the specified program area is displayed. In both cases, the display format is as follows.

```
Pn:XXXXXXXX     YYYYY Bytes
```

Here, n indicates the number of the applicable program area, XXXXXXXX indicates the name assigned to that program area by the TITLE command, and YYYYY indicates the size of the program area (i.e., the size of the program in that area) in bytes. Spaces are displayed for XXXXXXXX if no name has been assigned with the TITLE command.

When an asterisk ("*") is displayed between Pn and the program area name, the edit inhibit attribute has been set for that area with the "TITLE ...,P" command.

Executing STAT ALL displays the status of all program areas as follows.

```
P1:AAAAAAAA    aaaaa  Bytes
P2:BBBBBBBB    bbbbb  Bytes
P3:CCCCCCCC    ccccc  Bytes
P4:DDDDDDDD    ddddd  Bytes
P5:EEEEEEEE    eeeee  Bytes
   xxxxx Bytes free
```

The number displayed for xxxxx indicates the current number of bytes of unused memory.

FC error (Illegal function call) — A number other than 1 to 5 was specified in <program area no.>.

STOP

Format **STOP**

Purpose Terminates program execution and returns BASIC to the command level.

Remarks STOP statements are generally used to interrupt program execution during debugging to allow the contents of variables to be examined or changed in the direct mode. Program execution can then be resumed by executing a CONT command.

The following message is displayed upon execution of a STOP statement:

Break in nnnnn

Unlike the END statement, no files are closed when a STOP statement is executed.

See also **CONT**

Example

```
10 PRINT "Program line 10"
20 STOP
30 PRINT "Program line 20"

run
Program line 10
Break in 20
Ok
cont
Program line 20
Ok
```

STOP KEY

Format

STOP KEY | **ON** / **OFF** |

Purpose

Disables or reenables the STOP key.

Remarks

Executing STOP KEY OFF disables the STOP key and CTRL + C . This prevents processing from being interrupted if the STOP key is accidentally pressed during execution of an application program.

Executing STOP KEY ON reenables the STOP key (and CTRL + C) function after it has been disabled by executing STOP KEY OFF.

Since the STOP key and CTRL / C are completely disabled when STOP KEY OFF is executed, be careful to avoid executing it during program debugging.

The STOP key is not disabled unless a STOP KEY OFF command is executed. Once executed, the STOP KEY OFF command is cancelled and the STOP key reenabled by a hot start or execution of a RUN, MENU, or STOP KEY ON command.

STR$

Format STR$(X)

Purpose Converts numeric data to string data.

Remarks This function returns a string of ASCII characters which represent the decimal number corresponding to the value of X. X must be a numeric expression.

This function is complementary to the VAL function.

See also VAL

Example

```
10 FOR X=1 TO 10
20 PRINT X;
30 NEXT
40 PRINT
50 FOR X=1 TO 10
60 A$=STR$(X)
70 PRINT A$;
80 NEXT

run
 1  2  3  4  5  6  7  8  9  10
 1 2 3 4 5 6 7 8 9 10
Ok
```

STRING$

Format

STRING$(J, K)
STRING$(J, X$)

Purpose

Returns a string of characters.

Remarks

The length of the character string returned by this function is determined by the value of J. If K is specified the function returns a string of J characters whose ASCII code corresponds to the value of K. If a non-integer value is specified for K its value is rounded to the nearest integer before the string of characters is returned.

If X$ is specified this function returns a string of J characters made up of the first character of the specified string.

Example

```
10 A$=STRING$(5,"A")
20 B$=STRING$(5,66)
30 PRINT A$:PRINT B$

run
AAAAA
BBBBB
Ok
```

SWAP

Format **SWAP <variable 1>,<variable 2>**

Purpose The SWAP statement exchanges the values of variables specified in <variable 1> and <variable 2>.

Remarks The SWAP statement may be used to exchange the values of any type of variable, but the same variable types must be specified in both <variable 1> and <variable 2>; otherwise a TM error (Type mismatch) will occur.

Example

```
10  Using SWAP for alphabetization
20 FOR J=1 TO 5
30 READ A$(J)
40 NEXT J
50 FOR J=2 TO 5
60 IF A$(J-1)>A$(J) THEN SWAP A$(J-1),A$(J):J=1
70 NEXT J
80 FOR J=1 TO 5
90 PRINT A$(J)
100 NEXT J
110 DATA Mary,Charlie,Angie,Jane,Andy

run
Andy
Angie
Charlie
Jane
Mary
Ok
```

SYSTEM

Format **SYSTEM**

Purpose The SYSTEM command returns control from BASIC to CP/M.

Remarks The SYSTEM command ends operation of the BASIC interpreter and returns control to CP/M. All programs in the BASIC program areas are lost when this command is executed.

TAB

Format

TAB(J)

Purpose

Spaces to column J on the LCD screen or printer. If the cursor/print head is already past column J, it is spaced to that column on the next line.

Remarks

The character position on the far left side of the LCD screen or printer is column 0, and that on the far right side is the device width minus one. For the LCD screen, the device width is the number of columns determined for the currently selected screen by the SCREEN or WIDTH statement. For a printer, it is the number of columns determined by the WIDTH LPRINT statement.

If the value specified for J is greater than the device width minus one, the number of spaces generated is equal to J MOD n, where n is the device width.

In the expression

PRINT TAB (J) ; A$

the string A$ will be printed with the first character starting at position J. However, if the length of string A$ added to the value of J is greater than 81, the string will be printed on the next line.

The TAB function can only be used with the PRINT and LPRINT statements, and cannot be used to generate strings of spaces for other purposes.

Example

```
10 SCREEN 0
20 PRINT 1;2;3;4;5
30 PRINT 1;TAB(4);2;TAB(9);3;TAB(15);4;TAB(22);5

run
 1  2  3  4  5
 1  2    3     4      5
Ok
```

NOTE:
If a space is included between TAB and the opening bracket in TAB (J), PX-4 BASIC will interpret this as item J of an array with the name TAB. Rather than print the next string at position J, the value 0 will be printed because the value of all the items in this array will be 0. If J is greater than 10, a BS error (Subscript out of range) will be generated because the TAB array has not been dimensioned.

TAN

Format **TAN(X)**

Purpose Returns the tangent of X, where X is an angle in radians.

Remarks The tangent of angle X is calculated to the precision of the type of numeric expression specified for X.

To convert an angle from degrees to radians, multiply it by 1.57080 (for single precision) or by 1.570796326794897/90 (for double precision).

See also **ATN, COS, SIN**

Example

```
10 INPUT "Enter angle in degrees";A
20 PRINT "Tangent";A;"degrees is";TAN(A*3.14159/180)
30 GOTO 10

Enter angle in degrees? 30
Tangent 30 degrees is .57735
Enter angle in degrees? 45
Tangent 45 degrees is .999999
Enter angle in degrees? 60
Tangent 60 degrees is 1.73205
Enter angle in degrees?
```

TAPCNT

Format **TAPCNT**

Purpose Reads or sets the value of the microcassette drive counter.

Remarks The TAPCNT function reads the value of the microcassette drive counter. The value returned will be in the range from −32768 to 32767.

The TAPCNT function can be used at any time it is necessary to determine the counter value.

The TAPCNT function can also be used to set the counter value. However, in this case, the tape in the microcassette drive must be in the unmounted condition.

AC error (Tape access error) — An attempt was made to set the counter value while the tape in the microcassette drive was in the mounted condition.

IO error (Device I/O error) — An attempt was made to read or set the counter value when microcassette drive is not installed.

See also **WIND**

TIME$

Format **TIME$**

Purpose Reads the time of the built-in clock.

Remarks The TIME$ function returns the time of the built-in clock as an 8-byte character string. The format of this string is "HH:MM:SS", where HH indicates the hour (00 to 23), MM indicates the minute (00 to 59), and SS indicates the second (00 to 59).

TIME$ is a system variable and can be set by executing TIME$ = "HH:MM:SS".

TITLE

Format **TITLE (<program area name>][,P]**

Purpose Sets the name and protect attribute of the currently selected program area.

Remarks The TITLE command assigns a name to the currently selected program area. This name is specified in <program area name> as a string of from 0 to 8 letters. If more than 8 letters are specified, the ninth and following letters are ignored. After execution of this command, the specified program area name is displayed upon execution of the STAT command and in the BASIC start-up menu. Further, when a program file is loaded by executing the LOAD or RUN <filename> commands, the file name of the program loaded is set as the program area name.

The program area name can be cancelled by executing the TITLE command with a null string ("") specified for <program area name>. The program area name is also cancelled by executing the NEW command.

The program area name is not affected if the <program area name> parameter is omitted.

If the P (protect) option is specified, executing the TITLE statement sets the protect attribute for the currently selected program area. Once a program area has been protected in this manner, any attempt to edit that program or to execute a DELETE command in that program area will result in an FC error (Illegal function call).

MO error (Missing operand) — A required operand was not specified in the command.

TRON/TROFF

Format

TRON
TROFF

Purpose

Used to enable or disable the trace mode of execution.

Remarks

In the trace mode, the number of each line of a program is displayed on the screen in square brackets at the time that line is executed. This makes it possible to determine the sequence in which program lines are executed, and as such can be used with the STOP and CONT commands during program debugging.

The trace mode is enabled by executing TRON and is disabled by executing TROFF.

See also

CONT, STOP

Example

```
10 FOR I=1 TO 3:PRINT I;:NEXT
20 PRINT
30 TRON
40 FOR I=4 TO 6:PRINT I;:NEXT
50 TROFF

run
 1  2  3
[40] 4  5  6 [50]
Ok
```

USR

Format USR [<digit>]<argument>

Purpose Passes the value specified for <argument> to a user-written machine language routine and returns the result of that routine.

Remarks <digit> is an integer from 0 to 9 which corresponds to the digit specified in the DEF USR statement for the machine language routine. If <digit> is omitted, USRØ is assumed.

A string or numeric expression must be specified for <argument>; this argument is passed to the machine language routine as described in Appendix G.

VAL

Format **VAL(X$)**

Purpose Converts a string composed of numeric ASCII characters into a numeric value.

Remarks This function returns the numeric value of a character string consisting of numeric characters. The first character of string X$ must be "+", "−", ".", "&" or a numeric character (a character whose ASCII code is in the range from 48 to 57); otherwise, this function returns 0.

Some examples of use of the VAL function are shown below.

(1) VAL(X$)

Returns the decimal number which corresponds to the string representation of that decimal number. X$ is composed of the characters "0" to "9" and may be preceded by "+", "−" or ".". Complementary to the STR$ function.

(2) VAL("&H"+X$)

Returns the decimal number which corresponds to the string representation of a hexadecimal number. X$ is composed of the characters "0" to "9" and "A" to "F". This is complementary to the HEX$ function.

(3) VAL("&O"+X$)

Returns the decimal number which corresponds to the string representation of an octal number. X$ is composed of the characters "0" to "7". This is complementary to the OCT$ function.

See also **HEX$, OCT$, STR$**

Example

```
10 INPUT "Type in a hexadecimal number ";A$
20 PRINT "The decimal value of &H";A$;" using VAL(";CHR$(34)
;"&H";CHR$(34);"+X$) is ";VAL("&H"+A$)
30 INPUT "Type in an octal number ";B$
40 PRINT "The decimal value of &O";B$;" using VAL(";CHR$(34)
;"&O";CHR$(34);"+X$) is ";VAL("&O"+B$)
run
Type in a hexadecimal number ? 4F
The decimal value of &H4F using VAL("&H"+X$) is  79
Type in an octal number ? 32
The decimal value of &O32 using VAL("&O"+X$) is  26
Ok
```

VARPTR

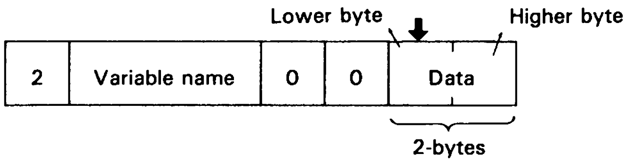
Format **VARPTR(<variable name>)**
VARPTR(# <file number>)

Purpose The first format returns the address in memory of the first data byte of the variable specified in <variable name>.
The second format returns the starting address of the I/O buffer assigned to the file opened under <file number>.

Remarks With the first format, a value must be assigned to <variable name> before executing VARPTR; otherwise, an "Illegal function call" error will result. Any type of variable name (numeric, string, or array) may be specified, and the address returned will be an integer in the range from −32768 to 32767. If a negative number is returned, add it to 65536 to obtain the actual address.

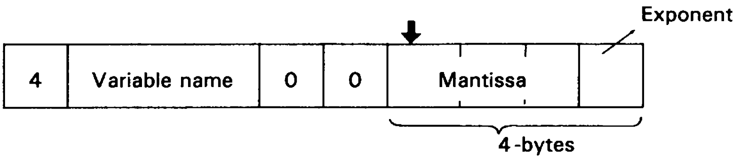
Storage of the various types of data in memory is as follows. (⬇ indicates the byte which corresponds to the value returned by VARPTR.)

(1) Integer variables
The data section of integer variables occupies two bytes in memory. Lower-order bits of this number are contained in the byte whose address is returned by the VARPTR function, and high-order bits are contained in the byte at the following address. Thus, if variable A% contains the integer 2, the address returned by the VARPTR function (the low-order byte) will contain 2, and that address plus 1 (the high-order byte) will contain 0.

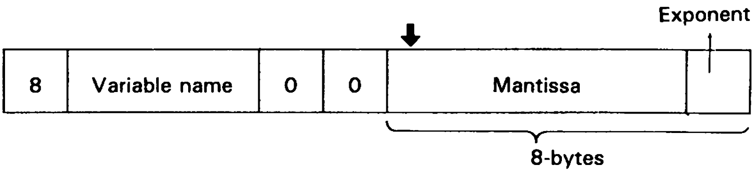


(2) Single precision variables
With single precision variables, numeric values are stored in two parts, using a total of four bytes of memory. The first part which is referred to as the exponent, and the remaining three bytes are referred to as the mantissa.

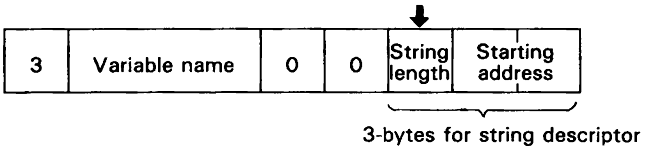
The VARPTR function returns the address of the least significant byte of the mantissa; the VARPTR address + 1 contains the middle byte of the mantissa; and the VARPTR address + 2 contains the most significant byte of the mantissa. The exponent is at the VARPTR address + 3.



(3) Double precision variables
The storage format for double precision values in variables is the same as with single precision variables. However, the mantissa portion of a double precision variable consists of seven bytes instead of three, so the data portion of a double precision variable occupies a total of eight bytes in memory.



(4) String variables
With string variables, the VARPTR function returns the length of the string. The low-order byte of the string's starting address in memory is indicated by the VARPTR address + 1, and the high-order byte of the string's starting address in memory is indicated by the VARPTR address + 2.



The second format returns the address of the first byte of the I/O buffer assigned to the file opened under <file number>.

This function is generally used to obtain the address of a variable prior to passing it to a machine language program. In the case of array variables, the format VARPTR(A(0)) is generally used so that the address returned is that of the lowest-numbered element of the array.

VARPTR is an abbreviation for VARiable PoinTeR.

Example

```
30 A$="abcdefghijklmnopqrstuvwxyz"
40 A=VARPTR(A$):PRINT "Address of variable A$ is ";A
50 B=PEEK(A+2)*&H100+PEEK(A+1)
60 PRINT "Address of string in variable A$ is";B
70 PRINT "String in variable A$ is ";:FOR I=0 TO 25:PRINT CH
R$(PEEK(B+I));
90 NEXT

run
Address of variable A$ is -29624
Address of string in variable A$ is 35638
String in variable A$ is abcdefghijklmnopqrstuvwxyz
Ok
```

NOTE:
The addresses of array variables change whenever a value is assigned to a new simple variable; therefore, all simple variable assignments should be made before calling VARPTR for an array.

WAIT

Format **WAIT <port number>, J[,K]**

Purpose Suspends program execution while monitoring the status of a machine input port.

Remarks The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern. The data read at the port is exclusive ORed with the value of integer expression K, then ANDed with J. If the result is zero, BASIC loops back and reads the port again. If the result is not 0, execution resumes with the next statement. If K is omitted, 0 is assumed.

NOTE:
Use of this statement requires in-depth knowledge of the PX-4 firmware, and using it incautionsly can result in loss of system control and other problems.

WHILE...WEND

Format

```
WHILE <expression>
      .
      .
[<loop statements>]
      .
      .
WEND
```

Purpose

Allows the series of instructions between WHILE and WEND to be repeated as long as the condition specified by <expression> is satisfied.

Remarks

This statement causes program execution to loop through the series of instructions between WHILE and WEND as long as the condition specified by <expression> is satisfied. <expression> is specified as any expression which has a truth value of 0 (false) or other than 0 (true). Thus numeric, logical or relational expressions may be used to specify the condition which controls looping.

As with FOR/NEXT loops, WHILE/WEND loops may be nested to any level. They may also be included within FOR/NEXT loops or vice versa. When loops are nested, the first WEND corresponds to WHILE of the innermost loop, the second WEND corresponds to WHILE of the next innermost loop, and so forth.

A WE error (WHILE without WEND) will occur if WHILE is encountered without a corresponding WEND, and a WH error (WEND without WHILE) will occur if WEND is encountered without a corresponding WHILE.

See also

FOR...NEXT

Example

```
10 INPUT"Enter arbitrary number";X
20 WHILE X^A<1E+06
30 PRINT STR$(X);"^";MID$(STR$(A),2,5);"=";MID$(STR$(X^A),2,
6)
40 A=A+1
50 WEND

run
Enter arbitrary number? 42.5
 42.5^0=1
 42.5^1=42.5
 42.5^2=1806.2
 42.5^3=76765.
Ok
```

WIDTH

Format **WIDTH <no. column>,<no. lines 1>**

Purpose Specifies the size of the virtual screens.

