6.3 Keyboard

PX-8 BASIC also allows the keyboard to be handled as a sequential access input device. When the keyboard is opened as a file, data input is assigned to variables using INPUT#, LINE INPUT# and INPUT$ (X,<file no.>) instead of the corresponding dedicated keyboard input statements. This makes it possible to use common routines for input of data from the keyboard, disk device files and the RS-232C interface.

The device name used to OPEN the keyboard as a device file is "KYBD:".

(1) Statements
Statements which can be used for input from the keyboard when it is handled as a device file are as follows:

CLOSE, INPUT#, INPUT$(X,<file no.>),
LINE INPUT#, LOAD, OPEN "I"

(2) Errors
A "Bad file descriptor" error will occur if an attempt is made to open the keyboard in the "O" mode.

Appendix A ERROR CODES AND ERROR MESSAGES

When an error occurs in a BASIC program, it is detected by the interpreter and a message is printed. In most cases the error stops the program and will not allow it to continue. BASIC will return to the direct mode and present the error message. It will not always be obvious what exactly has caused the error. It may be something as simple as a mistyped command which BASIC does not recognise, an error of logic or any one of a series of programming faults. This appendix is an attempt to help the user/programmer to find out what exactly he has done wrong. It is not easy to cover each and every cause of an error, because some errors are particular to the logic of a program and simply cannot be predicted. However, many are due to definite reasons, and these are described below.

Each error has a code associated with it, which is useful for trapping errors and also simulating them. See ERROR, ON..ERROR, ERR and ERL in Chapter 4 for details of their use. A list of errors in numerical order is given at the end of this section. However, as the error is normally encountered as a message, the details of each error are given in alphabetical order. The number at the left of each error is the error code.

54 Bad file mode

A statement or function was used with a file of the wrong type.

Possible causes:
(i) An attempt was made to use PUT, GET or LOF with a sequential file.
(ii) A non BASIC program file was specified in a LOAD command.
(iii) A file mode other than I, O, or R was specified in an OPEN statement.
(iv) An attempt was made to MERGE a file that was not saved in ASCII format.

64 Bad file descriptor

An illegal file name was specified in a LOAD, SAVE or KILL command or an OPEN statement (for example, a file name with too many characters).

52 Bad file number

A statement or command references a file that has not been opened, or the file number specified in an OPEN statement is outside of the range of file numbers that was specified when BASIC was started.

63 Bad record number

The record number specified in a PUT or GET statement was either zero or greater than the maximum allowed.

17 Can't continue

An attempt was made to resume execution of a program when continuation was not possible.

Possible causes:
(i) Program execution was terminated due to an error.
(ii) The program was modified while execution was suspended.
(iii) The STOP key was pressed during execution of an INPUT statement.
(iv) The program had not yet been executed.

28 Communication buffer overflow

The receive buffer overflowed during receipt of data via the RS-232C interface. This error is likely to occur when the speed with which receive processing is performed is lower than that at which data is being received, but is unlikely if the communication rate is set to 1200 bps or less.

25 Device fault

The level of the signal on the DSR or DCD line became low during input from the RS-232C interface after the DSR receive check or DCD check had been set to ON (by option "c" of the communications format specification in the OPEN"I" statement executed to open the interface).

57 Device I/O error

An error occurred involving input or output to a peripheral device.

Possible causes:
(i) An I/O error occurred during access to a disk device. This is a fatal error; that is, one from which the operating system cannot recover.
(ii) A parity error, overrun error or framing 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.
(iii) The printer power was off or a fault occurred when data was output to the printer.

24 Device time out

Possible causes:

(i) Transmission via the RS-232C interface was not enabled within a certain period of time after an OPEN"O" statement was executed with the DSR send check set to ON by option "c" of the communications format specification.
(ii) The STOP key was pressed while output to the RS-232C interface was being deferred for some reason.
(iii) The DSR or DCD line did not become high within a certain period of time after an OPEN"I" statement was executed with the DSR receive check or DCD check set to ON by option "c" of the communications format specification.
(iv) The printer was not ready when output to the printer was attempted.

68 Device unavailable

An attempt was made to access a drive which did not contain a floppy disk or the RS-232C interface was not available.

66 Direct statement in file

A program line without a line number was encountered during execution of a LOAD or MERGE command, or an attempt was made to LOAD a data file or machine language program.

61 Disk full

Either the disk directory or the disk itself has no space left.

7Ø Disk read error

An error occurred while data was being read from a disk device.

71 Disk write error

An error occurred while data was being written to a disk device.

69 Disk write protect

Possible causes:
(i) An attempt was made to write data to a disk which is protected by a write protect tab.
(ii) An attempt was made to write data to a disk drive without executing the RESET command after replacing the disk in that drive.
(iii) An attempt was made to write data to a ROM capsule.

11 Division by zero

An operation was encountered which included division by zero.

Possible causes:
(i) Zero was used as a divisor possibly because a variable or expression was zero at that point in the program.
(ii) Division was attempted using an undefined variable as a divisor.

10 Duplicate Definition

A variable array was defined more than once.

Possible causes:
(i) A second DIM statement was executed for an array without erasing that array with an ERASE statement.
(ii) An undeclared array was used, then an attempt was made to re-dimension that array with a DIM statement.
(iii) The OPTION BASE statement was executed more than once, or was executed after an array had already been dimensioned, either by a DIM statement or implicitly by assignment of a value to a variable with a subscripted name.

50 FIELD overflow

A FIELD statement attempted to allocate more bytes in a random file buffer than were specified for that buffer when the file was opened.

58 File already exists

The new file name specified in a NAME statement is already being used with another file on the disk.

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.

53 File not found

The file name specified in a LOAD, KILL, NAME or OPEN statement does not exist on the disk in the accessed drive.

26 FOR without NEXT

A FOR statement was encountered without a corresponding NEXT.

12 Illegal direct

A statement that is illegal in the direct mode (such as DEF FN) was entered as a direct mode command.

5 Illegal function call

A statement or function was incorrectly specified.

Possible causes:

(i) Specification of a negative number or a number which is too large as an array variable subscript.
(ii) Specification of zero or a negative number as the argument in the LOG function.
(iii) Specification of a negative number as the argument of the SQR function.
(iv) Specification of a non-integer exponent with a negative mantissa.
(v) A call to a USR function for which the starting address has not yet been defined.
(vi) An incorrectly specified argument in any of the following functions or statements:
ALARM, ALARM$, ASC, CSRLIN, INP, INSTR, LEFT$, LOCATE, MID$, ON...GOSUB, ON...GOTO, OUT, PEEK, POKE, POWER, PRESET, PSET, RIGHT$, SCREEN, SPACE$, SPC, STRING$, TAB, VARPTR, WAIT, WIND.
(vii) Specification of a non-existent line number in a DELETE statement.
(viii) Attempting to erase a non-existent variable array with an ERASE statement.
(ix) Specification of a number other than 1 to 5 as the parameter of a LOGIN, PCOPY or STAT statement.
(x) Execution of a RENUM command with parameters which do not conform to the rules for specifying such commands.
(xi) Specification of an undefined array variable or a variable whose value has not yet been defined in a SWAP statement.
(xii) Execution of the EDIT command when the virtual screen window was less than 38 columns wide.
(xiii) Specification of a number other than 1 to 10 as the parameter of a KEY command.

62 Input past end

Possible causes:
(i) An INPUT statement was executed for a file which was empty or one from which all data had been read. To avoid this error, use the EOF function to detect the end of the file.

(ii) The STOP key was pressed while input from the RS-232C interface was pending with INPUT#, INPUT$ or a similar command.

51 Internal error

An internal malfunction occured in BASIC.

23 Line buffer overflow

An attempt was made to input a line that contains too many characters.

22 Missing operand

Possible causes:
(i) An expression contains an operator without a following operand.
(ii) A required parameter is missing from the AUTO START or LOCATE commands.

1 NEXT without FOR

A NEXT statement was encountered without a corresponding FOR statement.
Possible causes:
(i) Improperly nested FOR/NEXT loops or variables specified in the wrong order in a common NEXT statement for loops that end at the same point.
(ii) The variable in a NEXT statement does not correspond to any previously executed FOR statement variable.
(iii) More than one NEXT statement was specified for one FOR statement.
(iv) Execution branched to a point within a FOR/NEXT loop from elsewhere in the program.

19 No RESUME

No RESUME statement was included in an error processing routine. All error processing routines must conclude with an END or RESUME statement.

4 Out of DATA

A READ statement was executed when there was no unread data remaining in the program's DATA statements.

Possible causes:
(i) Insufficient number of data items in DATA statement(s).
(ii) Incorrect specification of a RESTORE statement.
(iii) Incorrect delimiting punctuation used in a DATA statement.

7 Out of memory

Memory available is insufficient for processing required.

Possible causes:
(i) Program is too long.
(ii) The program uses too many variables.
(iii) The subscript range specified in a DIM statement is too large.
(iv) An expression has too many levels of parentheses.
(v) FOR...NEXT loops or GOSUB...RETURN sequences are nested to too many levels.
(vi) The stack area size or machine language area specified in a CLEAR statement is too large.
(vii) Insufficient memory was available to allow a program to be copied with the PCOPY statement.

14 Out of string space

Insufficient memory space is available for storage of characters in string variables.

6 Overflow

A numeric value was encountered whose magnitude exceeds the limits prescribed by PX-8 BASIC. If underflow occurs, zero is assumed and execution continues without error.

Possible causes:
(i) The result of an integer calculation was outside the range from -32768 to 32767.
(ii) The result of a single or double precision number calculation was outside the range from 1.70141E38 to −1.70141E38.
(iii) One of the operands of a logical operation was not in the range from −32768 to 32767.
(iv) The argument specified for the CINT function or POINT statement was outside the range from −32768 to 32767.
(v) The argument specified for the HEX$ or OCT$ function was outside the range from −32768 to 65535.
(vi) The number specified as one of the parameters of the LOCATE or WIND statements was outside the prescribed range.

20 RESUME without error

A RESUME statement was encountered outside an error processing routine.

Possible causes:
(i) Transfer of execution to an error processing routine by a GOTO or GOSUB statement.
(ii) Lack of an END statement at the end of the main routine to keep execution from moving into an error processing routine.

3 RETURN without GOSUB

A RETURN statement was encountered which did not correspond to a previously executed GOSUB statement.

Possible causes:
(i) Execution was transferred to a subroutine by a GOTO statement.
(ii) The line number specified in a RUN command was a line in a subroutine.
(iii) No END statement was included following a main routine to keep execution from moving into a subroutine.

16 String formula too complex

The complexity of a string operation is too great.

15 String too long

An attempt was made to create a string whose length exceeds 255 characters.

9 Subscript out of range

The subscript specified in a statement referencing an array element is outside the range permitted for that array.

Possible causes:
(i) Subscript specified was greater than the maximum specified in the DIM statement defining that array.
(ii) Wrong number of subscripts specified in a statement referencing an array variable.
(iii) A subscript greater than 10 was used without executing a DIM statement to define that array.
(iv) Zero was used as a subscript after executing OPTION BASE 1.

2 Syntax error

A statement does not conform to the syntax rules of PX-8 BASIC.

Possible causes:
(i) A space was not left between a command and a parameter, e.g. LIST10.
(ii) Incorrectly typed keywords.

(iii) Unmatched parentheses.
(iv) Wrong delimiting punctuation (commas, full stops, colons or semicolons) used between statements, expressions or arguments.
(v) Variable name beginning with a character other than a letter.
(vi) Keyword used as the first letters of a variable name.
(vii) Wrong number or type of arguments specified in a function or statement.
(viii)Type of value included in a DATA statement did not match the corresponding variable in the list of variables specified in a READ statement.

72 Tape access error

Possible causes:
(i) An attempt was made to access an access-inhibited microcassette file.
(ii) An attempt was made to mount a tape without executing the REMOVE command to unmount the previous tape.
(iii) The REMOVE command was executed while the tape in the microcassette drive was in the unmounted condition.
(iv) An attempt was made to change the setting of the tape counter while the tape in the microcassette drive was in the mounted condition.

67 Too many files

An attempt was made to create a new disk file after all directory entries were full.

13 Type mismatch

A string expression was used where a numeric expression is required, or vice versa.

Possible causes:
(i) An attempt was made to assign a numeric value to a string variable.
(ii) An attempt was made to assign a string to a numeric variable.
(iii) The wrong type of value was specified as the argument of a function.

8 Undefined line number

A non-existent line number was specified in one of the following commands or statements — EDIT, GOTO, GOSUB, RESTORE, RUN, RENUM — or when attempting to delete a non-existent line by typing a number and pressing the RETURN key.

18 Undefined user function

A call was made to an undefined user function.

Possible causes:

(i) The letters FN were used at the beginning of a variable name.
(ii) The function name was specified incorrectly in the DEF FN statement or when the function was called.
(iii) The user function was called before the corresponding DEF FN statement was executed.

21 Unprintable error

No error message has been assigned to the error condition which exists. This message is also issued for error codes 27, 31-49, 56, 59, 60, 65 and 73-255, usually due to execution of an ERROR statement specifying one of these codes.

3Ø WEND without WHILE

WEND statement was encountered without a corresponding WHILE.

29 WHILE without WEND

A WHILE statement was encountered without a corresponding WEND.

TABLE OF ERROR CODES AND ERROR MESSAGES

1	NEXT without FOR
2	Syntax error
3	RETURN without GOSUB
4	Out of DATA
5	Illegal function call
6	Overflow
7	Out of memory
8	Undefined line number
9	Subscript out of range
10	Duplicate Definition
11	Division by zero
12	Illegal direct
13	Type mismatch
14	Out of string space
15	String too long
16	String formula too complex

17	Can't continue
18	Undefined user function
19	No RESUME
20	RESUME without error
21	Unprintable error
22	Missing operand
23	Line buffer overflow
24	Device time out
25	Device fault
26	FOR without NEXT
28	Communication buffer overflow
29	WHILE without WEND
30	WEND without WHILE
50	FIELD overflow
51	Internal error
52	Bad file number
53	File not found
54	Bad file mode
55	File already open
57	Device I/O error
58	File already exists
61	Disk full
62	Input past end
63	Bad record number
64	Bad file descriptor
6	Direct statement in file
67	Too many files
68	Device unavailable
69	Disk write protect
70	Disk read error
71	Disk write error
72	Tape access error

Appendix B TABLE OF RESERVED WORDS

ABS
ALARM
ALARM$
AND
ASC
ATN
AUTO
BEEP
CALL
CDBL
CHAIN
CHR$
CINT
CLEAR
CLOSE
CLS
COMMON
CONT
COPY
COS
CSNG
CSRLIN
CVD
CVI
CVS
DATA
DATE
DATE$
DAY
DEF
DEFDBL
DEFINT
DEFSNG
DEFSTR
DELETE
DIM
DSKF
EDIT
ELSE
END
EOF
EQV
ERASE
ERL
ERR
ERROR
EXP
FIELD
FILES
FIX
FN
FOR
FRE
GET
GO
GOSUB
GOTO
HEX$
IF
IMP
INKEY$
INP
INPUT
INPUT #
INPUT$
INSTR
INT
KEY
KILL
LEFT$
LEN
LET
LINE
LIST
LLIST
LOAD
LOC
LOCATE
LOF
LOG
LOGIN
LPOS
LPRINT
LSET
MENU
MERGE
MID$
MKD$
MKI$
MKS$
MOD
MOUNT
NAME
NEW
NEXT
NOT
OCT$
OFF
ON
OPEN
OPTION
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$
SPC
SQR
STAT
STEP
STOP
STR$
STRING$
SUB
SWAP
SYSTEM
TAB
TAN
TAPCNT
THEN
TIME
TIME$
TITLE
TO
TROFF
TRON
USING
USR
VAL
VARPTR
WAIT
WEND
WHILE
WIDTH
WIND
WRITE
WRITE #
XOR

Appendix C PX-8 BASIC CONSOLE ESCAPE SEQUENCES

Whereas BASIC as a high level language has a large number of commands and functions, it is also possible to print sequences of characters which will allow further or additional commands which affect output to the screen. This appendix deals with the use of these commands. Some of them are not additonal to BASIC but duplicate BASIC commands in a way which can make programming easier for advanced programmers in some circumstances.

The sequences involve the printing of the ESCAPE character, ASCII code 27 decimal (1B in hexadecimal), followed by one or more characters, the values of which determine the command to be carried out. In the remainder of this appendix the ESCAPE character is denoted by the letters "ESC". The User's Manual contains further information on using the sequences under CP/M or in machine code programs. Not all the commands are supported in BASIC, for example because they interact with the screen editor.

The following table lists the character sequences for the commands alphabetically to make them easy to find. Notes on the use of the commands and parameters are given in numerical order following the table. The numerical values are given in decimal notation in the table and headings.

Control Code	Function
ESC "%"	Access CGROM directly
ESC 243	Arrow key code
ESC 246	Buffer clear key
ESC "C"	Character table
ESC 246	Clear keyboard buffer
ESC "*"	Clear screen
ESC 245	CTRL key code
ESC 215	Cursor find
ESC 243	Cursor key code
ESC "="	Cursor position set
ESC 214	Cursor type select
ESC "P"	Dump screen
ESC "T"	Erase to end of line
ESC "Y"	Erase to end of screen
ESC 210	Display characters on real screen
ESC 208	Display mode set
ESC 198	Dot line write
ESC 213	End locate
ESC 215	Find cursor
ESC 177	Function key code returned
ESC 176	Function key string returned
ESC 211	Function key display select
ESC "C"	International Character Sets
ESC 161	INS LED off
ESC 160	INS LED on
ESC 242	Key repeat interval time
ESC 240	Key repeat on/off
ESC 241	Key repeat start time
ESC 244	Key code scroll
ESC 247	Key shift set
ESC "T"	Line erase
ESC 198	Line dot draw
ESC 213	Locate end of screen
ESC 212	Locate top of screen

Control Code	Function
ESC 125	Non secret
ESC 165	NUM LED off
ESC 164	NUM LED on
ESC 199	PSET/PRESET
ESC 242	Repeat interval time for keys
ESC 240	Repeat on/off for keys
ESC 241	Repeat start time for keys
ESC "*"	Screen clear
ESC 209	Screen display select
ESC "P"	Screen dump
ESC 213	Screen window end
ESC "Y"	Screen erase
ESC 212	Screen window top
ESC 145	Scroll down
ESC 244	Scroll key code

Control Code	Function
ESC 148	Scroll step
ESC 149	Scroll mode
ESC 144	Scroll up
ESC 151	Screen down n lines
ESC 150	Scroll up n lines
ESC 123	Secret mode
ESC 125	Secret mode cancel
ESC 214	Select cursor type
ESC 209	Select virtual screen
ESC 211	Select function key display
ESC 247	Shift key set
ESC 163	CAPS LED off
ESC 162	CAPS LED on
ESC 212	Top locate
ESC 224	User defined character

Use of the ESCAPE Code control sequences

ESC "%"

Reads the character corresponding to the specified code from the character generator ROM and displays it at the present cursor position in the currently selected screen (in the virtual screen for modes 0, 1, and 2, and in the real screen for mode 3). The sequence is as follows:

PRINT CHR$(27);"%";CHR$(n)

The value of n is the ASCII code corresponding to the character to be displayed.

ESC "*"

Clears the currently selected screen and moves the cursor to the home position.

ESC "="

Moves the cursor to the specified position in the screen being written. In the tracking mode, the screen window is moved so that the cursor is positioned at screen centre if the position specified is outside the screen window. The tracking mode is turned on and off by pressing the SHIFT and SCRN keys together. The sequence for moving the cursor is as follows:

PRINT CHR$(27);" = ";CHR$(m+31);CHR$(n+31);

Here, m specifies the vertical cursor position and n specifies the horizontal position. The value of n should be greater than 1 and less than the screen width in the particular screen mode being used. The value of m should be greater than 1 and less than the number of lines in the virtual screen.

The ESC "=" sequence duplicates the LOCATE command with its first two parameters.

ESC "C" <character>

Used to select one of the nine international character sets as follows:

The <character> is a letter which corresponds to the character sets of one of the following countries. It must be an uppercase character.