Remarks The number of columns in the virtual screen is determined by the value specified for <no. columns>. The number of lines is determined by the value specified for <no. lines>. The value specified for <no. columns> must be either 40 or 80; when 40 is specified for <no. columns>, the value specified for <no. lines> must be in the range from 8 to 50. When 80 is specified for <no. columns>, the value specified for <no. lines> must be in the range from 8 to 25.

The screen is cleared when this statement is executed.

MO error (Missing operand) — A required operand was not specified in the statement.

FC error (Illegal function call) — The number specified in one of the statement operands was outside of the prescribed range.

WIND

Format

```
WIND [ | <counter value> | ]
       | ON              |
       | OFF             |
```

Purpose

Controls forward or reverse movement of the microcassette tape and audio output to the speaker.

Remarks

When this command is executed without specifying any parameter, the tape is rewound to its beginning and the counter is reset to 0. When <counter value> is specified, the tape is wound in one direction or the other until the counter reaches the specified value. <counter value> must be specified as a numeric expression whose value lies in the range from −32768 to 32767.

Executing WIND ON places the microcassette drive in the PLAY mode and outputs the read signal to the speaker. After the drive has been placed in the PLAY mode in this manner, the BASIC interpreter goes on to execute any subsequent statements. Microcassette operation in the PLAY mode is terminated by executing WIND OFF.

The BEEP and SOUND statements cannot be executed while the microcassette drive is in the PLAY mode. Further, the WIND OFF statement is ignored if executed while the drive is not in the PLAY mode.

Note that the WIND statement cannot be executed if the tape in the drive has been installed by executing the MOUNT statement.

FC error (Illegal function call) — The value specified for <counter value> was outside the prescribed range.

OV error (Overflow) — The value specified for <counter value> was outside the prescribed range.

AC error (Tape access error) — The tape in the microcassette drive has been installed by executing the MOUNT statement.

IO error (Device I/O error) — Some problem has occurred with the microcassette drive.

WRITE

Format **WRITE[<list of expressions>]**

Purpose Displays data specified in <list of expressions> on the LCD screen.

Remarks If <list of expressions> is omitted, a blank line is output to the LCD screen. If <list of expressions> is included, the values of the expressions are displayed on the LCD screen.

Numeric and string expressions can both be included in <list of expressions>, but each expression must be separated from the one following it with a comma. Commas are displayed between each item included, and strings displayed are enclosed in quotation marks. After the last item has been output, the cursor is automatically advanced to the next line.

The WRITE statement displays numeric values using the same format as the PRINT statement; however, no spaces are output to the left or right of numbers displayed.

See also **PRINT**

WRITE

Format **WRITE # <file number>,<list of expressions>**

Purpose Used to write data to a sequential disk device file.

Remarks <file number> is the number under which the file was opened for output, and expressions included in <list of expressions> are the numeric and/or string expressions which are to be written to the file. Data is written to the file in the same format as it is output to the screen by the WRITE statement; that is, commas are inserted between individual items and strings are delimited with quotation marks.
Therefore, it is not necessary to specify explicit delimiters in <list of expressions>, as is the case with the PRINT # statement. The following illustrates the difference between use of the PRINT # and WRITE # statements (the statements indicated perform identical functions).

**PRINT # 1,CHR$(34);"SMITH,JOHN";CHR$(34);",";
CHR$(34);"SMITH, ROBERT";CHR$(34)**

WRITE # 1, "SMITH, JOHN","SMITH, ROBERT"

A carriage return/line feed sequence is written to the file following the last item in <list of expressions>.

See also **PRINT #** and **PRINT # USING**

Example

```
10 CLS
20 OPEN "O",#1,"A:DATA"
30 FOR I=1 TO 2
40 PRINT "Enter item";I:LINE INPUT A$(I)
50 NEXT I
60 WRITE#1,A$(1),A$(2)
70 CLOSE
80 OPEN"I",#1,"A:DATA"
90 INPUT#1,A$,B$
100 PRINT A$:PRINT B$
110 CLOSE
120 END
```

```
Enter item 1
SMITH, JOHN
Enter item 2
SMITH, ROBERT
SMITH, JOHN
SMITH, ROBERT
Ok
```

Chapter 4

FILES

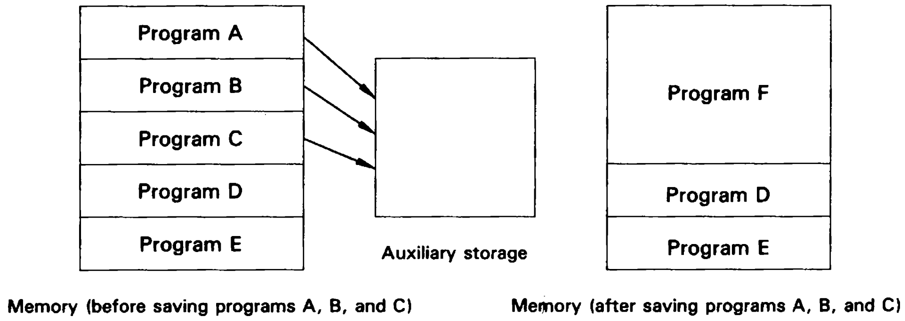
4.1 Program Files

You can load up to five programs in the PX-4 memory by dividing the user area into five.

If there are five programs already existing in memory, or if your new program is too large to fit in the space allotted to it, you may have to delete one of the existing programs to make room for the new one. Before doing this however, you must save on auxiliary storage (such as the RAM disk or microcassette tape.) any programs you want to use again.

The PX-4 supports auxiliary storage methods such as RAM disk, microcassette tape, RAM cartridges, floppy disks, and external cassette tape.

As shown below, you can create program F in the area occupied programs A, B, and C by saving these programs on auxiliary storage then deleting them with the NEW command.



To reexecute Program A, load it back into memory after saving one of the other programs.

Programs saved on auxiliary storage are called program files and treated as files with BASIC. A file is labelled by a file specification consisting of a drive name, a filename, and a file extension. A program file is also identified by its drive name, filename, and file extension. BASIC will use the logged-in drive for CP/M if the drive name is omitted, and automatically appends .BAS if the file extension is omitted. Here is a review of the commands and statements used for program file operations.

(i) SAVE "[<drive name>:]<filename>[.<file extension>]"[, $\left|\begin{matrix}\mathbf{A}\\ \mathbf{P}\end{matrix}\right|$ **]**

The SAVE command writes a program existing in the program area to the auxiliary storage device specified by <drive name>. The specified filename and file extension must then be assigned to the saved program file.

If the A option is included, the program is saved in the ASCII format; otherwise it is saved in binary format.

The ASCII format allows a program to be saved in the same form as it is displayed on the screen. In the binary format, the program is converted into an intermediate language and written onto a disk in a compressed form. This method of program storage uses less memory space; however, any programs which are to be merged must be saved in the ASCII format.

If the P option is specified, the program is saved in a protected form. This means that program file cannot be edited or listed, and once the option is specified, the protection cannot be cancelled.

(ii) LOAD "[<drive name>:]<filename> [.<file extension>]"[,R]

The LOAD command loads the program from the auxiliary storage device specified by <drive name> to the current program area. Specify the filename and file extension of the program file to be loaded. Add the R option if the program is to be executed as soon as it is loaded. All open files are kept open if this option is specified. Thus, a LOAD command issued with the R option may be used to chain the execution of programs which use the same data file. Executing this command without the R option closes all data files which are currently open.

Example

```
SAVE "A:TEST.BAS",P
Ok
LOAD "A:TEST.BAS"
Ok
LIST
FC Error
Ok
```

(iii) RUN "[<drive name>:]<filename> [.<file extension>]"[,R]

If the file specification is omitted, the RUN command executes the program in the current program area. Using the file specification causes the named program to be loaded into the current program area and to be executed. This operation deletes the contents of any existing loaded program. If the R option is included, all open data files are kept open; otherwise they are closed.

Example

```
5 ' RUN2
10 A$="RUN COMMAND TEST"
20 PRINT#1,A$
30 PRINT A$
40 CLOSE

SAVE "A:RUN2"
Ok
NEW
Ok

5 ' RUN1
10 OPEN "O",#1,"TEST.DAT"
20 RUN "RUN2.BAS",R

RUN
RUN COMMAND TEST
Ok
```

(iv) MERGE "[<drive name>]<filename> [.<file extension>]"

The MERGE command merges a program in the current program area with a program on the auxiliary storage device specified by <drive name>. Once <drive name> is specified, add the filename and file extension of the program file to the file specification. The program files to be merged must have been saved in the ASCII format.

If the two programs use some of same line numbers, the lines of the program in memory (the current program area) are replaced by those from the program on the auxiliary storage device.

BASIC always returns to the command level after executing the MERGE command.

Example

```
5 ' MERGE1
10 DIM B(10)
20 FOR I=1 TO 10
30  READ A : PRINT A;
40  B(I)=A
50 NEXT : PRINT
60 DATA 72,61,43,100,86,37,56,55,65,45

SAVE "A:MERGE1",A
Ok
```

```
NEW
Ok

100   merge2
110 S=0
120 FOR I=1 TO 10
130    S=S+B(I)
140 NEXT
150 PRINT "SUMMING = ";S
160 PRINT "AVERAGE = ";S/10
170 END

MERGE "A:MERGE1"
Ok
RUN

 72  61  43  100  86  37  56  55  65  45
SUMMING =  620
AVERAGE =  62
Ok


LIST
5 'MERGE1
10 DIM B(10)
20 FOR I=1 TO 10
30    READ A : PRINT A;
40    B(I)=A
50 NEXT : PRINT
60 DATA 72,61,43,100,86,37,56,55,65,45
100 ' merge2
110 S=0
120 FOR I=1 TO 10
130    S=S+B(I)
140 NEXT
150 PRINT "SUMMING = ";S
160 PRINT "AVERAGE = ";S/10
170 END
```

(v) KILL "[<drive name>:]<filename> [.<file extension>]"

The KILL command deletes a file from the auxiliary storage device named in the file specification. Care must be taken when using this command because it will delete any file having the specified name regardless of whether it is a system, program, or data file.

(vi) NAME <old file specification> AS <new file specification>

The NAME command changes the name of a file on the auxiliary storage device. This command, as with the KILL command, can be used for both program and data files.

Specify the existing filename for <old file specification> and the new filename to be assigned for <new file specification>. Both filenames must designate the same drive.

Example

```
FILES
STOCKLST.BAS  ACCOUNT .BAS  DEMO    .BAS
STOCK   .DAT  CHART   .DAT  TEST1   .DAT
ITEM    .DAT  SALES1  .BAS  GRAPH   .BAS
Ok
```

```
KILL "DEMO.BAS"
Ok
FILES
STOCKLST.BAS  ACCOUNT .BAS  STOCK   .DAT
CHART   .DAT  TEST1   .DAT  ITEM    .DAT
SALES1  .BAS  GRAPH   .BAS
Ok
```

```
NAME "SALES1.BAS" AS "RESULT.BAS"
Ok
FILES
STOCKLST.BAS  ACCOUNT .BAS  STOCK   .DAT
CHART   .DAT  TEST1   .DAT  ITEM    .DAT
RESULT  .BAS  GRAPH   .BAS
Ok
```

"FE Error" will be displayed if the new filename specified already exists.

4.2 Sequential Date Files

The sequential data file contains data that can only be read in the sequence that it is written. You cannot update part of a sequential data file once it has been created. However, sequential files are easy to create, requiring no special consideration about the length or structure of the data to be entered.

4.2.1 Creating sequential data files

The steps involved in creating a sequential data file are as follows:

(1) Execute an OPEN statement using the "O" mode. The file number specified in this statement will be assigned to the output file.

Example:
```
OPEN "O",#1,"A:OUTDT.DAT"
```

This example creates a data file named "OUTDT.DAT" on the RAM disk and assigns file number 1 to that file.

(2) Write data into the file using the PRINT# or WRITE# statement.

Example:
```
PRINT #1,A$;",";B$;",";C$
WRITE #1,A$,B$,C$
```

Both examples write the contents of A$, B$, and C$ into the file.

Contents of A$	Contents of B$	Contents of C$

(3) End the write operation with a CLOSE statement when all the data has been written.

Example:
```
CLOSE #1
```

This statement closes file number 1.

4.2.2 Reading sequential data files

The steps for reading a sequential data file are as follows:

(1) Execute an OPEN statement in the "I" mode. The file number specified in this statement will be assigned as an input file, in a similar manner to the output file.

Example:
```
OPEN "I",#1,"A:OUTDT.DAT"
```

This statement declares that BASIC will find the file named "OUTDT.DAT" and start reading the file.

(2) Read data from the file into variables in memory by executing either an INPUT# or LINE INPUT# statement.

Example:
```
INPUT #1,A$,B$,C$
LINE INPUT #1,A$
```

Both examples assign the data items in the file to the variables A$, B$, and C$ in the same order as they are written.

(3) End the read operation by executing a CLOSE statement after all the data has been read.

Example:
```
CLOSE #1
```

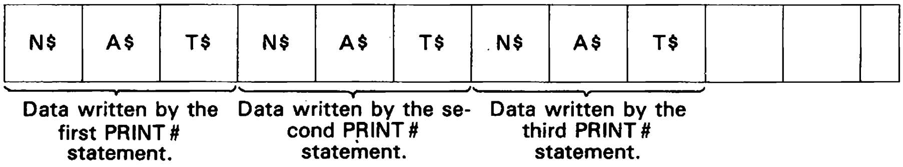
This example closes file number 1.

4.2.3 Sample programs

```
5 'SEQUENTIAL FILE1
10 OPEN "O",#1,"A:ADRSBOOK.DAT"
20 INPUT "NAME    :";N$
30 IF N$="*" THEN 80
40 INPUT "ADDRESS :";A$
50 INPUT "PHONE   :";P$
60 PRINT #1,N$;",";A$;",";P$
70 PRINT : GOTO 20
80 CLOSE
90 END
```

```
NAME    :JOHN......
ADDRESS :LONDON....
PHONE   :01-000-0000

NAME    :SALLY.....
ADDRESS :NY........
PHONE   :02-0000-0000

NAME    :*
Ok
```

The above program writes the data form the INPUT statement into a sequential data file named "ADRSBOOK.DAT" on the RAM disk. Type in a name, address, and phone number after "Name:?", "Address:?", and "Phone:?", respectively. The data written into the file, as shown in the figure below, each time the PRINT# statement on line 60 is executed. Typing "*" in response to the "Name:?" message closes the data file and terminates the program.


N$
A$
T$
N$
A$
T$
N$
A$
T$
Data written by the first PRINT# statement.
Data written by the second PRINT# statement.
Data written by the third PRINT# statement.

You may use the PRINT#USING or WRITE# statement instead of the PRINT# statement.

The next program reads the data file "ADRSBOOK.DAT" created in the previous program and displays it on the screen. The program checks for an EOF (indicating the end of file) at line 20. On detecting an EOF, it jumps to line 90 and closes the file since there is no data left to be read. If no EOF is detected, the program goes on to line 30 where it reads the data, and displays it on lines 40 to 60. An "IE Error" message will appear if an attempt is made to read further data after an EOF has been detected.

The program executes the statements on lines 20 to 50 repeatedly until it encounters an EOF.

```
5  SEQUENTIAL FILE2
10 OPEN "I",#1,"A:ADRSBOOK.DAT"
20 IF EOF(1) THEN 90
30 INPUT #1,N$,A$,P$
40 PRINT "NAME    :";N$
50 PRINT "ADDRESS :";A$
60 PRINT "PHONE   :";P$
70 INPUT "READ NEXT DATA (Y/N)";Y$
80 IF Y$="Y" OR Y$="y" THEN 20
90 CLOSE
100 END
```

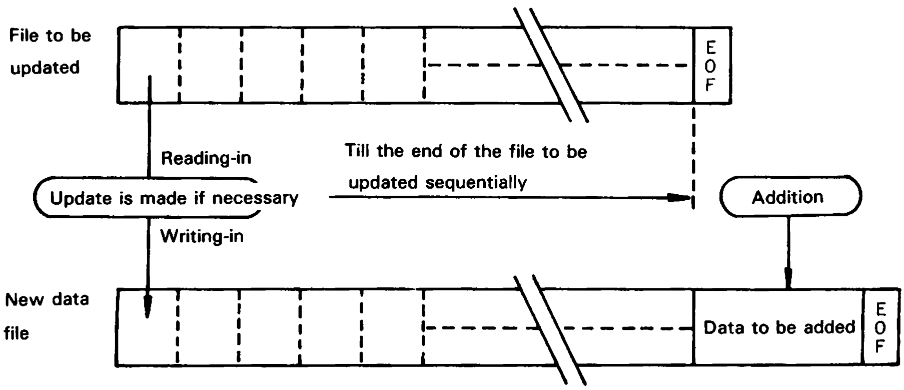
```
NAME    :JOHN......
ADDRESS :LONDON....
PHONE   :01-000-0000
READ NEXT DATA (Y/N)? Y
NAME    :SALLY.....
ADDRESS :NY........
PHONE   :02-0000-0000
READ NEXT DATA (Y/N)? Y
Ok
```

4.2.4 Updating sequential data files

After a sequential file has been created, it its not possible to update data in that file once it has been closed. The contents of a sequential file are destroyed whenever that file is opened in the "O" mode. To overcome this, the following procedure can be used:

(1) Open the data file to be updated in the "I" mode.
(2) Open a second data file in the "O" mode.
(3) Read in data from the original data file and write it to the new file, making necessary updates. .... (Updating of data.)
(4) After all the data included in the original data file has been written to the second data file, delete the original data file with the KILL command.
(5) Write the data to be added to the second data file... (Addition of data.)
(6) Close the second data file after all the data to be added has been written. Using the NAME command, rename the second data file with the name previously assigned to the original data file.