US ASCII	PRINT CHR$(27);"CU"
France	PRINT CHR$(27);"CF"
Germany	PRINT CHRS(27);"CG"
England	PRINT CHR$(27);"CE"
Denmark	PRINT CHR$(27);"CD"

Sweden	PRINT CHR$(27);"CW"
Italy	PRINT CHR$(27);"CI"
Spain	PRINT CHR$(27);"CS"
Norway	PRINT CHR$(27);"CN"

This code sequence is equivalent to the BASIC OPTION COUNTRY command.

ESC "P"

In modes 0, 1, and 2 this escape sequence outputs the contents of the screen window currently being displayed to a printer in ASCII format. In mode 3 it outputs the contents of the entire physical screen in bit image format. It duplicates the COPY or screen dump function obtained by pressing the CTRL and PF5 key.

ESC "T"

Clears the line currently containing the cursor from its present position to the end of that logical line.

ESC "Y"

Clears the screen from the current position of the cursor to the end of the screen.

ESC CHR$(123)

Causes all characters to be displayed on the screen as blanks (the secret mode). The secret mode is not active in the System Display.

WARNING:
You should make sure that a program returns the user to normal non-secret mode, for example with an error handling routine. If the user is placed in immediate mode and the secret mode is still active, it is impossible to know what is happening. Also the reset button on the left of the PX-8 must be pressed in order to see any printed output except for the clock on the MENU screen and the System Display.

ESC CHR$(125)

Terminates the secret mode.

ESC CHR$(144)

Scrolls (n – 1) lines up, starting at line (n+1) so that line (n+m – 1) becomes

blank. This is done as follows:

PRINT CHR$(27);CHR$(144);CHR$(n – 1);CHR$(m);

Numbers specified for n and m must satisfy all of the following conditions.

$$\mathbf{0 \leqq (n-1) \leqq (R-1)}$$
$$\mathbf{1 \leqq m \leqq R}$$
$$\mathbf{(n-1+m-1) \leqq R}$$

Here, R is the number of virtual screen lines in mode 0, 1, or 2 and is the number of screen window lines in mode 3.

ESC CHR$(145)

Scrolls (n – 1) lines down starting at line n so that line n becomes blank. This is done as follows:

PRINT CHR$(27);CHR$(145);CHR$(n-1);CHR$(m);

Numbers specified for n and m must satisfy all of the following conditions:

$$\mathbf{0 \leqq (n-1) \leqq (R-1)}$$
$$\mathbf{1 \leqq m \leqq R}$$
$$\mathbf{(n-1+m-1) \leqq R}$$

Here, R is the number of virtual screen lines in mode 0, 1, or 2 and is the number of screen window lines in mode 3.

ESC CHR$(148)

In modes 0, 1, and 2 this escape sequence sets the number of lines n which are moved by one scrolling operation. The actual scrolling is carried out by printing an ESC 150 sequence. The number of lines are set up using the following sequence:

PRINT CHR$(27);CHR$(148);CHR$(n);

The number specified for n must be greater than 1 and less than the number of lines in the screen window.

This escape sequence does nothing in mode 3.

ESC CHR$(149)

In modes 0, 1, and 2 this escape sequence determines whether scrolling is performed automatically. The automatic scrolling mode is referred to as the tracking mode, and the mode in which automatic scrolling is not performed is referred to as the non-tracking mode. The tracking mode is used unless otherwise specified. The escape sequence for determining the tracking mode is as follows:

PRINT CHR$(27);CHR$(149);CHR$(<mode>);

In this sequence, <mode> is specified as either 0 or 1. The tracking mode is selected when 0 is specified, and the non-tracking mode is selected when 1 is specified.

ESC CHR$(150)

In modes 0, 1, and 2 this escape sequence displays the contents of the virtual screen containing the cursor after moving the screen window up n lines where n is the value specified by ESC CHR$(148), or 1 if ESC CHR$(148) has not been executed. If scrolling the screen up n lines would move the screen window beyond the home position, the virtual screen is displayed starting at the home positon. The cursor remains in its original position in the virtual screen.

ESC CHR$(151)

In modes 0, 1, and 2 this escape sequence displays the contents of the virtual screen containing the cursor after moving the screen window down n lines, where n is the value specified by ESC CHR$(148), or 1 if ESC CHR$(148) has not been executed. If scrolling the screen down n lines would move the screen window beyond the end of the virtual screen, the screen window is positioned so that the virtual screen's last line is displayed in the last line of the screen window. The cursor remains in its original position in the virtual screen.

ESC CHR$(160)

Lights the INS LED. It does not put the user in the insert mode.

ESC CHR$(161)

Turns off the INS LED.

ESC CHR$(162)

Lights the CAPS LED. It does not set the CAPS LOCK key to the on position.

ESC CHR$(163)

Turns off the CAPS LED.

ESC CHR$(164)

Lights the NUM LED, but does not select the numeric keypad.

ESC CHR$(165)

Turns off the NUM LED.

ESC CHR$(176)

This ESC code is used to disable the string printed by a programmable function key. However, with input from the command line or from an INPUT statement the PX-8 will be returned to the normal string printing mode of the programmable function keys. If you wish to determine if any of the programmable function keys have been pressed, use the ESC CHR$(176) mode in combination with INPUT$ or with INKEY$. An example of this is shown below.

```
10 PRINT CHR$(&H1B);CHR$(&HB0);
20 PRINT HEX$(ASC(INPUT$(1)));" ";
30 GOTO 20

run
E0  E1  E2  E3  E4  E5  E6  E7  E8  E9
↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
PF1 PF2 PF3 PF4 PF5 PF6 PF7 PF8 PF9 PF10
```

ESC CHR$(177)

This ESC code re-enables the programmable function keys so that a string is printed when they are pressed.

ESC CHR$(198)

In mode 3, this escape sequence draws a line on the graphic screen using the dot pattern specified by the user. No operation is performed when this sequence is executed in modes 0, 1, or 2. The elements of the sequence are as follows:

Byte 1:	CHR$(27)
Byte 2:	CHR$(198)
Byte 3:	High byte of horizontal starting position
Byte 4:	Low byte of horizontal starting position
Byte 5:	High byte of vertical starting position
Byte 6:	Low byte of vertical starting position
Byte 7:	High byte of horizontal ending position
Byte 8:	Low byte of horizontal ending position
Byte 9:	High byte of vertical ending position

Byte 10:	Low byte of vertical ending position
Byte 11:	First byte of mask pattern
Byte 12:	Second byte of mask pattern
Byte 13:	Function

The starting and ending positions are specified as two-byte hexadecimal numbers which indicate coordinates in the graphic screen. For example, starting coordinates of 400,20 (&H0190,&H0014) would be specified as follows:

Byte 3:	1 (&H01)
Byte 4:	144 (&H90)
Byte 5:	0 (&H00)
Byte 6:	20 (&H14)

The mask pattern used for drawing the line is specified in bit image format as described in the explanation of the LINE statement in Chapter 4. Calculations for diagonal lines are performed automatically. Function is specified as a number from 1 to 3 with the following meanings:

1:	OFF
2:	ON
3:	Complement

Dot positions corresponding to "1" bits in the mask pattern are reset (turned off) when 1 is specified for the function and are set (turned on) when 2 is specified. When 3 is specified, the complements of dots corresponding to "1" bits are displayed (ON dots corresponding to "1" bits are turned off, and OFF dots are turned on).

An example of specification of this sequence as follows draws a line from point (400,18) of the screen to point (18,18):

```
PRINT CHR$(27);CHR$(198);CHR$(1);CHR$(144);CHR$(Ø);
CHR$(18);CHR$(Ø);CHR$(18);CHR$(Ø);CHR$(18);
CHR$(&HAA); CHR$(&HAA);CHR$(2);
```

This command duplicates the LINE command of BASIC, but also allows the dots to be inverted (i.e. switch them on if they are off and vice versa), which LINE does not.

ESC CHR$(199)

This escape sequence sets or resets the specified points of the graphic screen. No operation is performed if this sequence is executed in modes 0, 1, or 2. The sequence consists of six bytes as follows:

Byte 1:	CHR$(27)
Byte 2:	CHR$(199)
Byte 3:	Function code (1:PSET, 0: PRESET)
Byte 4:	Vertical dot position — n1
Byte 5:	High byte of horizontal dot position } n2
Byte 6:	Low byte of horizontal dot position } n2

Numbers specified for n1 and n2 must be in the following ranges:

Ø ≦ n1 ≦ 63, Ø ≦ n2 ≦ 479

ESC CHR$(2Ø8)

Switches the display mode. Mode specification is as follows:

Mode 0		Mode 1	
Byte 1:	CHR$(27)	Byte 1:	CHR$(27)
Byte 2:	CHR$(208)	Byte 2:	CHR$(208)
Byte 3:	CHR$(0)	Byte 3:	CHR$(1)
Byte 4:	CHR$(n1)	Byte 4:	CHR$(n1)
Byte 5:	CHR$(n2)		

Mode 2		Mode 3	
Byte 1:	CHR$(27)	Byte 1:	CHR$(27)
Byte 2:	CHR$(208)	Byte 2:	CHR$(208)
Byte 3:	CHR$(2)	Byte 3:	CHR$(3)
Byte 4:	CHR$(n1)		
Byte 5:	CHR$(n2)		
Byte 6:	CHR$(p)		

The meanings of n1, n2, m, and p are as follows:

n1	Number of lines in virtual screen 1
n2	Number of lines in virtual screen 2
m	Number of columns in virtual screen 1
p	ASCII code corresponding to desired boundary character

The following sequence selects screen mode 2, sets the number of lines in virtual screen 1 to 10, the number of columns to 20 and "#" as the boundary character.

PRINT CHR$(27);CHR$(208);CHR$(2);CHR$(10);CHR$(20); "#";

ESC CHR$(209)

In modes 0, 1, or 2 this escape sequence specifies which of the two virtual screens is to be displayed. The operation is performed if this sequence is executed in mode 3. This is done as follows:

PRINT CHR$(27);CHR$(209);CHR$(n);

The first virtual screen is selected when 0 is specified for n, and the second virtual screen is selected when 1 is specified for n. If the third byte is not specified the default is 1.