The result is a new data file which has the same file name as the original file, and which includes both the original data and the new data.



```
5 'SEQUENTIAL FILE3
100 OPEN "I",#1,"A:ADRSBOOK.DAT"
105 'Opens old file in input mode
110 OPEN "O",#2,"A:WORK.DAT"
115 'Opens temporary file in output mode
120 IF EOF(1) THEN 270
130 INPUT #1,N$,A$,P$:'Reads old data
140 PRINT "NAME    :";N$
150 PRINT "ADDRESS :";A$
160 PRINT "PHONE   :";P$
170 INPUT "  <CORRECT DATA (Y/N)>";Y$
180 IF Y$="N" OR Y$="n" THEN 250
190 INPUT "NAME    :";NX$
200 IF NX$="" THEN 210 ELSE N$=NX$
210 INPUT "ADDRESS :";AX$
220 IF AX$="" THEN 230 ELSE A$=AX$
230 INPUT "PHONE   :";PX$
240 IF PX$="" THEN 250 ELSE P$=PX$
250 PRINT #2,N$;",";A$;",";P$
260 PRINT : GOTO 120
270 INPUT "  ** ADD MORE DATA (Y/N) **";Y$
280 IF Y$="N" OR Y$="n" THEN 340
290 INPUT "NAME    :";N$
300 INPUT "ADDRESS :";A$
310 INPUT "PHONE   :";P$
320 PRINT #2,N$;",";A$;",";P$
330 GOTO 270
340 CLOSE
350 KILL "A:ADRSBOOK.DAT":'Erases old file
360 NAME "A:WORK.DAT" AS "A:ADRSBOOK.DAT":'Changes filename
370 END


run
NAME    :JOHN......
ADDRESS :LONDON....
PHONE   :01-000-0000
  <CORRECT DATA (Y/N)>? Y
NAME    :? BEN.....
ADDRESS :?
PHONE   :?


NAME    :SALLY.....
ADDRESS :NY........
PHONE   :02-0000-0000
  <CORRECT DATA (Y/N)>? N

  <CORRECT DATA (Y/N)>? Y
NAME    :? TIMOTHY....
ADDRESS :? BERLIN.....
PHONE   :? 000-0000

  ** ADD MORE DATA (Y/N) **? N
Ok
```

4.3 Random Data File

The random data file allows data to be accessed anywhere on a disk. Although sequential data files may have different lengths, a fixed length must be set for a random data files.

Random data files are more useful than sequential data files when there are large quantities of data which must be frequently updated. They require less disk space for storage because data is recorded using a packed binary format, whereas sequential files are written as series of ASCII characters. In sequential data files, it is possible to read or write only the parts of data that need to be updated; there is no need to read or write data in sequence as is the case with sequential files. The unit of data handled in a single random access read or write operation is called a record. You can identify records in a random data file by assigning them record numbers.

4.3.1 Creating random data files

The steps required to create a random data file are as follows:

(1) Execute an OPEN statement in the "R" mode (random mode).

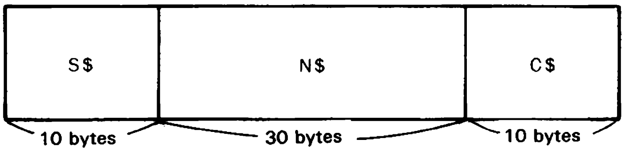
The OPEN statement assigns a file number to the file and defines the record length. If the record length is omitted, records of 128 bytes are assumed.

Example: `OPEN "R",#1,"A:STOCKLST.DAT",50`

The record length is set to 50 bytes in the above example.

(2) Specify the variables to be used for reading or writing data (buffer variables) and allocate space for them in the random file buffer using the FIELD statement.

Example: `FIELD #1,10 AS S$,30 AS N$,10 AS C$`



Note that the total equals the record length declared in the OPEN statement.

(3) Move the data to be written into the buffer variables using the LSET or RSET statement. Any numeric variables must be converted to character strings by using the MKI$, MKS$, or MKD$ function before executing the LSET or RSET statement.

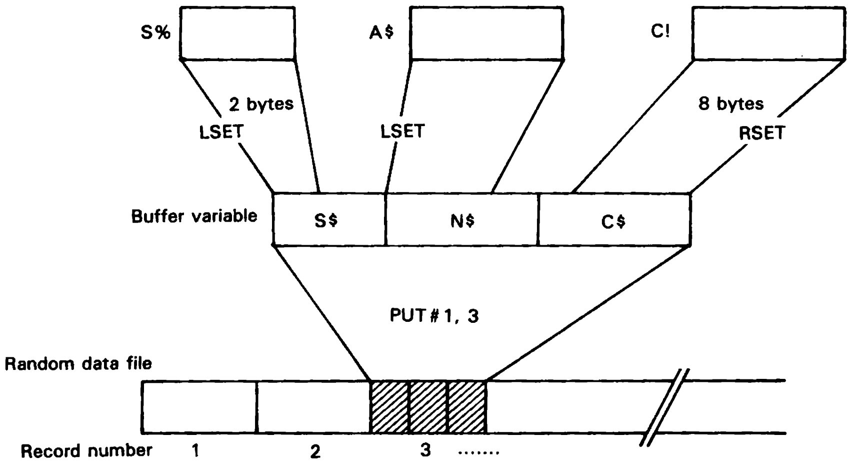
Example:

`LSET S$=MKI$(S%)`	Converts the contents of a integer variable into a character string and load it into buffer variable S$, left-justified.
`LSET N$=A$`	Loads the contents of a character variable into buffer variable N$, left-justified.
`RSET C$=MKS$(C!)`	Converts the contents of a single-precision variable into a character string and loads it into buffer variable C$, right-justified.

(4) Write records into the random data file using the PUT statement.

Example: `PUT #1,S%`

The record number indicating the position of a record in the file is placed in variable S%. When S% is specified as 3, the record is written into record position 3 of the file.



```
5 'RANDOM FILE1
100 OPEN "R",#1,"A:STOCKLST.DAT",40
110 FIELD #1,2 AS S$,30 AS N$,4 AS C$
120 PRINT
130 INPUT "STOCK NO.  (0 : End) ";S%
140 IF S%=0 THEN 220
150 INPUT "ITEM NAME ";A$
160 INPUT "PRICE     ";C!
170 LSET S$=MKI$(S%)
180 LSET N$=A$
190 C$=MKS$(C!)
200 PUT #1,S%
210 GOTO 120
220 CLOSE
230 END
```

The above program writes data entered from the keyboard into a random access file. Line 110 specifies buffer variables S$, N$, and C$, and their sizes. Lines 130, 150, and 160 allow data to be entered from the keyboard for storage in the random access file. In this example, STOCK NO. is also used as the record number. Lines 170, 180, and 190 load the input data into the buffer variables, and line 200 writes the record into the file.

NOTE:
Once a buffer variable is specified in a FIELD statement, do not use that variable on the left-hand side of an INPUT or LET statement. If you do, the specification for the variable in the FIELD statement will be canceled; the variable will be treated as an ordinary variable.

4.3.2 Reading random data files

The following steps are required to retrieve data from a random data file:

①Execute an OPEN statement in the "R" mode (random mode).

Example: `OPEN "R",#1,"A:STOCKLST.DAT",50`

②Specify the buffer variables and their sizes using the FIELD statement.

Example: `FIELD #1,10 AS S$,30 AS N$,10 AS C$`

③Read records using the GET statement.

Example: `GET #1,S%`

④Since numeric values are converted into binary format character strings when they are placed in the random data files, they must be converted back into the original format before reuse by the program. This is done using the CV1, CVS, or CVD functions.

Example: `PRINT CVI(S$),N$,CVS(C$)`

The following sample program reads the random file created in section 5.3.1. The number of records to be read is entered from the keyboard at line 130. Line 150 reads the record specified by the record number, and lines 160 to 180 display the contents of the record.

```
5 'RANDOM FILE2
100 OPEN "R",#1,"A:STOCKLST.DAT",40
110 FIELD #1,2 AS S$,30 AS N$,4 AS C$
120 PRINT
130 INPUT "STOCK NO. (0 : End) ";S%
140 IF S%=0 THEN 200
150 GET #1,S%
160 PRINT USING "####";CVI(S$)
170 PRINT N$
180 PRINT USING "####.##";CVS(C$)
190 GOTO 120
200 CLOSE
210 END
```

While a file is open, data can be read or written until the file is closed. The LOC function is useful to control the flow of program execution according to the total number of records which have been written to the file. With random files, the LOC function returns the record number last written to or read from the file.

The following statement ends program execution if the record number is greater than 50:

Example:

```
IF LOC(1)>50 THEN END
```

Chapter 5

DATA COMMUNICATIONS SUPPORT

5.1 PX-4 Communications Support

The PX-4 computers come as standard with an RS-232C interface, a serial (SIO) interface, and a cartridge serial interface. These interfaces are treated as communications devices through which the PX-4 communicate with external peripheral equipment. The devices allow serial input/output and are assigned the following drive names:

COM0:	For RS-232C input/output
COM1:	For SIO input/output
COM2:	For RS-232C input or SIO output
COM3:	For Cartridge serial input/output

NOTE:
Only one serial device can be opened at a time.

The cartridge serial and RS-232C interfaces can be handled in the same way using the same commands except for the communications procedures. The differences in communications protocol are due to the presense or absence of control signals for the respective interfaces. The control signals available for the above interfaces are listed below.

□ Control signals □

RS-232C	SIO	Cartridge serial
DSR	SIN	None
DTR	SOUT	None
CTS	None	None
RTS	None	None
DCD	None	None

NOTE:
For descriptions of the control signals for the individual interfaces, refer to Chapter 3, "STANDARD INPUT/OUTPUT INTERFACE" of the Operating Manual.

When connecting an external device other than those available for the PX-4 to the SIO or cartridge serial interface, it is necessary to select the proper cable considering the pin assignments and communications procedures.

5.2 RS-232C Interface

The RS-232C interface is an interface whose electrical characteristics or specifications are stipulated by standard known as the RS-232C standards. Presently, the use of each signal line and communications protocol for this interface differ from equipment to equipment. When connecting an acoustic coupler or another computer to the PX-4, therefore, the user must carefully check the electrical specifications of the external device and match the conditions for connecting these devices. The PX-4 permits you to set up these conditions from BASIC. You can set up the similar conditions for the SIO and cartridge serial interfaces except for some control signals.

NOTE:
Refer to the Operating Manual for the cables for connecting between the PX-4 and external devices.

5.2.1 Opening the RS-232C interface

The RS-232C communications interface is opened for data transmission and reception by executing the OPEN statement.

(1) Format
OPEN <mode>,<file number>,"COM0:[(<options>)]"

mode:
Specify "I" when the file is to be opened for input (receive) and "O" when it is to be opened for output (transmit). When using the file for both input and output, open it using two OPEN statements with different file numbers as if it were two separate files. In this case, BASIC ignores any options specified in the latter OPEN statement and takes the ones specified in the first OPEN statement.

file number:
You can specify any number from 1 to 15 smaller than the number specified in the /F: parameter of the BASIC command. Thereafter the file is identified by this number in the program.

drive name:
Specify one of the communications device names from COM0: to COM3:. Specify COM0: when using the RS-232C interface.

options:
Specifies the data communication protocol and control options. Specify options with one to seven characters in the (blpscxh) format.

b........ Specifies the baud rate. Specify one of the following letters representing the available baud rates:

Letter	Baud rate (bps)
0	(Send = 1200/Receive = 75)
1	(Send = 75/Receive = 1200)
2	110
3	....
4	150
5	200
6	300
7	....
8	600
9	....
A	1200
B	....
C	2400
D	4800
E	9600
F	19200
G	38400

l .... Specifies the number of bits comprising character.
7: 7 bits/character
8: 8 bits/character

p........ Specifies the type of parity check to be made.
N: No parity check
E: Even parity
O: Odd parity

s .... Specifies the number of stop bits.
1: 1 bit
3: 2 bits

c........ Specifies which of the four control lines are to be checked. Specify a hexadecimal number 0 to F. Bit 3 has no meaning and may be either 1 or 0. Bits 2 to 0 has the following meanings:

bit 2 ... Specifies whether the DSR level is to be checked in the send mode (DSR corresponds to SIN in the SIO interface).
- 1: Off (no check).
- 0: On (DSR checked). If DSR is off, the interface suspends the transmission (output) of a character until DSR goes on.

bit 1 ... Specifies whether the DSR level is to be checked in the receive mode.
- 1: Off (no check).
- 0: On (DSR checked). If DSR is off, the interface suspends the reception (input) of a character until DSR goes on.

bit 0 ... Specifies whether the DCD (Carrier Detect) level is to be checked in the receive mode.
- 1: Off (no check).
- 0: On (DCD checked). If DCD is off, the interface generates an error.

x........ Specifies whether the XON/XOFF protocol is to be used for communications control.
- X: XON/XOFF protocol enabled.
- N: XON/XOFF protocol disabled.

h........ Specifies whether the shift-in/shift-out (SI/SO) control sequences are to be used.
- S: Shift-in/shift-out control enabled in the 7-bit/character communications mode. S is disallowed in the other communications mode.
- N: Shift-in/shift-out control disabled.

The (blpscxh) options can be omitted entirely or partially. Spaces must be specified, however, for options which are omitted if there are any following options. When BASIC is started, these options are initialized to (D8N3FNN). These settings may be changed using the optional CONFIG command.

(2) Controls lines

Note that the controls lines to be used differ from device to device.

DTR:
Set to ON whether the interface is opened in the "I" or "O" open mode. DTR is set to OFF when the interface is closed (valid for COM0:, COM1:, and COM2:).

RTS:
Set to ON when the interface is opened in the "O" open mode. RTS is set to OFF when the interface is closed (valid for COM0:).

DSR:
This line is valid when DSR check is enabled in the receive mode. When the interface is opened in the "I" mode with the DSR receive check bit set to 0, the execution of the OPEN statement is not completed until DSR is set to ON. (valid for COM0:, COM1:, and COM2:).

DCD:
This line is valid when DCD check is enabled in the receive mode. When the interface is opened in the "I" mode with the DCD check bit set to 0, the execution of the OPEN statement is not completed until DCD is set to ON. (valid for COM0: and COM2:).

(3) Errors that may occur during open processing

- **FC Error (Illegal Function Call)**
  An attempt was made to open two communications devices at a time.
- **DU Error (Device Unavailable)**
  The specified communications device is not ready.
- **DT Error (Device Time Out)**
  The DSR or DCD line did not go on within a certain period of time after an OPEN "I" statement was executed with the DSR receive check or DCD check bit set to ON.

5.2.2 Output to the RS-232C interface

You can send data to the RS-232C interface in the same way as writing data to a file on an auxiliary storage device such as the RAM disk using the PRINT #, PRINT # USING, and WRITE # statements.

(1) Control lines

CTS:
When set to ON, enables data transmission to the external device.

DSR:
When the DSR send check bit is OFF, data is sent to the RS-232C port regardless of the state of the DSR line. When the DSR send check bit is set to ON, data transmission to the external device is deferred until the DSR line goes on (valid for COM0:, COM1:, and COM2:; COM1: uses the DSR line on the input port).

(2) Errors that may occur during send operations

- **DT Error (Device Time Out)**
  - The CTS line did not go on within a specified time.
  - The DSR line did not go on within a certain period of time after an OPEN "I" statement was executed with the DSR receive check bit set to ON.
- The STOP key was pressed for some reason when the interface was waiting for receive data.

5.2.3 Input from the RS-232C interface

You can receive data from the RS-232C interface in the same way as reading data from a file on an auxiliary storage device such as the RAM disk. The statements and functions that are used to receive data from the RS-232C are INPUT # and LINE INPUT # (statements) and EOF, LOC, LOF, and INPUT$ (functions).

(1) Functions

- **EOF(<file number>)**
  The EOF function returns – 1 (true) if executed when the receive buffer is empty and 0 (false) when the buffer is not empty.
- **LOC(<file number>)**
  The LOC function returns the number of data bytes remaining in the receive buffer.

- **LOF(<file number>)**
  The LOC function returns the number of free bytes remaining in the receive buffer.
- **INPUT$(<no. of characters>,<file number>)**
  The INPUT$ function receives the number of data bytes specified in <no. of characters> from the RS-232C interface and returns them as a character string.

  NOTE:
  BASIC has a receive buffer of 240 bytes.

(2) Control lines

DSR:
When the DSR check bit is set to ON, the DSR line is monitored during a receive operation and an error is generated if it is not ON (valid for COM0:, COM1:, and COM2:).

DCD:
When the DCD check bit is set to ON, the DCD line is monitored during a receive operation and an error is generated if it is not ON (valid for COM0: and COM2:).

(3) Errors that may occur during receive operations

- **DF Error (Device Fault)**
  The DCD or DSR line was found OFF during a data receive operation. This error can occur when the RS-232C port is opened for input with the DSR receive or DCD check bit set to ON.
- **IO Error (Device I/O Error)**
  A parity, overrun, or framing error occurred during a receive operation. The computer can continue processing ignoring the error, but the validity of the data in the receive buffer at that time cannot be guaranteed.
- **IE Error (Input Past End)**
  The STOP key was pressed while the PX-4 was waiting for receive data during the execution of an INPUT #, LINE INPUT #, or INPUT$.

5.2.4 Closing the RS-232C interface

You can close the RS-232C interface in the same way as a file on an auxiliary storage device such as the RAM disk using the CLOSE statement.

Sample data communications programs:

<Computer A>

```
5 'COMPUTER A(SEND)
10 OPEN "O",#1,"COMO:(D8N3FNN)"
20 PRINT "ENTER MESSAGE TO SEND...";
30 LINE INPUT A$
40 IF A$="*" THEN 70
50 PRINT #1,A$
60 GOTO 20
70 PRINT #1,A$
80 CLOSE
90 END
```

```
run
ENTER MESSAGE TO SEND...TEST! TEST! THIS
 IS COMPUTER A
ENTER MESSAGE TO SEND...*
Ok
```

<Computer B>

```
5 'COMPUTER A(SEND)
10 OPEN "I",#1,"COMO:(D8N3FNN)"
20 LINE INPUT A$
30 IF A$="*" THEN 60
40 PRINT A$
50 GOTO 20
60 PRINT "MESSAGE COMPLETE"
70 CLOSE
80 END
```

```
TEST! TEST! THIS IS COMPUTER A
MESSAGE COMPLETE
```

5.2.5 Sending a BASIC program

You can send and receive BASIC programs to and from external peripheral devices or other computers via an RS-232C interface using the LOAD, SAVE, and LIST commands. When using these commands for the RS-232C interface, you may specify communications options in the same way as in the OPEN statement. The option character 1 (bits/character) must always be set to 8 or 7 bits and shift-in/shift-out be enabled. Some examples follow.

SAVE"COM0:"
LIST"COM0:"

The above two commands send the program in the active program area to the standard RS-232C interface. These commands send the program in the ASCII format and appends a CTRL + Z (&H1A) code to the end of the file. The P option has no meaning for the SAVE"COM0:" command; it sends the program in the ASCII format whether the Option is specified or not.

LOAD"COM0:"

This command receives an ASCII format program via the standard RS-232C interface and loads it into the active program area. The command continues receiving program data until it receives a CTRL + Z (&H1A) code. Loading from the RS-232C interface can be stopped by pressing the STOP key.

NOTE:
You can also specify COM1: to COM3: interfaces to direct program data to other external devices if an optional RS-232C board is installed.

Examples:

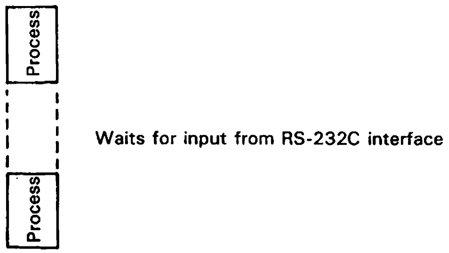
```
SAVE "COM0:"
SAVE "COM0:(88N1)"
LIST "COM0:(67N   S)"


LOAD "COM0:"
LOAD "COM0:(E8E1)"
```

5.2.6 Handling interrupts from the RS-232C interface

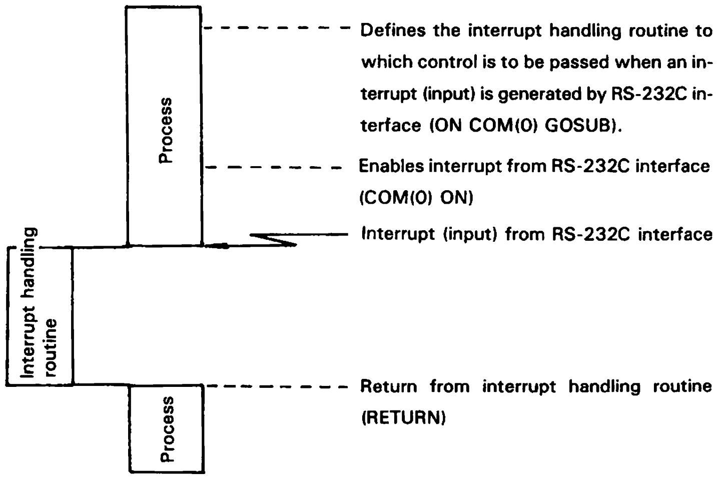
When waiting for data from the RS-232C interface, BASIC normally can do nothing but wait. By using interrupt handling commands, however, BASIC instructs BASIC to do other processing while it is waiting for input from the RS-232C interface.

When interrupt is disabled

BASIC can do nothing while it is waiting for input from the RS-232C interface as illustrated below.



- **When interrupt is enabled**
  As shown below, BASIC can do some tasks while it is waiting for input from the RS-232C interface. When input data is received at the RS-232C interface, BASIC executes the interrupt handling routine specified in the ON COM GOSUB statement. Control is returned from the interrupt handling routine via a RETURN statement in the interrupt handling routine.


Process
Defines the interrupt handling routine to which control is to be passed when an interrupt (input) is generated by RS-232C interface (ON COM(0) GOSUB).
Enables interrupt from RS-232C interface (COM(0) ON)
Interrupt (input) from RS-232C interface
Interrupt handling routine
Return from interrupt handling routine (RETURN)
Process

Appendix A DRIVE NAMES

Drive	Peripheral
A:	RAM disk
B:	ROM capsule-1
C:	ROM capsule-2
D:,E:,F:,G:	Floppy disk drives
H:	Microcassette drive
I:	RAM cartridge
J:	ROM cartridge-1
K:	ROM cartridge-2
SCRN:	LCD screen
LPT0:	Printer
COM0:	RS-232C interface
COM1:	Serial interface (SIO)
COM2:	RS-232C input, or SIO output
COM3:	Cartridge serial
KYBD:	Keyboard
CAS0:	External cassette

Appendix B BASIC COMMANDS, STATEMENTS, AND FUNCTIONS AVAILABLE FOR I/O DEVICES

BASIC commands, statements, and functions which can be used with various I/O devices are given in the table below. Disk-1 and Disk-2 denote one of the following disk drives:
Disk-1: RAM disk, RAM cartridge, or floppy disk drives
Disk-2: ROM capsules, or ROM cartridges

Drive name / Command, Statement, or Function	KYBD:	SCRN:	LPT0:	COMn	Disk-1	Disk-2	H:	CAS0:
CLOSE	○	○	○	○	○	○	○	○
DSKF	×	×	×	×	○	○	Note 2	×
EOF	–	×	×	○	○	○	○	○
GET	×	×	×	×	○	○	Note 3	×
INPUT #	○	×	×	○	○	○	○	○
INPUT$	○	×	×	○	○	○	○	○
LINE INPUT #	○	×	×	○	○	○	○	○
LIST	×	○	○	○	○	×	○	○
LOAD	○	×	×	○	○	○	○	○
LOC	–	×	×	○	○	○	○	×
LOF	–	×	×	○	○	○	○	×
OPEN "I"	○	×	×	○	○	○	Note 3	○
OPEN "O"	×	○	○	○	○	×	Note 3	○
OPEN "R"	×	×	×	×	○	○	Note 3	×
POS	×	○	○	○	○	○	○	○
PRINT #	×	○	○	○	○	×	○	○
PRINT USING #	×	○	○	○	○	×	○	○
PUT	×	×	×	×	○	×	Note 3	×
SAVE	×	Note 1	Note 1	Note 1	○	×	○	○
WIDTH	×	×	○	○	×	×	×	○
WRITE	×	○	○	○	○	×	○	○

○ Available
× Not available
– No meaning

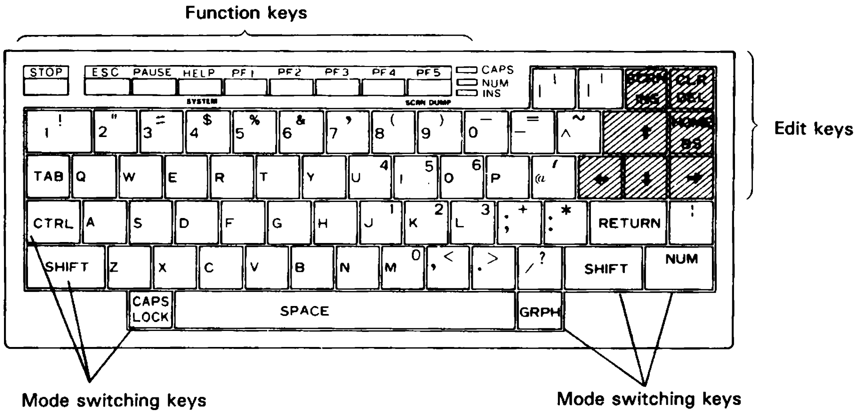
Note 1: Data is output in an ASCII format even if the A option is omitted.
Note 2: Exact value is not returned due to the limitations imposed by cassette tape.
Note 3: There are some restrictions on the use of the file. For example, record numbers must be sequentially assigned, beginning with 1 for a random data file.

Appendix C FORMATTING CHARACTERS

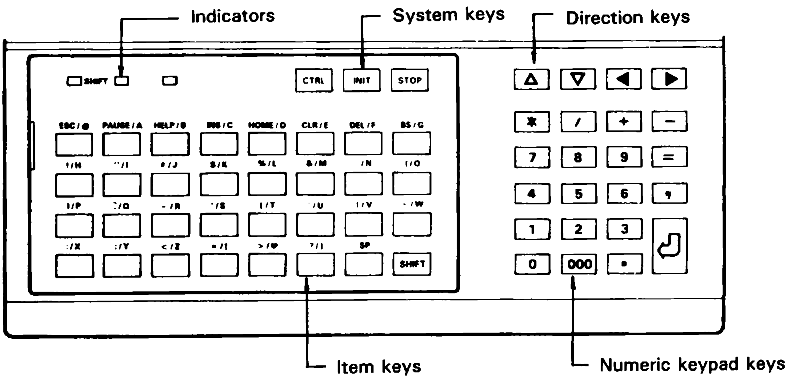
Formatting character	Remarks
!	Causes the first character of a string to be displayed.
&n_spaces&	Causes (n+2) characters to be displayed from a string, starting at the first character.
@	Causes the contents of a string to be displayed as is.
#	Causes the number of digits equal to the number of #'s to be displayed. The result is right-justified.
.	Causes a decimal point to be displayed in the same position when it appears.
+	Causes the sign to be displayed before or after a number.
−	Causes a minus sign to be displayed after a negative number.
* *	Causes leading spaces to the left of a number in the print area to be filled with *'s.
\\	Causes \ (only one) to be displayed to the left of a number in the print area.
* *\ or * *	Causes \ to be displayed just before a number and * to the left of the \, to fill the leading spaces in the print area.
,	Causes , to be displayed every three digits in the integer part of a number.
^^^^	Causes a number to be displayed using an exponent.
_ (under line)	Causes the string following the _ to be displayed as is.

Appendix D KEYBOARDS

ASCII keyboard

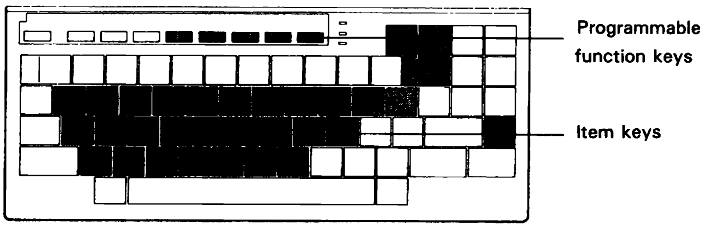


Item keyboard

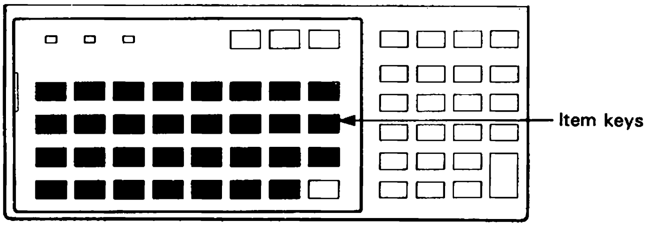


Appendix E PROGRAMMABLE FUNCTION KEYS AND ITEM KEYS

You can assign any string to the programmable function keys and item keys. The keys that can be assigned strings are identified by shaded boxes in the figures below.



ASCII keyboard



Item keyboard

①**Programmable function keys (ASCII keyboard only)**

Load a string for the programmable function keys in the following format:

KEY PF_key_number,string

Specify a number from 1 to 10 for the PF_key_number. 1 to 5 correspond to PF1 to PF5, and 6 to 10 correspond to SHIFT + PF1 to SHIFT + PF5, respectively. The number of characters in the string must not exceed 15. Extra characters are ignored. Use the CHR$(n) to define nonprintable characters such as control codes. See Appendix J "CHARACTER CODES" for the control codes.

Examples:

```
KEY 1,"AUTO"
```

Pressing the [PF1] key provides the same effect as typing [A] [U] [T] [O] .

```
KEY 5,"RUN"+CHR$(13)
```

Pressing the [PF5] key provides the same effect as typing [R] [U] [N] and pressing the [RETURN] key.

②Item keys

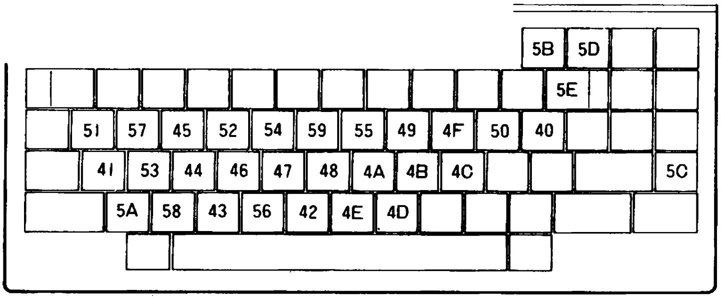
Load a string for the item keys in the following format:

KEY item_key_number,string

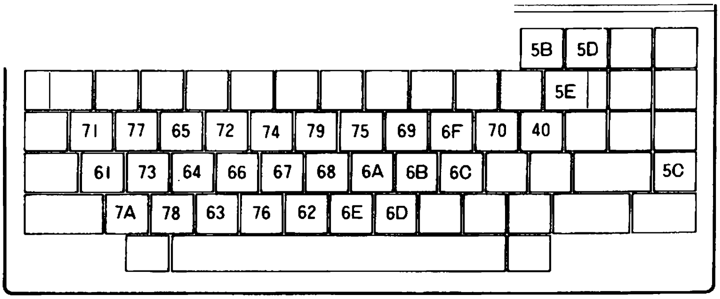
The item_key_number must be within the range of &H40-&H5E, or &H60-&H7E. Keys corresponding to the ASCII and item keyboards and the item key numbers are given below.

ASCII keyboard (numbers in hexadecimal)

CAPS lock mode (the CAPS LED on the keyboard is on)



Normal mode (three LEDs on the keyboard are off)



Item keyboard (numbers in hexadecimal)

Normal mode (the SHIFT LED is off)

40	41	42	43	44	45	46	47
48	49	4A	4B	4C	4D	4E	4F
50	51	52	53	54	55	56	57
58	59	5A	5B	5C	5D	5E	SHIFT

Shift mode (the SHIFT LED is on)

60	61	62	63	64	65	66	67
68	69	6A	6B	6C	6D	6E	6F
70	71	72	73	74	75	76	77
78	79	7A	7B	7C	7D	7E	SHIFT

The number of characters in <string> must not exceed 15. BASIC ignores the 16th and subsequent characters. Use the CHR$(n) function to define non-printable characters such as the control codes.

The item keys on the keyboard are enabled immediately once they are loaded with a string.

On the ASCII keyboard however, you must first switch the key mode to use the function keys. The key mode is set by:

KEY 253: Accepts any defined strings (item key mode)

KEY 254: Ignores any defined strings and accepts normal characters (default mode established when BASIC is started).

You can enter a programmed string by pressing the item key while holding CTRL and SHIFT keys, even if you are in another keyboard mode.
Normal characters will be entered even after the execution of KEY,253, if no item key has been defined.

Strings assigned to item keys on both the item and ASCII keyboards can be cancelled by executing KEY 255.

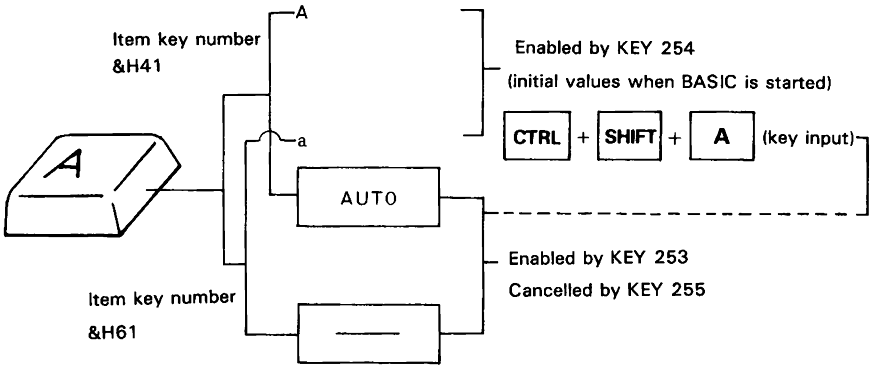
NOTE:
Executing this command on the item keyboard will inhibit entry from the keyboard.

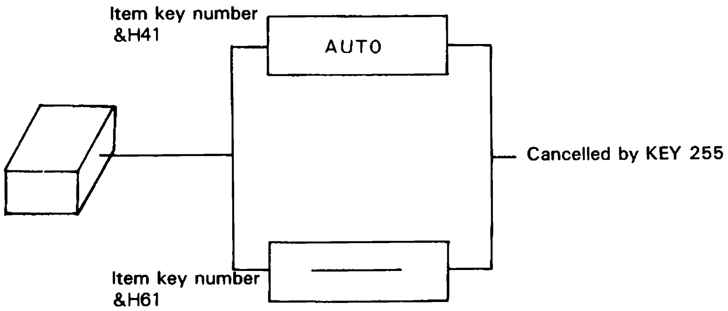
Example:

```
KEY &H41,"AUTO"
```

BASIC will display "AUTO" if you press the [A] key after executing KEY 253.

If you further press the [A] key after executing KEY 254, then BASIC will display "A". Pressing the [CTRL], [SHIFT], and [A] keys together will also display "AUTO".

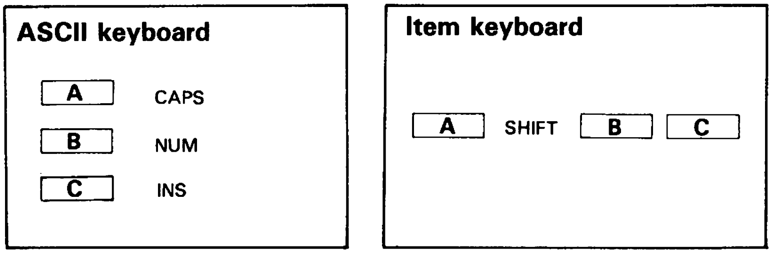




The three LEDs on the ASCII or item keyboard can be turned on and off with a program by using the PRINT statement in the following format:

```
PRINT CHR$(&H1B);CHR$(n)
```

Specify n as a number between &HA0 and &HA5 (160 to 165). The LEDs will turn on and off as follows:



A: Is turned on when the value of n is &HA2 and off when it is &HA3.
B: Is turned on when the value of n is &HA4 and off when it is &HA5.
C: Is turned on when the value of n is &HA0 and off when it is &HA1.