ESC CHR$(210)

Displays the specified character in the specified position on the real screen. This is done as follows:

PRINT CHR$(27);CHR$(210);CHR$(x);CHR$(y);CHR$(p)

The meanings of x, y and p are as follows:

x	Vertical position (1 to 8)
y	Horizontal position (1 to 80)
p	ASCII character code

This sequence makes it possible to output characters to any location in the real screen, regardless of the position of the cursor or number of lines in the screen window.

ESC CHR$(211)

Turns on or off display of function key definitions. This is done as follows:

PRINT CHR$(27);CHR$(211);CHR$(n)

Function key definitions are displayed when 0 is specified for n, and are not displayed when 1 is specified. The default value is 1.

ESC CHR$(212)

In modes 0, 1, and 2 this escape sequence moves the screen window to the top of the virtual screen containing the cursor. No operation is performed if this sequence is executed in mode 3. The position of the cursor remains unchanged.

ESC CHR$(213)

In modes 0, 1, and 2 this escape sequence moves the screen window to the end of the virtual screen containing the cursor. No operation is performed if this sequence is executed in mode 3. The position of the cursor remains unchanged.

ESC CHR$(214)

In modes 0, 1, and 2 this escape sequence selects the type of cursor to be displayed. This sequence does nothing if executed in mode 3. The sequence consists of three bytes as follows:

Byte 1:	CHR$(27)
Byte 2:	CHR$(214)
Byte 3:	CHR$(n)

Here, n specifies the type of cursor displayed as follows:

0	Block cursor, flashing
1	Block cursor, non-flashing
2	Underline cursor, flashing
3	Underline cursor, non-flashing

The cursor will be set to the normal flashing block cursor if the return key or one of the cursor keys is pressed.

ESC CHR$(215)

In modes 0, 1, and 2 this escape sequence moves the screen window to the position occupied by the cursor. This sequence does nothing if executed in mode 3. The screen window is positioned so that the cursor is located near its centre.

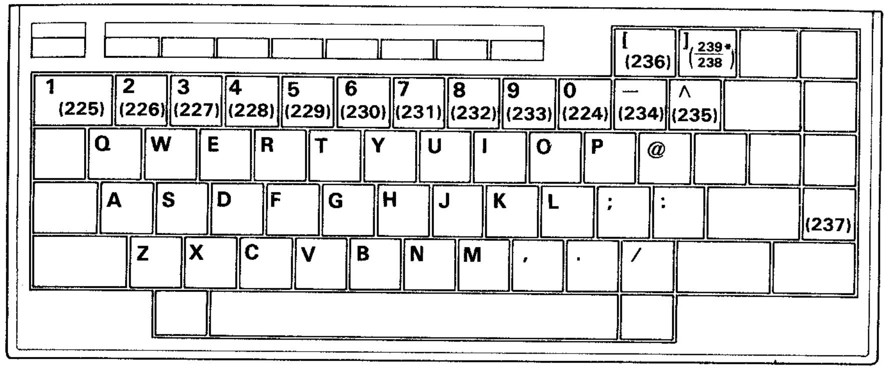
ESC CHR$(224)

This escape sequence defines those characters corresponding to ASCII codes 224 (&HE0) to 254 (&HFE). This sequence consists of eleven bytes as follows:

Byte 1:	CHR$(27)
Byte 2:	CHR$(224)
Byte 3:	Character code
Byte 4:	Pattern for dot row 1
Byte 5:	Pattern for dot row 2
Byte 6:	Pattern for dot row 3
Byte 7:	Pattern for dot row 4
Byte 8:	Pattern for dot row 5
Byte 9:	Pattern for dot row 6
Byte 10:	Pattern for dot row 7
Byte 11:	Pattern for dot row 8

The pattern making up each dot row is specified as the ASCII code equivalent of the binary number whose "1" bits correspond to dots which are turned on, and whose "0" bits correspond to dots which are turned off. For example, specifying CHR$(63) (where 63 is the decimal equivalent of 1111111B) for byte 1 causes all dots in dot row one to be turned on when the character code specified in byte 3 is displayed; conversely, specifying CHR$(0) (i.e.,00000000B) causes all dots in the applicable row to be turned off.

Note:
User character definitions for codes 224 to 239 can be displayed by pressing the graph key together with certain other keys on the keyboard. Keys pressed for each code are as shown in the figure below.



* Press together with SHIFT key.

A sample definition for character code 230 is shown below:

```
PRINT CHR$(27);CHR$(224);CHR$(23Ø);CHR$(12);
CHR$(12);CHR$(3Ø);CHR$(63);CHR$(12);CHR$(18);
CHR$(Ø);CHR$(Ø);
```

After executing this sequence, the character corresponding to code 230 can be displayed by pressing GRAPH and the key marked "230" in the figure above.

ESC CHR$(24Ø)

Controls the key repeat function. This sequence consists of three bytes as follows:

Byte 1:	CHR$(27)
Byte 2:	CHR$(240)
Byte 3:	CHR$(n)

If 0 is specified for n, the repeat function is turned off. If 1 is specified, it is turned on.

ESC CHR$(241)

Sets the starting time for the key repeat function. The sequence consists of three bytes as follows:

Byte 1: CHR$(27)
Byte 2: CHR$(241)
Byte 3: CHR$(n)

The keyboard repeat function starting time is equal to n/64 seconds where n is a number from 1 to 127.

ESC CHR$(242)

Sets the duration of the key repeat interval. This sequence consists of three bytes as follows:

Byte 1: CHR$(27)
Byte 2: CHR$(242)
Byte 3: CHR$(n)

The key repeat interval is equal to n/256 seconds, where n is a number from 1 to 127.

ESC CHR$(243)

Sets the arrow key codes. This sequence consists of six bytes as follows:

Byte 1: CHR$(27)
Byte 2: CHR$(243)
Byte 3: Code for $\boxed{\rightarrow}$
Byte 4: Code for $\boxed{\leftarrow}$
Byte 5: Code for $\boxed{\uparrow}$
Byte 6: Code for $\boxed{\downarrow}$

This sequence only changes the arrow key codes during program execution. Normal code assignments are restored automatically when BASIC returns to the command mode.

ESC CHR$(244)

Sets the scroll key codes. This sequence consists of six bytes as follows:

Byte 1: CHR$(27)
Byte 2: CHR$(244)
Byte 3: Code for $\boxed{\text{SHIFT}}$ + $\boxed{\rightarrow}$
Byte 4: Code for $\boxed{\text{SHIFT}}$ + $\boxed{\leftarrow}$
Byte 5: Code for $\boxed{\text{SHIFT}}$ + $\boxed{\uparrow}$
Byte 6: Code for $\boxed{\text{SHIFT}}$ + $\boxed{\downarrow}$

ESC CHR$(245)

Sets the $\boxed{\text{CTRL}}$ + arrow key codes. This sequence consists of six bytes as follows:

Byte 1: CHR$(27)
Byte 2: CHR$(245)
Byte 3: Code for $\boxed{\text{CTRL}}$ + $\boxed{\rightarrow}$
Byte 4: Code for $\boxed{\text{CTRL}}$ + $\boxed{\leftarrow}$
Byte 5: Code for $\boxed{\text{CTRL}}$ + $\boxed{\uparrow}$
Byte 6: Code for $\boxed{\text{CTRL}}$ + $\boxed{\downarrow}$

ESC CHR$(246)

Clears the keyboard buffer of all unprocessed input characters.

ESC CHR$(247)

The ESC 247 code allows the programmer to switch the various shift keys on and off. Thus the numeric key pad can be set on, or the shift key 'held down'. The key state is set to normal by the user pressing the appropriate key, so it is advisable to program with the possiblity in mind that the key may be reset outside program control.

The sequence of characters is as follows:

Byte 1: CHR$(27)
Byte 2: CHR$(247)
Byte 3: CHR$(n)

Numbers which may be specified for n and their meanings are as follows:

n (Decimal)	Shift state
0	Normal
2	SHIFT
4	CAPS LOCK
6	CAPS LOCK SHIFT
16	NUM
18	Numeric SHIFT
32	GRPH
34	GRPH SHIFT
64	CTRL
66	CTRL SHIFT

This sequence does nothing if numbers other than those above are specified for n.

Appendix D MACHINE LANGUAGE SUBROUTINES

The CALL and USR statements of BASIC make it possible to execute machine language subroutines from programs written in BASIC. Such subroutines must be written into memory in machine language with the POKE statement before they can be called. It is also possible to use an assembler such as the MACRO-80 assembler and LINK-80 linker/loader to assemble and load routines written in assembly language; however these programs are not included in the transient program ROM capsule provided with the PX-8, and must be loaded from a flexible disk which is compatible with CP/M and the PX-8. See any of the various handbooks available on the Z80 microcomputer or the Z80 assembly language for the Z80 instruction code set.

When preparing machine language subroutines, remember that the presence of even a single error in the machine code is likely to result in destruction of all data included in the PX-8's memory (including BASIC itself). Therefore, be sure to back up all data and programs in memory on a disk before attempting to test or debug such routines.

1. Memory Allocation

Memory space must be reserved for storage of the instruction codes of machine language subroutines before they can be written into memory with the POKE statement. This is done using the CLEAR statement of BASIC or the /M: option of the BASIC command. When using the /M: option, the starting address of the machine language area is the address specified, and the ending address is that immediately preceding the starting address of BDOS. Locations 6 and 7 in page zero hold the current BDOS starting address. This will change depending on the USER BIOS and RAM disk sizes.

When a machine language subroutine is called, the stack pointer is set up for 8 levels (16 bytes) of stack storage. If more stack space is required, BASIC's stack can be saved and a new stack set up for use by the machine language subroutine. BASIC's stack must be restored, however, before returning from the subroutine.

2. USR Function Calls

With BASIC, the format used for calling USR functions is as follows.

USR[<digit>](argument)

Here, <digit> is a number from 0 to 9 and the argument specified is any numeric or string expression. <digit> specifies which of the USR routines is being called, and corresponds to the digit specified in the DEF USR statement for that routine. If <digit> is omitted USR0 is assumed. The address specified in the DEF USR statement determines the starting address of the subroutine.

When a USR function call is made a value is placed in CPU register A which specifies the type of argument specified. The value placed in register A may be any of the following.

Value	Type of argument
2	Two-byte integer (two's complement)
3	String
4	Single precision floating point number
8	Double precision floating point number

If the argument is an integer

FAC+0 contains the lower 8 bits of the argument
FAC+1 contains the upper 8 bits of the argument.

If the argument is a single precision floating point number
FAC+0 contains the lowest 8 bits of the mantissa
FAC+1 contains the middle 8 bits of the mantissa
FAC+2 contains the highest 7 bits of the mantissa (with leading 1 suppressed). Bit 7 is the sign of the number (0 for positive and 1 for negative)
FAC+3 is the exponent minus 128. The binary point is the bit to the left of the most significant bit of the mantissa.

If the argument is a double precision floating point number, FAC-4 to FAC-1 contain four more bytes of the mantissa with the lowest 8 bits in FAC-4.

If the argument is a string, the DE register pair points to three bytes called the "string descriptor." Byte 0 of the string descriptor contains the length of the string (0 to 255); and bytes 1 and 2 are the lower and upper 8 bits of the starting address of the string in string space.

CAUTION:
If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program in this way. To avoid unpredictable results, add +" " to the string literal in the program.

Example A$=”STRING CHARS” + “”

This will copy the string literal into string space and prevent alteration of program text during a subroutine call.

3. CALL Statement

BASIC user function calls may also be made with the CALL statement. A CALL statement with no arguments generates a simple “CALL” instruction. The corresponding subroutine should return to the BASIC program via a simple “RET” instruction (“CALL” and “RET” are Z80 opcodes; see a Z80 reference manual for details.)

A subroutine CALL with arguments results in a somewhat more complex calling sequence. For each argument in the CALL argument list, a parameter is passed to the subroutine. That parameter is the address of the low byte of the argument. Therefore, parameters always occupy two bytes each, regardless of type.

The method of passing parameters depends on the number of parameters to be passed as follows:

(1) If the number of parameters is less than or equal to 3, they are passed in the registers. Parameter 1 will be in HL, 2 (if present) in DE, and 3 (if present) in BC.

(2) If the number of parameters is greater than 3, they are passed as follows:
   (a) Parameter 1 in HL.
   (b) Parameter 2 in DE.
   (c) Parameters 3 through n in a contiguous data block. Register pair BC will point to the low byte of this data block (i.e., to the low byte of parameter 3).

Note that with this scheme the subroutine must know how many parameters to expect in order to find them. Conversely, the calling program is responsible for passing the correct number of parameters. There are no checks for correct number or type of parameters.

When accessing parameters in a subroutine, don’t forget that they are pointers to the actual arguments passed.

NOTE:
It is entirely up to the programmer to ensure that arguments in the calling program match those expected by the subroutine in number, type and length. This applies to BASIC subroutines as well as those written in machine language.

4. Interrupts

Machine language subroutines can be written to handle interrupts. All interrupt handling routines should save the stack, registers A to L and the PSW. Since an interrupt received automatically disables all further interrupts, they should always be re-enabled before returning from the subroutine.

Appendix E DERIVED FUNCTIONS

Functions that are not intrinsic to PX-8 BASIC may be calculated as follows.

Function	**BASIC Equivalent**
SECANT	SEC(X)=1/COS(X)
COSECANT	CSC(X)=1/SIN(X)
COTANGENT	COT(X)=1/TAN(X)
INVERSE SINE	ARCSIN(X)=ATN(X/SQR(1−X∗X+1))
INVERSE COSINE	ARCCOS(X)= −ATN(X/SQR(1−X∗X)) +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)+1.570796326794897
HYPERBOLIC SINE	SINH(X)=(EXP(X)−EXP(−X))/2
HYPERBOLIC COSINE	COSH(X)=(EXP(X)+EXP(−X))/2
HYPERBOLIC TANGENT	TANH(X)=(EXP(X)−EXP(−X))/(EXP(X) +EXP(−X))
HYPERBOLIC SECANT	SECH(X)=2/(EXP(X)+EXP(−X))
HYPERBOLIC COSECANT	CSCH(X)=2/(EXP(X)−EXP(−X))
HYPERBOLIC COTANGENT	COTH(X)=(EXP(X)+EXP(−X))/(EXP(X) −EXP(−X))
INVERSE HYPERBOLIC SINE	ARCSINH(X)=LOG(X+SQR(X∗X+1))
INVERSE HYPERBOLIC COSINE	ARCCOSH(X)=LOG(X+SQR(X∗X−1)
INVERSE HYPERBOLIC TANGENT	ARCTANH(X)=LOG((1+X)/(1−X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X)=LOG((SQR(1−X∗X)+1)/X)
INVERSE HYPERBOLIC COSECANT	ARCCSCH(X)=LOG((1+SGN(X)∗SQR(1+X∗X))/X
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X)=LOG((X+1)/(X−1))/2

Any of these functions can easily be used in a program by defining it with a DEF FN statement. This is illustrated in the example below.

Example

Function definition: DEF FN SINH(X)−(EXP(X)−(EXP(X))/2
Function call: A=FNSINH(Y)

Appendix F ASCII 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	0 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	[illegible] 152	168	184	200	216	232	248
9	1001	9	25	) 41	9 57	I 73	Y 89	i 105	y 121	└ 137	[illegible] 153	169	185	201	217	232	249
A	1010	10	26	* 42	: 58	J 74	Z 90	j 106	z 122	┘ 138	[illegible] 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

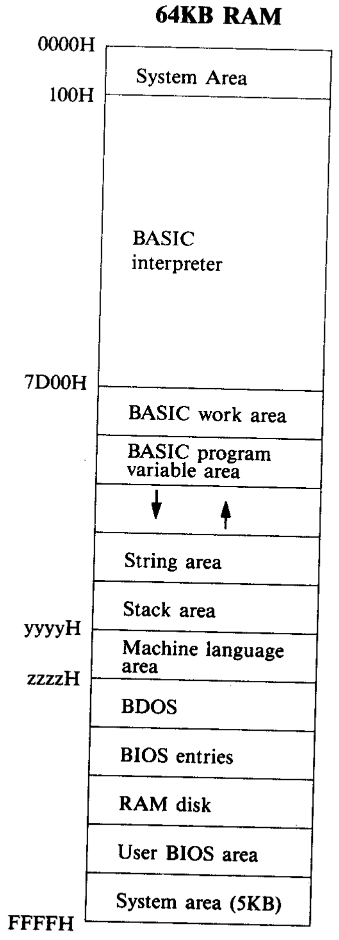
ASCII

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 section 2.6.2, "Control Characters," and the User's 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 G MEMORY MAP



yyyyH........ Can be found at memory addresses 7D38H and 7D39H. Address 7D38H contains the lower byte of yyyyH and address 7D39 contains the higher byte.

zzzzH......... Varies according to the size of RAM disk.
zzzzH can be found in locations 6 and 7 in page zero. From BASIC,

PEEK(6)+PEEK(7)*256

will return the present value of zzzzH.

Appendix H SOME EXAMPLE PROGRAMS

This manual is not meant to be a tutorial manual to teach BASIC — there are many books which teach the use of MICROSOFT BASIC. However, some aspects of programming are specific to the PX-8. This appendix is meant to illustrate some of these specific points and provide examples of how the computer can be programmed in ways which exploit the features of the machine.

1. Use of the User-Defined Characters

Appendix F shows the character set of the PX-8. It is normally only possible to program the characters which have an ASCII code of 22 and above. It is possible with a machine code routine to alter the VRAM and reconfigure it to allow the characters from ASCII code 160 to 254 to be programmed. This is beyond the scope of this manual. The downloading of a character from software is outlined in Appendix C, under the escape sequence ESC CHR$(224). The following programs and descriptions extend this information by showing practical examples.

(i) A simple program to illustrate the definition of a character and printing it to the screen.

A character is defined by sending the sequence:

BYTE 1:	CHR$(27)	The ESC character
BYTE 2:	CHR$(224)	The code to download
BYTE 3:	CHR$(n)	The code for the character to be changed
BYTE 4:	CHR$(r1)	The pattern for the top row
BYTE 5:	CHR$(r2)	The pattern for row 2
BYTE 6:	CHR$(r3)	The pattern for row 3
BYTE 7:	CHR$(r4)	The pattern for row 4
BYTE 8:	CHR$(r5)	The pattern for row 5
BYTE 9:	CHR$(r6)	The pattern for row 6
BYTE 10:	CHR$(r7)	The pattern for row 7
BYTE 11:	CHR$(r8)	The pattern for the bottom row

The pattern which makes up each row is specified by the bit settings of the number, sent as an ASCII code. It is easiest to design the characters on squared paper and translate it into numbers. Dots in the pattern which are turned on correspond to a "1" in the binary number and dots which are turned off correspond to a "0". The design which gives the pattern must be converted from binary into a decimal or hexadecimal number. For those not familiar with converting the binary numbers to decimal the procedure is best explained with an example.

128	64	32	16	8	4	2	1
		*		*		*	*

The pattern in the diagram shows which dots of the line will be set. This particular pattern would correspond to the binary number "00101011". To convert the number to decimal, add the numbers above the boxes where there is dot to be set. Thus 32+8+2+1 gives a total of 43. For a whole character a pattern would be produced as follows. The numbers at the right are the ones which would define the row in a program.

128	64	32	16	8	4	2	1	
				*	*			= 12
			*	*	*	*		= 30
		*					*	= 33
		*					*	= 33
			*			*		= 18
				*	*			= 12
		*	*	*	*	*	*	= 63
								= 0

Characters are six dots wide by eight high. The two left-hand positions are always ignored. The numbers used to define the rows will thus be in the range 0 to 63 (0 to 3F hexadecimal). Any attempt to use the left-hand two positions of the full eight bits of the byte will be ignored.

If the bottom row of the character is filled in there will be no space between the character printed and the character on the next row of the screen. If you want the two characters to be contiguous, dots on this row should be set; otherwise the row should be left blank.