NOTE:
This statement turns the LEDs on and off but does not change the key entry modes.

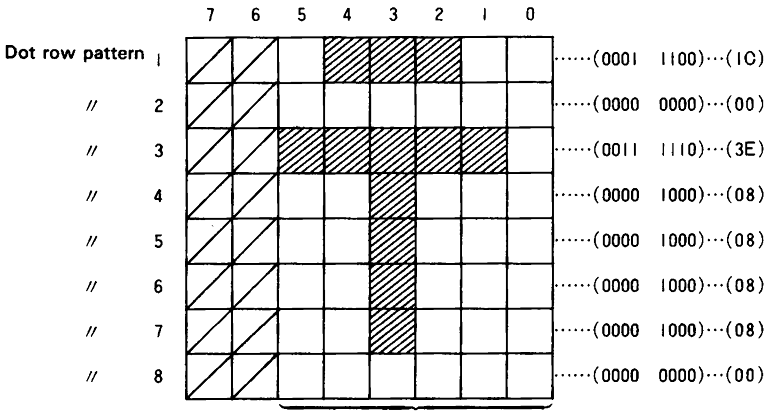
Appendix F USER-DEFINED CHARACTERS

(1) User-defined Characters

The PX-4 allows you to define your own character designs. You can assign up to 30 user-defined characters to any of the unused character codes between &HE2 and &HFF (the other character codes are already assigned letters, symbols, or control characters: see Appendix J). You can display these characters on the screen by using the CHR$ function.

Generating a character pattern

One character can consist of a matrix 6 dots wide by 8 dots long (one of the rectangular segments in the drawing below is referred to as a dot). To define a character pattern, you must compose it with eight bytes of dot data, with each dot row represented by the ASCII code equivalent to the desired binary format, as shown below.



6 × 8 dot matrix

The dot pattern shown above corresponds to the numbers on the right.

0000··· 0	1000··· 8
0001··· 1	1001··· 9
0010··· 2	1010··· A
0011··· 3	1011··· B
0100··· 4	1100··· C
0101··· 5	1101··· D
0110··· 6	1110··· E
0111··· 7	1111··· F

NOTE:
Bits 6 and 7 of each byte are ignored.

Assigning a character pattern to an ASCII character code.

Once you have designed a character pattern, it can be registered in the system as an ASCII character code by using the following statement.

```
PRINT CHR$(&H1B) + CHR$(&HE0) + CHR$(character code)
                              + CHR$(pattern for dot row 1)
                              + CHR$(pattern for dot row 2)
                              + CHR$(pattern for dot row 3)
                                .        .
                                .        .
                                .        .
                              + CHR$(pattern for dot row 8)
```

The ASCII character codes &HE2 to &HFF are available as user-defined characters.

```
5 ' USER-DFEINED CHARACTERS
10 PRINT CHR$(&H1B)+CHR$(&HE0)+CHR$(&HF0
)+CHR$(&H1C)+CHR$(&H0)+CHR$(&H3E)+CHR$(&
H8)+CHR$(&H8)+CHR$(&H8)+CHR$(&H8)+CHR$(&
H0)
20 PRINT CHR$(&HF0)

RUN
〒
Ok
```

A sample program for defining a user-defined character is given below.

You can enter user-defined characters from the keyboard if you load them directly into programmable function keys or item keys.

Example:

Assuming that "〒" is assigned to character code &HF0. You can define "〒" in the system by typing

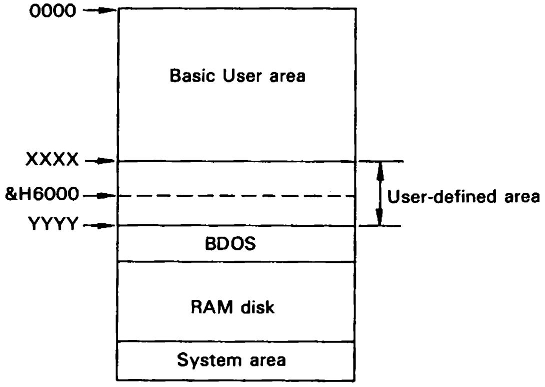
```
KEY 1,CHR$(&HF0)
```

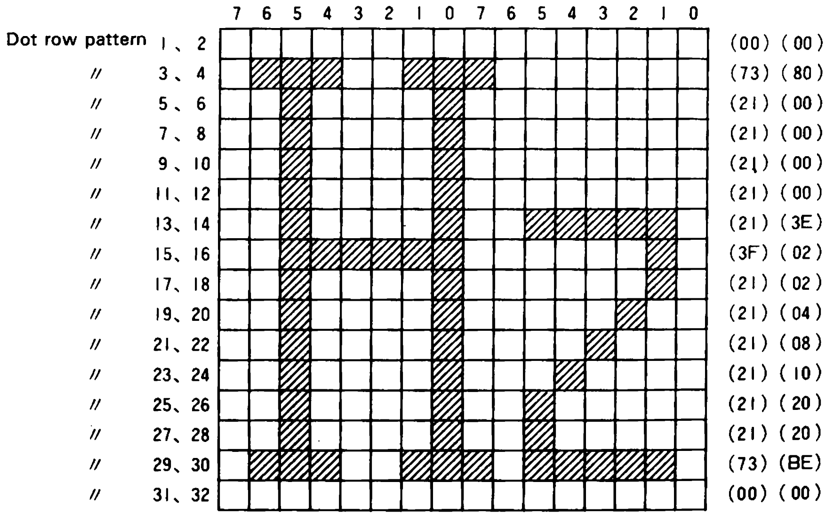
and pressing the PF1

(2) User-defined Pattern

Generating a user-defined pattern

A user-defined pattern consists of a 16 dots by 16 dots matrix and is specified with a total of 32 bytes. The pattern "Hz", for example, can be defined as shown below.



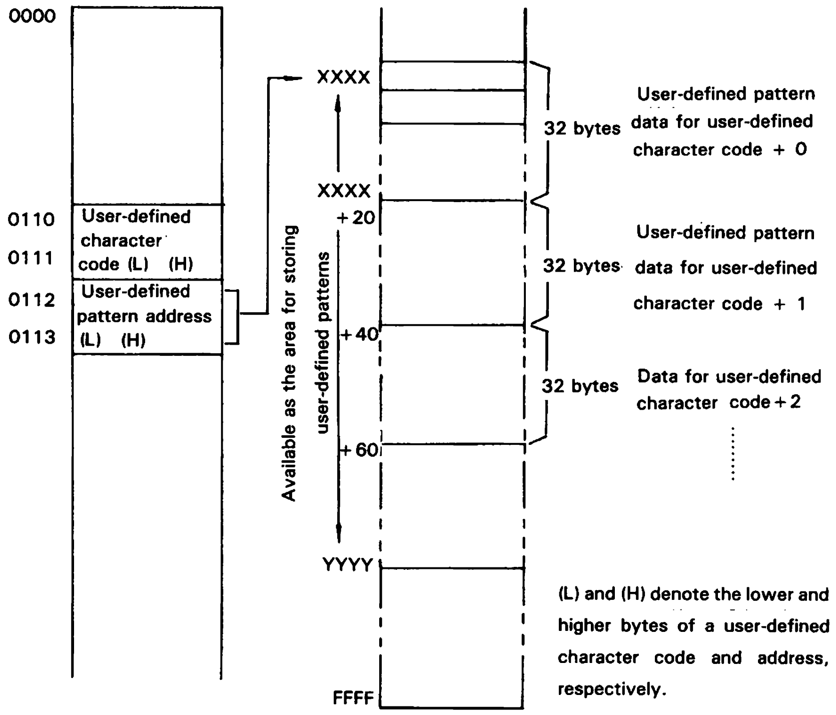


This dot pattern can be stored in memory in the form of binary bytes 00, 00, 73, 80, 21, 00, 21, 00, and so on, in that order.

User-defined pattern area configuration

BASIC stores the address of the area used for storing user-defined patterns in addresses &H0112 and &H0113 and the code for the first user-defined pattern in addresses &H0110 and &H0111. The user can specify the storage addresses and character codes of user-defined patterns by using the POKE statement.

User-defined patterns are defined by consecutive user-defined character codes in the area starting at address XXXX designated by the user-defined pattern address stored in memory addresses &H0112 and &H0113.



0000
0110
0111
0112
0113
User-defined character code (L) (H)
User-defined pattern address (L) (H)
Available as the area for storing user-defined patterns
XXXX
XXXX +20
+40
+60
YYYY
FFFF
32 bytes
User-defined pattern data for user-defined character code + 0
32 bytes
User-defined pattern data for user-defined character code + 1
32 bytes
Data for user-defined character code + 2
(L) and (H) denote the lower and higher bytes of a user-defined character code and address, respectively.

All memory addresses in hexadecimal.

User-defined character codes are assigned to codes from 0 to 65535. When storing code 2000 for example, you must load &HD0 in memory address &H0110 and &H07 in memory address &H0111 because the hexadecimal representation of 2000 is &H07D0. (The decimal to hexadeciaml conversion can be accomplished easily by using the BASIC PRINT HEX&(2000) statement.)

When using an area at &H5000 for storing user-defined patterns, load &H00 in memory address &H0112 and &H50 in memory address &H0113. This allows user-defined patterns to be defined in the memory area starting at &H5000 as 32-byte consecutive character codes (2000, 2001, 2002, and so on).

Defining user-defined patterns into the system

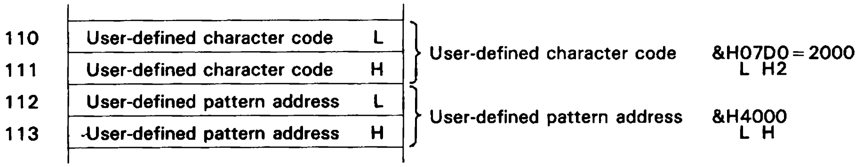
To define user-designed patterns in the system, load the pattern bytes at the address for the first byte of the corresponding character code. In the example below, the pattern corresponding to user-defined character code 2000 is loaded into the 32-byte area from &H4000 through &H401F. Next, the pattern corresponding to 2001 is loaded into the 32-byte area from &H4020 through &H403F.

Example:

```
5 'USER-DEFINED CHARACTERS
100 CLS
110 CLEAR ,&H3FFF
120 FOR I=0 TO 31
130   READ A
140   POKE &H4000+I,A
150 NEXT
160
170 POKE &H110,&HD0 :'2000 = &h07D0
180 POKE &H111,&H7
190 POKE &H112,0
200 POKE &H113,&H40
210 FONT(16,16),PSET,2000
220 END
230
240 DATA 0,0,&h73,&h80,&h21,0,&h21,0,&h2
1,0,&h21,0,&h21,&h3E,&h3F,2,&h21,2,&h21,
4,&h21,8,&h21,&h10,&h21,&h20,&h21,&h20,&
h73,&hBE,0,0
```

```
run
  Hz
```

Explanation:
Line 110 reserves the area for storing the user-defined pattern. The FOR NEXT loop from lines 120 through 150 reads user-defined patterns defined on line 240 into the area starting at address &H4000. Lines 170 and 180 load the user-defined character code and lines 190 and 200 load the start address of the pattern area. The program finally prints the user-defined pattern on the screen at line 210.

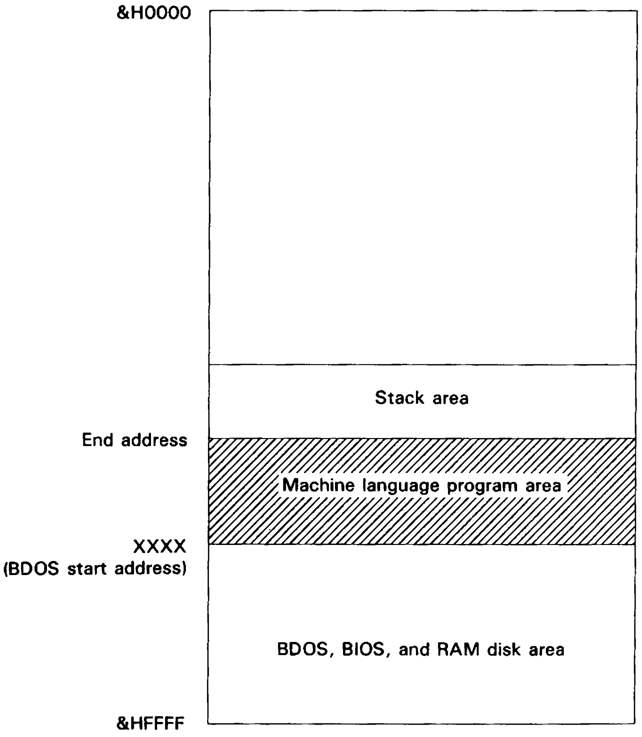

110
111
112
113
User-defined character code L
User-defined character code H
User-defined pattern address L
User-defined pattern address H
User-defined character code
&H07D0 = 2000
L H2
User-defined pattern address
&H4000
L H

You can store the user-defined patterns onto an auxiliary storage device with the BSAVE command, then load them back into memory with the BLOAD command.

Appendix G MACHINE LANGUAGE PROGRAMS

①Memory allocation

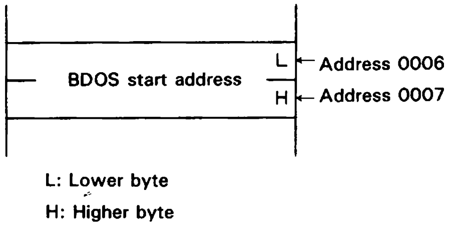
When using machine-language programs or defining user-defined characters, it is necessary to reserve a memory area that does not overlap the one used by BASIC. This is to prevent the BASIC programs or data from being destroyed by the machine language programs, and vice versa. To reserve the memory area, you must specify the end address of the area used by BASIC by using the CLEAR command, or in the /M: parameter when starting BASIC. The machine language program area start address must be lower than the BDOS start address or &H6000, whichever is smaller.



NOTE: All memory addresses are in hexadecimal.

Machine language programs can use the memory area from the address you specified up to XXXX-1. The address XXXX is determined by the sum of the areas available for the RAM disk and user BIOS. The address XXXX is stored in addresses 0006 and 0007.

Example:



```
PRINT HEX$(PEEK(7)*256+PEEK(6))
6206
Ok
```

The BDOS start address is &H6206 in this example.

Use the POKE command to load short machine language programs, one byte at a time. To load a machine language program file, use the BLOAD command.

- To invoke a machine language program from a BASIC program, use the USR function or CALL statement.

② USR function

To invoke a machine language program with the USR function, specify the start address of the machine language program. Then assign a subroutine number to the machine language program with the DEF USR statement. The subroutine number must be an integer between 0 and 9.

```
DEF USR<subroutine number> = <start address of the memory area
                              that is to be loaded with the
                              machine language program>
```

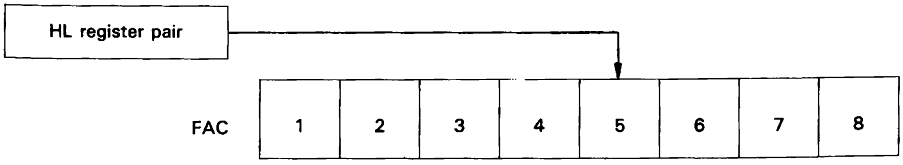
You can now call a machine language program by specifying the subroutine number in the USR function. The format is as follows:

```
<variable name> =USR[<subroutine number>](argument)
```

<variable name> is used to store any results from the machine language program execution that are to be returned to the BASIC program. When there are no results to be returned, specify the same variable name for the argument.
Specify for <subroutine number> the same subroutine number that is defined in the DEF USR statement. If the number is omitted, 0 is assumed.
Specify in (argument) the value (of a numeric or string variable) to be passed from the main program to the machine language program. A variable name must be specified even when there are no values to be passed.

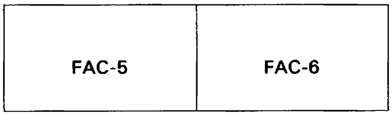
When the USR function is used, register A is loaded with one of the following values which identifies the type of the argument to be passed to the machine language program. If the argument is a number, the HL register pair is loaded with the address of the fifth byte of the FAC (Floating Point Accumulator) memory area, which is used to hold the argument.

Type of argument	Value in register
A Integer	2
String	3
Single-precision real number	4
Double-precision real number	8



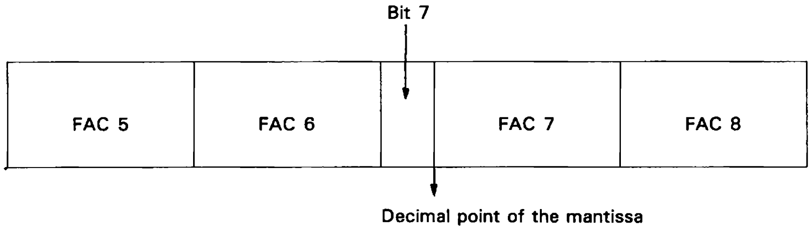
The argument is loaded into the FAC in one of the following ways:

- When the argument is an integer



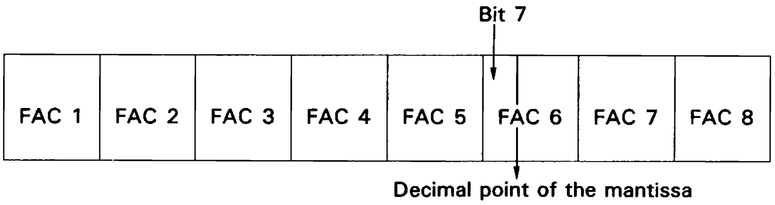
Bytes 5 and 6 of the FAC are loaded with the lower 8 bits and higher 8 bits of the argument, respectively.