The first program defines the character shown in the diagram and prints it on the screen. It is downloaded into the user-defined character area as the character with ASCII code 231 (or E7 in hexadecimal notation).

```
10 CLS
20 PRINT CHR$(27);CHR$(224);CHR$(231);
30 FOR Y=1 TO 8
40 READ A
50 PRINT CHR$(A);
60 NEXT
70 PRINT
80 PRINT CHR$(231)
90 PRINT
100 DATA 12,30,33,33,18,12,63,0
```

```
run
```

```
Ok
```

In line 20 it it very important that the semi-colon is placed at the end of the line. Without this the first two bytes of the row will be interpreted as the carriage return and line feed, which would normally cause the cursor to move to the beginning of the next line. Carriage return is ASCII code 13 and line feed is ASCII code 10 in decimal notation. Try leaving the semi-colon out and note the change in the character. Because these two extra characters are inserted the bottom two rows defined are lost, with the two extra rows being inserted at the top to correspond to the line feed and carriage return.

The data for the character is read in from the series of DATA statements. If a series of characters are being defined, they are best arranged in sets of eight DATA statements on different lines. This makes it easier to find out which data byte corresponds to which row of which character when you wish to change the character or are debugging a program.

(ii) The next program shows how blocks of graphics characters can be used to make larger characters. The example shows a set of ARABIC characters where each character is made up of a block of four user-defined characters.

```
10 'User defined graphics
20 FOR X=&HE0 TO &HFB
30 PRINT CHR$(27);CHR$(&HE0);CHR$(X);
40 FOR Y=1 TO 8
50 READ A
60 PRINT CHR$(A);
70 NEXT Y
80 NEXT X
85 CLS
90 FOR X=&HE0 TO &HFB STEP 4
```

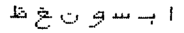
```
100 PRINT CHR$(X);CHR$(X+1);
110 NEXT:PRINT
120 FOR X=&HE2 TO &HFB STEP 4
130 PRINT CHR$(X);CHR$(X+1);
140 NEXT:PRINT
1000 DATA &H02,&H02,&H02,&H02,&H02,&H03,&H0F,&H00
1010 DATA &H00,&H30,&H00,&H18,&H24,&H04,&H3C,&H00
1020 DATA &H00,&H00,&H00,&H00,&H00,&H00,&H00,&H00
1030 DATA &H00,&H00,&H00,&H00,&H00,&H00,&H00,&H00
2000 DATA &H00,&H03,&H00,&H07,&H04,&H02,&H01,&H02
2010 DATA &H00,&H00,&H00,&H30,&H10,&H20,&H3C,&H00
2020 DATA &H04,&H08,&H08,&H08,&H07,&H00,&H00,&H00
2030 DATA &H00,&H00,&H00,&H08,&H30,&H00,&H00,&H00
3000 DATA &H00,&H00,&H00,&H00,&H01,&H08,&H08,&H08
3010 DATA &H00,&H00,&H00,&H00,&H20,&H04,&H04,&H04
3020 DATA &H04,&H03,&H00,&H00,&H00,&H00,&H00,&H00
3030 DATA &H08,&H30,&H00,&H00,&H00,&H00,&H00,&H00
4000 DATA &H00,&H00,&H00,&H00,&H00,&H01,&H01,&H00
4010 DATA &H00,&H00,&H00,&H1C,&H24,&H04,&H3C,&H04
4020 DATA &H00,&H00,&H00,&H07,&H00,&H00,&H00,&H00
4030 DATA &H08,&H10,&H20,&H00,&H00,&H00,&H00,&H00
5000 DATA &H00,&H00,&H00,&H00,&H04,&H04,&H0F,&H00
5010 DATA &H00,&H00,&H00,&H00,&H24,&H24,&H3C,&H00
5020 DATA &H00,&H00,&H00,&H00,&H00,&H00,&H00,&H00
5030 DATA &H00,&H00,&H00,&H00,&H00,&H00,&H00,&H00
6000 DATA &H00,&H00,&H00,&H00,&H00,&H00,&H0F,&H00
6010 DATA &H00,&H00,&H00,&H00,&H04,&H04,&H3C,&H00
6020 DATA &H00,&H00,&H00,&H00,&H00,&H00,&H00,&H00
6030 DATA &H08,&H08,&H00,&H00,&H00,&H00,&H00,&H00
7000 DATA &H01,&H01,&H01,&H01,&H01,&H01,&H01,&H01
7010 DATA &H00,&H00,&H00,&H00,&H00,&H00,&H00,&H00
7020 DATA &H00,&H00,&H00,&H00,&H00,&H00,&H00,&H00
7030 DATA &H00,&H00,&H00,&H00,&H00,&H00,&H00,&H00
```

Lines 1000 onwards contain the data for successive characters. Each line contains the data for successive rows of each character. All characters and data have been entered in hexadecimal notation. See HEX$ in Chapter 4 for conversion between decimal and hexadecimal numbers. The four individual characters making up each large character are defined in the order top left, top right, bottom left, and bottom right.

Lines 20 to 80 read this data from the DATA statements and download each character in the same manner as the previous program.

Lines 90 to 140 print the top halves and then the bottom halves of the characters. Since four user-defined characters are used together to make one large screen character, the STEP to find the next character in the loops is four.

User-defined characters can only be printed from screen mode 3 — by bit image mode printing of a screen dump — unless they have also been downloaded into a suitable printer which is capable of receiving characters in a downloadable form. Such a screen dump of the output of the screen appears as follows when the program has been run:

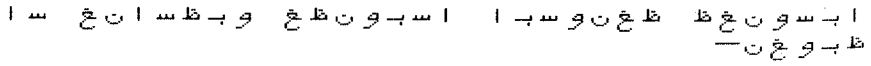


```
Ok
```

NOTE:
When using these characters in the following programs you should LOGIN to another program area before typing them in. If the screen is changed by either the SCREEN command or WIDTH command, or by going via the menu, the first two user-defined characters, and possibly more will be altered. Simply using LOGIN will not reset them.

(iii) When combinations of characters are used in this way, it is often more convenient if the characters are grouped as a variable, so that simply saying PRINT A$ for example prints the block as a whole. This can be achieved using string concatenation. The following program shows how the ARABIC characters of the previous program can be defined as variables, and how they can be printed as one character by typing a key. The characters have been designed so that the cursor moves from right to left to illustrate ways of using control codes.

```
10 STOP KEY OFF
20 SCREEN 3,0,0 : CLS
30 XP = 1: YP = 1
40 FOR N = 0 TO 6
50 J = 4 * N
60 C$(N) = CHR$(&HE0+J)+ CHR$(&HE0+J+1)+CHR$(8)+CHR$(8)+CHR$
(10)+CHR$(&HE0+J+2)+CHR$(&HE0+J+3)+CHR$(&H1E)+STRING$(4,8)
70 NEXT N
80 CSR$ = CHR$(133)+CHR$(133)+STRING$(2,8)
90 SP$ = "  " + STRING$(4,8)
100 LOCATE XP+68,YP*2,0
110 PRINT CSR$;
120 A$ = INKEY$ : IF A$ = "" THEN 120
130 IF A$ = CHR$(27) THEN CLS : STOP KEY ON : END
140 IF A$ = " " THEN PRINT SP$;: GOTO 180
150 IF A$ < "A" OR A$ > "G" THEN 120
160 V = 71 - ASC(A$)
170 PRINT C$(V);
180 XP = XP + 1 : IF XP > 34 THEN XP = 1 : YP = YP + 1: IF Y
P > 4 THEN CLS : YP = 1 : GOTO 100 ELSE GOTO 100
190 GOTO 110
```



The program disables the STOP key, then initialises variables XP and YP which are used to position the first character.

The loop forming lines 40 to 70 defines the characters which will be printed if the keys "A" to "G" are pressed. They are stored in the array C$( ). The characters are built up as follows: the first two characters are the top pair of the block of user defined characters. Next, two backspace characters are added (the ASCII code for a backspace is 8) because the position of the cursor is one to the right of the characters when they have been printed. This leaves the cursor in the position of the first character of the pair. By adding a linefeed character (ASCII code 10) the cursor is moved down one line to the bottom left of the block of four. The bottom row of the block is now added as the next two characters. If the block is printed at this stage the cursor will be placed to the right of the bottom of the block of four user-defined characters. The next group of characters must be printed two places to the left on the top of the first group. This is achieved by adding a 'cursoı up' character (ASCII code 30, or 1E in hexadecimal), and then four backspace characters. This gives a total string length of twelve characters. When this string is printed, the sequence appearing on the screen will be as follows: the top two of the block will appear; the cursor will then move back two positions and down one, print the bottom two characters, then move up, back four, and be ready to print again.

The program as a whole prints a cursor which is defined as two horizontal lines using the predefined graphics character whose ASCII code is 133. This together with the backspaces to move to the left is defined as the variable CSR$ in line 80. To allow a space to be printed, line 90 defines SP$ as two normal spaces with backspaces added to allow printing from left to right.

Lines 100 to 190 form the main part of the program. The cursor is placed on the second line of the screen at the extreme right. It cannot be placed directly on the edge. When one of the groups of characters from the array C$( ) is printed the LOCATE command tests to see if the string length added to the horizontal position of the LOCATE command is greater than 81. If it is, the complete string will be printed on the next line of the screen. The strings of the array are twelve characters long — it does not matter that some are cursor movement control codes. The maximum position the strings of C$( ) can be printed from is thus 69. The cursor is printed on the second line so that when the screen scrolls up on reaching the bottom line the top half of the character groups are not lost.

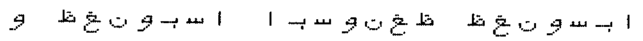
Line 110 prints the cursor and line 120 waits for a key to be pressed. Lines 130 to 150 test which key has been pressed. The $\boxed{\text{ESC}}$ key (ASCII code 27) allows the user to exit from the program. Line 140 prints a space and line 150 eliminates all characters other than those in the range "A" to "G".

When a key in the permitted range is pressed, the ASCII code is subtracted from the constant 71 to index the array C$( ), and the corresponding character is printed. The counter XP is then incremented so that a check can be made on the number of characters per line. When this is exceeded the line counter YP is incremented and the cursor moved to the right hand position on the next line. When the screen is full it is cleared.

The program loops back to line 110 to print the cursor and wait for another character, until the user exits by pressing the $\boxed{\text{ESC}}$ key.

(iv) The restrictions of the LOCATE command in the previous program can be overcome if the position of the characters is calculated instead of being printed as a block having previously been defined in a string. This allows the characters to go up the edge of the screen, but does requires some sophisticated numerical computation.

```
10 STOP KEY OFF
20 SCREEN 3,,0:CLS
30 XP=1:YP=1
40 LOCATE 81-XP*2,YP*2+1:PRINT CHR$(133);CHR$(133);
50 A$=INKEY$:IF A$="" THEN 50
60 IF A$=CHR$(27) THEN CLS:STOP KEY ON:END
70 IF A$=" " THEN LOCATE 81-XP*2,YP*2+1,0:PRINT"  ";:GOTO 150
80 IF A$<"A" OR A$>"G" THEN 50
90 LOCATE 81-XP*2,YP*2,0
100 V=71-ASC(A$)
110 C=&HE0+V*4
120 PRINT CHR$(C);CHR$(C+1);
130 LOCATE 81-XP*2,YP*2+1,0
140 PRINT CHR$(C+2);CHR$(C+3);
150 XP=XP+1:IF XP>40 THEN XP=1:YP=YP+1:IF YP>4 THEN CLS:YP=1
160 GOTO 40
```



This program is a modification of the previous one.

After initialising the variables in lines 10 to 30, the main body of the program begins at line 40 by printing the cursor, using the same characters as in the previous program. The position is calculated by means of the counter XP which is used later in the program to determine how many blocks

of characters are on the line.

Lines 50 to 80 check for input from the keyboard and exclude unwanted characters. If the space bar is pressed line 70 calculates the position in which to place two spaces so that the cursor is erased.

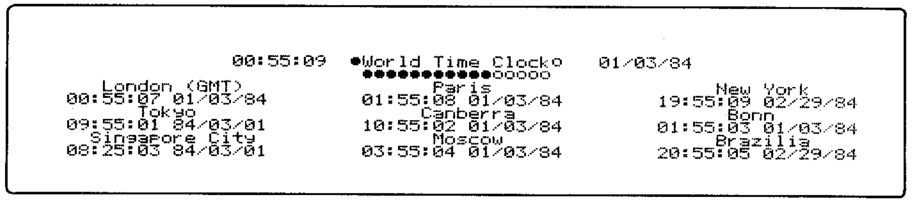
Line 90 calculates the position for the first character of the block to be printed whenever a key in the range "A" to "G" is pressed.

Lines 100 and 110 determine which of the the blocks of characters to print from the ASCII code of the key pressed. The corresponding user-defined characters are printed in positions calculated according to the number of blocks already on the line. Line 120 prints the upper pair of the block, and line 140 the lower pair.

Line 150 then updates the counter for the number of blocks of characters printed. As with the previous program, when the line is full the cursor is moved to the next line.

This program is best understood by working through what will actually happen when the program runs. You can do this by calculating the values of XP and YP, or having the PX-8 print them to a file or to an external printer. It is rather difficult to write a program such as this because if the program has to be altered the recalculation may not be easy. The previous program is easier to re-program and understand.

2. A Clock Program



```
                    00:55:09   ●World Time Clock○   01/03/84
                                ●●●●●●●●●●●○○○○○
     London (GMT)                     Paris                          New York
  00:55:07 01/03/84             01:55:08 01/03/84             19:55:09 02/29/84
        Tokyo                       Canberra                         Bonn
  09:55:01 84/03/01             10:55:02 01/03/84             01:55:03 01/03/84
    Singapore City                   Moscow                        Brazilia
  08:25:03 84/03/01             03:55:04 01/03/84             20:55:05 02/29/84
```

```
10 STOP KEY OFF
20 DEF FNT(P1$,P2$)=VAL(MID$(P1$,INSTR("HMS",P2$)*3-2))
30 DEF FND(P1$,P2$)=VAL(MID$(P1$,INSTR("MDY",P2$)*3-2))
40 DEF FNN(P1,P2) = (P1>7)*(((P1 MOD 2)=0)*31+((P1 MOD 2)=1)
*30) + (P1<8)*(((P1 MOD 2)=0)*30+((P1 MOD 2)=1)*31+(P1=2)*((
(P2 MOD 4)=0) + ((P2 MOD 4)<>0)*2))
50 DEF FNS$(P1,P2,P3,P4$)=RIGHT$("0"+MID$(STR$(P1),2),2)+P4$
+RIGHT$("0"+MID$(STR$(P2),2),2)+P4$+RIGHT$("0"+MID$(STR$(P3)
,2),2)
60 DEF FNP(P1)=-MN*(P1=1)-DY*(P1=2)-YR*(P1=3)
70 OH=FNT(TIME$,"H"):T=0:SS=0
80 READ MAX
90 DIM XP(MAX),YP(MAX),AS$(MAX),HR(MAX),ME(MAX),SD(MAX),L(3,
MAX)
100 FOR X=1 TO MAX
110 READ XP(X),YP(X),AS$(X),NT$,L(1,X),L(2,X),L(3,X)
120 HR(X)=FNT(NT$,"H")
130 ME(X)=FNT(NT$,"M")
140 SD(X)=FNT(NT$,"S")
150 NEXT
160 SCREEN 0,0,0:CLS
170 LOCATE 29,1:PRINT CHR$(143);"World Time Clock";CHR$(143)
180 LOCATE 30,2:PRINT STRING$(16,143)
190 PRINT"        London (GMT)                    Paris
     New York"
200 PRINT:PRINT"          Tokyo                       Canberra
          Bonn"
210 PRINT:PRINT"        Singapore City                Moscow
         Brazilia"
220 A=FRE(0):A=FRE(A$)
230 H=FNT(TIME$,"H"):M=FNT(TIME$,"M"):S=FNT(TIME$,"S")
240 TD=FND(DATE$,"D"):TM=FND(DATE$,"M"):TY=FND(DATE$,"Y")
250 T=1-T:IF T=0 THEN SS=1-SS
260 FOR X=1 TO MAX
270 DY=TD:MN=TM:YR=TY
280 IN$=INKEY$:IF IN$=CHR$(3) THEN 420
290 LOCATE 19,1:PRINT TIME$:LOCATE 50,1:PRINT FNS$(FNP(2),FN
P(1),FNP(3),"/")
300 IF OH<>H THEN SOUND 1200,5:OH=H
310 H=FNT(TIME$,"H"):M=FNT(TIME$,"M"):S=FNT(TIME$,"S")
320 IF T=0 AND X=1 THEN ZX=29:ZY=1:GOTO 350
```

```
330 IF T=1 AND X=9 THEN ZX=46:ZY=1:GOTO 350
340 ZX=T*9+X+28:ZY=2
350 LOCATE ZX,ZY,0:IF SS=0 THEN PRINT CHR$(143); ELSE PRINT
CHR$(144);
360 LOCATE XP(X),YP(X),0
370 H1=HR(X):M1=ME(X):S1=SD(X)
380 IF AS$(X)="-" THEN GOSUB 2000 ELSE GOSUB 1000
390 PRINT USING "& &";FNS$(HRS,MIN,SEC,":");FNS$(FNP(L(1,X))
,FNP(L(2,X)),FNP(L(3,X)),"/");
400 NEXT
410 GOTO 220
420 STOP KEY ON:CLS:END
1000 SEC=S+S1
1010 MIN=M+M1+SEC\60:SEC=SEC MOD 60
1020 HRS=H+H1+MIN\60:MIN=MIN MOD 60
1030 IF HRS<24 THEN 1080
1040 DY=DY+HRS\24:HRS=HRS MOD 24
1050 ND=FNN(MN,YR)
1060 MN=MN+DY\ND:DY=DY MOD ND
1070 IF MN>12 THEN YR=YR+1:MN=MN-12
1080 RETURN
2000 SEC=S-S1:IF SEC<0 THEN SEC=SEC+60:M1=M1+1
2010 MIN=M-M1:IF MIN<0 THEN MIN=MIN+60:H1=H1+1
2020 HRS=H-H1:IF HRS>=0 THEN 2060
2030 DY=DY-1+HRS\24:HRS=HRS+(HRS\24+1)*24
2040 ND=FNN(MN-1,YR):IF DY<1 THEN MN=MN-1:DY=DY+ND
2050 IF MN<1 THEN YR=YR-1:MN=MN+12:DY=FNN(MN,YR)
2060 RETURN
3000 DATA 9
3010 DATA 05,04,"+","00:00:00",2,1,3
3020 DATA 30,04,"+","01:00:00",2,1,3
3030 DATA 55,04,"-","05:00:00",1,2,3
3040 DATA 05,06,"+","09:00:00",3,1,2
3050 DATA 30,06,"+","10:00:00",2,1,3
3060 DATA 55,06,"+","01:00:00",2,1,3
3070 DATA 05,08,"+","07:30:00",3,1,2
3080 DATA 30,08,"+","03:00:00",2,1,3
3090 DATA 55,08,"-","04:00:00",1,2,3
```

This program, while being useful, is meant to illustrate the use of string handling and other commands in BASIC in a practical program. It also contains subroutines which handle the time and date. These will be of use in programming the PX-8 in combination with the ALARM command. When combined with the type of program given as an example in the ALARM section of Chapter 4 these subroutines allow a wide range of time based applications for the PX-8.

The program shows the time at various places around the world relative to GMT.

Line 10 switches off the STOP key while the program is running, allowing lines 280 and 420 to end the program in an orderly manner by re-enabling the key only when the user wishes to cease program execution.

Lines 20 to 60 define a number of functions which allow values to be determined related to the date and time.

Line 20 defines a function which returns the hours, minutes or seconds from the TIME$ string. Line 30 performs the same function on the DATE$ string. They are used in lines 230 and 240 and elsewhere in the program. As an example of their operation, consider the problem of returning the minutes from the time in the second statement in line 310:

M=FNT(TIME$,"M")

The two strings TIME$ and "M" are substituted for the values of P1$ and P2$ respectively in function FNT. The INSTR function is then used to return a value of 1, 2 or 3 depending on whether string P2$ is "H", "M" or "S" since P2$ is being searched for in the string "HMS". In this example it is "M", so INSTR returns a value of 2. The characters corresponding to the value of the hour begin at position 1 in the TIME$ string, the minutes at position 4 and the seconds at position 7. By multiplying the value returned by the INSTR function by 3 and subtracting two, the value returned will correspond to the correct position. The function MID$ is then used to extract the string beginning with this position, and the numerical value of the minutes is found using the VAL function. This value is then stored in the variable M.