- When the argument is a single-precision real number



The highest 7 bits, intermediate 8 bits, and lowest 8 bits of the argument's mantissa are stored into bytes 7, 6, and 5 of the FAC respectively. Bit 7 of byte 7 contains the sign of the argument (0 for positive and 1 for negative). The decimal point of the mantissa is assumed to be placed between bits 6 and 7 of byte 7. Byte 8 is used to hold the value (exponent minus 128).

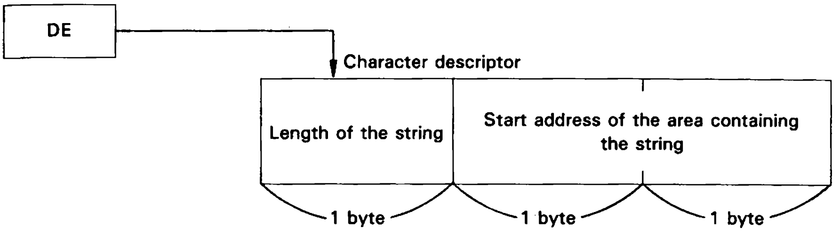
- When the argument is a double-precision real number



Bytes 5 to 8 of the FAC are used in the same way as for a single-precision number. Bytes 1 to 4 contain the lowest 4 bytes of the mantissa.

• When the argument is a string

The start address of the 3-byte area called the "string descriptor" is loaded into the DE register pair. Byte 1 contains the length of the string (0-255); and bytes 2 and 3 contain the start address of the area in which the string is stored.

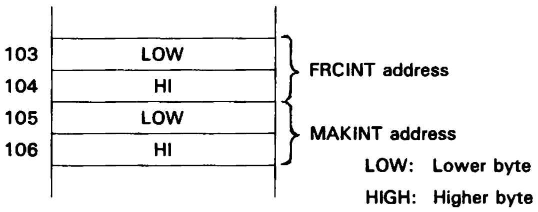


If the argument is a literal string in the program, bytes 2 and 3 of the string descriptor point to the program text.

To make the USR function return a value to BASIC, place the function value in the FAC. Normally, a USR function returns the same type of value as the argument originally passed to the function.

This is easily accomplished by the FRCINT and MAKINT routines provided by PX-4 BASIC. FRCINT converts the value in the FAC to an integer. The converted value is placed into the HL register pair. MAKINT converts the value in the HL register pair into an integer and places it in the FAC. These routines are used to convert to integers values passed from a USR function to a user-defined machine language program, or to return a value to the USR function.

The start addresses of FRCINT and MAKINT are shown in the figure below.



③Invoking a machine language program with the CALL statement

You may also invoke a machine language program from a BASIC program with the CALL statement.

A CALL statement with no argument simply invokes a machine language program and executes it. Control must be returned from the machine language program to the main program via a RET instruction. A CALL statement with arguments passes the addresses of areas that contain them. The method of passing arguments depends on the number of arguments to be passed as follows:

If there are 3 or less arguments, they are placed in registers. The address of the area containing the first argument is placed in the HL register pair, the addresses of the areas containing the second and third arguments, if any, are placed in the DE and BC register pairs, respectively.
If there are more than 3, they are passed as follows:

(a) The address of the area containing the first argument is placed in the HL register pair.
(b) The address of the area containing the second argument is placed in the DE register pair.
(c) For arguments 3, 4,...and so on, the start address of the table containing the start addresses of the arguments are placed in register pair BC. (i.e., BC points to the start address of the area containing the third argument).

④BSAVE, BLOAD

The BSAVE command saves a machine language program or data in memory into a file. The BLOAD command loads a machine language program or data file into memory.

The BSAVE command writes a 5-byte header at the beginning of the file in addition to the data in memory. The BLOAD command cannot load a file which has an illegal header. The 5-byte header has the following format:

1	&HFD
2-3	Address
4-5	Length (in bytes)
⋮	} data

The address and length are specified in the BSAVE command. If no address is specified in the BLOAD command, the file will be loaded into the memory space starting at the address originally specified in the BSAVE command.

Appendix H CONTROL CODES

Code		Function	Key usage
Decimal	**Hexadecimal**		
1	**01**	Move the cursor to the beginning of the current line.	CTRL + A
2	**02**	Move the cursor one word to the left.	SHIFT + B
3	**03**	Stop the program execution. These keys also stop the automatic number generation that is initiated by the AUTO command.	STOP , CTRL + C
5	**05**	Delete characters to the end of the line.	CTRL + E
6	**06**	Move the cursor one word to the right.	CTRL + F
8	**08**	Delete the character to the left.	BS , CTRL + H
9	**09**	Move the cursor to the next tab position.	TAB, CTRL + I
11	**0B**	Move the cursor to the home position (upper left corner) of the virtual screen.	HOME , CTRL + K
12	**0C**	Clear the virtual screen.	CLR , CTRL + L
13	**0D**	Execute a command or statement.	RETURN CTRL + M
18	**12**	Toggle between the overwrite and insert modes of the BASIC screen editor.	INS CTRL + S
19	**13**	Suspend the execution of a BASIC program.	PAUSE , CTRL + S
24	**18**	Move the cursor to the end of the line.	CTRL + X
26	**1A**	Delete all characters to the end of the screen.	CTRL + Z

Code		Function	Key usage
Decimal	Hexadecimal		
27	1B	Escape code.	$\boxed{\text{ESC}}$
28	1C	Moves the cursor to the right.	$\boxed{\rightarrow}$
29	1D	Moves the cursor to the left.	$\boxed{\leftarrow}$
30	1E	Moves the cursor upward.	$\boxed{\uparrow}$
31	1F	Moves the cursor downward.	$\boxed{\downarrow}$

GET
GO
GOSUB

HEX$

IF
IMP
INKEY$
INP
INPUT
INPUT #
INSTR
INT

KEY
KILL

LEFT$
LEN
LET
LINE
LIST
LLIST
LOAD
LOAD?
LOC
LOCATE
LOF
LOG
LOGIN
LPOS
LPRINT
LSET

MENU
MERGE
MID$
MKD$
MKI$
MKS$
MOD
MOTER
MOUNT

NAME
NEW
NEXT
NOT

OCT$
OFF
ON
OPEN
OPTION
OPTION COUNTRY
OPTION CURRENCY
OR
OUT

PCOPY
PEEK
POINT
POKE
POS
POWER
PRESET
PRINT
PRINT #
PSET
PUT

RANDOMIZE
READ
REM
REMOVE
RENUM
RESET
RESTORE
RESUME
RETURN
RIGHT$
RND
RSET
RUN

SAVE
SCREEN
SGN
SIN
SOUND
SPACE$
SQR
STAT
STEP
STOP
STR$
STRING$
SWAP
SYSTEM

TAN
TAPCNT
THEN
TIME$
TITLE
TO
TROFF
TRON

USING
USR

VAL
VARPTR

WAIT
WEND
WHILE
WIDTH
WIND
WRITE

XOR

Appendix J CHARACTER CODES

	Hex. No.	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Hex. No.	**Binary No.**	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	0000	0	16	SP 32	@ 48	@ 64	P 80	` 96	p 112	┼ 128	○ 144	160	176	192	208	⌂ 224	240
1	0001	1	17	! 33	1 49	A 65	Q 81	a 97	q 113	┴ 129	♠ 145	161	177	193	209	↵ 225	241
2	0010	2	18	" 34	2 50	B 66	R 82	b 98	r 114	┬ 130	♥ 146	162	178	194	210	226	242
3	0011	3	19	# 35	3 51	C 67	S 83	c 99	s 115	┤ 131	♦ 147	163	179	195	211	227	243
4	0100	4	20	$ 36	4 52	D 68	T 84	d 100	t 116	├ 132	♣ 148	164	180	196	212	228	244
5	0101	5	21	% 37	5 53	E 69	U 85	e 101	u 117	─ 133	♪ 149	165	181	197	213	229	245
6	0110	6	22	& 38	6 54	F 70	V 86	f 102	v 118	│ 134	🚗 150	166	182	198	214	230	246
7	0111	7	23	' 39	7 55	G 71	W 87	g 103	w 119	┌ 135	🌲 151	167	183	199	215	231	247
8	1000	8	24	(40	8 56	H 72	X 88	h 104	x 120	┐ 136	🚗 152	168	184	200	216	232	248
9	1001	9	25	) 41	9 57	I 73	Y 89	i 105	y 121	└ 137	🏆 153	169	185	201	217	232	249
A	1010	10	26	* 42	: 58	J 74	Z 90	j 106	z 122	┘ 138	🏃 154	170	186	202	218	234	250
B	1011	11	27	+ 43	; 59	K 75	[91	k 107	{ 123	▒ 139	↑ 155	171	187	203	219	235	251
C	1100	12	28	, 44	< 60	L 76	\ 92	l 108	| 124	■ 140	↓ 156	172	188	204	220	236	252
D	1101	13	29	- 45	= 61	M 77	] 93	m 109	} 125	▬ 141	× 157	173	189	205	221	237	253
E	1110	14	30	. 46	> 62	N 78	^ 94	n 110	~ 126	▮ 142	÷ 158	174	190	206	222	238	254
F	1111	15	31	/ 47	? 63	O 79	_ 95	o 111	⌂ 127	● 143	± 159	175	191	207	223	239	255

ASCII

NOTES:

1. *$(0)_D$ through $(31)_D$ are control characters.*
2. *$(32)_D$ through $(127)_D$ are ASCII characters.*
3. *Characters displayed for codes E0 to FF can be defined by the user. For further details, see Appendix H and the Operating Manual.*

Differences between the USASCII character set and the character sets of other countries are as shown below.

Country / Dec. Code	United States	France	Germany	England	Denmark	Sweden	Italy	Spain	Norway
35	#	#	#	£	#	#	#	₧	#
36	$	$	$	$	$	¤	$	$	¤
64	@	à	§	@	É	É	@	@	É
91	[	°	Ä	[	Æ	Ä	°	¡	Æ
92	\	ç	Ö	\	Ø	Ö	\	Ñ	Ø
93	]	§	Ü	]	Å	Å	é	¿	Å
94	^	^	^	^	Ü	Ü	^	^	Ü
96	`	`	`	`	é	é	ù	`	é
123	{	é	ä	{	æ	ä	à	¨	æ
124	|	ù	ö	|	ø	ö	ò	ñ	ø
125	}	è	ü	}	å	å	è	}	å
126	~	¨	ß	~	ü	ü	ì	~	ü

Appendix K ERROR CODES AND MESSAGES

BASIC displays an error message and enters the command mode when one of the errors listed in this appendix occurs.

Message	Code	Desciption
/0	11	**Division by zero** An attempt was made to divide by zero. < Possible causes > 1. Zero was used as a divisor. 2. Division was attempted using an undefined variable as the divisor.
AC	72	**Tape access error** An attempt was made to access an access-inhibited file. < Possible causes > 1. An attempt was made to access an access-inhibited microcassette file. 2. An attempt was made to mount a microcassette without executing the REMOVE command to demount the previous one. 3. The REMOVE command was executed while the microcassette was in the demounted condition. 4. The WIND command was executed while the microcassette was in the mounted condition. 5. The TAPCNT function was evaluated when the microcassette was in the mounted condition.
AO	55	**File already open** An OPEN "O" statement was executed for a file which was already open, or a KILL command was executed for a file that was open.

Message	Code	Description
BF	**54**	**Bad file mode** An illegal statement or function was specified for a file. <Possible causes> 1. A PUT or GET statement or the LOF function was executed for a sequential file. 2. A LOAD command was executed for a random file. 3. A file mode other than "I", "R", and "O" was specified in an OPEN statement. 4. An attempt was made to MERGE a file that was not saved in ASCII format.
BN	**52**	**Bad file number** A statement or command references a file that has not been opened. Or, the file number specified in the OPEN statement was outside of the range of file numbers that was specified when BASIC was first started.
BS	**9**	**Bad subscript** The subscript for an array variable was outside the range permitted for that array. 1. The subscript specified was greater than the maximum specified in the DIM statement defining that array. 2. A subscript greater than 10 was used when no array variable was specified by a DIM statement. 3. Zero was used after executing OPTION BASE 1.
CN	**17**	**Can't continue** BASIC cannot continue toexecute the program <Possible causes> 1. Program execution was terminated due to an error. 2. The program was modified while execution was suspended. 3. The STOP key was pressed during execution of an received faster than it can be processed by the computer.
CO	**28**	**Communication buffer overflow** The receive buffer overflowed during receipt of data via the RS-232C interface. This error occurs when data is received faster than it can be processed by the computer.

Message	Code	Description
DD	**10**	**Duplicate definition** An array was defined more than once. <Possible causes> 1. A second DIM statement was executed for an array without freeing the memory area allocated to that array by the ERASE statement. 2. An undefined array was used, then an attempt was made to redimension that array with a DIM statement. 3. The OPTION BASE statement was used more than once. 4. An OPTION BASE statement was executed after an array had been dimensioned by a DIM statement or after the array variable had been used.
DF	**61**	**Disk full** All the disk storage space is in use.
DR	**70**	**Disk read error** An error occurred while a file on a disk was being input.
DS	**66**	**Direct statement in file** A program line with no line number was encountered during the execution of a LOAD or a MERGE command. An attempt was made to execute a LOAD command for a data file or a machine language program.
DT	**24**	**Device time out** A peripheral device was not ready for input or output processing. 1. CTS was not held high and transmission via the RS-232C interface was not enabled within the needed period of time after an OPEN "O" statement was executed. Or, the transmitter was not ready within the needed period of time after a DSR send check was specified with the "c" option at interface open time. 2. The STOP key was pressed while output to the RS-232C interface was being deferred for some reason. 3. DSR or DCD did not go ON within the needed period of time after an OPEN"I" statement was executed. Both the DSR receive check and DCD check were specified with the "c" option at interface open time. 4. The printer was not ready when output to the printer was attempted.

Message	Code	Description
DU	68	**Device unavailable** An attempt was made to access a peripheral device not connected to the computer.
DW	71	**Disk write error** An error occurred while a file was being written to a disk device.
FA	25	**Device fault** DSR or DCD went OFF during input from the RS-232C interface after a DSR check or DCD check was specified with the "c" option in the OPEN"I" statement at interface open time.
FC	5	**Illegal function call** A statement or function was incorrectly specified. <Possible causes> 1. Specification of a negative number as an array variable subscript 2. Specification of a negative number as the argument in the LOG function 3. Specification of a negative number as the argument of the SQR function 4. Specification of a non-integer exponent with a negative mantissa 5. A call to a USR function for which the start address of the machine language program has not been defined 6. An incorrectly specified argument in any of the following statements or functions: MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, ON...GOSUB, ON...GOTO, ASC, SCREEN, VARPTR, LOGIN, STAT 7. Specification of a non-existent line number in a DELETE statement 8. Attempting to erase a non-existent array variable with an ERASE statement 9. Execution of an EDIT command for a program that was saved on an auxiliary storage device with a SAVE command with the "p" option then loaded into the program area again 10. Execution of a RENUM command with a number greater than 65529. Or, attempting to change the order of the program lines with a RENUM command 11. Execution of a SWAP statement for a variable to which no value has been given by a LET statement

Message	Code	Description
FD	**64**	**Bad file descriptor** An illegal file descriptor was specified in a LOAD, SAVE, KILL command, or OPEN statement.
FE	**58**	**File already exists** The new file name specified in a NAME statement is already being used by another file on the disk.
FN	**26**	**FOR without NEXT** A FOR statement was encountered without a corresponding NEXT.
FO	**50**	**Field overflow** The size of the random file variables specified in a FIELD statement exceeds the size of a record specified by an OPEN statement.
ID	**12**	**Illegal direct** A statement that is illegal in the direct mode was entered in direct mode.
IE	**82**	**Input past end** An attempt was made to input a file which was empty or a file from which all data had been read. (To avoid this error, use the EOF function.)
IE	**62**	**Input past end** The STOP key was pressed while the RS-232C interface was waiting for input with an INPUT#, INPUT$, or a similar command.
IO	**57**	**Device I/O error** An error occurred involving input or output to a peripheral device. <Possible causes> 1. An I/O error occurred during access to a disk device. 2. An error occurred during input from the RS-232C interface. (In this case, the error condition will be reset if input is continued, but there is no assurance that data received will be correct.) 3. The printer power was off or a fault occurred when data was output to the printer.
IT	**51**	**Internal error** An internal malfunction occurred in BASIC.
LO	**23**	**Line buffer overflow** A line that did not fit in the line buffer was input.
LS	**15**	**String too long** The length of a string variable exceeds 255 characters.
MF	**67**	**Too many files** An attempt was made to create a new disk file after all directory entries were full.
MO	**22**	**Missing operand** An operand required for an expression is missing.

Message	Code	Description
NE	53	**File not exist** The file name specified in a LOAD, KILL, NAME, or OPEN statement does not exist on the disk.
NF	1	**NEXT without FOR** A NEXT statement was executed without a corresponding FOR statement. <Possible causes> 1. A FOR statement corresponding to the NEXT was not executed previously. 2. The variable in a NEXT statement does not correspond to any previously executed FOR statement variable. 3. Execution branched to a point within a FOR/NEXT loop from elsewhere in the program.
NR	19	**No RESUME** No RESUME statement was included in an error processing routine. All error processing routine must conclude with an END or RESUME statement.
OD	4	**Out of data** A READ statement was executed when there was no unread data remaining in the program's DATA statement. <Possible causes> 1. The number of data items in a DATA statement was smaller than that of the variables in a READ statement. 2. Incorrect specification of a RESTORE statement. 3. Incorrect delimiting punctuation used in a DATA statement
OM	7	**Out of memory** The memory available is insufficient for executing a program. <Possible causes> 1. The program is too long. 2. The program uses too many variables. 3. The suscript range specified in a DIM statement is too large. 4. An e4xpression has too many levels of parentheses. 5. FOR...NEXT loop or GOSUB...RETURN sequences are nested to too many levels.
OS	14	**Out of string space** Insufficient memory space is available for storing string variables.

Message	Code	Description
OV	**14**	**Overflow** A numeric value was encountered whose magnitude exceeds the limits prescribed by BASIC. <Possible causes> 1. The result of an integer calculation was outside the range −32768 to 32767. 2. The result of a single or double precision number calculation was outside the range −1.70141E38 to 1.70141E38. 3. The argument specified for the CINT function was outside the range −32768 to 32767. 4. The argument specified for the HEX$ or OCT$ function was outside the range −32768 to 65535.
RG	**3**	**RETURN without GOSUB** A RETURN statement was encountered which did not correspond to previously executed GOSUB statement. <Possible causes> 1. Execution was transferred to a subroutine by a GOTO statement (GOSUB must be used). 2. The line number specified in a RUN command indicated a line in a subroutine. 3. The main program included no GOTO or END statement at the end and the program flow moved into a subroutine.
RN	**63**	**Bad record number** The record number specified in a PUT or GET statement was zero.
RW	**20**	**RESUME without error** A RESUME statement was executed outside an error processing routine. <Possible causes> 1. A RESUME statement was encountered in an error processing routine to which the program was passed by a GOTO or GOSUB statement. 2. A RESUME statement was encountered in an error processing routine that was executed due to the absence of an END statement between the main and error processing routines.

Message	Code	Description
SN	2	**Syntax error** A statement does not conform to the BASIC syntax rules. <Possible causes> 1. Wrong reserved word(s) 2. Unmatched parentheses 3. Wrong delimiting punctuation (commas, periods, colons, or semi-colons) 4. Variable name beginning with a character other than a alpha-betic character 5. Reserved words used as the first characters of a variable name 6. Wrong number or type of arguments specified in a function or statement 7. Type of a value included in a DATA statement did not match the corresponding variable in the list of variables specified in a READ statement.
ST	16	**String formula too complex** A string expression has too many levels of parentheses.
TM	13	**Type mismatch** A string was assigned to a numeric variable or a numeric value to a string variable. <Possible causes> 1. An attempt was made to assign a numeric value to a string variable. 2. An attempt was made to assign a string to a numeric variable. 3. The wrong type of value was specified as the argument of a function.
UF	18	**Undefined user function** A call was made to an undefined user function. <Possible causes> 1. The letters FN were used at the beginning of a variable name. 2. The function name was specified incorrectly in the DEFFN state-ment or when the function was called. 3. The user function was called before the corresponding DEFFN statement was executed.

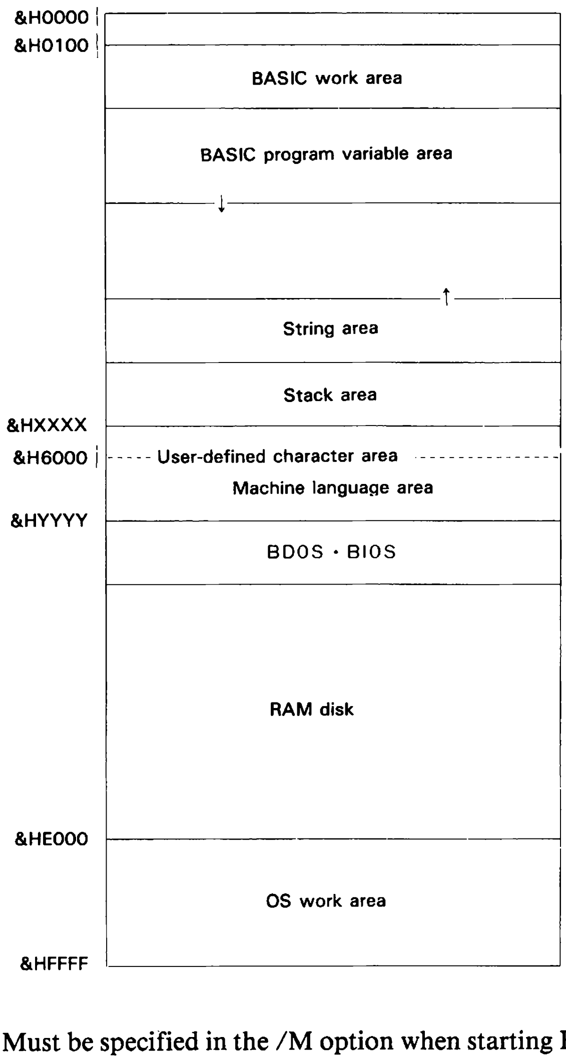
Message	Code	Description
UL	8	**Undefined line number** A non-existent line number was specified in one of the following commands or statements: GOTO, GOSUB, RESTORE, RUN, RENUM.
UP	21	**Unprintable error** No error message has been provided for the error codes, 27, 31-49, 56, 59, 60, 73-255.
WE	29	**WHILE without WEND** A WHILE statement was encountered without a corresponding WEND.
WH	30	**WEND without WHILE** A WEND statement was encountered without a corresponding WHILE.
WP	69	**Disk write protect** An attempt was made to write a file to a disk which was write-protected by a write protect tab.

Appendix L   DERIVED FUNCTIONS

Functions that are not intrinsic to PX-4 BASIC may be calculated as follows:

Function	HC-40/41 BASIC Equivalent
SECANT	SEC(X) = 1/COS(X)
COSECANT	CSC(X) = 1/SIN(X)
COTANGENT	COT(X) = COS(X)/SIN(X)
INVERSE SINE	ARCSIN(X) = ATN(X/SQR(− X * X + 1))
INVERSE COSINE	ARCCOS(X) = − ATN(X/X * X + 1)) + 1.570796326794897
INVERSE SECANT	ARCSEC(X) = ATN(SQR(X * X − 1)) + (SGN(X) − 1 * 1.570796326794897
INVERSE COSECANT	ARCCSC(X) = ATN(1/SQR(X * X − 1)) + (SGN(X) − 1) * 1.570796326794897
INVERSE COTANGENT	ARCCOT(X) = − ATN(X) + 1.570596326794897
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)) * 2 + 1
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)) * 2 + 1
INVERSE HYPERBOLIC SINE	ARCSONH(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((SQR(− X * X + 1) + 1)/X)
INVERSE HYPERBOLIC COSECANT	ARCCSH(X) = LOG((SGN(X) * SQR(X * X + 1) + 1/X)
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X) = LOG((X + 1)/(X − 1))/2

Appendix M MEMORY MAP

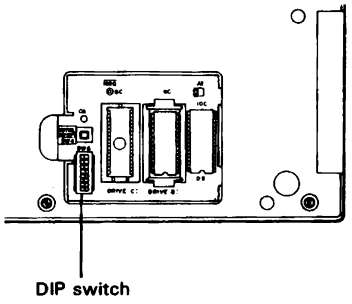


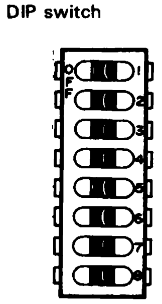
&HXXXX:	Must be specified in the /M option when starting BASIC, or with the CLEAR command. Must be less than &H6000 or &HYYYY, whichever is smaller.
&HYYYY:	This value depends on the sum of the sizes of the RAM disk and user BIOS areas. See APPENDIX G "Machine Language Programs" for further information.

Appendix N   DIP SWITCH SETTING

The DIP swith located inside the ROM capsule compartment is used to select some of system configurations.
To change DIP switch setting:

1) Turn off the power switch.
2) Change the desired setting by a ball-point pen, twizers or small screw driver.
3) Push the reset switch on the right side of the computer.
4) Turn on the power switch.





1. Country selection

International character set can be selected by setting of four switches.
It can be also selected by using CONFIG.COM of CP/M utilities. See Operating Mannal for details.

Keyboard type	SW-1	SW-2	SW-3	SW-4
ASCII	ON	ON	ON	ON
France	OFF	ON	ON	ON
German	ON	OFF	ON	ON
England	OFF	OFF	ON	ON
Denmark	ON	ON	OFF	ON
Sweden	OFF	ON	OFF	ON
Italy	ON	OFF	OFF	ON
Spain	OFF	OFF	OFF	ON
Norway	OFF	ON	ON	OFF

2. LST: device selection

When an external printer or optional Cartridge Printer is used, select appropriate setting according to the device to be used.
When the RS-232C or serial interface is used, confirm the parameters such as baud rate match with those of the device connected. Use CONFIG.COM to select interface parameters.

Device name	SW-5	SW-6
Serial interface	OFF	OFF
RS-232C interface	OFF	ON
Cartridge printer	ON	OFF
Printer interface	ON	ON

Index

A

Abbreviation for PRINT, 3-147
ABS, 3-5
Absolute value, 3-5
Address
- for storage, 3-26
- in memory, 3-199
- redefine starting, 3-45
- starting, 3-45, 3-17
- the highest used by BASIC, 3-26, 3-45

ALARM, 3-6
- setting from BASIC, 3-6
- using wildcards with, 3-6

ALARM$ to obtain ALARM settings, 3-8
AND, 2-14, 2-16
Ambiguous file name, 3-59
Antilogarithms, 3-56, 3-109
Arithmetic operations, 2-8, 2-10
Array
- cancelling definitions, 3-52
- dimensioning, 3-47
- erasing, 3-52
- subscripts, 3-47, 3-135
- variables, 2-7, 3-47, 3-135
- subscripts base of, 3-135
- subscripts minimum value of, 3-135

ASC, 3-9
ASCII
- character set, J-1
- code, 3-22, 3-136, 3-186, Appendix J
- code table, 3-136, Appendix J
- format, 3-100, 3-101, 4-4
- option for saving BASIC programs, 4-2
- string, 3-120, 3-122
- value of the first character of a string, 3-9

ATN, 3-10
Audio cassette, 2-26
- connecting, 2-27
- loading from, 2-28, 3-105
- control, 3-123
- saving onto, 2-27

AUTO, 3-12
Auto line numbering, 3-12
AUTO START, 3-13
Autostarting the PX-4, 1-10, 3-6, 3-13

B

Back spacing, 1-23
BASIC
    command syntax format, 3-2
    editing lines, 1-17, 1-21, 2-1
    ending, 1-7, 1-9, 3-144, 3-189
    enhancements to, i
    entering from CP/M, 1-1
    entering from the MENU, 1-2
    entering with extended format commands, Chapter 1
    EPSON enhanced PX-4, i
    extensions to, i
    features of, i
    free memory available, 3-66
    garbage collection, 3-66
    installing, iii
    Microsoft, i
    program, 2-1
    program areas of, 1-8, 1-11, 3-111, 3-117
    program area selection, 1-8, 1-11, 3-111
    program menu, 1-8, 3-117
    program names, 1-8, 3-103, 3-117, 3-195
    starting, 1-1, 1-7
    terminated, 1-7, 3-144
BEEP, 3-14
BF (bad file mode) error, 3-27
Binary digits, 3-96
BLOAD, 3-15
Box, 3-96
Brackets, 3-3
BSAVE, 3-16
BS key, 1-23
Buffer
    file output, 3-57, 3-68, 3-115, 3-143, 3-158, Chapter 4
    printer output, 3-112
    random file, 3-115, 3-122
    RS-232C receive, 3-108
Buzzer, 2-24
    BEEP, 3-14
    pitch of sound by 3-178
    duration of sound by, 3-14, 3-178
    to "beep", 4-25
    tone of, 3-178
Byte
    (s),free in RS-232C buffer, 3-108
    free in BASIC, 3-66
    free on disk, 3-48

C

Calendar/clock
  date, 3-39
  day of the week, 3-40
  time, 3-194
CALL, 3-17, G-2, G-6
Call
  a machine language subroutine, G-1, G-6
  user defined function, 3-41
Cancel array definitions, 3-52
Change
  filename, 3-125
  microcassette tape, 3-24, 3-124, 3-164
  variables, 3-188
Characters
  special, 3-147
  type declaration, 2-6
  user defined graphics, Appendix F
CDBL, 3-18
CHAIN, 3-19
Chaining and merging BASIC programs, 3-19
Character from ASCII code, 3-22
Checking keyboard input, 3-78, 3-85
CHR$(X), 3-22
CINT, 3-23
CLOCK, *see* calendar, date, day
CLOSE, 3-27
Close all files, 3-26
CLEAR clears variables and memory space, 3-26
Clock setting, 3-194
CLR key, 1-23
CLS clear screen, 3-29
Code
  ASCII, J-1
  Console Escape Sequences, H-1
Cold start, 1-1, 1-3
  parameters, 1-4
Comma, including, 3-81
Commands in BASIC, 3-1
Command level, 1-12, 3-102, 3-118, 4-127
COM0:, 2-23, B-1
COMMON, 3-19, 3-30
Communication protocol, 5-1
Communication trap, 3-128
Comparison of strings, 2-17
Concatenation of strings, 2-17
Conditional branching, 3-75

Constants
  Double precision numerical, 2-5
  Integer, 2-4
  list of, 3-37
  numeric, 2-4
  single precision numeric, 2-5
  string, 2-14
  substitute, 3-37
  types of, 2-14; 3-37
CONT, 3-32
Control characters, H-1
  see also CTRL and the respective keys
Control key, *see* CTRL
Control signals, 5-1
Converted numbers containing random digits, 3-18
Conversion
  in arithmetic and relational operations, 2-10
  of hexadecimal numbers to decimal, 3-73
  of Logs and Antilogs, 3-109
  of line style, 3-96
  of numbers to integers, 3-23
  of numbers to strings, 3-186
  of numeric expressions, 3-18, 3-23
  of numeric values, 3-34, 3-115, 3-122
  of radians to degrees, 3-10,3-43
  of single precision to double precision, 3-18
  string to numbers, 3-36, 3-171, 3-198
  to double precision numbers, 3-18, 3-23
  to single precision numbers, 3-23, 3-34
Coordinates
  absolute, 1-20
  character, 1-19
  graphic, 1-20
  relative, 1-20
COPY, 3-33
COS, 3-33
  incorrect values, 3-33
"CP/M", command line, 1-1, 3-142
  return to, 1-9, 3-189
"CP/M", system prompt, 1-1
"CP/M", warm starts, 3-189
CSNG conversion to single precision numbers, 3-34
CSRLIN returns cursor line, 3-35
CTRL, *see also* Control key and the Operating Manual
CTRL key
  with cursor keys, 1-18
  with HELP key, 1-16
  with PF5 key, 1-15
  with STOP key, 1-16
CTRL + ←, scroll window to left, 1-18

CTRL + →, scroll window to right, 1-18
CTRL + ↑, scroll window upward, 1-18
CTRL + ↓, scroll window downward, 1-18
CTRL + A, move to beginning of logical line, 1-23
CTRL + B, move back one word, 1-23
CTRL + C, halt BASIC program execution, 1-15, 1-22
CTRL + E, erase rest of line, 1-23
CTRL + F, move to following word, 1-23
CTRL + H, move cursor to left, 1-23
CTRL + I, move cursor to next tab position, 1-23
CTRL + K, home cursor, 1-21
CTRL + L, clear screen, 1-21
CTRL + M, carriage return
CTRL + R, insert mode, 1-21
CTRL + S, to pause listing, 1-15
CTRL + X, move to end of line, 1-23
CTRL + Z, erase reset of screen, 1-23
CTRL + SCRN, center cursor, 1-18
Cursor keys, 1-22
Cursor
    center, 1-18
    current position, 3-35
    displaying on screen, 3-107
    file access buffer, 3-57
    moving, 3-107
    record length, 3-58
    switch, 3-107
CVI/CVS/CVD convert strings for random access files, 3-36
    see also MKI$/MKS$/MKD$, Chapter 4

D

DATA statement for storing data, 3-37
Data
    left-justified, 3-115
    OD(out of data) error, K-6
    read 3-37, 3-168
    right-justified, 3-115
    substitute into variables, 3-161, 3-168
    temporary storage of, 3-57
    transfer of, 3-57

DATE$, 3-39
DAY, 3-40
Decimal, 2-4
Declaring types variables, 3-44
Defining functions, 3-45
    numerical variables, 3-44

DEF DBL, 3-44
DEF FN, 3-41
DEF INT, 3-44
DEF SNG, 3-44
DEF USR for call machine language subroutines, 3-45
DEL key, 1-21
DELETE, 3-46
Delimiters
    between items, 3-83, 3-85, 3-100, 3-155, 3-207
    explicit, 3-155, 3-207
Derived functions, L-1
Destroy
    contents of files, 3-27
    variables, 3-26, 3-117, 3-173
Device
    names for file descriptors, 2-19
Difference
    between INPUT# and LINE INPUT# , 3-100
    between International Character Sets, J-2
Dimensioning of arrays, 2-7, 3-47
DIM, 3-47
Directory of disk, 3-59
    printing, 3-60
Direct mode, 1-12
Display data in specified format, 3-148
Display screen
    changing size of, 3-205
    changing width of, 3-205
    drawing graphics on, *see* LINE, PRESET,PSET
    hard copy of, 1-15, 3-33
    locating characters on, 3-107
    output to printer, 1-15, 3-33
Division, 2-27
    Integer, 2-28
/0 (division by zero) error, 2-10
Dot
    coordinates, 1-20, 3-95, 3-157
    display, 3-157
    erase, 3-96
    reset, 3-96, 3-145
    return setting of, 3-141
    segment, 3-95
    set, 3-95, 3-157
    set or reset, 3-157
    turn off, 3-141, 3-145
    turn on, 3-157
Double precision, 2-4
Draw
    lines, 3-95
    rectangle, 3-96