Line 40 defines a function which returns the value of the number of days in the month. It is used in the two subroutines which add or subtract the time difference, for example in line 1050. In this example, MN is a variable which contains the number of the month and YR is a variable containing the last two digits of the year. These values are passed to the variables P1 and P2 in the function FNN. The function uses an algorithm involving logical operations to return the

number of the days in the month and year required. In evaluating the function P1 will correspond to the number of the month and P2 to the number of the year. The algorithm is as follows:

If the number of the month is greater than 7 and is even then the number of days is equal to 31, and if it is odd then the number of days is 30. This is achieved with the part of the logical statement which reads as follows:

(P1>7)*(((P1 MOD 2)=Ø)*31+((P1 MOD 2)=1)*3Ø)

If the month has an even number the expression ((P1 MOD 2)=0) is true and a value of −1 will be returned; when multiplied by 31 this will give a value of −31. The expression ((P1 MOD 2)=1) will be false and so return a value of 0. If the number of the month is greater than 7 the expression (P1>7) will be true and hence return a value of −1. Thus the total value returned will be 31. This is easier to see if the values are placed under the expressions as follows :

```
(P1>7)*(((P1 MOD 2)=Ø)*31+((P1 MOD 2)=1)*3Ø)
  -1   *((     -1     )*31+(     Ø      )*3Ø)=31
```

If the month has a value greater than 7 and is an odd month the expression ((P1 MOD 2)=1) will evaluate as true and the expression ((P1 MOD 2)=0) will be false. Thus the total value for the complete expression above will be 30.

If the month is less than 7 the expression (P1>7) will be false and by returning a value of 0 for false makes the whole expression have a value of 0.

A second expression evaluates the part of the algorithm which deals with the remaining months. This reads as follows:

```
(P1<8)*(((P1 MOD 2)=Ø)*3Ø+((P1 MOD 2)=1)*31
 +(P1=2)*(((P2 MOD 4)=Ø)+((P2 MOD 4)< >Ø)*2))
```

This is built up from a similar expression to that for the months from August onwards. However, it also has to take account of the month of February and the fact that it has a different number of days in a leap year. Apart from February, note that even months have 30 days and odd months have 31 days in contrast to the other part of the year. The part involved with February is

(P1=2)*(((P2 MOD 4)=Ø)+((P2 MOD 4)< >Ø)*2)

If P1 corresponds to February, the expression (P1=2) will be true and thus return a value of −1. The rest of the expression involves deciding on whether the year is a leap year or not. A leap year occurs if the year is divisible by 4. If this is the case ((P2 MOD 4)=0) will be true, otherwise ((P2 MOD 4)< >0) will be true. This means that the expression will return a total value of 1 if the year is a leap year and 2 if it is not. The way this is built up is again easier to see if the values are placed under the expressions:

```
              (P1=2)*(((P2 MOD 4)=Ø)+((P2 MOD 4)< >Ø)*2)
leap       -1     *((       -1       )+(       Ø         )*2)=1
non-leap -1     *((        Ø        )+(       -1        )*2)=2
```

If the month is not February the expression (P1=2) is false and therefore returns 0 and so the whole expression evaluates to 0.

If the rest of the expression is broken down so that the part concerned with February is marked FEB, the decision as to the days of the months becomes:

```
(P1<8)*(((P1 MOD 2)=Ø)*3Ø+((P1 MOD 2)=1)*31+FEB)
```

Apart from allowing for February the logic is the same as in the other months of the year except that even months have 30 days and odd months 31.

Suppose that the month is February. A value of −30 will be returned by the expression ((P1 MOD 2)=0)*30 since February is an even month, and also FEB will return a value of 1 for a leap year, and 2 for a non leap year. The total expression thus gives 28 days for February in a normal year and 29 in a leap year, as follows:

```
            (P1<8)*(((P1 MOD 2)=Ø)*3Ø+((P1 MOD 2)=1)*31+FEB)
leap       -1    *((    -1        )*3Ø+(       Ø        )*31+ 1  )=29
non leap -1    *((    -1        )*3Ø+(       Ø        )*31+ 1  )=3Ø
```

Although this might seem complicated to begin with, it is extremely compact, and therefore the program runs faster. Try writing the same algorithm as a series of IF..THEN statements and see how many lines it involves.

Line 50 defines a function to cope with the case where the number returned for any part of the date or time is not a two digit number. It adds the leading "0" if required. It is used in line 390 to print the time and date. For example FNS$(HRS,MIN,SEC, ":") passes the values for the hour, the minutes, the seconds and the separator ":" to the variables P1,P2,P3 and P4$ of the function. Each numerical value is then converted to a string, using the string expression:

```
RIGHT$ ("Ø"+MID$(STR$(P),2),2)
```

A number is always printed with a leading and trailing space, so that an expression such as PRINT "PROFIT";PR; "percent" does not print "PROFIT20 percent" but a legible "PROFIT 20 percent". When a number or a numerical

variable is converted to a string the leading space is added to the string. Thus STR$(15) would give a string of length 3, the string " 15". The function MID$(STR$(P),2) returns the string starting with the second character. Thus if P has the value 5, STR$(P) has the value " 5" and the value returned by MID$(STR$(P),2) would be "5", whereas if it were 15 it would be "15". The leading zero is added in each case to give "05" and "015" respectively. By taking the two characters at the right of this string, the zero is lost if the number had two digits before being converted to a string, but is retained if there was only one. The string returned by the function is of the form "HH:MM:SS" in the example described.

Line 60 is a function which is used for manipulating the order of the day, month and year in the date to cope with the fact that different countries display the order of these values in their own way. The order is determined by three items in the DATA statement associated with that particular country. This makes it easy to add more countries or change the ones already displayed.

Each DATA statement (lines 3000−3090) ends with the three numbers 1,2 and 3. These are in a different order depending on the country. By using this number with the function defined in line 60, the month, date or year can be found as follows. The number is passed to the function as the variable P1 will be either 1, 2 or 3, and the value determines which of the statements (P1=1), (P1=2) or (P1=3) is true. Suppose P1 has the value 1. The statement (P1=1) will be TRUE and thus return a value of −1, whereas all the other logical statements will be false and so return a value of zero. The value returned by the function will be the value of the variable MN when the signs have been taken into account.

Line 70 initializes the variables used temporarily at various stages in the program. The variable OH is used to see if the hour has changed in line 300 and if it has, a sound is made. Variables T and SS are used to change the display around the heading as the time changes.

Line 80 reads the number of countries from the first DATA statement in line 3000. This makes it easy to alter the number of countries without having to alter the program. On the basis of the number read, the following arrays are dimensioned in line 90:

The arrays XP and YP hold the position to print the time and date for each country;

The array AS$ is used to hold the variable which denotes whether the time is ahead of or behind the local time;

The three arrays HR, ME and SD hold the values by which the time is different, and L the order in which the date is displayed for each country as used in the function FNP in line 60.

The loop 100 to 150 reads the values into the arrays from the data statements, using the function FNT to convert the time difference from a string into the appropriate numerical value.

Lines 160 to 210 set up the screen. The graphics character of ASCII code 143 is printed around the title, and is used later to give a visual indication of the seconds ticking by.

Line 220 is an important line in helping the program run without delays. A large number of variables are used and they are constantly changing. Initially BASIC stores them until it begins to run out of space, then it has to clear out the unwanted old values to make space to work. This is known as "garbage collection" and when it occurs with a large number of variables, it can cause the program to appear to have stopped working for a time. By executing the functions A=FRE(0) and A=FRE(A$), both the areas used for numerical and for string variables are cleaned up. Although the program will actually stop whilst this process is carried out, many small stops are invisible to the user because they are forced to happen. This line is the start of the main program loop and so happens before the screen is changed each time.

Lines 230 and 240 use the functions FNT and FND to determine the current hour and day, date and year from the internal clock in the PX-8. The hour is used in line 300 to sound the hour if there has been a change. If the hour is not set in line 230, the hour will be sounded when the program starts because H will have the value zero and will be seen as not equal to the variable OH in line 300.

Line 250 sets the variables T and SS which are used in lines 320 to 350 to define the position of the changing graphics character around the heading.

The loop in lines 260 to 400 prints out each time and date. The particular country is indexed by the loop counter X. Line 280 checks whether the STOP key or CTRL + C has been pressed. If it has an orderly exit is made in line 420.

In line 270 the day, month and year which were determined in line 240 are transferred into variables DY, MN and YR which are used for the subtraction and addition subroutines in lines 1000 to 1080 and 2000 to 2060 respectively.

Line 290 prints the current time and the current date on either side of the main heading. The date is printed in the format used in Great Britain since the program as it stands has the values set for GMT (Greenwich Mean Time). The order is found by using the function FNP of line 60, with the values set to 2, then 1 and then 3 to give day, month and then year. The leading zeros and separator are inserted using the function FNS$ of line 50.

Line 300 sounds the hour.

Line 310 determines the current time, using the function FNT of line 20 to split the string returned by TIME$ into hours, minutes and seconds.

Lines 320 to 350 form a routine to change the graphics characters around the heading.

Line 360 takes the location of the position to print the time and date from the arrays XP and YP using the value of X from the loop as index.

Line 370 determines the time offset for the particular country by indexing the arrays where they were stored when read from the DATA statements.

Line 380 checks to see if the time offset is positive or negative, and goes to the appropriate subroutine to determine the time in that country.

Line 390 formats the printout using the function FNS$ of line 80. The order of printing the date is determined from the array L.

When all the countries have been updated line 410 redirects execution to line 220 to repeat the process.

The addition and subtraction of time is carried out in the subroutines which precede the DATA statements.

Lines 1000 to 1080 form the addition subroutine. These routines are straightforward arithmetic additions. The seconds are converted to minutes and seconds, and the process is repeated for hours and minutes. The minutes offset is added as variable M1, and the hours offset as H1. When the hours have been determined a correction is made for the day and if necessary the year. The function FNN of line 40 is used in line 2040 to determine the number of days in the month.

Lines 2000 to 2060 form a similar subroutine to subtract the time.

Index