Drive name, 2-19
DSKF, 3-48
Duration
    of sound, 3-178
    of power, 3-144

E

EDIT command, 3-49
EDIT Mode, 1-21, 3-49
    cursor keys in, 1-21
    termination, 1-22, 3-49
Editing BASIC lines, 1-21, 2-1
Editor, 1-21
END, 3-50
End
    of file, 3-51
    of program execution, 3-50
EOF, 3-51
ERASE arrays, 3-52
Erase
    dots, 3-95, 3-145
    lines, 3-95
    variables, *see* CLEAR
ERL, 3-53
ERR, 3-55
ERROR, 2-29, 3-54
Error(s)
    codes, 3-54, Appendix K
    in direct mode, 3-53
    interrupted by, 3-32
    messages, 2-29, Appendix K
    numerical table of codes, K-1
    processing routine, 3-53, 3-55, 3-129, 3-169
    recovery procedures, 3-53, 3-54, 3-55, 3-129
    rounding, 3-24
    simulation of, 3-54
    syntax, 3-129
    trapping, 3-53, 3-54, 3-55, 3-129
    user defined, 3-55
Errors
    interrupted by, 3-32
    rounding, 3-24
ESC key, *see also* Operating Manual
    sequences, H-1
Execution
    interrupted, 3-32
    stopping, 3-129
    resume, 3-32, 3-165

EXP, 3-56
Exponentiation, 3-56
Expressions, 3-4

F

FIELD, 3-57
FILES obtaining the disk directory, 3-59
Files, 2-19, 4-1
File(s), 2-4, Chapter 4
    closing, 3-27
    creating, 4-1
    data, 3-103
    disk, 3-108, 4-1
    disk device file, 3-133, Chapter 4
    keeping open while loading a new program, 3-103
    microcassette, 2-24
    name extension, 3-103, 3-118
    number, 2-21, 3-106, 3-108, 3-155, 5-3
    program, 4-1
    random access, 4-13
    sequential, 4-7
FIX, 3-61
FOR .....NEXT, 3-63
Format for communications, 5-3
Format notation of commands
    J or K, 3-4
    X or Y, 3-4
    X$ or Y$, 3-4
    abbreviations in, 3-4
    angle brackets, 3-2
    format notation, 3-2
    full stops in, 3-4
    options in, 3-3
    quotation mark, 3-2
    reserved words, 3-2, Appendix I
    square brackets, 3-2
    vertical bars, 3-4
Formatting characters,in international character set, 3-136, 3-154
Free memory available, 3-66
FRE, 3-66
Functions, 2-17
    derived, L-1
    formula for obtaining, 3-43
    inverse trigonometric, 3-43
    user-defined, 3-41

G

Garbage collection, 3-66
GET, 3-68
GOSUB....RETURN, 3-70
GOTO or GO TO, 3-72
Graphic
    commands, see draw, LINE, PSET, PRESET
    screen coordinates, 1-20, 3-141

H

Hexadecimal, 2-4, 3-73
HEX$, 3-73
Highest precision, 2-5
HOME key, 1-21

I

IF....THEN .... ELSE, 3-53, 3-55, 3-75
IF....GOTO, 3-75
IMP, 3-80
Indirect mode, 1-12
INKEY$, 3-78
INP obtaining data from the Input Port, 3-80
INPUT, 3-81
INPUT#, 3-83
INPUT$, 3-85
Input/Output devices, *see* I/O devices
Input
    all characters, 3-100
    fixed number of characters, 3-85
INS key, 1-21
Inserting characters, 1-21
INSTR, 3-87
INT, 3-23, 3-88
Integer
    errors, 3-24
    expressions, 3-4, 3-87
International character sets, 3-136, 3-154, *see also* Operating Manual
    Denmark, 3-136
    England, 3-136
    France, 3-136
    Germany, 3-136
    Italy, 3-136
    Norway, 3-136
    Spain, 3-136

Sweden, 3-136
U.S.A., 3-136
International currency symbols, 3-137
International formatting characters, 3-154
I/O device support, 3-23
I/O devices, 2-23
and BASIC commands statements and functions, B-1
as file descriptors, 2-19, 5-1, A-1
input/output modes for, 3-133
range of, 2-19, 2-23
RAM disk, 2-19, 2-23, 3-103
RS-232C interface, 2-19, 5-1
summary table of, 5-1, A-1
I/O operation
halting, 1-16
of Micro cassette, 2-4, *see also* Operating Manual
Interruption of BASIC program, 1-15

K

KEY command to set PF keys, 3-89
KEY LIST/KEY LLIST to list PF, 3-89
Keyboard, 1-13
ASCII, 4-11
buffer, 3-78, 3-85
checking for input, 3-78, 3-85
defining function keys, 3-89
item, 1-13, 4-11
modes, 4-12
Programmable Function Keys, 1-15, 3-89
KILL, 3-91
KYBD:, 3-23, A-1

L

Last Reference Pointer, 3-95, 3-145, 3-157
LEFT$, 3-92
LEN, 3-93
LET, 3-94
LINE, 3-95
Line
complete, 3-96
dashed, 3-96
draw, 3-95
feed, 3-208
style, 3-96
LINE INPUT, 3-98
LINE INPUT#, 3-100

Line numbers, *see* program line numbers
LIST, 3-101
LLIST, 3-102
LOAD, 3-103
LOAD?, 3-105
LOC, 3-106
LOCATE, 3-35, 3-107
LOF, 3-108
LOG, 3-109
logarithm (LOG), 3-109
Logical operation, 2-10, 2-13
LOGIN, 3-111
Logging in to BASIC program areas, 1-8, 3-111
Loops, 3-63, 3-203
LPOS, 3-112
LPRINT / LPRINT USING, 3-113
LRP, 3-95, 3-145, 3-157
LSET, 3-115

M

Machine language
    program area, 3-26
    programs memory for, 3-26
    programs starting address, 3-45
    programs used as subroutines by BASIC, 3-45
    programs, user-written, 3-45
    programs, writing to memory, 3-142
    subroutine, calling, 3-17
    subroutine, parameters for, 3-17
    subroutine, starting address of, 3-17, 3-26
    see also USR
Memory
    buffer for random access files, 3-57
    for machine language programs, 3-26
    location, 3-26, 3-140
    map, 3-141, M-1
    OM(out of memory) error, K-6
    write data into, 3-141
MENU, 3-117, 4-17
MENU
    entering BASIC from, 1-2
    setting up to run BASIC programs, 1-4, Operating Manual
MERGE, 3-118
Messages
    "?Redo from start", 3-81
    AC (Tape access error), 3-124
    BF (Bad file mode), 3-27, 3-118
    /0 (Division by zero), 2-10

DD (Duplicate definition), 3-135
FC (Illegal function call), 3-87, 3-111, 3-126, 3-129, 3-139
FE (File already exists), 3-125
FO (Field overflow), 3-57, 3-159
NE (File not exist), 3-59, 3-125
OD (Out of data), 3-161
RW (RESUME without error), 3-169
TM(Type mismatch error), 3-44, 3-188
UL(Undefined line number), 3-72
"Undefined line xxxxx in yyyyy", 3-165
MICROCASSETTE drive (Drive H:), 2-24
counter, 3-193, 3-206
files on, 2-24
MOUNT, 3-124, 3-164
open mode for, 4-140
play mode, 3-206
REMOVE, 3-124, 3-164
removing tape, 3-164
restriction on use, 2-24
sound output to speaker, 3-206
stop/non-stop modes, 2-25
tape directory, 2-24, 3-124, 3-164
verify/non-verify modes, 2-25
WIND, 3-206
write directory to, 3-164
MID$, 3-120
MKI$/MKS$/MKD$, 3-122, *see also* CVI/CVS/CVD
MOD, 2-10
Mode
continue, 3-144
trace, 3-196
MOTOR, control audio cassette, 3-123
MOUNT, 3-124, *see also* REMOVE

N

NAME, 3-125
Natural logarithms, 3-109
Nested loops, 3-63, 3-64
NEW, 3-126
Non-stop mode, 2-25
Non Tracking mode, 1-18
horizontally, 1-18
Non-verify mode, 2-25
NOT, 2-13, 2-16
Null string, 3-78, 3-92, 3-170
Numbers
hexadecimal, 2-4, 3-73, 3-198
octal, 2-4, 3-127, 3-198

Numeric
 expressions, 3-4
 constants, 2-4
 value rounding, 3-23
 value whole number, 3-23

O

Octal, 2-4, 3-127
OCT$, 3-127
ON COM(n) GOSUB....RETURN, 3-128
ON ERROR GOTO, 3-129
ON GOSUB/ON GOTO, 3-130
OPEN, 3-133
OPEN "I", 2-25
OPEN "O", 2-25
OPEN "R", 2-25
Operations, 2-10
OPTION BASE, 3-26, 3-135
OPTION COUNTRY, 3-136
OPTION CURRENCY, 3-137
Options
 ASCII to save BASIC programs as text files, 3-174
 asterisks as wildcards, 3-6
 communications "blpscxh", 5-4
 wildcard, 3-6, 3-59
OR, 2-14, 2-16
OUT, 3-138
Output
 data to a printer, 3-33, 3-112, 3-113
 program in memory to devices, 3-174
 screen to printer 3-33

P

PAUSE key, 1-16
PCOPY, 3-139
PEEK, 3-140
PF keys, 1-3, 1-15, 3-89
 setting, *see* KEY
Physical screen, 1-17
PI (ratio of circumference to diameter of a circle), 3-10
POINT, 3-141
Points of graphic screen, *see* Dot, LINE, PRESET, PSET
Pointer
 last reference, 3-95
 print head, 3-112, 3-143
POKE, 3-142

POS, 3-143
Position
    of cursor, 3-107, 3-143
    of file output buffer, 3-143
    of print head, 3-143
POWER, 4-144
    ALARM, 3-6, 3-144
Power
    automatically turn off, 3-144
    duration, 3-144
    on and off, 1-9, 3-6, 3-13, 3-144
    switch, 1-9, 3-144
    turn back on, 1-9, 3-144
    turn off, 1-9, 3-144
Precision, single and double, 2-5, 3-36, 3-122, 4-14, 4-16
PRESET, 3-145
PRINT, 3-146
PRINT, abbreviation for, 3-147
PRINT#, 3-155
PRINT USING, 3-148
PRINT# USING, 3-155
Printer
    LPOS, 3-112
    control characters of, 3-112
    column of, 3-190
    list to, 3-102
    output mode for, 3-113
    output to, 3-113
    output screen to, 3-33
    print head pointer, 3-112
    width of, 3-190
Program
    commenting, 3-163
    delete from memory, 3-126
    merge, 3-118
    protect, 3-126, 3-174
    transfer between program areas, 3-139
Programmable Function Keys, 3-89
Program areas, 1-5, 1-11
    currently logged in, 3-117
    executed in, 3-13
    LOGIN to, 1-5, 1-11, 3-111
    no. 3-111, 3-117
    selecting, 1-11, 3-111, 3-117
    titling, 1-8, 3-195
    transfer between, 3-111, 3-139
Program
    error trapping, 3-54, 3-129
    with ASCII codes, 3-22
    branched, 3-70, 3-72, 3-128, 3-129, 3-130

continuing, 3-33, 3-169
resuming, 3-54, 3-72, 4-32
stopping, 1-15, 3-50, 3-184
using the same data with different, 3-30
Program lines, 1-12, 2-1
maximum length, 2-1
Program line numbers, 2-1
deleting, 3-46, 3-126
listing, 3-101, 3-102
merging, 3-19, 3-118
non-existent, K-9
renumbering, 3-165
resume execution at a particular, 3-33, 3-169
subroutines, 3-70
Program listings,
pausing, 1-15
PSET, 3-157
PUT, 3-158, 4-64

Q

Question mark abbreviation for PRINT, 3-147
Quotation marks character, 3-83, 3-85, 3-98, 3-100

R

RAM cartridge, 2-26, 3-103
RAM disk, 3-103
Random Access
file buffer, 3-57, 3-68, 3-115, 3-158
file, field, 3-57
record length, 3-57
RANDOMIZE, 3-160, 3-171
Random numbers in fixed ranges, 3-171
READ, 3-161
REM, 3-163
REMOTE terminal of audio cassette, 2-27, 3-123
REMOVE, 3-164, *see also* MOUNT
RENUM, 3-165
Replace string, 3-120
Reset screen size, 3-117, 3-205
RESET, 3-167
RESTORE, 3-168
RESUME, 3-169
Resuming an interrupted program, 3-32
RETURN key, 1-12, 3-12
RIGHT$, 3-170

RND, 3-171
ROM Capsules, iii
ROM cartridge, 2-26, 3-103
RS-232C, 3-103, 3-173
    merging program with, 3-118
RUN, 3-173
RUN and GOSUB, 3-70

S

SAVE, 3-174
    in ASCII format, 3-174
    with protect option, 3-174
SCREEN, 3-175
Screen
    character coordinates, 1-18
    graphic coordinates, 1-20
    dump to printer, 1-15, 3-32
Screen editor, 1-21
Screen modes, 1-18
SCRN: 2-23, B-1
SCRN key, 1-18, *see also* the Operating Manual
SCRN DUMP key (CTRL-PF5), 1-15
Scrolling, 1-18
    control keys, 1-18
    modes, 1-18
Search for a substring, 3-87
Sequential
    access devices, 2-24
    input mode, 3-85
    output file, 3-155
    output mode, 3-133
    open, 3-85, 3-133, 3-155
    read data into variables from file, 3-85, 3-98
    read items from, 3-85
SGN, 3-176
SIN, 3-177
Size
    of BASIC memory, 1-2
    of CP/M, 1-2
    of a file, 3-108
    of virtual screen, 3-205
    record, 3-108
    single precision, 2-4
    stack area, 3-26
SOUND, 3-178
SPACE$, 3-180
Space on drive, 3-48
SPC, 3-181

SQR, 3-182
Stack
 clearing, 3-26
 space, 3-26
 stack area size, 3-26
STAT, 3-183
Statements, 3-1
STOP, 3-184
STOP key, 1-15, 1-16, 3-185
 CTRL and STOP key, 1-16
 disabling, 3-185
STOP KEY, 3-185
Stop mode, 2-25, 2-26
STR$, 3-186
String
 2-byte, 4-byte or 8-byte, 3-36, 3-122
 alarm message, 3-6
 auto start, 3-6
 character, 3-85
 comparisons, 2-17
 concatenation, 2-17
 expressions, 3-4
 first occurrence of, 3-87
 format, 3-148 to 3-154
 joining, 2-17
 justify, 3-116, 3-148
 length of, 3-93
 literals, 3-87
 manipulation, *see* LEN, LEFT$, MID$, RIGHT$
 null, 3-13, 3-87
 numeric to string, 3-186
 operations, 2-17
 prompt, 3-98
 searching, 3-87
 string to numeric, 3-198
 substitute in a, 3-120
 values, 3-36
 variables, 3-18
STRING$, 3-187
Suppress carriage return after INPUT, 3-81
Subroutines, 3-70
SWAP, 3-188
SYSTEM, 1-9, 3-189
System Display, 1-16, 3-6
System variables, 3-39, 3-40, 3-194

T

TAB, 3-190
TAB key, 1-23
TAN, 3-192
TAPCNT, 3-193
TIME alarm, 3-6
    altering, 3-194
    clear alarm or wake, 3-6
    setting, 3-194
    system variable of, 3-6, 3-194
TIME$, 2-6, 3-194
TITLE, 3-195
Trigonometric functions, 3-43, *see also* ATN, COS, SIN, TAN
    deriving, Appendix L
TRON/TROFF, 3-196
Tracking mode, 1-18
Trailing spaces, 3-37
True and False, 2-13, 2-16, 3-75
Type of variables, 3-44

U

Undefined line number, 3-72
User defined
    characters, F-1
    error code, 3-54
    function, 3-41
USR, 3-197

V

VAL, 3-73, 3-198
Variable name(s)
    type declaration of, 3-44
    array, 2-22, 3-30
    clear all, 3-26
    memory limit, 3-26
Variables
    passing between programs, 3-30
    resetting all, 3-26
    system, 3-6, 3-39, 3-40, 3-194
    numerical type declaration characters, 2-6
    type declaration of, 2-6, 3-44
    types of, 2-6, 3-37, 3-44
    wildcard, 3-6
VARPTR, 3-199

Verify-mode, 2-25
Virtual screen, 1-17
Virtual Screen Window, 1-17

W

WAIT, 3-202
WAKE from ALARM setting, 3-6
Warm start, 1-7
WHILE...WEND, 3-203
WIDTH, 1-19, 3-205
WIND ON, 3-206
    OFF, 3-206
Window, 1-17
WRITE, 3-207, B-1
WRITE#, 3-208

X

XOR, 2-14, 2-16

EPSON OVERSEAS MARKETING LOCATIONS

EPSON AMERICA, INC.
2780 Lomita Blvd., Torrance, Calif. 90505,
U.S.A.
Phone: (213)539-9140
Telex: 182412

EPSON DEUTSCHLAND GmbH
Zulpicher Strasse 6, 4000 Düsseldorf 11,
F.R. Germany
Phone: (0211)56030
Telex: 8584786

EPSON UK LTD.
Dorland House, 388 High Road, Wembley,
Middlesex, HA9 6UH, U.K.
Phone: (01)902-8892
Telex: 8814169

EPSON FRANCE S.A.
55, Rue Deguingand, 92300, Levallois-Perret,
France
Phone: (1)739-6770
Telex: 614202

EPSON AUSTRALIA PTY. LTD.
Unit 3, 17 Rodborough Road, Frenchs Forest,
NSW 2086, Australia
Phone: (02)452-5222
Telex: 75052

EPSON ELECTRONICS (SINGAPORE) PTE. LTD.
No. 1 Maritime Square, #02-19, World Trade
Centre Singapore 0409
Phone: 2786071/2
Telex: 39536

EPSON ELECTRONICS TRADING LTD.
30/F, Far East Finance Centre
Harcourt Road, Central, Hong Kong
Phone: 5-282555
Telex: 65542

EPSON ELECTRONICS TRADING LTD. (TAIWAN BRANCH)
1, 8F K.Y. Wealthy Bldg. 206, Nanking, E. Road,
Sec. 2, Taipei, Taiwan R.O.C.
Phone: 536-4339; 536-3567
Telex: 24444

EPSON CORPORATION
80 Hirooka, Shiojiri-shi, Nagano 399-07
Japan
Phone: (0263)52-2552
Telex: 3342-214